Journal of Software Engineering and Applications, 2012, 5, 21-28
http://dx.doi.org/10.4236/jsea.2012.51004 Published Online January 2012 (http://www.SciRP.org/journal/jsea)

LR(K) Parser Construction Using Bottom-Up Formal Analysis

Nazir Ahmad Zafar

Department of Computer Science, King Faisal University, Hofuf, Saudi Arabia.
Email: nazafar@kfu.edu.sa

Received October 13th, 2011; revised November 16th, 2011; accepted November 28th, 2011

ABSTRACT

Design and construction of an error-free compiler is a difficult and challenging process. The main functionality of a compiler is to translate a source code to an executable machine code correctly and efficiently. In formal verification of software, semantics of a language has more meanings than the syntax. It means source program verification does not give guarantee the generated code is correct. This is because the compiler may lead to an incorrect target program due to bugs in itself. It means verification of a compiler is much more important than verification of a source program. In this paper, we present a new approach by linking context-free grammar and Z notation to construct LR (K) parser. This has several advantages because correctness of the compiler depends on describing rules that must be written in formal languages. First, we have defined grammar then language derivation procedure is given using right-most derivations. Verification of a given language is done by recursive procedures based on the words. Ambiguity of a language is checked and verified. The specification is analyzed and validated using Z/Eves tool. Formal proofs are presented using powerful techniques of reduction and rewriting available in Z/Eves.

Keywords: Compiler Construction; LR(K) Parser; Context-Free Grammar; Z Specification; Correctness; Verification

1. Introduction

A compiler is a program that translates a source code into its equivalent machine readable code. The translation process is termed as compilation which then can be used to execute the resultant code specified in the original source code. It is noted that the source language is at higher level as compared to machine code. The higher level languages not only increase abstraction level between source and resulting codes but also increase complexity to formalize such abstract structures. The target language is normally a low level language generated from a source code.

Compiler construction has always been considered as an advanced research area than other programming practices mainly due to the size and complexity of the code generated. The design and construction of a fully verified compiler will remain a challenge of twenty first century. As mentioned above, the main functionality of a compiler is to translate a source code written by programmers to an executable machine code correctly and efficiently. Although a lot of work is done in this area but compiler construction is a mature area of research which needs further investigation. This is because the bugs in the compiler can lead to an incorrect machine code even the source code is fully verified to be correct. Further, as executable generated code is tested and if bugs are detected it might be due to the source program or compiler itself. This issue has led to verification of a compiler that proves that a source program is correct before allowing it to run on the machine.

Formal methods are mathematical-based techniques used for specification, proving and verification of software and hardware systems [1]. The process of formal verification means applying these approaches to verify the properties ensuring correctness of a system. Formal verification of software targets the source program where semantics of the language gives precise meanings to the program analyzed. On the other hand, program verification does not mean that the resultant executable code is correct as specified by the semantics of the source program. This is because the compiler may lead to an incorrect target program because of the bugs in the compiler and it can invalidate the guarantees ensured by the formal methods. It proves that verification of a compiler is much more important than verification of a source program to be compiled.

Parser or syntactic analyzer is an important part of a compiler. Parsing is the process of analyzing a sequence of tokens generated by the lexical analyzer to determine its grammatical structure with respect to a given grammar. More precisely, task of a parser is to determine how an input string can be derived from the start symbol of the grammar using set of rules called production of the grammar. There are two main approaches of parsing, *i.e.*, top-down and bottom-up parsing. In top-down, left most derivations are used to accept an input stream and tokens are consumed from left to right. Whereas in case of bottom-up parsing, right-most derivations are used to accept an input stream and tokens are consumed from left to right. LR(K) and shift-reduce parsers are examples of bottom-up parsers.

In the previous work [2], formal verification of top down parsing was done. Few other preliminary results of this research were presented in [3,4] by formalizing some important concepts of context-free grammar. In this paper, the bottom-up parsing analysis of a sequence of tokens generated from the lexical analyzer is presented using Z by right most derivations. Ambiguity of the language is checked and its well-defined-ness is verified. Initially, formal definition of context-free grammar is given. In next, a right most derivation procedure is described by replacing non-terminals with terminals and non-terminals using bottom up approach. Then LR(1) parser is described first for a word and then extended to a language. The derivation procedure is defined to analyze a sequence of tokens using production rules of the context-free grammar. The parsing analysis for a language is specified by introducing recursion using derivations used in generation of a word. Ambiguity of a word is checked by specifying if there exists more than two right most derivation trees for a given words. The same notion is formalized for the language to check if it is ambiguous or well-defined. The formal specification is analyzed and validated using Z Eves tool set. The results of this paper will be used in our ongoing project on construction and verification of a compiler. The major objectives of this research are:

- Linking context-free grammar and formal techniques to be useful in the verification of a compiler;
- Preparing a synthesis of approaches to be used in the development of automated tools;
- Identifying and proposing an integration of existing traditional and formal approaches;
- Establishing a syntactically and semantically verified relationship between Z and context-free grammar.

Under the current development in formal methods, it is not possible to develop a complete and consistent software system using a single formal technique and hence integration of approaches is required. Although integration of approaches is a well-researched area [5-11], but there does not exist much work on formalization of automata and context-free languages. Dong *et al.* have described an integration of timed automata and Object Z [12,13]. Constable has proposed a formalization of few important concepts of automata theory using Nuprl which is a formal language [14,15]. A formal linkage is investigated between Petri-nets and Z notation in [16]. An integration of B, a formal technique, and UML, a semi-formal technique, is presented in [17,18]. Wechler has introduced few algebraic structures using fuzzy automata [19]. A formal treatment of fuzzy automata and language theory is discussed in [20]. In [21], an important notion of algebraic theory and automata theory is presented. Rest of the paper is organized as follows:

In Section 2, an introduction to formal methods is given. In Section 3, the role of context-free grammar in parsers for compiler construction is provided. Formal construction of LR(K), for K = 1, is given in Section 4. Model analysis for validating the specification is given in Section 5. Finally, conclusion and future work are discussed in Section 6.

