Diseño de Compiladores I – Cursada 2022
Trabajo Práctico Nro. 1

La entrega se hará en forma conjunta con el Trabajo Práctico Nro. 2. Fecha de Entrega: A definir

Objetivo

Desarrollar un Analizador Léxico que reconozca los siguientes tokens:

- Identificadores cuyos nombres pueden tener hasta 25 caracteres de longitud. El primer debe ser una letra, y el resto pueden ser letras, dígitos y "_". Los identificadores con longitud mayor serán truncados y esto se informará como Warning. Las letras utilizadas en los nombres de identificador pueden ser minúsculas y/o mayúsculas.
- Constantes correspondientes al tema particular asignado a cada grupo.
  Nota: Para aquellos tipos de datos que pueden llevar signo, la distinción del uso del símbolo '-' como operador aritmético o signo de una constante, se postergará hasta el trabajo práctico Nro. 2.
- Operadores aritméticos: "**+**", "**-**" , "*****", "***/***" agregando lo que corresponda al tema particular.
- Operador de asignación: "**=:**"
- Comparadores: "**>=**", "**<=**", "**>**", "**<**", "**=**", "=!"
- "**(**", "**)**","**{**", "**}**", "**,**" y "**;**"
- Cadenas de caracteres correspondientes al tema particular de cada grupo.
- Palabras reservadas (en minúsculas):
  if, then, else, end-if, out, fun, return, break
- y demás símbolos / tokens indicados en los temas particulares asignados al grupo.

El Analizador Léxico debe eliminar de la entrada (reconocer, pero no informar como tokens al Analizador Sintáctico), los siguientes elementos.

- Comentarios correspondientes al tema particular de cada grupo.
- Caracteres en blanco, tabulaciones y saltos de línea, que pueden aparecer en cualquier lugar de una sentencia.

Analizador Léxico. Especificaciones

a) El Analizador Léxico deberá leer un código fuente, identificando e informando:
   - Tokens detectados en el código fuente. Por ejemplo:
     Palabra reservada **if**
     (
     Identificador **var_x**
     +
     Constante entera **25**
     Palabra reservada **else**
     etc.
   - Errores léxicos detectados en el código fuente, indicando: nro. de línea y descripción del error. Por ejemplo:
     Línea 24: Constante entera fuera del rango permitido
   - Contenidos de la Tabla de Símbolos.

Se sugiere la implementación de un consumidor de tokens que invoque al Analizador Léxico solicitándole tokens. En el trabajo práctico 2, esta funcionalidad estará a cargo del Analizador Sintáctico.

b) El código fuente **debe ser leído desde un archivo**, cuyo nombre **debe poder ser elegido** por el usuario del compilador.

c) La numeración de las líneas de código debe comenzar en 1. Si se implementa una interfaz que permite mostrar o editar el código fuente, incluir alguna manera de identificar el número de cada línea del código.

d) Para la programación se podrá elegir el lenguaje. Para esta elección, tener en cuenta que el analizador léxico se integrará luego a un Parser (Analizador Sintáctico) generado utilizando una herramienta tipo Yacc. Por lo tanto, es necesario asegurarse la disponibilidad de dicha herramienta para el lenguaje elegido.

e) El Analizador Léxico deberá implementarse mediante una matriz de transición de estados y una matriz de acciones semánticas, de modo que cada cambio de estado y acción semántica asociada, sólo dependa del estado actual y el carácter leído.

f) Implementar una Tabla de Símbolos donde se almacenarán identificadores, constantes, y cadenas de caracteres. Es requisito para la aprobación del trabajo, que la tabla sea implementada con una estructura dinámica.

g) La aplicación deberá mostrar, además de tokens y errores léxicos, los contenidos de La Tabla de Símbolos.

Entrega

La entrega se pactará con el docente asignado al grupo. La asignación se encuentra publicada junto con los temas particulares.
El material entregado debe incluir:
- Ejecutable del compilador

- Código fuente completo del compilador
- Casos de prueba
- Informe

***Informe*:**

Debe incluir:
- **NRO. DE GRUPO** e Integrantes. Incluir **DIRECCIONES DE CORREO** para contacto.
- Temas particulares asignados (esta información deberá repetirse en los informes de los trabajos prácticos subsiguientes).
- Introducción.
- Decisiones de diseño e implementación.
- Diagrama de transición de estados.
- Matriz de transición de estados.
- Descripción del mecanismo empleado para implementar la matriz de transición de estados y la matriz de acciones semánticas.
- Lista de acciones semánticas asociadas a las transiciones del autómata del Analizador Léxico, con una breve descripción de cada una.
- Errores léxicos considerados.

Este informe deberá ser completado con las consignas indicadas en el Trabajo Práctico 2.

Casos de Prueba

Se debe incluir, como mínimo, ejemplos que contemplen las siguientes alternativas:
(Cuando sea posible, agregar un comentario indicando el comportamiento esperado del compilador)

- Constantes con el primer y último valor dentro del rango.
- Constantes con el primer y último valor fuera del rango.
- Para números de punto flotante: parte entera con y sin parte decimal, parte decimal con y sin parte entera, con y sin exponente, con exponente positivo y negativo.
- Identificadores de menos y más de 25 caracteres.
- Identificadores con letras, dígitos y "-".
- Intento de incluir en el nombre de un identificador un carácter que no sea letra, dígito o “-“.
- Palabras reservadas escritas en minúsculas y mayúsculas.
- Comentarios bien y mal escritos.
- Cadenas bien y mal escritas.

