Scientia Iranica D (2013) 20(6), 1939–1952

SCIENTIA IRANICA

Sharif University of Technology
Scientia Iranica
Transactions D: Computer Science & Engineering and Electrical Engineering
www.scientiairanica.com

Formal ambiguity-resolving syntax definition with asserted shift reduce sets

G. Jaberipur[a,b,*] and M. Dorrigiv[a]

a. *Department of Electrical and Computer Engineering, Shahid Beheshti University, Tehran, Iran.*
b. *School of Computer Science, the Institute for Research in Fundamental Science (IPM), Tehran, Iran.*

Received 4 April 2012; received in revised form 27 November 2012; accepted 19 February 2013

KEYWORDS
Parsing;
Shift reduce parsers;
Parser generators;
Ambiguous grammars;
Programming languages;
Compilers.

Abstract. There are parser generators that accept ambiguous context-free grammars, where ambiguities are resolved via disambiguation rules, with the outcome of smaller parse tables and more efficient parsers. However, the compiler writers are expected to develop compact ambiguous grammars and extract ambiguity-resolving information from the syntax and semantics of the language. The aforementioned tasks require considerable expertise, not often owned by casual compiler writers, or even expert programmers who are assigned a serious compiler-writing task, while programming language designers are usually capable of providing a concise and compact ambiguous description of the language that may include ambiguity-resolving information. In this paper, we aim to provide a powerful notation for syntax definition, which enables the language designer to assert some shifts and reduce sets associated with each production rule of the possibly ambiguous grammar. These sets of language tokens guide the parser generator to resolve the parse table conflicts that are caused by the ambiguities in the grammar or by other sources. The practicality of the proposed asserted shift reduce notation is supported by several examples from the constructs of contemporary programming languages, and is tested to work properly via developing a parser generator that constructs conflict-free LALR (1) parse tables.

1. Introduction

Parsers, or syntax analyzers, are used in computer science, linguistics and other disciplines as enumerated in [1]. They commonly serve as the principal subprogram in conventional compiler construction. The state of the art in compiler construction uses table-driven parsers, where all the syntactic information, which is needed for parsing, is provided in the parse table. Parsing algorithms often simulate the generation of a parse tree in a top-down (e.g., LL parsing) or bottom-up (e.g., LALR parsing) process. For example, Table 1 summarizes how both methods are used in actual compiler projects, which we have extracted from [2].

Parse tables are normally generated by programs, called parser generator, that are easily available (e.g., LLGen [3] for LL parsing and Yacc [4] for LALR). Although "Yacc is dead" has been chosen as the title of a paper [5], both LL and LR parser generators go on to serve as the principal compiler writing tools [6], where it appears that the most commonly used parsing method in the domain of programming languages is LALR (1) and its parser generator [7]. Nevertheless, since LL parsing offers some unique advantages [8,9], despite the weakness of LL grammars, to cover some particular language constructs, there are parser generators that generate LL parsers with facilities to resolve conflicts including switching to a small LR parser.

*. *Corresponding author. Tel.: +98 21 29904165; Fax: +98 21 22431804*
E-mail addresses: jaberipur@sbu.ac.ir (G. Jaberipur) and Dorrigiv@sbu.ac.ir (M. Dorrigiv)

Table 1. Parsing approach in actual compiler projects.

		Approach		
		Top-down		Bottom-up
		Recursive-descent	LL (1)	LALR (1)
Project	**Java (Sun/Oracle)**	✓		
	Java (Eclipse)			✓
	Go (Google)	✓		✓
	GCC			✓
	C++ (GCC 3.4.0)	✓		
	C/Objective-C (GCC 4.1.0)	✓		
	Python		✓	
	Ruby			✓
	PHP			✓
	Haskell	✓		✓

Such experiences, as surveyed in [6], have appeared in [6,10,11].

The LALR (1) parsing algorithm, like any other deterministic parser, requires an unambiguous context-free grammar in order to parse the input in time and space linearly-dependent on the input length, where the linear behavior of the parser is vital to the overall compiler performance. On the other hand, ambiguous grammars lead to conflicting actions in some entries of the parse table which in turn, result in an undesirable over-linear behavior of the parser. Nevertheless, they give shorter description of syntax with considerably fewer production rules and nonterminals, as compared to unambiguous grammars for the same language [12]. This generally leads to smaller parse tables and faster parsers [7,13], which has motivated the use of ambiguous grammars for deterministic parsers, although it is well-known that ambiguity and determinism cannot coexist [14]. The trick is to keep only one of the conflicting actions in the relevant entries of the parse table [13]. This is usually decided upon with the help of ambiguity-resolving information (e.g., operator precedence [15-17]) or disambiguating rules. For example, three rules for disambiguating grammars are proposed in [18]; namely reduce as soon as possible, use the production with the shortest right hand side, and use the first listed production. This technique resolves shift-reduce conflicts in favor of reducing [19], but it fails whenever a shift is desirable. The same task can be trusted to the parser generator itself to dynamically prompt the user to decide.

On the other hand apparently, based on this belief that shift-action is the right one to choose on most shift-reduce conflicts, Yacc's default decision is in favor of shift on conflicts that are not resolved by Yacc's user.

Other benefits of ambiguous grammars, besides smaller parse tables and faster parsers, include ease of comprehension and smaller and more understandable semantic rules. There are several parser generating techniques [13,18,20-23], and parser generator tools that facilitate user/parser generator interaction (e.g., Yacc [4], GNU Bison [24], CUP [25], Bertha [26], SAIDE [27], Tris [28], Elkhound [29], eyacc [30], Eli [31], Tatoo [32], LISA [33], YAJCo [34], and two others that are not specifically named [35,36]). Although some language descriptions use ambiguous grammars with ambiguity resolving description in English [37], many misunderstandings of exact definition of some language constructs have been reported on the part of compiler writers, which have resulted in incompatibilities between different implementations of the same language description [38-42]. Therefore, notwithstanding the very helpful role of the aforementioned facilities, the compiler constructor is usually faced with three uneasy tasks in developing a parser:

1. Converting the syntax description provided by the language designer to a concise ambiguous grammar;
2. Extracting the ambiguity-resolving rules from the syntax and semantic description of the language;
3. Transforming the latter to the special format required by the parser generator.

These tasks are error-prone and the compiler constructor may easily make mistakes in doing task 1 and be induced to wrong or inaccurate perceptions of the syntax and semantics of the language, while undertaking task 2, and be annoyed by the not very user-friendly input format of the conventional parser generators in task 3. Some automated tools spawned by Tomita's Generalized LR (GLR) algorithm like SDF [43], Elkhound [29] or GNU Bison [24] actually try to ease the latter tasks for the compiler writers. However, they usually provide an over-linear parser when operating in full automation mode, or otherwise need expertise help of the user (e.g., the two operation mode of GNU Bison [24]). Several programming languages have been originally implemented by the language designers (e.g., Pascal [44], Modula-2 [45], Python [46], Lua [47], Ruby [48], Java [49]). In fact, the task of designing a programming language requires skills and knowledge that are far more than required for compiler writing. This may suggest that it is more appropriate to expect language designers to take the burden of performing the preliminary static tasks listed above, indeed, on behalf of compiler designers [18]. Therefore, we are motivated to design an augmenting notation for context-free grammars, which can be used by the language designer to describe the language with a concise ambiguous grammar, and provide disambiguation rules in a formal way, which may be directly used by an automatic parser generator to resolve possible conflicting actions in the parse table (e.g., a shift-reduce conflict in LALR (1) parsing). In such an environment, the compiler constructor would not be faced with any of the above three difficult tasks nor would have any interaction with the parser generator. The rest of this paper is organized as follows: In Section 2, we review the construction of standard LALR (1) parsers, where the familiar reader might prefer to fast-forward the following section. Conventional use of ambiguous grammars is taken up in Section 3. Auxiliary functions for description of ambiguity-resolving information are introduced in Section 4; namely the proposed *last, followed-by*, and *look-afore* sets, in contrast to the conventional first, follow and look-ahead sets, respectively. Then we define and explain the computation of no-shift and reduce sets for each production rule of the grammar. Section 5 briefly describes our special parser generator written to accept the proposed formal notation, and Section 6 concludes the paper.

2. The LALR (1) parsing algorithm

The syntax of a programming language is normally described by a context-free grammar, i.e., a quadruple $G = (S, V, T, P)$, where:

- $S \in V$ is the start symbol,
- V is the set of nonterminals,
- T is the set of terminals or tokens of the programming language,
- P is the set of production rules of the form $A \rightarrow_i \alpha$, where $A \in V$, the direction of replacement in the application of production i is shown by "$\rightarrow_i$", and $\alpha \in (V \cup T)^*$ is a string of zero or more terminals and nonterminals.

