UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

TFG del Grado en Ingeniería Informática

Diseño e implementación de un lenguaje de programación y su compilador

Presentado por Adrián Zamora Sánchez
en Universidad de Burgos — 4 de diciembre de 2025
Tutor: Dr. César Ignacio García Osorio

UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

D. nombre tutor, profesor del departamento de nombre departamento, área de Lenguajes y Sistemas Informáticos.

Expone:

Que el alumno D. Adrián Zamora Sánchez, con DNI 48838775T, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado título de TFG.

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección del que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 4 de diciembre de 2025

Vº. Bº. del Tutor:

César Ignacio García Osorio

Resumen

En este proyecto se aborda la creación de un lenguaje de programación, herramienta fundamental dentro del campo de la ingeniería informática. Para que este lenguaje sea utilizable, se implemnetará un compilador propio, capaz de llevar este lenguaje a ejecución. Este proyecto se limitará a algunas funcionalidades clave, puesto que querer abordar cuestiones como OOP podrían aumentar la complejidad del proyecto demasiado.

Para la realización del proyecto se tratan de utilizar las herramientas más modernas y extendidas en la creación de lenguajes y compiladores: ANTLR, LLVM, CMake y C++ como lenguaje de programación.

Descriptores

Compilador, lenguaje de programación, ANTLR, LLVM

Abstract

This project focuses on the creation of a programming language, a fundamental tool within the field of computer engineering. To make this language usable, a custom compiler will be implemented, capable of executing programs written in it. The scope of the project will be limited to a set of key functionalities, as tackling more advanced features such as OOP could increase the complexity beyond what is feasible.

For the development of the project, the most modern and widely adopted tools in language and compiler design will be used: ANTLR, LLVM, CMake, and C++ as the programming language.

Keywords

Compiler, programmin language, ANTLR, LLVM

Índice general

Índice general iii

Índice de figuras iv

Índice de tablas v

1. Introducción 1

2. Objetivos del proyecto 3

3. Conceptos teóricos 5

4. Técnicas y herramientas 9
   4.1. Metodología de desarrollo 9
   4.2. Metodología de programación 10
   4.3. Elección de herramientas 10

5. Aspectos relevantes del proyecto 15

6. Trabajos relacionados 23

7. Conclusiones y Líneas de trabajo futuras 25

Bibliografía 27

Índice de figuras

5.1. Ejemplo de expresión 2 + 3 * 5 18

Índice de tablas

5.1. Correspondencia entre tipos del AST y tipos de LLVM IR . . . 20

1. Introducción

En la informática moderna estamos acostumbrados a disponer de numerosos lenguajes de programación, los cuales empleamos sin detenernos a analizar más allá de sus potenciales usos y librerías disponibles. Sin embargo, estas complejas herramientas cuentan con un gran trabajo detrás, el cual permite a los programadores utilizar estas herramientas sin preocuparse de cómo funcionan realmente.

La creación de los compiladores es una ciencia que ha evolucionado mucho a lo largo de los años, no solo por el surgimiento de nuevos lenguajes y sus compiladores asociados, muchos lenguajes que llevan con nosotros desde los 70s han experimentado cambios significativos en sus compiladores, además de versiones alternativas, cada una enfocanda en unos objetivos diferentes.

Ejemplos de proyectos que comenzaron siendo pequeños lenguajes de programación o lenguajes de dominio concreto que han crecido hasta ser herramientas ampliamente utilizadas son:

- **HTML + CSS**: Nacieron como lenguajes muy específicos que hoy día son estándares globales, tienen un ecosistema inmenso, disponen de importantes frameworks e incluso han inspirado a lenguajes y herramientas fuera del entorno web.
- **MATLAB**: Diseñado en los 80s para ser un lenguaje específico de algebra lineal, que hoy día se utiliza ampliamente en ingeniería, robótica, procesamiento de señales, visión por computadora, etc.
- **SQL**: Comenzó en los 70s como un lenguaje que IBM desarrollaría para hacer consultas en sus bases de datos relacionales, sin embargo,

fue ampliamente adoptado como un estándar que lo ha llevado a estar presente en prácticamente todos los motores de bases de datos relaciones modernos.

