Compilador para el Lenguaje HULK

Adrián Hernández Castellanos, Laura Martir Beltrán, and Yesenia Valdés Rodríguez

Facultad de Matemática y Computación, Universidad de La Habana

Resumen Este documento presenta el diseño e implementación de un compilador para HULK usando como lenguaje de programación C++ y apoyándose en las herramientas Flex, Bison y LLVM.
Para la aplicación de los conocimientos adquiridos en la asignatura, se realizó la implementación en 4 etapas fundamentales que debe realizar un compilador: análisis léxico, análisis sintáctico, análisis semántico y generación de código. Cada etapa será explicada detalladamente.

1. Presentación del flujo

Este proyecto tiene la finalidad de compilar un conjunto de características que definen al lenguaje HULK, siguiendo las especificaciones que se pueden encontrar en el siguiente enlace. Para compilar HULK, se ha llevado a cabo la construcción del compilador siguiendo el siguiente flujo:
El archivo `hulk.cpp` representa el punto de entrada a nuestro programa, particularmente la función `main` que intenta leer un archivo `script.hulk` e invoca al *lexer* y al *parser* para generar un árbol de sintaxis abstracta (AST). Posteriormente, si no hubo errores durante estos procesos, se realiza el *análisis semántico*, el cual comienza analizando a partir del nodo raíz del AST. Finalmente, si no hubo errores semánticos, se procede a la *generación de código* intermedio, que traduce a código LLVM. En este punto concluye el proceso de compilación, generando un archivo `Hulk-IR.ll` en la carpeta `Hulk`. Posteriormente este archivo se puede compilar a un ejecutable y mostrarse en consola.

Para una ejecución sencilla de las funcionalidades del proyecto, existe una receta en el archivo `Makefile`:

```
1 make compile // Compila script.hulk, genera Hulk-IR.ll
2 make execute // Ejecuta el archivo compilado
3 make clean   // Limpia el proyecto compilado
```

2. Lexer

Para la correcta implementación del análisis léxico del compilador de HULK se realizó un estudio de la definición proporcionada. El lexer del compilador

utiliza como librería principal Flex, el cual permite realizar un análisis léxico profundo y robusto, facilitando el desarrollo de esta etapa. Dentro del mismo se definen las principales expresiones regulares que intervienen en el lengujae, así como la representación en string de cada token con su correspondiente valor de retorno. El archivo que implementa esta funcionalidad es llamado `lexer.l`, y es el encargado de transformar la entrada del programa (`script.hulk`) en una secuencia de tokens. En el programa, los tokens se definen en el archivo `lexer.l`, donde además se especifican las expresiones regulares para cada tipo de token.

Detección de errores: En el *lexer* se detectan todos los caracteres inválidos que se presenten en el código HULK propiciado, con especificación de línea, columna y caracter problemático.

