Proyecto final

*Compiladores 2024-2

1st Jorge Sebastian Tenorio Romero
Ciencia de la Computacion
Universidad de Ingenieria y Tecnologia (UTEC)
Lima, Peru
jorge.tenorio@utec.edu.pe

2nd Gonzalo Daniel Suarez
Ciencia de la Computacion
Universidad de Ingenieria y Tecnologia (UTEC)
Lima, Peru
gonzalo.suarez@utec.edu.pe

***Index Terms*—compilers, kotlin, machine code**

I. Introduction

Kotlin es uno de los lenguajes más modernos y populares de la actualidad. Estrenado en 2011 con propósito de ofrecer un lenguaje expresivo y seguro que se pueda adaptar y trabajar en conjunto con Java. Kotlin es el lenguaje oficial soportado por Google para la creación y desarollo de aplicativos móviles en dicho sistema operativo.

II. Características del lenguaje

- Kotlin presenta un sistema enfocado en manejar nativamente la nulabilidad, por ello minimiza significativamente errores como NUllPointerException.
- Kotlin es respaldado oficialmente por Google, lo cual brinda un alto soporte y su integración en Android Studio, el framework más popular para el desarollo de aplicativos móviles en dicho sistema operativo.
- Debido a su enfoque principal, Kotlin puede interactuarar con código en Java, esto facilita la adaptación y adición de codigo Kotlin a proyectos escritos en Java.
- Kotlin, a la par de Java, permite una sencilla Programación Orientada a Objetos, siendo este uno de los paradigmas más populares en el desarollo de software.

III. Análisis Léxico

El análisis léxico del lenguaje propuesto consiste en extraer los Tokens de la entrada, es decir, en separar e identificar los elementos del lenguaje encontrados en la entrada. Por otro lado, dicho análisis también permite identificar errores, en caso alguna parte de la entrada no sea uno de los tokens del compilador, se indica que existe un error léxico.

Para el análisis léxico se define primero la grámtica del lenguaje y los tokens existentes en esta.

A. Gramática

```
Program ::= VarDeclList FunDeclList

VarDeclList ::= (VarDec)*

FunDeclList ::= (FunDec)*

FunDec ::= fun id "(" [ParamDeclList] ")" [":" Type]
      "{" Body "}"

Body ::= VarDeclList StmtList

ParamDeclList ::= id ":" Type ("," id ":" Type)*

VarDec ::= (var | val) id ":" Type

Type ::= id // tipos de variables: Int, Long y Bool

StmtList ::= Stmt (";" Stmt)*

Stmt ::=
    id "=" CExp |\
    println "(" CExp ")" |
    for "(" id in CExp ".." CExp ")" "{" Body "}" |
    while "(" CExp ")" "{" Body "}" |
    do "{" Body "}" while "(" CExp ")" |
    if "(" CExp ")" "{" Body "}" [else "{" Body "}"]
         |
    return CExp

CExp ::= Exp (("<" | ">" | "<=" | ">=" | "==" | "!="
    ) Exp)*

Exp ::= Term (("+" | "-") Term)*

Term ::= Factor (("*" | "/") Factor)*

Factor ::= id | Num | Bool | "(" Exp ")" | id "(" [
    ArgList] ")"

ArgList ::= CExp ("," CExp)*

Bool ::= true | false
```

B. Tokens

Dada la gramática propuesta, se procede a ennumerar los Tokens existentenes en esta, y adicionalmente un Token de error, el cual indicaría que dicha parte de la entrada no corresponde a ningún Token válido.

1) **PLUS (+)**: Operador de suma.
2) **MINUS (-)**: Operador de resta.
3) **MUL** ($\times$): Operador de multiplicación.
4) **DIV (/)**: Operador de división.
5) **NUM**: Representa un número.
6) **ERR**: Representa un error léxico.
7) **PD (}**): Paréntesis derecho.

Identify applicable funding agency here. If none, delete this.