2. Formal Methods

Formal methods are mathematical techniques and notations used for describing and analyzing properties of software and hardware systems. These techniques are based on discrete mathematics such as sets, sequences, relations, functions, graphs, automata, first order logic and higher order logic. Formal approaches may be classified mainly in terms of property oriented and model descriptive methods.

Property oriented formal methods are used to describe software in terms of properties and invariants defined that must be true. Model oriented formal methods are used to construct a model of a system focusing on both statics and dynamics of the system [22]. Although use of formal methods can be observed in almost all major areas of computer science but mainly their use can be found to improve quality by describing and specifying software systems in a well-defined and structured manner. Although there are various notations of formal methods but at the current stage of their development, it needs an integration of formal and existing traditional approaches for a consistent design and complete description of a system.

Z notation is a specification language used at an abstract level of modeling the systems. The Z is a model centered approach based on sets, sequences, bags, relations and first order predicate logic [23]. Usually, Z is used for specifying behavior of sequential programs by the abstract data types. Z is selected for this research to be linked with context-free language because both have abstract power of expressing the systems. The Z has standard set operators, for example, union, intersection, comprehensions, Cartesian products and power sets. The logic of Z is formulated using first order predicate calculus and refinements. The Z allows organizing a system into its smaller components using a powerful structure named as schema. The schema defines a way in which state of a system can be specified, refined and modified. Mathematical refinement is a promising aspect of Z supporting verifiable stepwise transformation of an abstract specification into an executable code [24]. Once a formal specification is written in Z, it can further be refined and transformed into an implemented system.

3. Parsers and Context Free Grammar

Verifying compiler is a branch of software engineering that deals to show or prove that a compiler behaves according to its language specification. Developing compiler using testing and formal methods are two most common techniques for compiler validation and verification. Testing of compiler has various disadvantages similar to computer programs testing, for example, it is hard to prove that compiler is completely error-free or optimized. On the other hand, the primary objective of writing a compiler is to prove that it is correct and error free. There are various research papers referring that many tested compilers have significant number of bugs and errors in the code [25]. Formal methods are used in compiler validation and verification to find proofs reducing complexity and ensuring correctness of the construction procedure.

Because of an accuracy needed, complexity in size and optimization required, compiler construction is an advanced research area, and as a result construction of a fully verified compiler is a challenge of the twenty first century. The main functionality of a compiler is to translate a source code to an executable optimized machine code correctly. The accuracy in compiler construction is required because the bugs in it can lead to an incorrect generated machine code even the source code is verified. Constructing and verifying parser is an important phase of a compiler whose functionality is to analyze a sequence of tokens to determine the grammatical structure with respect to a given grammar. Top-down and bottom-up parsing are two main approaches of parsing. In bottom-up parsing, tokens are consumed from left to right and right-most derivations are used from the given set of productions of the grammar. LR(K) and shift-reduce parsers are examples of bottom-up parsers. The context-free grammar (CFG) has an important role in verification of parser in a compiler.

CFG was developed by Chomsky who described linguistics in a grammatical form and converted into mathematical models providing a precise mechanism of describing the languages. The CFG provides a simple and an efficient approach of parsing, it can be determined to show that a particular pattern can be generated, and the way of generation is determined as well. Every CFG without null production has an equivalent grammar in Chomsky Normal Form (CNF). By equivalence, we mean both the grammars generate the same language. The CNF grammar is important in both theoretical and practical aspects. For example, using CNF, it can be decided for a given input if it can be accepted in a polynomial time algorithm. Context-free languages have their own limitations as well. For example, some operators which are well-defined in other models of automata theory do not behave well in context-free grammar. As an example, the intersection of two context-free languages, is not context-free in general. Similarly, the complement of a CFG may not be context-free.

There are various applications of context-free grammar in addition to compilers. Robotics, software engineering and maintenance, speech recognition are few application areas of it [26]. Applications of context-free grammar in pattern recognition increase an accuracy of the patterns to be recognized. This is because it can provide a higher level of abstraction by defining the semantics rules for patterns as compared to other specifications techniques, for example, strings and regular expressions. This abstract level semantic analysis can be used to reduce the false identification of the patterns [27]. The applications of pattern recognition can be observed everywhere from language processing to computer networks. In speech recognition, the spoken words can be generated by CFG using dynamic programming algorithms. In software engineering, the components in a source code are recognized using context-free grammar [28]. As the output of parsing is larger and less-ambiguous, therefore, for interactive voice response systems, the use of CFG can be highly effective [29,30].

4. LR(K) Formal Analysis

In this section, formal specification of LR(K), for K = 1, is described for parsing analysis of an input strings and a language. The ambiguity of a language is checked and its verification is done to generate the correct code. To formalize the parser, first the formal definition of context-free grammar is given. The context-free grammar is a 4-tuple (N, T, R, S0) where:

- N is a finite set of variables called non-terminal representing different types of clauses in a sentence and showing states in the parsing tree;
- T is a finite set of terminals where final contents of an input sentence are based on a set of terminals;
- R is a relation consisting of set of all the rules or productions of the grammar;
- S0 is a start variable used to represent the whole input string and initial state in the parsing tree.