3. Parser y AST

Para desarrollar el parser, encargado del análisis sintáctico, se realizó la implementación usando Bison, el cual al obtener la gramática, bien definida, crea un AST en función del conjunto de expresiones que se utilizaron en el programa Hulk. La definición de la gramatica se puede encontrar en el archivo `parser.y`. En dicha gramática se definen inicialmente los tipos de datos con los que se trabajará y se enlazan con el tipo de token que se espera recibir del análisis léxico. De esta manera el archivo `parser.y` es quien hace la llamada al análisis léxico y recibe los datos retornados para su posterior análisis sintáctico. Luego se definen producciones para cada tipo de *statement* o expresión de HULK, así como una precedencia para los operadores. Estas precedencias determinan una gramática seleccionada para nuestro parser, la cual está pensada para ser LR(1) según el orden de análisis de cada token y la generación sistemática del AST. Cada producción genera un nodo del AST, con lo cual se obtiene finalmente una lista de nodos.

```
params:
    /* nada */                      {$$ = new std::vector<Parameter>();}
    | ID                            {Parameter p; p.name = *$1; $$ = new std::vector<Parameter>(); $$->push_back(p);}
    | ID COLON ID                   {Parameter p; p.name = *$1; $$ = new std::vector<Parameter>(); $$->push_back(p);}
    | params ',' ID                 {Parameter p; p.name = *$3; $1->push_back(p); $$ = $1;}
    | params ',' ID COLON ID        {Parameter p; p.name = *$3; $1->push_back(p); $$ = $1;}
;

func_call_expr:
    ID '(' args ')'                 {$$ = new FuncCall(*$1, *$3, yylloc.first_line);}
;

args:
    /* nada */                      {$$ = new std::vector<ASTNode*>();}
    | expression                    {$$ = new std::vector<ASTNode*>(); $$->push_back($1);}
    | args ',' expression           {$1->push_back($3); $$ = $1;}
;

assign_expr:
    id REASSIGN expression          {$$ = new Assignment($1, $3, yylloc.first_line);}
    | self_call REASSIGN expression {$$ = new Assignment($1, $3, yylloc.first_line);}
;

let_expr:
    LET decl IN body                {$$ = new LetExpression($2, $4, yylloc.first_line);}
    | LET decl IN '(' body ')'      {$$ = new LetExpression($2, $5, yylloc.first_line);}
    | LET decl IN body ';'          {$$ = new LetExpression($2, $4, yylloc.first_line);}
    | LET decl IN '(' body ')' ';'  {$$ = new LetExpression($2, $5, yylloc.first_line);}
    | LET decl ',' let_expr         {$$ = new LetExpression($2, $4, yylloc.first_line);}
;
```

Figura 1. Ejemplo de creación de nodos del AST desde Bison

Existen varios tipos de nodos que se pueden crear en el parser, y cada uno representa una construcción del lenguaje. El tipo `ASTNode` se representa con una estructura que posee campos generales, de la cual los demás tipos de nodo heredan. Además, existe una función para cada tipo de nodo que permite inicializarlo. La declaración de estos tipos de nodos se encuentra en el archivo `AST.hpp`, donde se realiza una implementación del constructor de cada tipo de nodo, de un método *accept* que usaremos posteriormente para realizar el análisis semántico, de la línea en la que cada nodo fue declarado y del tipo de dato que representa cada nodo según su funcionalidad.

```
class ASTNode
{
public:
    virtual ~ASTNode() = default;
    virtual void accept(NodeVisitor &visitor) = 0;
    virtual int line() const = 0;
    virtual std::string type() const = 0;
};
```

Figura 2. Clase ASTNode

```
class VarDeclaration : public ASTNode
{
public:
    std::string name;
    std::string declaredType;
    ASTNode *initializer;
    bool isMutable;
    int _line;
    std::string _type;

    VarDeclaration(std::string name, std::string type, ASTNode *init, bool isMut, int ln)
        : name(name), declaredType(type), initializer(init), isMutable(isMut), _line(ln), _type("") {}

    void accept(NodeVisitor &visitor) override
    {
        visitor.visit(*this);
    }

    int line() const override { return _line; }
    std::string type() const override { return _type; }
};
```

Figura 3. Ejemplo de declaración de un nodo del AST

Detección de errores: En el *parser* se detectan varios errores en una sola pasada, pero teniendo en cuenta que un error de sintaxis pudiera provocar otros errores de sintaxis en otras partes del código, se recomienda comenzar por resolver el primer error sintáctico detectado.

