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: Implementar en el AST y visualizar las mismas sentencias implementadas en la etapa anterior. Soportar la validación de tipos y reportar los errores apropiados.
No se exigirá agregar conversiones implícitas.

Fecha de entrega: **4 de junio de 2021**.

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: **25 de junio de 2021**.

[1] https://www.graphviz.org

GENERACIÓN DE CÓDIGO

Identificadores:

Dado que los nombres de los identificadores definidos en este lenguaje admiten varios caracteres incluidos en el estándar Unicode pero LLVM sólo admite un subconjunto de los caracteres ASCII en sus identificadores, será necesario realizar algún tipo de reemplazo y/o traducción, de forma que cada nombre de variable o de función tenga una representación válida y única dentro del código del programa generado.

Por ejemplo, los nombres **`día`** y **`dia`** son diferentes pero si ambos se convirtieran en el nombre **`dia`**, se generaría una colisión no presente en el código original. Deberá asegurarse que cualquier adaptación de los nombres de caracteres Unicode a ASCII genere nombres únicos.

Corto-circuito booleano:

El corto-circuito booleano consiste en no evaluar completamente las expresiones involucradas en los operadores **`and`** y **`or`**, dado que en ciertas situaciones, observar el valor lógico de la expresión de la izquierda alcanza para definir el valor lógico que arroja el operador.

En el caso del **`and`**, si el valor de la expresión de la izquierda es **`false`**, no es necesario evaluar la expresión de la derecha ya que, independientemente del valor de dicha expresión, el resultado de la operación está garantizado que será falso.

Lo mismo sucede con el operador **`or`**: Cuando el valor de la expresión de la izquierda es **`true`**, queda garantizado que el resultado de toda la operación será verdadero, sin importar el valor lógico de la expresión de la derecha.

Sentencias **`switch`** y **`for`**:

Dichas sentencias pueden convertirse en construcciones equivalentes usando sentencias **`if`** o **`while`** junto con bloques, definición de variables locales y asignaciones.

Entonces, resulta conveniente hacer transformaciones sobre el AST para convertir dichas sentencias en nodos equivalentes, en vez de implementar la generación de código tanto para el **`if`** con para el **`switch`** así como para el **`while`** y para el **`for`**.

REQUISITOS PARA LA APROBACIÓN DE LA CURSADA:

Primera etapa:

- Análisis léxico parcial, pudiendo no implementar el procesamiento del texto dentro de las cadenas de caracteres o el manejo de comentarios correctamente anidados (una de las dos).
- Análisis sintáctico parcial, pudiendo no implementarse:
  - Una de las alternativas a la selección (implementando sólo el **`if`** o sólo el **`switch`**).
  - Una de las alternativas a la iteración (implementando sólo el **`while`** o sólo el **`for`**).

Segunda etapa:

- Generación del AST, pudiendo no mostrar a cada uno de los nodos del árbol con el mayor detalle posible.
- Análisis semántico:
  - Resolución de los identificadores a las variables o funciones que realmente nombran.
  - Verificación de tipos en expresiones, parámetros, etc.
  - Opcionalmente, realizar conversiones implícitas entre las partes de una expresión.
  - Opcionalmente, la verificación de la aparición de **`break`**, **`continue`** y **`return`** dentro de los ámbitos apropiados.
  - Opcionalmente, el cálculo de constantes en tiempo de compilación.

Tercera etapa:

- Implementar la generación de código IR de LLVM para todas los nodos del AST requeridos en las etapas anteriores. Se esperaría que el código generado pueda compilarse y ejecutarse.
- Opcionalmente, implementar corto-circuito booleano para los operadores lógicos.
- Opcionalmente, implementar la alteración del flujo de control que producen las sentencias **`break`** y **`continue`** sobre las estructuras de iteración **`while`** y **`for`**.