In context-free grammar, every rule is of the form: S $\rightarrow$ t where S is a non-terminal consisting of a single character and t is a string containing only terminals or combination of terminals and non-terminals. The t might be an empty string. All notations of the type S $\rightarrow$ t are called rules or productions and are applied in a sequence to produce a parsing tree. The parsing tree ends with terminals termed as leaves of the tree where each internal node of the tree is a non-terminal producing one or more further nodes. The left hand side of a production rule is always a single non-terminal. Since all rules have non-terminals on the left hand side and, hence, can easily be replaced with the string on the right hand side of the production rule. The context in which the symbols occur is not important and, hence, the grammar is called context-free grammar. The CFG is always recognized by a finite state machine having a single infinite tap thereat. The current state is pushed at the start and is recovered at the end for keeping track of the nested units.

In the formal analysis, CFG is represented using Z notation consisting of 4-tuple as defined above. Mathematically, R in the definition of CFG is a relation from N to $(N \cup T)^*$ such that $\exists\ t \in (N \cup T)^*$, $S \in N$ and $(S, t) \in R$. The notation "*" represents to any combination of symbols of N and T. In the specification of CFG, X is defined as a set of symbols which is a collection of terminals or non-terminals. We define the sets of non-terminals by N and set of terminals by T based on the definition of X. The X, N and T are defined as sets at an abstract level of specification over which operators cannot be defined.

$$[X]; T \quad = X; N \quad = X$$

Formal definition of context-free grammar is given below and is represented by the schema *Grammer*. The schema consists of five components, *i.e.*, *terminals*, *non-terminals*, *symbols*, *productions* and *inistate* representing set of terminals, set of non-terminals, set of all symbols of the grammar, set of productions and start variable. The set of terminals is a type of power set of T, the set of non-terminals is a type of power set of N and the set of symbols is a power set of X. The *productions* are a set of rules defined by the relation between N and *seq* X. The start variable is of type of N. In the schema, it is described that there exists exactly one rule, (S0, *t*) ∈ *productions* where S0 is the start non-terminal and t is a string of type seq X. The components of the grammar are given in first part of the schema and invariants are defined in the second part of it.

```
Grammer
terminals: 𝔽 T
nonterminals: 𝔽 N
symbols: 𝔽 X
productions: N ↔ seq X
inistate: N
────────────────
inistate ∈ nonterminals
terminals ≠ ∅ ∧ nonterminals ≠ ∅
∀s: X | s ∈ symbols • s ∈ terminals ⇒ s ∉ nonterminals
∀s: X | s ∈ symbols • s ∈ nonterminals ⇒ s ∉ terminals
∀t: T | t ∈ terminals • t ∈ symbols
∀n: N | n ∈ nonterminals • n ∈ symbols
∀x: X | x ∈ symbols • x ∈ terminals ∨ x ∈ nonterminals
∀n: N | n ∈ dom productions • n ∈ nonterminals
∀s: seq X | s ∈ ran productions • ran s ⊂ symbols
∃s: seq X | s ∈ ran productions • (inistate, s) ∈ productions
```

4.1. Invariants

- The start variable is an element of non-terminals.
- The sets of terminals and non-terminals are non-empty.
- There does not exist any element which is common to both sets of terminals and non-terminals.
- Each element in the sets of terminals and non-terminals is an element of the set of symbols.
- Each element in the set of symbols is an element of the sets of terminals or non-terminals.
- The domain of production relation is a subset of the non-terminals.
- Each element in the range of production relation is a subset of set of symbols.
- There exists at least one production rule which contains start variable on the left hand side of it.

4.2. Derivation from Production Rules

In this section, process of derivations is described to be used for parsing analysis of words and languages using right most derivations by the production rules of the given grammar. In the formal procedure, the substitution are performed recursively to derive a string of terminal and non-terminal. Formal definition is given by the schema *Right Derivations*. The schema consist of three components *gram*, *single* and *multiple* representing grammar, single derivation and multiple derivations based on the production rules. First, a process of generating a word is described and then extended to generate the whole language. If *s*1 and *s*2 are two strings, we say *s*1 yields *s*2 if $\exists$ a ∈ N and b, *s*3, *s*4 ∈ seq X such that $s_1 = s_3 \frown \langle a \rangle \frown s_4$ and $s2 = s3 \frown b \frown s4$ where *s*4 is a sequence of terminals. It is noted that *a* is an element in set of variables, the ranges of sequences b, *s*3 are subsets of symbols and (a, b) is a production rule.

```
RightDerivations
gram: Grammer
single: seq X ↔ seq X
multiple: seq X ↔ seq X
────────────────
∀s1, s2: seq X | ran s1 ⊆ gram . symbols ∧ ran s2 ⊆ gram .
symbols • (s1, s2) ∈ single
    ⇒ (∃a: N; b: seq X; s3, s4: seq X
          | a ∈ gram . nonterminals
           ∧ (∀x: X • (x ∈ ran b ⇒ x ∈ gram . symbols))
           ∧ (a, b) ∈ gram . productions
           ∧ (∀t: T • (t ∈ ran s3 ⇒ t ∈ gram . symbols))
           ∧ (∀x: X • (x ∈ ran s4 ⇒ x ∈ gram . terminals))
          • s1 = s3 ⁀ ⟨a⟩ ⁀ s4 ∧ s2 = s3 ⁀ b ⁀ s4)
∀s1, s2: seq X | ran s1 ⊆ gram . symbols ∧ ran s2 ⊆ gram .
symbols • (s1, s2) ∈ multiple ⇒ (∃s3: seq (seq X)
          | 1 ⩽ #s3 ∧ (∀ss: seq X | ss ∈ ran s3 • ran ss ⊆ gram .
terminals) • (s1, s3 1) ∈ multiple ∧ (∀i: ℕ | i ∈ 2 .. #s3 • (s3 (i -
1), s3 i) ∈ multiple) ∧ (s3 (#s3), s2) ∈ multiple)
```

