2016

Tango: A Spanish-Based Programming Language

Ashley Zegiestowsky
Butler University, azegiest@butler.edu

BUTLER UNIVERSITY HONORS PROGRAM

Honors Thesis Certification

Please type all information in this section:

Applicant Ashley Michelle Zegiestowsky
(Name as it is to appear on diploma)

Thesis title Tango: A Spanish-Based Programming Language

Intended date of commencement May 7, 2016

Read, approved, and signed by:

Thesis adviser(s) _____ April 21, 2016
Date

_____ Date

Reader(s) _____ 20 April 2016
Date

Alex Quintanilla 21 April 2016
Date

Certified by _____ 4/22/16
Director, Honors Program Date

For Honors Program use:

Level of Honors conferred: University _____

Departmental _____

Tango: A Spanish-Based Programming Language

A Thesis

Presented to the Department of Computer Science & Software Engineering

and

The Department of Modern Languages, Literatures, and Cultures

College of Liberal Arts and Sciences

and

The Honors Program

of

Butler University

In Partial Fulfillment

of the Requirements for Graduation Honors

Ashley Michelle Zegiestowsky

April 20, 2016

CONTENTS

PART I:

Tango: A Spanish-Based Programming Language

(English Version)

Section 1: Introduction

The purpose of this thesis is a two-part project. The first part of the project deals with the creation of my own Spanish-based programming language, Tango, using Spanish key words (instead of English key words). The second part of the project relates to the design and implementation of a compiler that follows the grammar rules outlined in the Tango language in order to successfully lexically analyze, parse, semantically analyze, and generate code for Tango.

This project, to write my own Spanish programming language, provided me with a way to combine both of my fields of study, Computer Science and Spanish, under one intriguing endeavor. I came across many challenges in relation to both Computer Science and the Spanish language, but it was inspiring to know that what I was creating was something very few people have actually done and implemented. Before starting this project, I knew very little about the nitty-gritty details that went into designing a programming language and compiler. From a high-level perspective, I learned about the importance of context-free grammars and the different phases that make up a compiler, which will be explained in further detail later.

Moving forward, the structure of this thesis begins with a description of the specific goals achieved in the Tango language, an explanation and brief examples of the Tango Grammar, a high-level overview of the compiler design and data structures used, concluding with ideas for future work and helpful advice. The full grammar, list of keywords, and source code for the compiler can be found in the Appendices.

Section 2: Specific Goals

In creating the Tango programming language as well as the accompanying compiler, there are five main goals around which I centered my overall thesis. These goals are listed as follows:

1. To create a user-friendly programming language utilizing key words and a syntactic structure that closely resembles the Spanish language.
2. To design the language with a similar technical feel to that of some of the most widely used languages in both an educational and professional environment, such as Java and C++.
3. To allow the compiler to read and interpret both keywords and variable identifiers with special characters native to the Spanish language, such as accents and tildes above certain letters.
4. To provide a compiler that cross-compiles to run against the Java compiler instead of directly generating machine code.
5. To assure that the Tango language is Turing complete, or computationally universal.

The first goal forms an essential backbone to the entire project as a whole. There are over thousands of different programming languages utilized and available across a wide array of countries and cultures, and new languages and frameworks are being created everyday. However, a huge majority of these programming languages are English-based, meaning the keywords and syntactic structure are based on the English language [6]. Less than a handful of Spanish-based programming languages have been created including (but not limited to):

- GarGar, a Spanish procedural programming language based on Pascal for learning purposes [6].
- Latino, a language with completely Spanish-based syntax [6].
- RoboMind, an educational programming language available in multiple languages (including Spanish) [6].

The purpose and goal behind creating Tango is to contribute to the growing technical community that spans across many languages and cultures. With the creation of this prototype version of a Spanish-based programming language, the educational and professional reach of such a tool could prove to be very beneficial in engaging and inspiring both young minds and experienced professionals within the field of technological development.

The second of the above-mentioned goals relates closely to the first goal in the sense of the basic design and feel of the programming language. It is essential that Tango is both intuitive to a native Spanish speaker, yet also compatible from a logical and syntactical standpoint to other more commonly used programming languages.

The third goal, which deals with the handling of special characters, proves to be highly unique to Tango. Unlike the English language, the Spanish language employs the use of accents and tildes above select letters forming characters that do not exist in English. These accents and/or tildes are essential to the language and are crucial to the meaning of certain words. The lack of an accent or tilde

could change the meaning of the word entirely. Therefore, allowing for the compiler to adequately scan and process the possibility of these special characters is vital to the correctness and representation of the Spanish language as demonstrated in the Tango programming language.

The fourth goal mentioned pertains to a more technical implementation of compiler design. When designing the compiler, I chose to cross-compile to Java instead of generating machine code directly in the code generation phase. This decision, given the relatively short time provided to complete such an ambitious project, allowed me to create a more verbose and comparatively functioning programming language. If I had chosen to implement a compiler that generated machine code, the final product would be more limited both in scope and overall functionality. Another benefit to choosing to cross-compile to Java is that the Java compiler is machine independent. Also important to note, the target language is Java and the compiler is written in Java. However, these two choices are independent of one another. More specifics of the compiler design and implementation will be discussed later.

The fifth and final goal assures that the Tango language will be Turing complete, meaning computationally universal. A language is considered Turing complete if it can be used to simulate any single-taped Turing machine [9]. Moreover, a "Turing machine can do everything that a real computer can do" [9]. A universal language does not have to be complex [2]. In the case of Tango, the language itself is not extremely complex, but it does meet the requirements to be considered Turing complete.

Overall, these five main goals helped drive and inspire both the research and implementation process necessary to successfully create a prototype programming language based primarily on the Spanish language.

Section 3: Context-Free Grammar (CFG)

Context-Free Grammar and Tango

The grammar for the Tango programming language is a combination of original productions and recognized production patterns from outside resources. The basic definition of a context-free grammar is as follows:

> A grammar consists of a collection of substitution rules, also called productions. Each rule appears as a line in the grammar, comprising a symbol and a string separated by an arrow. The symbol is called a variable, or nonterminal. The string consists of variables and other symbols called terminals. One variable is designated as the start variable [9].

The purpose of a context-free grammar is to provide rules from which a "syntactically valid string of terminals" can be generated [7]. The process of generating a valid string of terminals is referred to as a derivation that can be described as "a series of replacement operations that shows how to derive a string of terminals from the start symbol" [7]. This same information can be represented pictorially with the use of a parse tree (which will be utilized later when providing sample productions) [9].

There are two main types of derivations: rightmost derivations and leftmost derivations. For the purposes of simplicity and consistency, the select examples shown will be utilizing a leftmost derivation meaning that "in every step of the derivation, the leftmost nonterminal" is selected for replacement [5]. Furthermore, the Tango grammar can be classified as LL(1) which stands for the leftmost derivation when the input is scanned from left to right with one-token

look ahead. To further clarify, a grammar is said to be LL(1) given that, "any two rules defining the same nonterminal must have disjoint selection sets [1]. Meeting the condition to declare a grammar LL(1) is crucial in order to construct a recursive descent parser which will be further explained in Section 4: Compiler Design.

Spanish Language Influence & Nuances in Grammar

Other important aspects of the Tango grammar worth mentioning relate to the selection of key words and syntactic structures in regards to the Spanish language. Three main challenges occurred when selecting key words and syntax:

1. How should the unpredictability of masculine or feminine nouns or descriptive keywords be handled?
2. What verb tense should be used when using verb-like keywords?
3. How should the overall syntax be structured in a way that closely resembles the overall structure of the Spanish language?

As for question one, a simple solution was utilized in order to account for descriptive keywords with different character endings depending upon the object being described. For example, in the Spanish language masculine words end in the letter 'o' where as feminine words in the letter 'a'. Since these word endings are dependent upon the noun or object being described, this created a problem of consistency when selecting keywords. However, instead of such words ending in an 'o' or 'a', all key words that fit this criteria end with the symbol '@'. The symbol '@' has become more commonly used and accepted within the Spanish community for words that could be either masculine or feminine. Some of the keywords for the Tango language that fall under this

category include: `nul@` (null), `vaci@` (void), `públic@` (public), `nuev@` (new), `ciert@` (true), `fals@` (false). Note, the word in parenthesis following the Spanish key word is an English equivalent for reference for those unfamiliar with the Spanish language.

As for the second question regarding verb tense for verb-like keywords, yet another simple solution was implemented. In most English-based programming languages examples of verb-like keywords include words such as `do`, `return`, and `print`, to list a few [10]. In English, these verbs are in a command form (which there is only one kind of command conjugation for each verb regardless of the subject or audience). However, in Spanish, there are multiple different ways to conjugate the command form of a verb and even more differences amongst different countries and dialects. In order to establish a consistent representation of verb-like keywords that would be understood by all Spanish speakers, the infinitive form of the verb serves as the best method of representation. For example, the following are Spanish verb-like keywords present in the Tango programming language: `hacer` (do), `regresar` (return), `imprimir` (print). Again, the word in parenthesis, is the English equivalent to the Spanish key word.

