unibz
Fakultät für Informatik
Facoltà di Scienze e Tecnologie informatiche
Faculty of Computer Science

Project Technical Report

Boo-Compiler

A boolean programming language compiler

Author	Andriy Tyyko[1], Marco Mondini[2], Luca Sabiucciu[3]
Course	Formal Languages and Compilers
Institution	Free University of Bolzano
Academic Year	a.y. 2016/2017 - Winter Semester
Contact	[1]andriy.tyyko@stud-inf.unibz.it [2]marco.mondini@stud-inf.unibz.it [3]luca.sabiucciu@stud-inf.unibz.it
Last update	Wednesday 25th January, 2017

Contents

1 Introduction

1.1 Language

The input language for *Boo-compiler* was designed with some basic functionalities and addresses, in particular, computations with boolean operations. Other functionalities have been added in order to make it more usable and flexible.
Following, the main aspects that characterize the source language.

- Operations of boolean algebra: and, or, not
- Operations on real numbers: addition, subtraction, multiplication, division, absolute value (i.e. +, -, /, *, | |)
- Overriding the of operation precedence convention by means of round brackets
- Operations with mathematical signs: (un)equal, greater and less than (i.e. ==, != , >, <)
- Operations with if statements. If can be followed by an optional else statement
- Declaration, access and operations with variables: boolean, int and integer
- Variable scoping: global and local scoping by means of curly brackets (i.e. { and }).
- Variable shadowing. Note that only the last declared variable will be used until the end of its scope
- Single- and multi-line comments
- Result printing (print the result of a boolean or mathematical operation)
- Variable's value printing
- String printing (possibly with escape characters)
- String concatenation by means of the plus sign (i.e. +)
- Exit the program whenever the "exit" keyword is encountered (in case of if or if-else condition, the program will exit only if the condition for reaching the instruction is satisfied)

1.2 Symbols and Reserved Keywords

The following table illustrates the reserved keywords of the source language. All other words different than the following (also with uppercase letters), are not recognized as keywords, but rather as identifiers for variables, if they start with a letter, which can be followed by other letters or digitis.

Keyword	Category
true	Ground value
false	
0 .. 9	
or	Boolean operator
and	
not	
bool	type
int	
integer	
show	interaction
print	
if	conditional
else	
exit	exit

Table 1: Table of Keywords

In *Boo-compiler* every instruction is ended with a dot (" ."), allowing for multiple instructions on the same line. An *expression* is the result of one or multiple variables/ground values (combined with *boolean operators*). *Boolean operators are surrounded by an expression*, like in the following example

expr <**boolean operator**> *expr* .

When declaring a variable, it's type has to be defined, along with the name and, optionally, the value. One or multiple variables can be declared on the same line, assigning and non assigning values to them. Following an example is shown

<**type**> *id* .
<**type**> *id* = *expr* .
<**type**> *id* , *id* = *expr* , *id* , ... , *id* = *expr* .

Interacting with the user is a fundamental role and with the *print* command it is possible. It can print the value of a variable (that has been declared and has a value) and the result of an operation. It automatically adds a new line. Instead *show* doesn't automatically print a new line, as in the next example

print expr .
print identifier .
show expr .
show identifier .

Strings can be printed with the *print* keyword. Special characters can be escaped with the escape sequences shown in 2. Moreover strings can also be concatenated as shown below

print <**string**> .
print <**string**> + <**string**> .

In order to exit the program, the *exit* keyword is used

exit .

Conditions can be made of only an *if* condition or an *if-else* condition. If the condition is evaluated true, then the code will be executed, otherwise the code belonging to *else* (if present). Condition's code can not contain variable declarations or other *if* conditions (leading to nested *ifs*). Below the allowed instructions are shown as example

if expr { *expr . assignment . print expr . show expr . exit .* }
if expr { *expr . assignment . print expr . show expr . exit .* }
else { *expr . assignment . print expr . show expr . exit .* }

Following a list of the symbols that can be used

Symbol	Name	Category
=	equals	assignment
<	less than	comparison
>	greater than	
==	equals equals	
!=	not equal	
.	dot	separator
,	comma	
(	opened parenthesis	parenthesis
)	closed parenthesis	
{	opened curvy parenthesis	
}	closed curvy parenthesis	
+	plus	operator
-	minus	
*	star	
/	slash	
| |	modulo	
\"	quotes	escape sequence for strings
\t	tab	
\n	new line	

Table 2: Table of Symbols

2 Compiler's architecture

A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language). In general, compilers go from high-level languages to low-level languages.

Figure 1: Compiler overview

Following a short description of the phases of a compiler.