Memoria

La memoria ofrece una visión completa del proyecto, incluyendo la definición de objetivos, el marco teórico necesario, las técnicas y herramientas utilizadas, así como los aspectos más relevantes del desarrollo. También analiza trabajos relacionados y presenta las conclusiones obtenidas, así como posibles líneas de trabajo futuras donde se sugieren posibles mejoras o ampliaciones del trabajo realizado.

Anexos

Los anexos complementan la memoria proporcionando documentación adicional que detalla el proyecto en algunos aspectos, como el plan de trabajo, los requisitos, el diseño del sistema, manuales de programación y usuario, y la justificación de su valor educativo y curricular.

2. Objetivos del proyecto

- **Diseñar un lenguaje de programación**: Definir la sintaxis y semántica de un nuevo lenguaje, estableciendo sus estructuras básicas, tipos de datos, operadores y reglas gramaticales.
- **Desarrollar un compilador capaz de reconocer el lenguaje**: Implementar un compilador con un diseño modular y moderno, capaz de analizar y procesar el código fuente del lenguaje, abarcando las fases de análisis léxico, sintáctico, semántico, generación de IR, optimización y finalmente generación de un ejecutable.
- **Estructurar de forma modular el proceso de compilación**: Separar las diferentes fases de compilación manteniendo las responsabilidades bien definidas dentro de cada una de las fases, evitando así una sobrecarga de responsabilidades que dificulte realizar ampliaciones o modificaciones.
- **Generación y gestión de errores**: Incorporar un sistema de detección y reporte de errores léxicos, sintácticos y semánticos, proporcionando mensajes claros y detallados que faciliten la depuración del código.
- **Herramienta de análisis y depuración visual**: Desarrollar una utilidad visual que permita representar y explorar estructuras internas del compilador, como el Árbol de Sintaxis Abstracta (AST), facilitando la comprensión y depuración del lenguaje.
- **Generación de una representación intermedia (IR)**: Implementar la traducción del lenguaje diseñado a una representación intermedia basada en LLVM IR, permitiendo la optimización y posterior generación de código máquina para distintas arquitecturas.

- **Implementación de utilidades internas del lenguaje**: Desarrollar un conjunto de funciones o bibliotecas básicas que proporcionen utilidades estándar dentro del entorno del lenguaje diseñado.
- **Proporcionar un runtime personalizado**: Para poder implementar las funcionalidades específicas del lenguaje, se requiere de un runtime personalizado capaz soportar estas características.

Objetivos técnicos

El desarrollo del proyecto se apoya en el uso de herramientas y tecnologías ampliamente reconocidas en el ámbito de los compiladores modernos:

- **ANTLR**: Se utiliza ANTLR como generador de analizadores léxicos y sintácticos, dada su modernidad, robustez y amplia adopción en el desarrollo de lenguajes formales.
- **LLVM**: Se emplea LLVM como infraestructura de compilación, debido a su capacidad para generar una representación intermedia (IR) portable y optimizable, independiente de la arquitectura de hardware.
- **C++**: El proyecto se desarrolla con el lenguaje C++, por su excelente integración tanto con ANTLR (a través del ANTLR C++ Runtime) como con LLVM, además de sus ventajas en programación orientada a objetos y gestión eficiente de memoria.
- **LaTex forest**: Esta herramienta ampliamente utilizada en el ámbio académico se utiliza para visualizaciones y diagramas.
- **Cmake**: Una herramienta como CMake hace que el proceso de compilación y construcción del proyecto sea sencillo, fácilmente replicable y personalizable.

3. Conceptos teóricos

Sobre lenguajes de programación

Lenguaje de programación : Un lenguaje de programación es todo aquel lenguaje que permite escribir instrucciones las cuales, tras compilarse, pueden ser ejecutadas en un sistema informático.

Compilación : Es el proceso por el cual un lenguaje de programación se convierte en un programa ejecutable por un computador.

Clasificación de lenguajes : Los lenguajes de programación suelen clasificarse según su nivel de abstracción (alto o bajo nivel), paradigma (imperativo, declarativo, OOP, funcional), propósito del lenguaje (propósito general, lenguaje específico del dominio) y por forma de ejecución (compilados o interpretados).

