CONSIDERACIONES GENERALES:

El trabajo deberá realizarse en grupos de dos o tres integrantes.

El compilador se implementará en tres etapas:

- Análisis léxico y sintáctico.
- Análisis léxico, sintáctico y semántico con generación del AST.
- Análisis léxico, sintáctico y semántico con generación de código ejecutable a partir del AST.

Para la aprobación de la cursada no se exigirá la implementación total del compilador pero si se pedirá la entrega de una parte en cada una de las etapas, dejando el resto de la implementación para ser presentada al momento de rendir el examen final.

La aprobación de cada etapa será requerida para acceder a la siguiente etapa.

ETAPAS:

1ra Etapa: Realizar un analizador sintáctico utilizando las herramientas vistas para tal fin.

El programa debe mostrar por pantalla las reglas sintácticas que va realizando el analizador, teniendo como entrada un archivo con código fuente dado para la validación del compilador.

Será requerido: Implementar el análisis léxico y sintáctico para el lenguaje descripto en este documento, pudiendo hacer una entrega en la que sólo se incluya una de las estructuras de selección (**`if`** o **`when`**), una de las estructuras de iteración (**`while`** o **`for`**) y uno de los casos especiales a considerar en el análisis léxico (ignorar comentarios correctamente balanceados o realizar el procesamiento de los códigos de escape dentro de las cadenas de caracteres).
No está de más aclarar que siempre será preferible una entrega soportando todo lo mencionado.
En esta etapa no se pedirá realizar una validación semántica del código analizado.

Fecha de entrega: **5 de mayo de 2021**.

2da Etapa: Aumentar el analizador sintáctico desarrollado en la etapa anterior para que pueda construir un AST semánticamente correcto para el código de un programa dado como entrada.

El AST del programa compilado se deberá mostrar a través de un archivo escrito con sintaxis DOT para generar una visualización mediante graphviz[1]. En esta etapa, el compilador deberá verificar que los tipos de datos de las expresiones, variables, funciones y parámetros sean correctos, agregando conversiones implícitas donde se las necesite o reportando errores donde no se las admita (por ejemplo, un valor booleano en una expresión numérica o un valor numérico en una expresión booleana). Además, deberá asegurar que los identificadores (variables o funciones) existan en el ámbito donde se los utiliza y su uso sea correcto.

Será requerido: A definirse.

Fecha de entrega: **A definirse**.

3ra Etapa: Realizar un compilador utilizando el código desarrollado en la etapa anterior.

El compilador deberá generar un AST para el código del programa dado como entrada y, tras validarlo semánticamente y realizar las optimizaciones pertinentes, generar código intermedio apropiado (usando la IR de LLVM) para producir una versión ejecutable del programa dado.

Será requerido: A definirse.

Fecha de entrega: **A definirse**.

1 https://www.graphviz.org

DEFINICIÓN DEL LENGUAJE

Comentarios:

Deberán estar delimitados tanto por los caracteres "`(*`" y "`*)`" así como por los caracteres "`{`" y "`}`", permitiendo y controlando posibles niveles de anidamiento, asegurando que cada apertura se corresponda con su cierre correspondiente.

También es posible definir un comentario con "`#`", indicado que el comentario termina al final de la línea. Ejemplos:

```
..... # ................#.......#...........   (válido)
..... (* ............................*) .....   (válido)
..... { .......(*........*).......... } .....   (válido)
..... (* ................*).........*) .....   (inválido, no están balanceados)
..... { ..........(*..........}.......*).....   (inválido, están mal balanceados)
```

Identificadores:

Los nombres de los identificadores definidos en este lenguaje deberán cumplir la restricción de comenzar con una letra o un guión bajo, siguiendo con una o más letras, dígitos o guiones bajos ('_'), pudiendo terminar opcionalmente con un signo de interrogación ('`?`').

Las letras soportadas serán aquellas que el estándar Unicode considera como tal, pudiendo ser tanto mayúsculas como minúsculas.