Temas particulares

Cada grupo de trabajo tendrá asignada una combinación de temas particulares.
La información de los temas asignados a cada grupo, estará disponible en el Aula Virtual de la materia.

1. **Enteros cortos (8 bits):** Constantes enteras con valores entre -2^7 y $2^7 - 1$.
   Se debe incorporar a la lista de palabras reservadas la palabra **i8**.
2. **Enteros cortos sin signo (8 bits):** Constantes enteras con valores entre 0 y $2^8 - 1$.
   Se debe incorporar a la lista de palabras reservadas la palabra u**i8**.
3. **Enteros (16 bits):** Constantes enteras con valores entre -2^{15} y $2^{15} - 1$.
   Se debe incorporar a la lista de palabras reservadas la palabra **i16**.
4. **Enteros sin signo (16 bits):** Constantes con valores entre 0 y $2^{16} - 1$.
   Se debe incorporar a la lista de palabras reservadas la palabra ui**16**.
5. **Enteros largos (32 bits):** Constantes enteras con valores entre -2^{31} y $2^{31} - 1$.
   Se debe incorporar a la lista de palabras reservadas la palabra **i 32**.
6. **Enteros largos sin signo (32 bits):** Constantes enteras con valores entre 0 y $2^{32} - 1$.
   Se debe incorporar a la lista de palabras reservadas la palabra ui**32**.
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 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
   Se debe incorporar a la lista de palabras reservadas la palabra f**32**.
8. **Dobles:** Números reales con signo y parte exponencial. El exponente comienza con la letra D (mayúscula) y el signo es opcional. La ausencia de signo, implica exponente positivo. La parte exponencial puede estar ausente.
   Puede estar ausente la parte entera, o la parte decimal, pero no ambas. El '.' es obligatorio.
   DjDmplos válidos: 1. .6 -1.2 3.D–5 2.D+34 2.5D-1 13. 0. 1.2D10
   ConsidDrar Dl rango 2.2250738585072014D-308 < x < 1.7976931348623157D+308 ∪
   -1.7976931348623157D+308 < x < -2.2250738585072014D-308 ∪ 0.0
   Se debe incorporar a la lista de palabras reservadas la palabra f**64**.
9. Incorporar a la lista de palabras reservadas la palabra **discard**.
10. Incorporar a la lista de palabras reservadas la palabra **when**.
11. Incorporar a la lista de palabras reservadas la palabra **while y continue**.
12. Incorporar a la lista de palabras reservadas las palabras **while**.
13. Incorporar a la lista de palabras reservadas las palabras **do, until y continue**.
14. Incorporar a la lista de palabras reservadas las palabras **do** y **until**.
15. Incorporar a la lista de palabras reservadas la palabra **for y continue**.
16. Incorporar a la lista de palabras reservadas las palabras **for**.
17. A definir en Trabajos Prácticos 2/3.
18. A definir en Trabajos Prácticos 2/3.
19. A definir en Trabajos Prácticos 2/3.
20. A definir en Trabajos Prácticos 2/3.
21. A definir en Trabajos Prácticos 2/3.
22. A definir en Trabajos Prácticos 2/3.
23. A definir en Trabajos Prácticos 2/3.
24. **Comentarios de 1 línea:** Comentarios que comiencen con "**<<**" y terminen con el fin de línea.
25. **Comentarios multilínea:** Comentarios que comiencen con "**<<**" y terminen con "**>>**" (estos comentarios pueden ocupar más de una línea).
26. **Cadenas de 1 línea:** Cadenas de caracteres que comiencen y terminen con " **'** " (estas cadenas no pueden ocupar más de una línea).
    Ejemplo: ' ¡Hola mundo ! '
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! **'**

Diseño de Compiladores I - 2022
Trabajo Práctico Nº 2
Fecha de entrega: 06/10/2022

ATENCIÓN

Antes de comenzar el desarrollo de esta etapa, revisar las asignaciones de temas particulares que se publicaron con el Trabajo Práctico 1. Para algunos grupos, se efectuaron modificaciones que afectan la lista de palabras reservadas a considerar, y los temas a desarrollar en la segunda etapa.

Para indicar cuáles son esas modificaciones, se puso entre paréntesis el tema de la asignación original, y con un * el tema que lo debe reemplazar.

Por ejemplo:

Para la asignación: 3 7 9 **(13) 14*** 18 20 **21* (23)** 24 27 se debe reemplazar el tema 13 por el 14, y el 23 por el 21

OBJETIVO

Construir un Parser (Analizador Sintáctico) que invoque al Analizador Léxico creado en el Trabajo Práctico Nº 1, y que reconozca un lenguaje con las siguientes características:

SINTAXIS GENERAL:

Programa:

- Programa constituido por un nombre de programa, seguido de un conjunto de sentencias, que pueden ser declarativas o ejecutables.
  Las sentencias declarativas pueden aparecer en cualquier lugar del código fuente, exceptuando los bloques de las sentencias de control.
- El conjunto de sentencias estará delimitado por llaves '**{**' y '**}**'.
- Cada sentencia debe terminar con " **;**".

