

PROYECTO ESPECIAL

DISEÑO E IMPLEMENTACIÓN DE UN LENGUAJE

v2.2.1

DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

12 de Agosto de 2023

1 Introducción

LOS compiladores son elementos clave de la informática y los que permiten convertir prosa digital en acción. Hay cientos, miles, y probablemente en cada clase de *Lenguajes y Compiladores* del mundo se construya uno. No vamos a hacer menos entonces...

1.1 Objetivos

El objetivo principal de este trabajo es el diseño y desarrollo completo de un lenguaje y su compilador, construyendo sus dos componentes principales: el *frontend* (**Sec. §2**), y el *backend* (**Sec. §3**). Para ello, se deberá idear un lenguaje y sus prestaciones, se redactará una gramática que defina su sintaxis y se implementará una aplicación (*i.e.*, el compilador), que transformará un programa en un artefacto final sobre alguna plataforma. El compilador deberá estar desarrollado en lenguaje C, utilizando Flex y Bison como analizador léxico y sintáctico, respectivamente.

1.2 Comunicación

Las consultas por e-mail, y el detalle de la información requerida por escrito, se deben enviar hacia las siguientes direcciones (con copia, a todas ellas):

- juaarias@itba.edu.ar (Juan Pablo Arias)
- lferreiro@itba.edu.ar (Lucas Agustín Ferreiro)
- mgolmar@itba.edu.ar (Agustín Golmar)

2 Parte I: Frontend

LA arquitectura básica de un compilador se puede explicar mediante el diagrama de la **Fig. (1)**. Esta estructura general permite transformar un programa de entrada codificado en el lenguaje expresado por un gramática (interna al compilador), en un programa ejecutable final para cierta plataforma (*i.e.*, hardware, sistema operativo, etc.).

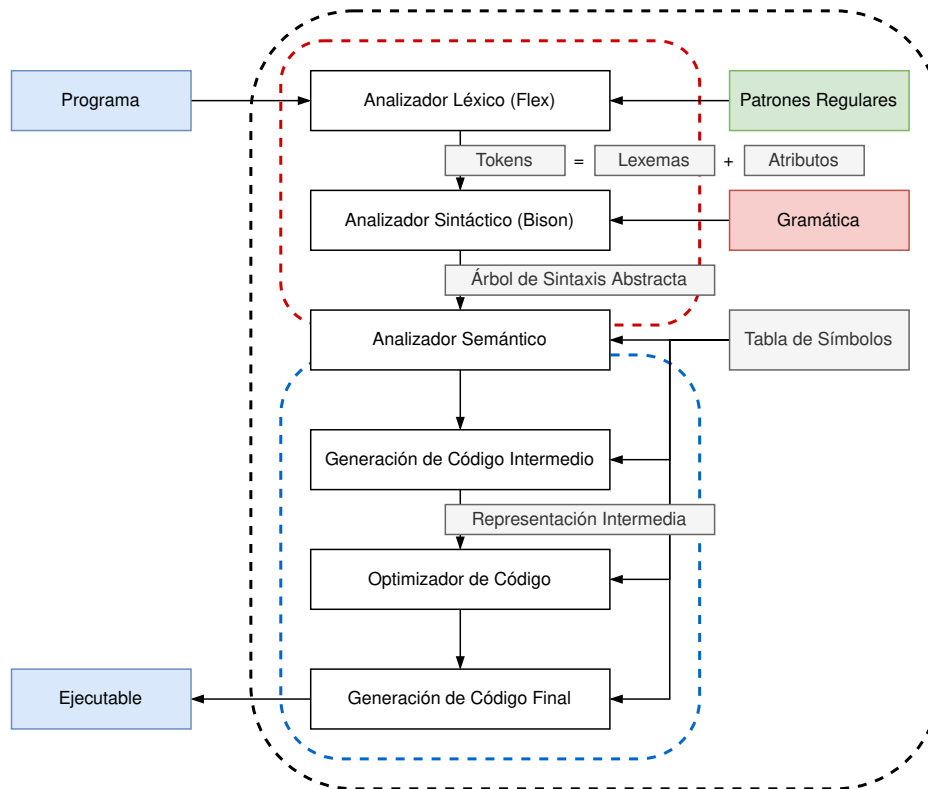


Figura 1: La arquitectura de un compilador y sus componentes principales. Las fases agrupadas en rojo conforman el *frontend*, y las agrupadas en azul, el *backend*.

Internamente, el compilador aplica varias transformaciones sobre el programa de entrada, donde cada una de estas transformaciones se denomina *fase*, y produce como resultado una representación aumentada del programa, la cual tiene por objetivo proveer cierta información, así como también una estructura eficiente para el procesamiento realizado durante la siguiente fase.

