Proof-Carrying Model for Parsing Techniques

Mohamed A. El-Zawawy[1,2]

[1] College of Computer and Information Sciences,
Al Imam Mohammad Ibn Saud Islamic University (IMSIU)
Riyadh, Kingdom of Saudi Arabia
[2] Department of Mathematics, Faculty of Science, Cairo University
Giza 12613, Egypt
maelzawawy@cu.edu.eg

Abstract. This paper presents new approaches to common parsing algorithms. The new approach utilizes the concept of inference rule. Therefore the new approach is simple, yet powerful enough to overcome the performance of traditional techniques. The new approach is basically composed of systems of inference rules.

Mathematical proofs of the equivalence between proposed systems and classical algorithms are outlined in the paper. The proposed technique provides a correctness verification (a derivation of inference rules) for each parsing process. These verifications are required in modern applications such as mobile computing. Results of experimental proving the efficiency of the proposed systems and their produced verifications are shown in the paper.

Keywords: Inference rules, LL(1) parsers, Operator-precedence passers, LR parsers, Parsing.

1 Introduction

The parsing process [14,5] aims at analyzing and checking the structure of a given input (computer program) with respect to a specific grammar. Each grammar can typically produce an infinite number of *sentences* (programs). However grammars [28,8] have finite sizes. Therefore a grammar abstracts briefly an infinite number of program architectures of a specific type (structural, functional, object-oriented, etc).

Parsing is very critical for several reasons [21]. Parsing process is essential for further processing of parsed objects (programs). For example in natural languages processing, recognizing the verb of a sentence facilitates the sentence translation. Parsing process is also important to understand grammars which summarize our realization of a class of programs. Error-recovering parsers [29] are important tools for correcting some program faults. Theoretical foundations of parsing excuse it from being described as esoteric or as a mathematical discipline. Using string cutting and pasting, parsing can be realized and applied.

Parsing techniques [28,8] are classified into two categories; top-down and bottom-up. A parsing technique that tries to establish a derivation for the input program starting from the start symbol is classified as a top-down algorithm. If a parsing technique starts with the input program and rolls back trying to establish the derivation in the reverse

B. Murgante et al. (Eds.): ICCSA 2014, Part V, LNCS 8583, pp. 1–14, 2014.
© Springer International Publishing Switzerland 2014

order until reaching the start symbol, the technique is then described as bottom-up. Derivations produced by a bottom-up (top-down) algorithm are right-most (left-most). In top-down (bottom-up) parsers, parse trees are constructed from the top (bottom) downwards (upwards). Common examples of top-down and bottom- up parsers are LL(1) and LR(1) [3], respectively.

Some applications like proof-carrying code [23,17] and mobile computing [30] require the parser to associate each parse tree with a correctness proof. This proof ensures the validity of the tree. Unfortunately most of existing parsers are algorithmic in their style and hence build the parse tree but not the correctness proof. Proposing new sittings for common parsing algorithms so that correctness proofs are among outputs of these algorithms (rather than parse trees) is a requirement of many modern computing applications.

This paper introduces new models for common existing parsing techniques such as LL(1), operator-precedence, SLR, LR(1), and LALR. The proposed models are built mainly of inference rules whose outputs in this case are pairs of parse trees and correctness proofs. Rather than providing the required correctness proofs, the new models are faster than their corresponding ones as is confirmed by experimental results. Mathematical proofs of the equivalence between each proposed model and its corresponding original parsing technique are outlined in the paper. The proposed models are supported with illustrative parsing examples that are detailed.

Motivation

Systems of inference rules were found very useful for optimization phases of compilers [11,12]. This is so as they provide compact correctness proofs for optimizations. These proofs are required by applications like proof-carrying code and mobile computing. The motivation of this paper is the need for parsing techniques that associate each parsing process with a simply-structured correctness proof. The research behind this paper reveals that parsing techniques having the form of inference rules are the perfect choice to build the required proofs. The resulting proofs have the shape of rule derivations as in Figures 2 and 5.

Contributions

Contributions of the paper are the following:

1. A new approach to LL parsing techniques in the form of system of inference rules. The new approach was found more efficient than the traditional one.
2. A new setting for operator-precedence parsers.
3. A novel technique to achieve LR parsing methods.

Paper Outline

The rest of the paper is organized as follows. Related work is discussed in Section 2. Section 3 presents the new technique for LL(1) parsers. The systems of inference rules

for LR parsers and operator-precedence parsers are shown in Sections 4.1 and 4.2, respectively. Each of Sections 3, 4.1, and 4.2 includes mathematical proofs of the correctness of its proposed technique. Section 5 shows the experimental results. A conclusion to the paper is in Section 6.

2 Related Work

Generalized LR (GLR) parsers [25,18] are used to handle ambiguous grammars whose parsing tables compilation may last for minutes for large grammars. These parsers are not convenient for natural-language grammars because parsing tables can become very large. For natural languages more specific parsers [9,20] were designed. Such parsers, typically, support PM context-free grammars. Some attempts [10,2,13] towards parsers for visual languages were done. However problems with parsing visual languages include that parsing tables need to be recomputed with each grammar. Most parsers [24,19] using no parsing tables are inefficient and restricted to a small categories of grammars.

