Instituto Tecnológico de Buenos Aires

Proyecto Especial - Segunda Entrega

72.39 - Autómatas, Teoría de Lenguajes y Compiladores

Autores:

Ballerini, Santiago[1] - 61746
Martone, Gonzalo Alfredo[2] - 62141
Bafico, Juan Cruz [3] - 62070
Bosetti, Franco [4] - 61654

Docentes:

Ana Maria Arias Roig

Mario Agustin Golmar

Rodrigo Ezequiel Ramele

Fecha de entrega: 22 de Junio de 2023

[1]sballerini@itba.edu.ar - @Cuinardium (github)
[2]gmartone@itba.edu.ar - @ImNotGone (github)
[3]jbafico@itba.edu.ar - @jBafico (github)
[4]fbosetti@itba.edu.ar - @francobosetti (github)

Índice

1. Introducción

En este informe se detallara la implementación de un compilador de un lenguaje de programación que permita manipular arboles. El objetivo del lenguaje propuesto es proveer una forma sencilla de crear arboles de enteros, manipularlos y persistirlos en archivos para un uso posterior. En concreto se ha elegido el formato de archivo *.dot* para la persistencia de los arboles creados. Esto nos permite incluso que los arboles creados por nuestro lenguaje puedan ser leídos por un programa escrito en cualquier lenguaje de programación. Esto se realiza con las siguientes funciones.

2. Consideraciones de uso

Para utilizar nuestro compilador, creamos reglas de make para un facil uso. Al correr la regla *make all*, se compila todo el codigo y se genera el compilador. Al correr la regla *make test*, se corren todos los archivos de testeo y se verifica si estos son reconocidos o no por el compilador sin generación de código. Para compilar un archivo escrito en nuestro lenguaje basta con correr la regla *make run FILE=<path del archivo>*, donde luego de FILE se especifica el path del archivo a compilar.

Las funcionalidades especificas de nuestro lenguaje son las siguientes:

- **Variables:** Tenemos variables enteras, booleanas y de arbol, que pueden ser declaradas en una sentencia o declaradas e inicializadas en una sentencia. Los arboles solo pueden contener enteros. Esto se hace de la siguiente forma:

```
int a;
int b = 2;
bool c;
c = true;
bool d = b == 2;

new AVL t1;
new BST t2;
// Instancio t3 a base de t1
new RBT t3 <- t1
```

- **Constantes:** Nuestro lenguaje soporta constantes enteras y booleanas.

- **Estructuras de control:** Se pueden utilizar estructuras *if*, *if-else*, *while*, *for*. Se utilizan de la siguiente manera:

```
int a = 0;
int b = 100;
if (a == b) {
    //...
}
while (a != b) {
    a = a + 1;
}
for n in (0, 40) {
    a = a + n
}
```

- **Operadores:** Tenemos operadores aritmeticos para operar entre enteros y operadores booleanos para operar ente expresiones booleanas. Estas son las siguientes.

```
a + b // a y b expresiones enteras, devuelve un entero
a - b // a y b expresiones enteras, devuelve un entero
a * b // a y b expresiones enteras, devuelve un entero
a / b // a y b expresiones enteras, devuelve un entero
a % b // a y b expresiones enteras, devuelve un entero
a == b // a y b expresiones enteras, devuelve un booleano
a != b // a y b expresiones enteras, devuelve un booleano
a > b // a y b expresiones enteras, devuelve un booleano
a < b // a y b expresiones enteras, devuelve un booleano
a <= b // a y b expresiones enteras, devuelve un booleano
a > = b // a y b expresiones enteras, devuelve un booleano
a & b // a y b expresiones booleanas, devuelve un booleano
a | b // a y b expresiones booleanas, devuelve un booleano
! a // a expresin booleanas, devuelve un booleano
```

- **Llamados a función de árbol:** Tenemos funciones para operar sobre arboles que son las siguientes:
  - **max t1** : devuelve el entero máximo del árbol.
  - **min t1** : devuelve el entero mínimo del árbol.
  - **height t1** : devuelve la altura del árbol.
  - **present a t1** : devuelve un booleano sobre si el a esta en el árbol o no.
  - **insert t1 a** : inserta a en t1, a debe ser entero.
  - **remove t1 a** : elimina a de t1, a debe ser entero.
  - **reduce even t1** : elimina los elementos pares de t1.