4. Análisis semántico

Tras ejecutar todos los procedimientos mencionados anteriormente, tenemos un AST completamente conformado al cual procederemos a analizar semánticamente, basándonos en la definición de nodos de AST mostrada anteriormente. Se hace uso del Patrón Visitor para realizar esta tarea. Aquí es donde tiene importancia el método *accept* implementado anteriormente, ya que ejecuta el método visit de cada implementación realizada para la estructura NodeVisitor correspondiente. NodeVisitor es una estructura diseñada para asimilar los datos de cada nodo, en el orden correspondiente, y realizar validaciones de los mismos según la implementación que se le dé a sus métodos *visit*. A cada nodo del AST le corresponde una implementación diferente de visit, que indica cómo se realizará la lógica de revisión de cada nodo respecto a lo que querramos validar. En el contexto del análisis semántico tendremos particularmente 2 tipos de NodeVisitor, y cada uno realizará un recorrido distinto por el AST.
El primer recorrido tendrá como objetivo recolectar todas las definiciones de funciones realizadas a lo largo del código. Para esto se cuenta con la clase Function Collector, que se encargará de recopilar todas las funciones declaradas a lo largo del código para poder usarlas luego en cualquier parte del mismo. Al declarar estas funciones, pasan a estar definidas en un ámbito global en el cual previamente se agregaron todas las funciones Built-In de HULK, y al cual se puede acceder desde cualquier otro ámbito en el cual se está trabajando.
Posteriormente el segundo recorrido se realizará tanto para analizar definiciones de tipos y sus propiedades como para finalmente para validar el uso de las variables dentro de operaciones o funciones. Para esto se usa una segunda implementación de un node visitor, una clase llamada Semantic Validation cuya única función es implementar todos los métodos visit para cada nodo correspondiente, lo cual definirá exactamente cómo ocurrirá el proceso de validación semántica necesario.
Para realizar la salva de variables, funciones y tipos declarados durante la recisión del código, se cuenta con una estructura llamada Tabla de Símbolos, la cual contendrá Símbolos que representan declaraciones de objetos y sus tipos específicos, dividido en ámbitos locales y globales, para facilitar el proceso de validación semántica. La estructura mencionada además contiene implementaciones de creación y destrucción de ámbitos, búsqueda de declaraciones específicas, operaciones CRUD sobre estas declaraciones y otras funciones diseñadas para facilitar el proceso de visita de los nodos durante este análisis, donde el objetivo es mantener un registro de datos declarados con tipos de cada dato para mantener sistemáticamente una forma de comprobar cada nodo y cada operación realizada en código y validar o devolver un error en caso que sea requerido. Estos errores se expresan en la implementación visit de cada nodo en la estructura Semantic Validation.

4.1. Inferencia de tipos

El compilador intentará inferir los tipos de los datos a partir del uso de los parámetros en el cuerpo de la función. Este enfoque permite escribir funciones más flexibles, pero exige un sistema de inferencia y verificación de tipos robusto para detectar errores. Una vez inferido el tipo de un parámetro, este se fija y las llamadas posteriores deben respetar el tipo deducido, pues de lo contrario se devolverá un error al respecto. Para implementar este comportamiento, se utilizaron las estructuras de Símbolos y Tablas de Símbolos mencionadas anteriormente para mantener la consistencia sobre los tipos inferidos por cada nodo revisado:

- La estructura Symbol almacena las propiedades de cada símbolo (variable o función), como su tipo, si es constante, parámetros, cuerpo asociado, y un vector que indica qué parámetros han sido ya inferidos.
- La estructura SymbolTable representa los distintos contextos léxicos del programa y mantiene información jerárquica sobre todos los símbolos definidos en cada ámbito.

Gracias a estas estructuras y al módulo de validación semántica, el compilador puede detectar errores de tipo, realizar inferencia estática cuando es necesario, y rechazar llamadas a funciones mal tipadas.

4.2. Chequeo de tipos

El chequeo de tipos es otra forma de realizar análisis de tipado en HULK, el cual consiste en confirmar la definición explícita del tipo de dato a trabajar y verificación del correcto uso de datos de este tipo. Está representada por una implementación sencilla gracias a las estructuras mencionadas en los puntos anteriores, ya que solo se debe comprobar que el tipo especificado se comporte como el tipo esperado. La definición de comportarse como el tipo esperado no necesariamente requiere que sea específicamente de ese tipo, sino de cualquier tipo que herede del mismo. Todos los tipos son candidatos a que otros tipos hereden de él, excepto el tipo Object del cual todos heredan pero ninguno puede heredar de él explícitamente. Esto permite hacer llamadas y retornos que no necesariamente sean del tipo esperado, sin embargo permiten que se comporten como él ya que heredan su estructura y funcionalidades base.