En general, se establece una distinción principal entre las fases según el grado de dependencia que posee la representación que producen con respecto a la plataforma final sobre la que el programa será ejecutado. En este sentido, si la representación es independiente de la plataforma objetivo final, entonces pertenece al *frontend*; caso contrario, si la representación depende en alguna medida, entonces la fase que la produce pertenece al *backend*.


La literatura establece diferentes “puntos de corte” entre el *frontend* y el *backend*, debido a que la estructura particular del compilador depende del lenguaje que manipula en su entrada. Para este trabajo en particular, se considerará que el **analizador semántico** es aquella fase que diferencia cada mitad del compilador. Debido a esto, se espera que la primera entrega del

compilador contemple la construcción de:

- Un documento `*.l` de **Flex** que contenga la definición de los patrones/expresiones regulares utilizados por el escáner de entrada del compilador (*i.e.*, el analizador léxico).
- Un documento `*.y` de **Bison** que contenga la definición de la gramática \mathcal{G} completa del lenguaje propuesto $\mathcal{L}(\mathcal{G})$, utilizando los *tokens* provistos por el analizador léxico. Esta gramática permite que **Bison** construya el analizador sintáctico *LALR(1)*.
- Un *recognizer*, esto es, una función $\mathfrak{R} : \mathcal{L}(\mathcal{G}) \rightarrow \{0,1\}$, donde $\mathcal{L}(\mathcal{G})$ es el lenguaje implementado, y $\mathfrak{R}(l) = 1$ si el programa l es válido (*i.e.*, $l \in \mathcal{L}(\mathcal{G})$), o $\mathfrak{R}(l) = 0$ de otro modo.

El lenguaje $\mathcal{L}(\mathcal{G})$ propuesto, deberá tener ciertas prestaciones a definir por el equipo que lo desarrolle, siendo que algunos ejemplos generales de ellas podrían ser (no todas son obligatorias y depende del caso concreto de cada equipo):

- Tipos de datos fundamentales, tanto numéricos como de texto (*i.e.*, *strings*). Podría ser posible expresar valores constantes y/o literales para estos tipos de dato.
- Un conjunto básico de operadores aritméticos, relacionales, lógicos y de asignación. Los operadores deben establecer un orden definido de *precedencia* y *asociatividad*, ya sea, mediante directivas de **Bison** (`%left`, `%right`, `%nonassoc` y `%precedence`), o mediante una estructura jerárquica dentro de la gramática (usando factores, términos, expresiones, etc.).
- Estructuras de control *condicionales*, al estilo *IF-THEN-ELSE* o similar. Es posible ofrecer otro tipo de estructura de control análoga, como por ejemplo *pattern-matching* \bowtie , pero de cualquier manera, estas permiten alterar el flujo normal de ejecución de alguna forma.
- Estructuras de control *iterativas*, al estilo *WHILE* o similar. La idea detrás de este tipo de control es poder procesar datos, objetos, entidades o cualquier otro concepto análogo de forma masiva, incrementando la reusabilidad del código.
- Un punto de entrada principal, al estilo *MAIN*. Dependiendo del tipo de lenguaje a desarrollar, puede ser que este requerimiento se modifique o relaje de alguna forma, pero en todo caso debería existir una justificación objetiva para ello.
- Un mecanismo de entrada y salida de datos, por ejemplo, para ingresar datos desde la consola (`stdin`), y para imprimir sobre la misma (`stdout`).
- Algún anidamiento (como el uso de bloques, módulos o namespaces) para controlar el *scoping* de los objetos, variables, constantes, métodos, clases y funciones declarados en el lenguaje.
- Comentarios single-line o multi-line. Comentarios que permitan documentación, como Javadoc o JSDoc, entre otros.

Para el desarrollo de esta primera parte, y para luego continuar sobre la misma solución en la entrega final, se recomienda utilizar la versión `v0.2.0` (branch `master`), del proyecto base [agustin-golmar/Flex-Bison-Compiler](#) , aunque no es obligatorio.

2.1 Validación y Verificación


Test-Driven Development (*a.k.a.* TDD), es una metodología de desarrollo de software que implica redactar un requerimiento inicial, un caso de uso que lo valide y represente, y su correspondiente *test* de verificación, antes siquiera de realizar la implementación de la funcionalidad asociada al requerimiento.