4.3. Words Parsing Analysis

Now we describe bottom-up parsing analysis of a sequence of tokens generated from the lexical analyzer using the schema LRW1. The schema consists of three components that is *gram*, *word*? and *multiple*. The definitions and types of *gram* and *multiple* are same as defined above. The variable *word*? is an input string to be generated by the parser using bottom-up approach based on the productions rules. A sequence of derivations using right most derivations is used as defined above. In the specification, it is described that *word*? can be generated if there exists a sequence of derivations consisting of order pairs (w(i), w(i + 1)) where w(i) produces w(i + 1). As we are describing LR(1), the input is read from left to right where production are used from the right side of the derivation procedure.

```
LRW1
gram: Grammer
word?: seq X
multiple: seq X ↔ seq X
────────────
∀w1, w2: seq X | ran w1 ⊆ gram . symbols ∧ ran w2 ⊆ gram .
terminals • (w1, w2) ∈ multiple ⇒ (∃w3: seq (seq X)
        | 1 ⩽ # w3 ∧ (∀w: seq X | w ∈ ran w3 • ran w ⊆ gram .
terminals) • (w1, w3 1) ∈ multiple
        ∧ ran w1 = {gram . inistate}
        ∧ (∀i: ℕ | i ∈ 2 .. # w3 • (w3 (i - 1), w3 i) ∈ multiple)
        ∧ (w3 (# w3), w2) ∈ multiple∧ w2 = word?)
```

Now we check the ambiguity of the word generated using the schema LRW1A. The schema consists of same three components *gram*, *word*? and *multiple* as in case of derivation of the word. In the schema, it is stated that word is ambiguously generated if there exists two derivations or parsing trees for the same word.

```
LRW1A
gram: Grammer
word?: seq X
multiple: seq X ↔ seq X
────────────
∀w1, w2: seq X | ran w1 ⊆ gram . symbols ∧ ran w2 ⊆ gram .
terminals • (w1, w2) ∈ multiple
   ⇒ (∃w3, w4: seq (seq X) | 1 ⩽ # w3 ∧ (∀w: seq X | w ∈ ran
w3 • ran w ⊆ gram . terminals) ∧ 1 ⩽ # w4
        ∧ (∀w: seq X | w ∈ ran w4 • ran w ⊆ gram . terminals)
       • (w1, w3 1) ∈ multiple ∧ ran w1 = {gram . inistate}
        ∧ (∀i: ℕ | i ∈ 2 .. # w3 • (w3 (i - 1), w3 i) ∈ multiple)
        ∧ (w3 (# w3), w2) ∈ multiple ∧ w2 = word?
        ∧ (w1, w4 1) ∈ multiple ∧ ran w1 = {gram . inistate}
        ∧ (∀i: ℕ | i ∈ 2 .. # w4 • (w4 (i - 1), w4 i) ∈ multiple)
        ∧ (w4 (# w4), w2) ∈ multiple ∧ w2 = word?)
```

In the schema given below, it is verified that the given word of the language is generated unambiguously. In the schema, it is stated that word is unambiguously generated if there exists two derivations or parsing trees for the same word then both the parsing trees must be same.

```
LRW1U
gram: Grammer
word?: seq X
multiple: seq X ↔ seq X
────────────
∀w1, w2: seq X | ran w1 ⊆ gram . symbols ∧ ran w2 ⊆ gram .
terminals • (w1, w2) ∈ multiple
   ⇒ (∃w3, w4: seq (seq X) | 1 ⩽ # w3
        ∧ (∀w: seq X | w ∈ ran w3 • ran w ⊆ gram . terminals)
        ∧ 1 ⩽ # w4 ∧ (∀w: seq X | w ∈ ran w4 • ran w ⊆ gram .
terminals) • (w1, w3 1) ∈ multiple
        ∧ ran w1 = {gram . inistate}
        ∧ (∀i: ℕ | i ∈ 2 .. # w3 • (w3 (i - 1), w3 i) ∈ multiple)
        ∧ (w3 (# w3), w2) ∈ multiple ∧ w2 = word?
        ∧ (w1, w4 1) ∈ multiple ∧ ran w1 = {gram . inistate}
        ∧ (∀i: ℕ | i ∈ 2 .. # w4 • (w4 (i - 1), w4 i) ∈ multiple)
        ∧ (w4 (# w4), w2) ∈ multiple ∧ w2 = word? ⇒ w3 = w4)
```

4.4. Language Parsing Analysis

Finally, verification of a language generated from a context-free grammar is done using right most derivation. The verification procedure of a word is defined above which is extended now to the whole language. For this purpose, the schema LRL1 given below is defined which consists of three components that is *gram*, *language*? and *multiple*. The variable *language*? is an input language to be generated by the parser using bottom-up approach. A sequence of derivations using right most derivations is used as in case of derivation of a word. In the specification, it is described that *language*? is generated if for any word in the language there exists a sequence of derivations consisting of order pairs (s(i), s(i + 1)) where s(i) produces s(i + 1).

```
LRL1
gram: Grammer
language?: seq (seq X)
multiple: seq X ↔ seq X
────────────
∀word: seq X | word ∈ ran language?
  • ∀w1, w2: seq X | ran w1 ⊆ gram . symbols ∧ ran w2 ⊆ gram
  terminals • (w1, w2) ∈ multiple ∧ ran w1 = {gram . inistate}
        ⇒ (∃w3: seq (seq X) | 1 ⩽ # w3
        ∧ (∀w: seq X | w ∈ ran w3 • ran w ⊆ gram . | terminals)
          • (w1, w3 1) ∈ multiple
            ∧ (∀i: ℕ | i ∈ 2 .. # w3 • (w3 (i - 1), w3 i) ∈ multiple)
            ∧ (w3 (# w3), w2) ∈ multiple ∧ word = w2)
```

