AUTOMATED CODE GENERATION SYSTEM FOR THE SYNTACTIC PHASE OF A COMPILER

Irina RANCEA[1], Valentin SGÂRCIU[2]

Contextul informatic curent implică aplicații uşor extensibile capabile de a rezolva probleme complexe prin efort minim al dezvoltatorilor. Astfel a apărut conceptul de generare de cod automat permițând astfel utilizatorilor să-şi concentreze eforturile în direcția dezvoltării de noi concepte şi arhitecturi şi să folosească module care pot fi generate în mod automat. Lucrarea propune o abordare automată a generării regulilor pentru faza analizei sintactice a unui compilator pornind de la documente de intrare semi-structurate ce reprezintă manuale de referința ale limbajului de programare în cauză.

The current information context involves various applications that are easily extensible, and can solve complex problems with minimal developers' effort. Thus the automated code generator concept appeared as a manner of saving developers' time and allowing them to design new concepts and architectures. Our paper proposes an automatic approach for the syntactic analysis phase of a compiler. The input for our system is represented by semi-structured documents describing programming languages references.

Keywords: code generator, syntactic analyzers

1. Introduction

In the software area there are various modules and applications that can be automatically generated, saving developers' time. Anything which is repetitive can be automated. This is an approach that is already gathering speed in the agile community. [1] Given the correct conditions, a lot of source code can be automatically generated and then the programmer is free to "fill in the gaps".

For code to be generated, the following three areas must be predictable and understood: [2]

- design patterns – the template to which the code will be generated
- domain meta-data – the topology that will be modeled in the generated code; sometimes it receives extra-information from the developers
- domain rules – the expected behavior of the application

[1] Eng., Faculty of Automatic Control and Computers, University of POLITEHNICA Bucharest, Romania, e-mail: irina.rancea@gmail.com

[2] Prof., Eng, Faculty of Automatic Control and Computers, University of POLITEHNICA Bucharest, Romania, e-mail: vsgarciu@aii.pub.ro

Some of the advantages of automatically generated code are: it is very consistent, being much cleaner and simpler than the programmers code; it is stable and customizable; the developers have more time to focus on designing new architectures.

The major disadvantages of an automatically generated code are related to the fact that it works only on a specific set of conditions and there will always be some code that will need user adapting.

The entry point of this paper was the research direction of Aho [3] about automatically generating code for the compilers back-end phases starting from declarative specifications that map the intermediate representations tree in machine code (*instruction select, code emission* – as stated below in the Theoretical background chapter). Different contributions in this area can be found in [4] [5] [6].

The innovation of our work consists of a code generator system that will automatically generate other stage from a compiler architecture than Aho and Proebsting proposed [3] – the syntactic and lexical analysis included in the front-end phase (described below in the Theoretical background chapter). The system has as input the references manuals for the language programming in cause, parsed by an information extraction system. The paper presents the system architecture for the code generator, the algorithm behind the system, and a testing platform for the proposed case study.

2. Theoretical background

A compiler is a program that translates a source program written in some high level programming language into machine code for some computer architecture. For having portability an intermediate representation (IR) is used that makes the compiler as a two phase system – the first one, called *front-end* maps the source code to an intermediate representation, and the second one, called *back-end* maps the IR representation to machine code. In conclusions, the *front-end* can be written just once and the *back-end* has to adapt to each needed architecture. [7] Each phase has a stages subset. [8]

The *front-end* phase consists of the following processes:

- *scanning*: the characters are grouped in atomic units called tokens (Fig. 1)
- *parsing*: an analyzer recognizes tokens sequences complying with the grammar rule and generated the *Abstract Syntax Tree*
- *symbol table:* a data structure consisting of one record for each identifier containing its attributes. Some examples of such stored attributes are: type, scope (program location where the identifier is valid)
- *semantic analysis*: performs data types analysis and translates the AST tree to the intermediate representation IR

- *optimization*: optimizes the intermediate representation of the source code
The *back-end* phase consists of the following processes:
- *instruction select*: maps the intermediate representation IR to assembly code
- *code optimization*: optimizes the assembly code using different techniques such as data flow analysis, registers allocation
- *code emission*: generates the machine code

