International Journal For Technological Research In Engineering
Volume 2, Issue 2, October-2014 *ISSN (Online): 2347 - 4718*

IMPLEMENTING LEFTMOST PARSING TECHNIQUES FOR FINITE STATE AUTOMATA

Karuna Hazrati[1], Priyanka Mittal[2], Komal Chauhan[3]

ABSTRACT: Implementation of a new compiler usually requires making frequent adjustments to grammar definitions. An incremental technique for updating the parser tables after a minor change to the grammar could potentially save much computational effort. More importantly, debugging a grammar is made easier if the grammar is re-checked for correctness after each small change to the grammar. The basic design philosophy of an incremental parser generator, and incremental algorithms for LR(0), SLR(1) and LALR(1) parser generation are discussed in this paper. Some of these algorithms have been incorporated into an implementation of an incremental LALR(1) parser generator.
Keywords: Compilers, Compilers Tools, Program Development Environments, LR Parsers, Grammar Debugging.

I. INTRODUCTION

A compiler represents a major software development effort. Simple, non-optimizing compilers for relatively small languages like Pascal may consist of several thousand lines of source code. Production compilers for large languages like Ada or that performs sophisticated optimization may consist of hundreds of thousands of lines of code. A conventional compiler is normally organized into phases. A simple compiler would have phases for lexical analysis, syntactic analysis, semantic analysis and code generation. Many tools exist to help the compiler writer develop the lexical and semantic analysis phases. There are tools based on attribute grammar formalisms which can be used to construct semantic analysis and code generation phases. Automatic techniques for developing a code generator phase from a description of the target architecture for the compiler are a subject of current research. There are two main approaches to parsing top-down parsing and bottom-up parsing. Top-down parsing is usually implemented by a method known as recursive descent, which uses a collection of mutually recursive procedures. This method has been successfully used in many compilers (for example, in the original compiler for Pascal). But recursive descent is criticized for various reasons. Here are four. First, the class of grammars it can be used with is smaller than for bottom-up methods that accept LALR (1) and LR (1) grammars. Second, recursion is not always implemented efficiently and therefore parsing speed may be adversely affected. Third, good recovery from syntactic errors is not easy in a recursive descent compiler. Fourth, semantic analysis and code generation actions are often included inside the recursive descent procedures, and that tends to spoil the modularity of the compiler. Since we were forced to choose one approach or the other, we picked the bottom-up approach {mostly because it is capable of being used with a larger class of grammars. When using any existing tool for creating a syntactic analyzer, the user must create a grammar for the language to be compiled. The grammar is processed by the tool, which we will call a parser generator, and it outputs a parser suitable for inclusion in the compiler (or, equivalently, it outputs tables that drive a standard parsing procedure). The form of the grammar is constrained by the class of grammars that the parser generator can accept and by the need to associate semantic actions with the production rules. Parser generators exist for various classes of grammars, including: operator precedence, LL(1), SLR(1), LALR (1), and LR (1). The compiler writer can rarely use the grammar provided as part of the formal language description. Published grammars are usually designed for people to read and not for the implementer to use. The implementer is therefore likely to and that the grammar does not belong to the class of grammars accepted by the parser generator. Transformations on the grammar need to be performed, while being careful not to change the language that it accepts. Even after the grammar has been changed to satisfy the requirements of the parser generator, further changes are likely to be required when the implementer attempts to attach semantic actions to the production rules. The term grammar debugging is often applied to the activity of transforming a grammar in this way.

II. DESIGNING AN INCREMENTAL PARSER GENERATOR

There are two main design issues which must be decided before we can discuss the algorithms needed to implement an IPG. First, what quantum of change to the grammar should be input by the tool before the grammar is re-checked? At one extreme, we can recheck after the user adds or deletes single characters to or from the grammar specifications. At the other extreme, we can wait until the user has typed all the desired changes before re-checking. Second, what grammar class should the tool accept? By choosing a small class, such as the class of regular grammars, we would make the implementation of the tool easy but the tool would not be useful to compiler writers. By choosing a large class, such as LR (1), we might make the update algorithms too complicated to implement easily. Complicated update algorithms may also be too slow to be able to provide the user with sufficiently fast responses. We decided that the unit of change should be a single production rule. After each addition of a new rule and after each deletion of a rule, the grammar is re checked for acceptability. A change to a rule is considered as a deletion of the original rule followed by an