In next, we check the ambiguity of the language using the schema LRL1A. The schema consists of same three components *gram*, *language*? and *multiple* as in case of derivation of the language. In the schema, it is stated that language is ambiguously generated if there a word in the language such that there exists two derivations or parsing trees for the word.

$$
\begin{array}{l}
\underline{\mathit{LRL1A}} \\
\mathit{gram}: \mathit{Grammer} \\
\mathit{language?}: \mathrm{seq}\,(\mathrm{seq}\,X) \\
\mathit{multiple}: \mathrm{seq}\,X \leftrightarrow \mathrm{seq}\,X \\
\hline
\exists \mathit{word}: \mathrm{seq}\,X \mid \mathit{word} \in \mathrm{ran}\,\mathit{language?} \\
\quad \bullet\ \forall \mathit{w1}, \mathit{w2}: \mathrm{seq}\,X \mid \ \mathrm{ran}\,\mathit{w1} \subseteq \mathit{gram}\,.\,\mathit{symbols} \wedge \mathrm{ran}\,\mathit{w2} \subseteq \mathit{gram} \\
.\,\mathit{terminals}\ \bullet\ (\mathit{w1}, \mathit{w2}) \in \mathit{multiple} \wedge \mathrm{ran}\,\mathit{w1} = \{\mathit{gram}\,.\,\mathit{inistate}\} \\
\qquad \Rightarrow (\exists \mathit{w3}, \mathit{w4}: \mathrm{seq}\,(\mathrm{seq}\,X) \mid\ 1 \leqslant \#\,\mathit{w3} \\
\qquad\qquad \wedge (\forall w: \mathrm{seq}\,X \mid\ w \in \mathrm{ran}\,\mathit{w3}\ \bullet\ \mathrm{ran}\,w \subseteq \mathit{gram}\,. \\
\mathit{terminals}) \wedge 1 \leqslant \#\,\mathit{w4} \wedge (\forall w: \mathrm{seq}\,X \mid\ w \in \mathrm{ran}\,\mathit{w4}\ \bullet\ \mathrm{ran}\,w \subseteq \\
\mathit{gram}\,.\,\mathit{terminals})\ \bullet\ (\mathit{w1}, \mathit{w3}\ 1) \in \mathit{multiple} \\
\qquad\qquad\qquad \wedge (\forall i: \mathbb{N} \mid\ i \in 2\,..\,\#\,\mathit{w3}\ \bullet\ (\mathit{w3}\ (i - 1), \mathit{w3}\ i) \in \mathit{multiple}) \\
\qquad\qquad\qquad \wedge (\mathit{w3}\ (\#\,\mathit{w3}), \mathit{w2}) \in \mathit{multiple}\ \wedge \mathit{word} = \mathit{w2} \\
\qquad\qquad\qquad \wedge (\mathit{w1}, \mathit{w4}\ 1) \in \mathit{multiple} \\
\qquad\qquad\qquad \wedge (\forall i: \mathbb{N} \mid\ i \in 2\,..\,\#\,\mathit{w4}\ \bullet\ (\mathit{w4}\ (i - 1), \mathit{w4}\ i) \in \mathit{multiple}) \\
\qquad\qquad\qquad \wedge (\mathit{w4}\ (\#\,\mathit{w4}), \mathit{w2}) \in \mathit{multiple} \wedge \mathit{word} = \mathit{w2})
\end{array}
$$

In the schema given below, it is verified that the given language is generated unambiguously. In the schema, it is stated that language is unambiguous if for any word if there exists two derivations or parsing trees for the same word then both the parsing trees must be same. Grammar used, input language to be generated and derivation rules are defined in first part of the schema and language derivation process is described in the second part of the schema. This complete our formal model for construction of LR(1).

