Computer and Information Science; Vol. 6, No. 2; 2013
ISSN 1913-8989 E-ISSN 1913-8997
Published by Canadian Center of Science and Education

A Generic Tool for Teaching Compilers

Riad Jabri[1]

[1] University of Jordan, Amman, Jordan

Correspondence: Riad Jabri, Computer Science Department, King Abdullah II School for Information Technology, University of Jordan, Amman 11942, Jordan. E-mail: jabri@ju.edu.jo

Received: March 11, 2013 Accepted: April 24, 2013 Online Published: April 26, 2013

doi:10.5539/cis.v6n2p134 URL: http://dx.doi.org/10.5539/cis.v6n2p134

Abstract

In this paper, we propose a two-fold generic tool for compiler construction. First, it facilitates teaching compilers. Second, it constitutes a new approach for compiler construction. In addition, it enables a smooth transition from theory to practice and introduces a unified approach for the implementation of the different compiler phases. Such unification is achieved based on the representation of the compiler phases as a generic domain that is then mapped into a generic automaton. The generic automaton simulates the behavior of finite and shift-reduce automata, annotated by respective translation schemes. Thus, the tool acts as a scanner, a parser or as syntax directed translator. Without loss of generality, the proposed tool is used within a compiler-teaching framework. Comparisons with similar and well-known approaches have shown that our approach is pedagogical, conceptually simpler, requires less student efforts and more relevant to core curriculum.

Keywords: teaching framework, scanning, parsing, grammars, syntax directed translation

1. Introduction

In this paper, we propose a generic tool for teaching compilers. Usually, compiler construction is taught as a one-semester course having as prerequisites courses in programming languages; data structures and computer organization. Within the framework of the compiler construction course, the students are requested to write a compiler from a given language specification, using automated tools such as Lex and Yacc (Mason & Brown, 1990). However, such teaching approach loosely-couples the theory of scanning and parsing. In addition, it constitutes an ambiguous shift from theory to practice, associated by the inherent difficulties and the typical errors of automated tools (Mallozzi, 2005). Hence, in this paper we set an objective to tightly couple scanning, parsing and translation, as well as to keep a connection and a smooth transition between theory and practice.

Such coupling is based on a proposed generic tool for the construction of scanners, parsers, and syntax directed translators. The proposed tool is used as a unifying tool within the following framework for teaching compilers

1) Introduce compilation, including its different phases.

2) Introduce regular expressions and context-free grammars. Emphasize their roles in specifying the lexical and syntactic levels of high level programming languages.

3) Introduce scanning in terms of finite automata. Emphasize the role of automata as scanners of languages specified by regular expressions (definitions).

4) Introduce parsing in terms of push down storage automata. Emphasize the role of automata as parsers of languages specified by context free grammars.

5) Introduce the concepts of a generic grammar that can be instantiated either by regular expressions and context-free grammars. Emphasize its role in constructing a generic automaton (scanner-parser).

6) Construct the generic automaton as a scanner.

7) Construct the generic automaton as a parser.

8) Introduce the concepts of the syntax directed translation. Emphasize the role of the translation schemes.

9) Consider the remaining compilation phases in their subsequent order. Extend the automaton as type-checker, intermediate code generator and as code generator.

The outlined framework is consistent with ACM's Curriculum 2001. It is motivated by the strategies and the teaching approaches suggested in (Aho, 2008; Waite, 2006). However, the proposed framework adopts a unified approach for teaching theory and its practical application. Such unification is achieved by adopting a generic grammar that can be either instantiated by context-free grammars or by regular definitions. Such grammar is annotated by appropriate translation schemes. As such, the augmented grammar is then transformed into an augmented generic automaton (AGA). The states and the transitions of AGA are defined based on combining concepts from LR automata, finite automata, with embedded translation schemes (Aho et al., 2007). Hence, the parsing behavior of AGA is generic. It simulates finite automata, shift–reduce automata and syntax directed translator. However, AGA is a nondeterministic. It is efficiently transformed into a reduced one, called RAGA, using a proposed subset construction approach. In addition, the subset construction produces parsing and translation tables that specify the RGA parsing/ translation actions, respective a given state and each one of the input symbols. A simulator is then constructed to simulate the run (scanner/parser/translator) of RGA on strings derived from a generic grammar.

Our parsing approach is based on position parsing automata (PPA), as proposed in (Jabri, 2009, 2012). However, it extends PPA to act as a scanner/parser/translator and uses different construction approach that handles recursion by reduced stack activities in a way similar to the generalized bottom up parsers and the reduction-incorporated parsers, as proposed in (Ayock, 2001; Johnstone & Scott, 2007; Scott & Johnstone, 2005) respectively. In addition, it handles nondeterminism in a way similar to SLR parsing as proposed in (Jabri, 2012).

The remainder of this paper is organized as follows. Section 2 presents preliminaries. Section 3 presents the proposed generic tool and the respective algorithms for its implementation. Section 4 presents experimental results, followed by a discussion and a conclusion that are given in Section 5 and Section 6 respectively.

2. Preliminaries

For our further discussions, we assume the following definitions based on the ones given by Aho et al. (2007) and by Jabri (2012). These definitions are used to specify the subsequent phases of a compiler as follows:

- The specification needed for lexical analysis and parsing are based on regular definition and context free grammars as given in Definition 1 and Definition 2 respectively. A unified definition for both is based on a generic grammar as given in Definition 3.
- The specification needed for the remaining compiler phases are based on syntax directed translation schemes (Aho et al., 2007) where program fragments are embedded within the productions of the generic grammar. These fragments are enclosed between braces ({...}) and denoted as translation schemes.

Definition 1. Given an alphabet Σ, a regular expression (RE) is defined as a notation to describe all languages that can be built from the symbols of Σ by applying the operations: union (|); concatenation (.) and exponentiation (*). RE is then inductively defined as follows:

- The symbol ε is RE and if $x \in \Sigma$ then x is RE.
- Let x, y be RE .Then RE is inductively defined as: xy is RE; x | y is RE; x* is RE and (X) is RE.

Definition 2. Given an alphabet Σ, a regular definition (RD) is a sequence of definitions of the form: $RD_1 \rightarrow RE_1$, $RD_2 \rightarrow RE_2, \ldots, RD_n \rightarrow RE_n$, where $RD_i \notin \Sigma$ and RE_i is defined over $\Sigma \cup \{RD_1, RD_{2,\ldots,} RD_{i-1}\}$.

Definition 3. An Action-annotated Generic Grammar is defined by the 5-tuple AGG = (Σ, N, P, S, TS) that is either instantiated by a context free grammar (CFG) or by RD. In addition, it is annotated by translation schemes to specify a particular translation as follows:

1) AGG is instantiated by CFG, if:
   - Σ is an alphabet of terminal symbols.
   - N is a finite set of nonterminal symbols, where $S \in N$ is a starting symbol.
   - P is a finite set of productions *p* having the form $p: A \rightarrow V$, where $A \in N$ and $V \in (\Sigma \cup N)$.
   - TS is set of translation schemes to specify particular translation phase, where the individual productions are annotated by such schemes as in $p: A \rightarrow V$:: { translation scheme}.

