# Proyecto Final de Compilación: Diseño e Implementación de un Compilador para el Lenguaje HULK

Claudia Hernández Pérez, Joel Aparicio Tamayo, and Kendry J. Del Pino Barbosa

Facultad de Matemática y Computación, Universidad de La Habana

**Resumen** Este documento describe el diseño e implementación de un compilador para HULK en el lenguaje C. El diseño del compilador agrupa 4 partes: *lexer*, *parser*, *chequeo semántico* y *generación de código*.
El *lexer* y el *parser* se basan en los frameworks proporcionados por Lex y Yacc, respectivamente, aunque el proyecto cuenta con una implementación propia de un *lexer* basado en la teoría de autómatas finitos. El *chequeo semántico* es responsable de validar las reglas del lenguaje, reduciendo los errores en tiempo de ejecución. Para ello se ha implementado el patrón Visitor, que permite recorrer el árbol de sintaxis abstracta (AST) y aplicar las reglas semánticas correspondientes. Además, cuenta con una arquitectura que permite la unificación de expresiones con tipos, lo que facilita la inferencia. Finalmente, la *generación de código* se realiza mediante un generador de código intermedio que también implementa el patrón Visitor para traducir el AST a código LLVM.
Al ejecutar el compilador de HULK, se genera un `output.ll` en la carpeta *build*, donde se encuentra el código generado, y luego se invoca al compilador de LLVM para ejecutar dicho archivo, y devolver la salida del programa.

**Keywords:** Compilador · Lexer · Parser · Chequeo Semántico · AST · Scope · Generación de código · LLVM · Inferencia · Expresión · Tipo

## 1. Introducción

Los compiladores son herramientas fundamentales en la ciencia computacional, actuando como puentes entre abstracciones de alto nivel y ejecución eficiente. Específicamente, HULK (Havana University Language for Kompilers), es un lenguaje de programación didáctico, seguro de tipos, orientado a objetos e incremental. Este proyecto tiene la finalidad de compilar un subconjunto de dicho lenguaje, siendo consistente con las especificaciones en su definición formal, aunque añadiendo algunas extensiones sintácticas propias.

Para compilar HULK, se ha diseñado un flujo de trabajo sencillo y por capas. El punto de entrada del programa es el archivo `main.c`, que intenta leer un archivo `script.hulk` en formato de cadena de texto y luego invoca al *lexer* y

al *parser* para generar un árbol de sintaxis abstracta (AST). Posteriormente, si no hubo errores léxicos ni sintácticos, se realiza el *chequeo semántico* a partir del nodo raíz del AST. Finalmente, si no hubo errores semánticos, se procede a la *generación de código* intermedio, que se traduce a código LLVM. En este punto concluye el proceso de compilación, generando un archivo `output.ll` en la carpeta *build*. Para cada proceso del programa (compilación, ejecución, limpieza) existe una receta en el `Makefile` que lo ejecuta:

```
1 make compile // para compilar y generar el build/outout.ll
2 make execute // para ejecutar el compilador de LLVM
3 make clean // para limpiar el directorio de compilacion
```

A continuacón se detallan los disitntos aspectos de la implementación.

## 2. Lexer y parser

**Figura 1.** Arquitectura modular del compilador

## 3. Análisis semántico

Como se ha mencionado anteriormente, la entrada del *chequeo semántico* es el nodo raíz del AST, que es donde comienzan los chequeos, utilizando el patrón Visitor. La implementación de este patrón (ver Figura 2) contiene:

- **Funciones de visita para cada tipo de nodo del AST**: Para cada nodo hay una función que sabe analizarlo y aplicar las reglas semánticas correspondientes.
- **Lista de errores semánticos encontrados**: Se mantiene una lista de errores semánticos encontrados durante el análisis, que se reportan al usuario al finalizar el chequeo.
- **Cantidad de errores semánticos encontrados**: Longitud de la lista de errores encontrados (recordar que en C no se puede obtener el tamaño de un array dinámico, por lo que se debe mantener un contador)
- **Recolector de contextos**: Se encarga de recolectar todas las declaraciones de funciones y tipos en el contexto actual, antes de analizarlas
- **Nombre de la función actual**: Se mantiene el nombre de la función que se está analizando actualmente, para evitar tener que subir de nuevo en el AST cuando se necesite, como por ejemplo al usar: *base(...)*
- **Tipo actual**: Similar al caso anterior e igualmente necesario para el ejemplo brindado, pues en el caso del uso de *base(...)* se debe saber el tipo de la clase actual para poder buscar el método en algún ancestro.