En el contexto de este proyecto, no será solicitada la implementación de una batería de *tests* sobre las fases del compilador, ni un porcentaje de cobertura de código determinado, pero en su lugar, se utilizarán las prestaciones del lenguaje ideado descritas en la **entrega de la idea inicial y la definición del grupo**, para ofrecer a cambio una cantidad determinada de *programas de validación de requerimientos* que deberán ser redactados y entregados, tanto con el *frontend*, como en la entrega final del proyecto.

Estos programas permiten verificar que las prestaciones evidentemente fueron implementadas, incluso parcialmente (como es el caso para el *frontend*, quien solo podrá indicar si el programa es válido o no, pero nada más). Junto a los programas de validación, se solicitarán programas que **deben fallar**, los cuáles serán rechazados por la solución, ya sea por una semántica errónea (en el caso del *backend*), o por un error de sintaxis (*frontend* y *backend*).

Se recomienda para todos los equipos el agregado de más casos de uso por encima del mínimo requerido, tanto de aceptación como de rechazo, a medida que desarrollen el proyecto, para mejorar la calidad del mismo y acelerar su desarrollo mediante un testing continuo de sus funcionalidades.

3 Parte II: Backend

 E la **Sec. §2**, y en particular como se observa en la **Fig. (1)**, quedó establecido que el *analizador semántico* es quien separa el *frontend* del *backend*. Esto se debe a que incluso en esta fase, y dependiendo del lenguaje subyacente, es posible que la aplicación ya se encuentre frente alguna dependencia con respecto a la plataforma objetivo, que limite la portabilidad del compilador.

Dicho esto, la segunda entrega de este proyecto especial contempla la construcción completa de todas las fases restantes, desde el análisis semántico, hasta la generación final del ejecutable. Es posible, no obstante, que el compilador no tenga por objetivo producir un ejecutable final, sino más bien una representación en otro lenguaje intermedio que luego pueda ser compilado mediante otra aplicación.

La **Fig. (2)** representa un *diagrama T*, los cuales se utilizan para describir los 3 lenguajes

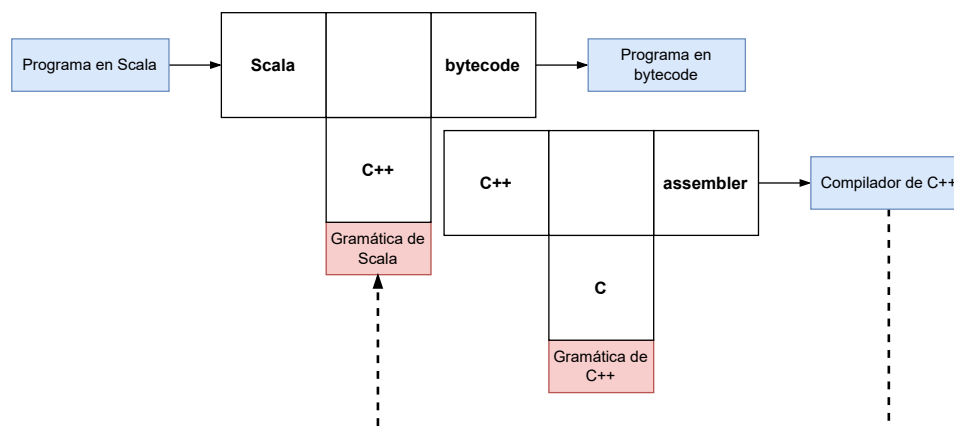


Figura 2: Diagrama T de un sistema que transforma código en lenguaje Scala a bytecode, el cual puede ejecutarse sobre una JVM (*Java Virtual Machine*). A su vez, el compilador de Scala se desarrolló en lenguaje C++, pero fue compilado a **assembler** mediante un compilador escrito en C.

que todo compilador manipula. En este caso, un programa escrito en lenguaje Scala (lenguaje *fuentes*), se transforma en **bytecode** mediante un compilador escrito en C++. Este compilador contiene una gramática de Scala en su interior (no necesariamente de forma explícita), la cual le permite a su analizador sintáctico interpretar y transformar los programas de entrada. A su vez, el compilador de C++ también es obtenido de forma análoga, aunque bajo diferentes lenguajes, ya que en definitiva, el compilador también es un programa.