As for the third and final question, mirroring the Spanish language from a syntactical perspective proved to be the most difficult to emulate. However, one specific way in which this is prevalent in the grammar is the placement of the access identifiers in relation to a function definition. In the English language, adjectives are typically placed *before* the noun they are describing; take for example the phrase, "the long red dress". In the Spanish language, adjectives are

typically placed *after* the noun they are describing: take for example, the phrase, "el vestido largo y rojo" (which would directly translate to English as "the dress long and red"). This same principle can be seen in the syntax relating to a function definition demonstrated in the figure on the following page:

Example Java Function Definition (English):

```
public void func_name (int param1, int param2) {…}
```

Example Tango Function Definition (Spanish):

```
func func_name públic@ vaci@ (ent param1, ent param2) {…}
```

Figure 3.1: Note the differences between the two function definitions are very subtle but reflect the structural nuances of the Spanish language.

Sample Tango Productions and Derivations

This section will walk through a simple example of productions and derivations using Tango's grammar. The full grammar and list of keywords (as well as their relative English equivalent) are listed in detail in Appendix A.

As mentioned at the beginning of this section, Tango's grammar is a mix of original productions and predefined productions from outside sources. Henceforth, the Tango grammar utilizes portions of the calculator grammar detailed in Michael Lee Scott's *Programming Language Pragmatics* [7]. Select rules, or productions, are outlined and displayed in the figure below:

stmt → id = expr;
expr → term term_tail
term_tail → add_op term term_tail | *ε*
term → factor factor_tail
factor_tail → mult_op factor factor_tail | *ε*
factor → (expr) | id | number
add_op → + | -
mult_op → * | /

Figure 3.2: Grammar productions (rules) from the LL(1) calculator grammar [7].

FIRST

stmt{ id }
expr{ (, id, number }
term_tail { +, -}
term { (, id, number }
factor_tail { *, / }
factor { (, id, number }
add_op { +, - }
mult_op { *, / }

FOLLOW

id { +, -, *, /, =, id}
number { +, -, *, /,), id }
({ (, id, number }
) { +, - , *, /, id}
= { (, id, number }
+ { (, id, number }
- { (, id, number }
* { (, id, number }
/ { (, id, number }
expr {), id, ; }
term_tail {), id }
term { +, -,), id }
factor_tail { +, -,), id }
factor { +, -, *, /,), id }
add_op { (, id, number }
mult_op { (, id, number }

PREDICT

1. stmt−→id = expr {id}
2. expr −→ term term_tail {(, id, number}
3. term tail −→ add_op term term_tail {+, -}
4. term tail −→ {), id}
5. term −→ factor factor_tail {(, id, number}
6. factor tail −→ mult_op factor factor_tail {*, /}
7. factor tail −→ {+, -,), id}
8. factor−→ (expr) {(}
9. factor−→ id {id}
10. factor −→ number {number}
11. add_op−→ + {+}
12. add_op−→ - {-}
13. mult_op−→ * {*}
14. mul_top−→ / {/}

Figure 3.3: First, Follow, & Predict Sets for the grammar productions in Figure 3.2 [7].

stmt → id = expr;
stmt → x = expr;
stmt → x = term term_tail;
stmt → x = factor factor_tail term_tail;
stmt → x = 5 factor_tail term_tail;
stmt → x = 5 term_tail; (* factor_tail → ε)
stmt → x = 5; (* term_tail → ε)

Figure 3.4: Leftmost derivation for the string, `x=5;`

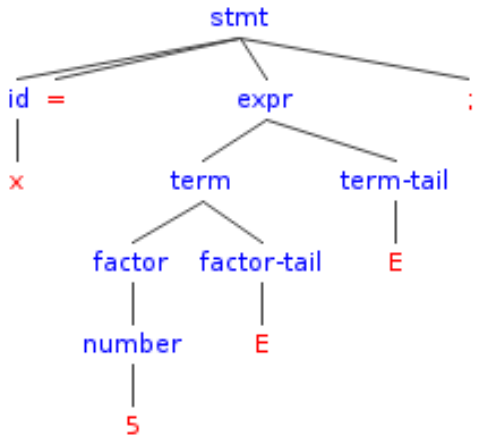

Figure 3.5: Parse Tree for the string, `x=5;` [8].
*Note: E stands for ε, the empty string

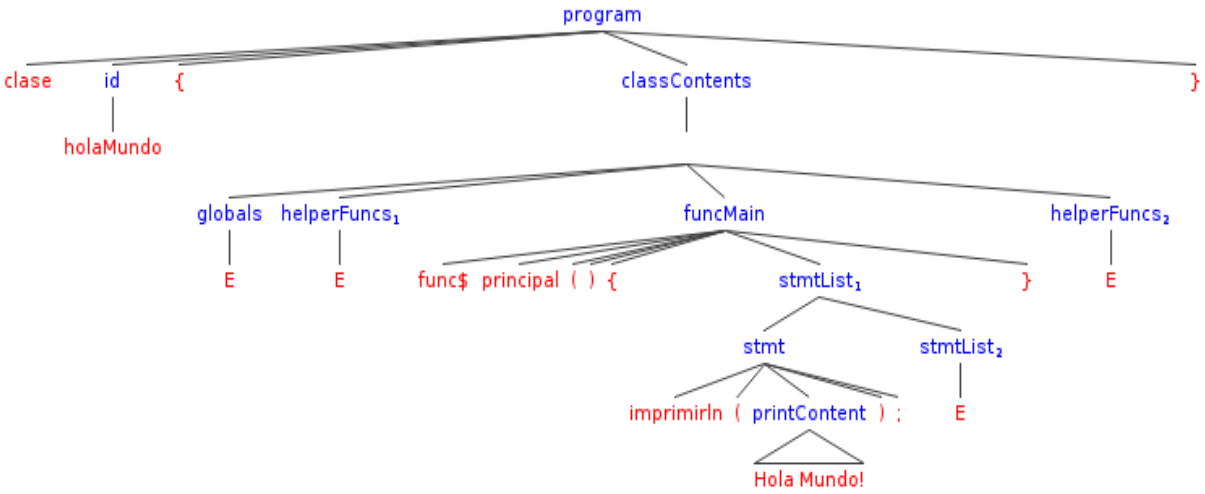

Figure 3.6: Parse tree for the sample Hello World program as seen in Appendix B [8].
*Note: the E stands for ε, the empty string

As mentioned earlier, only a small portion of the Tango grammar is illustrated in the above figures (Figure 3.2-3.6). To see the complete grammar and list of keywords, refer to Appendix A.

Section 4: Compiler Design

The compiler design and implementation comprise a large portion of this thesis and are closely related to the grammar outlined in Section 3. The compiler construction can be broken into four different phases: lexical analyzer (scanner), parser, semantic analyzer, and code generator [2]. The lexical analyzer, or scanner, scans the source program character by character "recognizing which strings of symbols from the source program represent a single entity, or token" [2]. The lexical analyzer also identifies and categorizes the tokens according to their value in relation to the rest of the program. Some of the token types present in the Tango scanner include: keywords, variable identifiers, numeric values, arithmetic operators, special characters, etc.

The second phase is the parser. Since the Tango language has an LL(1) context-free grammar, a recursive descent parser is the method utilized in completing this phase of the compiler. The parser will check for "proper syntax, issue appropriate error messages, and determine the underlying structure of the source program" [1]. The recursive descent parser illustrates "top-down (predictive) parsing" which relates directly to the grammar [7].

The third phase consists of semantic analysis. The semantic analyzer is intertwined with the parser. Whereas the parser checks the program for syntactic correctness according to the grammar rules, the semantic analyzer checks data types and other necessary checks that can not be performed by the parser alone [1].

The fourth and final phase is code generation. During the code generation phase, a traditional compiler translates the successfully parsed tokens or syntax

trees into "machine language (binary) instructions, or to assembly language" [2]. As mentioned previously in Section 2: Specific Goals, the Tango compiler does not directly generate machine code. In fact, the tango compiler, in the code generation phase, generates equivalent Java code, which is then run against the Java compiler. The reasoning for choosing to cross compile to Java is explained in further detail in Section 2.

If an error occurs during any phase of the compiler (lexical scanner, parser, semantic analyzer, or code generator), the compiler terminates and an error message is displayed in the console with the line number and a potential error message depending on the error.