**Figura 2.** Implementación del patrón Visitor para el chequeo semántico

El inicio del análisis se encuentra en el archivo `semantic.c`, donde primero se inicializa el visitor y luego se visita al nodo raíz del AST. La implementación de la función de visita de ese nodo, primero realiza las declaraciones de tipos y funciones builtin del lenguaje, luego recoge el contexto del programa y finalmente visita cada uno de sus descendientes. Al concluir el análisis, se imprimen en consola (con fuente en color Rojo) los errores reportados, y continúa el flujo del compilador. Para el chequeo semántico, cada nodo contará con un tipo de retorno, un ámbito (scope) y un contexto asociados (cada campo cumple una función específica y serán analizados posteriormente), y en caso de no especificar lo contrario, se asumirá de aquí en adelante que en cada visita el ámbito y contexto padres son los del nodo padre (NULL en caso de la raíz).

Ahora se explicará detalladamente cada uno de los chequeos que se realizan a cada nodo.

### 3.1. Análisis de expresiones básicas

En la implementación, se consideran como expresiones básicas a los literales (números, cadenas, booleanos), operaciones unarias (!, -), operaciones binarias ($+, -, *, /, \%, \hat{}, \&, |, ==, !=, <, >, <=, >=, @, @@$), así como los bloques de expresiones.

1. **Literales**: En el caso de los números y booleanos, las funciones de visita están vacías, pues no hay nada que chequear; sin embargo, en el caso de las cadenas, se verifica que los caracteres de escape utilizados (si los hay) sean correctos.
2. **Operaciones unarias**: Primero se visita al nodo de la expresión hija y luego se chequea la compatibilidad del tipo de dicha expresión con la operación unaria dada. (La compatibilidad de tipos con operadores se explicará más adelante). Luego se actualiza el tipo de retorno del nodo según el operador.
3. **Operaciones binarias**: Similiar a las operaciones unarias, se visitan primeros los nodos de las expresiones con las que se quiere operar y luego se chequea la compatibilidad de ambos tipos con el operador binario dado (La compatibilidad de tipos con operadores se explicará más adelante). Luego se actualiza el tipo de retorno del nodo según el operador.
4. **Bloques de expresiones**: Similar al nodo raíz, el nodo bloque funciona como un mini programa, por lo cual se recolecta primero su contexto y luego se visita cada uno de sus descendientes, manteniendo siempre el último visitado para poder al final actualizar el tipo de retorno del nodo bloque con el tipo de su último descendiente. Si el bloque esta vacío, automáticamente se le otorga tipo de retorno `Void`.

**Compatibilidad de tipos con operadores**: En el archivo `type.c` se definen todas las reglas de compatibilidad de tipos. En el arreglo `operator_rules`

(ver Figura 3) se almacenan las reglas para los operadores de la forma {`tipo de la izquierda, tipo de la derecha, tipo de retorno, operador`}. En caso que un operador pueda utilizarse con varios tipos, como es el caso del '==' por ejemplo, aparece una regla por cada uso del operador. De esta forma, es más sencillo revisar la compatibilidad de tipos con operadores, pues solo se tienen que recorrer las reglas y comparar los tipos izquierdos y derechos (NULL para los unarios) según el operador.

**Figura 3.** Reglas para los operadores

**Errores**: De manera general y como se expuso con anterioridad, en las expresiones básicas los errores más comunes son los respectivos a secuencias de escapes inválidas en cadenas e incompatibilidad con los operadores (ver ejemplo Figura 4).

**Figura 4.** Errores comunes en expresiones básicas

### 3.2. Análisis de variables y ámbitos (scopes)

La revisión de variables, en general, contempla el chequeo de asignaciones, el uso de variables y las expresiones *let-in*. Para comprender mejor esta sección, se comenzará explicando la estructura de un *scope* y las posibles acciones que se pueden realizar sobre él.

Un **scope**, en la implementación dada, es una estructura que contiene, a grandes rasgos, una tabla de símbolos, una tabla de funciones y una tabla de tipos definidos, las cuales actúan como bases de datos para las respectivas definciones en los distintos ámbitos del programa, además de una referencia al *scope* padre. Sobre los *scopes* se pueden realizar diversas acciones, iniciando claramente por su creación y finalizando con su destrucción al concluir el chequeo semántico (en C la liberación de memoria es manual). Esas acciones incluyen:

- Guardar símbolos, funciones o tipos en las tablas.
- Buscar específicamente algún símbolo, función o tipo definidos.
- Inicializar tipos y funciones *builtin*.
- Liberar memoria para cada tabla.

