Building a Compilator on Java for a mini-C Programming Language

J. Agustín Barrachina
IEEE Student Member
École Polytechnique
Université Saclay-Paris

Philémon Poux
École Polytechnique
Université Saclay-Paris

CONTENTS

***Abstract*—In this project, a compilator was created to generate a x86-64 assembler code from a C fragment called mini-C. This is a 100% C-compatible fragment, in the sense that any Mini C program is also a C program.**

I. INTRODUCTION

"Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is error-free! Thus, the most important objective in writing a compiler is that it is correct" [**?**]

The reader is suppose to have some basic knowledge of C and for that reason almost no explanation regarding that language will be treated in this report. For further information about C language refer to [**?**].

Simply stated, a Compiler is a program that can read a code written in a specific programming language and translate it into an equivalent code of another language. A fairly good analogy can be made by a translator between two different languages like Spanish and French for example.

The objective of this project is to create a compiler for a fragment of C denominated *Mini C*. Produce a reasonably effective code x86-64. *Mini C* is a fragment of the language C which contains integers and pointers to structures. *Mini C* is 100% compatible with C in the sense that every *Mini C* program is also a C program. This will enable to use a C compiler such us **gcc** to use as reference.

A. Structure of a Compiler

A compiler can be divided into two parts. The *analysis* (front end) and the *synthesis* (back end)

The *analysis* brakes the source program into constituent pieces and imposes a grammatical structure of them in order to create a intermediate representation of the source program. During this part, syntactical formation and semantical unsound is checked. The analysis also collects information about the source program and stores it in a data structure called a *symbol table* which will be used by the *synthesis* part.

The *synthesis* part makes use of the *symbol table* and the intermediate representation constructed by the analysis part and creates the target program.

A more detailed diagram of the structure can be seen in figure 1. Where the last part (Code Generation) correspond to the *synthesis* phase and the rest are all from the *analysis* phase. The diagram is longer that the one displayed, having

also a converser from the assembler to the machine language and from there to the executable code. But in this project, those stages are not treated.

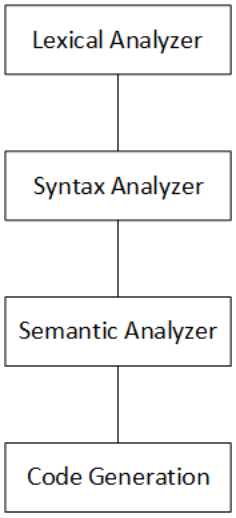

Fig. 1. Structure of a Compiler

II. Lexical Analyzer

A lexical analyzer (*Lexer*) is the first front-end step in compilers, matching keywords, comments, operators, etc, and generating a token stream for parsers called *lexemes* consisting of a *token name* and the *attribute value*. The *token name* is an abstract symbol that will be used during syntax analysis, while the *attribute value* is an entry in the *symbol table* (discussed on I-A) which will be used during the semantic analysis and the code generation.

The Lexer reads input from the programming language to compile (mini c in our case) and matches it against regular expressions and runs a corresponding action if such an expression is matched.

To make the Lexer are going to use:

- Regular Expressions: To describe the lexemes
- Finite Automata: To recognize the expressions

A. Regular Expressions

The concept of regular expression arose in the 1950's when the American mathematician Stephen Cole Kleene formalized the description of a regular language.

A regular expression is a sequence of characters that define a search pattern. In other words, there are a conjunction of letters and digits that follow a certain rule.

Let us define *letter* as any letter in the Latin alphabet and *digit* any number [0-9]. Then we can define rules as follow:

$$0|[1-9](<digit>*|[]) \tag{1}$$

Last equation 1 is a declaration of a decimal digit. The "|" is a logic or, it means, either the digit is 0 or it will be another thing. If it is not only 0, the number cannot start by 0 in C syntax, so it must start with a digit different from 0, which is range from 1 to 9 (encoded as [0-9]). Secondly, this digit can be followed by either nothing (represented by: []) or by any digit for as many digits are they must be. The format <rule>* means the repetition of a rule for as many times as necessary, or no repetition at all.

B. Finite Automata

A *finite automata* is basically a binary graph which just say "yes" or "no" by means of a *recognizer* to each possible string.

There are two different classes of automatas:

1) *Nondeterministic Finite Automata* (NFA)
2) *Deterministic Finite Automata* (DFA)

The first class (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state. The DFA on the other hand have for each state and symbol exactly one edge with that symbol leaving that state.

C. Implementation

For the lexical analyzer, a flex library was used [**?**]. A .flex file was created and then, by means of jflex, converted to the final java class.

Jflex lexers are based on a DFA automata. For more information about jflex library please refer to [**?**].

III. Syntax Analyzer

The syntax of a programming language describes the proper form of its programs.

The *syntax analyzer* or *parser* uses the first component of the *lexemes*, the *tokens*. The *parser* creates some kind of tree representation that depicts the grammatical structure of the token stream called the *syntax tree*. In this tree, each node represent an operation and the children of the node represent the arguments of that operation.

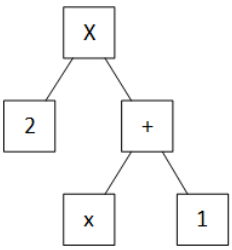

Fig. 2. Syntax Tree Example

On figure 2 an example of a tree representation can be seen. Many syntaxes can be the cause of that tree, for example, both 3 and 2 can be used to generate such tree.

$$2 * (x + 1) \tag{2}$$

$$(2 * ((x) + 1)) \tag{3}$$

A. Implementation

The Syntax Analyzer was done using CUP, a parser generator for java [**?**].