- **reduce odd t1** : elimina los elementos impares de t1.
- **find t1 a** : marca a en el arbol (luego se vera en otro color en el archivo). a debe ser entero.
- **inorder t1** : imprime el recorrido inorder de t1 a un archivo en la carpeta traversals.
- **preorder t1** : imprime el recorrido preorder de t1 a un archivo en la carpeta traversals.
- **postorder t1** : imprime el recorrido postorder de t1 a un archivo en la carpeta traversals.
- **print t1** : imprime t1 en un archivo en la carpeta dots. Este archivo esta en formato *.dot*

3. Desarrollo Del Proyecto

3.1. Front-End

3.1.1. Analisis Léxico

Para el análisis léxico de la entrada se ha utilizado FLEX como generador del analizador léxico. Las expresiones regulares de los lexemas aceptados por nuestro lenguaje se encuentran en el archivo *flex-patterns.l*. En cuanto a las acciones realizadas al *matchear* un lexemas, si se encuentra un entero, constante booleana o variable, se guardan en el estado del compilador.

```
token IntegerPatternAction(const char * lexeme, const int length) {
   LogDebug("IntegerPatternAction: '%s' (length = %d).", lexeme, length);
   yylval.integer = atoi(lexeme);
   return INTEGER;
}

token BooleanPatternAction(const char * lexeme, const int length) {
    LogDebug("BooleanPatternAction: '%s' (length = %d).", lexeme, length);
    bool isTrue = (strcmp(lexeme, "true") == 0);
    yylval.boolean = isTrue;
    return BOOLEAN;
}

token DeclarePatternAction(const char* lexeme, const int length) {
    LogDebug("DeclarePatternAction: '%s' (length = %d).", lexeme, length);
    char * varName = (char*) calloc((length + 1), sizeof(char));
    strncpy(varName, lexeme, length);
    yylval.varname = varName;
    return VARIABLE;
}
```

3.1.2. Análisis Sintáctico

Para el análisis sintáctico del lenguaje se ha utilizado GNU Bison para generar el analizador sintáctico. Las reglas gramáticas del lenguaje se pueden encontrar en el archivo *bison-grammar.y*. Las acciones realizadas al reducir una regla de sintaxis incluirán la creación del árbol de sintaxis abstracta y, si es necesario, la población de la tabla de símbolos y la realización del *type-checking*. El detalle de como se implementa esto se encuentra en la sección de *back-end*. A continuacion se muestra el codigo que se corre al reducir una regla de asignación.

```
Assignment *AssignmentGrammarAction(char *var, Expression *exp, FunctionCall
    *functionCall) {
    LogDebug("\tAssignmentGrammarAction");

    // Busco en tabla de tipos
    struct key key = {.varname = var};
    struct value value;
    if (!symbolTableFind(&key, &value)) {
        LogError("Variable %s undeclared", var);
        exit(1);
    }

    // Checkeo de tipos
    if (value.type != VAR_INT && value.type != VAR_BOOL) {
        LogError("Variable %s is not an integer nor a boolean", var);
        exit(1);
    }

    if (functionCall != NULL && getFunctionCallType(functionCall) !=
        value.type) {
        LogError("Function return cannot be assigned to %s", var);
        exit(1);
    }

    if (exp != NULL && getExpressionType(exp) != value.type) {
        LogError("Expression cannot be assigned to %s", var);
        exit(1);
    }

    // Se actualiza la tabla de tipos
    value.metadata.hasValue = true;
    symbolTableInsert(&key, &value);

    return createAssignment(var, exp, functionCall);
}
```