Code Structure & Data Structures

The overall structure of the compiler is a Java based application with five classes:

- *TangoCompiler.java* – This is the main class from which the program runs. When the program starts, the program prompts the user to enter the name of an input file to be processed. Instances of the *TangoScanner* and *TangoParser* objects are instantiated in order to begin processing the source program.
- *TangoScanner.java* – This class constitutes the lexical analysis, or scanner, phase of the compiler. The *TangoScanner* class utilizes instances of the *Token* class in order to create an array of tokens. Each time the scanner recognizes a new Token, a new Token object is created and added to the array of tokens which utilizes the ArrayList data structure.

- *Token.java* – This class contains all of the details and data related to the different token types available (including keywords) in the Tango language. A mix of simple arrays and single line functions are utilized inside the *Token* class. The array of Token objects is then processed in the *TangoParser* in which some of the simple functions created in the Token are used to perform necessary checks when moving through the parsing process. The *Token* class is also where the symbol table is stored which uses the Hash Table data structure.
- *TangoParser.java* – This class is the most intensive portion of the compiler. The *TangoParser* parses the array of tokens produced by the scanner using a recursive descent parser. Therefore, recursion is employed in order to move through the token stream. Semantic analysis also occurs during this phase in which stack data structures are utilized. The *CodeGenerator* class is also instantiated inside of the parser and recursively generates Java code as the tokens are successfully parsed.
- *CodeGenerator.java* – This class uses a FileWriter in order to write the equivalent Java code to a new file. The class consists of a multitude of void functions that are called within the recursive descent parser in order to successfully write to a file the newly generated code that can then be compiled and run against the java compiler using the javac and java command.

Section 5: Conclusion

Future Work

After investing close to eight months on the creation of the Tango programming language and compiler, I am proud to say that I have produced a functional, yet limited, prototype language. If I had more time to invest in this project, there is plenty of work left in order to make Tango a fully functional and bug free programming language that could be utilized primarily for educational purposes. Some of the additions and improvements, I would integrate into the language would include:

- *Expand the Functionality* – With such limited time, I was only able to implement some of the most basic functionality. Integrating more complex data structures, object-oriented principles, and additional libraries would both complement and improve upon the existing source code.
- *Optimization & Debugging* – Very little thought or effort was put into optimizing both performance and memory when implementing the compiler. This would be an important improvement in order to truly test the power and limitations of the compiler.
- *Modify Code Generation* – As detailed in both Section 2 & Section 4, the current complier cross compiles to Java by generating equivalent Java code that is run against the Java compiler. By modifying the code generation phase to directly generate machine code instead of Java code would allow the Tango language to no longer be dependent upon the Java compiler. Although this may increase efficiency and performance

for the Tango compiler, this implementation would limit Tango to a fixed platform.

- *Interactive Website* – It would be beneficial as well to create an online platform or downloadable resource to which the public could access freely and directly in order to both write and run Tango programs. Along with this resource, creating a simple tutorial to aid users in programming in the Tango language with a link to the complete documentation would greatly complement all of the work that has been put forth in creating this project.

Helpful Advice

To anyone interested in completing a similar project, there are several tidbits of advice that are helpful and important to keep in mind when undergoing a project of this magnitude. First of all, make a timeline of important goals and milestones, and stick to it! Although this is a simple and obvious strategy when undergoing any type of project, it becomes even more crucial when dealing with a large code base with lots of moving parts.

Secondly, the grammar, derivations, and syntax trees are just as important as the actual code written to implement the compiler. If your grammar has errors (such as not meeting LL(1) criteria or highly ambiguous), then your compiler will have errors (making it even more difficult to implement). It is easier to fix an error when it is still just a grammar rule rather than when it has already been faultily integrated into the compiler. Lastly, start small and build up from there. It is tempting to want to implement everything at once. However, start with the

basic functionality and then continue to modify and expand the language and compiler accordingly. Those are some of the pieces of advice that I would give to anyone with the desire to delve into compiler theory and design.

Whether or not the Tango programming language will ever be used or viewed outside the scope of this undergraduate thesis does not equate to the success or failure of the final product produced. The learning curve, work ethic, and technical knowledge I gained from completing this project cannot be monetized. Nevertheless, I have attained a new-found pride in the progress I have made in regards to the Tango language and developed a well-earned confidence in my technical abilities as a whole. The Tango programming language demonstrates a small but vital attempt to widen the reach of technological advancement across language and cultural barriers beginning in the educational realm and edging towards a professional environment. The language itself is young and in need of maturation and finesse. If nothing else, Tango can serve as a jumping off point and inspiration into the world of Spanish-based programming languages.

Appendix A: List of Keywords & Full Grammar

List of Keywords

Tango Keyword	Java Equivalent
ent	int
dec	double
cadena	String
lista	[]
bool	Boolean
ciert@	true
fals@	false
nuev@	new
si, sino si, sino	if, else if, else
para	for
mientras, hacer mientras	while, do while
clase	class
func$ principal()	public static void main(String [] args)
estátic@	static
vaci@	void
públic@	public
nul@	null
regresar	return
escáner	scanner
imprimpirln	println
imprimir	print
sigEnt	nextInt
# (single line comment)	/ / (single line comment)

*Both wordrefernce.com [4] & The Oxford Spanish Dictionary [3] were referenced when selecting the appropriate Spanish keyword.

Full Grammar

****High-Level Productions****

program → *clase* id accessMod { classContents }

accessMod → *públic@*

classContents → funcMain

funcMain → *func$ principal*() { stmtList }

stmtList → stmt stmtList | ε

Library Call Productions

stmt → *imprimirln*(printContent);

printContent → "stringValue" | id

Declaration & Assignment Productions

stmt → id = expr;

stmt → dataType id decTail;

decTail → = expr | ε

dataType → *ent*

dataType → *dec*

dataType → *cadena*

dataType → *bool*

expr → term termTail

expr → boolOp

termTail → addOp term termTail | ε

term → factor factorTail

factorTail → multOp factor factorTail | ε

factor → (expr)

factor → id

factor → number

addOp → +

addOp → -

multOp → *

multOp → /

boolOp → *ciert@*

boolOp → *fals@*

If Statement Productions

stmt → *si* (condition) { stmtList } siTail

siTail → *sino* sinoTail | ε

sinoTail → { stmtList } | *si* (condition) { stmtList } siTail

condition → expr conditionTail

conditionTail → compOp expr | ε

compOp → == | != | > | < | >= | <=

While Loop Productions

stmt → mientras (condition) { stmtList }

stmt → hacer { stmtList } mientras (condition);

First, Follow Predict Sets

*separated by similar grammar concepts for organization

KEY:

Red = keyword / terminal

Blue = terminal

Normal = non-terminal

****HIGH LEVEL SETS****

FIRST:

program { clase }

accessMod { públic@ }

classContents { func$ }

funcMain { func$ }

stmtList { mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln }

stmt { mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln }

FOLLOW:

classContents { '}' }

funcMain { '}' }

stmtList { '}' }

stmt { '}', mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln }

PREDICT:

program → clase id accessMod { classContents } { clase }

accessMod → públic@ { públic@ }

classContents → funcMain { func$ }

funcMain → func$ principal() { stmtList } { func$ }

stmtList → stmt stmtList { mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln}

stmtList → { '}' }

****LIBRARY CALL SETS****

FIRST:

stmt { imprimirln } *limited to FIRST set of library call productions

printContent { “, id }

FOLLOW:

printContent { ‘)’ }

PREDICT:

stmt → imprimirln(printContent); { imprimirln }

printContent → “ stringValue “ { “ }

printContent → id { id }

****DECLARATION & ASSIGNMENT SETS****

FIRST:

stmt {id, ent, dec, cadena, bool} *limited to FIRST set of declaration productions

dataType { ent, dec, cadena, bool }

decTail { =, }

expr { ‘(‘, id, number, ciert@, fals@ }

term { ‘(‘, id, number}

termTail { +, -, }

factor { ‘(‘, id, number }

factorTail { *, /, }

addOp { +, - }

multOp { *, / }

FOLLOW:

id { =, *, /, +, -, ;, ‘)’ }

number { =, *, /, +, -, ;, ‘)’ }

= { ‘(‘, id, number, ciert@, fals@ }

({ ‘(‘, id, number, ciert@, fals@ }

) { *, /, +, -, ;, ‘)’ }

+ { ‘(‘, id, number }

- { ‘(‘, id, number }

* { ‘(‘, id, number }

/ { ‘(‘, id, number }

ciert@ { ;, ‘)’ }

fals@ { ;, ‘)’ }

ent { id }

dec { id }

cadena { id }

bool { id }

stmt { ‘}’, mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln}

dataType { id }

decTail { ; }

expr { ;, ‘)’ }

term { +, - , ;, ‘)’ }

termTail { ;, ‘)’ }

factor { *, /, + , -, ;, ‘)’ }

factorTail { +, -, ;, ‘)’ }

addOp { ‘(‘, id, number}

multOp { ‘(‘, id, number}

PREDICT:

stmt → id = expr; {id}

stmt → dataType id decTail; { ent, dec, cadena, bool }

dataType → ent { ent }

dataType → dec { dec }

dataType → cadena { cadena }

dataType → bool { bool }

decTail → = expr { = }

decTail → { ; }

expr → term termTail { ‘(‘, id, number }

expr → boolOp { ciert@, fals@ }

termTail → addOp term termTail { +, - }

termTail → { ;, ')' }

term → factor factorTail { '(', id, number }

factorTail → multOp factor factorTail { *, / }

factorTail → { +, -, ;, ')' }

factor → (expr) { (}

factor → id { id }

factor → number { number }

addOp → + { + }

addOp → - { - }

multOp → * { * }

multOp → / { / }

boolOp → ciert@ { ciert@ }

boolOp → fals@ { fals@ }

****IF STATEMENT SETS****

FIRST:

stmt { si } *limited to FIRST set of if statement productions

siTail { sino }

sinoTail { '{', si }

condition { '(', id, number, ciert@, fals@ }

conditionTail { == , != , > , < , >= , <= }

compOp { == , != , > , < , >= , <= }

FOLLOW:

stmt { '}', mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln }

siTail { si, '(', id, number, ciert@, fals@, imprimln, '}' }

sinoTail { si, '(', id, number, ciert@, fals@, imprimln, '}' }

condition { ')' }

conditionTail { ')' }

compOp { '(', id, number, ciert@, fals@ }

PREDICT:

stmt → si (condition) {stmtList} siTail { si }

siTail → sino sinoTail { sino }

siTail → { si, '(', id, number, ciert@, fals@, imprimln, '}' }

sinoTail → {stmtList} { '{' }

sinoTail → si (condition) {stmtList} siTail { si }

condition → expr conditionTail { '(', id, number, ciert@, fals@ }

conditionTail → compOp expr { == , != , > , < , >= , <= }

conditionTail → { ')' }

compOp → == { == }

compOp → != { != }

compOp → < { < }

compOp → > { > }

compOp → <= {<=}

compOp → >= {>=}

****WHILE LOOP SETS****

FIRST:

stmt { mientras, hacer } *limited to FIRST set of while loop productions

FOLLOW:

stmt { '}', mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln}

PREDICT:

stmt → mientras (condition) {stmtList} { mientras }

stmt → hacer {stmtList} mientras (condition); { hacer }

Appendix B: Example Programs

Example Program #1: Hola Mundo! (Hello World!)

Tango Sample Code

```
clase holaMundo públic@ {
      func$ principal() {
            #variable declaration
            bool URC = ciert@;

            #if statement
            si ( URC ) {
                  imprimirln("Hola Mundo! Hoy es el URC");
            }
            sino {
                  imprimirln("Hola Mundo! Hoy NO es el URC");
            }
      }
}
```

Generated Java Code

```
public class holaMundo {
      public static void main(String [] args ) {
            #variable declaration
            Boolean URC = true;

            #if statement
            if ( URC ) {
                  System.out.println("Hello World! Today is the URC");
            } else {
                  System.out.println("Hello World! Today is NOT the
                  URC");
            }
      }
}
```

Example Program #2: Guessing Game (Juego de Advinivar)

Tango Sample Code

```
clase juegoDeAdivinar públic@{
      func$ principal() {
            ent n = 27;  #hard coded number for now
            ent usuario = 0;
            escáner e = escáner() nuev@;

            mientras (usuario != n) {
                  imprimirln("Elige un número entre 1 y 100");
                  usuario = e.sigEnt();

                  si (usuario < 1) {
                        imprimirln("Su número es invalido.");
                        imprimirln("Elige un número entre 1 y 100");
                  } sino si(usuario > 100) {
                        imprimirln("Su número es invalid.");
                        imprimirln("Elige un número entre 1 y 100");
                  } sino si(usuario > n) {
                        imprimirln("Que boludo...demasiado alto!");
                  } sino si(usuario < n) {
                        imprimirln("Que idiota...demasiado bajo!");
                  } sino {    #usuario == n
                        imprimirln("Perfecto! Está correcto!");
                  }
            }
            imprimirln("Gracias por jugar!");
      }
}
```

Generated Java Code:

```
import java.util.Scanner;

public class juegoDeAdvinar {
      public static void main(String [] args ) {
            int n = 27; //hard coded number for now
            int user = 0;
            Scanner e = new Scanner(System.in);

            while (user != n) {
                  System.out.println("Choose a number b/w 1 & 100");
                  user = e.nextInt();
                  if(user < 1) {
                      System.out.println("Your number is invalid");
                      System.out.println("Choose number b/w 1 & 100");
                  } else if (user > 100) {
```

```
                    System.out.println("Your number is invalid");
                    System.out.println("Choose number b/w 1 & 100");
                } else if (user > n) {
                    System.out.println("Too High!");
                } else if (user > n) {
                    System.out.println("Too Low!");
                } else { //user == n
                    System.out.println("Perfect! You got it right!");
                }
            }
            System.out.println("Thanks for playing!");
        }
}
```

Appendix C: Source Code

See the full source code repository via the online resource GitHub: https://github.com/ashleyzeg/HonorsThesis.

Contact Information:

Author: Ashley Zegiestowsky
Primary Email: ashleyzeg@gmail.com
Secondary Email: azegiest@butler.edu

PART II:

Tango: Un Lenguaje de Programación Basado en Español

(Versión Española)

Sección 1: Introducción

El propósito de esta tesis es un proyecto de dos partes. La primera parte del proyecto tiene que ver con la creación de mi propio lenguaje de programación basado en Español, Tango, usando palabras claves en español. La segunda parte del proyecto se relaciona con el diseño e implementación de un compilador que sigue las reglas de la gramática descritas en el lenguaje Tango para analizar léxicamente, sintácticamente, semánticamente y generar el código de Tango.

Este proyecto me proporcionó una manera de combinar mis dos campos de estudio, la informática y el español, bajo un esfuerzo intrigante. Me encontré con muchos desafíos relacionados tanto con las Ciencias de la Computación y con la lengua española, pero era inspirador saber que lo que estaba creando era algo que muy pocas personas realmente han hecho y puesto en práctica. Antes de iniciar este proyecto, sabía muy poco sobre los detalles esenciales que entraban en el diseño de un lenguaje de programación y un compilador. Desde una perspectiva de alto nivel, he aprendido sobre la importancia de las gramáticas libres de contexto y las diferentes etapas que componen un compilador, que se explicará con más detalle más adelante.

Además, la estructura de esta tesis comienza con una descripción de los objetivos específicos alcanzados en el lenguaje de Tango, una explicación y ejemplos breves de la gramática de Tango, una descripción de alto nivel del diseño del compilador y de las estructuras de datos utilizadas, concluyendo con ideas para trabajos futuros y consejos útiles. La gramática completa, la lista de palabras claves y el código fuente para el compilador se pueden encontrar en los Apéndices.

Sección 2: Metas Específicas

En la creación del lenguaje de programación de Tango así como el compilador, hay cinco objetivos principales alrededor de los cuales centré mi tesis general. Estas metas son las siguientes:

1. Crear un lenguaje de programación fácil de usar que utilice palabras claves y una estructura sintáctica que se parezca mucho a la lengua española.
2. Diseñar el lenguaje técnico con una sensación similar a la de algunos de los idiomas de programación más utilizados ampliamente en un entorno educativo y profesional, como Java y C ++.
3. Permitir que el compilador pueda leer e interpretar las palabras claves y los identificadores de las variables con caracteres especiales propios de la lengua española, como acentos y tildes.
4. Proporcionar un compilador que compile cruzadas para correr contra el compilador de Java en lugar de generar un código de máquina directamente.
5. Garantizar que el lenguaje de tango sea Turing-completo, o computacionalmente universal.

La primera meta forma una espina dorsal esencial para todo el proyecto en su conjunto. Hay más de miles de diferentes lenguajes de programación utilizados y disponibles en una amplia variedad de países y culturas, y nuevos lenguajes se están creando cada día. Sin embargo, una gran mayoría de estos lenguajes de programación están basados en el inglés, es decir, las palabras claves y la estructura sintáctica se basan en el idioma inglés [6]. Menos de un puñado de

lenguajes de programación basados en el español han sido creados, de los cuales podemos mencionar (pero no limitarnos solo a ellos):

- GarGar, un lenguaje de programación procedimental español basado en Pascal con fines de aprendizaje [6].
- Latino, un lenguaje con una sintaxis basada en el español [6].
- RoboMind, un lenguaje de programación educativo disponible en varios idiomas (incluido el español) [6].