In dependent grammars [22,7], semantic values are associated with variables. Therefore these grammars are convenient for constrain parsing [6,15]. Common attributes not covered by context-free grammars, like intended indentations and field lengths in data types of programming languages are determined by dependent grammars. Researchers have been trying to extend classical parsing algorithms to support dependent parsing [16,32]. For example, in [16] a point-free algorithm language for dependent grammars is proposed. The language algorithms are similar to classical algorithms for context-free parsing. Hence the point-free language acts as a tool to get point-free grammars from classical dependent ones.

Boolean grammars [27,26] are grammars equipped with operations that are explicitly set-theoretic and used to express rules formally defined by equations of the language equations. Boolean grammars are extensions of context-free grammars. Much research was done to extend algorithms of the later grammars to that of the former ones. Other attempts were done to generalize LR parsing algorithm to cover Boolean grammars. This extension included traditional LR operations, Reduce and Shift, and a new third operation called, Invalidate. This third action reverses a prior reduction. The generalization idea was to make the algorithm complectly different from its prototype.

Although classical top-down parsers [21,29] simplicity and ability to treat empty rules, they can not treat many sorts of left-recursions. In the same time, it is not preferred to normalize a left-recursion grammar as the rules semantic-structure gets affected. On the other hand, bottom-up parsers [21,29] are able to treat left-recursion but not empty rules. The research in [31,4] proposed a deterministic cancelation parser whose structure is recursive-descent. This parser can manipulate left-recursions and empty rules via a combination of bottom-up and top-down applications towards building the parse tree. The challenge of these parsers is to manipulate all sorts of left-recursions, such as hidden and indirect left recursions.

Unfortunately, none of the parsers reviewed above associates each parsing process with a justification or a correctness proof. Therefore all these parsers are not applicable in modern applications of mobile computing and proof-carrying code.

3 New Approach for Top-Down Parsers: LL(1)

LL(1) [3] is a top-down parsing algorithm. For a given input string and starting from the start symbol of the given grammar, top-down parsers tries to create a parse tree using depth-first methods. Hence top-down parsers can be realized as searching methods for leftmost derivations for input strings. LL(1) is predictive in the sense that it is able to decide the rule to apply based only on the currently processed nonterminal and following input symbol. This is only possibly if the grammar has a specific form called LL(1). The LL(1) parser uses a stack and a parse table called LL(1)-table. The stack stores the productions that the parser is to apply.

The LL(1)-table columns correspond to terminals of the grammar plus an extra column for the \$ symbol (the marker for the input symbol end). Rows of LL(1)-table correspond to nonterminals of the LL(1) grammar. Each cell of the LL(1)-table is either empty or including a single grammar production. Therefore the table includes the parser actions to be taken based on the top value of the stack and the current input symbol.

$$\frac{}{\epsilon : \epsilon \rightarrow \$}\ (\text{Base}) \qquad \frac{a \in \text{first}(T) \qquad (N \rightarrow T) \in P \qquad a\alpha : TM \rightarrow \$}{a\alpha : NM \rightarrow \$}\ (\text{Predict}_1)$$

$$\frac{a \in \text{follow}(N) \qquad N \rightarrow \epsilon \qquad a\alpha : M \rightarrow \$}{a\alpha : NM \rightarrow \$}\ (\text{Predict}_2) \qquad \frac{\alpha : \beta \rightarrow \$}{a\alpha : a\beta \rightarrow \$}\ (\text{Match})$$

Fig. 1. Inference rules for LL(1) parser

Figure 1 presents a new model for the LL(1) functionality in the form of a system of inference rules. The following definition is necessary for linking the semantics and the use of inference rules in Figure 1 to functionality of LL(1). The definition is also necessary for proving equivalence between the original model and our new one.

Definition 1. *Let (S, N, T, P) be a LL(1) grammar. Then* ***LL(1)-infer*** *is the set of strings $w \in T^+$ such that $w : S \rightarrow \$$ is derivable in the system of inference rules of Figure 1. Moreover* ***path(w,ll(1))*** *is the string derivation obtained by applying the grammar rules in the derivation of $w : S \rightarrow \$$. This rule applications is intended to respect the order of the rule appearances in the direction bottom-up of the derivation.*

Our proposed model works as follows. For a given string w, we start by trying to derive $w : S \rightarrow \$$ using the system of inference rules. This is equivalent to the start of the LL(1) algorithm; reading the first token of input and pushing on the stack the start symbol. In the LL(1) method and using the LL(1)-table, there are three possible cases for the input token, a, and the top stack symbol x. In the first case $x = a = \$$, meaning that w is correct. This case is modeled through the rule *(Base)*. In the second case, $x = a \neq \$$, the algorithm pops x and moves to next input token. This is achieved via the rule *(Match)*. In the third case x is a nonterminal and $x \neq a$. In this case, the LL(1)-table is referenced at the cell (x, a) towards a rule application. There are three

possible sub-cases for this case. The first subcase happens when the cell (x, a) is empty meaning that it is not possible to construct the derivation and hence w is wrong. The second subcase happens when there is a rule at in the cell (x, a) and the left hand side of the rule is not empty. The application of this rule is achieved by the rule ($Predict_1$). The third subcase happens when the left hand side of the rule is empty. This subcase is treated via the rule ($Predict_2$).