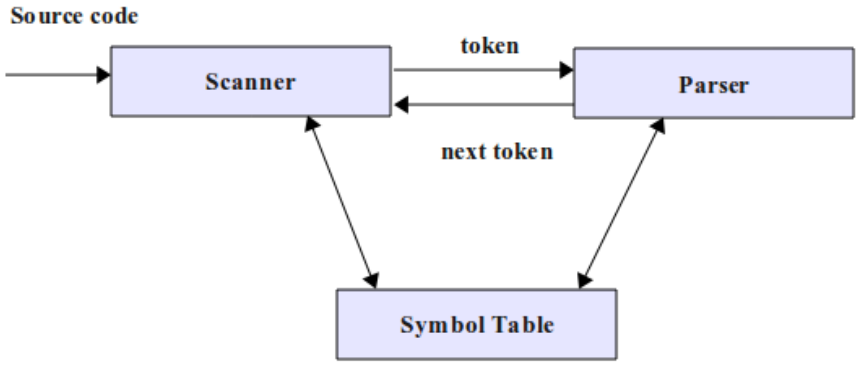

Fig. 1. Scanner-Parser Interaction

The lexical scanner is also responsible with the following issues at the user interface level: [8]

- eliminate comments and spaces (blanks, tabs, newline characters) from the source code
- correlating error messages from the compiler to the source code (the scanner stores a map with tokens and their corresponding lines in the source code)
- in some compilers the scanner creates a copy of the source code and the corresponding error messages
- if the parsed language allows preprocessing macros they are solved at the scanner level

One of the most popular parser generator tool is ANTLR – it is a recursive-descent parser generator that accepts a large class of grammars called LL(*) that can be augmented with semantic and syntactic predicates [9] to solve the grammar ambiguities.

ANTLR has a consistent syntax for specifying scanners, parsers and tree parser; it uses EBNF grammars (Extended Backus-Naur) [10] that can handle optionals and repetitive elements. BNF grammars [7] need a more flexible syntax in order to allow such structures. EBNF grammars also allow sub-rules. Our code generator is using ANTLR language for grammar development.

3. System Architecture

The syntactic analyzer involves generating lexical and syntactic rules that must comply with the language programming grammar. The syntactic rules can be divided in two classes: rules that can and must be present in every parser and specific rules for the the language structures.

The common rules describe whitespace handler, strings and identifiers handlers. The specific rules refers to special characters that can be present in a language programming syntax, by example operators. The syntactic rules are specific to each language programming, following its hierarchical structure.

Our code generator system has been tested on the *Draft Standard for the Functional Verification Language e* [11] [12] [13] document. The tree grammar is generated according to the ANTLR language. [14]

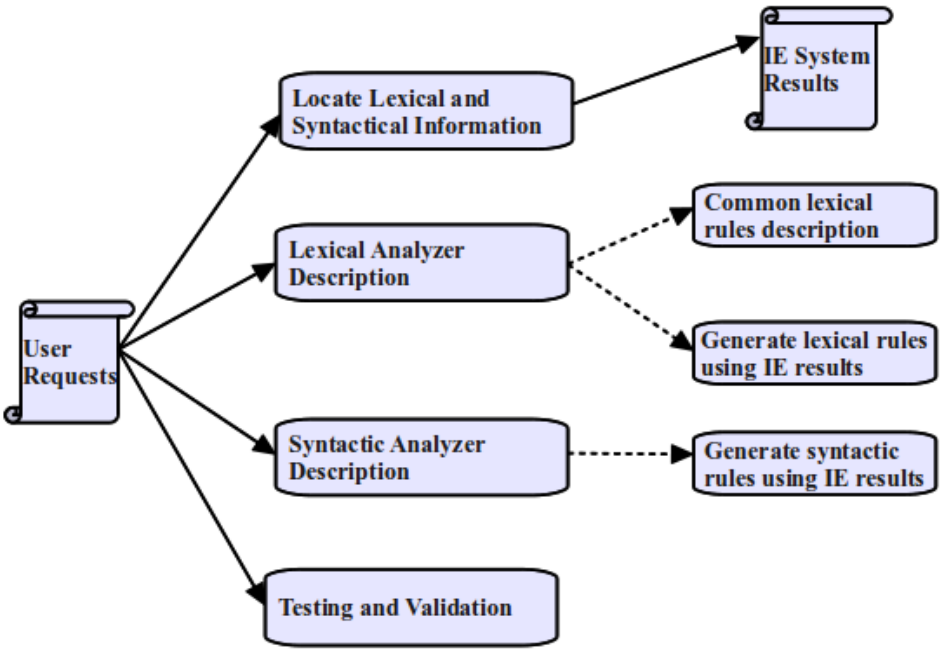

Fig. 2. Entities Diagram

As one can see in the diagram above (Fig. 2), our system involves two entities. The first one is the user that can ask the application to generate syntactic rules for the given class of structures. The second entity is represented by the results given by an information extraction system that stores in specific data files information about structures syntax. [15]

The wrapper system is able to extract information from semi-structured input documents that represents tutorial for different language programming, storing them in a suggestive manner.