Example 1. (Three equivalent grammars for simple arithmetic expressions). Figure 1 depicts sets of production rules P_1, P_2, and P_3, for three equivalent context-free grammars, G_1, G_2, and G_3, respectively. They describe simple arithmetic expressions with + and $*$, as operators, and standard parenthesizing, where id (for identifier) is an anonymous variable name, λ denotes a null string and grammar quadruples are:

$E \rightarrow_1 TE'$	$E \rightarrow_1 E + T$	$E \rightarrow_1 E + E$
$E' \rightarrow_2 + TE'$	$E \rightarrow_2 T$	$E \rightarrow_2 E * E$
$E' \rightarrow_3 \lambda$	$T \rightarrow_3 T * F$	$E \rightarrow_3 id$
$T \rightarrow_4 FT'$	$T \rightarrow_4 F$	$E \rightarrow_4 (E)$
$T' \rightarrow_5 * FT'$	$F \rightarrow_5 id$	
$T' \rightarrow_6 \lambda$	$F \rightarrow_6 (E)$	
$F \rightarrow_7 id$		
$F \rightarrow_8 (E)$		
P_1	P_2	P_3

Figure 1. Three equivalent context-free grammars for simple arithmetic expressions.

$$G_1 = (E, \{E, E', T, T', F\}, \{+, *, id, (,)\}, P_1),$$

$$G_2 = (E, \{E, T, F\}, \{+, *, id, (,)\}, P_2),$$

$$G_3 = (E, \{E\}, \{+, *, id, (,)\}, P_3).$$

A parser or a syntax analyzer is a computer program, which decomposes an input program to its syntactic constructs in order to guide the process of syntax directed translation of the input program. A comprehensive coverage of parsing techniques can be found in any compiler construction textbook (e.g., [1,7,12]). In this section, we briefly describe the dominating technique; the bottom-up LALR (1) parsers and parser generators.

2.1. LALR (1) parsing

We begin with two reminder definitions on the first and follow sets.

Definition 1 (First set). Consider all strings x, of terminals, derivable from a given string γ of terminals and nonterminals. Then First (γ) is the set of tokens that can start any of the strings x, or more formally:

$$\text{First } (\gamma) = \begin{cases} \phi & \textbf{if } \gamma = \lambda \\ a & \textbf{if } \gamma = a\beta \\ \text{First } (A) \cup & \\ (\textbf{if } A \Rightarrow^* \lambda \textbf{ then } \text{First}(\beta)) & \textbf{if } \gamma = A\beta \end{cases}$$

where $a \in T$, $\beta \in (V \cup T)^*$, $A \in V$, $\Rightarrow^*$ means zero or more steps of a derivation and:

$$\text{First } (A) = \cup_{A \rightarrow \alpha} \text{ First } (\alpha).$$

Definition 2 (Follow set). The set of all terminals that can appear after a grammar symbol (i.e., terminal or nonterminal) B in any string of terminals and nonterminal, derivable from the start symbol, is called Follow (B) or constructively:

$$\text{Follow }(B) = \cup_{A \to \alpha B \beta} \text{First }(\beta)$$

$$\cup\,(\textbf{if } \beta \Rightarrow^* \lambda \textbf{ then } \text{Follow }(A)),$$

where $\alpha,\ \beta \in (V \cup T)^*$, $A \in V$, and $B \in V \cup T$. It is postulated that $\$ \in$ Follow (S), where S is the starting symbol of the grammar, and \$ is a virtual token marking the end of input.

The LALR (1) parsing algorithm builds up a parse tree in a bottom-up manner starting from the leaves (i.e., the input tokens) up to the root (i.e., the starting symbol), which is equivalent to producing a backward rightmost derivation of the input. Figure 2 depicts a rightmost derivation of $id + id * id$ under G_2 of Figure 1, where the newly generated nonterminals are underlined.

The LALR (1) parser is derived by a parse table. To generate the latter, the LALR (1) parser generator produces a state diagram representing the states of a push down automata [14]. It then derives a parse table, where each row represents one of the states of the state diagram, and each column represents one of the grammar symbols (i.e., terminals and nonterminals). For example, the LALR (1) state diagram for grammar G_2 is shown in Figure 3, where each state consists of one or more LALR (1) items (to be defined below) and Table 2 is the corresponding parse table.

$$\underline{E} \Rightarrow_1 E + \underline{T} \Rightarrow_3 E + T * \underline{F} \Rightarrow_5 E + \underline{T} * id \Rightarrow_4$$
$$E + \underline{F} * id \Rightarrow_5 \underline{E} + id * id \Rightarrow_2 \underline{T} + id\ * id \Rightarrow_4$$
$$\underline{F} + id * id \Rightarrow_5 id + id * id$$

Figure 2. Rightmost derivation of $id + id * id$ under grammar G_2.

Definition 3 (LALR (1) item). An LALR (1) item "$A \to \alpha \bullet \beta, \mathcal{L}$" in some state of a LALR (1) state diagram represents a parsing status, where that part of the input tokens, which are derivable from the string α, has been so far read and the parser is expected to reduce a string of next input tokens to β. The look-

Table 2. LALR (1) parse table for G_2.

	$+$	$*$	id	(	)	\$	E	T	F
1			$\mathcal{S}_5$	$\mathcal{S}_6$			$\mathcal{G}_2$	$\mathcal{G}_3$	$\mathcal{G}_4$
2	$\mathcal{S}_7$					$\mathcal{A}$			
3	$\mathcal{R}_2$	$\mathcal{S}_8$			$\mathcal{R}_2$	$\mathcal{R}_2$			
4	$\mathcal{R}_4$	$\mathcal{R}_4$			$\mathcal{R}_4$	$\mathcal{R}_4$			
5	$\mathcal{R}_5$	$\mathcal{R}_5$			$\mathcal{R}_5$	$\mathcal{R}_5$			
6			$\mathcal{S}_5$	$\mathcal{S}_6$			$\mathcal{G}_9$	$\mathcal{G}_3$	$\mathcal{G}_4$
7			$\mathcal{S}_5$	$\mathcal{S}_6$				$\mathcal{G}_{10}$	$\mathcal{G}_4$
8			$\mathcal{S}_5$	$\mathcal{S}_6$					$\mathcal{G}_{11}$
9	$\mathcal{S}_7$				$\mathcal{S}_{12}$				
10	$\mathcal{R}_1$	$\mathcal{S}_8$			$\mathcal{R}_1$	$\mathcal{R}_1$			
11	$\mathcal{R}_3$	$\mathcal{R}_3$			$\mathcal{R}_3$	$\mathcal{R}_3$			
12	$\mathcal{R}_6$	$\mathcal{R}_6$			$\mathcal{R}_6$	$\mathcal{R}_6$			

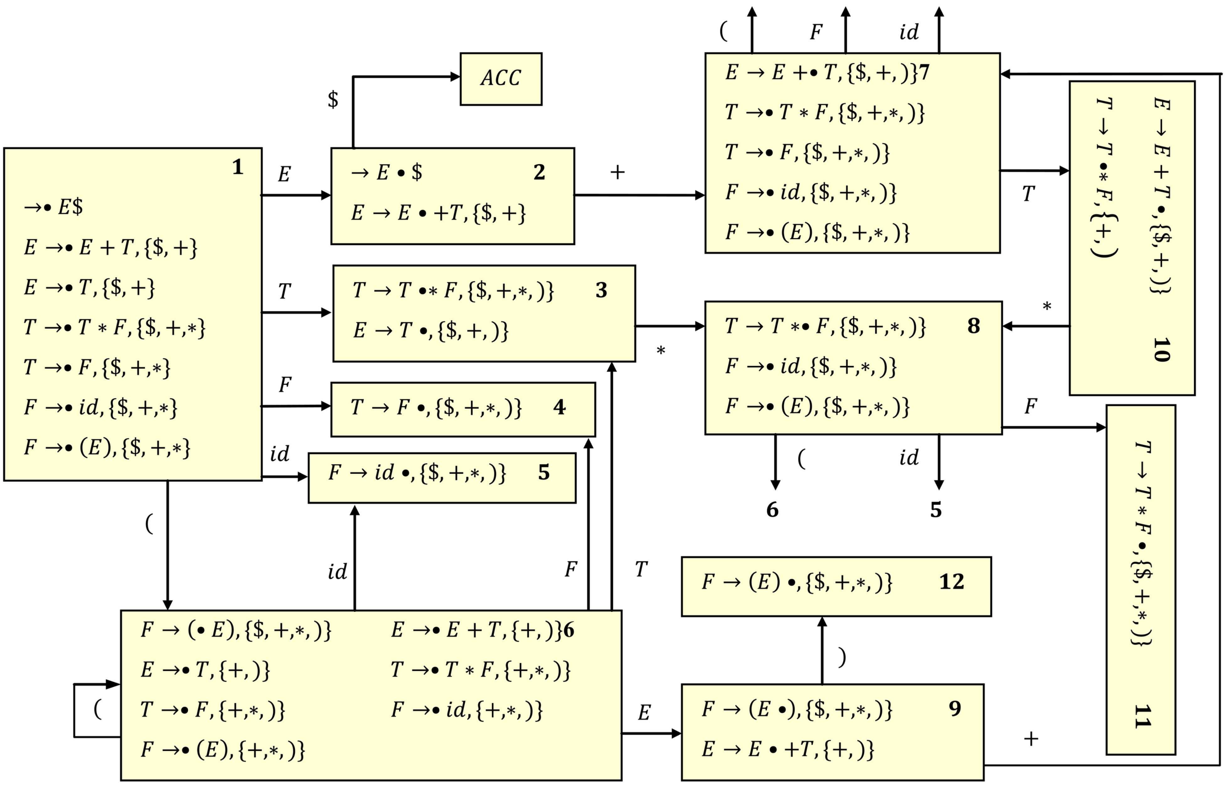

Figure 3. LALR (1) state diagram for G_2.

