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

Para el *lexer* y el *parser* se ha utilizado el framework proporcionado por `Lex` y `Yacc`, aunque se tiene una implementación manual del *lexer* utilizando la teoría de autómatas finitos.

2.1. Lex. Definición de los tokens

El *lexer* es el encargado de transformar la entrada del programa (un archivo de texto) en una secuencia de tokens, que son las unidades léxicas del lenguaje. En el programa, los tokens se definen en el archivo `lexer.l`, donde se especifican las expresiones regulares para cada tipo de token.

Errores: En el *lexer* se detecta únicamente un error a la vez, asociados a caracteres inválidos (ver ejemplo Figura 1).

```
joel@ubuntu:~/Documents/GitHub/Hulk-Full_Compiler$
!!LEXICAL ERROR: Invalid character '#'. Line: 196
```

Figura 1. Error en lexer

2.2. Yacc. Definición de la gramática y el AST

El *parser* es el encargado de analizar la secuencia de tokens generada por el *lexer* y construir un árbol de sintaxis abstracta (AST) a partir de ellos. En el programa, la gramática del lenguaje se define 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, que se conectan con un nodo raíz, que se hace llamar *program node*.

Existen diversos tipos de nodos que se pueden crear en el parser, cada uno representando una construcción del lenguaje (Ver Figura 2). Como C no es orientado a objetos, el tipo `Node` se representa con una estructura que posee campos generales que necesitan la gran mayoría de los tipos de nodos, y entre ellos un campo `data` que es un *union* que tiene las propiedades específicas de cada tipo (Ver Figura 3). Además, existe una función para cada tipo de nodo que permite inicializarlo.

```
typedef enum {
    NODE_NUMBER,
    NODE_VARIABLE,
    NODE_BINARY_OP,
    NODE_UNARY_OP,
    NODE_ASSIGNMENT,
    NODE_D_ASSIGNMENT,
    NODE_STRING,
    NODE_BOOLEAN,
    NODE_PROGRAM,
    NODE_FUNC_CALL,
    NODE_BLOCK,
    NODE_FUNC_DEC,
    NODE_LET_IN,
    NODE_CONDITIONAL,
    NODE_Q_CONDITIONAL,
    NODE_LOOP,
    NODE_FOR_LOOP,
    NODE_TEST_TYPE,
    NODE_CAST_TYPE,
    NODE_TYPE_DEC,
    NODE_TYPE_INST,
    NODE_TYPE_GET_ATTR,
    NODE_TYPE_SET_ATTR,
    NODE_BASE_FUNC
} NodeType;
```

Figura 2. Tipos de nodos del AST

```
typedef struct ASTNode {
    int line; int is_param; int checked;
    NodeType type; Type* return_type;
    char* static_type;
    Scope* scope; Context* context;
    ValueList* derivations;
    union {
        double number_value;
        char* string_value;
        char* variable_name;
        struct {
            Operator op; char* op_name;
            struct ASTNode *left; struct ASTNode *right;
        } op_node;
        struct {
            struct ASTNode** statements; int count;
        } program_node;
        struct {
            char* name; struct ASTNode** args;
            int arg_count; struct ASTNode *body;
        } func_node;
        struct {
            struct ASTNode *cond; struct ASTNode *body_true;
            struct ASTNode *body_false;
        } cond_node;
        struct {
            char* name; char* parent_name;
            struct ASTNode** p_args; int p_arg_count;
            struct ASTNode** args; int arg_count;
            struct ASTNode** definitions;
            int def_count; int id;
        } type_node;
        struct {
            char* type_name; Type* type;
            struct ASTNode* exp;
        } cast_test;
    } data;
} ASTNode;
```