The following theorem proves the equivalence between our proposed model and the original (one which does not provide the correctness verifications).

Theorem 1. *Let (S, N, T, P) be a $LL(1)$ grammar and $w \in T^+$. Then $w \in LL(1) -$ infer if and only if w is accepted by the $LL(1)$-algorithm. Moreover the left-most derivation obtained by the LL(1)-algorithm is the same as* ***path(w,ll(1))****.*

Proof. The LL(1)-algorithm has the following cases:

- The case (*Match*) meaning a symbol a is on the top of the stack and is also the first symbol of the current input string. In this case, the $LL(1)$ algorithm removes a from the stack and input. Also in this case, the rule *(Match)* of Figure 1 applies and does the same action as the $LL(1)$-algorithm.
- The case ($Predict_1$): meaning the top of the stack is a nonterminal symbol x and the first symbol of the current input string is a terminal symbol a. In this case, the algorithm references the LL(1)-table at location (x, a) towards a rule application and pushes into the stack the r.h.s. of rule. Suppose this rule is $N \to T$. In this case $a \in \text{first}(T)$ and the rule ($Predict_1$) of Figure 1 applies and does the same action as the $LL(1)$-algorithm.
- The case ($Predict_2$): this case is similar to the previous one except that $T = \epsilon$. In this case $a \in \text{follow}(N)$ and the LL(1)-algorithm removes N form the stack. Clearly, in this case, the rule (Predict$_2$) of Figure 1 applies and does the same action as the $LL(1)$-algorithm.
- The case (*Accept*), in this case the stack and input have only \$. This case is covered by the rule *(Base)* of Figure 1.

The following two lemmas formalize the relationship between parse trees obtained in our prosed model of inference rules and that obtained using the original LL(1)-algorithm. The lemmas are easily proved by a structure induction on the inference-rule derivations.

Lemma 1. *Let (S, N, T, P) be a $LL(1)$ grammar and $w \in LL(1) -$ infer. Then path(w) is left-most.*

Lemma 2. *Let (S, N, T, P) be an $LL(1)$ grammar. Then for $w \in T^+$,*

$$S \Rightarrow^+ w \iff w : S \to \$.$$

For the grammar consists of the rules:

$$S \to aBa, B \to bB, \text{ and } B \to \epsilon.$$

The upper part of Figure 2 shows the derivation for proving the correctness of the statement *abba* using inference rules of Figure 1. The obtained derivation is the following:

$$S \Longrightarrow_{lm} aBa \Longrightarrow_{lm} abBa \Longrightarrow_{lm} abbBa \Longrightarrow_{lm} abba.$$