ahead set $\mathcal{L}$ (to be formally defined in Section 4) is extraneous unless $\beta = \lambda$, in the case in which the reduction by production rule $A \rightarrow \alpha\beta$ is valid only on input tokens that belong to $\mathcal{L}$.

The parser starts in State 1, and a correct input leads it to a special accepting state (ACC in Figure 3). Two parsing actions, called shift and reduce are possible in each state, which are described as follows:

- *Shift*: This action occurs if there is an exiting arc labeled by the current token. The action is composed of the following subtasks:
  - Advancing the input by one terminal token, normally by a call to scanner.
  - Following the aforementioned arc to the next state.
- *Reduce*: This action takes place if a dotted production rule ends with a $\bullet$ (marking that the entire right context has been read from the input), and the current token exists in the corresponding look-ahead set. The action consist of the following subtasks:
  - Invoking a semantic action hooked to the reduce production (e.g., generation of code for addition in State 10).
  - Returning to the state where the same LALR (1) item exist, but with the dot located before the leftmost symbol of the right hand side (e.g., returning from State 10 to State 1 or 6).
  - Following the arc labeled with the nonterminal in the left hand side of the reduce production (e.g., following the arc labeled E from State 1 or 6 to State 2 or 9, respectively).

To facilitate the parser actions, the parser generator produces a parse table, where each cell contains the appropriate parsing action.

Table 2 depicts the LALR (1) parse table derived from the state diagram of Figure 3, where $\mathcal{R}_i$ and $\mathcal{S}_j$ per the definitions above mean reduction by production i and shift to State j, respectively. However, an intermediate action $\mathcal{G}_k$ is used to guide the third step of a reduce action in forwarding the parser to State k. For example, the third step of $\mathcal{R}_1$ in State 10 can be $\mathcal{G}_2$ (see the third step of reduce action above). Empty entries indicate a parsing error, and $\mathcal{A}$ signals a successful parse.

3. Using ambiguous grammars

Ambiguous grammars are not theoretically desirable because they lead to nondeterministic parsers. However, some programming constructs, when described by an ambiguous grammar, consume fewer nonterminals and productions and often ambiguous grammars are more concise and readable [13]. The parse table generated by an LALR (1) parser generator includes conflicting actions in some entries when the input grammar is ambiguous.

$$E \rightarrow E + E\, \bullet, \{\$, +, *,)\}$$
$$E \rightarrow E \bullet * E, \{\$, +, *,)\}$$
$$E \rightarrow E \bullet + E, \{\$, +, *,)\}$$

Figure 4. The content of a conflicting LALR (1) state for grammar G_3.

Example 2 (Ambiguous grammar for simple expressions). G_3 in Figure 1 is an ambiguous grammar for simple expressions. One conflicting state of the LALR (1) state diagram for G_3 is shown in Figure 4. There are two shift-reduce conflicts, that is the parser may perform a reduction with production 1 on inputs $+$ and $*$, or a shift on the same inputs.

3.1. *Resolving the ambiguity conflict*

The classical approach for resolving conflicts in some cells of an LALR (1) parse table is that either the user [13,50] or the parser generator [4], using extra syntactic or semantic information about the language construct that is not embedded in the ambiguous grammar, edit the contents of conflicting cells. For example, the reduce (shift) action on input $*$ ($+$) may be permanently removed from the conflicting cells related to the conflicting LALR (1) state of Figure 4, due to precedence of $*$ over $+$ (left associativity of $+$).

3.2. *Advantage of parsing ambiguous grammars*

Ambiguous grammars are normally shorter, which leads to smaller parse tables. But the main advantage is the parser speed-up that may be gained, as in Example 3, where two derivations for the same expression shows that number of derivation steps is 24% less in case of ambiguous grammar.

Example 3 (Parsing speed-up by using ambiguous grammars). Recalling grammar G_2 and its ambiguous equivalent G_3, rightmost derivations of simple expression $a \times a + t \times a \times b + b \times b$ are given in Figure 5, where 17 and 13 derivation steps are required, respectively. In case of an arbitrary expression with p $+$ operators, m $\times$ operators, and $p+m+1$ identifiers (e.g., $p = 2$, $m = 4$, and there are 7 id occurrences in the expression of Figure 3), there would be $p + m + 1$ derivation steps with production 5, p with 1, m with 3, $p + 1$ with 4, and 1 with 2 (i.e., $3p + 2m + 3$ steps ensemble), while similar elaboration for G_3 leads to $2p+2m+1$ derivation steps (i.e., more than 25% parser speed-up for $m + p > 7$).

Case 1, in Section 4, shows a grammar for arithmetic, Boolean and relational expressions with only one nonterminal. This super-ambiguous grammar could

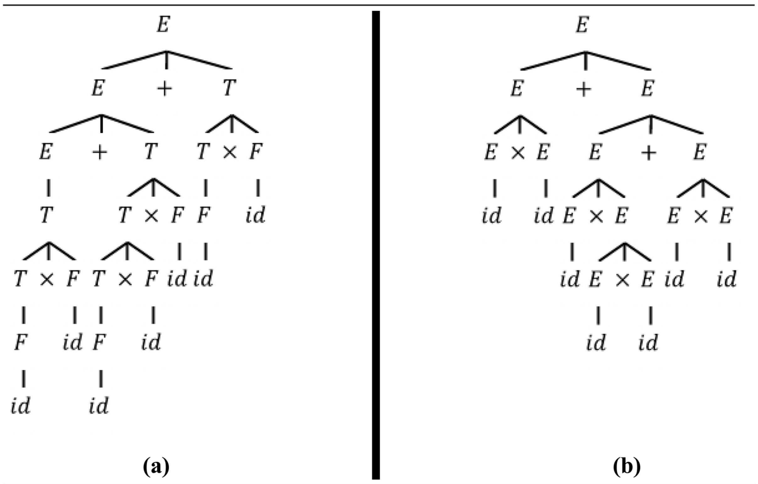

Figure 5. Derivation of $a \times a + t \times a \times b + b \times b$ under G_2 (a), and G_3 (b).

also lead to an LALR (1) parse table with resolved conflicting cells due to precedence and error rules.

The above conventional ambiguity-resolving approach, whether done by the user or the parser generators (e.g., Yacc [4]), requires specialized knowledge on the part of the user of the parser generator. The OCFG notation [26, 51], in contrast to Yacc, associates at most one disambiguation rule to a production rule of the ambiguous grammar, such that the associativity or precedence of a given operator may differ in different contexts corresponding to different production rules. This approach can be further empowered to associate the disambiguation information to each token in each production. This is actually what we carry on in the next section, where we design a notation for augmenting each production rule of the grammar with ambiguity-resolving or other restricting information, which is meant to force the language designer to explicitly provide such information. Then we describe how our special parser generator uses the augmented ambiguity-resolving information to adjust the conflicting cells of the parse table.

4. The new disambiguation formal notation

To introduce our new disambiguation formalism, we need to formally define some new (e.g., look-afore) and existing (e.g., look-ahead) auxiliary sets of grammar symbols. The look-ahead sets, in LALR (1) parsing, are computed as a subset of follow sets to determine the look-ahead tokens, for which a reduce action is justified.