En la implementación de la fase de **generación de código** (probablemente la última fase del compilador), es común recorrer el *árbol de sintaxis abstracta* (*a.k.a.* AST), construido luego de aplicar el análisis sintáctico y semántico (*i.e.*, las fases anteriores), y producir sobre cada nodo de la estructura un fragmento del código final. La generación de código, por lo tanto, puede ser tan simple como ejecutar instrucciones del tipo:

Código 3.1 Un ejemplo simplificado de cómo generar código en C para una función determinada. Esta función pertenece al programa que se está compilando, que seguramente proviene de otro lenguaje diferente a C.

```

1: /**
2:  * Imprime en lenguaje C, la definición de una función que no retorna
3:  * nada y que recibe un único parámetro. Se utiliza la salida estándar
4:  * para imprimir el fragmento de código.
5:  */
6: void GenerateCodeForFunction(const char * functionName,
7:   const char * parameterType, const char * parameterName) {
8:   // Utilizar otro descriptor en lugar de 'stdout' de ser necesario:
9:   fprintf(stdout, "void %s(%s %s)\n", functionName, parameterType, parameterName);
10: }
```

En general, el lenguaje de salida tiende a ser de más bajo nivel (*i.e.*, más cercano y dependiente de la plataforma objetivo), aunque esto no siempre ocurre. Es importante considerar

que la generación de código está ligada a la arquitectura sobre la cual se desea o pretende ejecutar/interpretar el programa compilado (*e.g.*, IA-32e, IA-64, ARM, etc.).


En el contexto de este trabajo, el diagrama T asociado debe:

- Tomar como lenguaje fuente el lenguaje diseñado por el grupo, e implementado parcialmente en el *frontend*.
- Utilizar C para redactar el compilador de este lenguaje, continuando las siguientes fases restantes del *backend*.
- Producir un lenguaje de salida también arbitrario, que puede ser binario ejecutable para alguna plataforma (en particular, debe funcionar en la plataforma de los servidores que cada alumno posee en el ITBA), o bien un lenguaje intermedio, que deberá recompilarse para obtener finalmente un ejecutable. En otros casos, la salida es un artefacto de otro tipo, como un documento de texto, una imagen o un sonido.

Adicionalmente, el compilador final debe:

- Implementar la fase de análisis semántico y generación de código final de forma obligatoria, generando un programa compilable sin errores, o un ejecutable completamente funcional.
- Implementar **opcionalmente** una fase de optimización de código y/o de generación de código intermedio (se evaluará como un adicional, en caso de que el grupo desee implementar dichas fases, o alguna de ellas).
- Garantizar que los *tests* provistos en la primera entrega para verificar el funcionamiento del *frontend* continúen operando en la versión final, hayan sufrido o no modificaciones (se deben adaptar de ser necesario). Aquellos tests que debían fallar pero que eran aceptados debido a que el compilador no poseía backend, ahora deben ser rechazados como es esperado.

4 Entregables

 La aprobación del proyecto requiere de 3 entregas obligatorias (la primera es conceptual).

4.1 Parte I: Frontend

La entrega del *frontend* se realizará a los mismos e-mails utilizados en la **Sec. §1.2**. Esta entrega debe incluir:

- El nombre de la rama (*branch*) utilizada para almacenar el código en el repositorio en el que se está desarrollando el proyecto (el mismo que se presentó al proponer la idea).

- El *commit hash* que representa la entrega. Esto implica que las sucesivas modificaciones luego del *commit* asociado no se tendrán en cuenta, lo que implica que el grupo puede seguir trabajando sin problemas sobre el mismo repositorio de código antes de obtener una evaluación de la entrega realizada.
- Un conjunto de programas para verificar los casos de uso requeridos. Los mismos deben ser versionados en el repositorio que contiene el *frontend*, y deben estar claramente diferenciados aquellos que forman programas válidos de aquellos que no lo son, y que por lo tanto deben ser rechazados por la solución. Para ello, se dispone en el repositorio de ejemplo una carpeta **test** donde se ubican los casos de uso de aceptación y rechazo. Además, se provee un script con el cual ejecutar toda la batería de tests en dicho directorio, lo que simplifica el testing continuo (ver README del proyecto). La cátedra utilizará dicho script para probar el proyecto, con lo cual, de modificarse el mismo, se deberá garantizar que funciona **exactamente igual**.
- Un *recognizer* desarrollado mediante las herramientas **Flex** y **Bison**. Si bien el *recognizer* no representa la totalidad de lo que se considera un *frontend*, es suficiente para esta entrega que sea posible simplemente reconocer qué programas respetan la sintaxis del lenguaje ideado y cuáles no. Este recognizer debe aceptar el programa por **entrada estándar** (**stdin**), y emitir los posibles mensajes de *logging* por **salida estándar** (**stdout**). Además, el programa debe retornar **0** (cero), en caso satisfactorio (aceptación), y cualquier otro número en caso de rechazar el programa de entrada. No respetar esta interfaz hará que el script de testing falle de manera inesperada.