Figura 3. AST

Durante la construcción del AST se realizan algunas modificaciones que permiten el uso de *azúcar* sintáctico, además de facilitar las restantes fases de la compilación:

```
1    elif --> else if
2    x += 1 --> x := x + 1
3    x -= 1 --> x := x - 1
4    x *= 1 --> x := x * 1
5    x /= 1 --> x := x / 1
6    x %= 1 --> x := x % 1
7    x ^= 1 --> x := x ^ 1
8    x @= "hi" --> x := x @ "hi"
```

```
9     x &= true --> x := x & true
10    x |= false --> x := x | false
11    while(<exp>) <body> --> if(<exp>) while(<exp>) <body>
```

Errores: En el *parser* se detecta, al igual que en el *lexer*, un único error a la vez, pero en este caso por ser incompatible con todas las producciones, es decir, por no ajustarse a la sintaxis del lenguaje (ver ejemplo Figura 4).

Figura 4. Error en parser

2.3. Generador de lexer propio. NFA y DFA

Como un *lexer* clásico recibe una cadena de caracteres y genera una lista de tokens definidos por las expresiones regulares que describen el lenguaje. El flujo general se basa en:

1. Crear los autómatas finitos no deterministas (NFA) de cada expresión regular.
2. Construir el autómata de la unión de todos los NFAs para definir el lenguaje que reconocerá el lexer.
3. Convertir el NFA de la unión a un DFA (autómata finito determinista).
4. Generar la secuencia de tokens a partir del DFA construido.

Intérprete de expresiones regulares

El objetivo de este intérprete es construir un NFA a partir de una expresión regular. Cuenta con un *parser* específico que, a partir de los tokens resultantes de tokenizar la expresión regular, construye un árbol de sintaxis abstracta (AST). Este AST es la entrada del método para generar el NFA correspondiente. Reconoce la siguiente sintaxis de expresiones regulares:

- | : Unión.
- * : Clausura.
- + : Clausura positiva.
- [ ]: Conjunto.
- [^]: Conjunto negado.
- [-]: Conjunto expandido.
- . : Cualquier caracter.

Algoritmo para transformar un NFA a un DFA

El algoritmo utiliza un método auxiliar para computar la ϵ-clausura (estados alcanzables por ϵ-transiciones) de un conjunto de estados del NFA. El DFA tendrá en cada estado, subconjuntos de los estados del NFA.

El algoritmo comienza calculando la ϵ-clausura del estado inicial del NFA y esa será el estado inicial del DFA. Luego, por cada símbolo del alfabeto, por cada estado del NFA en el estado actual del DFA, se crea un conjunto de estados nuevo, que serán el resultado de aplicar la función de transición del NFA al estado y símbolo dados. Entonces el siguiente estado del DFA será la ϵ-clausura de dicho conjunto recién construido, y sus estados finales serán los que contengan al menos un estado que era final en el NFA.

Generación de tokens

Los estados finales del DFA contienen los posibles tokens con que puede coincidir la cadena. En el proceso de construcción de un token se transita por el DFA mientras sea posible y, si se llega a un estado final, se guarda el lexema que se está construyendo y la lista de tokens correspondiente al estado en cuestión. La construcción de un token se puede detener por dos razones: no hay ninguna transición disponible en el autómata o se encuentra un espacio en blanco. En ambos casos, se guarda el (los) último (s) que coincidió (coincidieron) y se reinicia el recorrido por el DFA, comenzando por el caracter que detuvo la máquina. Finalmente, se retorna el token que más prioridad tenga (en caso de tener varios tokens, sino se retorna el único que se tiene).