Sentencias declarativas:

- Sentencias de declaración de datos para los tipos de datos correspondientes a cada grupo según la consigna del Trabajo Práctico 1, con la siguiente sintaxis:

  <tipo> <lista_de_variables>;

  Donde <tipo> puede ser (Según tipos correspondientes a cada grupo): **i8, i16, i32, ui8, ui16, ui32, f32, f64**
  Las variables de la lista se separan con coma ("**,**")
- Incluir declaración de funciones, con la siguiente sintaxis:

  fun ID (<lista_de_parametros>) : <tipo>

  {

  <cuerpo_ de_la_funcion>

  }

  Donde:
  - <lista_de_parametros> será una lista de parámetros formales separados por ",", con la siguiente estructura para cada parámetro:

    <tipo> ID

    El número máximo de parámetros permitidos es 2, y puede no haber parámetros. **Este chequeo debe efectuarse durante el Análisis Sintáctico**
  - <cuerpo_de_la_funcion> es un conjunto de sentencias declarativas (incluyendo declaración de otras funciones) y/o ejecutables, incluyendo sentencias de retorno con la siguiente estructura:

    return (<retorno>);

    <retorno> podrá ser cualquier expresión aritmética

// **LÉXICO**: Incorporar el reconocimiento del símbolo '**:**'

Ejemplos válidos:

```
fun f1 (i16 y) : i16
{
        i16 x;
        x =: y;
        …
        return (x);
}
```

```
fun f2 (i32 x, i16 y) : i16
{
         i16 x;
        x =: y;
        …
        return (x);
}
```

```
fun f3 () : f32
{
        f32 x;
        x =: 1.2;
        …
        if (x > 0.0) then
                return (x);
        else
                return (x + 1.2);
        end_if;
}
```

Sentencias ejecutables:

- Asignaciones donde el lado izquierdo puede ser un identificador, y el lado derecho una expresión aritmética. Los operandos de las expresiones aritméticas pueden ser variables, constantes, u otras expresiones aritméticas.
  No se deben permitir anidamientos de expresiones con paréntesis.
- Considerar como posible operando de una expresión, la invocación a una función, con el siguiente formato:

  ID(<lista_de_parametros_reales>)

  Donde cada parámetro real puede ser un identificador o una constante.
- Cláusula de selección (**if**). Cada rama de la selección será un bloque de sentencias. La estructura de la selección será, entonces:

  if (<condicion>) **then** <bloque_de_sent_ejecutables> **else** <bloque_de_sent_ejecutables> **end_if;**

  El bloque para el **else** puede estar ausente.
  La condición será una comparación entre expresiones aritméticas, variables o constantes, y debe escribirse entre **"(" ")"**.
  El bloque de sentencias ejecutables puede estar constituido por una sola sentencia, o un conjunto de sentencias ejecutables delimitadas por llaves.

// **LÉXICO**: La palabra reservada **end_if** debe utilizar guión bajo (en el tp1 se indicó end-if)

- Sentencia de salida de mensajes por pantalla. El formato será

  out(<cadena>);

  Ejemplos:

```
out('Hola mundo');                //Tema 26
out('Hola /                       //Tema 27
        mundo');
```

TEMAS PARTICULARES

Nota: La semántica de cada tema particular, se explicará y resolverá en las etapas 3 y 4 del trabajo práctico

Temas 9 y 10

- Tema 9: **discard**
  Incorporar a la gramática, la o las reglas que permitan reconocer invocaciones a funciones como si fueran procedimientos. Es decir:

  ID(<lista_de_parametros>);

  Si el compilador detecta una invocación como la indicada, debe informar Error Sintáctico.
  Sin embargo, si la invocación es precedida por la palabra reservada **discard**, la sentencia no generará error, y la invocación podrá considerarse como válida.
  Ejemplo:

```
discard f1(2);
```

- Tema 10: **Cláusulas de compilación**
  Incorporar, a las sentencias declarativas, la posibilidad de definir constantes, con la siguiente estructura:

  const <lista_de_constantes>;

  donde la lista incluirá una lista de constantes separadas por ','. Cada constante se definirá con la siguiente sintaxis:

  <nombre_de_constante> =: <valor_de_constante>

Ejemplo:

```
const c1 =:10, x=:5;
```

Incorporar, a la sentencias ejecutables, la siguiente estructura:

when (<condición>) **then** <bloque_de_sentencias>

donde:

<condición> tendrá la misma estructura que las condiciones de las sentencias de control
<bloque_de_sentencias> podrá contener tanto sentencias ejecutables como declarativas.

// **LÉXICO**: Incorporar a la lista de palabras reservadas, la palabra **const**.

Temas 11 a 16: Sentencias de Control

- ***While con continue*** (tema 11 en TP1)

  while (<condicion>) <bloque_de_sentencias_ejecutables> **;**

  <condición> tendrá la misma definición que la condición de las sentencias de selección**.**

  <bloque_de_sentencias_ejecutables> podrá contener una sentencia, o un grupo de sentencias ejecutables delimitadas por llaves. Además, podrá contener sentencias **break** y sentencias **continue**. Ambas se escriben con la palabra reservada seguida de ';'

- ***While con expresión*** (tema 12 en TP1)

  while (<condicion>) **:** (<asignación>) <bloque_de_sentencias_ejecutables> **;**

  <condición> tendrá la misma definición que la condición de las sentencias de selección.