La solución entregada puede implementar ciertas validaciones del *analizador semántico*, o algunas de las estructuras de datos necesarias (como el *árbol de sintaxis abstracta*, o la *tabla de símbolos*), pero esto no es obligatorio, es decir, el *frontend* puede encontrarse en un estadio más avanzado del requerido, siempre y cuando cumpla con los requerimientos de esta primera parte.

4.2 Parte II: Backend

La entrega del *backend* se realizará a los mismos e-mails utilizados en la **Sec. §1.2**. Esta es la última entrega, y por ende, debe incluir:

- *Branch* y *commit hash* final de todo el proyecto completo. Cualquier modificación posterior al *commit* asociado no será tenida en cuenta para la nota final del proyecto.
- Un compilador funcional del lenguaje propuesto, con la implementación de todas las fases definidas y requeridas en la **Sec. §2** y en la **Sec. §3**. Nótese que este compilador ya no será más un *recognizer* como lo fue solicitado en la primera entrega, sin embargo, esto no quita que la aplicación pueda (opcionalmente) proveer este modo alternativo de operación (quizás mediante alguna opción por parámetro, o variable de entorno).

- Todos los programas de validación del compilador utilizados para la entrega del *frontend*. Los mismos deben ser modificados de ser necesario para esta versión final del compilador. En caso de que algunas validaciones ya no apliquen, se deben reemplazar por otras validaciones en diferentes o en nuevas prestaciones introducidas luego de la primera entrega, pero la cantidad de casos de testeo mínimos se debe mantener (10 de aceptación y 5 de rechazo).
- Un informe digital en formato PDF o en formato MD (*Markdown*¹). Si el informe se encuentra en formato PDF, entonces se debe versionar en el repositorio de la solución, dentro de una carpeta con nombre `doc` en la raíz del mismo. Si se redacta en formato MD, se debe colocar en el mismo documento `README.md` del repositorio. El informe debe contener:
 - Una portada (si se utilizó el formato PDF).
 - Una tabla de contenidos. Recordar que no es lo mismo que un índice. La tabla se ubica al inicio del informe y enumera las secciones, mientras que un índice se coloca al final y representa una larga lista de términos y conceptos junto con todos los números de página en los que se trata dicha *keyword*.
 - Introducción con idea subyacente y objetivos del lenguaje.
 - Consideraciones adicionales no previstas en este enunciado que afecten a la implementación del compilador y que sean específicas del dominio atacado por el lenguaje desarrollado. No es necesario copiar y pegar la gramática o la lista de reglas de Flex, ya que las mismas se encuentran versionadas en el repositorio. Esto no quita que se pueda mostrar cierto extracto de las mismas para especificar alguna decisión de diseño que sea relevante. En caso de optar por incluir la gramática y reglas de todas formas, se deberá hacer en un apéndice, al final del informe.
 - Descripción del desarrollo del proyecto y de las fases del compilador. Es importante describir lo que es específico al proyecto y lenguaje elegido por el equipo; no es necesario redactar una explicación teórica de qué es un analizador sintáctico, léxico, semántico, etc., o cualquier concepto que ya se haya visto en la materia, a menos que se haya modificado de alguna manera que haga necesaria la explicación debido a su relevancia tecnológica. Es útil incluir diagramas de flujo o *snippets* de código que evidencien el uso del lenguaje y la arquitectura del compilador.
 - Dificultades a la hora de desarrollar el proyecto o alguna de sus partes (incluso problemas al concebir la idea original).
 - Futuras extensiones y/o modificaciones, indicando brevemente la complejidad de cada una y las razones por las cuáles sería útil disponer de las mismas.
 - Referencias y bibliografía. La diferencia entre referencia y bibliografía es que las referencias son aquellos materiales de consulta (sitios web, libros, papers, posts, etc.), para los cuales se emplea una referencia dentro del informe de forma explícita (*e.g.*,


¹ <https://www.markdownguide.org/> ↗

“... para más información, ver Aho et al., 2006...”). La bibliografía, en cambio, es material consultado que no se referencia dentro del informe, pero que sí fue accedido y/o leído para comprender o para implementar el proyecto (e.g., la documentación de C/C++, o las clases de la cátedra).