1. **Lexical Analysis**
   This is the first phase of compilation. Input characters are read and a sequence of tokens is produced and transmitted to the *Parser*

2. **Syntax Analysis**
   This phase checks that the received tokens are in the right order and that they follow the rules defined in the grammar

3. **Semantic Analysis**
   The *Semantic Analysis* is the last part of Analysis and computes additional information related to the meaning of the program

4. **Intermediate Code Generation**
   As the name suggests, this phase generates intermediate code (three-address code)

5. **Code Optimization**
   This phase optimizes the previously generated three-address code, removing useless computations. Careful is given to preserving the meaning of the program

6. **Code Generation**
   This is the last phase and, given a three-address code, it generates the executable code

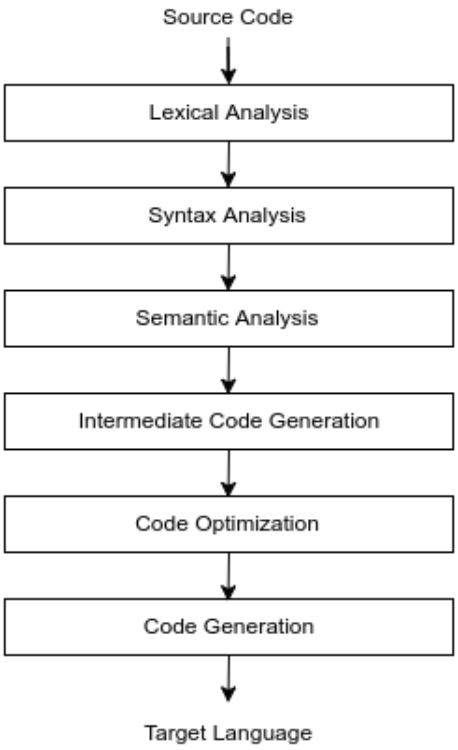

Figure 2: Compiler's phases

In this project, the program that has been used to automatically generate code for the lexical analyzer was Lex. YACC, from the other side, was the program (parser generator) used to take as input a specification of a syntax, and to produce as output a procedure for recognizing the language.

3 Lexical Analyzer

The *Lexical Analyzer* is the component of the compiler which task is to read the input program as a string, identify the lexemes and to transmit the corresponding token, along with the attribute, to the *Parser*. Usually a *Lexical Analyzer* has access to the *symbol table*, and passes the pointer to the record as attribute to the *Parser*.

3.1 Syntax

The syntax of *Boo-Compiler* is small, but powerful enough to compute boolean operations, solve mathematical expressions and to communicate to the user results (both of mathematical and boolean operations), as well as predefined strings.
The idea is that every instruction ends with a "." (dot), allowing to write multiple instructions on the same line. Only the variable declaration allows to define multiple variables on the same line without repeating the type, but the declarations are split by a "," (comma). In some cases, after an expression, the dot is not necessary to conclude the instruction, because some other symbol is going to signal the end of the instruction. For example, in the *if* statement there is no need of dots at the end of the conditional expression since the expression to evaluate is surrounded by the *if* keyword and the *opened curved bracket*.
Comments are also available in the two classic forms:

- Single line comment, achieved with "//" (two consecutive slashes) for commenting the following text until the end of the line
- Multiline comment, achieved with "/*" before the portion of text that is to be commented and "*/" after the portion of text that has to be commented

Moreover the language is key-sensitive, that is, it is different to write a letter uppercase rather than lowercase.

3.2 Lexical Analysis

The code for the *Lexical Analyzer* is contained within the *boo-lex.l* file.
In the first part of the file *C* code is used in order to include two lybraries, declare a char array, named *buffer*, and to declare a pointer to a char, named *s*. These two last variables are used for string recognition and storing.
The second part of the file contains regular expressions, which are used as regular definitions both in the current section as well as in the following section. Regular definitions match letters (lower and upper case), numbers, ids (variable names), types (int, bool or integer) and an empty space (space, tab and new line).
Following, in the third and last section, *Lex* uses the regular expressions to capture the different lexemes and to transmit to the *Parser* the corresponding token and attribute, if necessary. For space sake simple lexeme recognition commands will not be described, but attention will be rather given to more complex commands. In this last part single line comments, multi line comments and strings are recognized and handled. Comments do not need to transmit a token to the *Parser* since they have to be ignored, while strings have to be stored and transmitted with the corresponding token and attribute.