$$
\begin{array}{l}
\underline{\mathit{LRL1U}} \\
\mathit{gram}: \mathit{Grammer} \\
\mathit{language?}: \mathrm{seq}\,(\mathrm{seq}\,X) \\
\mathit{multiple}: \mathrm{seq}\,X \leftrightarrow \mathrm{seq}\,X \\
\hline
\forall \mathit{word}: \mathrm{seq}\,X \mid\ \mathit{word} \in \mathrm{ran}\,\mathit{language?} \\
\quad \bullet\ \forall \mathit{w1}, \mathit{w2}: \mathrm{seq}\,X \mid\ \mathrm{ran}\,\mathit{w1} \subseteq \mathit{gram}\,.\,\mathit{symbols} \wedge \mathrm{ran}\,\mathit{w2} \subseteq \mathit{gram} \\
.\,\mathit{terminals}\ \bullet\ (\mathit{w1}, \mathit{w2}) \in \mathit{multiple} \wedge \mathrm{ran}\,\mathit{w1} = \{\mathit{gram}\,.\,\mathit{inistate}\} \\
\qquad \Rightarrow (\exists \mathit{w3}, \mathit{w4}: \mathrm{seq}\,(\mathrm{seq}\,X)\ \mid\ 1 \leqslant \#\,\mathit{w3} \\
\qquad\quad \wedge (\forall w: \mathrm{seq}\,X \mid\ w \in \mathrm{ran}\,\mathit{w3}\ \bullet\ \mathrm{ran}\,w \subseteq \mathit{gram}\,. \\
\mathit{terminals}) \wedge 1 \leqslant \#\,\mathit{w4} \wedge (\forall w: \mathrm{seq}\,X \mid\ w \in \mathrm{ran}\,\mathit{w4}\ \bullet\ \mathrm{ran}\,w \subseteq \mathit{gram} \\
.\,\mathit{terminals})\ \bullet\ (\mathit{w1}, \mathit{w3}\ 1) \in \mathit{multiple} \\
\qquad\quad \wedge (\forall i: \mathbb{N} \mid\ i \in 2\,..\,\#\,\mathit{w3}\ \bullet\ (\mathit{w3}\ (i - 1), \mathit{w3}\ i) \in \mathit{multiple}) \\
\qquad\quad \wedge (\mathit{w3}\ (\#\,\mathit{w3}), \mathit{w2}) \in \mathit{multiple} \wedge \mathit{word} = \mathit{w2} \\
\qquad\quad \wedge (\mathit{w1}, \mathit{w4}\ 1) \in \mathit{multiple} \\
\qquad\quad \wedge (\forall i: \mathbb{N} \mid\ i \in 2\,..\,\#\,\mathit{w4}\ \bullet\ (\mathit{w4}\ (i - 1), \mathit{w4}\ i) \in \mathit{multiple}) \\
\qquad\quad \wedge (\mathit{w4}\ (\#\,\mathit{w4}), \mathit{w2}) \in \mathit{multiple} \wedge \mathit{word} = \mathit{w2} \Rightarrow \mathit{w3} = \mathit{w4})
\end{array}
$$

5. Model Analysis

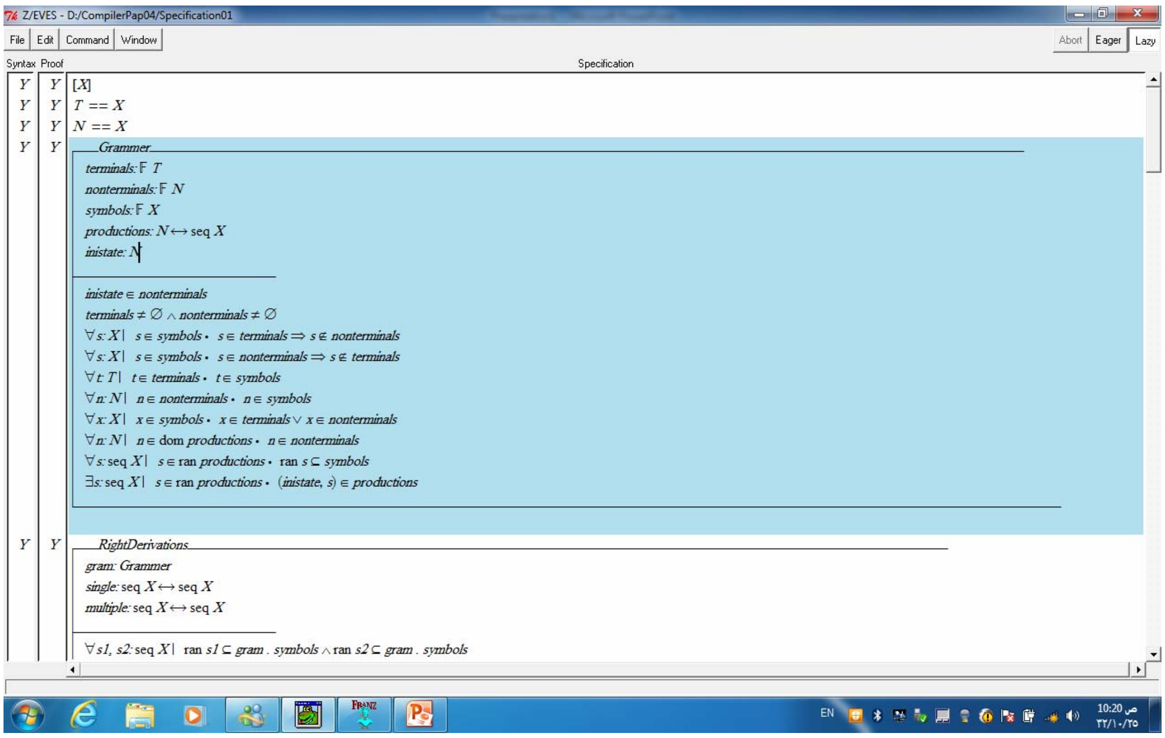
In this section, formal analysis is done for the specification. Although computer tools are rigorously used for the formal specification but, on the hand, there does not exist any real computer tool which may assure about complete correctness of a formal model. Therefore, even the specification is well-written using any of the formal specification languages it may contain potential bugs or errors. That is an art of writing a formal specification never guarantee that the system is correct, complete and consistent. But if the specification is checked and analyzed with a computer tool it certainly increases the confidence over the system to be developed by identifying the errors, if exists, in the syntax and semantics of the formal specification.

The Z/Eves is one of the powerful tools which is used for analyzing the specification written for construction of LR(1). A snapshot of the tool for analyzing the formal specification using Z/Eves tool is presented in **Figure 1**. The first column on the left of the figure shows status of the syntax checking and the second column represents the proof correctness of the specification. The symbol "Y" stands that the specification is correct syntactically and proof is also correct while the symbol "N" shows that errors exist which can be listed with the tool support. All the schemas are checked to prove that specification is correct in syntax and has a correct proof. Some proofs were conducted by reduction and rewriting techniques available in the tool.