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 5) 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.

```
typedef struct Visitor Visitor;

typedef void (*VisitProgram)(Visitor*, ASTNode*)    ; typedef void (*VisitNumber)(Visitor*, ASTNode*);
typedef void (*VisitString)(Visitor*, ASTNode*)     ; typedef void (*VisitBoolean)(Visitor*, ASTNode*);
typedef void (*VisitVariable)(Visitor*, ASTNode*)   ; typedef void (*VisitBinaryOp)(Visitor*, ASTNode*);
typedef void (*VisitUnaryOp)(Visitor*, ASTNode*)    ; typedef void (*VisitAssignment)(Visitor*, ASTNode*);
typedef void (*VisitFuncCall)(Visitor*, ASTNode*)   ; typedef void (*VisitBlock)(Visitor*, ASTNode*);
typedef void (*VisitFuncDec)(Visitor*, ASTNode*)    ; typedef void (*VisitLetIn)(Visitor*, ASTNode*);
typedef void (*VisitConditional)(Visitor*, ASTNode*); typedef void (*VisitLoop)(Visitor*, ASTNode*);
typedef void (*VisitForLoop)(Visitor*, ASTNode*)    ; typedef void (*VisitTypeDec)(Visitor*, ASTNode*);
typedef void (*VisitTypeInst)(Visitor*, ASTNode*)   ; typedef void (*VisitCastingType)(Visitor*, ASTNode*)
typedef void (*VisitTestType)(Visitor*, ASTNode*)   ; typedef void (*VisitAttrGetter)(Visitor*, ASTNode*);
typedef void (*VisitAttrSetter)(Visitor*, ASTNode*) ; typedef void (*VisitBaseFunc)(Visitor*, ASTNode*);

You, 1 minute ago | 2 authors (You and one other)
struct Visitor {
    int error_count;

    VisitProgram visit_program        ; VisitNumber visit_number;
    VisitString visit_string          ; VisitBoolean visit_boolean;
    VisitVariable visit_variable      ; VisitBinaryOp visit_binary_op;
    VisitUnaryOp visit_unary_op       ; VisitAssignment visit_assignment;
    VisitFuncCall visit_function_call ; VisitBlock visit_block;
    VisitFuncDec visit_function_dec   ; VisitLetIn visit_let_in;
    VisitConditional visit_conditional; VisitLoop visit_loop;
    VisitForLoop visit_for_loop       ; VisitTypeDec visit_type_dec;
    VisitTypeInst visit_type_instance ; VisitCastingType visit_casting_type;
    VisitTestType visit_test_type     ; VisitAttrGetter visit_attr_getter;
    VisitAttrSetter visit_attr_setter ; VisitBaseFunc visit_base_func;

    char** errors;
    char* current_function;
    Type* current_type;
};

void accept(Visitor* visitor, ASTNode* node);
void get_context(Visitor* visitor, ASTNode* node);
void report_error(Visitor* v, const char* fmt, ...);
void free_error(char** array, int count);
```

Figura 5. 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 ($+, -, *, /, \%,^{'} \&, |, ==, != , <, >, <=, >=, @, @@$), 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 6) 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.