Lex represents the regular expressions that it has to capture, with a *FA* (Finite Automata), which are all merged togheter in order to recognize different patterns. It is possible to modify this behaviour, entering a particular recognition mechanism, to which *C* actions can be associated.

```
1  %x COMMENT COMMENT_LINE
2  %%
3  /* Matches a multiline comment */
4  "/*"                      { BEGIN(COMMENT); }
5  <COMMENT>"*/"             { BEGIN(INITIAL); }
6  <COMMENT>\n               { }
7  <COMMENT>.                { }
8  /* Matches a single line comment */
9  \/\/                      { BEGIN(COMMENT_LINE); }
10 <COMMENT_LINE>\n          { BEGIN(INITIAL); }
11 <COMMENT_LINE><<EOF>>     { BEGIN(INITIAL); }
12 <COMMENT_LINE>.           { }
```

Lex Code 1: Lex comment recognition

The first line of code shown in 1 contains a declaration of an *exclusive or* through the *%x* sign, followed by one or more exclusive start names. The *exclusive or* operator con be also written uppercase (*%X*). *Exclusive start conditions* differ from *regular start condistions* because those rules that do not start with the *exclusive condition* are not matched. In order to enter an *exclusive state* a pattern with an action containing the instruction *BEGIN(exclusive_name)* has to be matched. With this function, *Lex* enters a particular state, where the only regular expressions that are matched are those that are preceeded by the current *exclusive state*. At line 4, when the lexical analyzer matches the pattern /*, an *exclusive or* is started with the command *BEGIN(COMMENT)* within the action. *Lex* enters an exclusive state, where it matches only patterns that follow the *exclusive state* in which it currently is (see lines 5-7). Actions may be associated, but in the current case there is no need to transmit any token to the *Parser*, since comments are ignored. An *exclusive state* can be exited by starting a new one or with *BEGIN(INITIAL)* (or *BEGIN 0*). *BEGIN(INITIAL)* restores the normal behaviour of *Lex*, like at line 5, where after encoutering the pattern that delimits the end of the comment, the normal behaviour is restored. As for the normal behaviour of the lexical analyzer, the order of *exclusive states* followed by a regular expression matters, because the first matched pattern will be returned. Thus, for comments, the last match (line 7, 12) contains an expression for matching every character, capturing what the previous expressions didn't. When the ending delimiter of the comment is encountered, the normal behaviour of the lexical analyzer is restored, while with everything else nothing is done. For single line comments the ending delimiter is a new line, which is matched with the new line (\n) sequence, where the normal state of the *lexical analyzer* is restored through the *BEGIN(INITIAL)* instruction. In the particular case where the single line comment is the last line of the file, *<<EOF>>* is used to match the end of the file, since a new line will not be matched (otherwise it will not be the last line of the file) and the instruction for returning to the starting state is executed.

```
1  %{
2          char buffer[100];
3          char *s;
4  %}
5  empty [ |\n|\t]
6  %x string
7  %%
8  /* Matches one or multiple strings, concatenated with a + */
9  \''                { BEGIN(string); s=buffer; }
10 <string>\'' {empty}*\+{empty}*\''    {   }
11 <string>\''         { *s=0; BEGIN 0; yylval.lexeme=buffer; return STRING; }
12 <string>\\n         { *s++='\n'; }
13 <string>\\\''       { *s++='\'''; }
14 <string>\\t         { *s++='\t'; }
15 <string>\n          {   }
16 <string>\t          {   }
17 <string>.           { *s++=*yytext; }
```

Lex Code 2: Lex string recognition

The first four lines of code shown in 2 contain the declaration of an array of type *char* of size 100 and a pointer to a char. In the following section an *exclusive or* named "string" is declared. On line 9, when the quotes are encountered, *Lex* enters an exclusive state and the pointer *s* is pointed to the same address of the *char* array. In order to ignore new lines (\n) and tabs (\t), the *exclusive state* and escape sequence have to be before the line 17, since in *Lex* the order of the regular expressions determines the recognition order. When these characters are found within the *exclusive state*, no action is executed. The only admitted *escape sequences* are new lines, tabs and quotes (see line 12-14). After starting the *exclusive start condition*, when *Lex* recognizes an escaped new line, tab or quote (also the regular expressions are escaped in order to be matched), the instruction for adding the escaped sequence to the buffer is executed. At line 17, where everything else is matched, the variable *yytext* is referenced, using a pointer and it's value is added to the buffer. In order to concatenate multiple strings toghether, using the *Java* style, the end of a string has to be recognized, followed by a +, followed by a new string. The character for cancatenating strings has not to be immediately after or before the end or start of a string, but instead it may be separated with spaces, tabs or new lines (matched through the *empty* regular definition). Such a match has no action associated since it is meant to be used from the user, in order to achieve a fancy arrangement of strings that are to be printed.
When only the quotes are found, meaning that the string ended and no other string is concatenated,the index of the *char* array is restored to 0 through the pointer, the initial condition is restored through the *BEGIN 0* instruction (*BEGIN 0* and *BEGIN(INITIAL)* are the same), the value of the lexeme is set to the one stored in the buffer during the matching of the string and the token "STRING" is returned.

