Compilador de HULK

Un informe detallado sobre la implementación de un Compilador para el lenguaje HULK

Integrantes:

Kevin Marquéz Vega
Javier Alejandro Gonzáles
José Miguel Leyva

Asignatura: Compilación
Universidad de La Habana

June 16, 2025

Semestre: Segundo Semestre | Tercer año

1 Introducción

Para la aplicación de los conocimientos adquiridos en la asignatura de Compilación, se llev'o a cabo la implementación y desarrollo de un compilador para el lenguaje HULK (Havana University Language for Kompilers) utilizando **C** como lenguaje base.

El compilador se divide en 4 etapas fundamentales: Análisis léxico *lexer*, Análisis sintáctico *parser*. Análisis semántico y Generación de código. Cada etapa será explicada detalladamente en el presente informe.

1.1 Flujo del Programa

El flujo del programa sigue el camino de cualquier compilador estándar. Comienza en la función *main()* dentro del script **main.c** la cual se encarga de leer el contenido del archivo **script.hulk** convirtiendo el mismo cadena de texto. Dicha cadena es enviada al *lexer* y *parser* para los análsis léxico y sintáctico respectivamente. Si no ocurre ningún error en las etapas descritas se generará el Árbol de Sintaxis Abstracta (AST por sus siglas en inglés). Posteriormente se comienza el Chequeo Semántico de las expresiones definidas en el archivo *.hulk* y finalmente se llega a la etapa de Generación de Código, en la cual se mostrarán los resultados de compilar un script del lenguaje HULK en un archivo generado llamado **output.ll** en la carpeta **build**.

2 Análisis Léxico: Lexer

Para el análisis léxico de cualquier lenguaje se necesitan, entre otras cosas, una definición de **Token**. Un Token es una cadena de carácteres que representa un símbolo que se utiliza en el lenguaje, ejemplo : palabras claves, operadores, variables etc. Se hizo un estudio a fondo del lenguaje **HULK** para obtener todos los tokens del mismo y preparar todo para comenzar el análisis.

El lexer utilizado es el brindado por la librería *Flex*, el cual es muy robusto y completo, facilitando el desarrollo de esta etapa de la compilación. 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.

Flex proporciona un generador de lexer, el cual, conociendo las expresiones regulares que intervienen en el lenguaje, y una secuencia de carácteres determina los tokens que participan en dicha cadena y los deja listos para el análisis sintáctico.

3 Análisis Sintáctico: Parser

En el Análisis sintáctico se hizo con el framework proporcionado por **Yacc**, el cual al obtener la gramática, bien definida, crea un AST en función del conjunto de expresiones que se utilizaron en el programa Hulk.

3.1 La gramática

La definición de la gramatica la puede encontrar en el archivo `parser.y` en la que se ve claramente las producciones. Debido a la comletitud y complejidad del lenguaje se hizo necesario definir una gramática que no fuera **LL1** para que todas las operaciones que intervienen

en las expresiones sean analizadas correctamente tanto por las tres primeras etapas como por la generación de código intermedio.

3.2 AST

El AST es una parte fundamental del desarrollo de compiladores, por lo que se hizo necesario poder modelar el comportamiento del mismo. Como en el lenguaje C no existe la programación orientada a objetos, es muy complicado simular la herencia y las clases abstractas, ambos conceptos facilitan la implementación y lectura de código cuando se trata de estructuras arboreas.

Para representar el AST se definió un nodo por cada tipo de expresión definida en la gramática del lenguaje ejemplo: `LetInNode`, `FunctionDefinitionNode`, `LiteralNode` etc. Además con el objetivo de simular la herencia, cada nodo tiene entre sus propiedades un struct de tipo **ASTNode**, el cual contiene todos los atributos que comparten los nodos del AST, como el tipo y el valor de retorno.

```c
typedef struct ASTNode {
    // Nodo base del AST (Abstract Syntax Tree)
    ASTNodeType type;            // Tipo de nodo AST
    TypeDescriptor* return_type; // Tipo de retorno del nodo AST
    LLVMValueRef (*accept)(struct ASTNode* self, struct LLVMCodeGe
} ASTNode;

typedef struct LiteralNode {
    // Nodo que representa un literal en el AST
    ASTNode base;
    union {
        double number_value;
        char *string_value;
        int bool_value;
    } value;
} LiteralNode;

typedef struct UnaryOperationNode {
    // Nodo que representa una operación unaria en el AST
    ASTNode base;
    HULK_Op operator; // Operador unario
    ASTNode *operand; // Operando del operador unario
} UnaryOperationNode;
```