2) AGG is instantiated by RD, if:

- Σ is an alphabet of symbols.
- $N = \{RD_1, RD_{2,\ldots,} RD_n, \} \cup OP$, where $\{RD_1, RD_{2,\ldots,} RD_n,\}$ is an ordered set of nonterminal symbols ; $S \in N$ is a starting symbol and OP are special nonterminals representing the operations: |, ( ), (.) and *.
- P is a finite set of productions *p* having the form *p*: $A_i \rightarrow RE_i$, where $A_i \in \{RD_1, RD_{2,\ldots,} RD_n,\}$ and RE_i defined by applying OP over $(\Sigma \cup \{RD_1, RD_{2,\ldots,} RD_{i-1}\})$.
- TS is a set of translation schemes to specify the translation of lexical entities into appropriate tokens as in *p*: $A_i \rightarrow RE_i$ :: { translation scheme }.

Subsequently, in our further discussion, we use a string as a generic term to either denotes strings, generated by RD, or sentences generated by CFG. These strings are annotated by appropriate translation schemes.

Example 1. Let G = (Σ, N, P, S, TS) be AGG instantiated by RD, where: (a-z, 0-9, +, *)$\in \Sigma$, (Id, num, op, ws, token, tokenlst) $\in$ N , P = { letter $\rightarrow$ a | b|…|z, digit $\rightarrow$ 0|…|9, ws $\rightarrow$ blank, id $\rightarrow$ letter (letter | digit)*, num $\rightarrow$ digit. digit*(.digit)*, op $\rightarrow$ + |*, token $\rightarrow$ id | num | op, tokenlst $\rightarrow$ tokenlst ws token} and TS is a set of translation schemes. Examples for strings generated by G are: xy2z, 23.5 and 56.9. A possible annotation of G is as follows:

id $\rightarrow$ letter (letter | digit)*::{ Return (ID)}, where ID is an encoding of a token of type id; num $\rightarrow$ digit. digit*(.digit)*::{ Return (NUM)}, where NUM is an encoding of a token of type num and op $\rightarrow$ + |* ::{ Return (OP)}, where OP is an encoding of a token of type op.

Example 2. Let G = (Σ, N, P, S, TS) be AGG grammar. It is instantiated by CFG as follows: Σ = {id, +, *}, N ={E, T, F }, P ={ E $\rightarrow$ E + T | T, T $\rightarrow$ T*F | F, F $\rightarrow$ id }and TS is a set of translation schemes . Examples for strings generated by G are: id*id, id, and id +id*id. A possible annotation of G for type checking is as follows: F $\rightarrow$ id ::{ Type.F = Type.id} and E $\rightarrow$ E + T:: { Type.E = f (Type.E, Type.T)}.

3. The Proposed Generic Tool

Let G = (Σ, N, P, S, TS) be an AGG grammar. The proposed generic tool (GCT) is defined either as a scanner, if G is instantiated by RD, or as a parser, if G is instantiated by CFG. In addition, GCT is defined in away that permit its transformation to a translator respective to the translation schemes (TS), annotating the individual productions of G. As such, GCT has the structure shown in Figure 1, where:

AGG $\rightarrow$ Construction Algorithm $\rightarrow$ AGA $\rightarrow$ Subset Construction $\rightarrow$ RAGA

Input string $\rightarrow$ RGAA simulator $\rightarrow$ Output

Figure 1. The structure of GCT

- AGA is a nondeterministic automaton defined and constructed as given by Definition 4 and Section 3.1.
- RGA is a reduced automaton obtained as a result of applying a subset construction on AGA as given in Section 3.2
- A simulator for RGA constructed as given in Section 3.3. Such simulator acts either as a scanner/ translator or a parser/ translator.
- Once a string is fed as an input to RGA simulator, an output is then produced as consisting of a respective sequence of AGG's production reductions and the results of the execution of the translation schemes annotating the individual AGG's productions.

Definition 4. An Augmented Generic Automaton (AGA)

Let (Σ, N, P, S, TS) be AGG grammar. AGA is then defined by the 5-tuple (($\Sigma \cup \varepsilon$), Q, q_{in}, q_{fin}, TA), where:

- Σ represents strings generated by the AGG grammar.
- Each grammar symbol V$\in$ ($\Sigma \cup$ N) is represented by the pair (V^i, V^f}) to represent a prediction and an acceptance of V in an assumed translation. Consequently, The GCCT states respective to each V$\in$ ($\Sigma \cup$ N) are defined as the set {($q^i = V^i$, $q^f = V^f$)}, where: q^i is an initial state instantiated by V^i and acts as a predictor (scanner) for V. The state q^f is a final one, instantiated by V^f and acts as its respective

acceptor. In addition and if V is associated with a translation scheme, the state q^f is augmented by such scheme as a translation action to be executed upon transition to q^f. Hence, the set of AGA states is defined as Q = $\{(q^i = V^i, q^f = V^f) \mid V \in (\Sigma \cup N)\}$ where, each state is instantiated by a respective grammar symbols.

- ($q_{in} = S^i$) and ($q_{fin} = S^f$) are the initial and final states, instantiated by the symbols respective to the starting grammar symbol of the AGG grammar.
- TA is a set of the translation actions of AGA. It consists of parsing actions annotated by translation schemes. The parsing actions consist of move transitions and reductions. Upon transitions and reductions, the action specified by such scheme is executed as an integral part of the parsing action. The move parsing actions are defined by the set SPA = $\{\sigma (q_{i,}, V) = (q_{j,}) :: \{$translation scheme$\}\}$, where $\sigma (q_{i,}, V) = (q_{j,})$ specifies a subsequent state $q_j \in Q$ for a given state $q_i \in Q$ and a given grammar symbol $V \in T$. Upon transitions, a program fragment is performed as indicated by the annotating translation scheme. The reduce parsing actions are defined by the set RPA = $\{\delta (q, V) =$ reduce(r):: {translation scheme}$\}$, where $\delta (q, V) =$ reduce(r) defines a reduction rule (r), for every $V \in N$, $q \in Q$ such that q has been instantiated by V^f and V is the LHS (r). RPA performs the indicated {translation scheme}, If the reduction is associated with such scheme.

The states and the parsing actions of AGA are dependent on the production types; the types of the grammar symbols and their respective positions in the individual productions of AGG, They are inductively constructed as given in Section 3.1.

3.1 AGA Construction Approach

Let G = (Σ, N, P, S, TS) be an AGG grammar, where:

- G is either instantiated by CFG or RD. Respectively, the set of productions P is generic and is either instantiated by grammar productions or regular definitions.
- The productions of CFG are classified into three types: simple productions, productions having alternatives and productions having embedded recursion.
- Given $p \in P$, the grammar symbols V in p are classified into the following types: terminals, ranked terminals, nonterminals, a recursive-head (V is LHS (p) and p is a production having embedded recursion) and a recursive-instance ($V \in$ RHS (p) and V is LHS(p)). In addition, each nonterminals with repeated occurrences in RHS of different productions is classified as repeated–instance.
- The right hand side of RD are regular expressions on which the following operations are applied: Concatenation; alternation; exponentiation.
- Considering a particular compilation phase, respective translation schemes (TS) are defined and used to annotate the set P.

Assuming that the right hand sides (regular expressions) of RD are decomposed into their constituent sub expressions, Algorithm 1 constructs AGA for RD (AGA (RD)) as well as for CFG (AGA(CFG)). The construction of AGA (RD) or (AGA (CFG)) proceeds according to the same steps from which Algorithm 1 is composed. However, the construction of AGA (RD) excludes the step covering embedded recursions, while the construction of AGA (CFG) excludes the one covering exponentiation.