Definition 4 (Look-ahead set). The look-ahead set $\mathcal{L}$ of an LALR (1) item $A \rightarrow \bullet\gamma$, $\mathcal{L}$, in a State i, is defined as follows, where $j \rightarrow^{\alpha} i$ indicates that there exist a path from State j to i via $\alpha \in (V \cup T)^*$.

$$LA_i(A \rightarrow \bullet\gamma) = \cup_{B \rightarrow \alpha\bullet A\beta \in i}(\text{First }(\beta) \cup$$

$$(\textbf{if } \beta \Rightarrow^* \lambda \textbf{ then } \cup_j (LA_j(B \rightarrow \bullet\alpha A\beta | j \rightarrow^{\alpha} i))).$$

Example 4 (Look-ahead set). Consider $E \rightarrow E + T\bullet, \{\$, +,)\}$ in State 10 of Figure 3. The look-ahead set is the union of $\{\$, +\}$ and $\{+,)\}$, from $E \rightarrow \bullet E + T, \{\$, +\}$ in State 1, and $E \rightarrow \bullet E + T, \{+,)\}$ in State 6, respectively.

4.1. Asserted reduce set

The conventional ambiguity-resolving approach of Section 3, would lead to deleting some look-ahead tokens from the look-ahead set of a reduce configuration in LALR (1) parsing. To select a token to be deleted from the look-ahead set, one uses the syntactic or semantic characteristics of the language. The language designer could take this decision on deletion of some token (s) from the look-ahead set. He or she, on describing the language by a context-free grammar, could use an ambiguous syntax, but in order to resolve the ambiguity, predict the tokens to be deleted from the look-ahead set of the LALR (1) item. Therefore, each production may be augmented by the set of remaining look-ahead tokens, each of which on appearing as the next input token validates a reduce action; hence the name asserted reduce set.

Example 5 (an asserted reduce-set grammar). Figure 6 depicts an asserted reduce-set version of

$$E \rightarrow E + E, \{\$, +,)\}$$

$$E \rightarrow E * E, \{\$, +, *,)\}$$

$$E \rightarrow (E), \{\$, +, *,)\}$$

$$E \rightarrow id, \{\$, +, *,)\}$$

Figure 6. An asserted reduce-set grammar for simple expressions.

grammar G_3. A reduction by production $E \rightarrow E{+}E$ on look-ahead $*$, is not allowed due to the precedence of $*$ over $+$. Therefore, $*$ is not included in the asserted reduce set of that production, while an LALR (1) parser generator would signal a reduction by $E \rightarrow E + E$, on look-ahead $*$.

Provision of the asserted reduce set for each production helps in automatic resolution of an LALR (1) reduce-reduce conflict, or a shift-reduce conflict, when it is to be resolved by removing the reduce action. This is the case in Figure 6 for LALR (1) item $E \rightarrow E+ E, \{\$, +,)\}$, which can lead a parser generator to delete $*$ from the set of valid look-ahead tokens for reduction. Nonetheless, the asserted reduce-set cannot help when the conflict should be resolved by deleting a shift action from an LALR (1) parse table. For example, consider the LALR (1) State 10 of Figure 7. The first item justifies a reduction on $+$, and the second item leads to a shift on $+$. The left associativity rule for $+$ operator requires the shift action to be deleted in this case. This decision is to be supported by some knowledge of the previously read input (e.g., whether a $+$ has been read just before the input string reduced to E). Therefore, we need a mechanism to inform the parser generator of the possible left context. We propose, below, the notion of look-afore and no-shift sets.

4.2. Look-afore sets

The conventional look-ahead set for any LALR (1) item, based on a production $A \rightarrow \alpha$, is always a subset of Follow (A). In contrast to the follow sets, we can define the followed-by sets with the help of an auxiliary function Last.

Definition 5 (Last and Followed-by sets). Last of a string, composed of terminals and nonterminals, is the set of rightmost terminal tokens derivable from the given string. Followed-by set of a terminal or nonterminal is the set of terminal tokens that can precede it in any string derivable from the start symbol. More formally:

$$\text{Followed-by } (B) = \cup_{A\rightarrow\alpha B\beta} \text{ Last } (\alpha) \cup$$

$$(\textbf{if } \alpha \Rightarrow^* \lambda \textbf{ then } \text{Followed-by}(A)), \text{ where :}$$

Last is defined as follows, where, $a \in T$, $\alpha, \beta \in (V \cup T)^*$, $A \in V$, and $B \in (V \cup T)$.

$$\text{Last } (\alpha) = \begin{cases} \phi & \textbf{if } \alpha = \lambda \\ a & \textbf{if } \alpha = \beta a \\ \cup_{A\rightarrow\gamma}(\text{Last } (\gamma)) & \textbf{if } \alpha = A \\ \text{Last } (A) \cup & \\ (\textbf{if } A \Rightarrow^* \lambda \textbf{ then } \text{Last } (\beta)) & \textbf{if } \alpha = \beta A \end{cases}$$

We postulate that $\$ \in$ Followed-by (S), where S is the starting symbol of the grammar.

Table 3. Follow and followed-by sets.

Symbol	Follow set	Followed-by set
E	$\$, +,)$	$\$, ($
T	$\$, +,), *$	$\$, +, ($
F, id	$\$, +,), *$	$\$, +, (, *$
$+, *$	$id, ($	$id,)$
$)$	$\$, +,), *$	$id,)$
$($	$id, ($	$\$, +, (, *$

Example 6 (Follow and followed-by sets in G_2). Table 3 shows the follow and followed-by sets for the symbols of grammar G_2 of Figure 1.

Definition 6 (Look-afore set). The look-afore set of an LALR (1) item $A \rightarrow \bullet\gamma$, $\mathcal{L}$, in a State i, is defined as follows, where $j \rightarrow^{\alpha} i$ indicates that there exist a path from State j to i via $\alpha \in (V \cup T)^*$.

$$LF_i(A \rightarrow \bullet\gamma) = \cup_{B\rightarrow\alpha\bullet A\beta\in i}(\text{Last } (\alpha) \cup$$

$$(\textbf{if } \alpha \Rightarrow^* \lambda \textbf{ then } \cup_j \, (LF_j(B \rightarrow \bullet\alpha A\beta | j \rightarrow^{\alpha} i))).$$

Note that the look-afore set is a subset of the corresponding followed-by set.

Example 7 (Look-afore set). The LALR (1) diagram, augmented with look-afore (LF) sets for the left hand side nonterminal of each LALR (1) item is shown in Figure 7. If the string preceding a right hand side symbol of an item is nullable (e.g., α in Definition 6), the given LF set can be used in computing the LF set of the right hand side symbol.

In LALR (1) parsing, an undesirable shift operation might be suggested, on input a, by an LALR (1) item $A \rightarrow \alpha \bullet a\beta$, $\mathcal{L}$. To help the parser generator to ignore such shifts, we introduce and use the no-shift sets through the following example.

Definition 7 (No-shift set). For each production rule $A \rightarrow_j \alpha a\beta$, and a token a in its right hand side, a no-shift set $ns_j^a \subseteq T$ is defined, such that no shift, on token $a \in ns_j^a$, from State i, holding an LALR (1) item $A \rightarrow_j \alpha \bullet a\beta$, is allowed iff $LF_i(A \rightarrow_j \alpha \bullet a\beta) \cap ns_j^a \neq \phi$.

Example 8 (Use of no-shift sets). Consider States 1 and 4 in Figure 7. The last input token, before the parser enters State 4 is $-$ and this state suggests a shift on another $-$ (see the last item in State 4). Suppose the occurrence of two consecutive $-$ tokens, in the input, is to be considered wrong. Therefore, a shift from this state on $-$ is not desirable. But an input expression in State 1 starts with $-$, which correctly qualifies a shift on it. Note that the $LF_4(E \rightarrow \bullet - E)$, for the last item, includes $-$, but $LF_1(E \rightarrow \bullet - E)$ does