8) **PI ({)**: Paréntesis izquierdo.
9) **END (END)**: Fin del programa o entrada.
10) **ID**: Nombres de variables o de funciones.
11) **PRINT (PRINT)**: Instrucción de impresión.
12) **ASSIGN (=)**: Operador de asignación.
13) **TWODOT (:)**: Dos puntos.
14) **DOTS (..)**: Rango del FOR.
15) **PC (;)**: Punto y coma.
16) **LT (¡)**: Operador menor que.
17) **LE (¡=)**: Operador menor o igual que.
18) **EQ (==)**: Operador igual que.
19) **NEQ (!=)**: Operador diferente de.
20) **GT (¿)**: Operador mayor que.
21) **GE (¿=)**: Operador mayor o igual que.
22) **IF (IF)**: Palabra reservada para la condicional.
23) **ELSE (ELSE)**: Palabra reservada para la condicional.
24) **WHILE (WHILE)**: Palabra reservada para un bucle 'WHILE' o 'DO WHILE'.
25) **DO (DO)**: Palabra reservada para ejecución de código en un bucle 'DO WHILE'.
26) **COMA (,)**: Representa la coma.
27) **VAR (VAR)**: Palabra reservada para declarar variables.
28) **VAL (VAL)**: Palabra reservada para declarar variables.
29) **FOR (FOR)**: Palabra reservada para bucles 'FOR'.
30) **TRUE (TRUE)**: Literal booleano verdadero.
31) **FALSE (FALSE)**: Literal booleano falso.
32) **RETURN(RETURN)**: Palabra reservada para retornar un valor desde una función.
33) **FUN (FUN)**: Palabra reservada para declarar funciones.
34) **LLD (}**): Llave derecha.
35) **LLI ({)**: Llave izquierda.
36) **IN (IN)**: Palabra reservada para iteraciones en el rango de un bucle FOR.

C. Implementación

```
%% no quitar es mi template
%%aca va el codigo
```

Listing 1. Archivo `nombrearchivo`

```
#ifndef TOKEN_H
#define TOKEN_H

#include <string>

class Token {
public:
    enum Type {
        PLUS, MINUS, MUL, DIV, NUM, ERR, PD, PI, END, ID, PRINT,
        ASSIGN, TWODOT, DOTS, PC, LT, LE, EQ, NEQ, GT, GE, IF, ELSE, WHILE, DO,
        COMA, VAR, VAL, FOR, TRUE, FALSE, RETURN, FUN, LLD, LLI, IN
    };
    Type type;
    std::string text;

    Token(Type type);
    Token(Type type, char c);
    Token(Type type, const std::string& source, int first, int last);

    friend std::ostream& operator<<(std::ostream& outs, const Token& tok);
    friend std::ostream& operator<<(std::ostream& outs, const Token* tok);
};

#endif // TOKEN_H
```

Listing 2. Archivo `token.h`

```
#include <iostream>
#include "token.h"

using namespace std;

Token::Token(Type type):type(type) { text = ""; }
Token::Token(Type type, char c):type(type) { text = string(1, c); }
Token::Token(Type type, const string& source, int first, int last):type(type) {
    text = source.substr(first, last);
}
std::ostream& operator << ( std::ostream& outs, const Token & tok )
{
    switch (tok.type) {
        case Token::PLUS: outs << "TOKEN(PLUS)"; break;
        case Token::MINUS: outs << "TOKEN(MINUS)"; break;
        case Token::MUL: outs << "TOKEN(MUL)"; break;
        case Token::DIV: outs << "TOKEN(DIV)"; break;
        case Token::NUM: outs << "TOKEN(NUM)"; break;
        case Token::ERR: outs << "TOKEN(ERR)"; break;
        case Token::PD: outs << "TOKEN(PD)"; break;
        case Token::PI: outs << "TOKEN(PI)"; break;
        case Token::END: outs << "TOKEN(END)"; break;
        case Token::ID: outs << "TOKEN(ID)"; break;
        case Token::PRINT: outs << "TOKEN(PRINT)"; break;
        case Token::ASSIGN: outs << "TOKEN(ASSIGN)"; break;
        case Token::PC: outs << "TOKEN(PC)"; break;
        case Token::LT: outs << "TOKEN(LT)"; break;
        case Token::LE: outs << "TOKEN(LE)"; break;
        case Token::EQ: outs << "TOKEN(EQ)"; break;
        case Token::NEQ: outs << "TOKEN(NEQ)"; break;
        case Token::IF: outs << "TOKEN(IF)"; break;
        case Token::ELSE: outs << "TOKEN(ELSE)"; break;
        case Token::WHILE: outs << "TOKEN(WHILE)"; break;
        case Token::COMA: outs << "TOKEN(COMA)"; break;
        case Token::DO: outs << "TOKEN(DO)"; break;
        case Token::VAR: outs << "TOKEN(VAR)"; break;
        case Token::VAL: outs << "TOKEN(VAL)"; break;
        case Token::FOR : outs << "TOKEN(FOR)"; break;
        case Token::TRUE : outs << "TOKEN(TRUE)"; break;
        case Token::FALSE : outs << "TOKEN(FALSE)";
```

```
            break;
        case Token::RETURN : outs << "TOKEN(RETURN)"
            ; break;
        case Token::FUN : outs << "TOKEN(FUN)";
            break;
        case Token::TWODOT : outs << "TOKEN(TWODOT)"
            ; break;
        case Token::DOTS : outs << "TOKEN(DOTS)";
            break;
        case Token::GE : outs << "TOKEN(GE)"; break;
        case Token::GT : outs << "TOKEN(GT)"; break;
        case Token::LLI : outs << "TOKEN(LLI)";
            break;
        case Token::LLD : outs << "TOKEN(LLD)";
            break;
        case Token::IN : outs << "TOKEN(IN)"; break;
        default: outs << "TOKEN(UNKNOWN)"; break;
    }
    return outs;
}
std::ostream& operator << ( std::ostream& outs,
    const Token* tok ) {
    return outs << *tok;
}
```