insertion of the corrected rule. If the unit of change were to be made any smaller, we would be faced with the problem of handling incomplete production rules. Larger units of change would simply delay reporting possible problems to the user. But, if we permit the user to add or delete rules in any order, we must be prepared to (temporarily) accept incomplete grammars. One symptom of an incomplete grammar is that some production rules and non-terminal symbols may be inaccessible.
For example, if we have entered only the following two rules:
S → begin statement list end
assignment → variable := expression
then the second rule cannot be used in any derivation that starts from (what is apparently) the goal symbol S. A second symptom may be that some non terminals cannot generate sentential forms that consist only of terminal symbols. Such non-terminals are called useless. For example, we might add just the rule: statement list → statement list ; statement to the grammar, above. Then, even if we temporarily consider statement to be a terminal symbol, the grammar is incapable of deriving sentential forms that are free from the non-terminal symbol statement list. If we wish to allow rules to be entered in any order and to check the grammar after each addition, it is clear that a relaxed form of checking must be employed. Ignoring the rules for inaccessible or useless non-terminals would be an unsatisfactory approach. The user could choose to enter rules in such an order that almost the entire grammar may remain inaccessible or useless until the last rule is defined. Current generators for bottom-up parsers usually accept one of the LR grammar classes. Wechose to implement the LALR(1) class of grammars because of its power {it contains the LL(1),LR(0) and SLR(1) classes. While it is a smaller class than the LR(1) class, the generated parserusually has far fewer states and therefore requires much less memory for its implementation. Italso appears to be the case that LR(1) parsing tables require much more work to update after asmall change to the grammar. Conversely, parsers for the LR(0) and SLR(1) classes of grammars require less computational effort to create than does LALR(1). A parser generator for either of4these smaller grammar classes may be more suitable in situations where the computational costis important.

III. HANDLING INCOMPLETE GRAMMARS

It should be possible to analyze grammars which have not been completely specified. Indeed, the start symbol of the grammar may be one aspect of the grammar that remains undefined until late in the specification process. It is therefore appropriate to add a goal symbol of our own, ^ S, and to invent extra rules of the form
S→ $NN $N
for each non-terminal symbol N in the partial grammar. The $N symbol is a delimiter symbol of type N. It is an invented symbol that represents both a beginning of input and an end of input delimiter. Its purpose is to provide a unique context within which N can appear if it were to be used as a goal symbol. If unique delimiter symbols are not provided, our support for multiple goal symbols can cause LR conflicts in the parser construction process. But addition of these extra rules requires us to know which symbols are non-terminals. A reasonable strategy would be to assume that every symbol encountered in the grammar so far is a terminal symbol, unless the symbol appears on the left-hand side of a rule.
While a grammar is under development, it is natural for some parts of the grammar to be incomplete [4]. Therefore, we should not complain about useless productions until the user claims to have completed the grammar. For example, the user may have entered the rule
L →L , x
and no others with L as a left-hand side. It is clear that L is a non-terminal symbol, but it is also useless. We can circumvent this difficulty by assuming that while the grammar is in an incomplete state, it is a grammar for sentential forms {not a grammar for sentences in the language. For example, with the rule for L, above, the language includes the sentential forms
$L L $L $L L, x $L $L L, x, x $L : : : etc.
where $L is the automatically generated context delimiter symbol. If the user makes an explicit request to check the grammar for completeness or requests that the LALR (1) parse tables be output, an algorithm to check the grammar can be applied. A suitable algorithm is given in [4]. When rules are missing from a grammar, it is impossible to know for certain which symbols is null (can produce the empty sentence in some derivation sequence). A symbol X may appear to be non-nullable, but the addition of the rule 5 X ! _ would immediately change the status of X. It seems best to assume that symbols are non-nullable until a derivation to the empty string becomes possible using rules in the grammar. This would also avoid generation of premature error messages about ambiguities in the grammar.