Por ser la entrega final, es posible que el grupo desee considerar el agregado de ciertos adicionales para mejorar la calidad del proyecto en sí mismo, o para obtener una mejor puntuación final. Algunos de estos posibles agregados incluyen:

- Optimizaciones sobre el compilador, en una o en varias de las fases implementadas, para mejorar la operación del mismo en algún sentido.
- Benchmarking² de los tiempos de ejecución de los programas compilados en el nuevo lenguaje, comparados con programas diseñados en otros lenguajes. En general, los programas de prueba utilizados deben realizar un cómputo intensivo en algún aspecto (memoria, tiempo de procesamiento, ancho de banda, IOPS³, etc.), como por ejemplo, un test de *primalidad* sobre números grandes.
- Extras del lenguaje, obviamente sujetos o limitados a la idea particular de cada solución. E.g.: manejo de concurrencia, la implementación de alguna librería para manipular estructuras de datos, operaciones vectorizadas, tipos de datos avanzados (matrices, vectores, números complejos, etc.), segregación del programa en múltiples archivos (i.e., como las directivas `#include` de C/C++, o los `import` de Java), entre otros.
- Un *syntax highlighter* del lenguaje creado, provisto mediante algún programa externo o librería fuera del compilador (e.g., un HTML monolítico o un sitio en GitHub Pages [↗](#), que utilice Prism.js [↗](#)).

5 Sugerencias

 La cátedra recomienda considerar las siguientes sugerencias:

- **Idea del lenguaje:** Proponer un lenguaje que se pretenda utilizar en la realidad, o que se pretenda evolucionar luego de completar el proyecto. Aprovechar el hecho de que la idea del mismo es prácticamente libre, salvo por el uso de Flex/Bison, y de un conjunto de requerimientos básicos que casi cualquier lenguaje de hoy día posee.

² <https://en.wikipedia.org/wiki/Benchmarking> [↗](#)

³ IOPS significa *inputs/outputs per second*, y es una medida comúnmente utilizada para definir la cantidad de operaciones que un sistema soporta o realiza en relación al acceso al hardware, usualmente y en particular, el acceso a disco o a un medio de persistencia.

- **Traducción de salida:** Si bien es posible utilizar otro lenguaje como código de salida y no necesariamente un ejecutable (la cátedra recomienda emitir una representación intermedia, por ejemplo, *bytecode* de Java, que además es multi-plataforma), la traducción entre el lenguaje de entrada y el de salida no debería ser trivial. Considerar un lenguaje que sea muy similar a C y que además produzca C como salida, no implica ningún desafío desde el punto de vista técnico del proyecto.
- **Desplegar con anticipación:** Frente a una modificación del proceso de construcción del compilador, la inclusión de una librería o dependencia externa, o incluso el uso de un proyecto base alternativo al provisto por la cátedra, se debería considerar una prueba de despliegue sobre el servidor del ITBA antes de continuar con el desarrollo, para garantizar que el día de la entrega el proyecto no quede descalificado por completo debido a un error de *deploy*.
- **Usar Flex-tokens:** Es posible crear *tokens* que agrupen múltiples *lexemas* bajo un mismo nombre/etiqueta, como por ejemplo el uso de un *token* ID para representar todos los identificadores de variables. Por ser un *token*, su atributo almacenará el valor concreto que acarrea, pero el analizador sintáctico utilizará la etiqueta ID de forma indistinta.
- **Sistema de tipos:** El lenguaje ideado posiblemente provea construcciones mediante tipos de datos conocidos (como `int` o `string`), pero quizás también sobre tipos de alto nivel (*e.g.*, matrices, objetos, neuronas, pokémons, gomitas, autos, grafos, etc.). Es importante, en particular durante la primera fase de diseño, determinar exactamente qué tipos va a ofrecer el lenguaje y qué operaciones son válidas entre ellos, lo que simplifica enormemente el desarrollo y la correctitud del lenguaje, incluso al usar el mismo.
- **Tabla de símbolos:** Al escanear el programa de entrada es posible almacenar los identificadores de variables, constantes o entidades similares en una estructura de tabla de símbolos. Esta estructura permite que luego un analizador semántico tenga una manera de determinar la declaración previa de variables, el *scope*, o el tipo de dato, entre otros.
- **Árbol de sintaxis abstracta:** Otra estructura de datos útil y necesaria que permite representar el programa como un árbol, donde los nodos hoja representan los terminales (*lexemas*), y los nodos internos representan no-terminales de la gramática. Como cada nodo puede contener atributos adicionales, el árbol completo representa toda la información asociada al programa en compilación, y facilita su recorrido mediante una o varias pasadas, en particular para realizar todas las fases del *backend*.
- **Generación de código:** Se puede compilar un programa de salida mediante GCC (en caso de que aplique), utilizando la opción `-S` para emitir código en ensamblador. Si la salida del compilador es un programa en Java, se puede emplear BCEL, CGLIB y ASM para generar *bytecode*, la representación intermedia de Java que permite ejecutar un programa en cualquier plataforma que posea una JVM (*Java Virtual Machine*). También se puede generar código intermedio (IR), mediante LLVM.