Luego de entender la estructura de los *scopes*, se abordarán los distintos tipos de análisis realizados en el caso de las variables:

1. **Asignaciones**: La revisión comienza chequeando que el nombre dado a la variable no sea un *keyword*, de lo contrario se reporta un error. También

con respecto a la variable, se verifica si fue tipada o no, y en caso que sí, se analiza que el tipo sea válido. Luego, se pasa al chequeo de la parte derecha de la asignación, invocando su función de visita. Una vez termina de visitar la expresión de la derecha, se revisa la compatibilidad del tipo de dicha expresión con el tipo impuesto a la variable (en caso de haber sido tipada, en otro caso no sucede nada) y se guarda en dependencia del operador de asignación. Si se utiliza `==` se crea un nuevo símbolo en el *scope* padre, y en caso del operador destructivo `:=`, se busca el símbolo en la tabla de símbolos (de algún *scope* ancestro) y se verifica que sean compatibles los tipos de la inicialización y la reasignación. En caso que no, o que no se haya encontrado el símbolo se reportará error. En el caso del uso de asignación destructiva el tipo de retorno del nodo es el mismo que el de la variable asignada (la cual toma el tipo de la expresión de la derecha si no fue tipada), y en caso contrario el retorno es `Void`.

2. **Uso de variables**: Esta revisión es más sencilla, pues basta con buscar la variable en algún *scope* ancestro y si no existe se reporta error.
3. **Let-in**: Esta expresión esta compuesta por un conjunto de asignaciones y un cuerpo. Por tanto, la revisión consta de visitar cada una de las asignaciones y luego visitar el cuerpo. El tipo de retorno es el mismo que el tipo de retorno del cuerpo.

**Errores**: Como se expuso anteriormente, los errores detectados en estas revisiones son: tipo no definido al tipar una variable, incompatibilidad de tipos en asignaciones (en el caso de las que son tipadas), reasignar una variable que no ha sido inicializada antes, intentar reasignar una variable con un tipo incompatible con el tipo de su inicialización, variable indefinida, entre otros. (ver ejemplo Figura 5).

**Figura 5.** Errores comunes en variables

### 3.3. Análisis de funciones

El análisis de funciones agrupa dos tipos de revisiones: chequeo de la declaración y chequeo del llamado. Como las funciones pueden declararse en cualquier sección del programa se necesita primero recoger el contexto de todas las declaraciones. Para eso se utiliza una estructura denominada *context*.

El *context*, al igual que el *scope* sirve como una base de datos, pero dedicada a guardar los nodos de las declaraciones de funciones (o tipos, que se verán más adelante). Sobre un *context* se puede:

- Guardar una declaración.
- Buscar una declaración.

Ahora se explicará cada una de las revisiones dichas anteriormente:

1. **Llamado a función**: Lo primero que se se hace es visitar cada uno de los argumentos. Luego se busca la función en el *context* y se revisa su declaración para que queden guardados sus datos en el *scope* (en caso de haber sido o estar en proceso de ser revisada, no sucede nada). Finalmente se chequea que la cantidad de argumentos y los tipos de los argumentos coincidan con la declaración de dicha función y se actualiza el tipo de retorno del nodo del llamado con el guardado en el *scope*.
2. **Declaración de función**: Al igual que con la asignación de variables, se comienza revisando que el nombre de la función no sea una *keyword*. Después se hace un pequeño chequeo para descartar que existan nombres de parámetros repetidos. Luego se pasa a declarar en el *scope* de la declaración cada uno de los parámetros para que sean accesibles en el cuerpo de la función, revisando primero que si son tipados, tengan un tipo válido. Al terminar con los parámetros, se chequea el cuerpo, comenzando por ver si la función tiene un tipo de retorno definido, y en ese caso igualmente revisar que sea un tipo válido y más adelante que sea compatible con el tipo inferido del cuerpo; y luego se llama a su función de visita.

**Errores**: Existen diversos tipos de errores que se detectan en estas revisiones: incompatibilidad de tipos en los parámetros, en el retorno, repetición de un mismo símbolo en los argumentos de una declaración, función no definida, cantidad de argumentos no compatible, entre otros (ver ejemplo Figura 6).

**Figura 6.** Errores comunes en funciones

## 4. Generación de Código

## Referencias

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley (2006)
2. Appel, A.W.: Modern Compiler Implementation in C. Cambridge University Press (1998)
3. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. CGO 2004, LNCS 3161, Springer (2004)