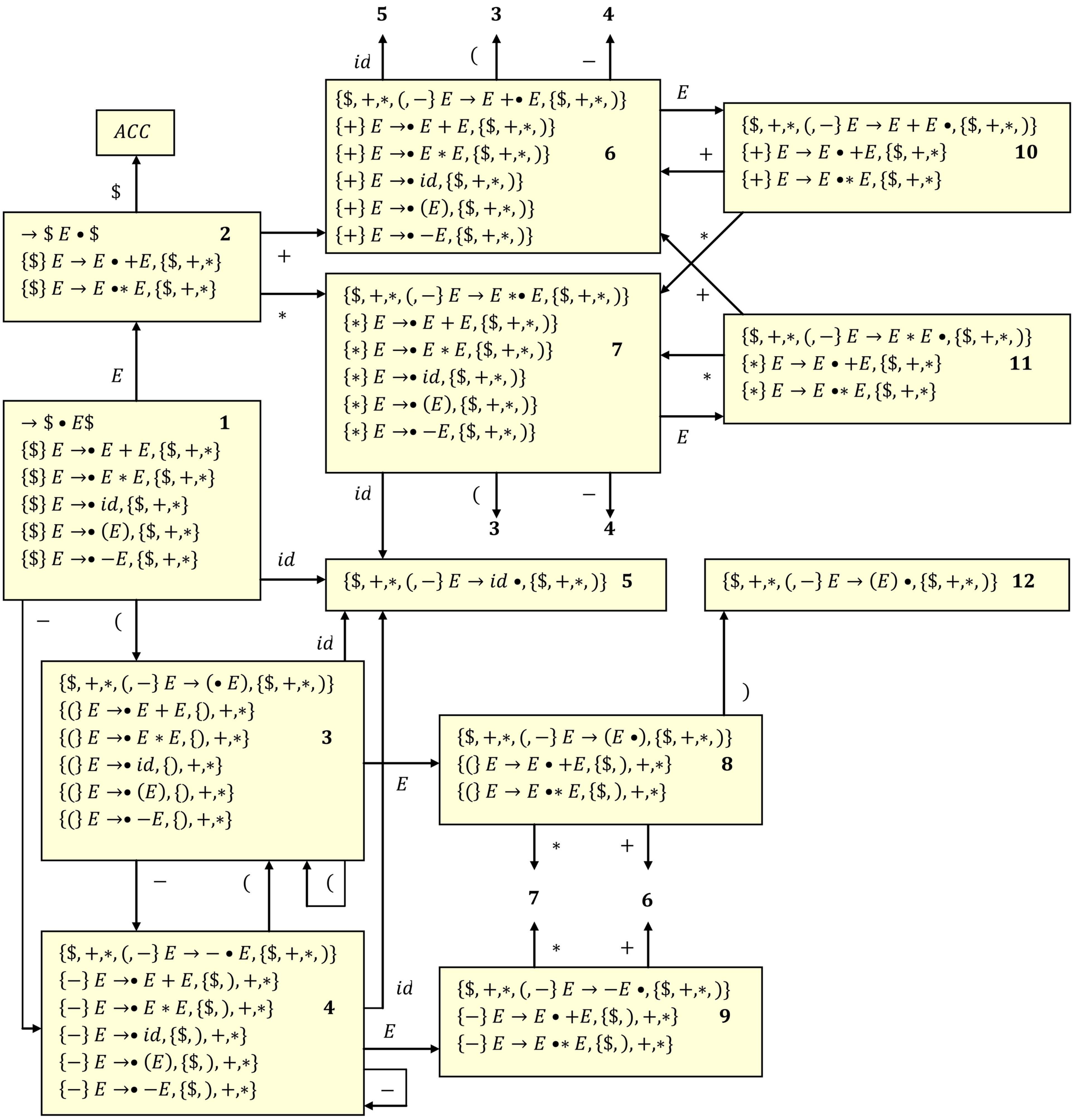

Figure 7. LALR (1) parsing diagram for simple expressions with unary minus.

not. As another example, assume that other arithmetic operators besides $-$ (e.g., $+$), are not allowed to appear just before $-$. Then such a faulty input may lead the parser to State 6 of Figure 7, where $LF_6(E \to \bullet - E)$ is $\{+\}$. To guide the parser generator to delete a wrong shift on $-$, one may provide the parser generator with a set of no-shift tokens. The no-shift token set, in this case, is shown within the last production rule of the augmented grammar G_4 in Figure 8.

$$E \to_1 E\{-,+,*\} + E, \{\$,+,)\}$$
$$E \to_2 E\{-,*\} * E, \{\$,+,*,)\}$$
$$E \to_3 (E), \{\$,+,*,)\}$$
$$E \to_4 id, \{\$,+,*,)\}$$
$$E \to_5 \{-,+,*\} - E, \{\$,+,*,)\}$$

Figure 8. Grammar G_4 for simple expressions with no-shift and asserted look-ahead sets.

It indicates that if $LF_i(E \to \bullet - E)$, for the LALR (1) item $E \to \bullet - E$, $\mathcal{L}$ in any State i, has a nonempty intersection with the no-shift set then a shift on $-$ is not allowed from State i. Therefore, the parser generator will delete shifts on $-$ in States 4, 6 and 7, because $LF_4(E \to \bullet - E) = \{-\}$, $LF_6(E \to \bullet - E) = \{+\}$, and $LF_7(E \to \bullet - E) = \{*\}$ have a common token with the no-shift set $\{-,+,*\}$, respectively. For an example of controlling left associativity, with the help of no-shift and look-afore sets, consider State 10 of Figure 7 and G_4 of Figure 8. The relevant item is $\{+\}E \rightarrow E \bullet +E, \{\$, +, *,)\}$. $LF_{10}(E \rightarrow E \bullet +E)$ in the given LALR (1) item is $\{+\}$, which indicates that a shift on $+$ violates the left associativity. Therefore, one way to enforce a no-shift on $+$ is to include a no-shift set for $+$ in the production rule $E \rightarrow E+E$ as in G_4 of Figure 8. Then the nonempty intersection of the no-shift set and the LF set of the rule in the grammar lead the parser generator not to put the shift on $+$ from the relevant entry of the parse table.

The latter observation on LF and no-shift sets leads to a general method for providing the parser generator with information it needs to ignore the undesirable shifts. The language designer may associate a set of no-shift tokens with a terminal token appearing in the right hand side of a production. The no-shift set is inserted just before the associated terminal symbol.

The parser generator ignores a shift on an input a, due to an item $A \rightarrow_i \alpha \bullet a\beta A$ if the last no-shift set before $\bullet a$, and the look-afore set of the symbol succeeding the no-shift set, have at least one common token.

4.3. The ASR LALR (1) grammars

We present a formal definition of asserted shift reduce grammars with some abbreviation facilities for augmenting the productions with ambiguity-resolving information. Each production is augmented by zero or more no-shift sets, and one asserted reduce set.

4.3.1. The Asserted Shift Reduce (ASR) grammars

Definition 8 (Asserted Shift Reduce (ASR) Grammars). An asserted shift reduce grammar is a quadruple $G = (S, V, T, P)$, where

- $S \in V$ is the starting symbol,
- V is a finite set of nonterminal symbols,
- T is a finite set of the terminal symbols (tokens in the language of the grammar),
- $P = \{A \rightarrow (V^*[ns]T)^*V^*[[-]rs]$.

In definition for $P, A \in V$, ns and rs are subsets of $T \cup \{\$\}$, where \$ is a special augmenting symbol neither in V nor in T. The ns sets guide the parser generator to restrict shift actions, and the rs set indicates the look-ahead tokens for a valid reduce action. Lack of the reduce set at the end of a production rule means that reduction by that rule is restricted to the tokens of the standard look-ahead set for that rule.

Similarly, shifts are allowed on a token in the right hand side of a production rule that is not preceded by a no-shift set. In other words, a null no-shift set need not appear before the corresponding token. A reduce set that is not preceded by a sign (i.e., $-$) means that the rs set contains exactly the tokens on which reduce action is allowed. A missing rs (not an empty rs as $\{\}$), means that exact look-ahead set is the same as

$$
\begin{aligned}
&ST \rightarrow_1 if\ BE\ then\ ST\ EP \\
&EP \rightarrow_2 else\ ST \\
&EP \rightarrow_3 \lambda, -\{else\}
\end{aligned}
$$

Figure 9. The ASR grammar, G_5, for the if-then-else construct.

$$
\begin{aligned}
&E \rightarrow_1 E\{-,+,*\} + E, -\{*\} \\
&E \rightarrow_2 E\{-,*\} * E \\
&E \rightarrow_3 (E) \\
&E \rightarrow_4 id \\
&E \rightarrow_5 \{-,+,*\} - E
\end{aligned}
$$

Figure 10. Grammar G_6 with asserted no-shift sets and abbreviated reduce sets.

standard LALR (1) look-ahead set. $-rs$ means that actual reduce set is the difference of standard look-ahead set and the given rs.

Example 9 (Dangling else problem). We can easily handle the well-known dangling else problem via restricting the reduce set of the relevant production rule. The look-ahead set of production rule 3 in the ASR grammar G_5 (Figure 9) does normally include else, which is the source of shift-reduce conflict. The $-\{else\}$ expression that augments production rule 3 signals the parser generator to remove *else* from the reduce set, which resolves the conflict.

Example 10 (ASR grammar). Grammar G_6 in Figure 10 is a reproduction of G_4, with the above abbreviating rules.