Tipado : El tipado de un lenguaje se refiere a la forma en la que se gestionan los tipos de los datos, en los lenguajes con tipado estático, los tipos deben ser especificados en las declaraciones y se hacen comprobaciones estrictas durante las asignaciones, por otro lado, los lenguajes de tipado dinámico permiten declaraciones sin especificar el tipo, en estos lenguajes normalmente se hacen transformaciones de datos en tiempo de ejecución.

Lenguaje máquina : El lenguaje máquina es aquel que puede ser ejecutado por un ordenador, este tipo de lenguaje suele ser generado por un compilador, puesto que sería extremadamente ineficiente de escribir por un ser humano.

Scope : Un alcance o scope se refiere normalmente a la accesibilidad a elementos como variables o funciones desde diferentes partes del código. Por ejemplo, una variable definida dentro de una función solo será accesible desde el bloque interno de la función pero no será accesible desde el bloque de código donde se ha definido la propia función.

Sobre compiladores

Compilador por fases : Es una técnica de construcción de compiladores que destaca por su alta modularidad, cada fase está bien definida y tiene una única responsabilidad, se caracterizan por ser fácilmente modificables y ampliables.

Análisis léxico : Es una fase del proceso de compilación en la cual se analizan las secuencias de caracteres de un texto y se separan en tokens o lexemas. A la herramienta que realiza este proceso se le llama scaner, lexer o tokenizer

Análsis sintáctico : Esta fase del proceso de compilación trata de agrupar los tokens obtenidos durante el análisis léxico, para ello se vale de reglas que generan otras estructuras de datos (como AST) desde la cadena de tokens.

AST : El Abstract Syntax Tree (árbol de sintaxis abstracta), es una estructura de datos que se utiliza comúnmente en compiladores para representar producciones sintácticas, donde los tokens forman una estructura gerárquica de árbol que representa el programa y ayuda a su posterior interpretación.

Tabla de símbolos : Es una estructura de datos utilizada en compiladores para asociar cada símbolo de un programa con su ubicación, alcance y tipo de dato. Dentro de un compilador cumple un papel fundamental entre el front end y el back end del compilador.

Análisis semántico : Es una fase en la cual el compilador comprueba la corrección de las producciones válidas formadas en el análisis sintáctico, algunas correcciones podrían ser la verificación de tipos y corrección en asignaciones y expresiones. Normalmente en esta fase se separa la información necesaria para rellenar la tabla de símbolos con los datos de los identificadores presentes en el código.

Generación de código intermedio : La creación de una representación intermedia o IR por sus siglas en inglés, hace referencia a un código

que queda a medio camino entre el código fuente y el código máquina. Especialmente en el caso del IR de LLVM utilizado en este trabajo, esta fase nos permite tener una mejor base para realizar diferentes optimizaciones y conversiones a diferentes formas de lenguaje máquina, según la arquitectura objetivo.

Optimización : Esta fase consisten en realizar cambios o mejoras en la forma del lenguaje, sin alterar el significado original. Estas pueden ser eliminaciones de código no alcanzable, redefiniciones de algunas estructuras para evitar generar más variables de las necesarias, gestión eficiente de la memoria, etc.

Generación de código final : Esta es la fase final del proceso de compilación, durante este paso se convierte el IR a código máquina, el cual está totalmente listo para ser ejecutado.

4. Técnicas y herramientas

Esta parte de la memoria tiene como objetivo presentar las técnicas metodológicas y las herramientas de desarrollo que se han utilizado para llevar a cabo el proyecto. Si se han estudiado diferentes alternativas de metodologías, herramientas, bibliotecas se puede hacer un resumen de los aspectos más destacados de cada alternativa, incluyendo comparativas entre las distintas opciones y una justificación de las elecciones realizadas. No se pretende que este apartado se convierta en un capítulo de un libro dedicado a cada una de las alternativas, sino comentar los aspectos más destacados de cada opción, con un repaso somero a los fundamentos esenciales y referencias bibliográficas para que el lector pueda ampliar su conocimiento sobre el tema.