Algorithm 1: AGA Construction Algorithm

Input: An AGG grammar G = (Σ, N, P, S, TS).

Output: GA defined by the 5-tuple (($\Sigma \cup \varepsilon$), Q, q_{in}, q_{fin}, TA) and its respective transition graph.

Method:

Initially, the states of AGA are constructed as the set as Q = $\{(q^i = V^i, q^f = V^f) \mid V \in (\Sigma \cup N)\}$. AGA is then inductively constructed as follows:

1) Basis:

- Let (q_i , q_f) $\in$ Q be an arbitrary states, denoted as an initial state and a final one respectively, AGA for the grammar symbol ε (AGA (ε)) is then constructed as shown in Figure 2.

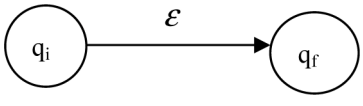

Figure 2. AGA (ε)

It constitutes an ε -transition (σ $(q_{i,}, \varepsilon) = (q_{f,})$ ::{ translation scheme}) between q_i and q_f, associated with a translation scheme.

- Let V $\in$ Σ , AGA (V) is then constructed as consisting of two states (Fig.3) with the following transitions σ $(q_i, V) = (q_f)$ :: { translation scheme}

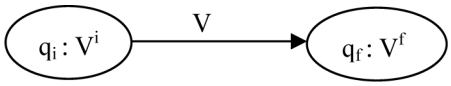

Figure 3. AGA (V)

2) Induction:

2.1 Let p$\in$ P: LHS(p) $\rightarrow$ RHS(p) be either a simple production or a concatenation of sub expressions; AGA (p) is then constructed as shown in Figure 4.

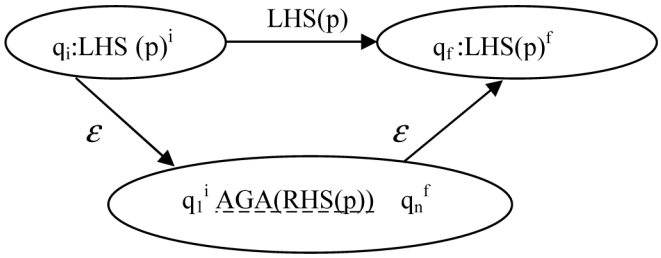

Figure 4. AGA (p)

where: q_i is the initial state and q_i the final state of AGA (p). AGA (RHS(p)) is an aggregation of AGA_1, …, and AGA_n respective to the individual grammar symbols from which RHS (p) is composed. Hence, it is constructed as shown in Figure 5.

Figure 5. AGA (RHS (p))

Each AGA_i has $q_i{}^i$ and $q_i{}^f$ as an initial and final states respectively. AGA(p) has the following translation actions :

SPA: σ $(q_{i,}, \varepsilon) = (q_1{}^i)$ ::{ translation scheme},where $q_1{}^i$ is an initial state of AGA (RHS(p)).

SPA: σ $(q_n{}^f, \varepsilon) = (q_{f,})$ ::{ translation scheme} where $q_n{}^f$, is the final state of AGA (RHS(p)).

SPA: { σ $(q_i{}^f, \varepsilon) = (q_{i+1}{}^i)$ ::{ translation scheme} | i=1,…, n-1 }.

RPA: δ $(q_f, LHS(p)) =$ reduce (LHS(p)) ::{ translation scheme}.

2.2 Let p$\in$ P: LHS(p) $\rightarrow$ $RHS_1(p)$ |…|$RHS_n(p)$ be either a production with alternative right hand sides or alternative sub expressions $RHS_1(p)$, RHS_2 (p),.., $RHS_n(p)$. AGA (p) is then constructed as shown in Figure 6.

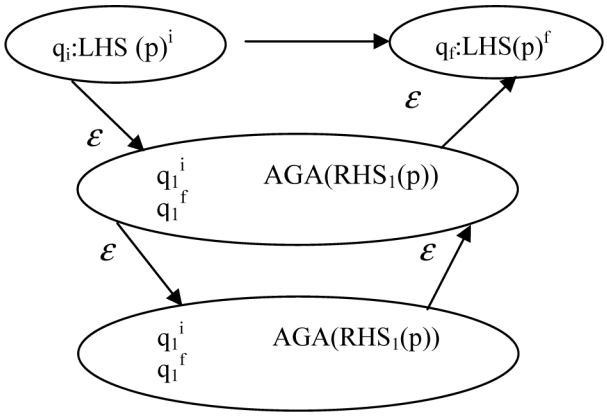

Figure 6. AGA for a production with alternatives

Each alternative RHS_i (p) has a respective AGA(RHS_i (p)) with initial and final states q_i^i and q_i^f respectively. The initial and the final states of AGA (p) are q_i and q_f. AGA(p) has the actions:

SPA: { σ ($q_{i,}$ ε) = (q_j^i,) ::{ translation scheme} | j=1,..n},where q_j^i is the initial state of AGA_j.

SPA: { σ (q_j^f, ε) = ($q_{f,}$) ::{ translation scheme}| j=1,..n}, where q_j^f is the final state of AGA_j.

RPA: δ (q_f, LHS(p)) = reduce (LHS(p)) ::{ translation scheme}.

2.3 Let p∈ P: LHS(p) $\rightarrow$ RHS(p) be a production having embedded recursion, where LHS(p) ∈ RHS(p) at position (i). AGA is constructed as shown in Figure 7.

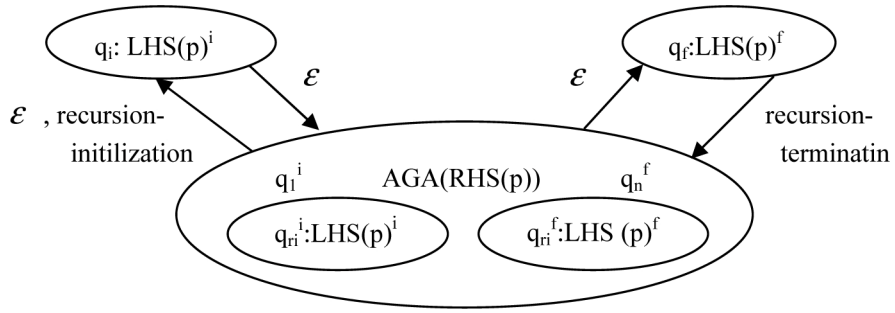

Figure 7. AGA for a production with embedded recursion

AGA(p) consists of : the initial state q_i, the final state q_f and the AGA respective to (RHS(p)).

AGA(RHS(p)) has the states q_1^i, …, q_{i-1}^f, q_{ri}^i, q_{ri}^f, q_{i+1}^i,…, q_n^f, where q_1^i and q_n^f are the initial and final states; q_{ri}^i and q_{ri}^f are the initial and final respective to the embedded recursion. AGA actions are as follows:

SPA: σ ($q_{i,}$ ε) = (q_1^i,) ::{ translation scheme}::{ recursion-initialization$_1$}, where{recursion-initialization}is a program segment, executed during parsing, that creates a stack (q_i) initialized with the symbol ε at the top.

SPA: σ (q_n^f, ε) = ($q_{f,}$) ::{ translation scheme }

SPA: σ (q_{i-1}^f, ε) = (q_{ri}^i,) ::{ translation scheme }

SPA: σ (q_{ri}^i,, ε) = (q_i) ::{ translation scheme}::{ recursion-initialization$_2$}, where{ recuersion-initialization$_2$} is a program segment, executed during parsing, that initializes a recursive path with a return address q_{ri}^f, pushed on the top of stack (q_i).

