Egyptian Computer Science Journal (ISSN-1110-2586)
Volume 41– Issue 2, May 2017

Parsing Strings Using a Subset of the Production Rules of the C-language

[1]I. A. Ismail, [2]N. Amein. Ali
[1]Department of Computer Science &6 October University in Cairo, Egypt.
[2]Institute of Suez of Management information systems, Suez, Egypt.
Amr44_2@hotmail.com, Nabil92471@gmail.com

Abstract

The parsing is applied to the given c-language which is a context free grammar (CFG) language, i.e., any CFG language together with its productions could be parsed as an LL(1) grammar language giving its parsing table and the parsing results to any input string.

Keywords: *context free grammar (CFG), LL(1), first, follow; input string.*

1. Introduction

In natural languages, sentence correction using grammar rules is a part of natural language processing. Such implementation in computer field requires syntax parsing and syntactical analysis of a sentence[1]. The grammars that describe these languages should be given and simple. Nevertheless, we don't deal here with natural languages but with formal languages. These formal languages are mainly programming languages that have their own grammars while their vocabulary is much less in length than the natural languages. We are interested in this research paper in top down parsing. In this paper we deal with context free grammar (CFG) which the programming languages are part of it.

First we start with the given production rules then we manipulate with these production rules to get the required language strings, which we will inspect its acceptance or rejection. This will necessitates that we should consider the compilation of some sentences with specific lengths, and compiling them by producing the necessary parsing tables.

These parsing tables are produced under different parsing rules. We consider first some of the parsing rules and then design c program that may compile some strings that satisfy these rules, by creating their parse tables first and then using these parse tables to compile some given input string. Nevertheless, we can change the production rules to produce some different language which we can use in programming and see the suitable strings that could be accepted using the compiler in hand and compilation rules in hand.

2. Programming languages and CFGs

When we design a compiler, a precise description of the language is needed. Among the ways in which programming languages can be defined precisely, context free grammars (CFGs) are perhaps the most widely used ones. Actual programming languages have many features that can be described elegantly by means of context-free languages. What formal language theory tells us about context-free languages that it has important applications in the design of programming languages as well as in the construction of efficient compilers [2]. The

CFGs can describe any typical language like C or any other programming language as we will see in the next section [2].

3. C-notations as a subset of production rules using CFG

When writing a CFG for a programming language like C, one normally starts by dividing the constructs of the language into different syntactic categories. A syntactic category is a sub-language that embodies a particular concept. Examples of common syntactic categories in programming languages are Expressions, Statements and Declarations.

When the nonterminal term appears to the L.H.S it is termed the main terminal, if not, it is termed the nonterminal. Each syntactic category is denoted by a main *nonterminal*. More non terminals might be needed to describe a syntactic category or provide structure to it, as we shall see, and productions for one syntactic category can refer to non-terminals for other syntactic categories. For example, statements may contain expressions, so some of the productions for statements use the main nonterminal for expressions [3].

Example1: In the following C statement, CFGs systematically describe the syntax of the C programming language constructs like *expressions* and *statements* [4]. Using a syntactic variable *stmt* to denote statements and variable *expression* to denote expressions, the production rules of figure (1) specify the structure of the form of expressions and define precisely what an *expression* is while the production rules of figure (2) specify some conditional C statements.

expression = *expression op expression*
op = +|-|*|/|<|>|==|<=|>=
expression =*num*
expression =**(***expression* **)**

Figure 1. Simple expression grammar

stmt= **id =** *expression ;*
| if (*expression* **)** *stmt*
| if (*expression* **)** *stmt* **else** *stmt*
| while (*expression***)** *stmt*
| do *stmt***while (** *expression) ;*
| for (*optexpr* **;** *optexpr* **;** *optexpr* **)** *stmt*
expression = *expression* ***op*** *expression*
op = +|-|*|/|<|>|==|<=|>=
expression =**num**
expression =**(***expression* **)**
optexpr —> **e**

Figure 2. A CFG for a subset of C statements

Where, expression, *stmt* and *optexpr* are nonterminal while +, -, *, /, (,),**num**and e (empty string) are terminals.

We must know that, all Constructs that begin with keywords like **while** or **int**, are relatively easy to parse, because the keyword guides the choice of the grammar production that must be applied to match the input. We therefore concentrate on *expressions*, which

present more of challenge, because of the as sociability and precedence of operators. For example, the following CFGs describe *expressions*, *terms*, and *factors*.

expression=expression -term|term
term= term / factor| factor
factor =pow^factor|pow
factor=(expression)|id

Figure 3. A CFG for a subset of C statements

Let E represents expressions, T represents *terms*, P represents *pow* and F represents *factors* that can be either parenthesized expressions or identifiers d, then we can rewrite the CFGs of figure(3) as follow:

E = E - T | T
T = T / F| F
F = P^F|P
F = (E)| id

These productions cannot be used foretop-down parsing because of the following:
The production rules for E and T are left recursive.
The production rules F = P^F|P are left factoring.
Therefore, the following non-left-recursive variant and non-left-factoring for the same grammar is to be used for top-down parsing:

E=TA
A=-TA
A=e
T=FB
B=/FB
B=e
F=PH
H=^F
H=e
P=(E)
P=d

Figure 4. A CFG for a subset of C statements

These CFGs are subset of c-statements will be passed to our LL(1) program where the lowercase symbols are terminals.