El propósito y el objetivo detrás de la creación de Tango es contribuir a la creciente comunidad técnica que se extiende a través de muchos idiomas y culturas. Con la creación de esta versión prototipo de un lenguaje de programación basado en elespañol, el alcance educativo y profesional de una herramienta de este tipo podría ser muy beneficioso en la participación e inspirar tanto a las mentes jóvenes y profesionales con experiencia en el campo del desarrollo tecnológico.

El segundo de los objetivos antes mencionados se relaciona estrechamente con la primera meta en el sentido de diseño básico y sensación del lenguaje de programación. Es esencial que Tango sea a la vez intuitivo para un hablante nativo español, sin embargo, también es compatible desde un punto de vista lógico y sintáctico a otros lenguajes de programación utilizados con más frecuencia.

El tercer objetivo, que se ocupa de la manipulación de caracteres especiales, demuestra ser altamente único de Tango. A diferencia del idioma inglés, el idioma español emplea el uso de acentos y tildes en algunas letras que forman caracteres que no existen en inglés. Estos acentos y / o tildes son esenciales para el lenguaje y son cruciales para el significado de ciertas palabras. La falta de un acento o tilde podría cambiar el significado de la palabra en su totalidad. Por lo tanto, esto permite que el compilador escanee y procese la posibilidad de estos caracteres especiales adecuados, siendo vital para la corrección y la representación de la lengua española como se demuestra en el lenguaje de programación de Tango.

El cuarto objetivo mencionado se refiere a una aplicación más técnica de diseño de compiladores. Al diseñar el compilador, he optado por una compilación cruzada a Java en lugar de generar un código de máquina directamente en la etapa de generación de código. Esta decisión, dado el relativamente corto tiempo proporcionado para completar un proyecto tan ambicioso, me permitió crear un lenguaje de programación más prolijo y comparativamente funcional. Si hubiera optado por implementar un compilador que genera código de máquina, el producto final sería más limitado en su alcance y funcionalidad en general. Otro de los beneficios de elegir compilar en forma cruzada a Java es que el compilador Java es independiente de la máquina. También es importante señalar queel idioma de destino es Java y que el compilador está escrito en Java. Sin embargo, estas dos opciones son independientes la una de la otra. Más detalles de la elaboración y aplicación del compilador se discutirán más adelante.

El quinto y último objetivo asegura que el lenguaje de tango es Turing-completo, lo que significa computacionalmente universal. Un lenguaje se considera Turing-completo si se puede utilizar para simular cualquier máquina de grabado solo Turing [9]. Por otra parte, una "máquina de Turing puede hacer todo lo que un ordenador de verdad puede hacer" [9]. Un lenguaje universal no tiene por qué ser complejo [2]. En el caso de Tango, el lenguaje en sí no es muy complejo, pero sí cumple con los requisitos para ser considerado Turing-completo.

En general, estos cinco objetivos principales ayudaron a impulsar e inspirar la investigación y el proceso de implementación necesarios para crear con éxito un lenguaje de programación prototipo basado principalmente en el idioma español.

Sección 3: Gramática Libre de Contexto (GLC)

Gramática Libre de Contexto y Tango

La gramática del lenguaje de programación de Tango es una combinación de producciones originales y los patrones de producción reconocidos de fuentes externas. La definición básica de una gramática libre de contexto es la siguiente:

> Una gramática consiste de una colección de reglas de sustitución, también llamadas producciones. Cada regla aparece como una línea en la gramática, que comprende un símbolo y una cadena separada por una flecha. El símbolo se llama una variable, o no-terminal. La cadena se compone de variables y otros símbolos llamados terminales. Una variable se designa como la variable de inicio [9].

El propósito de una gramática libre de contexto es proporcionar normas a las que una "cadena sintácticamente válida de terminales" se puede generar [7]. El proceso de generar una cadena válida de terminales se refiere como una derivación que puede ser descrita como "una serie de operaciones de reemplazo que muestra cómo derivar una serie de terminales desde el símbolo inicial" [7]. Esta misma información se puede representar gráficamente con el uso de un árbol de análisis sintáctico (que será utilizado más adelante en la prestación de producciones de muestra) [9].

Hay dos tipos principales de derivaciones: derivaciones por la derecha y derivaciones por la izquierda. A efectos de simplicidad y coherencia, los ejemplos mostrados utilizarán una derivación por la izquierda lo que significa que "en cada paso de la derivación, la más a la izquierda no-terminal" está seleccionada para la sustitución [5]. Además, la gramática de Tango puede ser

clasificada como LL (1), que representa la derivación por la izquierda cuando la entrada se explora de izquierda a derecha con la mirada de un identificador por delante. Para aclarar aún más, una gramática se dice que es LL (1) dado que, "cualquier par de reglas que definen el mismo no-terminal deben tener conjuntos de selección disjuntos [1]. El cumplimiento de la condición de declarar una gramática LL (1) es crucial para construir un analizador sintáctico descendente recursivo que se explicará más adelante en la Sección 4: Diseño de Compiladores.

La Influencia del Idioma Español & Matices en la Gramática

Otros aspectos importantes de la gramática de Tango que valen la pena mencionar se refieren a la selección de palabras claves y las estructuras sintácticas en lo que respecta a la lengua española. Tres desafíos principales ocurrieron durante la selección de palabras claves y la sintaxis:

1. ¿Cómo se debe manejar la imprevisibilidad de los sustantivos masculinos o femeninos o palabras claves descriptivas?
2. ¿Qué tiempo verbal se debe utilizar cuando se utilizan palabras claves que implican acción?
3. ¿Cómo debe la sintaxis general estar estructurada de una manera que se asemeje mucho a la estructura general de la lengua española?

En cuanto a la primera pregunta, una solución simple se utilizó con el fin de dar cuenta de palabras claves que terminan con caracteres diferentes dependiendo del objeto que se describe. Por ejemplo, en el idioma español las palabras masculinas terminan en la letra 'o' donde las palabras femeninas terminan en la letra "a". Dado que estas terminaciones de palabras dependen del sustantivo o un objeto que se ha descrito, esto creó un problema de coherencia en

la selección de palabras claves. Sin embargo, en lugar de esas palabras que terminan en una 'o' o 'a', todas las palabras claves que se ajustan a este criterio terminan con el símbolo "@". El símbolo "@" se ha convertido en el más usado y aceptado dentro de la comunidad española de palabras que podrían ser masculino o femenino. Algunas de las palabras clave para el lenguaje de Tango que caen en esta categoría son: `nul@` (null), `vaci@` (void), `públic@` (public), `nuev@` (new), `ciert@` (true), `fals@` (false). Nota, la palabra entre paréntesis después de la palabra clave española es el equivalente en Inglés para referencia para aquellos que no están familiarizados con el idioma español.

En cuanto a la segunda pregunta con respecto a los tiempos verbales para las palabras claves que implican acción, otra solución sencilla fue implementada. En la mayoría de los lenguajes de programación basados en Inglés los ejemplos de palabras claves que implican acción incluyen palabras como `do, return,` and `print`, por sólo mencionar algunos [10]. En Inglés, estos verbos están en un formato de mandato (que sólo hay una clase de conjugación de mandatos para cada verbo con independencia del objeto o de la audiencia). Sin embargo, en español, hay varias maneras diferentes para conjugar la forma de un verbo de comando y aún más las diferencias entre los distintos países y dialectos. Con el fin de establecer una representación coherente de palabras claves que implican acción que serían entendidos por todos los hablantes de español, el infinitivo del verbo sirve como el mejor método de representación. Por ejemplo, las siguientes son palabras claves que implican acción en el lenguaje de programación de Tango: `hacer` (do), `regresar` (return), `imprimir` (print). Una vez más, la palabra entre paréntesis, es el equivalente Inglés de la palabra clave española.

En cuanto a la tercera y última pregunta, lo que refleja la lengua española desde un punto de vista sintáctico, ha demostrado ser el más difícil de emular. Sin embargo, de un modo específico en el que este es frecuente en la gramática es la colocación de los identificadores de acceso en relación con una definición de función. En el idioma Inglés, los adjetivos normalmente se colocan antes del sustantivo que están describiendo; Tomemos por ejemplo la frase en ingles, "the long red dress". En el idioma español, los adjetivos se colocan normalmente después del sustantivo que están describiendo: veamos, por ejemplo, la frase, "El vestido largo y rojo" (lo que se traduciría directamente en Inglés como "el vestido largo y rojo" Este mismo principio se puede ver en la sintaxis relativa a una definición de función se demuestra en la figura siguiente:

Ejemplo de una definición de función de Java (Inglés):

```
public void func_name (int param1, int param2) {…}
```

Ejemplo de una definición de función de Tango (Español):

```
func func_name públic@ vaci@ (ent param1, ent param2) {…}
```

Figura 3.1: Tenga en cuenta las diferencias entre las dos definiciones de función son muy sutiles pero reflejan los matices estructurales de la lengua española.