3.2. Back-End

3.2.1. Árbol de Sintaxis Abstracta

Como fue dicho anteriormente el árbol de sintaxis se crea a medida de que se reducen las reglas de la gramática. Para esto, se ha creado una librería de funciones que permiten crear los nodos asociados a cada símbolo no terminal de la gramática junto a su contenido relevante. Esta librería también contiene funciones para liberar estos nodos. La librería se encuentra en el archivo *tree-utils.c*. Las estructuras de cada nodo del arbol se encuentran en el archivo *abstract-syntax-tree.h*. A continuacion se muestra un ejemplo del nodo del simbolo no terminal asociado a nuestro *for-loop* y sus funciones.

```
// Estructura del nodo
typedef struct {
   char * varname;
   RangeExpression * range;
   Block * block;
} ForStatement;

ForStatement * createForStatement(char * varname, RangeExpression * range,
    Block * block) {
    ForStatement * new = malloc(sizeof(ForStatement));

    new->varname = varname;
    new->block = block;
    new->range = range;

    return new;
}

void freeForStatement(ForStatement *forStatement) {
    if (forStatement == NULL) {
        return;
    }

    freeRangeExpression(forStatement->range);
    freeBlock(forStatement->block);
    free(forStatement);
}
```

3.2.2. Tabla de Símbolos

En nuestro compilador utilizamos una tabla de símbolos minimal. Donde se guarda por cada variable su tipo y si fue instanciada o no con un *scope* global. La implementación de esta tabla se realizo utilizando una tabla de *hashing* para una mayor eficiencia. La implementación de la tabla de simbolos se encuentra en el archivo *symbol-table.c* y la tabla de *hashing* se encuentra en el archico *hashmap.c*. A continuación se muestra la estructura de los elementos de la tabla de símbolos.

```
// Contenido de la tabla
typedef enum VarType {
    VAR_RBT,
    VAR_BST,
    VAR_AVL,
    VAR_INT,
    VAR_BOOL
} VarType;

struct metadata {
    bool hasValue;
};

struct value {
    VarType type;
    struct metadata metadata;
};
```

3.2.3. Type-checking

En nuestro lenguaje existen 5 tipos de datos. Enteros (int), booleanos (bool) y 3 tipos de arboles (AVL, RBT y BST sin balancear), donde solo se pueden agregar enteros a los arboles. Para implementar esto es necesario hacer *type-checking* de expresiones, asignaciones y llamados a función de arboles. Este *checkeo* se realiza cada vez que se evalúa una expresión y se hace un llamado a función de árbol puesto que estas también pueden retornar valores. La implementacion de estas funciones se encuentra en el archivo *bison-actions.c*. Los tipos de datos son los utilizados en la tabla de símbolos. A continuación se muestra la implementación del *checkeo* de tipos para funciones de árbol donde -1 implica que la función no tiene retorno.

```
static int getFunctionCallType(FunctionCall *functionCall) {
    switch (functionCall->type) {
    case MAX_CALL:
    case MIN_CALL:
    case HEIGHT_CALL:
        return VAR_INT;
    case PRESENT_CALL:
        return VAR_BOOL;
    default:
        return -1;
    }
}
```

3.2.4. Generación de Código

Nuestro lenguaje primero compila la entrada a código de Java. La generación de código se corre en el *main* una vez que se pudo crear el árbol de sintaxis sin ningún error. Esta generación es recursiva, arrancando por el nodo programa y recorriendo todos los nodos. El output final es una clase *Main* de Java con un método *Main* que contiene la traducción de nuestro lenguaje a código Java. Las funciones generadoras de código para cada nodo del árbol se encuentran en el archivo *generator.c*. A continuación se muestra la implementación del generador de código para el nodo principal (Programa).

```
void GeneratorProgram(Program *program) {

    Output("import java.io.IOException;\n");
    Output("public class Main {\n");
    Output("public static void main(String[] args) throws IOException {\n");

    // Llamado Recursivo
    GeneratorStatementList(program->statements);

    Output("}\n");
    Output("}\n");
}
```

La función Output permite abstraer el output del programa a donde se precise. En este momento la implementación de la función output es la siguiente:

```
#include <stdarg.h>

static FILE *outputFile;

// El archivo es seteado por main
// Puede ser stdout o un pipe incluso
void SetOutputFile(FILE *file) {
    outputFile = file;
}

void Output(char *format, ...) {
    va_list args;
    va_start(args, format);
    vfprintf(outputFile, format, args);
    va_end(args);
}
```

3.2.5. Runtime

En cuanto al runtime, hemos creado librerías de java para el manejo de nuestros arboles. Esto permite modularizar el código de manera tal que el output del compilador solo llame a los métodos de las clases implementadas. Para persistir los arboles del programa a archivos *.dot* se ha utilizado la librería **graphviz-java**. Decidimos utilizar *maven* para manejar esta dependencia por su facilidad de uso.