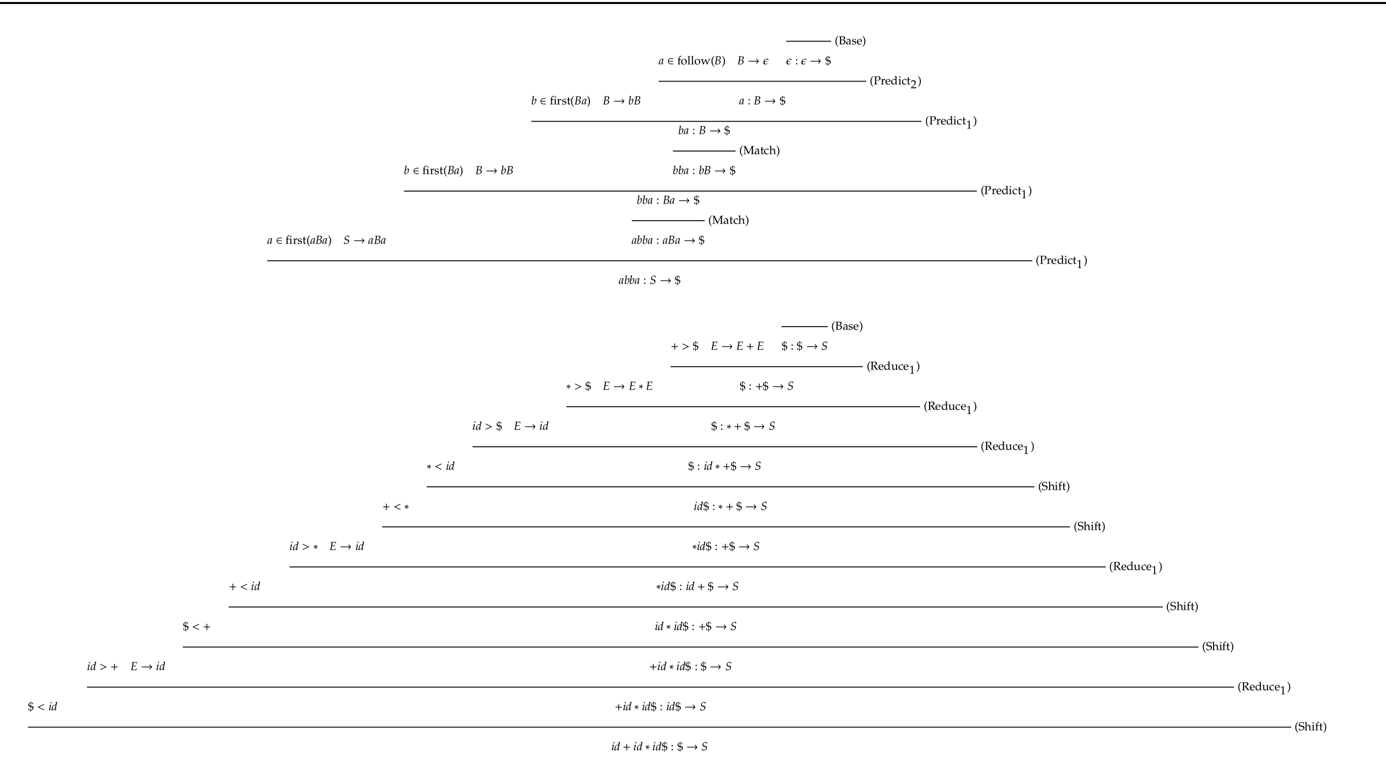

Fig. 2. Derivation Examples

4 New Approaches for Bottom-Up Parsers

As suggested by their names, bottom-up parsers [3] work oppositely to top-down parsers. Starting with the input string, a bottom-up parser proceeds upwards from leaves to the start symbol. Hence a bottom-up parser works backwards and applies the grammar rules in a reverse direction until arriving at the start symbol. This reverse application amounts to finding a substring of the stack content that coincides with the r.h.s. of some production of the grammar. Then the l.h.s. of this production replaces the found substring. Hence a reduction takes place. The idea is to keep reducing until arriving at the start symbol, therefore the string is correct, otherwise it is not correct. In the following, we show how common bottom-up parsers (LR and operator-precedence parsers) can be modeled using systems of inference rules.

4.1 LR Parsers Family

LR parsers [3] work as following. Fixing handles depends on the stack content and hence on the context of the parsing process. The LR parsers do not push tokens only into the stack, but push as well state numbers in alternation with tokens. The state numbers describe the stack content. The state on the top of the stack determines whether the next action is to move (shift) a new state to the stack (for the next symbol of input) or is to reduce.

Two tables are used by LR parsers; an action table and a goto table. For the action table, columns correspond to terminals and rows correspond to state numbers. For the goto table, columns correspond to nonterminals and rows correspond to state numbers. The entry of the action table at the intersection (s, a) determines the action to be taken when the next terminal is a and s is the state on the top of the stack. Figure 3 models possible actions in the form of inference rules. One possible action is to *shift* and is modeled by the rule *(Shift)*. Another action is to *reduce* and is modeled by the rule *(Reduce)*. The *accept* case is modeled by the rule *(Base)*. In case of error, it is not possible to find a derivation for the given input string. The goto-table entry at the intersection (s, x) determines the state to be inserted into the stack after reducing x when the top of stack is s.

For a given string w, if it is possible to derive $w\$: 0 \rightarrow S$ using inference rules of Figure 3, then w is correct (accepted), otherwise it is wrong. This is corresponding

$$\frac{(n, \$) = \text{accept}}{\$: \beta n \rightarrow S}\ (\text{Base}) \qquad \frac{\text{goto}(n_1, a) = sn_2 \qquad \alpha : \beta n_1 a n_2 \rightarrow S}{a\alpha : \beta n_1 \rightarrow S}\ (\text{Shift})$$

$$\frac{\begin{array}{l}\text{goto}(n_1, a) = r \text{ by } (T \rightarrow \gamma) \\ \# - \text{terminals}(\gamma) = m\end{array} \qquad \begin{array}{l}\text{deduct}(\beta n_1, 2m) = \delta n_2 \\ \text{goto}(n_2, T) = n_3 \\ a\alpha : \delta T n_3 \rightarrow S\end{array}}{a\alpha : \beta n_1 \rightarrow S}\ (\text{Reduce})$$

Fig. 3. Inference rules for SLR, LR(1), and LALR parsers

to pushing the state 0 into the stack and continuing in the shift-reduce process until reaching an accept or a reject.

Definition 2 recalls important concepts of grammars. Definition 3 introduces necessary concepts towards proving soundness of inference rules of Figure 3.