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 7).

```
OperatorTypeRule operator_rules[] = {

// ----------------------  OPERATIONS ALLOWED  --------------------------------

//      left_type     |      right_type      |   return_type    | operator

    // math binary
    { &TYPE_NUMBER, &TYPE_NUMBER, &TYPE_NUMBER, OP_ADD },// +
    { &TYPE_NUMBER, &TYPE_NUMBER, &TYPE_NUMBER, OP_SUB },// -
    { &TYPE_NUMBER, &TYPE_NUMBER, &TYPE_NUMBER, OP_MUL },// *
    { &TYPE_NUMBER, &TYPE_NUMBER, &TYPE_NUMBER, OP_DIV },// /
    { &TYPE_NUMBER, &TYPE_NUMBER, &TYPE_NUMBER, OP_MOD },// %
    { &TYPE_NUMBER, &TYPE_NUMBER, &TYPE_NUMBER, OP_POW },// ^
    //string binary
    { &TYPE_NUMBER, &TYPE_STRING, &TYPE_STRING, OP_CONCAT},// @
    { &TYPE_STRING, &TYPE_STRING, &TYPE_STRING, OP_CONCAT},
    { &TYPE_STRING, &TYPE_NUMBER, &TYPE_STRING, OP_CONCAT},
    { &TYPE_NUMBER, &TYPE_STRING, &TYPE_STRING, OP_DCONCAT},// @@
    { &TYPE_STRING, &TYPE_STRING, &TYPE_STRING, OP_DCONCAT},
    { &TYPE_STRING, &TYPE_NUMBER, &TYPE_STRING, OP_DCONCAT},
    //boolean binary
    { &TYPE_BOOLEAN, &TYPE_BOOLEAN, &TYPE_BOOLEAN, OP_AND },// &
    { &TYPE_BOOLEAN, &TYPE_BOOLEAN, &TYPE_BOOLEAN, OP_OR },// |
    //comparison
    { &TYPE_NUMBER, &TYPE_NUMBER, &TYPE_BOOLEAN, OP_EQ },// ==
    { &TYPE_STRING, &TYPE_STRING, &TYPE_BOOLEAN, OP_EQ },
    { &TYPE_BOOLEAN, &TYPE_BOOLEAN, &TYPE_BOOLEAN, OP_EQ },
    { &TYPE_NUMBER, &TYPE_NUMBER, &TYPE_BOOLEAN, OP_NEQ },// !=
    { &TYPE_STRING, &TYPE_STRING, &TYPE_BOOLEAN, OP_NEQ },
    { &TYPE_BOOLEAN, &TYPE_BOOLEAN, &TYPE_BOOLEAN, OP_NEQ },
    { &TYPE_NUMBER, &TYPE_NUMBER, &TYPE_BOOLEAN, OP_GRE },// >=
    { &TYPE_STRING, &TYPE_STRING, &TYPE_BOOLEAN, OP_GRE },
    { &TYPE_NUMBER, &TYPE_NUMBER, &TYPE_BOOLEAN, OP_GR },// >
    { &TYPE_STRING, &TYPE_STRING, &TYPE_BOOLEAN, OP_GR },
    { &TYPE_NUMBER, &TYPE_NUMBER, &TYPE_BOOLEAN, OP_LSE },// <=
    { &TYPE_STRING, &TYPE_STRING, &TYPE_BOOLEAN, OP_LSE },
    { &TYPE_NUMBER, &TYPE_NUMBER, &TYPE_BOOLEAN, OP_LS },// <
    { &TYPE_STRING, &TYPE_STRING, &TYPE_BOOLEAN, OP_LS },
    //unary
    { &TYPE_BOOLEAN, NULL, &TYPE_BOOLEAN, OP_NOT },// !
    { &TYPE_NUMBER, NULL, &TYPE_NUMBER, OP_NEGATE },// (-)
};
```

Figura 6. Reglas para los operadores

```
🔗 Getting ready...
🔄 Compiling...
!!SEMANTIC ERROR: Operator '+' can not be used between 'Number' and 'String'. Line: 350.
!!SEMANTIC ERROR: Operator '&' can not be used between 'Number' and 'Boolean'. Line: 351.
!!SEMANTIC ERROR: Operator '!' can not be used with 'String'. Line: 352.
!!SEMANTIC ERROR: Invalid scape sequence '\s'. Line: 353.
!!SEMANTIC ERROR: Operator '@@' can not be used between 'Number' and 'Boolean'. Line: 354.
```

Figura 7. 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 8).

```
!!SEMANTIC ERROR: Variable 'x' was defined as 'A', which is not a valid type. Line: 350.
!!SEMANTIC ERROR: Variable 'x' was defined as 'Number', but inferred as 'Boolean'. Line: 351.
!!SEMANTIC ERROR: Variable 'x' needs to be initializated in a 'let' definition before being reassigned. Line: 352.
!!SEMANTIC ERROR: Variable 'x' was initializated as 'Number', but reassigned as 'Boolean'. Line: 353.
!!SEMANTIC ERROR: Undefined variable 'y'. Line: 354
```