Summary of the results of the formal specification is presented in **Table 1**. In the first column of the table, name of schema is given for which the specification is described. These schemas are analyzed by using the model exploration techniques provided in the Z/Eves tool. The symbol "Y" in column 2 indicates that all the schemas are well-written and proved automatically. Similarly, domain checking, reduction and proof by reduction are represented in columns 3, 4 and 5, respectively. The character "Y*" annotated with "*" describes that the schemas are proved by performing reduction on the predi-

Table 1. Results of model analysis.

Schema Name	Syntax Type Check	Domain Check	Reduction	Proof
Grammer	Y	Y	Y	Y
Right Derivations	Y	Y	Y	Y
LRW1	Y	Y	Y*	Y
LRW1A	Y	Y	Y	Y
LRW1U	Y	Y	Y	Y
LRL1	Y	Y	Y	Y
LRL1A	Y	Y	Y*	Y
LRL1U	Y	Y	Y*	Y



```
Y  Y  [X]
Y  Y  T == X
Y  Y  N == X
Y  Y  Grammer
        terminals: 𝔽 T
        nonterminals: 𝔽 N
        symbols: 𝔽 X
        productions: N ↔ seq X
        inistate: N
        ─────────────
        inistate ∈ nonterminals
        terminals ≠ ∅ ∧ nonterminals ≠ ∅
        ∀s: X | s ∈ symbols • s ∈ terminals ⇒ s ∉ nonterminals
        ∀s: X | s ∈ symbols • s ∈ nonterminals ⇒ s ∉ terminals
        ∀t: T | t ∈ terminals • t ∈ symbols
        ∀n: N | n ∈ nonterminals • n ∈ symbols
        ∀x: X | x ∈ symbols • x ∈ terminals ∨ x ∈ nonterminals
        ∀n: N | n ∈ dom productions • n ∈ nonterminals
        ∀s: seq X | s ∈ ran productions • ran s ⊆ symbols
        ∃s: seq X | s ∈ ran productions • (inistate, s) ∈ productions
Y  Y  RightDerivations
        gram: Grammer
        single: seq X ↔ seq X
        multiple: seq X ↔ seq X
        ─────────────
        ∀s1, s2: seq X | ran s1 ⊆ gram . symbols ∧ ran s2 ⊆ gram . symbols
```

Figure 1. Snapshot of the model analysis.

cates to make the specification more meaningful.

6. Conclusions and Future Work

Design, construction and verification of a correct compiler is more important than verification of the source programs to be compiled. Verification of a compiler assures guarantee that the executable code generated from the source code behaves exactly as described in the source program. Parsing analysis is an important part of compiler construction. Bottom-up parsing is suitable for automatic parser generation and handles a larger class of grammars. In this paper, formal procedure of LR(1) parser is proposed and verified. Identification and analysis of ambiguities is provided which is a real challenge in parsers development. Regular expressions and context-free grammars are widely used in construction of the compiler. Regular expressions are not much powerful and are used to identify tokens from the source program. The design and construction of a complier can be benefited by linking context-free grammar to Z specification. This is because Z enhances reliability and correctness being abstract in nature and having computer tool support. In this research, formal specification helped us to make it possible describing unambiguous and easy to understand the resultant formal model. The specification is verified and validated using Z/Eves tool.

An extensive survey of existing work was performed before initiating this research. Some of the interesting works [31-39] were found but our approach is different because of abstract and conceptual level integration of CFG and Z. In the benefits of using Z, every object is assigned a unique type providing a useful programming practice. Several type checking tools exist to support the formal specification. The Z/Eves is a powerful tool to prove and analyze the specification. The rich mathematical notations made it possible to reason about behavior of a system more effectively. Formalization of some other concepts, useful in compiler verification, are in progress and will appear soon in our future work.

REFERENCES

[1] C. J. Burgess, "The Role of Formal Methods in Software Engineering Education and Industry," *Technical Report*, University of Bristol, Bristol, 1995.

[2] K. A. Buragga and N. A. Zafar, "Formal Parsing Analysis of Context-Free Grammar Using Left Most Derivations," *International Conference on Software Engineering Advances*, 2011.

[3] N. A. Zafar, S. A. Khan and B. Kamran, "Formal Procedure of Deriving Language from Context-Free Grammar," *International Conference on Intelligence and Information Technology*, Vol. 1, 2010, pp. 533-536.

[4] N. A. Zafar and B. Kamran, "Formal Construction of Possible Operators on Context-Free Grammar," *International Conference on Intelligence and Information Technology*, 2010.

[5] H. Beek, A. Fantechi, S. Gnesi and F. Mazzanti, "State/

Event-Based Software Model Checking," *Integrated Formal Methods*, Springer, Berlin, 2004, pp. 128-147.

[6] O. Hasan and S. Tahar, "Verification of Probabilistic Properties in the HOL Theorem Prover," *Integrated Formal Methods*, Springer, Berlin, 2007, pp. 333-352.

[7] F. Gervais, M. Frappier and R. Laleau, "Synthesizing B Specifications from EB3 Attribute Definitions," *Integrated Formal Methods*, Springer, Berlin, 2005, pp. 207-226. doi:10.1007/11589976_13

[8] K. Araki, A. Galloway and K. Taguchi, "Integrated Formal Methods," *Proceedings of the* 1*st International Conference on Integrated Formal Methods*, Springer, Berlin, 1999.