Figure 1: Estructura de ASTNode

3.3 Flujo de Yacc

Una vez definida la gramática y el formato que tendrán los nodos del AST, se puede empezar a explicar el flujo de trabajo de Yacc. En el programa en C se tiene una función de creación por cada uno de los nodos anteriormente definidos. Dichas funciones son asociadas a las reglas de la gramática de esta forma se va construyendo el AST, dejando siempre como raíz un nodo del tipo `ProgramNode` a partir del cual iniciarán los procesos que quedan.

```
AddExpr         : AddExpr ADD MultExpr      { $$ = create_binary_operation_node(PLUS_TK, $1, $3, type_table ); }
                | AddExpr SUB MultExpr      { $$ = create_binary_operation_node(MINUS_TK, $1, $3, type_table); }
                | AddExpr CONCAT MultExpr   { $$ = create_binary_operation_node(CONCAT_TK, $1, $3, type_table); }
                | AddExpr DCONCAT MultExpr  { $$ = create_binary_operation_node(D_CONCAT_TK, $1, $3, type_table); }
                | MultExpr                  { $$ = $1; }
                ;

MultExpr        : MultExpr MUL PowExpr      { $$ = create_binary_operation_node(MULT_TK, $1, $3, type_table); }
                | MultExpr DIV PowExpr      { $$ = create_binary_operation_node(DIV_TK, $1, $3, type_table ); }
                | MultExpr MOD PowExpr      { $$ = create_binary_operation_node(MOD_TK, $1, $3, type_table); }
                | PowExpr                   { $$ = $1; }
                ;

PowExpr         : T POW PowExpr             { $$ = create_binary_operation_node(EXP_TK, $1, $3, type_table); }
                | T                         { $$ = $1; }
                ;

T               : NUMBER                    { $$ = create_number_literal_node($1 , type_table); }
                | BOOLEAN                   { $$ = create_bool_literal_node($1, type_table); }
                | STRING                    { $$ = create_string_literal_node($1, type_table); }
                | LPAREN Expression RPAREN  { $$ = $2; }
                | NOT T                     { $$ = create_unary_operation_node(NOT_TK, $2, type_table); }
                | SUB T                     { $$ = create_unary_operation_node(MINUS_TK, $2, type_table); }
                | ExprBlock                 { $$ = $1; }
                | ID                        { $$ = create_variable_node($1, type_table); }
                | ID REASSIGN Expression    { $$ = create_reassign_node($1, $3, type_table);}
                | ID LPAREN ArgList RPAREN
                {
                   $$ = create_function_call_node($1, $3.nodes, $3.count, type_table);
                }
```

Figure 2: Algunos ejemplos de llamadas en funciones desde Yacc

```c
typedef struct ProgramNode {
    // Nodo que representa un programa completo
    ASTNode base; // Nodo base del AST
    FunctionDefinitionListNode* function_list;  // Lista de definiciones de funciones
    TypeDefinitionListNode* type_definitions;   // Lista de definiciones de tipos
    ASTNode* root; // Bloque principal del programa
} ProgramNode;
```

Figure 3: Estructura de ASTNode

4 Chqueo Semántico

A partir de este momento comienza el chequeo semántico, el cual está implementado sin usar librerías, basándose en la definición de nodos de AST mostrada anteriormente. Se hace uso del **Patrón Visitor** para realizar esta tarea. El idea es hacer varios recorridos por el AST Generado dividiendo el trabajo en varias etapas. La primera será para capturar todas las definiciones de funciones y creación de variables que se hallan realizado, la segunda para las definiciones de tipos y sus propiedades y finalmente para ver si el uso de las variables dentro de operaciones o funciones es el correcto de acuerdo a su tipo.

El compilador se pensó para la versión de HULK de tipado estático, *Type-HULK* es decir, tanto para la definición de funciones como para la de tipos es necesario definir el tipo de cada uno de los parámetros. Esto ayuda en el momento de inferir el tipo de retorno de las funciones y ayuda a que no se hagan llamadas con los parámetros incorrectos, como por ejemplo `Factorial("5")`.

Para llevar a cabo esta tarea, se definieron numerosas estructuras en C para manejar los contextos. Se tiene la struct `Symbol`, la cual se usa para guardar todas las propiedades referidas al uso de variables y llamado de funciones. Una struct con el nombre de `SymbolTable`, que representa el contexto, dicha tabla guarda la información de todas aquellas variables y funciones que fueron definida en un contexto determinado.