Figura 8. 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 funciones**: 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 funciones**: 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 9).

```
!!SEMANTIC ERROR: Parameter 'x' was defined as 'A', which is not a valid type. Line: 350.
!!SEMANTIC ERROR: Symbol 'x' is used for both argument '1' and argument '2' in function 'f1' declaration. Line: 351.
!!SEMANTIC ERROR: The return type of function 'f2' was defined as 'Number', but inferred as 'Boolean'. Line: 352.
!!SEMANTIC ERROR: Function 'f2' receives 1 argument(s), but 2 was(were) given. Line: 354.
!!SEMANTIC ERROR: Function 'f2' receives 'Number', not 'String' as argument 1. Line: 355.
```

Figura 9. Errores comunes en funciones

3.4. Análisis de condicionales y ciclos

La revisión de condicionales y ciclos se centra precisamente en revisar cada una de esas expresiones, además de un tipo especial de condicional: *q-conditional*. A continuacón se explican cada una de ellas detalladamente:

1. **Condicionales**: Las condicionales tienen tres partes: condición, bloque para el *if*, bloque para el *else* (el *elif* es convertido automáticamente por el compilador a un *else if*). La revisión ocurre secuencialmente por esas tres partes: se visita el nodo de la condición y se verifica que su tipo de retorno sea booleano, sino se reporta error; luego se visita el bloque del *if* y más adelante el del *else*. Entonces la expresión condicional toma como tipo de retorno el ancestro común más cercano entre los tipos de retorno de ambos bloques. En caso de no tener bloque *else*, se asume para ese caso un tipo por defecto para esta rama, que puede ser el mismo tipo que el del primer bloque para los tipos *builtin* (excepto `Object`) y `Null` para el resto. Este último genera que el valor de retorno de la condicional sea el tipo del primer bloque pero *nulleable*, por tanto, no se podrá utilizar luego en ningún caso donde se requiera ese tipo sin utilizar la sintaxis de *q-conditional*, que lo que hará será separar los casos en dos ramas: cuando el tipo sea `Null` y cuando sea exactamente el tipo deseado (se explicará luego).
2. **Ciclos *while***: El ciclo *while* es mucho más sencillo de verificar que las condicionales, pues solo admite un cuerpo. Por tanto, su chequeo se basa en visitar la condición y reportar error si su tipo de retorno no es booleano, y

más tarde visitar el cuerpo y actualizar el tipo de retorno del ciclo con el tipo de retorno del cuerpo.

3. **Ciclos *for***: Este ciclo es más complejo que el anterior y comprende varias transformaciones importantes. Se chequea primero que los parámetros de la funcion `range` cumplan los requisitos esperados (ser 1 o 2 parámetros numéricos). Luego el nodo *for* se transforma en un *let-in* para optimizar la generación de código y se revisa como tal:

```
// ---> for original <---
for (x in range(start, end)) {
    // cuerpo del for
}

// ---> Transformacion <---
let _iter = start - 1 in
// _ al inicio evita que el usuario pueda usarla
    if (_iter += 1 < end) // por si no entra al while
        while (_iter += 1 < end)
            let x = _iter in {
                // cuerpo del for
            }
// Esta transformacion simula un iterable
```

4. **Condicionales *q-conditional***: Este tipo de condicional es una extensión al lenguaje que permite separar los casos de un tipo nulleable en dos ramas: una para el caso donde el es `Null` y otra donde es el tipo original no *nulleable*. Posee la siguiente sintaxis:

```
if?(<expression>) // variable
    <expression> // asumiendo Null
else
    <expression> // asumiendo tipo original
```

   Por detalles de tiempo, esta nueva sintaxis se mantiene en una versión inicial donde solo es capaz de recibir como argumento una variable. No obstante, como en una variable se puede guardar cualquier expresión, pues es completamente funcional. Al igual que las condicionales, su tipo de retorno es el ancestro común más cercano entre ambos bloques (son obligatorios ambos bloques). Fuera de la expresión *q-conditional* la variable mantiene su estado original, así que si se desea que se comporte como el tipo no *nulleable* debe usarse dentro del bloque *q-conditional* necesariamente.

Errores: Los errores en estas revisiones son pocos, principalmente cuando las condiciones no son booleanas y cuando hay incompatibilidad con los parámetros de la función range. (ver ejemplo Figura 10).