4.1. Metodología de desarrollo

El desarrollo del proyecto se ha organizado siguiendo una metodología ágil, concretamente el marco de trabajo **SCRUM**, apoyado en la herramienta **ZenHub** para su gestión y seguimiento. La elección de una metodología ágil se justifica por su *flexibilidad* y *adaptabilidad*, características especialmente valiosas en un proyecto de investigación y desarrollo como este, donde la evolución del conocimiento y del propio software ocurre de forma simultánea e iterativa.

El flujo de trabajo se estructura mediante un tablero **Kanban** en ZenHub, el cual permite visualizar de manera clara el estado actual del proyecto. En dicho tablero se gestionan las *issues* y los *milestones* de GitHub, representando las distintas tareas, etapas o funcionalidades en desarrollo. Cada elemento del tablero incluye información relevante como la prioridad, la

duración estimada, el tipo de tarea y su relación con otras, lo que facilita la planificación y el control del progreso de manera visual y dinámica.

Esta metodología fomenta un desarrollo incremental y una mejora continua del código, permitiendo incorporar nuevas funcionalidades, corregir errores y ajustar los objetivos a medida que avanza el trabajo, garantizando así un resultado más coherente y alineado con los objetivos técnicos del proyecto.

4.2. Metodología de programación

El desarrollo del compilador y de las herramientas asociadas se ha abordado desde el paradigma de la **programación orientada a objetos** (OOP, por sus siglas en inglés). Este enfoque resulta especialmente adecuado para un proyecto de estas características, ya que facilita la modularización del código y la definición clara de las responsabilidades de cada componente del sistema (analizador léxico, analizador sintáctico, generador de código, etc.).

La OOP favorece la reutilización, extensibilidad y mantenibilidad del software, permitiendo aislar los distintos módulos del compilador y facilitar su evolución futura. Además, este paradigma se integra de forma natural con las herramientas empleadas, como **ANTLR** y **LLVM**, ambas diseñadas con arquitecturas orientadas a objetos.

Otro motivo determinante en la elección de este enfoque es el uso de **C++** como lenguaje de implementación. C++ ofrece un modelo de programación fuertemente orientado a objetos, combinado con una gestión eficiente de recursos y un rendimiento elevado, aspectos cruciales en el desarrollo de compiladores y sistemas de bajo nivel. La compatibilidad nativa de C++ con las bibliotecas de ANTLR y LLVM refuerza aún más la idoneidad de esta metodología para el proyecto.

4.3. Elección de herramientas

A continuación se presentan las principales herramientas empleadas en el desarrollo del proyecto, así como la justificación de su elección frente a otras alternativas disponibles. Las herramientas seleccionadas se han escogido teniendo en cuenta su madurez, soporte, documentación, integración con C++ y adecuación a los objetivos técnicos del proyecto.

ANTLR

ANTLR (Another Tool for Language Recognition) es una herramienta ampliamente utilizada para la generación automática de analizadores léxicos y sintácticos a partir de gramáticas formales. Su principal fortaleza radica en la facilidad con la que permite definir y mantener gramáticas complejas mediante una sintaxis clara y expresiva, generando código eficiente y legible para distintos lenguajes de programación, incluido C++.

Entre sus características más destacables se encuentran:

- Soporte nativo para gramáticas LL(*) que permiten manejar ambigüedades complejas.
- Generación automática de analizadores léxicos y sintácticos a partir de un único archivo de gramática.
- Aunque está diseñado para usarser con Java, actualmente cuenta con compatibilidad para muchos lenguajes de destino, entre ellos C++.

Alternativas consideradas:

- **Bison y Flex:** Son las herramientas clásicas en la generación de analizadores. Sin embargo, en el contexto elegido ANTLR cuenta con una estructura orientada a objetos más integrable con otras herramientas.
- **PEG (Parsing Expression Grammar):** Ofrecen gran expresividad y flexibilidad, pero carecen del soporte multiplataforma y de la madurez de ANTLR, además de no integrarse de forma fácil con C++.
- **JavaCC:** Alternativas orientadas a Java, no adecuadas para este proyecto al estar centrado en C++.