  <bloque_de_sentencias_ejecutables> podrá contener una sentencia, o un grupo de sentencias ejecutables delimitadas por llaves. Además, podrá contener sentencias break (se escribe con la palabra reservada seguida de ';')

  Ejemplo:

```
while (i < 10) : (i =: i + 1) {
out('ciclo');
};
```

- ***Do until con continue*** (tema 13 en TP1)

  do <bloque_de_sentencias_ejecutables> **until** (<condicion>);

  <condición> tendrá la misma definición que la condición de las sentencias de selección**.**

  <bloque_de_sentencias_ejecutables> podrá contener una sentencia, o un grupo de sentencias ejecutables delimitadas por llaves. Además, podrá contener sentencias **break** y sentencias **continue**. Ambas se escriben con la palabra reservada seguida de ';'

- ***Do until con expresión*** (tema 14 en TP1)

  do <bloque_de_sentencias_ejecutables> **until** (<condicion>) **:** (<asignación>) **;**

  <condición> tendrá la misma definición que la condición de las sentencias de selección**.**

  <bloque_de_sentencias_ejecutables> podrá contener una sentencia, o un grupo de sentencias ejecutables delimitadas por llaves. Además, podrá contener sentencias break (se escribe con la palabra reservada seguida de ';')

  Ejemplo**:**

```
do {
        out('ciclo');
} until (i > 10) : (i =: i + 1);
```

- ***for con continue*** (tema 15 en TP1)

  for (i =: n; <condición_for>; +/- j) < bloque_de_sentencias_ejecutables > **;**

  i debe ser una variable de tipo entero (1-2-3-4-5-6).
  n y j serán constantes de tipo entero (1-2-3-4-5-6).
  <condición_for> será una comparación de i con m. Por ejemplo: i < m
  Donde m puede ser una variable, constante o expresión aritmética de tipo entero (1-2-3-4-5-6).
  El signo '+' o '-' al final del encabezado es obligatorio..

  <bloque_de_sentencias_ejecutables> podrá contener una sentencia, o un grupo de sentencias ejecutables delimitadas por llaves. Además, podrá contener sentencias **break** y sentencias **continue**. Ambas se escriben con la palabra reservada seguida de ';'

Nota: Las restricciones de tipo serán chequeadas en la etapa 3 del trabajo práctico.

- ***for*** (tema 16 en TP1)

  for (i =: n; <condición_for>; +/- j) < bloque_de_sentencias_ejecutables > **;**

  i debe ser una variable de tipo entero (1-2-3-4-5-6).
  n y j serán constantes de tipo entero (1-2-3-4-5-6).
  <condición_for> será una comparación de i con m. Por ejemplo: i < m
  Donde m puede ser una variable, constante o expresión aritmética de tipo entero (1-2-3-4-5-6).
  El signo '+' o '-' al final del encabezado es opcional. La ausencia de signo implica incremento.

  <bloque_de_sentencias_ejecutables> podrá contener una sentencia, o un grupo de sentencias ejecutables delimitadas por llaves. Además, podrá contener sentencias **break, que s**e escriben con la palabra reservada seguida de ';'

  Nota: Las restricciones de tipo serán chequeadas en la etapa 3 del trabajo práctico.

Temas 17 y 18: Sentencias de Control con etiquetado

- ***Break con etiquetado*** (tema 17 en TP1)

  Incorporar, a la sentencia de control asignada (temas 11 al 16), la posibilidad de antecederla con una etiqueta.

  Incorporar, a la sentencia **break**, la posibilidad de incluir una etiqueta.

  Por ejemplo:

```
outer: while (i < 10) {
        while (j < 5) {
            break :outer;
        };
    };
```

- ***Continue con etiquetado*** (tema 18 en TP1)

  Incorporar, a la sentencia de control asignada (temas 11 al 16), la posibilidad de antecederla con una etiqueta.

  Incorporar, a la sentencia **continue**, la posibilidad de incluir una etiqueta.

  Por ejemplo:

```
outer: while (i < 10) : (i =: i + 1) {
        while (j < 5) : (j := j + 1) {
            continue :outer;
        };
    };
```

Temas 19 y 20

- Tema 19: **Diferimiento**
  Incorporar, a las sentencias ejecutables, la posibilidad de antecederlas por la palabra reservada **defer**. Esta funcionalidad podrá aplicarse a estructuras sintácticas de una sola sentencia, como asignación o emisión de mensajes, pero también a estructuras de bloque como sentencias de selección o iteración.
  Ejemplos:

```
defer  out(a);
```

```
defer while(i<10)
{
     out ('ciclo');
     i=:i+1;
}
```

  // **LÉXICO**: Incorporar a la lista de palabras reservadas, la palabra **defer**.

- Tema 20: **Sentencia de control como expresión**
  Incorporar, a la sentencia de control asignada en los temas 11 a 16, la posibilidad de devolver un valor utilizando la sentencia break, o un valor por defecto en la finalización normal de la sentencia.

Ejemplo para while: (11)

```
a =: while (i < end){
     if (i = number) {
        break 1;
        }
     } else 3;
```

Ejemplo para do until: (13)

```
a =: do {
     if (i = number) {
        break 1;
        }
     }until (i> end) else 3 ;
```

Ejemplo para for: (15/16)

```
a =: for (i =: 0 ;i < end ; +1){
     if (i = number) {
        break 1;
        }
     } else 3;
```

Ejemplo para while: (12)

```
a =: while (i < end) : (i=:i+1)
{
     if (i = number) {
        break 1;
        }
     } else 3;
```

Ejemplo para do until: (14)

```
a =: do {
     if (i = number) {
        break 1;
        }
     } until (i> end) : (i=:i+1) else 3 ;
```

Temas 21 a 23: Conversiones

- Tema 21: **Conversiones Explícitas**: . Se debe incorporar en todo lugar donde pueda aparecer una expresión, la posibilidad de utilizar la siguiente sintaxis:

  tof32(<expresión>) // para grupos que tienen asignado el tema 7

  tof64(<expresión>) // para grupos que tienen asignado el tema 8

// **LÉXICO**: Incorporar a la lista de palabras reservadas, la palabra **tof32** o **tof64** según corresponda.