Definition 2.
- *A LR parser using SLR parsing table for a grammar G is called a SLR parser for G.*
- *If a grammar G has a SLR parsing table, it is called SLR-grammar.*[1]

Definition 3. *Let (S, N, T, P) be a SLR grammar. Then* ***der-stg*** *is the set of strings $w \in T^+$ such that $w : 0\$ \rightarrow S$ is derivable in the system of inference rules of Figure 3. Moreover* ***way(w,slr)*** *is the string derivation obtained by applying the grammar rules in the obtained derivation. This rule applications is intended to respect the order of the rules appearances in the direction bottom-up of the derivation.*

The following theorem proves the soundness of the proposed model and its equivalence to the original SLR model.

Theorem 2. *Let (S, N, T, P) be a SLR grammar and $w \in T^+$. Then $w \in$ der-stg if and only if w is accepted by the SLR-algorithm. Moreover the right-most derivation obtained by the SLR-algorithm is the same as* ***way(w,slr)****.*

Proof. The SLR algorithm has the following possible actions cases:

- The *shift* action: in this case the intersection of the number, n_1, at the top of the stack and the symbol a at the beginning of the input, in the SLR table, is sn_2. In this case the SLR-algorithm pushes a followed by n_2 into the stack. Also in this case, the rule *(Shift)* of Figure 3 applies and does the same action as the SLR-algorithm.
- The *reduce* action: in this case the intersection of the number n_1 at the top of the stack and the symbol a at the beginning of the input, in the SLR-table, is r by $T \rightarrow \gamma$. In this case, a number (twice the number of symbols in γ) of elements is removed from the stack and T is added to the stack. If n_2 is the number prior to T in the stack, then the intersection of T and n_2 in the goto table is added to the stack. In this case, the rule *(Reduce)* of Figure 3 applies and does the same action as the SLR-algorithm.
- The case of an acceptance which is equivalent to applying the rule *Base*.

The relationship between parse trees resulting from our prosed system of inference rules and that resulting from original SLR-algorithm are formalized by the following two lemmas. A straightforward structure induction on the inference-rule derivations completes the lemma proofs.

Lemma 3. *Let (S, N, T, P) be a SLR-grammar and $w \in$ der-stg. Then* ***way(w,slr)*** *is a right-most derivation.*

[1] Recall that every SLR grammar is unambiguous, but not every unambiguous grammar is SLR grammar [3].

Lemma 4. *Let (S, N, T, P) be a SLR-grammar. Then for $w \in T^+$,*

$$S \Rightarrow^+ w \iff w\$: 0 \rightarrow S.$$

Consider the following grammar.

$$E \rightarrow E + E, E \rightarrow T, T \rightarrow T * F, T \rightarrow F, F \rightarrow (E), \text{ and } F \rightarrow id.$$

Figure 5 presents a derivation for proving the correctness of the statement $id * id + id$ using inference rules of Figure 3. The following is the obtained derivation.

$$E \Longrightarrow_{rm} E + T \Longrightarrow_{rm} E + F \Longrightarrow_{rm} E + id \Longrightarrow_{rm} T * F + id \Longrightarrow_{rm}$$

$$T * id + id \Longrightarrow_{rm} F * id + id \Longrightarrow_{rm} id * id + id.$$

Remark 1. All results above about SLR-grammars and parsers are correct and applicable for LR(1)-grammars and parsers and for LALR-grammars and parsers.

4.2 Operator-Precedence Parsing

Operator-precedence parsing [29] is a simple parsing technique suitable for particular grammars. This grammars are *operator-grammars* characterized by the absence of ϵ and consecutive nonterminals in l.h.s. of their productions. A precedence relation is defined on the set of terminals: $a < b$ means that b has higher precedence than a, where $a = b$ means that b enjoys the same precedence as a. Classical concepts of operators precedence and associativity determine the convenient precedence relations. The handle of a right-sentential form is determined by the precedence relation. Hence the left (right) end of the handle is marked by $<$ ($>$). Figure 4 presents a system of inference rules to carry the operator-precedence parsing. Hence the parsing process amounts to a derivation construction. Theorem 3 illustrates the use of the inference rules and their equivalence to the operator-precedence algorithm.

$$\frac{}{\$: \$ \rightarrow S}\ (\text{Base}) \qquad \frac{a >= b \qquad a\alpha : b\beta \rightarrow S}{\alpha : ab\beta \rightarrow S}\ (\text{Shift})$$

$$\frac{(T \rightarrow a) \in P \qquad a < b \qquad a\alpha : b\beta \rightarrow S}{a\alpha : \beta \rightarrow S}\ (\text{Reduce}_1)$$

$$\frac{(T \rightarrow \gamma_1\, o\, \gamma_2) \in P \qquad o < b \qquad o\alpha : b\beta \rightarrow S}{o\alpha : \beta \rightarrow S}\ (\text{Reduce}_2)$$

Fig. 4. Inference Rules for the Operator-Precedence Parser

The following definition introduces necessary concepts towards proving soundness of inference rules of Figure 4.