Justificación de la elección: ANTLR se considera la opción más equilibrada entre facilidad de uso, potencia y soporte técnico. Permite definir la gramática del lenguaje de manera declarativa y coherente, integrándose perfectamente con el entorno C++. Además, su ecosistema y documentación la convierten en una herramienta moderna y sólida frente a las opciones más tradicionales.

LLVM

LLVM (Low Level Virtual Machine) es una infraestructura modular y extensible para el desarrollo de compiladores, enlazadores y optimizadores de código. Su arquitectura se basa en una representación intermedia (IR) que actúa como un lenguaje ensamblador universal, lo que permite desarrollar compiladores portables, eficientes y escalables.

Entre sus principales ventajas destacan:

- **Representación intermedia (IR):** Permite una separación clara entre la fase de análisis y la de generación de código, haciendo posible optimizaciones y traducciones a múltiples arquitecturas.
- **Portabilidad:** LLVM IR puede compilarse a diferentes arquitecturas (x86, ARM, RISC-V, etc.) sin modificar el compilador fuente.
- **Integración con C++:** Al estar implementado en C++, se integra de manera natural con el entorno de desarrollo del proyecto.
- **Extensibilidad:** Su diseño modular permite hacer cambios como optimizaciones, cambio de backend o cambios en herramientas de análisis de forma sencilla.

Alternativas consideradas:

- **GCC (GNU Compiler Collection):** Ofrece una infraestructura madura, pero su API interna no está diseñada para un uso externo, lo que dificulta su integración en proyectos ajenos al entorno GNU.
- **QBE Compiler Infrastructure:** Es más ligera que LLVM, pero carece del ecosistema, documentación y soporte de comunidad que caracterizan a LLVM.
- **Cranelift:** Un backend moderno y rápido, pero enfocado en compilación JIT (just-in-time) y no tan versátil para la generación de IR o compiladores tradicionales.

Justificación de la elección: LLVM ofrece el equilibrio ideal entre potencial, documentación y flexibilidad, su diseño modular permite construir un compilador completamente funcional sin preocuparse por la dependencia de la arquitectura de destino. Además, su integración con C++ simplifica enormemente la generación de IR y la gestión de optimizaciones.

Entorno WSL

Justificación de la elección: El entorno Linux (conseguido mediente WSL) es especialmente sencillo para trabajar con las dependencias de este proyecto sin requerir capas de compatibilización adicionales requeridas por Windows (como MSYS2 o MinGW).

Sus principales ventajas frente a Windows:

- **Simpleza:** Con un entono únicamente de CLI permite un gran rendimiento y mantiene una sencillez coherente con el desarrollo de un compilador.
- **Shell scripting:** Permite crear rutinas de ejecución sencillas que ahorran repetir trabajo a la hora de compilar y probar código.
- **Integración:** Todos los paquetes están preparados para su uso en Unix-like, no se requiere ningún parche para compatibilizar las herramientas entre si.

Visual Studio Code

Justificación de la elección: La elección de Visual Studio Code como IDE se da por su simpleza y conocimiento previo, evitando tener que aprender sobre cómo usar un IDE más específico como Visual Studio, pero a su vez potente gracias a su sistema de extensiones, que permite modificarlo según las necesidades del desarrollo.

En mi caso lo he personalizado con las siguientes extensiones:

- **WSL:** Permite la integración con WSL como si se tratase del sistema host.
- **CMake y CMake Tools:** Permite ejecutar las rutinas de CMake creadas en el proyecto.
- **LaTex Workshop:** Integración con LaTex y comodidad de compilación al guardar los archivos para ver los cambios inmediatamente.
- **C/C++ Extension pack:** Agrega IntelliSense, linting, herramientas de debugging y code browsing para C y C++.
- **Clang-Format:** Herramienta configurable que permite establecer reglas de estilo consistentes en todos los archivos de código, en mi caso utilizo el estándar de LLVM.