4. Experimental results

Our algorithm for automatically generating grammar rules consists in the following stages:

Syntactic Analyzer - Code Generator Algorithm
Step 1. Generate headers
Step 2. Generate syntactic rules
- generate specific options
- load data files containing the syntactic structures coding
- process loaded files and generate rules according to the parser rules

Step 3. Generate lexical rules
- generate specific options
- generate common lexical rules
- load data files from the IE system containing keywords, operators and comments
- generates operators rules with their precedence
- generate keywords rules
- generate comments rules

The syntactic rules are automatically generated after processing data files grouped on categories; these files are compiled by an information extraction system that reads a tutorial of the programming language and extract the relevant information about the structures syntax. [15]

Each programming language has a predefined hierarchy for its instructions that specifies the compiler's actions. These instructions can be grouped in a limited set of basic concepts. Our data files are classified according to the major concepts of the programming language. For the “e” programming language, the hierarchy for its data structures is presented in Fig. 3. Each data file of our system contains one structure per line as shown below:

'struct' ID LBRACKET 'like' ID RBRACKET LBRACE LBRACKET ID DOT ID

'extend' LBRACKET ID RBRACKET ID LBRACE LBRACKET ID DOT ID

'unit' ID LBRACKET 'like' ID RBRACKET LBRACE LBRACKET ID DOT ID

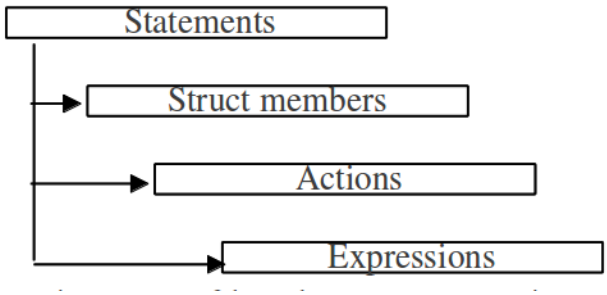

Fig. 3. Basic concepts of the 'e' language programming

The syntactic rules are grouped according to the above concepts and follow the next constraints:

- LBRACKET/ RBRACKET tokens are treated as optional arguments
- For the case of interior blocks our system will generate the rule to starting interior block marker – RBRACE – and then will complete the rule with the inside rule name, as in the following example:

'struct' ID LBRACKET 'like' ID RBRACKET LBRACE
LBRACKET ID DOT ID

will became:

struct' ID LBRACKET 'like' ID RBRACKET LBRACE
'struct_members' RBRACE

- For the case of syntactic structures that contain the call of other syntactic structures the system will generate the sub-rule as (sub_rule)*, meaning that the sub-rule can appear for zero or as many times, allowing to compile without syntactic errors source code as in the following example:

```
struct my_struct {
        a : int;
        b: bool;
}
```

The token analysis (Fig. 4) consists in discovering that a token is a possible identifier or an operator. For each possible identifier our system checks

that the token is found among the reserved keywords of the programming language or is a data type. If none of these conditions is met, then the token is an identifier. Each operator will be converted to a specific code, as in Table 1.

Table 1

Operators Codes

Operator	Coding format
[	LBRACKET
]	RBRACKET
(	LPAREN
)	RPAREN
{	LBRACE
}	RBRACE
<	LT
>	GT
:	COLON
;	SEMICOLON
.	DOT
~	TILDA
?	QUES
!	NOT_LOGIC
%	PERCENT
=	EQ
&	BIT_AND
|	BIT_OR
^	BIT_XOR
*	STAR
/	SLASH
+	PLUS
-	MINUS
@	AT