$$\frac{}{\epsilon : \epsilon \to \$}\ (\text{Base})$$

$$\frac{a \in \text{follow}(B) \quad B \to \epsilon \quad \epsilon : \epsilon \to \$}{a : B \to \$}\ (\text{Predict}_2)$$

$$\frac{b \in \text{first}(Ba) \quad B \to bB \quad a : B \to \$}{ba : B \to \$}\ (\text{Predict}_1)$$

$$\frac{ba : B \to \$}{bba : bB \to \$}\ (\text{Match})$$

$$\frac{b \in \text{first}(Ba) \quad B \to bB \quad bba : bB \to \$}{bba : Ba \to \$}\ (\text{Predict}_1)$$

$$\frac{bba : Ba \to \$}{abba : aBa \to \$}\ (\text{Match})$$

$$\frac{a \in \text{first}(aBa) \quad S \to aBa \quad abba : aBa \to \$}{abba : S \to \$}\ (\text{Predict}_1)$$

$$\frac{}{\$: \$ \to S}\ (\text{Base})$$

$$\frac{+ > \$ \quad E \to E + E \quad \$: \$ \to S}{\$: +\$ \to S}\ (\text{Reduce}_1)$$

$$\frac{* > \$ \quad E \to E * E \quad \$: +\$ \to S}{\$: * + \$ \to S}\ (\text{Reduce}_1)$$

$$\frac{id > \$ \quad E \to id \quad \$: * + \$ \to S}{\$: id * +\$ \to S}\ (\text{Reduce}_1)$$

$$\frac{* < id \quad \$: id * +\$ \to S}{id\$: * + \$ \to S}\ (\text{Shift})$$

$$\frac{+ < * \quad id\$: * + \$ \to S}{*id\$: +\$ \to S}\ (\text{Shift})$$

$$\frac{id > * \quad E \to id \quad *id\$: +\$ \to S}{*id\$: id + \$ \to S}\ (\text{Reduce}_1)$$

$$\frac{+ < id \quad *id\$: id + \$ \to S}{id * id\$: +\$ \to S}\ (\text{Shift})$$

$$\frac{\$ < + \quad id * id\$: +\$ \to S}{+id * id\$: \$ \to S}\ (\text{Shift})$$

$$\frac{id > + \quad E \to id \quad +id * id\$: \$ \to S}{+id * id\$: id\$ \to S}\ (\text{Reduce}_1)$$

$$\frac{\$ < id \quad +id * id\$: id\$ \to S}{id + id * id\$: \$ \to S}\ (\text{Shift})$$

Fig. 5. SLR Derivation Example for the statement id*id+id using rules of Figure 3

Definition 4. *Let (S, N, T, P) be an operator-grammar. Then **op-infer** is the set of strings $w \in T^+$ such that w has a rule derivation for $w\$: \$ \rightarrow S$ in the system of inference rules of Figure 4. The string derivation obtained by applying the grammar rules in the derivation is denoted by path(w, op). Respecting the order of the rules appearance bottom-up in the derivation is imperative for rule applications.*

Theorem 3 proves the soundness of the system of inference rules and the original operator-precedence algorithm which has the drawback of not providing correctness proofs.

Theorem 3. *Let (S, N, T, P) be an operator-grammar and $w \in T^+$. Then $w \in$ op-infer if and only if w is accepted by the operator-precedence algorithm. Moreover the right-most derivation obtained by the operator-precedence algorithm is the same as **path(w,op)**.*

Proof. The operator-precedence algorithm has the following action cases:

- The case of a *shift*: this means a symbol b is on the top of the stack and is greater than or equal to the first symbol, a, of the current input string. The order relation is obtained via the operator-precedence table. In this case the operator-precedence algorithm pushes a into the stack. Also in this case, the rule *(Shift)* of Figure 4 applies and does the same action as the operator-precedence algorithm.
- The case of a *reduce*: in this case a symbol b is on the top of the stack and is greater than the first symbol of the current input string. If b is a terminal but not an operator, then the operator-precedence algorithm discards b from the stack and records the grammar rule whose right hand side coincides b. For this case, the rule *(Reduce$_1$)* of Figure 4 applies and does the same action as the operator-precedence algorithm. If b is an operator and there is a grammar rule $T \rightarrow \gamma_1\ b\ \gamma_2$, then the operator-precedence discards b from the stack and records the grammar rule. In this case, the rule *(Reduce$_2$)* of Figure 4 applies and does the same action as the operator-precedence algorithm.
- The case of *accept*, in this case the stack and input have only \$. This case is simulated by the rule (*Base*) of Figure 4.

Parse trees obtained by inference rules of Figure 4 and that obtained by original operator-precedence algorithm are equivalent by the following two lemmas. The lemmas are proved by a straightforward structure induction on the inference-rule derivations.