The standard LALR (1) look-ahead set for production 1 (Figure 10), where the $\bullet$ symbol has reached the rightmost position, is $\{\$, +, *,)\}$ as in the LALR (1) configuration $E \rightarrow_1 E+E\bullet, \{\$, +, *,)\}$ of Figure 4. Note that rs_1 misses a $*$, which means that a reduction by production 1 is not valid on look-ahead $*$. The latter restriction guarantees the standard precedence of $*$ over $+$. To see the applicability of ns_1^+, consider the ASR LALR (1) States 6 and 10 of Figure 7. The look-afore set for the leftmost E in the right hand side of configuration $E \rightarrow_1 \mathbf{E}\{-, +, *\} + E, -\{*\}$ (i.e., Bold E) is the same as the look-afore set for the left hand side E, which is $\{+\}$. Since the first no-shift set before $+$, and the look-afore set of the first symbol after the no-shift set, both include $+$, the parser generator would not allow a shift on $+$. The latter restriction guarantees the left associativity of $+$. Note that the no-shift set associated to $*$ in production 2 asserts the shift on $*$. A similar (different) situation for production 1 (2) arises in State 11, where the token $*$ is shared by the corresponding look-afore set and the no-shift set of $+$ $(*)$ in production 1 (2), which signals the parser generator to delete shift on $+$ $(*)$.

To better appreciate the power of ASR grammars in providing ambiguity-resolving information, we con-

(a)		(b)	
$E \rightarrow_1 E + E$	$E \rightarrow_9 E \wedge E$	$E \rightarrow_1 E\{-,+,*,/,\vee,\wedge,!\} + E, -\{*,/,\vee,\wedge\}$	$E \rightarrow_9 E\{-,+,*,/,\wedge,!\} \wedge E, -\{\$,),\vee,\wedge\}$
$E \rightarrow_2 E - E$	$E \rightarrow_{10} !E$	$E \rightarrow_2 E\{-,+,*,/,\vee,\wedge,!\} - E, -\{*,/,\vee,\wedge\}$	$E \rightarrow_{10} \{-,+,*,/,!\}! E, -\{\$,),\vee,\wedge\}$
$E \rightarrow_3 E * E$	$E \rightarrow_{11} E = E$	$E \rightarrow_3 E\{*,/,\vee,\wedge,!\} * E, -\{\vee,\wedge\}$	$E \rightarrow_{11} E\{-,+,*,/,=\} = E, -\{\$,),\vee,\wedge\}$
$E \rightarrow_4 E/E$	$E \rightarrow_{12} E \neq E$	$E \rightarrow_4 E\{*,/,\vee,\wedge,!\}/E, -\{\vee,\wedge\}$	$E \rightarrow_{12} E\{-,+,*,/,\neq\} \neq E, -\{\$,),\vee,\wedge\}$
$E \rightarrow_5 - E$	$E \rightarrow_{13} E < E$	$E \rightarrow_5 \{-,+,*,/,\vee,\wedge,!\} - E, -\{\vee,\wedge\}$	$E \rightarrow_{13} E\{-,+,*,/,<\} < E, -\{\$,),\vee,\wedge\}$
$E \rightarrow_6 (E)$	$E \rightarrow_{14} E > E$	$E \rightarrow_6 (E)$	$E \rightarrow_{14} E\{-,+,*,/,>\} > E, -\{\$,),\vee,\wedge\}$
$E \rightarrow_7 id$	$E \rightarrow_{15} E \nleq E$	$E \rightarrow_7 id$	$E \rightarrow_{15} E\{-,+,*,/,\nless\} \nleq E, -\{\$,),\vee,\wedge\}$
$E \rightarrow_8 E \vee E$	$E \rightarrow_{16} E \ngtr E$	$E \rightarrow_8 E\{-,+,*,/,\vee,\wedge,!\} \vee E, -\{\$,),\vee\}$	$E \rightarrow_{16} E\{-,+,*,/,\ngtr\} \ngtr E, -\{\$,),\vee,\wedge\}$

Figure 11. (a) Grammar G_7 with arithmetic, Boolean, and relational operators. (b) The ASR grammar, G_8, for G_7.

sider the following cases and offer simple ASR grammars that fully resolve ambiguities statically, where other similar tools prompt the parser generator user to resolve some ambiguities.

Case 1 (Ambiguous grammar for arithmetic and Boolean expressions). Grammar G_7, in Figure 11(a), describes mixed arithmetic and Boolean expressions as is allowed in the C programming language. This is a super ambiguous grammar (with only one nonterminal) which leads to 143 conflicts in the corresponding LALR (1) parse table. However, all the conflicts are resolvable by the semantics of the language. We provide an ASR grammar G_8 in Figure 11(b) with asserted no-shift and reduce sets for arithmetic, Boolean, and relational expression. This ASR grammar, contrary to G_7, follows the syntax of Pascal language with conventional operator precedence and does not accept unnecessary operators such as a not operator preceding by another one. It is assumed that the operands of relational and arithmetic operators are only arithmetic expressions, and operands of Boolean operators are Boolean or relational expressions. The asserted no-shift and reduce sets are chosen such that any violation of the aforementioned assumptions will be detected as soon as they occur in the input.

Case 2 (General precedence). The authors of [13] have studied the problem of resolving the ambiguities that occur when describing expressions, with n different operators, via a single nonterminal grammar. The general case of this problem is described by grammar G_9 of Figure 12 borrowed from [13].

$$
\begin{array}{l}
E \rightarrow_1 E *_1 E \\
E \rightarrow_2 E *_2 E \\
\vdots \\
E \rightarrow_n E *_n E \\
E \rightarrow_{n+1} (E) \\
E \rightarrow_{n+2} a
\end{array}
$$

Figure 12. Generalization of single nonterminal grammar, G_9, for expressions.

$$
\begin{array}{l}
E \rightarrow_1 E\{*_1, \dots, *_n\} *_1 E, -\{*_2, \dots, *_n\} \\
E \rightarrow_2 E\{*_2, \dots, *_n\} *_2 E, -\{*_3, \dots, *_n\} \\
\vdots \\
E \rightarrow_n E\{*_n\} *_n E \\
E \rightarrow_{n+1} (E) \\
E \rightarrow_{n+2} a
\end{array}
$$

Figure 13. The ASR grammar, G_{10}, for G_9.

The dynamic ambiguity-resolving rule that is used in [13] works as follows: Assume that there are n left associative operators $*_1, *_2, \cdots, *_n$ with ascending precedence from $*_1$ to $*_n$. The rule states: "**if** $i > j$, shift; otherwise, reduce." The static ASR solution, however, does the same by the ASR grammar G_{10} of Figure 13.

Case 3 (Super- and subscripted expressions). The authors of [52] have developed a typesetting language for mathematics, where it is desirable to have superscripts and subscripts aligned in expressions such as x_i^2. This is described by the ambiguous grammar G_{11} as in Figure 14. For example, the expression x *sub* i *sup* 2 can be interpreted, by G_{11}, in three ways as ((x *sub* i) *sup* 2) meaning x_i^2, (x *sub* (i *sup* 2)) to mean x_{i^2}, or (x *sub* i *sup* 2) as the desired x_i^2, which can be achieved also by the ASR grammar of Figure 15.

Case 4 (Case Expressions in Standard ML). Figure 16 depicts the ambiguous grammar G_{12},

$$
\begin{array}{l}
E \rightarrow E\ sub\ E \\
E \rightarrow E\ sup\ E \\
E \rightarrow E\ sub\ E\ sup\ E \\
E \rightarrow id
\end{array}
$$

Figure 14. Grammar G_{11}, part of a typesetting language for mathematics.

$$
\begin{array}{l}
E \rightarrow E\{sub\}\ sub\ E, -\{sup\} \\
E \rightarrow E\ sup\ E \\
E \rightarrow E\{sub\} sub\ E\ sup\ E \\
E \rightarrow id
\end{array}
$$

Figure 15. The ASR grammar for G_{11}.

$E \to_1 case\ E\ of\ MATCH$
$E \to_2 id$
$MATCH \to_3 MRULE$
$MATCH \to_4 MATCH \mid MRULE$
$MRULE \to_5 PAT \Rightarrow E$
$PAT \to_6 id\ ATPAT$
$ATPAT \to_7 id$

Figure 16. An ML ambiguous grammar for case expressions.

$E \to_1 case\ E\ of\ MATCH, -\{|\}$
$E \to_2 id$
$MATCH \to_3 MRULE$
$MATCH \to_4 MATCH \mid MRULE$
$MRULE \to_5 PAT \Rightarrow E$
$PAT \to_6 id\ ATPAT$
$ATPAT \to_7 id$

Figure 17. The ASR grammar, G_{13}, for G_{12}.

adapted from [35], that describes the "case" expressions of ML [53]. For example, there are two parse trees for the ML case expression, $case\ a\ of\ b \Rightarrow case\ b\ of\ c \Rightarrow c|d \Rightarrow d$, based on G_{12}. The standard ML definition requires that the matching rule "$d \Rightarrow d$" should be attached to "*case b*". The static ASR solution for the same problem is described in Figure 17.