6 Lineamientos Generales

PARA todas las entregas, y según aplique, se deben considerar los siguientes lineamientos obligatorios:

- **Librerías:** No hay ningún inconveniente en utilizar librerías públicas, soluciones similares públicas, soluciones de foros, etc., pero es necesario aclarar y enumerar cada una de ellas en la sección *Referencias* correspondiente al informe final y/o README.md. No se aceptan bloques de código públicos implementados *verbatim* sin ningún tipo de análisis. Tampoco implementaciones que resuelven problemas que no están detallados (*e.g.*, implementar un *garbage collector* sin explicar cómo funciona, o cómo se relaciona con el resto de la aplicación, tanto técnica como conceptualmente). Tampoco hay inconveniente en que interactúen con otros grupos, siempre que se respeten las mismas consideraciones. La diferencia entre plagio y ciencia es una referencia.
- **Compilación sobre salida:** El compilador que no produzca un ejecutable como salida, y en su lugar produzca un nuevo programa en algún lenguaje alternativo (C, C++, Java, assembler, etc.), debe emitir un programa válido, es decir, debe emitir un programa que al compilar genere un ejecutable sin necesidad de ser modificado de ninguna forma. El programa de salida no puede contener errores de ningún tipo.
- **Despliegue del compilador:** El compilador desarrollado debe ser construido y probado desde el servidor del ITBA (`ssh://username@pampero.it.itba.edu.ar:22/`), antes de cada entrega. Esto implica que una prueba del sistema por parte de la cátedra, consiste en clonar el repositorio con la solución dentro de una carpeta vacía, ejecutar los comandos de construcción especificados en el README.md principal, e ingresar algún programa mediante entrada estándar o mediante el mecanismo especificado (*e.g.*, por parámetros de consola, o desde un archivo). No se aplicará ningún proceso adicional, de otro modo la entrega del proyecto se considera inválida. De requerir alguna configuración alternativa (como la ejecución de algún *script* al construir la solución), se deberá proveer como parte de la ejecución del comando principal de construcción (usualmente CMake, o similar).
- **Grupos:** El armado de los grupos y el trabajo en equipo son responsabilidad de los alumnos. La cátedra establece la cantidad máxima de participantes de cada grupo (4 en este caso), pero los alumnos pueden proponer realizar el trabajo con menos integrantes, incluso de forma individual. Esto no implica, bajo ningún motivo, que el trabajo presentado deberá tener alguna consideración especial por ser unipersonal o por poseer una menor cantidad de miembros.
- **Esfuerzo de los integrantes:** En caso de que un grupo presente problemas relacionados a un desbalance en la cantidad de trabajo realizado por cada individuo, el grupo debe entender que solo es posible determinar parcialmente este desbalance según el esfuerzo expresado en el versionado del repositorio que contiene la solución. Sin embargo, si el grupo

no presenta queja explícita en referencia al desbalance, la nota final se considerará única para todo el grupo (lo que aplica normalmente a todos los grupos), sin ningún tipo de penalización, ya que se sobreentiende que si no hubo quejas, es porque el grupo se organizó para distribuir el trabajo de esa forma.

6.1 Criterios de Corrección

Se emplea una puntuación de 0 a 10 para los siguientes criterios, sobre cada una de las entregas del proyecto, según aplique (idea, *frontend* y *backend*):

- Calidad del informe, secciones y estructura, prosa, faltas de ortografía y claridad en la exposición.
- Cumplimiento de todas las consignas técnicas, organizacionales y conceptuales.
- Implementación del compilador, investigación, impacto y pertinencia de la idea, creatividad, innovación, practicidad, definición y exposición de las ideas subyacentes, alineamiento con lo visto en la cátedra.
- Calidad de la presentación, armado del proyecto, documentación del código y de la construcción y despliegue de la solución, estructura del código y separación en fases, *scripts* de automatización, *testing* regresivo, validación de casos de uso, etc.
- Adicionales debido a agregados al proyecto con respecto al enunciado base y sus requerimientos que demuestren mejoras o interés en el mismo.
- Similitud entre la idea propuesta y las prestaciones iniciales con respecto al *frontend* y *backend* desarrollados en cada entrega realizada.