Se distinguen las diferencias entre mayúsculas y minúsculas. Ejemplos:

`i` **`variable`$_1$** **`monto_total`** **`esPar?`** **`_extra`** **`_3`** **`año`**

Tipos de datos:

Se soportan los tipos booleano (**`boolean`**), número entero (**`integer`**) y número representado en punto flotante (**`float`**).

Constantes booleanas literales:

Se utilizan los valores **`true`** y **`false`** para representar verdadero y falso respectivamente.

Constantes numéricas literales:

Pueden definirse con o sin parte decimal. Si tienen parte decimal, se asumen de punto flotante. En caso contrario, se asumen enteras. Para las constantes de valores reales deberá utilizarse obligatoriamente el punto (`.`) como separador decimal.

Algunos ejemplos de constantes de valores en punto flotante:

`99999.99` **`99.`** **`.9999`**

Constantes de cadenas de caracteres:

Se representan como un texto delimitado por comillas dobles ("`"`"), de la forma **`"xxx xxx xxxx"`**. Se acepta que dentro de la cadena puedan escribirse comillas dobles siempre que se las anteponga con una barra invertida ("`\"`").

De forma similar, la combinación "**`\n`**" se interpretará como un carácter de nueva línea y la combinación "**`\t`**", como carácter de tabulación.

La barra invertida, por usarse como carácter de escape dentro de la cadena, deberá siempre ser escrita anteponiéndole otra barra invertida (quedando finalmente como "\\") para que sea interpretada como un único carácter "\". Algunos ejemplos de cadenas literales:

```
"Esta es una cadena de caracteres."
"Primera línea\nSegunda \"línea\"\nLa \\última/ línea"
```

Operadores aritméticos:

Se deberán admitir las operaciones de suma (`a + b`), resta (`a - b`), multiplicación (`a * b`) y división (`a / b`) junto con la de valor opuesto (`-a`). Se podrán agregar otras operaciones adicionales que se consideren apropiadas.

Se cumplirá con la precedencia y asociatividad usual de las operaciones matemáticas, pudiendo igualmente usarse los paréntesis para modificarla.

Operadores de comparación:

Se podrán comparar dos expresiones por igualdad (`a == b`), desigualdad (`a != b`), mayor (`a > b`), mayor o igual (`a >= b`), menor (`a < b`) o menor o igual (`a <= b`).

Todos los operadores de comparación poseen la misma precedencia y no son asociativos.

Operadores lógicos:

Se deberán admitir las operaciones de conjunción (`a and b`), disyunción (`a or b`) y negación (`not a`). Se cumplirá con la precedencia y asociatividad usual de dichos operadores (negación, conjunción, disyunción), pudiendo usarse los paréntesis para modificarla.

Expresiones:

Una expresión puede combinar operaciones aritméticas entre constante y variables numéricas o invocación a funciones que retornan un valor numérico compatible.

Una expresión también puede incluir operadores lógicos y relacionales aplicados a constantes, variables o invocaciones a funciones que retornan un valor apropiado para ese contexto.

La precedencia y asociatividad de los operadores se indica en la siguiente tabla:

Operador	Asociatividad
`or`	Izquierda
`and`	Izquierda
`not`	Izquierda
`== != > >= < <=`	No es posible asociar
`+ -`	Izquierda
`* /`	Izquierda
`-` (unario)	Izquierda

donde la precedencia está dada de menor (operador "`or`") a mayor (operador "`-`" unario).

Definición de una variable:

Las variables se pueden definir en cualquier punto del programa, utilizando la sintaxis:

```
variable <nombre variable> is <tipo>;
```

Además, pueden inicializarse en la misma definición, utilizando la siguiente sintaxis:

```
variable <nombre variable> is <tipo> = <expresión>;
```

Donde **<nombre variable>** es un nombre de identificador válido, **<tipo>** es uno de los tipos soportados y **<expresión>** es una de las expresiones definidas previamente.

En el caso de tratarse de una variable global, dicha expresión no deberá admitir invocaciones a funciones como parte de la expresión (no ocurriendo lo mismo con variables locales a un bloque de sentencias o función, donde dicha restricción no existe).