4 Parser

The *Parser* obtains a sequence of Tokens from the lexical analyzer and verifies that the sequence can be generated by means of a Derivation in the CFG of the source program. As a result the parser outputs a (representation of a) Parse-Tree. Parsers are classified as *Bottom-UP* or *Top Down* depending whether the *parse-tree* is built starting from the leaves or from the root, respectively.

4.1 Data Type Implementation

Boo-Compiler admits only three data types: *bool*, *int* and *integer*. The *bool* type assumes a value that is either *true* or *false*. Such a value can only be used for boolean expressions along with the *int* data type, which, analogously to *bool*, can only assume the value 0 or 1. As for most of the programming languages, 0 is interpreted as *false* and 1 as *true*. Boolean operations may be performed with the two aforementioned types, but in case the two types mismatch, a *cast* operation has to be applied, specifying which of the two values has to be casted. The *integer* type may assume any integer value (32 signed bits, as in the *C* language). Such kind of type can be used for mathematical operations like sum, difference, multiplication and division, as well as for boolean operations. In the last case an automatic cast will be performed from *integer* to *int*, where values greater than 0, will be interpreted as 1 (true), 0 (false) otherwise. Moreover a notice will be printed on the terminal, warning that an automatic cast has been performed.
Thus, in case of a boolean operation between *bool* and *integer*, an explicit cast will be needed, since *integer* will only be casted to *int*, leaving to the aformentioned situation. In case of mathematical operation, all the types have to be of type *integer*, otherwise an error will be raised.

The only *C structure* implemented is the one named *typedData*, which contains

value : it is of type *int* and stores the value of the variable

scope : it is of type *int* and stores the scope's number at variable creation

init : it is of type *int* and it is used as boolean value (0 or 1) in order to retrieve if a value has been assigned to the variable

type : it is of type *char* and stores the type of the variable (int, bool, integer)

previous : it is a pointer of type *typedData* and points to the previous variable stored into the stack

next : it is a pointer of type *typedData* and points to the next variable stored into the stack

Variables are stored into the stack, one over the other and a pointer variable, named *topOfTheStack* is kept, in order to always have a reference to the top of the stack.

4.2 Parsing Table

Compilers usually use a *Parsing Table* for storing information like name, value and size. *Boo-compiler*, instead, uses a stack, an abstract data type that binds the different elements one after the other, following the *LIFO* (Last In, First Out) principle. Traditionally, *Parsing Tables* are

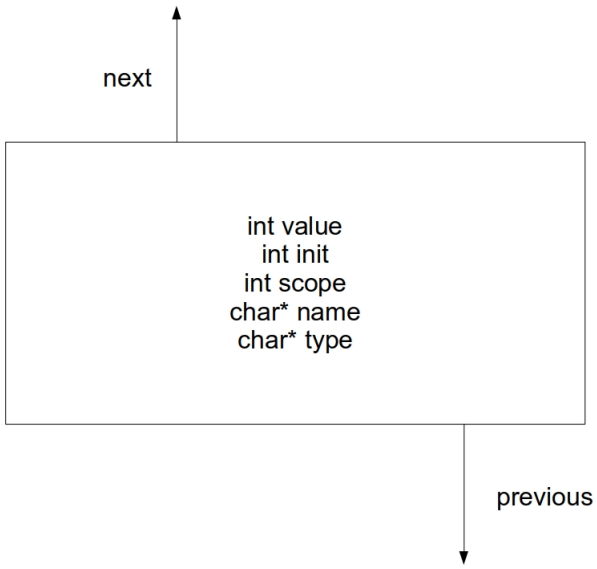

Figure 3: Schema of a Variable

nested one into the other when dealing with scopes (also called blocks). Thus looking for an element will require to look for it into the current table and, if not found, recursively, to look into the previous until the starting table is not reached or the element is found. This operations are time consuming if compared to a single *Parsing Table* which can be scanned linearly. Traditional *Parsing Tables* delimit a scope with the start and the end of themselfs, while with a stack there is no possibility to delimit a scope. Therefore each element of the stack must have an identifier, denoting the number of scope to which it belongs. Elements of the stack that belong to the ending scope, recognized thanks to their scope identifier, are popped out of the stack, since there will not be more any need of them. Thus the elements on top of the stack are the ones that do belong to the current scope, while those below belong to a parent scope, which still has not been ended.
The stack of the *Boo-compiler* is divided into to sections:

static section : this part of the stack is the first that is created and contains structures with predefined values. The types *int* and *bool* are stored in this part, with all the possible values (actually only two values each). Thus they do not occupy a lot of space and can be used multiple times, since their value will not be changed. Values within this section can be retrieved through a function that looks only in this section of the stack. Reference to the top value of this section is kept through a pointer named *staticStackValues*.