5. Aspectos relevantes del proyecto

El comienzo del proyecto parte del objetivo de conseguir una versión MVP (Minimal Viable Product) que pudiese aportar una base sólida y fácil de ampliar. En este caso el MVP se enfocó en la creación de un micro compilador capaz de procesar expresiones aritméticas y lógicas, puesto que esta estructura es básica y está presente en todos los lenguajes de programación.

Antes de compilar

El compilador antes de comenzar el proceso de compilación debe recibir en su ejecución datos que hagan referencia al archivo que se debe compilar, que archivo de salida se quiere generar y de que forma se debe realizar el proceso, permitiendo así al programador ajustar este proceso a sus necesidades.

Dentro del compilador se esperan algunos argumentos de llamada como:

- **<nombre de la entrada>**: argumento obligatorio, indica el nombre del fichero .T de entrada.
- **-o <nombre archivo>**: argumento obligatorio que especificar el fichero ejecutable de salida.
- –**debug**: permite ver información de la salida de cada una de las fases del compilador.
- –**visualizeAST**: genera un fichero AST.pfd con una visualización del AST en el directorio donde se ha ejecutado el compilador.

- –**IR**: genera un fichero .ll con el LLVM IR generado por el compilador.

Para conocer los datos necesarios para ejecutar los comandos anteriores en el compilador se almacenan flags de compilado que permiten modificar el comportamiento de cada fase.

Proceso de compilación

Como el compilador se ha planificado como modular, a continuación se recorrerá el proceso de compilación completo, valiéndonos del contenido del MVP, es decir, un compilador capaz de reconocer expresiones ariméticas y lógicas.

Análisis léxico

La primera fase de este desarrollo comienza con la creación de un lexer donde se definen algunas estructuras básicas como los literales de números enteros, números con decimales o las cadenas de caracteres, así como los operadores que se van a emplear para estas expresiones.

En este momento se definen los tipos básicos del lenguaje:

```
TYPE_INT     : 'int'    ;
TYPE_FLOAT   : 'float'  ;
TYPE_CHAR    : 'char'   ;
TYPE_STRING  : 'string' ;
TYPE_BOOLEAN : 'bool'   ;
TYPE_VOID    : 'void'   ;
TYPE_PTR     : 'ptr'    ;
```

Se definen los operadores aritméticos y lógicos:

```
PLUS  : '+' ;
MINUS : '-' ;
MUL   : '*' ;
DIV   : '/' ;

INC   : '++' ;
DEC   : '--' ;

EQ : '==' ;
```

```
NE : '!=' ;
LT : '<'  ;
LE : '<=' ;
GT : '>'  ;
GE : '>=' ;
```

Análisis sintáctico

Una vez obtenido el lexer el siguiente paso es construir un parser capaz de reconocer estructuras gramaticales más complejas donde se conectan el uso de los operadores y los literales definidos anteriormente.

Los operadores y las operaciones sobre estas estructuras de definen en el lenguaje ANTLR4 de la siguiente forma:

```
expr
    : expr op=(MUL|DIV) expr        # arithmeticExpr
    | expr op=(PLUS|MINUS) expr     # arithmeticExpr
    | expr comparisonOperator expr  # logicalExpr
    | operand                       # operandExpr
    | LPAREN expr RPAREN            # parenExpr
    ;

comparisonOperator
    : EQ
    | NE
    | LT
    | LE
    | GT
    | GE
    ;

operand
    : literal
    ;
```

Cabe destacar el uso de etiquetas (como parenExpr) que falicitan posteriormente acceder a cada tipo de expresión gracias a la separación en contextos que realzia ANTLR4.

A continuación se debe implementar una estructura de datos que sea capaz de representar cualquier tipo de estructura sintáctica bien formada

de este lenguaje, la estructura del árbol de sintaxis abstracta (AST por sus siglas en inglés) será la encargada de crear este representacion en tiempo de ejecución del compilador, almacenando los datos relevantes de cada estructura para su posterior análisis en las siguientes fases.

Para formar este AST se empleará un visitor patterns (patrón de visita), objeto que recibe el nombre de ASTBuilder, el cual se encarga de visitar cada contexto generado de forma automática por el parser de ANTLR, dando por resultado unos nodos enlazados, el AST.