- Tema 22: **Conversiones Implícitas**: Se explicará y resolverá en trabajos prácticos 3 y 4.
- Tema 23: **Sin conversiones:** Se explicará y resolverá en trabajos prácticos 3 y 4.

SALIDA DEL COMPILADOR

El programa deberá leer un código fuente escrito en el lenguaje descripto, y deberá generar como salida:

- Tokens detectados por el Analizador Léxico
- Estructuras sintácticas detectadas en el código fuente. Por ejemplo:
  Asignación
  Sentencia **while**
  Sentencia **if**
  etc.
  (Indicando nro. de línea para cada estructura)
- Errores léxicos y sintácticos presentes en el código fuente, indicando: nro. de línea y descripción del error. Por ejemplo:
  Línea 24: Constante de tipo **i8** fuera del rango permitido.
  Línea 43: Falta paréntesis de cierre para la condición de la sentencia **if**.
- Contenidos de la Tabla de símbolos

CONSIDERACIONES GENERALES

a) Utilizar **YACC** u otra herramienta similar para construir el Parser.

b) Adaptar el Analizador Léxico del Trabajo Práctico 1 para convertirlo en el método o función ***int yylex()*** (o el nombre que el Parser generado requiera). Tener en cuenta que el léxico deberá devolver al parser, en cada invocación, un token. Para los identificadores, constantes y cadenas, deberá devolver además, la referencia a la entrada de la Tabla de Símbolos donde se ha registrado dicho símbolo, utilizando **yylval** para hacerlo.

c) Para aquellos tipos de datos que permitan valores negativos (**i8, i16, i32, f32, f64**) durante el Análisis Sintáctico se deberán detectar **constantes negativas**, modificando la tabla de símbolos según corresponda. Será necesario volver a controlar el rango de las constantes, ya que un valor aceptado para una constante por el Analizador Léxico, que desconoce su signo, podría estar fuera de rango si la constante es positiva.

- Ejemplo: Las constantes de tipo **i16** pueden tomar valores desde –32768 a 32767. El Léxico aceptará la constante 32768 como válida, pero si se trata de una constante positiva, estará fuera de rango.

d) Cuando se detecte un error, la compilación debe continuar.
e) Conflictos: Eliminar **TODOS LOS CONFLICTOS SHIFT-REDUCE Y REDUCE-REDUCE** que se presenten al generar el Parser.

FORMA DE ENTREGA

Se deberá presentar:
a) Código fuente completo y ejecutable, **incluyendo librerías del lenguaje** si fuera necesario para la ejecución
b) Informe
   - Contenidos indicados en el enunciado del Trabajo Práctico 1
   - Descripción del proceso de desarrollo del Analizador Sintáctico: problemas surgidos (y soluciones adoptadas) en el proceso de construcción de la gramática, manejo de errores, solución de conflictos shift-reduce y reduce-reduce, etc.
   - Lista de no terminales usados en la gramática con una breve descripción para cada uno.
   - Lista de errores léxicos y sintácticos considerados por el compilador.
   - Conclusiones.
c) Casos de prueba que contemplen **todas** las estructuras válidas del lenguaje. Incluir casos con errores sintácticos.

Diseño de Compiladores I – Cursada 2022
Trabajo Práctico Nro. 3

La entrega se hará en forma conjunta con el trabajo práctico Nro. 4 (17/11/2022)
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	4	3
2	5	6
7	9	12
8	15	14
10	18	17
11	19	20
13	25	22
16		23
21		24

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

- Asignaciones, Selecciones, Sentencias de control asignadas al grupo, invocaciones a funciones y Sentencias **out.**

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 función, 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 redeclaradas (según reglas de alcance del lenguaje).
  - Toda otra situación que no cumpla con las siguientes reglas:

  Reglas de alcance:
  - Cada variable o función será visible dentro del ámbito en el que fue declarada/o y por los ámbitos anidados contenidos en el ámbito de la declaración.
  - Cada variable, o función 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 etiquetas con el mismo nombre, siempre que sean declaradas en diferentes ámbitos.
  - No se permiten variables, funciones y etiquetas con el mismo nombre dentro de un mismo ámbito.
- Chequeo de compatibilidad de tipos:
  - Temas 21 y 22: Sólo se permitirán operaciones con operandos de distinto tipo, si se efectúa la conversión que corresponda (implícita o explícita según asignación al grupo).