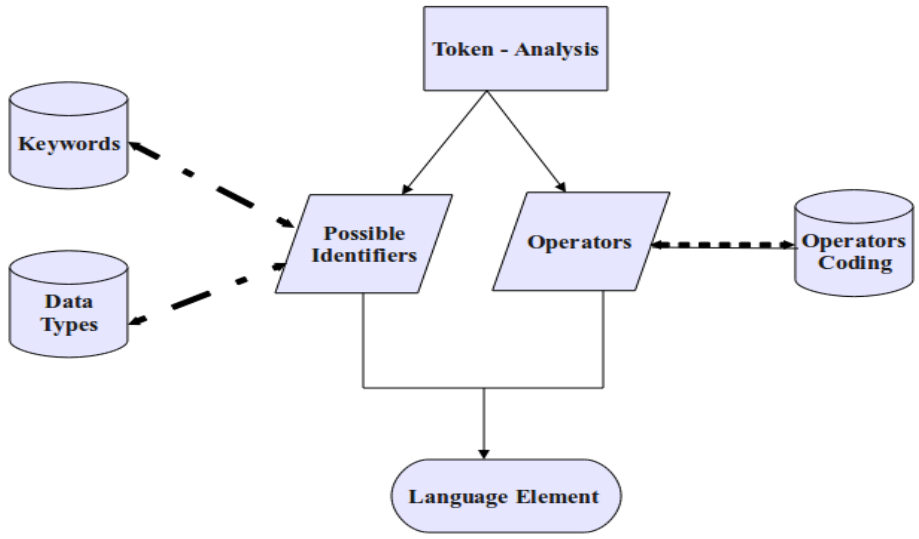

Fig. 4. Token Analysis

One of the principal issues for a grammar are the ambiguities. An ambiguous language is a language where the same sentence can be interpreted in various ways. A syntactic analyzer is non-deterministic if there is at least one decision point where it cannot solve the route. [16]

There are a few methods to solve grammar ambiguities – left factoring and syntactic predicates [17]. Using left-factoring, the following rule that is ambiguous because ANTLR cannot decide which alternative is the proper one can be rewritten as: [18]

```
a : L+ K                a : L+ (K | M)
  | L+ M                  ;
  ;
```

Our code generator for the grammar rule is able to correctly generate different alternatives, but it cannot handle ambiguities such as left-factoring. This aspect is purpose for future developments. At this moment the grammar ambiguities are manually handled. An example of grammar ambiguity for our grammar can be observed below:

```
statement
```

```
:"struct" ID ( "like" ID  )? LBRACE ( struct_members )* RBRACE
|"extend" ( ID  )? ID LBRACE ( struct_members )* RBRACE
|"unit" ID ( "like" ID  )? LBRACE ( struct_members )* RBRACE
;
```

and the solution for avoiding it is presented below:

```
statement
: "struct" ID ( "like" ID  )? LBRACE ( struct_members )* RBRACE
|"extend" ( ID ) + LBRACE ( struct_members )* RBRACE
|"unit" ID ( "like" ID  )? LBRACE ( struct_members )* RBRACE
;
```

5. Testing Framework

Validation for a syntactic analyzer involves testing each grammar rule. All the rules are tested against a set of positive and negative tests. The positive tests are the ones that represent valid source code that must be accepted by our automatically generated grammar. The negative tests represent invalid source code that must be rejected by our grammar. Our testing system consists in a set of 135 tests, distributed as in Table 2. Our testing platform is configured as in Fig. 5.

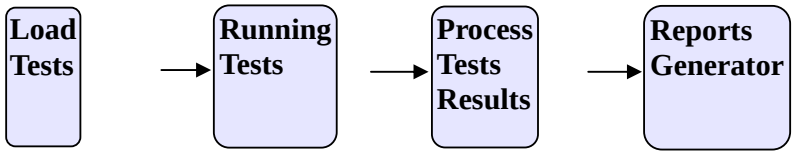

Fig. 5. Testing Platform

Table 2

Tests Classification

Syntactic structures category	Syntactic structure	Tests	
		Positive Tests	Negative Tests
Statement	'struct'	2	2
	'unit'	2	2
	'extend'	2	2

Struct member	'event'	3	7
	'field'	8	10
	'method'	10	10
Action	'var'	5	5
	'compute'	5	5
	'emit'	2	3
	'while'	5	5
	'for'	22	24
	'break'	3	4
	'continue'	3	4

The positive tests are considering the following issues: using of all options for a syntactic structure in different combinations, mixing various syntactic structures in order to perform a complicated source code closer to user needs.

The negative tests deal with introducing inexistent options, calling of multiple options of which only ones are valid, declaring syntactic structures in inappropriate contexts, omitting parentheses in functions declaration, omitting braces for blocks of instructions, using keywords when expecting identifiers, using data types for declarations that return no data type and many others.