5. The ASR LALR (1) parser generator

The input to the ASR LALR (1) parser generator is an ASR grammar. The ASR parser generator is able to produce a non-conflicting LALR (1) parse table, if the following hold:

a) The possible conflicts of an LALR (1) parse table produced by a standard LALR (1) parser generator, could be resolved by editing the parse table according to ambiguity-resolving characteristic of the language not described by the standard context-free grammar (without the asserted sets).
b) Asserted no-shift sets and the asserted reduce sets resolve all the possible ambiguities of the grammar.

When the above conditions hold, the ASR LALR (1) parser generator is basically the same as any LALR (1) parser generator, except that decisions to fill the parse table entries are affected by the asserted sets, and the parser generator needs to compute the look-afore sets of the grammar symbols located immediately to the right of a no-shift set. The shift and reduce entries are found as follows:

- Shift entries: For any configuration, in State i, of the form $A \to_j \alpha ns \bullet a\beta rs$ insert a shift action in row i and column a, iff $ns \cap LF_i(a) \neq \phi$.
- Reduce entries: For any configuration, in State i, of the form $A \to_j \alpha \bullet rs$, insert a reduce action in row i and all columns a, iff $a \in rs$. For any configuration, in State i, of the form $A \to_j \alpha \bullet -rs$ $(A \to_j \alpha \bullet +rs)$, insert a reduce action in row i and all columns a, iff $a \in LA\ (A \to_j \alpha\bullet) - rs(a \in LA\ (A \to_j \alpha\bullet) \cup rs)$.
- Goto entries: These entries are filled exactly in the same way as any LALR (1) parser generator does.

The above description has been implemented as a tool called ASR LALR (1) Parser Generator, based on the open-source and java-based CUP parser generator [25], by applying some non-structural modifications; for example, adding concepts such as look-afore, reduce and no-shift sets, and also by changing the way the parsing table is filled. The main reason for modifying an existing parser generator is to show the simplicity of the proposed method and ease of adapting an existing parser generator via minor modifications. A screenshot of the implemented tool is shown in the Appendix.

6. Conclusions

Ambiguous context-free grammars, theoretically, lead to nondeterministic parsers with over-linear behavior. Nevertheless, in practice, there are parser generators that accept shorter ambiguous grammars for programming languages that are supplemented with some ambiguity-resolving information and produce smaller parse tables capable of deriving deterministic parsers that operate in linear time and space. The history of this practice, which leads to more efficient parsers, tracks back to hand-editing of the conflicts in parse tables, dynamically prompting the user for deciding on a conflict, and augmenting the ambiguous grammar with disambiguation rules to help parser generators to resolve conflicts. None of the previous approaches can generate a correct parse table solely by examining the augmented ambiguous grammar, and more or less require the user interaction. We proposed a more powerful formalism for ambiguous context-free definition of programming languages, where each production rule may be augmented with a reduce set and one or more *no-shift sets*. The former is used by LALR (1) parser generators to resolve reduce-reduce and shift-reduce conflicts and the latter helps in avoiding an incorrect shift action. The corresponding asserted *shift reduce* (ASR) parser generator computes a *look-afore set* with the help of two computed auxiliary functions *Followed-by* and *Last*, in the same way as the conventional look-ahead set uses the follow and First functions. We showed, via several examples from the contemporary programming languages, that the proposed formalism is powerful enough to enable the appropriate parser generator to resolve ambiguities based on static disambiguation rules and without any dynamic interaction with the user (i.e., for instance compiler writer). We have developed an appropriate ASR parser generator to test and support the proposed notation. Finally, we believe that general purpose programming language designers, presumably with deep knowledge of parsing theory and compiler construction, can define the programming language syntax as a concise ambiguous grammar augmented with the required disambiguation rules, as proposed, such that the ASR syntax can be directly used by the ASR parser generator to provide the compiler writer with an efficient deterministic parser. This would not obviate the need for a simple context-free grammar or syntax graph that is usually provided for guiding the programmers to write syntactically correct programs.

For future works, the usefulness of presented ASR approach could be investigated for Domain-Specific Languages (DSLs) [54,55], and the possibility of devising a similar method for LL (1) grammars can be examined.

Acknowledgments

This research was founded in part by IPM under Grant CS1391-2-03, and in part by Shahid Beheshti University.

References

1. Grune, D. and Jacobs, C., *Parsing Techniques: A Practical Guide*, Springer-Verlag, New York Inc. (2008).

2. Slivnik, B. "The embedded left LR parser", *Proceedings of the Federated Conference on Computer Science and Information Systems*, pp. 863-870 (2011)

3. Grune, D. and Jacobs, C.J.H. "A programmer-friendly LL (1) parser generator", *Software Practice and Experience*, **18**(1), pp. 29-38, available at: http://www.cs.vu.nl/~ceriel/LLgen.html (1988).

4. Johnson, S.C. "YACC-yet another compiler compiler", *Computing Science Technical Report*, AT&T Bell Laboratories, New Jersey (1975).

5. Might, M. and Darais, D. "Yacc is dead", available online at CornellUniversity Library (arXiv.org:1010.5023) (2010).

6. Slivnik, B. "LL conflict resolution using the embedded left LR parser", *Computer Science and Information Systems*, **9**(3), pp. 1105-1124 (2012).

7. Aho, A., Lam, M., Sethi, R. and Ullman, J., *Compilers: Principles, Techniques, & Tools with Gradiance*, Addison-Wesley Publishing Company, USA (2007).

8. Sippu, S. and Soisalon-Soininen, E. "LR(k) and LL(k) parsing", *Parsing Theory*, **II**, Springer-Verlag, Berlin (1990).

9. Scott, E. and Johnstone, A. "GLL parsing", *Electronic Notes in Theoretical Computer Science*, **253**(7), pp. 177-189 (2010).

10. Ford, B. "Parsing expression grammars: A recognition-based syntactic foundation", *Proceedings of the 31st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages POPL'04*, pp. 111-122 (2004).

11. Parr, T. and Fischer, K. "LL(*): The foundation of the ANTLR parser generator", *ACM SIGPLAN Notices*, **46**(6), pp. 425-436 (2011).

12. Fischer, C.N., Richard, J. and LeBlanc, J., *Crafting a Compiler with C*, Benjamin-Cummings Publishing Co., Inc. (1991).

13. Aho, A.V., Johnson, S.C. and Ullman, J.D. "Deterministic parsing of ambiguous grammars", *Communication of ACM*, **18**, pp. 441-452, doi: http://doi.acm.org/10.1145/360933.360969 (1975).

14. Hopcroft, J.E., Motwani, R. and Ullman, J.D., *Introduction to Automata Theory, Languages, and Computation*, 3rd Edn., Addison-Wesley Longman Publishing Co., Inc. (2006).

15. Aho, A.V. and Ullman, J.D., *The Theory of Parsing, Translation, and Compiling*, Prentice-Hall, Inc. (1972).

16. Floyd, R.W. "Syntactic analysis and operator precedence", *Journal of ACM*, **10**, pp. 316-333, doi: http://doi.acm.org/10.1145/321172.321179 (1963).

17. Presser, L. and McAfee, J. "An algorithm for the design of simple precedence grammars", *Journal of ACM*, **19**, pp. 385-395, doi: http://doi.acm.org/10.1145/321707.321708 (1972).

18. Wharton, R.M. "Resolution of ambiguity in parsing", *Acta Informatica*, **6**(4), pp. 387-395, doi: http://doi.acm.org/10.1007/BF00268139 (1976).

19. Share, M. "Resolving ambiguities in the parsing of translation grammars", *SIGPLAN Notices*, **23**(8), pp. 103-109 (1988).

20. Schmitz, S. "Noncanonical LALR (1) parsing", *Lecture Notes in Computer Science*, pp. 95-110 (2006).

21. Scott, E. and Johnstone, A. "Right nulled GLR parsers", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **28**, pp. 618-624 (2006).

22. Thorup, M. "Controlled grammatic ambiguity", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **16**, pp. 1024-1050 (1994).

23. Thurston, A. "A computer language transformation system capable of generalized context-dependent parsing", PhD Dissertation, School of Computing, Queen's University at Kingston (2008).

24. Donnelly, C. and Stallman, R. "The bison manual", *Free Software Foundation* (2009).

25. Hudson, S., Flannery, F., Ananian, C., Wang, D. and

Appel, A. "Cup parser generator for java", Available at: http://www2.cs.tum.edu/projects/cup/manual.html (1999).

26. Laski, Z. "Ordered context-free grammars", *Technical Report 99-18*, University of California, Irvine (2000).

27. Passos, L.T., Bigonha, M.A.S. and Bigonha, R.S. "An LALR parser generator supporting conflict resolution", *Journal of Universal Computer Science*, **14**, pp. 3447-3464, doi:10.3217/jucs-014-21-3447 (2008).