Una vez terminada la generación de código, se compila el proyecto de *maven* y se ejecuta con una JVM. Creándose los archivos si el programa de entrada lo requiere. Esto se hace en el *main.c* con el siguiente *snippet*.

```
FILE * file = fopen("./src/backend/domain-specific/src/main/java/Main.java",
    "w");

SetOutputFile(file);

GeneratorProgram(state.program);

fclose(file);

LogInfo("El archivo Main.java fue generado exitosamente.");

// Compile the generated file
system("mvn -f ./src/backend/domain-specific/pom.xml clean compile");

// Run the generated file
system("mvn -f ./src/backend/domain-specific/pom.xml exec:java
    -Dexec.mainClass=\"Main\"");
```

4. Dificultades Encontradas

Hemos encontrado varias dificultades en la realización del proyecto empezando por la concepción de la idea puesto que no teníamos una idea en particular pero luego de un ida y vuelta con la cátedra terminamos eligiendo la idea actual. Esta idea también ha ido mutando con el tiempo puesto que en un principio pretendíamos mostrar los arboles generados en una ventana gráfica utilizando *javafx*. Debido a complicaciones en el uso de esta librería y por sugerencia de la cátedra se decidió por persistir los arboles en archivos.

En cuanto al desarrollo del *front-end* del compilador, no tuvimos grandes problemas ni en la definición de las expresiones regulares ni en la definición de la gramática. La única complicación que nos surgió fueron algunos falsos positivos en el reconocimiento de los programas de prueba. Resolver estos falsos positivos con reglas de gramática era complejo por lo que se decidió resolverlos en el *back-end*.

En cuanto a la implementación del *back-end*, uno de los grandes problemas que encontramos fue el como reconocer el uso de la variable iteradora de nuestro *for-loop*. La acción gramática del *for-loop*, donde se declara la variable iteradora, se ejecutaba despues de las

acciones de las expresiones dentro del bloque de código del loop. Por lo que nos saltaba el error de variable no declarada. Esto fue resuelto guardando los usos de la posible variable iteradora y una vez creado el arbol, verificamos que la variable haya sido creada.

También en un principio queríamos hacer una función reduce que reciba una expresión *lambda* definida por el programador. La distinción entre las expresiones para la función reduce y las expresiones propias del lenguaje nos causo muchos problemas. Por lo tanto decidimos por el momento definir dos usos del reduce con palabras clave para eliminar pares e impares.

5. Futuras Extensiones

En relación a futuras extensiones que se le podrían agregar al proyecto, consideramos las siguientes:

Primero, la posibilidad de agregar más tipos de datos a los arboles, cualquier tipo de dato que sea comparable en Java podría ser ya utilizado por nuestros arboles, ya que utilizan datos genéricos que extiendan Comparable. Solo habría que agregar el tipo de dato dentro de Flex y Bison respectivamente.

Segundo, la posibilidad de incluir todavía mas tipos de arboles en nuestro proyecto y así poder expandir el número de los mismos que le ofrecemos al usuario. Por ejemplo, uno de estos podría ser el BTree(Árbol de tipo B), el cual llevaría una implementación completamente diferente a todos los otros arboles ya implementados.

Tercero, podríamos darle toda la potencia que planificamos para la función reduce, con expresiones *lambda* definidas por el programador.

Por último, brindarle al usuario la posibilidad de poder operar entre árboles. Esto podría ser, por ejemplo, tener la capacidad de borrar un árbol que este dentro de otro, o por lo menos los nodos que coincidan, entre otras.