Adicionalmente, si la flag –visualizeAST está activada, el compilador puede generar una representación visual del árbol tras su construcción, para conseguir esto se escriben cada nodo en el formato de la biblioteca 'forest' de LaTex, permitiendo el análisis de la estructura del programa así como datos relacionados con los componentes, como los tipos u operadores.

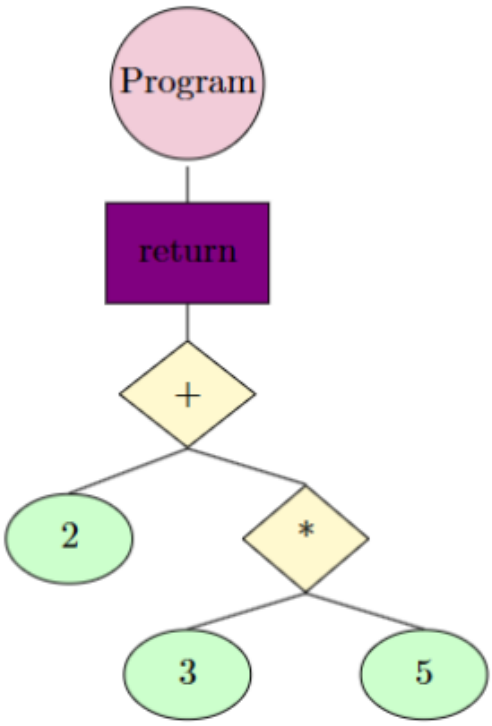

Figura 5.1: Ejemplo de expresión 2 + 3 * 5

Análisis semántico

Durante la fase de análisis semántico se examina principalmente el uso de los identificadores, como los nombres de variables y funciones. Entre las comprobaciones más relevantes se incluyen:

- La creación de un scope para cada bloque de código.
- La inserción de los identificadores en su scope correspondiente.
- La verificación del alcance en cada acceso a un identificador.
- Las comprobaciones de tipos en asignaciones, operaciones y sentencias de retorno.

La tabla de símbolos está estructurada como una lista de scopes. Para los distintos bloques anidados se establecen referencias entre cada scope hijo y su scope padre, lo que permite verificar fácilmente los alcances y resolver identificadores de forma correcta.

Para poder comprobar la corrección del uso de tipos, el análisis semántico debe ser capaz de entender los tipos, tanto de los literales como de las variables y funciones, para ello se implementa un sistema de representación de tipos en una enumeración llamada SupportedTypes que da soporte a los tipos:

- TYPE_INT
- TYPE_FLOAT
- TYPE_CHAR
- TYPE_STRING
- TYPE_BOOLEAN
- TYPE_VOID
- TYPE_PTR

Generación de LLVM IR

Para la generación de IR es emplea un struct que almacena los datos necesarios para que LLVM pueda generar el IR de forma automática, para ello se inicializan dentro del CodegenContext los siguientes tres elementos:

- Un módulo de LLVM
- Un contexto de LLVM
- Un builder de LLVM

Durante el proceso de generación de código IR, se vuelve a recorer el AST con un visitor pattern, el cual emplea principalmente el builder para generar las instrucciones a partir de los datos contenidos en los nodos del AST.

Algunas de las consideraciones necesarias para compatibilizar el AST y el generación de IR serían los tipos que se manejan en cada uno, para poder hacer una conversión correcta se implementa una función capaz de transformar:

Tipo AST	**Tipo LLVM**
TYPE_INT	i32
TYPE_FLOAT	float
TYPE_CHAR	i8
TYPE_STRING	i8*
TYPE_BOOLEAN	i1
TYPE_VOID	void
TYPE_PTR	ptrType

Tabla 5.1: Correspondencia entre tipos del AST y tipos de LLVM IR

Optimizaciones

Para asegurar la calidad del código final, se aplican sobre el LLVM IR algunas optimizaciones incluidas en la API de LLVM, las cuales aplican transformaciones, inserciones/eliminaciones de codigo o correcciones mediante passes que recorren todo el programa generado.