RPA: δ (q_f, LHS(p)) = reduce (LHS(p)) ::{ translation scheme} ∪ { recursion-termination}, where{ recursion-termination } is a program segment, executed during parsing, that terminates a recursive path by performing an ε -transition to the address at top of stack (q_i), if top(stack (q_i)) $\neq \varepsilon$.

SPA: σ (q_{ri}^f,, ε) = (q_{i+1}^i) ::{ translation scheme}.

2.4 Let {p∈ P: LHS(p) → RHS(p)} be a set of productions having (A∈ N) ∈ RHS(p), where A is classified as repeated-instance. AGA (A) is then constructed for a selected instance. An AGA for each one of the other instance is constructed following the steps for embedded recursion.

2.5 Let p∈ P: LHS(p) → RHS(p) be a production (regular definition) having a sub expression r* at position (i), where r*∈ RHS(p) at position (i). AGA (p) is constructed as shown in Figure 8.

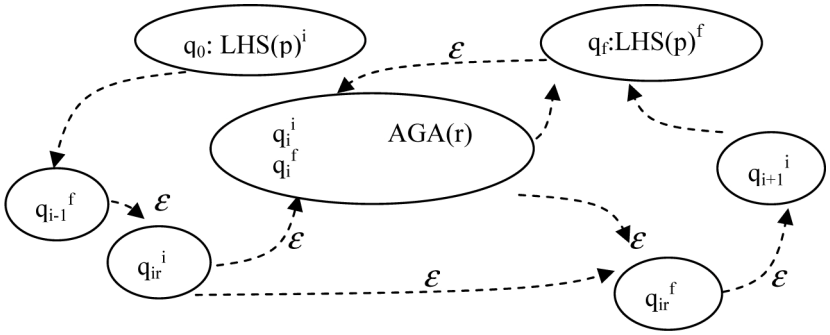

Figure 8. AGA for regular definitions

AGA (p) consists of: the initial state q_0, the final state q_f and AGA (RHS (p))respective to (RHS (p)). AGA (RHS (p)) has the states q_1^i, …, q_{i-1}^f, AGA(r*), q_{i+1}^i,…, q_n^f, where: q_i^i and q_i^f are the initial and final states for AGA(r*) ; q_{ri}^i and q_{ri}^f are the initial and final respective to the exponentiation. AGA actions are:

SPA: σ ($q_{0,}$ ε) = (q_1^i) ::{ translation scheme}

SPA: σ (q_n^f, ε) = (q_f) ::{ translation scheme }

SPA: σ (q_{i-1}^f, ε) = (q_{ri}^i) :: { translation scheme }

SPA: σ (q_{ri}^i, ε) = (q_i^i) ::{ translation scheme}

SPA: σ (q_i^f, ε) = (q_i^i) ::{ translation scheme}

SPA: σ (q_i^f, ε) = (q_{ri}^f) ::{ translation scheme}

SPA: σ (q_{ri}^i, ε) = (q_{ri}^f) ::{ translation scheme}

SPA: σ (q_{ri}^f, ε) = (q_{i+1}^i) ::{ translation scheme}

RPA: δ (q_f, LHS (p)) = reduce (LHS(p)) ::{ translation scheme}

Example 3. Let G = (Σ, N, P, S, TS) be GAA grammar with simple productions. Where: (id, +, *)∈ Σ, (E, T, F) ∈ N and P ={ E →E+T T, T→ T*F F, F→id }. A nondeterministic automaton representing AGA respective to G is shown in Figure 9.

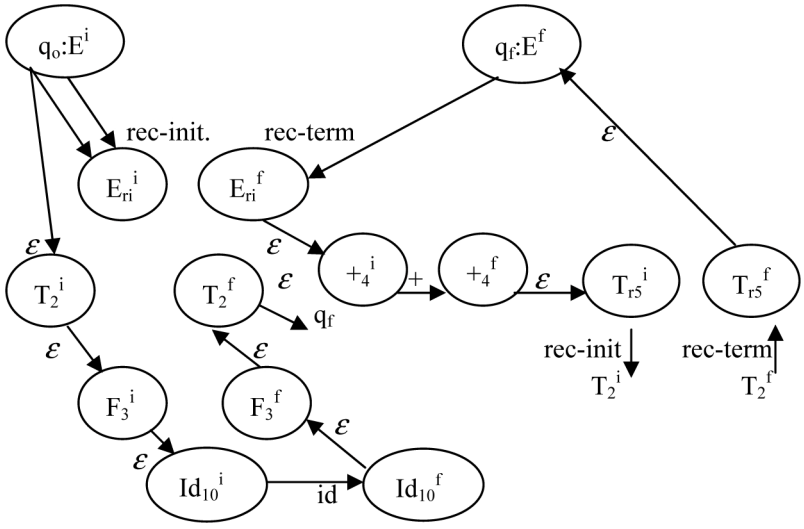

Figure 9. AGA for the grammar of Example 3

Example 4. Let G = (Σ, N, P, S, TS) be GGA grammar instantiated by regular definitions as given in Example 1. Fig. 10 shows a nondeterministic automaton representing AGA respective to the productions : letter → a | b|…|z; digit → 0|…|9, id → letter (letter | digit)* ; token → id; and tokenlst → token.

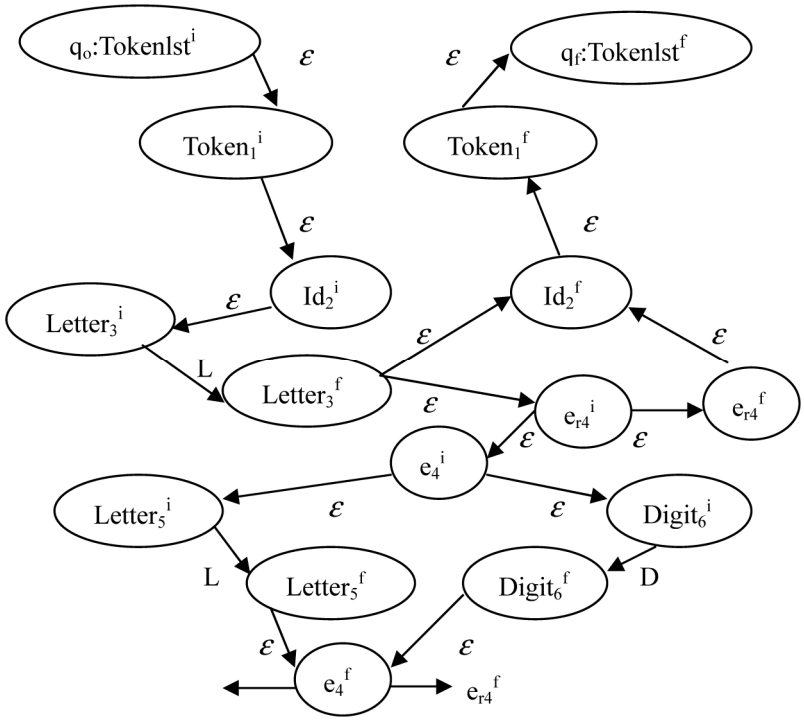

Figure 10. AGA for the grammar of Example 4

3.2 RAGA Construction Approach

In this section, we propose a construction approach for the reduced automaton RAGA based on extension of the subset construction algorithms given in (Aho et al., 2007; Jabri, 2012). Such extension is a twofold. First, it covers a wider class of grammars including RD and CFG. Second, it has a deterministic behavior upon the RGAG reduce actions, as well as upon return actions as determined by recursion-termination.