4. LL(1) program for Parsing a subset of c-statements

In our paper, we design a parser LL(1) using top down parsing to build parse trees for the input string of c-language, the parser scans the input string from left to right, one symbol at a time. Top-down parsing can be viewed as an attempt to find a leftmost or rightmost derivation for an input string. Equivalently, it can be viewed as an attempt to construct a parse tree for the input string from the root and creating the nodes of the parse tree in preorder [5].

The input string is accepted if and only if, at the moment when the blank symbolis read and the *top* symbol on the *stack* is $ [6].

In order to determine whether or not a grammar is LL(1), we introduce several kinds of grammar analyses, such as determining whether or not a nonterminal can derive the empty string, and determine the set of symbols that can appear as the first symbol in a derivation from a non terminal [7].

4.1 First and follow functions

First, The LL(1) program starts by isolating the LHS and the RHS of the given productions in figure(4) and then finding the terminals and nonterminals as follow:

LHS	RHS
E	TA
A	-TA
A	e
T	FB
B	/FB
B	e
F	PH
H	^F
H	e
P	(E)
P	d

Nonterminals	terminals
E	-
A	e
T	/
B	^
F	(
H	)
P	d

Second, the program calculates the first and follow of each nonterminals:

Nonterminals	**First**	**follow**
E	**(, d**	**$,)**
A	**- , e**	**$,)**
T	**(, d**	**- , $,)**
B	**/ , e**	**- , $,)**
F	**(, d**	**/ , - , $,)**
H	**^ , e**	**/ , - , $,)**
P	**(, d**	**^ , / , - , $,)**

4.2 Constructing the parsing table

Constructing of parsing table is an important activity in predictive parsing method [8]. Our LL(1) program uses the resulting first and follow values to construct the LL(1) parsing table which will be used in parsing strings of c-statements.

NT	-	/	^	(	)	d	$
E	Error	Error	Error	TA	Error	TA	Error
A	-TA	Error	Error	Error	e	Error	e
T	Error	Error	Error	FB	Error	FB	Error
B	e	/FB	Error	Error	e	Error	e
F	Error	Error	Error	PH	Error	PH	Error
H	e	e	^F	Error	e	Error	e
P	Error	Error	Error	(E)	Error	d	Error

4.3 Parsing C-strings using the parsing table

For example, the steps parsing the string (d-d)/d^d$ are given below.

Stack	Input	Action
$E	(d-d)/d^d$	
$AT	(d-d)/d^d$	E->TA
$ABF	(d-d)/d^d$	T->FB
$ABHP	(d-d)/d^d$	F->PH
$ABH)E(	(d-d)/d^d$	P->(E)
$ABH)AT	d-d)/d^d$	E->TA
$ABH)ABF	d-d)/d^d$	T->FB
$ABH)ABHP	d-d)/d^d$	F->PH
$ABH)ABHd	-d)/d^d$	P->d
$ABH)AB	-d)/d^d$	H->e
$ABH)A	-d)/d^d$	B->e
$ABH)AT-	-d)/d^d$	A->-TA
$ABH)ABF	d)/d^d$	T->FB
$ABH)ABHP	d)/d^d$	F->PH
$ABH)ABHd	d)/d^d$	P->d
$ABH)AB	)/d^d$	H->e
$ABH)A	)/d^d$	B->e
$ABH)	)/d^d$	A->e
$AB	/d^d$	H->e
$ABF/	/d^d$	B->/FB
$ABHP	d^d$	F->PH
$ABHd	d^d$	P->d
$ABF^	^d$	H->^F

$ABHP	d$	F->PH
$ABHd	d$	P->d
$AB	$	H->e
$A	$	B->e
$	$	A->e

Therefore, the input string gets parsed.

It is also observed that the input string is scanned from left to right and while parsing the input string. Also we must know that, only one input symbol is dealt with while taking the parsing action. Hence, the name LL (1).We confirm that the left recursion and ambiguous grammar are not allowed here.

Note that popping of similar symbols takes place when the stack and input symbols are the same.

5. Conclusion and Future work

We note that from the examples dealt with that the compiler used compilers properly all the input strings if they are to be accepted and rejected if not, using subset of the production rules.

We can apply the same method used above to different CFG grammars and to summarize grammars that uses subsets of the given grammars of some given language, which enables us to use smaller subsets of an larger grammar containing very large productions rules.

References

[1]. "*Sentence Correction For English Language Using Grammar Rules And Syntax Parsing*", Proceedings of 28th IRF International Conference, 7th June 2015, Pune, India, ISBN: 978-93-85465-29-1.

[2]. PETER LINZ,*"An Introduction toFORMAL LANGUAGES and AUTOMATA" Fifth Edition.* Publisher: Cathleen Sether 2012.

[3]. Torben Ægidius Mogensen "*Basics of Compiler Design*", University of Copenhagen, Published through lulu.com.2010.

[4]. Alfred V, Aho and Jeffrey D. Ulman, "*Compilers Principles, Techniques, and Tools*", Pearson Education,2007.

[5]. Tao Jiang, Ming Li, Bala Ravikumar, Kenneth W. Regan.*"Formal Grammars and Languages".*

[6]. Anil Maheshwari, MichielSmid "*Introduction to Theory of Computation*", Carleton University, Canada, 2017.

[7]. Johan Jeuring Doaitse Swierstra, "*Grammars and Parsing*"' 2001.

[8]. A.A. Puntambekar, "*Compiler Design*", Technical Publication, 2008.