- Tema 23: 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 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, deben coincidir con el número de los parámetros declarados para la función.
- Para el tipo de los parámetros, 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 las variables y funciones 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 con Nombre

```
pp // nombre del programa
{
      ...
      // Declaraciones en el ámbito global
      i32 a;    // la variable a se llamará a.pp
                // pp identifica el ámbito global
      ...
      fun aa(i32 x) : i16        // Ámbito aa
      {
              i16 a;  // la variable a se llamará a.pp.aa
              ...
              fun aaa(i16 w) : i32                    // Ámbito aaa
            {
                    ...
                    i32 x;   // la variable x se llamará
                             // x.pp.aa.aaa
                    ...
            }                                          // Fin Ámbito aaa
      }                                                // Fin Ámbito aa
      ...
      fun bb(i16 z): i32                               // Ámbito bb
      {
             ...
              i16 a;  // la variable a se llamará a.pp.bb
              ...
      }                                                // Fin Ámbito bb
      ...
      fun cc(i16 y) : i16                              // Ámbito cc
      {
              i16 a;  // la variable a se llamará a.pp.cc
              ...
      }                                                // Fin Ámbito cc
      ...
}                                                      // Fin Ámbito pp
```

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:
  - Tipo devuelto
  - Para los parámetros se deberá registrar el tipo
- 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 de los parámetros en la invocación, así como otros chequeos asociados al uso de dicha función en el contexto en que se está invocando.
- El pasaje de parámetros será por copia-valor. Se deberá generar el código para el pasaje de parámetros correspondiente a cada invocación.

Tema 9: discard

Ante una sentencia:

discard ID(<lista_de_parametros_reales>);

La invocación se efectuará normalmente, pero el valor de retorno de la función será descartado.

Ejemplo:

```
discard f1(2);      // el valor de retorno de la función f1 será descartado.
```

Tema 10: Cláusulas de compilación

Ante una sentencia de declaración de constantes:

const <lista_de_constantes>;

el compilador deberá registrar en la Tabla de Símbolos, la información de las constantes declaradas, registrando que se trata de constantes, el valor y el tipo que corresponda.

Ante una sentencia:

when (<condición>) **then** <bloque_de_sentencias>

Las sentencias del bloque sólo deberán ser consideradas en la compilación, cuando la condición pueda evaluarse como verdadera en tiempo de compilación.

Tema 11: While con continue

while (<condicion>) <bloque_de_sentencias_ejecutables> **;**

El bloque se ejecutará mientras la condición sea verdadera.
Si se detecta una sentencia **break**, se debe salir de la iteración en la que se encuentra.
Si se detecta una sentencia **continue**, se debe forzar el salto al próximo ciclo de la iteración.

Tema 12: While con expresión

while (<condicion>) **:** (<asignación>) <bloque_de_sentencias_ejecutables> **;**

El bloque se ejecutará mientras la condición sea verdadera.
Si se detecta una sentencia **break**, se debe salir de la iteración en la que se encuentra.

La asignación del encabezado, en caso de estar presente, se deberá ejecutar al final de cada ciclo.

Tema 13: Do until con continue

do <bloque_de_sentencias_ejecutables> **until** (<condicion>);

El bloque se ejecutará hasta que la condición sea verdadera
Si se detecta una sentencia **break**, se debe salir de la iteración en la que se encuentra.
Si se detecta una sentencia **continue**, se debe forzar el salto al próximo ciclo de la iteración.

Tema 14: Do until con expresión

do <bloque_de_sentencias_ejecutables> **until** (<condicion>) **:** (<asignación>) **;**

El bloque se ejecutará hasta que la condición sea verdadera
Si se detecta una sentencia **break**, se debe salir de la iteración en la que se encuentra.
La asignación del final de la sentencia, en caso de estar presente, se deberá ejecutar al final de cada ciclo.

Tema 15: for con continue

for (i =: n; <condición_for>; +/- j) < bloque_de_sentencias_ejecutables > **;**

- El bloque se ejecutará hasta que se cumpla la condición.
- En cada ciclo, el incremento (+) o decremento (-) de la variable de control será igual a j.
- Se deberán efectuar los siguientes chequeos de tipo:
  - i debe ser una variable de tipo entero (1-2-3-4-5-6).
  - n y j deben ser constantes de tipo entero (1-2-3-4-5-6).
  - <condición_for> debe ser una comparación de i con m. Por ejemplo: i < m
    Donde m puede ser una variable, constante o expresión aritmética de tipo entero (1-2-3-4-5-6).

  Nota:
  - Para Tercetos y Árbol Sintáctico, los chequeos de tipos se debe efectuar durante la generación de código intermedio
  - Para Polaca Inversa, los chequeos de tipos se deben efectuar en la última etapa (TP 4)
- Si se detecta una sentencia **break**, se debe salir de la iteración en la que se encuentra.
- Si se detecta una sentencia **continue**, se debe forzar el salto al próximo ciclo de la iteración, efectuando la actualización de la variable de control previamente a dicho ciclo.

Tema 16: for

for (i =: n; <condición_for>; +/- j) < bloque_de_sentencias_ejecutables > **;**

- El bloque se ejecutará hasta que se cumpla la condición.
- En cada ciclo, el incremento (+/ ausencia de signo) o decremento (-) de la variable de control será igual a j.
- Se deberán efectuar los siguientes chequeos de tipo:
  - i debe ser una variable de tipo entero (1-2-3-4-5-6).
  - n y j deben ser constantes de tipo entero (1-2-3-4-5-6).
  - <condición_for> debe ser una comparación de i con m. Por ejemplo: i < m
    Donde m puede ser una variable, constante o expresión aritmética de tipo entero (1-2-3-4-5-6).

  Nota:
  - Para Tercetos y Árbol Sintáctico, los chequeos de tipos se debe efectuar durante la generación de código intermedio
  - Para Polaca Inversa, los chequeos de tipos se deben efectuar en la última etapa (TP 4)
- Si se detecta una sentencia **break**, se debe salir de la iteración en la que se encuentra.