The constructed automaton in (Jabri, 2012) has multiple transitions and respective reductions. In contrast, according to the proposed approach, the sets of follow symbols for the nonterminals are used to determine their respective reduction in a deterministic way, similar to the one adopted for SLR parsers (Aho et al., 2007). Based on such approach a proposed algorithm, denoted by Algorithm 2 is given below. Having the AGA states Q_{GT} and their respective translation actions TA_{GA} = {AGA.PAS, AGA.PAR} as an input, Algorithm 2 constructs RAGA respective to the nondeterministic one. RAGA (G) is represented by its respective states Q_{RGA} and translation actions TA_{RGA}. In its turn, two tables represent TA_{RGA} The first one constitutes a transition table (TRT), where each entry (TRT [q_o, Vi]) of such table specifies the transitions of type SPA. The second table RTT specifies the reductions (RPA) and to the translation actions (as indicated by the translation schemes respective to SPA and RPA) to be performed by RAGA (G) during its run on an input alphabet, generated from the grammar G. Thus, Algorithm 2 computes Q_{RGA} using an ε -closure function (Aho et al., 2007).This function closes the initial states and the final states respective to the different grammar symbol types. However, it does not close the initial states instantiated by grammar symbols of types recursive-instances (V_{ri}^i, V_{ri}^f) and repeated-instance. Algorithm 2 handles these symbols by creating respective states and transitions, including the translation schemes {recursion-initialization} and {recursion-termination}.

Algorithm 2. RAGA Construction

Input: AGA automaton, defined by the 5-tuple: ($(\Sigma \cup \varepsilon)$, Q_{GA}, TA_{GA}, q_{in}, q_{fin}).

Output: RAGA automaton, defined by the 5-tuple: ($(\Sigma \cup \varepsilon)$, Q_{RGA}, TA_{RGA}, q_{in}, q_{fin}).

Method:

Step1: Construct the set Q_{RGA} of RAGA states and a respective set TA_{RGA} of type shift as ε - closures of the states in Q_{GA} and their respective transitions.

Step2: Complete the construction of the set TA_{RGA} and construct the respective transition and reduction/translation tables, represented by the tables TRT and RTT.

Step1 of Algorithm 1 proceeds by executing the program segment as shown in Figure 11, where it starts by constructing the initial state RAGA.qin and then repeatedly considers each constructed state q in QRGA to constructs the subsequent states qi as ε - closure (σ (qpii , V)) such that qpii in q and (σ (qpii , V) $\in$ TAGA. The transitions TARGA respective to q are then constructed as σ (q, V) = qi). In addition, it registers the states representing recursion and repetition and constructs ε - closures respective the states initialized by grammar symbols of type qpif. Step2 of Algorithm 1 proceeds by executing the program segment as shown in Figure 12, where: First, it constructs the transitions respective to the registered-heads as well as the states and transitions respective to the registered-instances. Second, it constructs the reduce / translation actions respective RAGA states and represents these actions (and the RAGA transitions) as entries in the tables RTT and TRT.

Example 5. Let G = (Σ, N, P, S) be a grammar with simple productions as given in Example 3. G is annotated by translation schemes to translate expressions in infix notation into ones in postfix notation as given in (Aho et al., 2007). The automaton representing RAGA (G) as constructed by Algorithm 2 is given in terms of its transition graph and the tables TRT and RTT as shown in Figure 13 and Table 1 respectively.

Example 6. Let G = (Σ, N, P, S) be a generic grammar instantiated by regular definitions as given in Example 2.2 but annotated by translation schemes to produce recognized tokens. The automaton representing RAGA(G) as constructed by Algorithm 2 is given in terms of its TNT and TRT as shown in Table 2.

$RAGA.q_{in}$ = ε -closure ($AGA.q_{in}$), Add $RAGA.q_{in}$ to Q_{RGA} unmarked;
While unmarked states in Q_{RGA}
{ Select next-unmarked state q from Q_{RGA}; mark q;
For (each state q_{pi}^{i} in q such that $q_{pi}^{i} = v_{pi}^{i}$, V∈ N) and
(v_{pi}^{i} is (recursive-head or repeated- head))
{ Add q_{pi}^{i}) to registered-heads };
For (each state q_{pi}^{i} in q such that $q_{pi}^{i} = v_{pi}^{i} \in$ N and
(v_{pi}^{i} is (recursive-instance or repeated- instance))
{ Add q_{pi}^{i} to to Q_{RGA} ,marked);
q^{f} = NewState (ε - closure (selectFinalState (Q_{GA}, q_{pi}^{i})))
Add q^{f} to Q_{RGA} unmarked; Add (q^{f}, q_{pi}^{i}) to registered-instances; }
For (each state q_{pi}^{i} in q such that $q_{pi}^{i} = v_{pi}^{i}$ and V∈ Σ)
{qi = ε - closure (σ (q_{pi}^{i} , V)) such that (σ (q_{pi}^{i} , V) $\in$ TA_{GA};
Add qi to Q_{RGA} unmarked; Add (σ (q , V)= qi) to TA_{RGA};

Figure 11. The construction of RAGA states and transitions

For each state q in registered-head
{ Add (j, RTT (recursion-initialization$_1$ (q));Add(TRT[q,-] =j)
For each pair (q_{pi}^i, q^f) in registered-instances
{ q = Select (q_{pi}^i, registered-head); Add (TRT[q , ε] = q_{pi}^i)
For all V∈ Σ such that (σ (q , V)= qi) ∈ TA_{RGA} {Add (TRT[q_{pi}^i,V] = qi }
Add (j, RTT(recursion- initialization$_2$ (q^f)); Add(TRT[q,-]=j);
}
For each state q in Q_{RGA}
{ For each V∈ Σ such that such that (σ (q , V)= qi) ∈ TA_{RGA}
{ Add (TRT[q , V] = q_i)};
For (every X∈ N such that X^f∈ q)) and (every V∈ Follow (X));
{Add (δ (q, V) = reduce r (X) ::{ translation scheme}) to TA_{RGA}
Add (j, RTT (TA_{RGA} (δ (q, V)) Add(TRT [q,V]=j;
If (X^f is a recursive-head or a repeated-head)
{Add (j, RTT (recursion- termination), Add(TRT [q,V]=j)}
If (X is a starting symbol) {Add (j, RTT (Accept), Add(TRT [q,$]=j)}
}

Figure 12. The Construction of the RAGA reduce/ translation actions and its tables