5. Generación de Código

La última fase del compilador es la generación de código intermedio (LLVM-IR). En este fase, se utilizó el framework de C++ para LLVM, a través del cual se genera el archivo `Hulk-IR.ll`. La estrategia de generación de código utilizada sigue un enfoque de descomposición de expresiones en operaciones más simples.

En este punto, se implementó nuevamente el patrón Visitor con una nueva componente encargada de generar código LLVM a partir del análisis de cada

nodo del AST. De manera similar a la anterior, cada nodo del AST tiene un método **accept** que llama al método **visit** correspondiente en el Visitor (en este caso, el IRGenerator). Esto permite separar la estructura de los nodos del algoritmo de generación de código.

Para lograrlo, se implementó una estructura que permita mantener el contexto durante la generación, que gestiona los diferentes scopes mediante pilas. Esta estructura contiene los siguientes elementos:

- LLVM Context: Mantiene información global de LLVM.
- IR Builder: Facilita la creación de instrucciones LLVM.
- Métodos visit: Cada método es responsable de generar el código LLVM para un tipo específico de nodo

La clase Context es fundamental para gestionar el estado durante la generación de código. Esta presenta los siguientes elementos:

- IRGenerator: Clase que hereda de NodeVisitor e implementa un método visit para cada tipo de nodo AST.
- Contexto: Mantiene una referencia al contexto de generación que contiene el estado actual (módulo, builder, mapas de variables, etc.).
- Módulo: Contiene funciones y variables globales.
- localScopes: Pila de mapas (cada mapa es un scope) para variables locales. Permite herencia de scopes (para bloques internos) o scopes aislados (para funciones).
- functionScopes: Pila para declaraciones de funciones, permitiendo resolución en scopes anidados.
- Sistema de Tipos: Gestiona tipos definidos por el usuario, herencia, y atributos y métodos.
- También cuentan con funciones para el manejo de estos scopes, de manera similar a como ocurre con las componentes del análisis semántico.

Además, se realizó una integración de cada nodo con un sistema de tipos para manejar adecuadamente los tipos definidos por el usuario, manteniendo relaciones entre instancias y tipos y permitiendo la correcta manipulación del código para establecer características como herencia y polimorfismo. Posee varias estructuras clave:

- type_definition: Representa un tipo definido por el usuario.
- Atributos: Nombre, tipo padre (opcional), mapas de atributos y métodos.
- Parámetros del constructor y argumentos para la base (en herencia).
- typeTable: Mapa de nombres de tipos a sus definiciones.
- instanceTable: Mapa de instancias (nombre) a su tipo.
- instanceVars: Mapa de instancias a sus atributos (cada atributo se identifica por nombre y tipo real).

El flujo seguido es similar al mostrado durante el análisis semántico, reimplementando los métodos visit para cada nodo necesario. Este flujo sigue los siguientes pasos generales:

1. Inicialización: Crear módulo LLVM y configurar el builder. Declarar funciones intrínsecas (ej: printf, puts).
2. Recorrido del AST: Invocar Generate en el nodo raíz del AST, que inicia el Visitor. Los nodos se procesan en un orden específico (tipos, funciones, expresiones).
3. Generación de la Función main: Punto de entrada del programa. Contiene las expresiones de nivel superior.
4. Manejo de Expresiones: Cada expresión genera valores LLVM que se almacenan en la pila valueStack del contexto. Las expresiones complejas (operaciones, llamadas) desapilan operandos, generan código, y apilan resultados.
5. Salida del Código IR: Al final, se imprime el módulo LLVM en un archivo.

La estructura de datos responsable contiene funciones de inserción, eliminación, búsqueda y asignación para cada función, tipo o variable que se incluya en ella, y todas las implementaciones del código eventualmente pasarán a tener una representación en la mencionada estructura a partir de la cual se ejecutará un método para generar el código.