Tema 17: Break con etiquetado

Al detectar una etiqueta precediendo a la sentencia de control asignada (temas 11 al 16), y un break con una etiqueta, se deberá proceder de la siguiente forma:

- Se debe salir de la iteración etiquetada con la etiqueta indicada en el break.
- Se deberá chequear que exista tal etiqueta, y que la sentencia break se encuentre en la sentencia de control que lleva esa etiqueta, o en una sentencia anidada en la misma.

Por ejemplo:

```
outer: while (i < 10) {
        while (j < 5) {
            break :outer;  //El break provocará salir de la iteración etiquetada con outer
        };
    };
```

Tema 18: Continue con etiquetado

Al detectar una etiqueta precediendo a la sentencia de control asignada (temas 11 al 16), y un continue con una etiqueta, se deberá proceder de la siguiente forma:

- Se debe saltar al próximo ciclo de la iteración etiquetada con la etiqueta indicada en el continue.
- Se deberá chequear que exista tal etiqueta, y que la sentencia continue se encuentre en la sentencia de control que lleva esa etiqueta, o en una sentencia anidada en la misma.

Por ejemplo:

```
outer: while (i < 10) : (i =: i + 1) {
  while (j < 5) : (j := j + 1) {
      continue :outer; // provocará el salto al próx. ciclo del while etiquetado con outer.
   };
};
```

Tema 19: Diferimiento

Al detectarse la presencia de la palabra reservada **defer** al comienzo de una sentencia o bloque de selección o iteración, la ejecución de dicha sentencia o bloque deberá diferirse al final del ámbito donde se encuentra la sentencia diferida.

Tema 20: Sentencia de control como expresión

Incorporar, a la sentencia de control asignada en los temas 11 a 16, la posibilidad de devolver un valor utilizando la sentencia break, o un valor por defecto en la finalización normal de la sentencia.

Al detectarse la utilización de una sentencia de control (temas 11 a 16) como expresión, se deberá proceder de la siguiente manera:

- Si se produce la ejecución del break, el valor retornado por la sentencia de control, que se utilizará como expresión, será el indicado en la sentencia break.
- Si no se produce la ejecución del break, el valor retornado por la sentencia de control, que se utilizará como expresión, será el indicado en el else al final de la sentencia.

Ejemplos:

Ejemplo para while: (11)	Ejemplo para do until: (13)	Ejemplo para for: (15/16)

```
a =: while (i < end){
     if (i = number) {
        break 1;
        }
     } else 3;
```

```
a =: do {
     if (i = number) {
        break 1;
        }
    }until (i> end) else 3 ;
```

```
a =: for (i =: 0 ;i < end ; +1){
     if (i = number) {
        break 1;
        }
     } else 3;
```

Ejemplo para while: (12)

```
a =: while (i < end) : (i=:i+1)
{
     if (i = number) {
        break 1;
        }
     } else 3;
```

Ejemplo para do until: (14)

```
a =: do {
     if (i = number) {
        break 1;
        }
    } until (i> end) : (i=:i+1) else 3 ;
```

Para todos los ejemplos, si se ejecuta el break, el valor asignado a la variable a será 1. En caso contrario, será 3.

Dado que el valor retornado participa en una expresión aritmética, se deberá chequear que el tipo del valor devuelto sea compatible con el resto de la expresión en la que participa.

Tema 21: Conversiones Explícitas

- Todos los grupos que tienen asignado el tema 21, deben reconocer una conversión explícita, indicada mediante la palabra reservada para tal fin.
- Por ejemplo, un grupo que tiene asignada la conversión **tof64**, debe considerar que cuando se indique la conversión **tof64**(expresión), el compilador debe efectuar una conversión del tipo del argumento al tipo indicado por la palabra reservada. (en este caso **f64**). Dado que el otro tipo asignado al grupo es entero (1-2-3-4-5-6), el argumento de una conversión, debe ser de dicho tipo. Entonces, sólo se podrán efectuar operaciones entre dos operandos de distinto tipo, si se convierte el operando de tipo entero (1-2-3-4-5-6) al tipo de punto flotante asignado, mediante la conversión explícita que corresponda. En otro caso, se debe informar error.
- En el caso de asignaciones, si el lado izquierdo es del tipo entero (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 22: Conversiones Implícitas

- Todos los grupos que tienen asignado el tema 22, deben incorporar las conversiones en forma implícita, cuando se intente operar entre operandos de diferentes tipos. En todos los casos, la conversión a incorporar será del tipo entero (1-2-3-4-5-6) al otro tipo asignado al grupo. Por ejemplo, el compilador incorporará una conversión del tipo **i16** a **f32**, si se intenta efectuar una operación entre dos operandos de dichos tipos.
- En el caso de asignaciones, si el lado izquierdo es del tipo entero (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.
- ***Nota***: La incorporación de las 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 23: Sin conversiones

Los grupos que tienen asignado el tema 23, deben prohibir cualquier operación (expresión, asignación, comparación, etc.) entre operandos de tipos diferentes.

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:

Tercetos:

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

Polaca Inversa:

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

Árbol Sintáctico:

Por ejemplo:

Raíz

Hijo izquierdo

Hijo izquierdo

Hijo derecho

Hijo derecho

Hijo izquierdo

…

Hijo derecho

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.