4.1 Primera Etapa: Funciones

Como se mencionó anteriormente en esta etapa se busca capturar todas las defincciones de funciones que se realizaron en el script de HULK. Basicamente se recorre cada nodo del AST hasta encontrar un **FunctionDefinitionNode**. Aclarar que en el lenguaje HULK no se pueden definir funciones dentro de otras funciones ni dentro de cualquier otro nodo que implique la creación de un bloque de expresiones, por lo que este recorrido es posible y basta con analizar la C̈apa Superiord̈el script. Una vez encontradas todas las funciones se guardan en el Contexto global, permitiendo acceder a ellas desde cualquier punto del script de HULK.

4.2 Segunda Etapa: Tipos

Una vez guardadas todas las funciones en el Contexto global, se procede de manera similar para el analisis de tipo. Dentro de cada tipo se definirá un nuevo contexto, debido a que dentro del cuerpo de un tipo pueden existir propiedades y funciones, por tanto se guardan en un contexto propio que tienen los tipos nuevos definidos.

4.3 Tercera Etapa: Visita Semántica

Se implementó un struct **TypeDescriptor** en C para poder tener control de los tipos que intervienen en el lenguaje HULK así como los tipos definidos por el usuario, para estos últimos se definió la struct **TypeInfo**. El patrón visitor lo que hará es entrar en cada nodo del AST, los cuales tienen su propia forma de chequearse e irá retornando los TypeDescriptors correspondientes.

```c
typedef struct TypeDescriptor {
    // Describe un tipo de dato en el lenguaje Hulk.
    char* type_name;                // Nombre del tipo
    HULK_Type tag;                  // Especifica el tipo de dato(puede ser un tipo primitivo o un tipo definido por el usua
    TypeInfo* info;                 // NULL para tipos primitivos, apunta a la información del tipo para tipos definidos por
    struct TypeDescriptor* parent;  // Tipo padre, Object por defecto
    bool initialized;               // Especifica si el tipo ya ha sido inicializado(para tipos del usuario)
    LLVMTypeRef llvm_type;          // Referencia al tipo de dato en LLVM, NULL si no se ha generado
    int type_id;               // Identificador único del tipo, se usa para identificar tipos en el compilador
} TypeDescriptor;

typedef struct TypeInfo {
    // Información adicional sobre un tipo definido por el usuario.
    char** params_name;             // Nombres de los parámetros del tipo, NULL si no tiene parámetros
    int param_count;                // Cantidad de parámetros del tipo
    struct SymbolTable* scope;      // Tabla de simbolos donde iran atributos y metodos
    TypeDefinitionNode* type_def;      //Definición explícita del tipo
} TypeInfo;
```

Figure 4: Type Descriptor y TypeInfo

4.4 Declaración y uso de Variables

Al declararse una variable en un programa de HULK, como bien se explicó anteriormente, 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 definicioón, se chequea primera 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: Chequear si dicha variable fue definida en el contexto actual o en uno superior y posteriormente analizar si su uso es correcto debido a su tipo de retorno.

```
let a = 10 in
  if (a >= 0)
  {
    print(a + 5);
  }
  else
  {
    let a = 11 in print(a);
  }
```

4.5 Definición y llamado de Funciones

Las funciones siempre serán definidas en el Contexto global del programa, como se explicó anteriormente, el tipado es estático por lo que se debe especificar el tipo de cada parámetro de las funciones para poder declararlas. Como esto se conoce, es más fácil determinar si las operaciones que se realizan dentro del cuerpo de una función tienen semántica correcta.

Respecto al llamado de funciones, es sabido que en el HULK el tipo de retorno de una función es el mismo que el de su cuerpo, como el cuerpo de una función puede ser un bloque de expresiones, entonces el tipo de retorno de una función es el de la última expresión de su cuerpo. De esa forma al hacer algo como:

```
function Sucesor(n : Number) => n + 1;
let a : Number = 3 in print(Square(a) + 4);
```

No ocurrirá ningún error semántico, debido a que se especifica el tipo de dato del parametro `n` y es fácil determinar que la expresión `n + 1` es semánticamente correcta y devuelve un tipo Number.

4.6 Definición de tipos y acceso a propiedades

HULK es lenguaje con POO, por tanto se puede definir tipos nuevos en cualquier momento y utilizarlos en el programa