Algunas de las más importantes son:

- Dead code elimination: Aunque se trate de aplicar durante la generación del código, es importante utilizarla para casos donde el código está bien formado pero hay elementos sin usar, como variables o funciones que no sean externas y nunca son llamadas en el programa.
- Dead store elimination: Se eliminan escrituras en memoria que nunca son leidas, ahorrando recursos internos.
- Dead argument elimination: Esta optimización específica de las funciones permite eliminar los argumentos no utilizados, simplificando el programa.
-

El uso de estas combinaciones junto a las comprobaciones de calidad mínimas para compilar sin errores, generan un código de una calidad similar al código optimizado con -O2 de compiladores como Clang, Rustc o Swiftc

Generación del ejecutable

La generación de un programa requiere conocer el "target triplet", es decir, la familia de CPU en la que se ejecuta el sistema. La API del sistema (llvm::sys) de LLVM contiene funciones que nos permiten conocer el triplet y así poder establecer un data layout correcto para el sistema en el que estamos trabajando.

Una vez hemos generado el código objeto del programa, el compilador ejecutra un enlace utilizando la herramienta clang/clang++, de forma que el archivo generado se enlaza con la librería estándar del compilador, la librería estándar de C y el runtime.

Runtime

El runtime es una parte necesaria para la ejecución de cualquier programa, normalmente aporta un punto de entrada valido mediante un system call en ensamblador.

Además aporta un entorno de ejecución correcto y personalizado para poder llevar a cabo acciones concretas, en el caso de este lenguaje, es el runtime el responsable de tener en cuenta los eventos generados para su planificación y ejecución.

6. Trabajos relacionados

En el aspecto más teórico destaca el Libro del Dragon de Aho, Sethi Ullman, el cual es un referente en cuanto a la teoría de lenguajes, lexers, parsers, construcciones de AST, analizadores semánticos, generación de IR y optimizaciones. Aunque es un material muy completo, su extensión me ha llevado a consultar apartados concretos y resumenes enfocados en ciertos aspectos relevantes para mi proyecto.

Destaca como proyecto de alcance y complejidad similar el tutorail/lenguaje didactivo llamado Kleidoscope el cual es parte del tutorial oficial de LLVM sobre compiladores. El proyecto Kleidoscope permite comprender de forma práctica y ordenada los conceptos de lexing, parsing, generación de AST y, especialmente, la generación de código intermedio mediante LLVM, sirviendo como una referencia clara para la construcción de compiladores modernos.

El compilador de Rust pese a su extensión y complejidad está excelentemente documentado, lo cual permite comprender sin demasiado esfuerzo como está estructurado y su funcionamiento interno. Este recurso me ha permitido contemplar como funciona un verdadero compilador del más alto nivel.

7. Conclusiones y Líneas de trabajo futuras

Conclusiones

Tras realizar el proyecto, he conseguido una comprensión profunda sobre como funcionan los lenguajes de programación así como los procesos de compilación. He lanzado una mirada crítica a herramientas y lenguajes que llevo usando desde hace muchos años pero nunca me había preguntado hasta tal punto como operaban fuera de la mirada del programador.

Tras comprender el trabajo tan complejo que supone crear un compilador de la calidad de GCC o interpretes como javac, me siento muy agradecido de tener acceso a tecnologías tan punteras y sofisticadas que nos permiten desarrollar de forma tan cómoda nuestra labor en el desarrollo de software.

Lineas de trabajo futuras

Puesto que el proyecto está extremadamente acotado por su complejidad y el tiempo disponible, considero que hay numerosas lineas de trabajo con las que me gustaría continuar proximamente, algunas de las mas destacables:

- Migrar de ANTLR a un lexer y parsers propios. Pese a que ANTLR4 tiene una calidad notable, la mayoría de grandes compiladores actuales proporcionan sus propios lexers y parsers, puesto que son mucho más optimizables y personalizables. - Ampliar el lenguaje con POO, una de las metodologías de la programación más populares y de las que mejor encajan con un lenguaje que pretende modelar eventos. - Un runtime más sofisticado, con mayor precisión en el control del tiempo y rutinas más optimizadas hechas directamente en ensamblador.

[1]

Bibliografía

[1] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.