Ejemplos de Producciones & Derivaciones de Tango

Esta sección le llevará a través de un simple ejemplo de producciones y derivaciones utilizando la gramática de Tango. La gramática completa y la lista de palabras claves (así como su relación equivalente en Inglés) se enumeran en detalle en el Apéndice A.

Como se mencionó al principio de esta sección, la gramática de Tango es una mezcla de producciones originales y producciones predefinidos de fuentes externas. A partir de entonces, la gramática de Tango utiliza partes de la gramática calculadora que se detalla en *Programming Language Pragmatics* [7]. Algunas de las reglas, o producciones, se muestran en la figura siguiente:

stmt → id = expr;
expr → term term_tail
term_tail → add_op term term_tail | ε
term → factor factor_tail
factor_tail → mult_op factor factor_tail | ε
factor → (expr) | id | number
add_op → + | -
mult_op → * | /

Figura 3.2: Producciones (reglas) de la gramática de la LL (1) gramática calculadora [7].

FIRST

stmt{ id }
expr{ (, id, number }
term_tail { +, -}
term { (, id, number }
factor_tail { *, / }
factor { (, id, number }
add_op { +, - }
mult_op { *, / }

FOLLOW

id { +, -, *, /, =, id}
number { +, -, *, /,), id }
({ (, id, number }
) { +, - , *, /, id}
= { (, id, number }
+ { (, id, number }
- { (, id, number }
* { (, id, number }
/ { (, id, number }
expr {), id, ; }
term_tail {), id }
term { +, -,), id }
factor_tail { +, -,), id }
factor { +, -, *, /,), id }
add_op { (, id, number }
mult_op { (, id, number }

PREDICT

1. stmt →id = expr {id}
2. expr → term term_tail {(, id, number}
3. term tail → add_op term term_tail {+, -}
4. term tail → {), id}
5. term → factor factor_tail {(, id, number}
6. factor tail → mult_op factor factor_tail {*, /}
7. factor tail → {+, -,), id}
8. factor → (expr) {(}
9. factor → id {id}
10. factor → number {number}
11. add_op → + {+}
12. add_op → - {-}
13. mult_op → * {*}
14. mul_top → / {/}

Figura 3.3: First, Follow, & Predict Sets para las producciones en Figura 3.2 [7].

stmt → id = expr;
stmt → x = expr;
stmt → x = term term_tail;
stmt → x = factor factor_tail term_tail;
stmt → x = 5 factor_tail term_tail;
stmt → x = 5 term_tail; (* factor_tail → ε)
stmt → x = 5; (* term_tail → ε)

Figura 3.4: Derivación por la izquierda de la cadena, `x=5;`

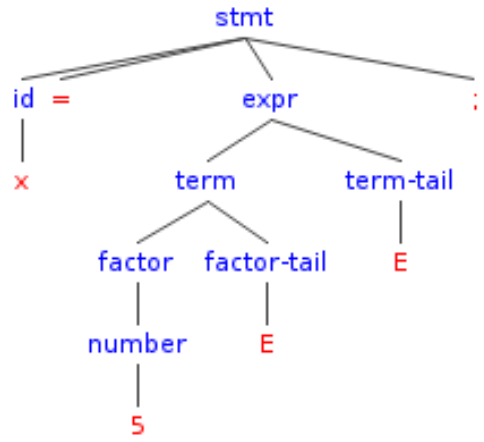

Figura 3.5: Árbol de análisis sintáctico para la cadena, `x=5;` [8].
*Nota: E significa ε, la cadena vacía

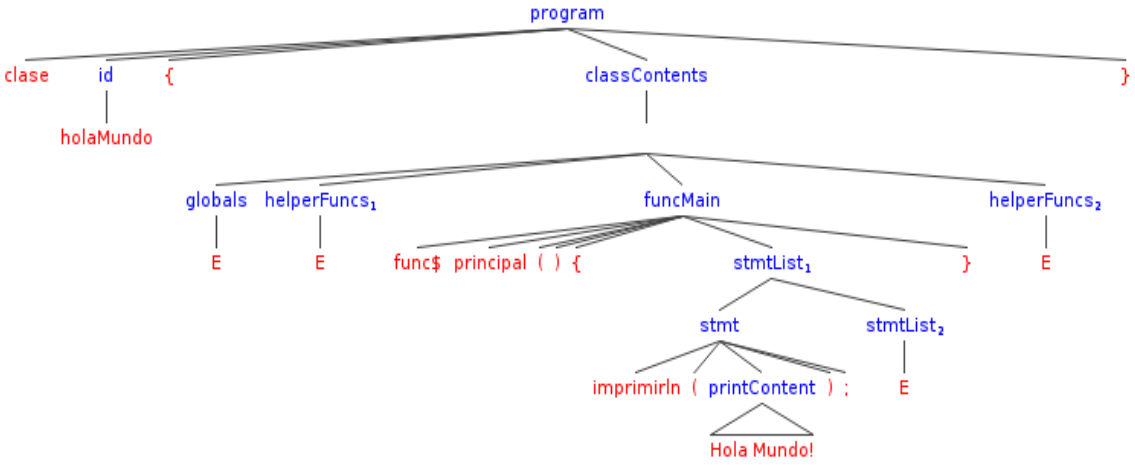

Figura 3.6: Árbol de análisis sintáctico para el programa breve Hello World que se puede ver en Apéndice B [8].
*Nota: E significa ε, la cadena vacía

Como se mencionó anteriormente, sólo una pequeña parte de la gramática de Tango se ilustra en las figuras anteriores (Figura 3.2 a 3.6). Para ver la gramática completa y la lista de palabras claves, consulte el Apéndice A.

Sección 4: Diseño de Compilador

El diseño e implementación del compilador constituyen una gran parte de esta tesis y están estrechamente relacionados con la gramática descrita en la Sección 3. La construcción de compiladores se puede dividir en cuatro etapas diferentes: analizador léxico (escáner), analizador sintáctico, analizador semántico y el generador de código [2]. El analizador léxico, o escáner, escanea el código fuente del programa carácter a carácter para "reconocer cuáles cadenas de símbolos del programa de origen representan una sola entidad o identificador" [2]. El analizador léxico también identifica y clasifica los identificadores según su valor en relación con el resto del programa. Algunos tipos de identificadores presentes en el analizador léxico de Tango que podemos incluir son: palabras claves, identificadores de variables, valores numéricos, operadores aritméticos, caracteres especiales, etc.

La segunda etapa es el analizador sintáctico. Dado que el lenguaje del Tango es un LL (1) e independiente del contexto gramatical, un analizador sintáctico descendente recursivo es el método utilizado en la realización de esta fase del compilador. El analizador buscará "sintaxis correcta, emitir mensajes de error apropiados y determinar la estructura subyacente del programa fuente" [1]. El analizador sintáctico descendente recursivo ilustra "de arriba hacia abajo de análisis sintáctico (predictivo)", que se relaciona directamente con la gramática [7].

La tercera etapa consiste en el análisis semántico. El analizador semántico se entrelaza con el analizador sintáctico. Mientras que el analizador sintáctico comprueba el programa para la sintaxis correcta según las reglas de la gramática,

el analizador semántico cumple con algunos cheques semánticos que no pueden ser realizados por el analizador sintáctico solamente [1].

La cuarta y última etapa es la generación de código. Durante la fase de generación de código, un compilador tradicional traduce los identificadores analizados correctamente o árboles de sintaxis a " instrucciones del lenguaje de máquina (binarios) o lenguaje ensamblador" [2]. Como se mencionó anteriormente en la Sección 2: Objetivos específicos, el compilador de Tango no genera código de máquina directamente. De hecho, el compilador de Tango, en la fase de generación de código, genera código equivalente a Java, que luego se ejecuta en el compilador de Java. El razonamiento para la elección de cruzar compilar para Java se explica con más detalle en la Sección 2.

Si se produce un error durante cualquier fase del compilador (escáner léxico, analizador sintáctico, analizador semántico o generador de código), el compilador termina y aparece un mensaje de error en la consola con el número de la línea y un mensaje de error potencial en función del error.

Estructura del Código & Estructuras de Datos

La estructura general del compilador es una aplicación basada en Java con cinco clases:

- *TangoCompiler.java* – Esta es la clase principal en la que se ejecuta el programa. Al iniciar el programa, el programa pide al usuario que introduzca el nombre de un archivo de entrada para ser procesada. Las instancias de los objetos *TangoScanner* y *TangoParser* crean instancias con el fin de comenzar a procesar el programa fuente.