Diseño de Compiladores I
Trabajo Práctico Nº 4
Fecha de Entrega: 17/11/2022

Objetivo

Generar código Assembler, a partir del código intermedio generado en el Trabajo Práctico Nº 3.
El mecanismo para generar código será el de variables auxiliares.

El código Assembler generado debe poder ser ensamblado y ejecutado sin errores.

Los grupos podrán optar entre la generación de:

- Assembler para Pentium de 32 bits. El ensamblador a utilizar se puede descargar desde la sección Herramientas del Aula Virtual.
    - Para las operaciones entre datos de tipo entero se deberá generar código que utilice los registros del procesador (EAX, EBX, ECX Y EDX o AX, BX, CX y DX.
    - Para las operaciones entre datos de punto flotante se deberá utilizar el co-procesador 80X87.
- WebAssembly. https://webassembly.org/

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.

c) Overflow en sumas de datos de punto flotante
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.

d) Overflow en productos 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.

e) Overflow en productos de datos de punto flotante:
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.

f) Resultados negativos en restas de enteros sin signo:
El código Assembler deberá controlar el resultado de la operación indicada. Este control se aplicará a operaciones entre enteros sin signo. En caso que una resta entre datos de este tipo arroje un resultado negativo, deberá emitir un mensaje de error y terminar.

g) Recursión en invocaciones de funciones
El código Assembler deberá controlar que una función no pueda invocarse a sí misma.

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.

GRUPO	CHEQUEOS
1	a b g
2	a c h
3	a d g
4	a e h
5	b e g
6	c f h
7	b c g
8	d e h
9	a c g
10	a d h
11	a e g
12	a b h

GRUPO	CHEQUEOS
13	a f g
14	b e h
15	c f g
16	c f h
17	e f g
18	a f h
19	d e g
20	b c h
21	a e g
22	a c h
23	d f g
24	a d h

Conversiones explícitas (Tema 21):

- Los grupos cuyos lenguajes incluyen conversiones explícitas, deberán traducir el código de las conversiones a código Assembler.
-

Conversiones implícitas (Tema 22):

Para los grupos que deben considerar conversiones implícitas, el código Assembler deberá efectuar dichas conversiones cuando los tipos de los datos involucrados en una expresión lo requieran, y siempre que el lenguaje lo permita.

- Los grupos que debieron generar las conversiones en la representación intermedia, deberán traducir el código de las conversiones a código Assembler.
- Los grupos que deben considerar el chequeo de tipos y/o las conversiones en esta etapa, deberán agregar código para efectuar las conversiones cuando las mismas sean requeridas, siempre que el lenguaje lo permita. Si hubiera incompatibilidad de tipos, esto deberá ser informado por el compilador como un error.

Salidas del compilador

1) Los errores generados en tiempo de compilación (Errores Léxicos, Sintácticos y Semánticos) se deberán mostrar todos juntos, indicando la descripción de cada error y la línea en la que fue detectado.
2) Representación intermedia, según indicaciones del Trabajo Práctico 3.
3) Contenidos de la Tabla de Símbolos.
4) Archivo conteniendo el código Assembler.

Nota: No se deberán mostrar los tokens ni las estructuras sintácticas que se pidieron como salida de los Trabajos Prácticos 1 y 2, a menos que exista una reentrega pendiente que así lo requiera.

Forma de entrega

Se deberá entregar:

- **Código fuente completo y ejecutable, incluyendo librerías del lenguaje y todo otro componente que fuera necesario para la ejecución.**

- **Casos de Prueba:**
  - Casos válidos para cada estructura sintáctica solicitada
  - Casos semánticamente válidos
  - Casos con errores por Tipos incompatibles, y/o que requieran conversiones para los temas 21 y 22.
  - Casos con errores de redeclaración o no declaración de variables / funciones / etiquetas
  - Casos que violen las restriciones de alcance del lenguaje

- Casos de prueba para el tema particular 9 o 10, según corresponda
- Casos de prueba para el tipo de sentencia de control asignada en los temas 11 a 16. Incluir en las pruebas, el anidamiento de este tipo de sentencias, así como con sentencias de Selección.
- Casos de prueba para el tema particular 17 o 18, según corresponda. Incluir en las pruebas, el anidamiento de sentencias de control conteniendo el break o continue etiquetado, según corresponda.
- Casos de prueba para el tema particular 19 o 20, según corresponda.
- Casos que generen errores en tiempo de ejecución (para cada control asignado al grupo)

- **Informe conteniendo:**
  - Introducción
  - Descripción de la Generación de Código Intermedio, indicando:
    - estructura utilizada para el almacenamiento del código intermedio,
    - uso de notación posicional de Yacc ($$, $n), indicando en qué casos se usó, y con qué fin,
    - descripción tipo pseudocódigo del algoritmo usado para la generación de las bifurcaciones en sentencias de control,
    - nuevos errores considerados,
    - y todo otro aspecto que se considere relevante.
  - Descripción del proceso de Generación de Código Assembler, indicando:
    - Mecanismo utilizado para la generación del código Assembler
    - Mecanismo utilizado para efectuar cada una de las operaciones aritméticas
    - Mecanismo utilizado para la generación de las etiquetas.
    - Todo otro aspecto que se considere relevante
  - Modificaciones a las etapas anteriores, si hubieran existido.
  - Para los **temas 9 y 10**, 17 y 18, 19 y 20, incluir una descripción del modo en que fue resuelto el tema correspondiente en cada etapa de desarrollo del compilador.
  - Conclusiones