dynamic section : this part of the stack is the part that is created dynamically at run-time. Every time a new variable is declared, it is inserted on the top of the stack. Also *integers* are stored into the dynamic part of the stack and references to them are kept until the end of the scope.

The reason of using a static part is rather simple: instead of adding multiple times the same value into the stack, create the value once and reference it through pointers. In this way the

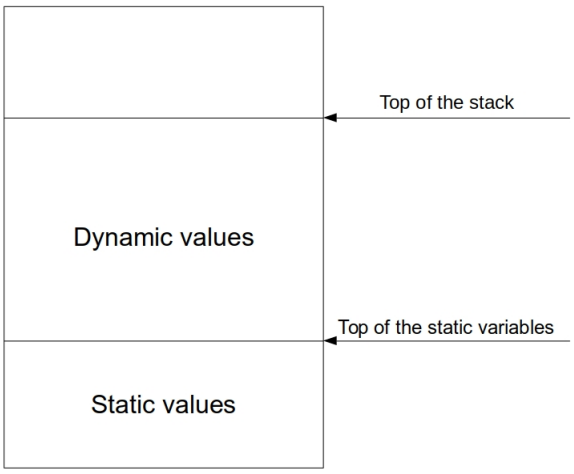

Figure 4: Schema of the Stack

stack will be lighter, reducing space consumption and elements to go through when performing search operations on the stack. On the other side, *integers* are stored in the dynamic section of the stack, since they can assume a huge variaety of values which, most of the times, will not be used. Thus *integers*, along with variables, are added when found at run-time and popped out of the stack when their scope ends. When started, the *topOfTheStack* variable and the *staticStackValues* variable are equivalent.
Every *struct* of the stack has a variable named *scope* of type *int*, which is used to store the number of the current scope, kept by a global variable. Every time a new scope is entered, denoted with a curved parenthesis $\{$, the global variable that keeps the count of the scopes is incremented, so all new variables that are created next will have a different *scope*. In order to exit the current scope, a closed curved parenthesis is used $\}$. The scope counter is decremented and all variables with a *scope* greater than the global are removed from the stack (using the *free() C function*). Thus all variables and *integers* that have been created during the previous scope are destroyed, leaving space for new ones.

4.3 Error Rising and Handling

Handling and recovering from an error is one of the most important aspects of a computer program. Users have to be warned if their input is wrong or if something went wrong during the execution of their input. Thus, the lexical analyzer has a regular expression for matching everything that can not be matched by the previous regular expressions which should only capture legal keywords. This last regular expression consists of only the dot symbol (.), which matches every character and, if reached it prints on the terminal a notification error, warning that the user entered an invalid sequence of characters at a particular line.
Semantic errors are not detected from the lexical analyzer, but from the control structure of the *Parser*. Sequences of tokens that do not match any rule defined in the *Parser*, will automatically

be captured and handled, by notifying the user about what happened. Differently, *semantic errors* that are not matched by just using rules, have to be detected by the logic of the compiler. For example using wrong data types will not be automatically detected from the *Parser*, but from the type check that has to be defined. In *Boo-compiler* type checking is done at run-time. When the lexical analyzer transmits a token, information about the type are stored into the stack and will then be used when firing rules. Operations can only be applyed between variables of the same type, so the stored information about the variables is used to check if types correspond (and also to compute the resulting value).

References

The report is mostly based on what we have learned during the *Formal Languages and Compilers* Course, taught by prof. *Alessandro Artale* during the academic year 2016/2017.
Main references:

1. Artale A. *Formal Languages and Compilers - Lecture Slides* Faculty of Computer Science - Free University Of Bolzano/Bozen

2. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques and Tools, Second Edition* 2006, Addison Wesley *ISBN 0-321-48681-1*

Other useful resources:

1. Tutorial on epaperpress.com
2. Tutorial on Yacc
3. IBM Tutorial on start conditions
4. Variable shadowing
5. Stack
6. Tutorial on C from *Tutorialspoint.com*