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, aunque añadiendo algunas extensiones sintácticas propias. 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 se traduce a código LLVM. En este punto concluye el proceso de compilación, generando un archivo `Hulk-IR.ll` en la carpeta `Hulk`.

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
```

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 y columna.

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 producciones para cada tipo de *statement* o expresión de HULK, así como una precedencia para los operadores. Cada producción genera un nodo del AST, con lo cual se obtiene finalmente una lista de nodos.

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.

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.

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

Figura 1. Clase ASTNode

```cpp
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 2. Ejemplo de declaración de un nodo del AST

```
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 3. Ejemplo de creación de nodos del AST desde Bison

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, haciendo 2 recorridos por el AST. El primer recorrido tendrá como objetivo recolectar todas las definiciones de funciones realizadas a lo largo del código, y 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. El compilador intentará inferir los tipos de los datos de manera automática 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. Para implementar este comportamiento, se definieron varias estructuras en C++:

- La estructura Symbol, que 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, que 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. Al declararse una variable en un programa de HULK, se crea el símbolo correspondiente y se guarda en el contexto en donde fue creada junto con su valor asignado. Si se encuentra el uso de la variable en algún posterior a su definición, se verifica primero si en el contexto existe ya esa variable definida, en HULK se pueden definir dos variables con el mismo nombre, al usar una dentro de un contexto, esta tomará como valor el último que se le asignó dentro del contexto actual, o en un contexto superior. El patrón visitor, al llegar a una expresión que indique el uso de una variable, deberá realizar las siguientes acciones: Comprobar si dicha variable fue definida en el contexto actual o en uno superior y analizar si su uso es correcto debido a su tipo de retorno. Las funciones siempre serán definidas en el Contexto global del programa.

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. Para lograrlo, se implementó una estructura que permita mantener el contexto durante la generación, que gestiona los diferentes scopes mediante pilas. 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. El flujo seguido es similar al mostrado durante el análisis semántico, reimplementando los métodos visit para cada nodo necesario. 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.