3.5. Análisis de Tipos

El análisis de tipos es el chequeo más abarcador de esta sección. Comprende la declaración de tipos, instancias de tipos, obtención y modificación de atributos,

```
!!SEMANTIC ERROR: Condition in 'if' expression must return 'Boolean', not 'Number'. Line: 367
!!SEMANTIC ERROR: Condition in 'while' expression must return 'Boolean', not 'Number'. Line: 368
!!SEMANTIC ERROR: Function 'range' receives 'Number', not 'String' as argument 1. Line: 369
!!SEMANTIC ERROR: Function 'range' receives 1 or 2 arguments, not 3. Line: 392
```

Figura 10. Errores comunes en condicionales y ciclos

operaciones de chequeo y conversión de tipos, y el caso especial de la función *base*.

Para comprender mejor estos análisis, es necesario explicar la estructura `Type` en el programa. Esta estructura es la que maneja, en todo momento del chequeo semántico, los tipos de retorno de las expresiones. Posee, de manera general, un nombre, un padre, los tipos de los parámetros que recibe su constructor, así como la cantidad, y además tiene un campo para acceder a su declaración y poder tener referencia a sus atributos y métodos. Sobre un `Type` se pueden hacer distintas operaciones:

- Crear un nuevo tipo
- Saber si el tipo es *builtin*
- Saber si dos tipos son iguales
- Saber si un tipo es ancestro de otro
- Saber el ancestro común más cercano entre dos tipos
- Saber si dos tipos están en la misma rama de la jerarquía de tipos
- Obtener el tipo *nulleable* asociado

Una vez comprendido el funcionamiento de un `Type` se explicará cada uno de los chequeos que se pueden realizar:

1. **Chequeo de tipos**: Este chequeo se puede realizar mediante el operador *is*. Primero se visita al nodo de la expresión que se quiere chequear y luego se revisa que el tipo por el que se pregunta exista en el *scope*, o en última instancia en el *context*. De no ser así, se reportará error.
2. **Conversión de tipos**: Al igual que el chequeo de tipos, se visita la expresión y se revisa que el tipo a convertir exista o bien en el *scope* o en el *context*. Además, hay una verificación extra que detecta si el tipo de la expresión y al que se quiere convertir estén en la misma rama de la jerarquía de tipos, de lo contrario se reporta también error.
3. **Instancia de tipos**: Las instancias de tipos, de cierta manera, son similares a los llamados de funciones, por lo tanto su chequeo es igual, solo que en vez de buscar funciones definidas se busca tipos. Incluso los errores reportados son los mismos (solo cambia *function* por *type* en la mayoría de los mensajes).
4. **Declaración de tipos**: La declaración de tipos tiene un chequeo robusto, pues puede dar lugar a decenas de errores distintos. Se comienza revisando que no exista herencia circular, para lo cual se mantiene una lista estática a la que se tiene acceso y que almacena nombres de tipos. Se chequea que el nombre del padre no esté en esa lista (en caso que sí se reporta la circularidad), y luego se añade. Esa lista se vacía nuevamente cuando se inicie el chequeo del cuerpo de la declaración del tipo. Sobre los parámetros del constructor

se realizan las mismas verificaciones que en la declaración de una función. Con respecto a la herencia, se verifica, en caso de tener padre, que dicho tipo sea válido, sino se reporta error, y además, se revisa la compatibilidad de los parámetros, que incluye añadir a la declaración del tipo los parámetros que recibe el constructor del padre en caso de no especificar los suyos propios, o verificar que se haya dado una inicialización correcta al padre a partir de los parámetros que reciba el tipo en cuestión. Con eso se da por concluida la revisión de la cabecera del tipo y se comienza la revisión del cuerpo. Lo primero es recoger un contexto interno, que incluye inicializaciones de campos y métodos. Luego se visita primero cada una de las inicializaciones y luego los métodos (antes de visitarlos se declara en el *scope* interno del tipo la variable especial *self* del mismo tipo que el declarado). Para estos últimos, cabe resaltar que además de los chequeos habituales de las declaraciones de funciones, también se verifica que cumpla con la signatura de la declaración previa en un ancestro (de haberla). Finalmente se crea el tipo y se le asocia dicho nodo de la declaración para poder acceder luego a sus miembros.