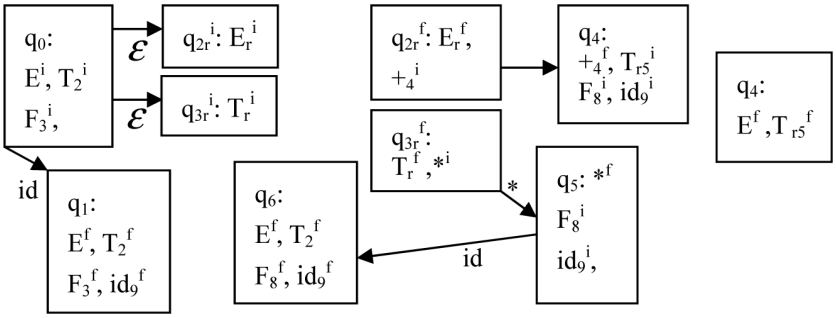

Figure 13. The transition graph of RAGA for the grammar of Example 5

Table1. The transition/reduction tables of RAGA for the grammar of Example 5

TRT table						RTT table	
State	Input Symbols					Entry	Action
	ε	id	+	*	$		
q_0	q_{2r}^i, q_{3r}^i	q_1				1	R (F ⟶ id, T ⟶ F, E ⟶ T), :: {(print(id)}, Accept
			2	3	1	2	R (F ⟶ id, T ⟶ F, E ⟶ T) ::{(print(id)), {(rec-termination)= q_{2r}^f}
q_{2r}^i	q_{2r}^i	q_1	4	4	4	3	R (F ⟶ id, T ⟶ F), S (print(id))::{ (rec-termination)= q_{3r}^f}
q_{2r}^f		q_4				4	{ (recursion-initialization)}
q_{3r}^i	q_{3r}^i	q_1	5	5	5	5	{ (recursion-initialization)}
q_{3r}^f		q_5				6	R (E ⟶ E+T)::{ (print(+))};Accept
q_{4r}^i	q_{3r}^i	q_1				7	R (E ⟶ E+T)::{(print(+))::{(recursion-termination)= q_{2r}^f }
q_{4r}^f		q_1	7	8	6	8	R (E ⟶ E+T)::{(print(+))};{(recursion- termination}
q_5		q_6				9	R (F ⟶ id , T ⟶ T*F)::{ (print(id), {(print(*)};Accept
q_6			10	11	9	10	R (F ⟶ id ,T ⟶ T*F)::{ (print(id)}, { (print(*)};{(recursion- termination)= q_{2r}^f }
						11	R (F ⟶ id ,T ⟶ T*F)::{ (print(id), {(print(*)};{(recursion- termination)= q_{3r}^f }

Table 2. The transition/reduction tables of RAGA for the grammar of Example 6

TRT table								RTT table	
State	Input Symbols							Entry	Action
	$\mathcal{E}$	+	*	L	D	other ws	$		
q_0	q_{1r}^{i}	q_3	q_4	q_7	q_5	1	1	1	{ (recursion-initialization) = q_{1r}^{f} }
q_{1r}^{i}	q_{1r}^{i}	q_3	q_4	q_7	q_5	2	2	2	S (recursion-initialization) = q_{1r}^{f} }
q_{1r}^{f}						q_2		3	R (OP $\rightarrow$ +, T $\rightarrow$ OP, TL $\rightarrow$ T)::
q_2		q_3	q_4	q_7	q_5				{(print(op,+)}, Accept
q_3						4	3	4	R (OP $\rightarrow$ +, T $\rightarrow$ OP, TL $\rightarrow$ T)::
q_4						6	5		{(print(op,+)}, { (recursion-termination)}
q_5				q_5	q_5	q_6		5	R (OP $\rightarrow$ *, T $\rightarrow$ OP, TL $\rightarrow$ T)::
q_6						8	7		{(print(op,*)}, Accept
q_7				q_7	q_7	q_8		6	R (OP $\rightarrow$ *, T $\rightarrow$ OP, TL $\rightarrow$ T)::
q_8						10	9		{(print(op,*)}, { (recursion-termination)}

3.3 RAGA Simulation

Let G be an AGG grammar, respective AGA(G) and RAGA(G) are constructed by Algorithm 1 and Algorithm 2. The run of RAGA(G) on input strings generated by G is then simulated by Algorithm 3, as given below.

Algorithm 3. RAGA Simulator

Input: An input string (w$) and the RAGA(G) automaton, respective to a grammar G. RAGA(G) is represented by two tables: the transition table TRT and the reduction-translation table RTT

Output: Set of reductions as they occur, if w L (G) Translation actions as indicated by parsing actions

Method:

Initially, the simulator is in its initial configuration, consisting from the RAGA(G) initial state q_{in} and the input string, represented as INPUT[ ] = w$. As the individual input symbols are read, the simulator performs their respective parsing actions. This is achieved by executing the following program segment.

NextStates= NextStates $\cup$ {q_{in}};

Create-simulation-path ({ q_{in} });

For each recursive-head q_r^{i} in NextStates

{ perform RTT [TRT[$q_{r,-}$]].S (recursion-initialization)};

For i = 1 to MaxSize (INPUT)

{Subsequentstates= ϕ

For each state q in NextStates

{ Movet-states = ϕ; Reduce-states = ϕ; Return-states = ϕ;

Shift-states = TRT [q, INPUT [i]];

If (Shift-states = ϕ) { output(Error)}

Elseif

{For each state s in Shift-states

{Movetstates = Movetstates $\cup$ s;

Create-simulation-path (s, simulation-path(q));

If (s =q_r^{i}) { perform RTT[[TRT[s, -]].S (recursion-initialization)};

If (Reduce-state (s))

{ Output (RTT[[TRT[s, LA]].R(reductions);

Output (RTT[[[TRT[s, LA]].S (translation scheme)

```
            If (s= q_r^i )
            { Return-states = Return-states ∪
                ( perform RTT[[[TRT[s, LA]].S (recursion-termination)};
        }
        Subsequent-states = Subsequent-states ∪ Shift-states ∪Return-states;
    }   NextStates = Subsequent-states;
}
If ( NextStates ∩ qfin ≠ ε ){ Accept ( Input); }
```

Example 7. Let a * c be a string generated by the grammar of examples 2.2 and 3.4. The run of RAGA on such string is shown in Table 3.

Example 8. Let id * id be a string generated by the grammar of examples 2.1 and 3.3. The run of RAGA on such string is shown in Table 4.

Table 3. The run of RAGA simulator on a string

Current state	Current input	simulation action
q_0; q_{1r}^i	a * b$	{ (recursion-initialization)}
q_7	* b$	
q_8		{ (recursion-termination)}
		R (T → id), R(TL → T); {(print(id,Value))}
q_{1r}^f; q_{1r}^i	* b$	{ (recursion-initialization)}
q_4	b$	{ (recursion-termination)}
		R (OP → *), {(print(op,*)}, R(T → op)
q_0; q_{1r}^i	b$	{ (recursion-initialization)}
q_7	$	
q_8		{ (recursion-termination)
		R (T → id), R(TL → T); {(print(id,Value)}

Table 4. The run of RGCCT simulator on a string

Current state	Current input	simulation action
q_0; q_{1r}^i	a * b$	{ (recursion-initialization)}
q_7	* b$	
q_8		{ (recursion-termination)}
		R (T → id), R(TL → T); {(print(id,Value))}
q_{1r}^f; q_{1r}^i	* b$	{ (recursion-initialization)}
q_4	b$	{ (recursion-termination)}
		R (OP → *), {(print(op,*)}, R(T → op)
q_0; q_{1r}^i	b$	{ (recursion-initialization)}
q_7	$	
q_8		{ (recursion-termination)
		R (T → id), R(TL → T); {(print(id,Value)}

4. Experimental Results

To validate the correctness, the simplicity and the applicability of our proposed approach, and in addition, to inspect how easily the transition from theory to practice can be made, the proposed GCT has been assigned as a graduation project. Based on the GCT concepts and its implementation algorithms, the assignment is reduced to the implementation of a version the can be used by students as tool for the construction of the different compiler's phases. The parsing phase has been selected as representative one. As such, GCT has been implemented, using Microsoft Visual C++ 6.0., by the program given in Figure 14, where:

- InteractionContext ( ) is a method to input a AGG grammar defined by the 4-tuple G = (Σ, N, P, S). It is implemented by the three functions: Terminals ( ); NonTerminals ( ); and Productions ( ) to facilitate the input of the terminal symbols (Σ), the nonterminal symbols (N) and the AGG productions (P) respectively. The productions are entered one by one (p), where LHS(p) is entered first followed by the symbols of RHS (p), from left to right. This enables automatic indexing of the grammar symbols by their respective positions within each production. As the productions are entered they are mapped into respective automaton (AGA) states (Q), where for each grammar symbol $V_{p,}$ its corresponding initial state $q_p^i \in Q$ and final state $q_p^f \in Q$ are created.

```
int main( )
{
InteractionContext( )
 { Terminals ( );       //A function to input the terminal    symbols.
     NonTerminals ( ); //A function to input the nonterminal symbols.
     Productions ( );    //A function to input the grammar productions.
 }
  construct_aga(    );    //A function    to construct AGA and
                                its respective transition graph
 construct_raga (    );    //A function    to construct RAGA and
                                its respective parsing table
  }
```

Figure 14. The GCT construction program

- Based on the construction algorithm, the method construct_aga( ) completes the construction of the AGA automaton by creating the parsing actions respective to Q . Such construction is achieved using the attached index of the individual states as indicator for the positions of their corresponding grammar symbols within their respective productions. In addition, construct_aga( ) produces as an output the transition graph respective to AGA.
- construct_raga ( ) is a method to construct RAGA automaton according to the subset construction algorithm. It produces as an output the parsing table respective to RAGA.

The outlined implementation of GCT and its use as a as scanner/parser/translator have been experimented by the students from three sections of a compiler construction course over three semesters. Several grammars, annotated by respective translation schemes have been used. Compared to previous semesters, the number of students completed their assignments has been increased by 30%. The following is an example of such experiments.

Example 9. Considering the grammar G = (Σ, N, P, S), where: (d, b, c)∈ Σ, (S, A, B, X) ∈ N and P = { $S \rightarrow Ad$, $A \rightarrow BX$, $B \rightarrow b$, $X \rightarrow c$ }. It's parsing using GCT proceeds as follows

The grammar G is inputted using an interaction context to enter Σ, N and P as shown in Figures 15 and 16 respectively.

The output produced by the method construct_aga ( ) consists of a transition graph of AGA respective to the inputted grammar as shown in Figure 17.

The output produced by the method construct_raga ( ) consists of the set of RAGA states and its representative parsing and translation tables as shown in Figure 18, Figure 19 and Figure 20 respectively.

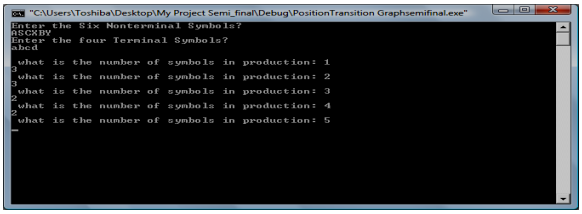

Figure 15. The GCT interaction context to input grammar symbols

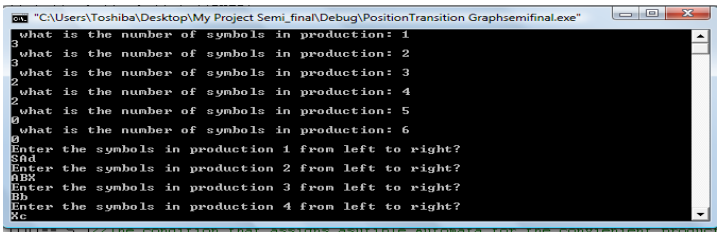

Figure 16. The GCT interaction context to input grammar productions

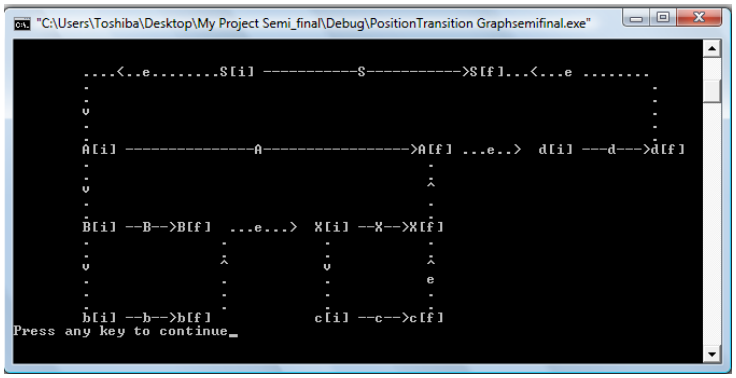

Figure 17. A transition graph of AGA respective to the grammar (3.5) as produced by GCT

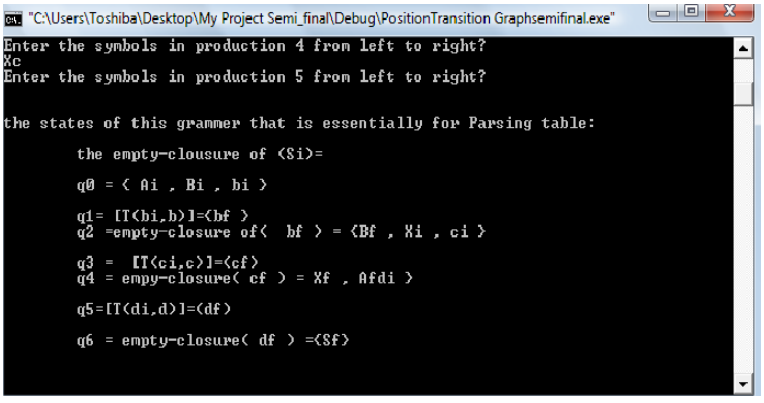

Figure 18. The set of RAGA states as constructed by GCT

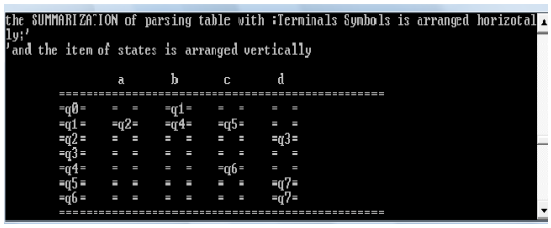

Figure 19. The transition table of RAGA as constructed by GCT

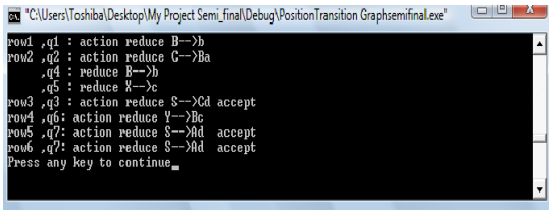

Figure 20. The reduction-translation table of RAGA as constructed by GCT

5. Discussion

The proposed tool has been analyzed and experimented against the set objectives and in two directions. The first one considers GCT as new compiler construction approach. The second direction considers GCT as a teaching tool.

As a compiler construction approach, GCT has demonstrated the following properties: Generality and soundness of its construction approach. GCT constructs a bottom up parsing automaton (RAGA) that simulates shift–reduce one, but with reduced stack activities. In addition, regular definitions are recognizable by RAGA. Semantic action can be inserted within grammar productions. As result, GCT can easily provide a syntax directed translator.