Listing 3. Archivo `token.cpp`

IV. Análisis Sintático

El análisis semántico esta directamente relacionado a la gramática, el llamado parser se encarga de analizar los tokens e identificar que estos respeten la gramática propuesta. Es decir, todos los tokens de la entrada forman parte de una derivación de las reglas de la gramática, en caso contrario, se daría un error sintáctico. Este análisis también tiene como objetivo la genración de un árbol sintáctico, este árbol representa la gramática y su correcta derivación de los tokens.

A. Implementación

```
#ifndef PARSER_H
#define PARSER_H

#include "scanner.h"
#include "exp.h"

class Parser {
private:
    Scanner* scanner;
    Token *current, *previous;
    bool match(Token::Type ttype);
    bool check(Token::Type ttype);
    bool advance();
    bool isAtEnd();
    list<Stm*> parseStmList();
    Exp* parseCExp();
    Exp* parseExpression();
    Exp* parseTerm();
    Exp* parseFactor();
public:
    Parser(Scanner* scanner);
    Program* parseProgram();
    Stm* parseStatement();
    StatementList* parseStatementList();
    VarDec* parseVarDec();
    VarDecList* parseVarDecList();
    Body* parseBody();
    FunDecList* parseFunDecList();
    FunDec* parseFunDec();

};

#endif // PARSER_H
```

Listing 4. Archivo `parser.h`

V. Análisis semántico

El análisis semántico es una de las fases clave en la implementación de un compilador, y su objetivo es garantizar que el programa sea lógicamente consistente según las reglas del lenguaje de programación. Para esto se abarcó:

A. Verificación de Tipos

El compilador revisa que las operaciones sean válidas con respecto a los tipos de datos de las variables y expresiones. Para eso resultó necesario el correcto manejo de los módulos de imp-type-checker, type-visitor y un environment.

B. Comprobación de Excepciones

También fue crucial verificar que las estructuras de control (como condicionales, bucles y excepciones) tengan sentido, como en el caso de los condicionales y bucles que se aseguran de que las expresiones que controlan las estructuras de flujo, como las condiciones de if o los while, sean de tipo booleano.

C. Verificación de expresiones y valores

Se trató de que las expresiones booleanas sean correctamente evaluadas, especialmente si involucran operadores lógicos o de comparación. También fue importante que los valores sean consistentes con los tipos que se esperan en el contexto en el que se usan.

VI. Generación de código

El objetivo es generar código intermedio SVM que sirva como una representación más cercana a la máquina. Para eso se tiene en cuenta que:

- El código intermedio se genera a partir del Árbol de Sintaxis Abstracta (AST), que es una representación jerárquica de la estructura del programa. Cada nodo del AST corresponde a una instrucción o una operación en el código SM.
- El compilador debe decidir cómo gestionar las variables y la memoria. En el caso de código SM, esto implica la asignación de registros o posiciones de memoria en la pila para las variables del programa.
- Durante la generación, es necesario seleccionar las instrucciones correctas para las operaciones aritméticas y lógicas, considerando cómo se representan estas operaciones a nivel de máquina.

VII. Implementación

Implementando todo usando C++, se trabajó en una simplificación general del lenguaje asignado que fuese capaz de soportar declaración y asignación de variables, operaciones, estructuras condicionales, la estructura repetitiva 'for' y el manejo de funciones. A nivel de parsing se optó por una

estructura de sentencias delimitadas por punto y coma que permite la sintaxis Kotlin, aunque no de manera obligatoria. Como agregado adicional se implementaron las estructuras repetitivas 'while' y 'do while', los cuales tienen sus propias clases y manejos para distintos niveles en que los trabaja el compilador. Por otro lado, se simplificaron algunas implicancias en los tests como la no diferenciación entre 'val' y 'var', uno como constante y otro como variable, siendo ambos manejados como una misma parte de la declaración de variable.