In that file, precedence where applied to each operator using the following table:

Operator	Associativity
=	right
||	left
&&	left
== !=	left
< <= > >=	left
+ -	left
* /	left
! -(negative)	right
->	left

TABLE I
PRECEDENCE

The higher the operator is in the table, the last will it be applied. For example, the operator "=" will be applied after all the other operators have been applied. There is a very simple way to apply this table in the .cup file by simply writing the following command: precedence ¡associativity¿ ¡token name¿. Where associativity is either left or right and the token name is the token from the *lexem* created by the Lexer.

After the precedence, the following grammar was implemented (table II). The details on how to implement it where omitted. In order to see how to implement the following grammar in cup please refer to the user manual [**?**].

< file >	< decl >* EOF
< decl >	< var > | < type > | < funct >
< var >	int < ident >$^+$, ; struct < ident > (*< ident >$^+$,) ;
< type >	struct < ident > < var >* ;
< funct >	int < ident > (< param >*,) < bloc > struct < ident > * < ident > (< param >*,) < bloc >
< param >	int < ident > | struct < ident > * < ident >
< expr >	< integer > < ident > < expr > -> < ident > < ident > (< expr >*,) ! < expr > | -< expr > < expr > < op > < expr > sizeof (struct < ident >) (< expr >)
< op >	= | == | != | + | - | * | / | && | < | <= | > | >= | ||

< instr >	;
	< expr >;
	if (< expr >) < instr >
	if (< expr >) < instr > else < instr >
	while (< expr >) < instr >
	< bloc >
	return < expr >;

< instr >	< var > * < instr > *

TABLE II
GRAMMAR

Where '|' is either one or the other is found. The '*' is as many as necessary (including none at all), while on the other hand, '+' means as many as necessary but at least one. If under any of those symbols there is a comma (',') it means there are comma indented.

Within a file, one will find a list of declarations. These declarations can be either a variable or structure (global variables) of functions. The declarations of functions have it's own parameters (or none) and a bloc in which it will be the code. The bloc is composed by the declarations of the variables followed by instructions. The instructions are loops such as if or while or simply expressions. Expressions can be all type of C & C ++ expressions such as integers, pointers, negation, call to functions, etc.

IV. Semantic Analyzer

"Well typed programs do not go wrong"

The semantics of a programming language defines what each program does when executing.

A *Semantic Analyzer* uses the *syntax tree* and the information in the *symbol table* to check the source program for semantic consistency with the language definition.

An important part of the *semantic analysis* is the *type checking* where it gathers type information and checks that each operator has matching operands. An example of the type checking will be to make sure the index which whom an array is accessed is an integer and not any other incompatible type. In a equation like $8.0 + 4$, the type checking will make sure to convert the integer "4" into a floating point before making the operation.

The *type checking* will make sure that the variables of a equation like $e1 + e2$ are from the same type and reject the incoherent programs. There are some languages that use **dynamic types**, which means they check they check the type of the variables dynamically. Such languages are for example PHP, Python or Lisp. On the other hand, there are also **static types** languages which is the compiler the one in charge on checking the types. For example OCaml, Java and C (which will be our case).

V. Code Generation

In this section we will actually generate the assembly code itself. It takes as input an intermediate representation of the source program and maps it into the target language. It is too difficult to be able to do this part in only one step. So it will actually be divided into 3 stages:

1) *Register Transfer Language* (RTL)
2) *Explicit Register Transfer Language* (ERTL)
3) *Location Transfer Language* (LTL)

Each stage will be explained in their corresponding sub section.

A. Register Transfer Language (RTL)

For this stage we will use the *syntax tree* created on III in order to create what will be called as *RTL tree*. We suppose that the local and global variables are already differenced and that the type of each variable recognized as it has been done in the last section.

The main objective (more precisely the first part) of the RTL is to create a set of instructions x86-64 from the operations of C.

The second phase is to create a a *Register Transfer Language*. Here a *Control Float Graph* (CFG) is created that will facilitate the ulterior phases and that will eliminate the distinction between statements and expressions. This RTL will create the so called pseudo registers, which are an infinite number of intermediate registers to realize operations. This registers will be converted into actual x86-64 registers in the future.

1) Implementation: Each file contains a list of declarations of functions (as can be seen on II) that contains a bloc statement. Each function, contains a list of parameters and the list of declarations of variables. A RTL graph is created for each function in a recursive way. As seen II, a function has a list of statements that will be converted into one or more assembler commands. For each command a label will be created and saved into the RTL graph. Each statement and expression class will have a ”toRTL” method that will save it's label into the graph. The function will have as arguments the register and label to exit and it will return it's own label to be given to the next toRTL call in order to line up every statement and expression. To make condition branches, another function will be created called ”toRTLc” which will receive two label, one to be done if the expression is true, and another in order to be done in the other case. The structure will be represented as toRTLc(e, s1Label, s2Label) where 'e' will be the expression with a true or false value. s1Label will be the label to go if 'e' is true and s2Label in the other case.

In order to realize the && expression, for example: if e1 && e2 do s1 else s2 The following conversion will be done: toRTLc(e1 && e2, s1Label, s2Label) -> toRTLc(e1, toRTLc(e2, s1Label, s2Label), s2Label) Using a similar logic, the expression: if e1 || e2 do s1 else s2 will be converted: toRTLc(e1 || e2, s1Label, s2Label) -> toRTLc(e1, s1Label, toRTLc(e2, s1Label, s2Label))

In order to make the negative sign, for example -2, the compiler actually does the operation 0 - 2. In order to make the not operation, for example !a. The compiler does an if statement such as: if a then 0 else 1. In which case, making something like !!41 will return 1 as a result. Which is what actually happens in C code.

B. Explicit Transfer Language (ERTL)

C. Location Transfer Language (LTL)

VI. Conclusion

Acknowledgment