[9] B. Akbarpour, S. Tahar and A. Dekdouk, "Formalization of Cadence SPW Fixed-Point Arithmetic in HOL," *Integrated Formal Methods*, Springer, Berlin, 2002, pp. 185-204.

[10] J. Derrick and G. Smith, "Structural Refinement of Object-Z/CSP Specifications," The Institute of Finance Management, Springer, Berlin, 2000, pp. 194-213.

[11] T. B. Raymond, "Integrating Formal Methods by Unifying Abstractions," Springer, Berlin, 2004, pp. 441-460.

[12] J. S. Dong, R. Duke and P. Hao, "Integrating Object-Z with Timed Automata," 2005, pp. 488-497.

[13] J. S. Dong, *et al*., "Timed Patterns: TCOZ to Timed Automata," *The* 6*th International Conference on Formal Engineering Methods*, 2004, pp. 483-498.

[14] R. L. Constable, *et al*., "Formalizing Automata II: Decidable Properties," *Technical Report*, Cornell University, Cornell, 1997.

[15] R. L. Constable, *et al*., "Constructively Formalizing Automata Theory," *Foundations of Computing Series*, MIT Press, Cambridge, 2000.

[16] M. Heiner and M. Heisel, "Modeling Safety Critical Systems with Z and Petri Nets," *International Conference on Computer Safety*, *Reliability and Security*, Springer, Berlin, 1999, pp. 361-374. doi:10.1007/3-540-48249-0_31

[17] H. Leading and J. Souquieres, "Integration of UML and B Specification Techniques: Systematic Transformation from OCL Expressions into B," *Asia-Pacific Software Engineering Conference*, 2002, pp. 495-504.

[18] H. Leading and J. Souquieres, "Integration of UML Views Using B Notation," *Proceedings of Workshop on Integration and Transformation of UML Models*, 2002.

[19] W. Wechler, "The Concept of Fuzziness in Automata and Language Theory," Akademic-Verlag, Berlin, 1978.

[20] N. M. John and S. M. Davender, "Fuzzy Automata and Languages: Theory and Applications," Chapman & Hall, London, 2002.

[21] M. Ito, "Algebraic Theory of Automata and Languages," World Scientific Publishing Co., Singapore, 2004. doi:10.1142/9789812562685

[22] M. Brendan and J. S. Dong, "Blending Object-Z and Timed CSP: An Introduction to TCOZ," 20*th International Conference on Software Engineering*, IEEE Computer Society, Kyoto, 1998.

[23] J. M. Spivey, "The Z Notation: A Reference Manual," Printice-Hall, Austin, 1989.

[24] J. M. Wing, "A Specifier, Introduction to Formal Methods," *IEEE Computer*, Vol. 23, No. 9, 1990, pp. 8-24. doi:10.1109/2.58215

[25] C. Lindig, "Random Testing of C Calling Conventions," ACM, 2005.

[26] J. A. Anderson, "Automata Theory with Modern Applications," Cambridge University Press, Cambridge, 2006. doi:10.1017/CBO9780511607202

[27] H. C. Young, J. Moscola and J. W. Lockwood, "Context-Free Grammar Based Token Tagger in Reconfigurable Devices," *Proceedings of International Conference of Data Engineering*, 2005, p. 78.

[28] M. V. D. Brand, A. Sellink and C. Verhoef, "Generation of Components for Software Renovation Factories from Context-Free Grammars," *Counselors of Real Estate*, 2001, pp. 144-153.

[29] M. Balakrishna, D. Moldovan and E. K. Cave, "Automatic Creation and Tuning of Context-Free Grammars for Interactive Voice Response Systems," *IEEE NLP-KE*, 2005, pp. 158-163.

[30] L. Pedersen and H. Reza, "A Formal Specification of a Programming Language: Design of Pit," 2*nd International Symposium on Leveraging Applications of Formal Methods*, *Verification and Validation*, 2008, pp. 111-118.

[31] D. P. Tuan, "Computing with Words in Formal Methods," Technical Report, University of Canberra, Canberra, 2000.

[32] S. A. Vilkomir and J. P. Bowen, "Formalization of Software Testing Criterion," South Bank University, London, 2001.

[33] A. Hall, "Correctness by Construction: Integrating Formality into a Commercial Development Process," *Praxis Critical Systems Limited*, Springer, Berlin, Vol. 2391, 2002, pp. 139-157.

[34] B. A. L. Gwandu and D. J. Creasey, "Importance of Formal Specification in the Design of Hardware Systems," Birmingham University, Birmingham, 1994.

[35] D. K. Kaynar and N. Lynchn, "The Theory of Timed I/O Automata," Morgan & Claypool Publishers, 2006.

[36] D. Jackson, I. Schechter and I. Shlyakhter, "Alcoa: The Alloy Constraint Analyzer," *Proceedings of the 22nd International Conference of Software Engineering*, 2000, pp. 730-733.

[37] D. Aspinall and L. Beringer, "Optimisation Validation," *Electronic Notes in Theoretical Computer Science*, Vol. 176, No. 3, 2007, pp. 37-59. doi:10.1016/j.entcs.2006.06.017

[38] S. Briaisa and U. Nestmannb, "A Formal Semantics for Protocol Narrations," *Theoretical Computer Science*, Vol. 389, No. 3, 2007, pp. 484-511.

[39] L. Freitas, J. Woodcock and Y. Zhang, "Verifying the CICS File Control API with Z/Eves: An Experiment in the Verified Software Repository," *Science of Computer Programming*, Vol. 74, No. 4, 2009, pp. 197-218. doi:10.1016/j.scico.2008.09.012