RAGA features the finite automata (Aho et al., 2007) in two aspects. The first one is the number of its states transitions. The second aspect is the complexity of its subset construction algorithm. However, these aspects are with additional overhead to handle recursion. Compared to LR parsers (Aho et al., 2007), RAGA has less number of states and less parser size. This is due to the fact that it performs shift-reduce operations with stack activities reduced to recursion handling. In addition, RAGA has no goto-transition on non terminals. Hence, their respective computational overhead and the construction of the goto-part of the parsing table are not included in the subset construction algorithm for RAGA.

RAGA is based on position parsing automata (PPA) as proposed in (Jabri, 2009) In addition, it handles recursion in a similar way as the one in recursion incorporated generalized parsers (RI) (Galves et al., 2006; Johnstone & Scott, 2007; Scott & Johnstone, 2005). However, compared to such approaches RAGA is distinguished by its annotation with translation schemes and the following properties:

(1) Compared to PPA, RAGA is considered as an extension since it handles left, embedded and direct right recursion, while PPA handles the embedded ones only. Furthermore, their handling approach is different. PPA handles recursion by state and transition instantiation while RAGA handles recursion by reduced stack activities. Finally, the reduced version of PPA (Jabri, 2012) has non deterministic behavior while RAGA has deterministic one. Such determinism is achieved by using look ahead symbols, upon the reduction actions of RAGA.

(2) Compared to RI, RAGA handles left and right recursion while RI handles embedded ones and follows different construction and parsing approach. In addition to stacks needed for recursion handling, in RI two parsing stacks are used while in RAGA no parsing stacks are used.

As a teaching tool, GCT has been experimented and compared against existing ones. It has been shown that GCT is distinguished by the following:

(1) GCT is based on concepts from finite and LR automata as well as syntax directed translation. Hence, it obeys strictly to theory and facilitates a smooth transition to practice.

(2) Its use within the proposed teaching framework is consistent with the compiler construction phases and their respective order. In contrast, the existing tools are based on a particular approach, for examples:

- The suggested tool in (Mallozzi, 2005) is based on recursive descent approach.
- The suggested one in (Demaille & Levillain, 2008) is an extension of the well-known Bison tool (Corbett et al., 2003) and based on LALR grammar (Aho et al., 2007).
- A third tool suggested in (Morell & Middleton, 2003) follows recursive ascent approach that simulates tabular-less bottom up parsers.

In contrast to such tools and a similar one (Pinaki et al., 2011), GCT simulates a generic bottom-parser and maintains its inherent tables but with reduced size and overhead. Furthermore, compared to Lex and Yacc (Mason & Brown, 1990) GCCT is a pedagogical tool. Its input features regular and context-free grammars. Its construction approach is a direct mapping to an automaton. It produces as an output the transition graph and the parsing table respective to the constructed automaton, associated with textual information. Furthermore, it enables more experiments to be conducted by students within the framework of the compiler construction course. For example, students were asked to construct a parser using the same grammar (Example 9) and following the proposed approach as well as the SLR approach (Aho et al., 2007). It has been shown that the parser constructed according to our approach requires less construction time and less states (8) as compared to the one constructed by SLR (15 states). Furthermore, the ease of use of GCT as automated tool is demonstrated by the fact that it use reduced to an interaction context consisting of the textual representation of the context-free grammars without any modification. A graphical, tabular and textual information representing GCT are then displayed as an output.

6. Conclusion

In this paper, we have proposed and implemented a tool for teaching compilers. The tool is based on a new compiler construction approach that is characterized by its soundness, generality and efficiency. Having as an input, a generic grammar (Regular definitions or context-free grams), annotated by translation schemes, the proposed tool respectively reacts either as a scanner, a parser, or as a syntax directed translator. Such reaction proceeds as follows. First, it constructs a nondeterministic bottom–up automaton (AGA). The states and the transitions of AGA are defined based on concepts from the LR (0) items and the finite deterministic automata. Second, AGA is transformed into a reduced one (RAGA) in efficient way. Such automaton simulates the parsing behavior of the shift-reduce automata. Finally, the tool produces as an output the transition graph and the parsing / translation tables respective to RAGA. The experiments and the analysis of the proposed approach for have shown its superiority over similar ones, either in terms of its generality or in terms of its performance and ease of use. In addition, the tool has proved its practical application within a compiler teaching framework and in a way that tightly couples theory and practice. Since such experiments have emphasized the syntax analysis phase of compilers, further experiments, as a future work, are to be conducted to demonstrate the capability of the proposed tool in the subsequent compilation phases such as code generation.

References

Aho, A. V. (2008). Teaching the Compilers Course. *ACM SIGCSE Bulletin, 40*(4), 6-8. http://dx.doi.org/10.1145/1473195

Aho, A. V., Sethi, M. L. R., & Ullman, J. D. (2007). *Compilers Principles, Techniques, & Tools*. Addison Wesely.

Ayock, J., Horspool, R. N., Janousek, J., & Melichar, B. (2001). Even faster generalized LR parsing. *Acta Infomtica, 37*(9), 631-651.

Corbett, R., Stallman, R., & Hilfinger, P. (2003). *Bison: GNU LALR(1) and GLR parser generator*. Retrieved from http//www.gnu.org/software/bison/bison.html

Demaille, A., Levillain, R., & Perrot, B. (2008). A set of tools to teach compiler construction, *Proceedings of the 13th annual conference on innovation and technology in computer science* (pp. 68-72). Madrid, Spain. http://dx.doi.org/10.1145/1597849/1384291

Galves, J. F., Schmitz, S., & Faree, J., (2006). Shift-resolve parsing: Simple unbounded look ahead, linear time. *Lecture Notes in Computer Science, 4094*, 253-264.

Jabri, R. S. (2009). Pattern Matching based on regular tree grammars. *International Journal of Electrical and Computer Engineering, 4*(1), 25-34.

Jabri, R. S. (2012). A generic parser for strings and trees. *Computer Science and Information Systems, 9*(1), 380-409. http://dx.doi.org/10.2298/CSIS101109004J

Johnstone, A., & Scott, E. (2007). Automatic recursion engineering of reduction incorporated parsers. *Science of Computer Programming, 68*, 95-110. http://dx.doi.org/10.1016/j.scico.2006.04.011

Mallozzi, J. S, (2005). Thoughts on and Tools for Teaching Compiler Design. *Journal of Computing Sciences in Colleges, 21*(2), 177-184.

Mason, T., & Brown, D. (1990). Lex & Yacc (p. 216). *O'Reilly &Associates, Inc*. CA, USA.

Morell, L., & Middleton, D. (2003). Recursive-ascent parsing. *Journal of Computing Sciences in Colleges, 18*(6), 186-201.

Panaki, C., Shweta T., Saxena, P. C., & Katti, C. P. (2011). Teaching purpose compiler: an exercise and its feedback. *ACM Inrods, 2*(2), 47-51. http://dx.doi.org/10.1145/1963533.1963549

Scott E., & Johnstone, A. (2005). Generalized bottom- up parsers with reduced stack activity. *The Computer Journal, 48*(5), 565-587. http://dx.doi.org/10.1093/comjnl/bxh102

Waite, W. M. (2006). The compiler course in today's curriculum: three strategies. *Proceedings of the 37th SIGCSE technical symposium on Computer science education* (pp. 87-91) Houston, Texas, USA. http://dx.doi.org/10.1145/1121341.1121371