5. **Obtención de atributos**: Para obtener atributos o métodos de un tipo específico, se emplea la notación punto: `instance.member`. Por tal motivo, primero se visita el nodo que representa la instancia del tipo. Una vez que se tiene la instancia, se verifica que el miembro solicitado esté definido en el tipo de la instancia, en caso contrario se reporta error. Finalmente, si existe el miembro, se visita su nodo y se actualiza el tipo de retorno del nodo principal, con el tipo de retorno del miembro asociado.
6. **Modificación de atributos**: Para modificar atributos se usa la misma sintaxis de notación punto: `instance.member := value`. Igualmente se visita el nodo de la instancia y se verifica que contenga el atributo a modificar, de lo contrario se reporta error. Luego se visita el nodo de la expresión que constituye la modificación que se quiere realizar. Luego se revisa que el tipo de dicha expresión sea descendiente del tipo con que fue inicializado el atributo, y si no, también se detecta error.
7. **Función *base***: Es una función especial, que necesita conocer el método y el tipo actuales, ambos datos guardados en el *visitor*. El chequeo comienza tomando esos datos y buscando el tipo más próximo que posee una declaración para dicha función. En caso de no existir, se reporta error. Luego, como es, en esencia, un llamado a función, debe chequearse como tal. Esto se logra creando un nodo auxiliar de tipo `NODE_FUNC_CALL` con el nombre de la función que se encontró y los argumentos que recibía *base*. Luego se visita ese nodo y cualquier error que pueda existir quedará registrado. Finalmente, el tipo de retorno de *base* es el tipo de retorno de la función que invoca.

Errores: Los errores en el manejo de tipos son muy diversos. Existen errores en las declaraciones que son similares a los de las declaraciones de funciones, así como en las instancias que son parecidos a los llamados. Además, tiene errores propios, referentes a la herencia y referencias circulares, heredar de tipos *builtin*, redefinición de atributos y métodos, acceder a los atributos desde fuera de la instancia (son privados), atributo o método inexistente en un tipo específico,

conversión de tipos inválida, intentar modificar un atributo asignando un valor de tipo distinto al de su inicialización, entre otros (ver ejemplo Figura 11).

```
!!SEMANTIC ERROR: Type 'A' inherits from 'B', but that type produces a cirular reference that can not be solved. Line: 362.
!!SEMANTIC ERROR: Type 'B' inherits from 'A', but that type produces a cirular reference that can not be solved. Line: 363.
!!SEMANTIC ERROR: Type 'C' can not inherit from 'Object'. Line: 364.
!!SEMANTIC ERROR: Member 'x' already exists in type 'D'. Line: 367.
!!SEMANTIC ERROR: Variable '_D_x' was initializated as 'Number', but reassigned as 'String'. Line: 369.
!!SEMANTIC ERROR: Impossible to access to 'y' in type 'D' because all attributes are private. Line: 371.
!!SEMANTIC ERROR: Undefined function '_D_getY'. Line: 371.
!!SEMANTIC ERROR: Type 'D' can not be downcasted to type 'Number'. Line: 372.
```

Figura 11. Errores comunes en el manejo de tipos

3.6. Inferencia de tipos. Unificación

La implementación propuesta del compilador de HULK, posee una inferencia de tipos fuerte, es decir, el tipado explícito solo es necesario cuando el programa detecte que es incapaz de inferir el tipo. Existe una primera inferencia simple y sencilla, basada en un conjunto de reglas básicas:

- Los literales son los más fáciles de inferir, ya que su tipo proviene directamente del analizador. Las expresiones aritméticas también son fáciles, ya que su tipo siempre es `Number`. Asimismo, los operadores de cadena y booleanos son sencillos.
- El tipo de expresiones complejas que tienen un cuerpo de expresión se determina por el tipo de dicho cuerpo. Este es el caso de *let-in*, *while* y *for*.
- El tipo de un bloque de expresión es el tipo de la última expresión del bloque.
- El tipo de una invocación de función o método es el tipo de su cuerpo.
- El tipo de expresiones que tienen más de una rama (*if*) es el ancestro común más cercano de los tipos de cada rama, o, en última instancia, `Object`.

Más allá de esa inferencia simple, también existe una mucho más compleja, referente a los parámetros de funciones y tipos. Esos parámetros no tienen asociado una expresión en las declaraciones, por tanto, si no están tipados, es imposible inferir su tipo basado solo en las reglas anteriores. Hay que deducirlo entonces a partir de su uso. Para eso, se ha diseñado una especie de motor de unificación, el cual intenta asignar un tipo específico a un parámetro.

Para su correcto funcionamiento, cada nodo del AST tiene un campo *derivations*, que puede ser visto como una lista de nodos. Cada nodo en esa lista repesenta una expresión de la cual el nodo principal podría haber obtenido un tipo `Any`, que es el que se le asigna inicialmente a los parámetros no tipados.

Dado que el programa solo asigna directamente `Any` a los parámetros y al retorno de las funciones que aún se están chequeando, cualquier expresión cuyo tipo de retorno sea `Any` necesariamente depende de un parámetro o un llamado a una de esas funciones. Dicho esto, el unificador se encargará de buscar la raíz del tipado `Any`, es decir, la función o el parámetro específico que derivó en la expresión más grande. Para hacerlo, simplemente toma un nodo y, o bien es un

parámetro o una función, o es una expresión que contiene alguno. Si se cumple lo primero, directamente se actualiza su tipo con el que se quiere unificar, y en otro caso se intenta unificar cada una de sus *derivations*. Luego de unificar un nodo, debe visitarse nuevamente para expandir los cambios producidos.

Todo este procedimiento de unificación necesita claramente un tipo con el cual unificar. Hay distintas maneras de obtener dicho tipo, en dependencia del nodo actual. Algunos pueden pedir unificación con un tipo directamente, como es el caso de las condicionales, que intentan unificar la expresión de la condición con el tipo `Boolean`. Sin embargo en otros casos, se necesita un proceso para obtener el tipo, antes de intentar unificarlo. Ese es el caso de las expresiones aritméticas, o los llamados a funciones e instancias de tipos. Otro de los más destacados, es el uso de parámetros en notación punto, donde la inferencia debe buscar entre todos los tipos cuál es el que tiene el miembro requerido e intentar unificar con ese (siempre se queda con el ancestro común más cercano que implemente el método entre todos los que implementan el método, o reportará error).

Errores: Los errores de inferencia se resumen en dos tipos: se infirieron dos tipos de ramas distintas de la jerarquía de tipos, o simplemente no se pudo inferir un tipo (ver ejemplo Figura 12).

```
joel@ubuntu:~/Documents/GitHub/Hulk-Full_Compiler$ make compile
 !!SEMANTIC ERROR: Impossible to infer type of parameter 'x' in function 'f'. It must be type annotated. Line: 362.
 !!SEMANTIC ERROR: Parameter 'x' behaves both as 'Number' and 'Boolean'. Line: 365.
```

Figura 12. Errores comunes en la inferencia de tipos

4. Generación de Código

La fase final del compilador es la generación de código intermedio (LLVM). En este fase, se utilizó el framework de C para LLVM, a través del cual se genera el archivo `output.ll`. En esta sección no se abordará específicamente la sintaxis de ese lenguaje, sino que se explicarán las ideas y abstracciones principales que permitieron llevar un lenguaje de alto nivel como HULK a un lenguaje intermedio.

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)