- *TangoScanner.java* – Esta clase constituye el análisis léxico, o un escáner, fase del compilador. La clase *TangoScanner* utiliza instancias de la clase de *Token* con el fin de crear una serie de identificadores. Cada vez que el escáner reconoce un nuevo identificador, un nuevo objeto de Token se crea y se añade a la matriz de identificadores que utiliza la estructura de datos ArrayList.
- *Token.java* – Esta clase contiene todos los detalles y datos relacionados con los diferentes tipos de identificadores disponibles (incluyendo palabras claves) en el idioma de Tango. Una mezcla de listas simples y funciones de sola una línea se utilizan dentro de la clase de *Token*. La lista de objetos de *Token* se procesa a continuación en el *TangoParser* en el que algunas de las funciones simples creadas en la clase de *Token* se utilizan para llevar a cabo las comprobaciones necesarias cuando se mueve a través del proceso de análisis sintáctico. La clase de *Token* es también el lugar donde se almacena la tabla de símbolos que utiliza la estructura de datos de HashTable.
- *TangoParser.java* – Esta clase es la parte más intensiva del compilador. El *TangoParser* analiza la lista de identificadores producidos por el escáner usando un analizador descendente recursivo. Por lo tanto, la recursividad se emplea con el fin de pasar a través de la cadena de componentes léxicos. El análisis semántico también ocurre durante esta fase en la que la estructura de datos de stack se utilizan. La clase *CodeGenerator* también se crea una instancia en el interior del analizador sintáctico y de forma recursiva genera código de Java.

- *CodeGenerator.java* -- Esta clase utiliza un FileWriter con el fin de escribir el código de Java equivalente a un nuevo archivo. La clase se compone de una multitud de funciones vacío que se invocan dentro del analizador descendente recursivo con el fin de escribir correctamente en un archivo el código recién generado que luego puede ser compilado y ejecutado contra el compilador de Java utilizando el comando javac y java.

Sección 5: Conclusión

Trabajo Futuro

Después de invertir cerca de ocho meses en la creación del lenguaje de programación y compilador de Tango, me siento orgullosa de decir que he producido un lenguaje funcional, pero limitado. Si tuviera más tiempo para invertir en este proyecto, hay mucho trabajo por delante para hacer del lenguaje de Tango un lenguaje de programación totalmente funcional y libre de errores que podría ser utilizado principalmente para fines educativos. Algunas de las adiciones y mejoras que me gustaría integrar en el idioma son:

- *Ampliar la Funcionalidad* - Con tan poco tiempo, lo único que fui capaz de poner en práctica fueron algunas de las funciones más básicas. La integración de las estructuras de datos más complejas, los principios orientados a objetos y bibliotecas adicionales complementaría y mejoraría el código fuente existente.
- *Optimización y Depuración* - Muy poco pensamiento o esfuerzo se puso a optimizar el rendimiento y la memoria durante la ejecución del compilador. Esta sería una mejora importante con el fin de probar realmente el poder y las limitaciones del compilador.
- *Modificar la Generación de Código* - Como se detalla tanto en la Sección 2 como en la Sección 4, el compilador de Tango genera código de Java para ejecutar contra el compilador de Java. Modificando la fase de generación de código para generar directamente el código de máquina en lugar de código de Java permitiría el idioma de Tango a ya no depender delcompilador Java. Aunque esto puede aumentar la eficiencia y el

rendimiento para el compilador de Tango, esta implementación limitaría a Tango a una plataforma fija.

- *Sitio Web Interactivo* - Sería beneficioso también crear una plataforma en línea o un recurso descargable en el que el público pudiera acceder libre y directamente con el fin de escribir y ejecutar programas de Tango. Junto con este recurso, crear un tutorial sencillo para ayudarles a los usuarios en programación en el lenguaje de Tango con un enlace a la documentación completa podría complementar en gran medida todo el trabajo que se ha planteado en la creación de este proyecto.

Consejos Útiles

Para cualquier persona interesada en la realización de un proyecto similar, hay varios fragmentos de consejos que son útiles e importantes a tener en cuenta cuando se someten a un proyecto de esta magnitud. En primer lugar, hacer una línea de tiempo de las metas y los hitos importantes ¡y adherirse a ella! Aunque se trata de una estrategia simple y obvia cuando se someten a cualquier tipo de proyecto, se vuelve aún más importante cuando se trata de una gran base de código con un montón de piezas móviles.

En segundo lugar, la gramática, las derivaciones y los árboles de sintaxis son tan importantes como el código real escrito para ejecutar el compilador. Si su gramática tiene errores (tales como no cumplir con LL (1) criterios o muy ambigua), entonces el compilador tendrá errores (lo que hace aún más difícil la aplicación). Es más fácil corregir un error cuando todavía es sólo una regla gramatical y no cuando ya se ha integrado defectuosamente en el compilador.

Por último, empezar desde abajo y construir desde allí. Es tentador querer poner en práctica todo a la vez. Sin embargo, comenzar con la funcionalidad básica y luego continuar a modificar y ampliar el lenguaje y el compilador en consecuencia. Estos son algunos de los consejos que daría a cualquier persona con el deseo de profundizar en la teoría y diseño de compiladores.

Aunque el lenguaje de programación de Tango nunca sea usado o visto fuera del alcance de este trabajo de grado no equivale al éxito o al fracaso del producto final que se ha hecho. La curva de aprendizaje, ética de trabajo y el conocimiento técnico que he obtenido al completar este proyecto no puede cuantificarse económicamente. Sin embargo, me enorgullece el progreso que he hecho en lo que respecta al lenguaje de Tango y he logrado mucha confianza en mis habilidades técnicas en su conjunto. El lenguaje de programación de Tango demuestra un pequeño pero vital intento de ampliar el alcance de los avances tecnológicos a través de las barreras idiomáticas y culturales que comienzan en el ámbito educativo y que se desplazan hacia un entorno profesional. El lenguaje mismo es joven y necesita de maduración y finura. Al menos, Tango puede servir como punto de inspiración y de arranque en el mundo de los lenguajes de programación basados en español.

Apéndice A: Lista de Palabras Claves & Gramática Completa

Lista de Palabras Claves

Palabra Clave de Tango	Palabra Clave Equivalente de Java
ent	int
dec	double
cadena	String
lista	[]
bool	Boolean
ciert@	true
fals@	false
nuev@	new
si, sino si, sino	if, else if, else
para	for
mientras, hacer mientras	while, do while
clase	class
func$ principal()	public static void main(String [] args)
estátic@	static
vaci@	void
públic@	public
nul@	null
regresar	return
escáner	scanner
imprimpirln	println
imprimir	print
sigEnt	nextInt
# (single line comment)	/ / (single line comment)

*Wordrefernce.com [4] & The Oxford Spanish Dictionary [3] se hace referencia para seleccionar la palabra clave española correspondiente.

Gramática Completa

****High-Level Productions****

program → *clase* id accessMod { classContents }

accessMod → *públic@*

classContents → funcMain

funcMain → *func$ principal*() { stmtList }

stmtList → stmt stmtList | ε

Library Call Productions

stmt → *imprimirln*(printContent);

printContent → "stringValue" | id

Declaration & Assignment Productions

stmt → id = expr;

stmt → dataType id decTail;

decTail → = expr | ε

dataType → *ent*

dataType → *dec*

dataType → *cadena*

dataType → *bool*

expr → term termTail

expr → boolOp

termTail → addOp term termTail | ε

term → factor factorTail

factorTail → multOp factor factorTail | ε

factor → (expr)

factor → id

factor → number

addOp → +

addOp → -

multOp → *

multOp → /

boolOp → *ciert@*

boolOp → *fals@*

If Statement Productions

stmt → *si* (condition) { stmtList } siTail

siTail → *sino* sinoTail | ε

sinoTail → { stmtList } | *si* (condition) { stmtList } siTail

condition → expr conditionTail

conditionTail → compOp expr | ε

compOp → == | != | > | < | >= | <=

While Loop Productions

stmt → mientras (condition) { stmtList }

stmt → hacer { stmtList } mientras (condition);

First, Follow Predict Sets
*separados por conceptos gramaticales similares para organización

CLAVE:

Rojo = palabra clave/terminal

Azul = terminal

Normal = non-terminal

****HIGH LEVEL SETS****

FIRST:

program { clase }

accessMod { públic@ }

classContents { func$ }

funcMain { func$ }

stmtList { mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln }

stmt { mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln }

FOLLOW:

classContents { '}' }

funcMain { '}' }

stmtList { '}' }

stmt { '}', mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln }

PREDICT:

program → clase id accessMod { classContents } { clase }

accessMod → públic@ { públic@ }

classContents → funcMain { func$ }

funcMain → func$ principal() { stmtList } { func$ }

stmtList → stmt stmtList { mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln }

stmtList → { '}' }

****LIBRARY CALL SETS****

FIRST:

stmt { imprimirln } *limited to FIRST set of library call productions

printContent { “, id }

FOLLOW:

printContent { ‘)’ }

PREDICT:

stmt → imprimirln(printContent); { imprimirln }

printContent → “ stringValue “ { “ }

printContent → id { id }

****DECLARATION & ASSIGNMENT SETS****

FIRST:

stmt {id, ent, dec, cadena, bool} *limited to FIRST set of declaration productions

dataType { ent, dec, cadena, bool }

decTail { =, }

expr { ‘(‘, id, number, ciert@, fals@ }

term { ‘(‘, id, number}

termTail { +, -, }

factor { ‘(‘, id, number }

factorTail { *, /, }

addOp { +, - }

multOp { *, / }

FOLLOW:

id { =, *, /, +, -, ;, ‘)’ }

number { =, *, /, +, -, ;, ‘)’ }

= { ‘(‘, id, number, ciert@, fals@ }

({ ‘(‘, id, number, ciert@, fals@ }

) { *, /, +, -, ;, ‘)’ }

+ { ‘(‘, id, number }

- { ‘(‘, id, number }

* { ‘(‘, id, number }

/ { ‘(‘, id, number }

ciert@ { ;, ‘)’ }

fals@ { ;, ‘)’ }

ent { id }

dec { id }

cadena { id }

bool { id }

stmt { ‘}’, mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln}

dataType { id }

decTail { ; }

expr { ;, ‘)’ }

term { +, - , ;, ‘)’ }

termTail { ;, ‘)’ }

factor { *, /, + , -, ;, ‘)’ }

factorTail { +, -, ;, ‘)’ }

addOp { ‘(‘, id, number}

multOp { ‘(‘, id, number}

PREDICT:

stmt → id = expr; {id}

stmt → dataType id decTail; { ent, dec, cadena, bool }

dataType → ent { ent }

dataType → dec { dec }

dataType → cadena { cadena }

dataType → bool { bool }

decTail → = expr { = }

decTail → { ; }

expr → term termTail { ‘(‘, id, number }

expr → boolOp { ciert@, fals@ }

termTail → addOp term termTail { +, - }

termTail → { ;, ')' }

term → factor factorTail { '(', id, number }

factorTail → multOp factor factorTail { *, / }

factorTail → { +, -, ;, ')' }

factor → (expr) { (}

factor → id { id }

factor → number { number }

addOp → + { + }

addOp → - { - }

multOp → * { * }

multOp → / { / }

boolOp → ciert@ { ciert@ }

boolOp → fals@ { fals@ }

****IF STATEMENT SETS****

FIRST:

stmt { si } *limited to FIRST set of if statement productions

siTail { sino }

sinoTail { '{', si }

condition { '(', id, number, ciert@, fals@ }

conditionTail { == , != , > , < , >= , <= }

compOp { == , != , > , < , >= , <= }

FOLLOW:

stmt { '}', mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln }

siTail { si, '(', id, number, ciert@, fals@, imprimln, '}' }

sinoTail { si, '(', id, number, ciert@, fals@, imprimln, '}' }

condition { ')' }

conditionTail { ')' }

compOp { '(', id, number, ciert@, fals@ }

PREDICT:

stmt → si (condition) {stmtList} siTail { si }

siTail → sino sinoTail { sino }

siTail → { si, '(', id, number, ciert@, fals@, imprimln, '}' }

sinoTail → {stmtList} { '{' }

sinoTail → si (condition) {stmtList} siTail { si }

condition → expr conditionTail { '(', id, number, ciert@, fals@ }

conditionTail → compOp expr { == , != , > , < , >= , <= }

conditionTail → { ')' }

compOp → == { == }

compOp → != { != }

compOp → < { < }

compOp → > { > }

compOp → <= {<=}

compOp → >= {>=}

****WHILE LOOP SETS****

FIRST:

stmt { mientras, hacer } *limited to FIRST set of while loop productions

FOLLOW:

stmt { '}', mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln}

PREDICT:

stmt → mientras (condition) {stmtList} { mientras }

stmt → hacer {stmtList} mientras (condition); { hacer }

Apéndice B: Programas de Ejemplo

Programa de Ejemplo #1: Hola Mundo! (Hello World!)

Código de Tango

```
clase holaMundo públic@ {
      func$ principal() {
            #variable declaration
            bool URC = ciert@;

            #if statement
            si ( URC ) {
                  imprimirln("Hola Mundo! Hoy es el URC");
            }
            sino {
                  imprimirln("Hola Mundo! Hoy NO es el URC");
            }
      }
}
```

Código Generado de Java

```
public class holaMundo {
      public static void main(String [] args ) {
            #variable declaration
            Boolean URC = true;

            #if statement
            if ( URC ) {
                  System.out.println("Hello World! Today is the URC");
            } else {
                  System.out.println("Hello World! Today is NOT the
                  URC");
            }
      }
}
```

Programa de Ejemplo #2: Guessing Game (Juego de Advinivar)

Código de Tango

```
clase juegoDeAdivinar públic@{
      func$ principal() {
            ent n = 27;  #hard coded number for now
            ent usuario = 0;
            escáner e = escáner() nuev@;

            mientras (usuario != n) {
                  imprimirln("Elige un número entre 1 y 100");
                  usuario = e.sigEnt();

                  si (usuario < 1) {
                        imprimirln("Su número es invalido.");
                        imprimirln("Elige un número entre 1 y 100");
                  } sino si(usuario > 100) {
                        imprimirln("Su número es invalid.");
                        imprimirln("Elige un número entre 1 y 100");
                  } sino si(usuario > n) {
                        imprimirln("Que boludo...demasiado alto!");
                  } sino si(usuario < n) {
                        imprimirln("Que idiota...demasiado bajo!");
                  } sino {    #usuario == n
                        imprimirln("Perfecto! Está correcto!");
                  }
            }
            imprimirln("Gracias por jugar!");
      }
}
```

Código Generado de Java

```
import java.util.Scanner;

public class juegoDeAdvinar {
      public static void main(String [] args ) {
            int n = 27; //hard coded number for now
            int user = 0;
            Scanner e = new Scanner(System.in);

            while (user != n) {
                  System.out.println("Choose a number b/w 1 and 100);
                  user = e.nextInt();
                  if(user < 1) {
                      System.out.println("Your number is invalid");
                      System.out.println("Choose number b/w 1 & 100");
                  } else if (user > 100) {
```

```
                    System.out.println("Your number is invalid");
                    System.out.println("Choose number b/w 1 & 100");
                } else if (user > n) {
                    System.out.println("Too High!");
                } else if (user > n) {
                    System.out.println("Too Low!");
                } else { //user == n
                    System.out.println("Perfect! You got it right!");
                }
            }
            System.out.println("Thanks for playing!");
        }
}
```

Apéndice C: Código Fuente

Ver el repositorio de código fuente completo a través del recurso en línea GitHub: https://github.com/ashleyzeg/HonorsThesis.

Información de Contacto:

Autora: Ashley Zegiestowsky
Email Primaria: ashleyzeg@gmail.com
Email Secundaria: azegiest@butler.edu

Bibliography/ Bibliografía

1. Bergmann, Seth. *Compiler Design: Theory, Tools, and Examples*. Dubuque, Iowa: W.C. Brown Publishers, c, 1994. Print.

2. Brookshear, J. Glenn, David T. Smith, and Dennis Brylow. Computer Science: An Overview. 11th ed. Harlow: Addison-Wesley, 1997. Print.

3. Galimberti Jarman, Beatriz, et al. *The Oxford Spanish Dictionary : Spanish-English/English-Spanish*. Oxford; New York: Oxford University Press, 2003. Print.

4. "Online Language Dictionaries." English-Spanish Dictionary. Web. <http://www.wordreference.com/es/translation.asp>.

5. Parsons, Thomas W. *Introduction to Compiler Construction*. New York: Computer Science Press, 1992. Print.

6. Pigott, Diarmuid. HOPL: An interactive roster of programming languages. Murdoch University, School of Information Technology, 1995.

7. Scott, Michael Lee. Programming Language Pragmatics. 3rd ed. Amsterdam: Elsevier/Morgan Kaufmann Pub., 2009. Print.

8. Shang, Miles. Syntax Tree Generator. 2011. Web. <http://mshang.ca/syntree/>.

9. Sipser, Michael. Introduction to the Theory of Computation. 3rd ed. Boston, MA: Cengage Learning, 2013. Print.

10. Weiss, Mark Allen. *Data Structures & Problem Solving using Java*. Boston: Pearson/Addison Wesley, 2010. Print.