28. Harford, A.G., Heurinc, V.P. and Main, M.G. "A new parsing method for non-LR(1) grammars", *Software Practice and Experience*, **22**, pp. 419-437, (1992).

29. McPeak, S. and Necula, G. "Elkhound: A fast, practical GLR parser generator", *Lecture Notes in Computer Science*, pp. 73-88 (2004).

30. Rodriguez-Leon, C. and Garcia-Forte, L. "Solving difficult LR parsing conflicts by postponing them", *Computer Science and Information Systems*, **8**(2), pp. 517-531 (2011).

31. Gray, R., Heuring, V. and Kram, S., Sloam, A. and Waite, W. "Eli: A complete, flexible compiler construction system", Research Reportin Univ. of Colorado at Boulder (1990).

32. Cervelle, J., Forax, R. and Roussel, G. "A simple implementation of grammar libraries", *Computer Science and Information Systems*, **4**(2), pp. 65-77 (2007).

33. Henriques, P.R., Pereira, M.J.V., Mernik, M., Lenic, M., Gray, J. and Wu, H. "Automatic generation of language-based tools using the LISA system", *IEE Proceedings Software*, **152**(2), pp. 54-69 (2005).

34. Porubän, J., Forgác, M., Sabo, M., and Bìhálek, M. "Annotation based parser generator", *Computer Science and Information Systems*, **7**(2), pp. 291-307 (2010).

35. Schmitz, S. "An experimental ambiguity detection tool", *Electronic Notes in Theoretical Computer Science*, **203**, pp. 69-84 (2008).

36. Van Den Brand, M., Scheerder, J., Vinju, J. and Visser, E. "Disambiguation filters for scannerless generalized LR parsers", *Lecture Notes in Computer Science*, pp. 143-158 (2002).

37. Paulson, L.C. "A compiler generator for semantic grammars", PhD Dissertation, Stanford University (1981).

38. Knuth, D.E. "The remaining trouble spots in algol 60", *Communication of ACM*, **10**, pp. 611- 618 (1967).

39. Poole, P.C. "Porable and adaptable compilers", *Compiler Construction: An Advanced Course*, Bauer, F. L. and Eickel, J., Eds., Springer-Verlag, pp. 427-497 (1976).

40. Welsh, J., Snecringer, W.J. and Hoare, C.A.R. "Ambiguities and insecurities in Pascal", *Software Practice and Experience*, **7**, pp. 685-696 (1977).

41. Chan, J., Yang, W. and Huang, J.W. "Traps in Java", *Journal of Systems and Software*, **72**(1), pp. 33-47 (2004).

42. Merril, G.H. "Parsing Non-LR(k) grammars with Yacc", *Software Practice and Experience*, **23**(8), pp. 829-850 (1993).

43. Heering, J., Hendriks, P., Klint, P. and Rekers, J. "The syntax definition formalism sdf-reference manual-", *ACM SIGPLAN Notices*, **24**, pp. 43-75 (1989).

44. Wirth, N. "The programming language Pascal", *Acta lnformatica*, **1**, pp. 35-63 (1971).

45. Wirth, N., *Programming in Modula-2*, 4th Edn., ISBN 0-387-50150-9 (1989).

46. Van Rossum, G., Fred, L. and Drake, Jr., *The Python Language Reference Manual*, ISBN 0-9541617-8-5 (2011).

47. Ierusalimschy, R., de Figueiredo, L.H. and Celes, W., *Lua 5.1 Reference Manual, Lua.org.*, ISBN 85-903798-3-3 (2006).

48. Flanagan, D. and Matsumoto, Y. "The ruby programming language", *Everything You Need to Know: Covers Ruby 1.8 and 1.9, O'Reilly*, ISBN 978-0-596-51617-8, pp. 1-429 (2008).

49. Gosling, J., Joy, B., Steele, G. and Bracha, G., *The Java Language Specification*, 2nd Edn., Addison-Wesley (2000).

50. Earley, J. "Ambiguity and precedence in syntax description", *Acta Informatica*, **4**, pp. 183-192 (1975).

51. Tse, S. and Zdancewic, S. "Concise concrete syntax", University of Pennsylvania (2008).

52. Kernighan, B.W. and Cherry, L.L. "A system for typesetting mathematics", *Communication of ACM*, **18**, pp. 151-157, doi: http://doi.acm.org/10.1145/360680.360684 (1975).

53. Milner, R., Tofte, M., Harper, R. and MacQueen, D., *The Definition of Standard ML*, Revised edition, MIT Press, **1** (1997).

54. Mernik, M., Heering, J. and Sloane, A.M. "When and how to develop domain-specific languages", *ACM Computing Surveys*, **37**(4), pp. 316-344 (2005).

55. Kosar, T., Oliveira, N., Mernik, M., Varanda Pereira, M.J., Črepinšek, M., da Cruz, D. and Henriques, P.R. "Comparing general-purpose and domain-specific languages: An empirical study", *Computer Science and Information Systems*, **7**(2), pp. 247-264 (2010).

Appendix

As was mentioned in Section 5, we have written and tested a computer program in Java programing language which receives an ASR grammar and produces the corresponding LALR (1) parse table. A screen shot of user interface of this ASR parser generator is found in Figure A.1.

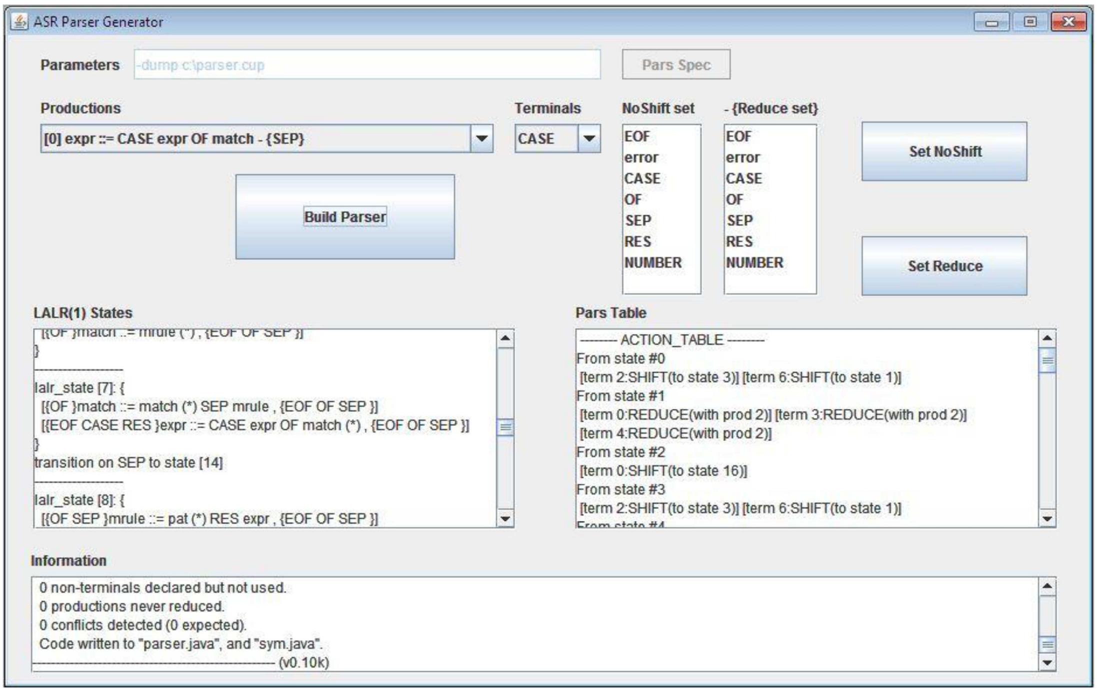

Figure A.1. A screenshot of ASR parser generator.

Biographies

Ghassem Jaberipur is an Associate Professor of Computer Engineering in the Department of Electrical and Computer Engineering at Shahid Beheshti University, Tehran, Iran. He received his BS degree in Electrical Engineering and PhD degree in Computer Engineering from Sharif University of Technology in 1974 and 2004, respectively, MS degree in Computer Engineering in Engineering from UCLA in 1976, and MS degree in Computer Science from University of Wisconsin, Madison, in 1979. His main research interest is in computer arithmetic. Dr. Jaberipur is also affiliated with the School of Computer Science, Institute for Research in Fundamental Sciences (IPM), in Tehran, Iran.

Morteza Dorrigiv received his BS and MS degrees in Computer Engineering from Shahid Beheshti University in 2008 and 2010, respectively. Since 2010, he has been a PhD student in the Department of Electrical and Computer Engineering, Shahid Beheshti University, Tehran, Iran. His research interests include computer arithmetic and compiler design.