Asignación:

Una asignación será de la forma :

```
<nombre variable> = <expresión>;
```

Entrada y salida:

Las funciones **read_integer()**, **read_float()** y **read_boolean()** permitirán leer desde el teclado valores del tipo indicado en el nombre de cada función.

Las funciones **write** y **writeln** muestran valores en la salida estándar, cada una admitiendo dos posibles variantes:

```
write(<expresión>);          |   write("un texto");
writeln(<expresión>);        |   writeln("un texto");
```

Las variantes de la izquierda mostrarán el valor asociado a **<expresión>** aplicando el formato apropiado para su tipo, que puede ser un valor entero, en punto flotante o booleano.

Las de la derecha, en cambio, admiten una constante de tipo cadena de caracteres, la que se mostrará en pantalla en forma literal.

Las dos variantes de **writeln** agregarán un fin de línea a sus salidas mientras que las otras, no.

Selección:

Se pueden expresar de las siguientes formas:

```
if <condición> then <sentencia>;
if <condición> then <sentencia1> else <sentencia2>;
```

Es decir, se debe soportar la posibilidad de usar un **if** con y sin un **else** asociado. Todo **else** se asociará implícitamente al último **if**, usando un punto y coma (“**;**”) para marcar un **if** sin un **else**. “**<sentencia>**” representa una de las posibles sentencias (o un bloque).

Adicionalmente, se soporta una forma de selección múltiple de la siguiente manera:

```
when <expresión>
  is <comparador1> <expresión1> then <sentencia1>
[ is <comparadork> <expresiónk> then <sentenciak> ]*
[ else <sentencian> ];
```

donde se admiten uno o más casos de **is** … **then** opcionalmente terminados con un **else**.

<expresión> es una expresión cualquiera cuyo valor será comparado secuencialmente con los de las distintas **<expresión$_k$>** (usando el **<comparador$_k$>** indicado para cada una) hasta hallar un caso que coincida, ejecutando la **<sentencia$_n$>** asociada.

Si ninguna de las comparaciones coincidiera, se ejecutará la sentencia **<sentencia$_n$>** asociada al **else** (eso es, si se especifica dicho caso).

Los comparadores soportados son los mismos que tendrían sentido en una expresión:

== != > >= < <=

La forma del **when** es equivalente a esta otra construcción alternativa usando **if**:

```
variable <temporal> is <tipo expresión> = <expresión>;
if <temporal> <comparador1> <expresión1> then <sentencia1>
[ else if <temporal> <comparadork> <expresiónk> then <sentenciak> ]*
[ else <sentencian> ];
```

El uso de la variable **<temporal>** es para calcular una única vez el valor de la expresión.

Iteración:

Se admitirán dos sentencias de iteración:

```
while <condición> do <sentencia>;
```

La sentencia **while** aceptará una **<condición>** para controlar la iteración y una **<sentencia>** que se repetirá hasta que la condición deje de cumplirse.

```
for <variable> from <inicio> to <fin> [ by <salto> ] do <sentencia>;
```

La sentencia **for**, en cambio, usará una variable de tipo **integer** (que previamente deberá ser definida) e iterará asignándole **<inicio>** como valor inicial, incrementándola en cada iteración de acuerdo al valor dado opcionalmente en **<salto>** (aunque, si se omite, se asume que es **1**) hasta alcanzar (o superar) el valor indicado por **<fin>**. Si **<fin>** fuera menor que **<inicio>**, la variable se decrementará en cada iteración en lugar de incrementarse.

Las sentencias **break** y **continue** pueden alterar el flujo de control habitual de las sentencias de repetición **while** y **for** y sólo serán admitidas dentro de las sentencias asociadas a dichas estructuras de control.

Bloques de sentencias:

El comienzo y final de un bloque de sentencias está delimitado por las palabras **begin** y **end**.