6. Conclusions

Our paper presents an automated code generation system that can complete grammar rules of a syntactic analyzer using ANTLR support. We apply the proposed algorithm to a various set of syntactic structures and it has proven to be stable, efficient and precise. We are also considering adding further features such as automatically solve grammar ambiguities and incipient semantic checks.

Acknowledgements

The work has been funded by the Sectoral Operational Programme Human resources Development 2007-2013 of the Romanian Ministry of Labor, Family and Social Protection Through the Financial Agreement POSDRU/6/1.5/S/16.

R E F E R E N C E S

[1] *D.S. Kolovos, R.F. Paige, F.A.C. Polack*, "An agile and extensible code generation framework", Proceedings of the 6th International Conference on eXtreme Programming and Agile Processes in Software Engineering, vol. **3556**, ISBN 9783540262770, pp. 226-229, Publisher: Springer, 2005

[2] *K. Czarnecki, U. Eisenecker,* Generative Programming Methods, Tools, and Applications, ISBN-10: 0201309777, ISBN-13: 978-0201309775, Publisher: Addison-Wesley Professional, 2000

[3] *A.V. Aho, M. Ganapathi, S.W.K. Tjiang,* "Code generation using tree matching and dynamic programming*",* ACM Transactions on Programming Languages and Systems 11, **4**,pp. 491 - 516, 1989

[4] *T. Proebsting*, "Optimizing an ANSI C Interpretor with Superoperators"*,* Proceedings of Principles of Programming Languages POPL'95, pp. 280-287, San Francisco, California, 1995

[5] *M. Ganapathi*, Code Generation and Optimization using Attribute Grammars, PhD thesis, University of Wisconsin, Madison, 1980

[6] *I. Böhm*, Automatic Code Generation using Dynamic Programming Techniques, Master Work Paper at Johannes Kepler Universität Linz, 2007

[7] *L. Fegaras*, "Design and Construction of Compilers", CSE 5317/4305, University of Texas at Arlington, CSE http://lambda.uta.edu/cse5317/notes/notes.html

[8] *A. Aho, R. Sethi, J. Ullman*, Compilers Principles, Techniques, and Tools, Addison-Wesley Publishing Company, 1986

[9] *T. Parr, R.W. Quong*, "Adding Semantic and Syntactic Predicates to LL(k) – pred-LL(k)", Proceedings of the International Conference on Compiler Construction, Edinburgh, Scotland, 1994

[10] **** *L.M. Garshol*, BNF and EBNF: What are they and how do they work?, 2008 http://www.garshol.priv.no/download/text/bnf.html

[11] **** *Design Automation Standard Committee of the IEEE Computer Science*, "IEEE P1647™/D9 Draft Standard for the Functional Verification Language e". www.ieee1647.org/downloads/P1647_Draft_6_071214.pdf

[12] *I. Rancea, V. Sgârciu*, "Functional Verification of Digital Circuits using a Software System", Automation, Quality and Testing, Robotics.2008.AQTR 2009. IEEE International Conference, vol. **1**, pp. 152-157, 22-25, Digital Object Identifier 10.1109/ AQTR.2008.4588725, Cluj-Napoca, 2008

[13] *I. Rancea, V. Sgârciu*, "Principles of Functional Verification for Digital Circuits", Annals of DAAAM for 2007 & Proceedings of the 18th International DAAAM Symposium, ISBN 3-901509-58-5, ISSN 1726-9679, vol. **18**, pp. 637-638, Editor B. Katalinic, Published by DAAAM International, Vienna, Austria, Location: Zadar, Croatia, 2007

[14] *T. Parr,* The Definitive ANTLR Reference: Building Domain Specific Language, ISSN: 978-0-9787-3925-6, 2007 http://www.antlr.org/

[15] *I. Rancea*, Automated code generation system for a compiler syntactic analysis phase based on free text processing, PhD Thesis, Politehnica University of Bucharest, 2011
[16] **** *T. Parr*, ANTLR - centric Language Glossary http://www.antlr.org/doc/glossary.html
[17] **** *W. Colaiuta,* ANTLR Predicates, 2007 https://wincent.com/wiki/ANTLR_predicates
[18] **** *J. Luber*, How to remove global backtracking from your grammar, 2009 http://www.antlr.org/wiki/display/ANTLR3/How+to+remove+global+backtracking+from+your+grammar