VIII. Casos de prueba

A. Test 1

1) Código: Input1

```
fun main() {
    var x: Int
    val y: Int
    val z: Long

    x = 1;
    y = 10;
    z = 1000000;
    x = 20;

    println(x)
    println(y)
    println(z)
}
```

Listing 5. Archivo `input1.kt`

2) Resultados: input1.txt.sm

```
start: skip
enter 0
alloc 0
mark
pusha Lmain
call
halt
Lmain: skip
enter 4
alloc 3
push  1
storer 1
push  10
storer 2
push  1000000
storer 3
push  20
storer 1
loadr 1
print
loadr 2
print
loadr 3
print
```

Listing 6. input1.txt.sm

B. Test 2

1) Código: Input2

```
fun main() {
    var x: Int
    val y: Int

    x = 5;
    y = 10;

    if (x > y) {
        println(x)
    } else {
        println(y)
    }
}
```

Listing 7. Archivo `iput3.kt`

2) Resultados: input2.txt.sm

```
start: skip
enter 0
alloc 0
mark
pusha Lmain
call
halt
Lmain: skip
enter 4
alloc 2
push  5
storer 1
push  10
storer 2
loadr 1
loadr 2
gt
jmpz L0
loadr 1
print
goto L1
L0: skip
loadr 2
print
L1: skip
```

Listing 8. input2.txt.sm

C. Test 3

1) Código: Input3

```
fun main() {
    var x: Int

    x = 1;

    for (i in 0..9) {
        x = x + i
    };

    println(x)
}
```

Listing 9. Archivo `input3.kt`

2) Resultados: input3.txt.sm

```
start: skip
enter 0
alloc 0
mark
pusha Lmain
call
halt
Lmain: skip
enter 5
```

```
alloc 1
push 1
storer 1
alloc 1
push 0
storer 1
L0: skip
loadr 1
push 9
sub
jmpn L1
loadr 1
load 4
add
storer 1
loadr 1
push 1
add
storer 1
goto L0
L1: skip
loadr 1
print
```

Listing 10. input3.txt.sm

D. Test 4

1) *Código:* Input4

```
fun suma(a: Int, b: Int): Int {
    return a + b
}

fun main() {
    var x: Int
    var y: Int

    x = 1;
    y = 20;

    println(suma(x, y))
}
```

Listing 11. Archivo `input4.kt`

2) *Resultados:* input4.txt.sm

```
start: skip
enter 0
alloc 0
mark
pusha Lmain
call
halt
Lsuma: skip
enter 2
alloc 0
loadr -4
loadr -3
add
storer -5
return 5
Lmain: skip
enter 5
alloc 2
push  1
storer 1
push  20
storer 2
alloc 1
loadr 1
loadr 2
mark
pusha Lsuma
call
print
```

Listing 12. input4.txt.sm

IX. Conclusiones

- El proyecto logró mayormente cumplir con su objetivo de implementar un compilador que traduce código Kotlin a un código intermedio (SM), lo que demuestra que es factible generar representaciones intermedias de un lenguaje de alto nivel en un lenguaje de bajo nivel como C++.
- El intérprete construido permitió validar la salida del compilador ante los casos propuestos
- Se requirió de una comprensión a detalle de la semántica general del lenguaje y una estrategia efectiva para representarlas en código intermedio.
- El proyecto brindó una comprensión más profunda de los conceptos fundamentales de los compiladores, tales como el análisis léxico, sintáctico y semántico, y cómo estos se aplican en la creación de compiladores reales.

References

[1] A. Aho, et al., "Compilers: Principles, Techniques, and Tools," 2nd ed., Pearson, 2011, ISBN: 10-970-26-1133-4.

[2] W. Appel, "Modern Compiler Implementation in Java," 2nd ed., Cambridge University Press, 2002.

[3] K. C. Louden, "Compiler Construction: Principles and Practice," Thomson, 2004.

[4] K. C. Louden, "Lenguajes de Programación," Thomson, 2004.

[5] B. Teufel and S. Schmidt, "Fundamentos de Compiladores," Addison Wesley Iberoamericana, 1998.

[6] JetBrains, "Kotlin Documentation." Disponible en: https://kotlinlang.org/docs/home.html. Último acceso: 25 de noviembre de 2024.