Lemma 5. *Let (S, N, T, P) be an operator-precedence grammar and $w \in$ op-infer. Then path(w, op) is right-most.*

Lemma 6. *Let (S, N, T, P) be an operator-precedence grammar. Then for $w \in T^+$,*

$$S \Rightarrow^+ w \iff w\$: \$ \rightarrow S.$$

Consider the grammar:

$E \rightarrow E + E, E \rightarrow E - E, E \rightarrow E * E, E \rightarrow E/E, E \rightarrow E^E, E \rightarrow (E), E \rightarrow -E,$ and $E \rightarrow id$

# Symbols	LL(1)	LL(1)-Inf	Proof Size
15	71.0 ms	52.0 ms	3.1 KB
30	122.0 ms	98.0 ms	5.7 KB
50	143.0 ms	112.0 ms	8.9 KB

# Symbols	SLR	SLR-Inf	Proof Size
15	83.0 ms	57.0 ms	3.5 KB
30	143.0 ms	112.0 ms	6.5 KB
50	195.0 ms	134.0 ms	9.8 KB

# Symbols	Operator-Precedence	OP-Inf	Proof Size
15	62.0 ms	46.0 ms	2.0 KB
30	93.0 ms	73.0 ms	4.3 KB
50	147.0 ms	129.0 ms	7.6 KB

Fig. 6. Experiential Results

The lower derivation of Figure 2 presents a derivation for proving the correctness of the statement $id + id * id$ using inference rules of Figure 4. The following is the obtained derivation.

$$E \Longrightarrow_{rm} E + E \Longrightarrow_{rm} E + E * E \Longrightarrow_{rm} E + E * id \Longrightarrow_{rm} E + id * id \Longrightarrow_{rm} id * id + id.$$

5 Implementation and Evaluation

Timing experimenters were performed to evaluate *LL(1)-Inf, SLR-Inf*, and *OP-Inf* and to compare them to LL(1), SLR, Operator-precedence parsing algorithms, respectively. All the algorithms were implemented in $C++$. The experiments were run on a Linux system whose processor is Intel(R)-Core2(TM)-i5-CPU(2.53GHz) and whose RAM is 4GB. Computed beforehand parse tables were stored and used internally in all experiments. In order to have realistic comparisons, some modifications were done to proposed methods to simulate overhead of lexical analysis and startup penalty of running compilers. The proposed algorithms as well as the original ones were run on string of different sizes (15, 30, 50 symbols). The same grammars used in Sections 3, 4.1, 4.2 to illustrate *LL(1)-Inf, SLR-Inf*, and *OP-Inf*, respectively, were used to build the test strings.

The experimental results are shown in Figure 6. For the sake of accuracy, all readings are averaged using results of 20 runs. Parameters used to measure the performance are timings and the size of produced correctness proofs. The second parameter, the proof size, is quite an important parameter. One of the main advantages of proposed techniques in this paper over their corresponding ones is the association of each parsing result with a correctness proof in the form of inference-rule derivations. This proof is to be delivered with the parsing result. In applications like proof-carrying code and mobil computing, this proof is to be communicated. Hence it is quite important to ensure that these proofs are of convenient sizes.

As expected and confirmed by experimental results, the proposed techniques are faster than their original corresponding ones. Moreover, the sizes of the proofs are

convenient compared to the lengths of the used strings. This guaranties scalability of the proposed techniques.

6 Conclusion

This paper revealed that using systems of inference rules for achieving the parsing problem is a convenient choice for many modern applications. The relative simplicity of inference rules required to express parsing algorithms (as it is confirmed in this paper for like LL(1), LR(1), and operator-precedence parsers) attracts compiler designers to use them instead of tractional algorithmic way. The paper showed mathematical proofs for the equivalence between proposed systems of inference rules and their classical corresponding algorithms. Results of carried experiments, endorsing the efficiency of the use of inference rules, were shown in the paper. The paper also presented detailed examples of using proposed systems of inference rules.

Acknowledgment. The author acknowledges the support (grant number 330917) of the deanship of scientific research of Al Imam Mohammad Ibn Saud Islamic University (IMSIU).