En el siguiente ejemplo, se muestra como **<sentencia$_1$>** y **<sentencia$_2$>** pertenecen a un mismo bloque (definido entre el **begin** y el **end** del **then**) mientras que **<sentencia$_3$>** no está contenida en un bloque, siendo una sentencia simple.

```
if <condición> then begin
  <sentencia1>;
  <sentencia2>;
end else
  <sentencia3>;
```

Ni la indentación ni los saltos de línea son relevantes para la definición de un bloque.

Las estructuras de control admiten tanto una única sentencia así como un bloque de sentencias delimitadas por **begin** y **end**. En ambos casos se creará un nuevo ámbito local.

Funciones:

La declaración de una función se realiza dentro del cuerpo principal del programa, de la forma:

`function <nombre> ( <parámetros> ) return <tipo_retorno> <sentencia>;`

Donde **`<nombre>`** cumple las mismas reglas que el resto de los identificadores y **`<parámetros>`** es una lista potencialmente vacía de definiciones para los parámetros, de la forma:

nombre_parámetro$_1$ is tipo$_1$, nombre_parámetro$_2$ is tipo$_2$ = valor_por_def$_2$

Es posible definir valores por defecto para los parámetros (como se muestra para el segundo parámetro con el valor **`valor_por_def`**$_2$). No podrán definirse parámetros sin valor por defecto luego de la primera aparición de un parámetro que incluya un valor por defecto.

Se utilizará la sentencia **`return`** para devolver una expresión como resultado de la invocación a una función. No se permite el anidamiento de funciones (funciones dentro de otras funciones).

En el caso que el cuerpo de una función consista en una única sentencia, no será necesario delimitarlo con **`begin`**/**`end`** (aunque, si bien sería redundante, igualmente se lo podría hacer).

Ejemplo:

```
function factorial(n is integer) return integer
  if n > 1 then
    return n * factorial(n – 1)
  else
    return 1;
```

El nombre de un identificador seguido de una lista (posiblemente vacía) entre paréntesis de expresiones será interpretado como una invocación a una función siempre que aparezca como parte de una expresión. Se admite invocar una función sin consumir el valor retornado.

En el caso que una invocación a una función tenga menos parámetros que los declarados, los parámetros restantes podrán tomar el valor por defecto indicado para el mismo. Si el parámetro faltante no tuviera asociado un valor por defecto, deberá considerarse un error.

Ejemplo:

```
resultado = 1 + factorial(4) / 2.5;
```

Programa:

En un programa pueden aparecer sin un orden particular definiciones de variables globales y funciones, finalizando con un bloque de código delimitado por las palabras claves "**`main is`**" y terminado con "**`end.`**".

Dicho bloque será considerado como el punto de inicio del programa y a continuación del mismo no podrán aparecer más definiciones de variables globales y funciones.

Ejemplo de un programa completo:

```
(* Se definen (* algunas *) variables globales *)
variable radio is float = 1.5;
variable superficie₁ is float = 0;

{ Una función }
function superficie_círculo(radio is float = 1) return float
begin
  writeln("Dentro de función superficie_círculo()");
  # Definición de una variable local.
  if radio < 0 then
    return 0
  else begin
    variable radio² is float = cuadrado(radio);
    return 3.14 * radio²;
  end;
end;

(* Otra función pero con una única sentencia *)
function cuadrado(valor is float) return float
  return valor * valor;

{ Otra variable global (esta vez, sin inicialización explícita) }
variable superficie₂ is float;

# Procedimiento principal
main is
  writeln("Inicio del programa");
  writeln(superficie₁);
  writeln(superficie₂);
  superficie₁ = superficie_círculo(radio);
  writeln(superficie₁);

  radio = read_float();
  superficie₂ = superficie_círculo(radio);
  writeln(superficie₂);

  variable justo is boolean = false;
  when superficie_círculo()
    is < 100 then writeln("Es pequeño")
    is == 100 then begin
      writeln("Qué puntería!");
      justo = true;
    end
    else writeln("Es grande");

  write("Acertó: "); writeln(justo);

  variable i is integer;
  for i from 5 to 1 do begin
      write("Se acaba en "); writeln(i);
  end;
  writeln("Se acabó\n\nFin!");
end.
```