7 Soporte

ESTA sección provee algunos detalles adicionales o material bibliográfico que pueden servir para desarrollar el proyecto, para comprender la teoría subyacente, o para avanzar en el estudio del desarrollo de compiladores más allá de lo visto en la cátedra:

- **Desarrollo en Microsoft Windows:** Para aquellos grupos o individuos que deseen construir el compilador en plataformas de Microsoft Windows, deberán adquirir las herramientas **Flex** y **Bison**. Estas se pueden obtener haciendo click [\[aquí\]](#). Las mismas ya fueron probadas sobre el repositorio base recomendado en la **Sec. §2**, compilando mediante Microsoft Visual Studio 2022 y utilizando una versión de **CMake** para Windows desde [\[aquí\]](#).

- **Ejemplos adicionales:** Se pueden ver y probar más ejemplos de expresiones regulares y gramáticas de Flex y Bison desde el repositorio `faturita/YetAnotherCompilerClass` [↗](#). Para quienes deseen utilizar LLVM, hay un repositorio de ejemplo en `faturita/LlvmCompiler` [↗](#). También es posible, desde el siguiente [enlace](#) [↗](#), obtener información adicional acerca de ideas para realizar en el proyecto, y descripciones de ciertos conceptos clave asociados.
- **Clases anteriores:** Se encuentran disponibles en YouTube, grabaciones de las clases anteriores acerca de los temas relacionados a este proyecto y a sus conceptos teóricos y prácticos, tanto de los compiladores como de los analizadores sintácticos:
 - **Introducción a los Compiladores** [↗](#), uno de los primeros videos de la serie donde se provee una idea general de los conceptos referentes al proyecto.
 - **Introducción Rápida a Lex** [↗](#), donde se muestran numerosos ejemplos de uso de la herramienta Lex (la versión anterior, pero totalmente compatible con Flex). Todos los ejemplos son compilables de igual forma bajo Flex.
 - **Introducción a Yacc** [↗](#), donde, al igual que para Lex, se proveen múltiples casos de uso de ejemplo sobre la herramienta Yacc, el predecesor de Bison. Nuevamente, todos los ejemplos se pueden portar sin modificaciones hacia la nueva herramienta sin inconvenientes.
 - **Ejemplos de Programas de Lex y Yacc** [↗](#). Más ejemplos prácticos de ambas herramientas.
 - **Arquitectura de Compiladores** [↗](#), con una descripción básica de la estructura de un compilador, y la complejidad asociada a la construcción de los mismos.
 - **Lenguajes y Analizadores Sintácticos Descendentes $LL(1)$** [↗](#).
 - **Lenguajes y Analizadores Sintácticos Ascendentes $LR(1)$ y $LALR(1)$** [↗](#); en particular, Bison es un analizador de tipo $LALR(1)$.
 - **Máquinas de Turing** [↗](#). Las máquinas de Turing representan los autómatas que describen todos los lenguajes de *tipo-0*, según la jerarquía de Chomsky.
 - **Gramática con Atributos** [↗](#), uno de los últimos temas de la materia, pero muy relacionado con las acciones asociadas a cada patrón regular y regla de producción en la especificación de Flex y Bison.
 - **Ejercicios de Gramáticas con Atributos** [↗](#), una continuación del video anterior.

Las presentaciones utilizadas y las guías de ejercicios se encontrarán disponibles en el campus de la cátedra.

- **Flex/Bison:** La bibliografía contiene varios libros acerca del diseño y desarrollo de compiladores, pero en particular, el libro de Levine (2009), posee un paso a paso completo del uso práctico de estas herramientas, y es clave para desarrollar el trabajo práctico, o para resolver ciertas dudas acerca de su implementación.

- **El Libro del Dragón:** Es el libro más conocido sobre compiladores, y su nombre se debe a la portada del mismo. Fue escrito por Aho et al. (2006). Los primeros capítulos del libro son clave para comprender el funcionamiento y la implementación del compilador, y de cada una de sus fases. Los últimos capítulos del libro se dedican a la generación de código y a las fases de optimización, conceptos avanzados que escapan al proyecto en sí, pero que son críticos en los compiladores modernos.

BIBLIOGRAFÍA

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques and Tools* (2.^a ed.). Addison—Wesley.
- Cooper, K. D., & Torczon, L. (2011). *Engineering a Compiler* (2.^a ed.). Elsevier.
- Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2007). *Introduction to Automata Theory, Languages and Computation* (3.^a ed.). Addison—Wesley.
- Hopcroft, J. E., & Ullman, J. D. (1969). *Formal Languages and their Relation to Automata* (1.^a ed.). Addison—Wesley.
- Levine, J. R. (2009). *flex & bison* (1.^a ed.). O'Reilly Media, Inc.
- Sipser, M. (2012). *Introduction to the Theory of Computation* (3.^a ed.). CENGAGE Learning.