References

1. 26th IEEE Canadian Conference on Electrical and Computer Engineering CCECE 2013, Regina, SK, Canada, May 5-8. IEEE (2013)
2. Al-Mulhem, M., Ather, M.: Mrg parser for visual languages. Inf. Sci. 131(1-4), 19–46 (2001)
3. Aho, R.S.A.V., Lam, M.S., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd edn. Prentice Hall, New Jersey (2007)
4. Bahrololoomi, M.H., Younessi, O., Moghadam, R.A.: Exploration of conflict situations in deterministic cancellation parser. In: CCECE [1], pp. 1–4
5. Bernardy, J.-P., Claessen, K.: Efficient divide-and-conquer parsing of practical context-free languages. In: Morrisett, G., Uustalu, T. (eds.) ICFP, pp. 111–122. ACM (2013)
6. Bodenstab, N., Hollingshead, K., Roark, B.: Unary constraints for efficient context-free parsing. In: ACL (Short Papers), pp. 676–681. The Association for Computer Linguistics (2011)
7. Brink, K., Holdermans, S., Löh, A.: Dependently typed grammars. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) MPC 2010. LNCS, vol. 6120, pp. 58–79. Springer, Heidelberg (2010)
8. Cooper, K., Torczon, L.: Engineering a Compiler, 2nd edn. Morgan Kaufmann, Waltham (2011)
9. de Piñerez Reyes, R.E.G., Frias, J.F.D.: Building a discourse parser for informal mathematical discourse in the context of a controlled natural language. In: Gelbukh, A. (ed.) CICLing 2013, Part I. LNCS, vol. 7816, pp. 533–544. Springer, Heidelberg (2013)
10. Deufemia, V., Paolino, L., de Lumley, H.: Petroglyph recognition using self-organizing maps and fuzzy visual language parsing. In: ICTAI, pp. 852–859. IEEE (2012)
11. El-Zawawy, M.A.: Detection of probabilistic dangling references in multi-core programs using proof-supported tools. In: Murgante, B., Misra, S., Carlini, M., Torre, C.M., Nguyen, H.-Q., Taniar, D., Apduhan, B.O., Gervasi, O. (eds.) ICCSA 2013, Part V. LNCS, vol. 7975, pp. 516–530. Springer, Heidelberg (2013)
12. El-Zawawy, M.A.: Distributed data and programs slicing. Life Science Journal 10(4), 1361–1369 (2013)

13. Ferrucci, F., Tortora, G., Tucci, M., Vitiello, G.: A predictive parser for visual languages specified by relation grammars. In: VL, pp. 245–252 (1994)
14. Grune, D., Jacobs, C.J.H.: Parsing Techniques: A Practical Guide, 2nd edn. Springer, Heidelberg (2007)
15. Hulden, M.: Constraint grammar parsing with left and right sequential finite transducers. In: Constant, M., Maletti, A., Savary, A. (eds.) FSMNLP, ACL Anthology, pp. 39–47. Association for Computational Linguistics (2011)
16. Jim, T., Mandelbaum, Y.: A new method for dependent parsing. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 378–397. Springer, Heidelberg (2011)
17. Jobredeaux, R., Herencia-Zapana, H., Neogi, N.A., Feron, E.: Developing proof carrying code to formally assure termination in fault tolerant distributed controls systems. In: CDC, pp. 1816–1821. IEEE (2012)
18. Kats, L.C.L., de Jonge, M., Nilsson-Nyman, E., Visser, E.: Providing rapid feedback in generated modular language environments: adding error recovery to scannerless generalized-lr parsing. In: Arora, S., Leavens, G.T. (eds.) OOPSLA, pp. 445–464. ACM (2009)
19. Koo, T., Collins, M.: Efficient third-order dependency parsers. In: Hajic, J., Carberry, S., Clark, S. (eds.) ACL, pp. 1–11. The Association for Computer Linguistics (2010)
20. Lei, T., Long, F., Barzilay, R., Rinard, M.C.: From natural language specifications to program input parsers. In: ACL (1), pp. 1294–1303. The Association for Computer Linguistics (2013)
21. Linz, P.: An Introduction to Formal Languages and Automata, 5th edn. Jones & Bartlett Learning, Burlington, MA 01803, USA (2011)
22. Middelkoop, A., Dijkstra, A., Swierstra, S.D.: Dependently typed attribute grammars. In: Hage, J., Morazán, M.T. (eds.) IFL. LNCS, vol. 6647, pp. 105–120. Springer, Heidelberg (2011)
23. Necula, G.C.: Proof-carrying code. In: Avan Tilborg, H.C., Jajodia, S. (eds.) Encyclopedia of Cryptography and Security, 2nd edn., pp. 984–986. Springer, Heidelberg (2011)
24. Nederhof, M.-J., Bertsch, E.: An innovative finite state concept for recognition and parsing of context-free languages. Natural Language Engineering 2(4), 381–382 (1996)
25. Okhotin, A.: Generalized lr parsing algorithm for boolean grammars. Int. J. Found. Comput. Sci. 17(3), 629–664 (2006)
26. Okhotin, A.: Recursive descent parsing for boolean grammars. Acta Inf. 44(3-4), 167–189 (2007)
27. Okhotin, A.: Parsing by matrix multiplication generalized to boolean grammars. Theor. Comput. Sci. 516, 101–120 (2014)
28. Scott, M.L.: Programming Language Pragmatics, 3rd edn. Morgan Kaufmann, Waltham (2009)
29. Sippu, S., Soisalon-Soininen, E.: Parsing Theory: Volume II LR(k) and LL(k) Parsing, 1st edn. Springer, Heidelberg (2013)
30. Yang, Y., Ma, M.: Sustainable mobile computing. Computing 96(2), 85–86 (2014)
31. Younessi, O., Bahrololoomi, M.H., Yazdani, A.: The extension of deterministic cancellation parser to directly handle indirect and hidden left recursion. In: CCECE [1], pp. 1–4
32. Zhou, G., Zhao, J.: Joint inference for heterogeneous dependency parsing. In: ACL (2), pp. 104–109. The Association for Computer Linguistics (2013)