IV. INCREMENTAL LR PARSER CONSTRUCTION ALGORITHMS

When the grammar class is restricted to one of the LR (0), SLR (1) or LALR(1) classes, the computation of parsing actions for a particular grammar can be separated into two stages. The first stage is the construction of the LR (0) sets of items for the grammar. The second stage computes the look-ahead sets that are associated with the LR (0) items. Computing these look-ahead sets is trivial for a LR(0) parser generator and is the most expensive for
LALR (1).

A. Terminology

A context-free grammar G is defined as a four-tuple G = < VT; VN; S; P >,
where VT is the set of terminal symbols, VN is the set of non-terminal symbols, S is a designated start symbol, and P isthe set of production rules. As explained above, the grammar that our algorithms process is not the same grammar as is entered by the user. It has been augmented by additional productions and additional symbols. We will use the name G to refer to this augmented grammar. We will use standard conventions when discussing grammar formalisms. The symbol _ represents an empty string of grammar

symbols. If R is a relation, then R_ represents the reflexive, transitive closure of R.

B. Incremental Update of the LR (0) Sets of Items

Construction of the LR (0) sets of items is covered in texts on compiler construction [2, 3, 8, 20]. A reader who is unfamiliar with the methods and concepts of LR (0) parser construction will and the formal definitions and notation difficult to read. We therefore begin with an informal introduction.

4.3.1 Informal Introduction to LR (0) Concepts

The LR(0) construction method is based on the concept of an item. An item is simply a production rule with a marker (frequently called a dot) inserted anywhere in the right-hand side of the rule. The marker indicates how many symbols of the right-hand side have been recognized at some point in a parse. That is, all symbols to the left of the marker have been recognized and symbols to its right have not yet been used. An LR parser is implemented as a finite state automaton, where each of its states corresponds to some set of items. Each such set of items represents parsing possibilities {showing which rules are eligible to be matched if appropriate symbols are read by the parser. The sets of items which give rise to the LR recognizer are constructed by a process of closure. There are some initial items, formed by inserting the marker at the beginnings of the right and sides of all goal rules in the grammar. There are two closure operations which we will call Item Closure and State Closure. Given a set of items, the Item Closure operation adds extra items to the set; these extra items are usually called completion items. If any item in the set has the marker immediately to the left of a non-terminal symbol N, then this item generates completion items. These completion items are formed by taking each rule which has N on its left-hand side and placing the marker at the beginning of the right-hand side. These completion items may, themselves, require the addition of more completion items {which is why a closure process is required. The Kernel operation is applied to a grammar symbol and a set of items that has been completed (using Item Closure), to yield a new set of items. This new set of items plus completion items added by the Item Closure operation corresponds to the next state in the LR recognizer as reached by a transition on the grammar symbol. The items for a new state before Item Closure is applied are called the kernel items of the new states [7]. The LR(0) collection of sets of items is formed by a process of applying Item Closure to the initial items, and applying State Closure to this set to yield the remaining sets of items. State Closure can be implemented using an iterative algorithm based on a work list. The algorithm is sketched out below. In this algorithm, Start is the set of items for the start state of the recognizer;
W is the work list. On termination, K is a set that contains all the recognizer's states.

```
State Closure:
Start:= f \the initial items" g;
W:= f Start g;
K:= f g;
while W is not empty doremove state S from W;
S:= Item  Closure(S);
K: = K [f S g;
for each grammar symbol X do compute the kernel items, S0
= Kernel(S,X);if S0 is a new state then
W:= W[ f S0g;
```

Since states are uniquely determined by their sets of kernel items, the test to see if G is a new state does not require that the Item Closure function be applied to G. The operation of Item Closure can also be implemented by an iterative algorithm based on a work list. In this case, the work list holds individual items. The algorithm has a similar structure to the one given above and therefore we omit giving its details. Construction of the LR recognizer will generate three sets of items (amongst about 20 other sets)that are related as shown in Figure 1. But after we add the extra rule A→ B, the corresponding states in the new LR automaton have sets of items as shown in
Figure 2.
States 3 and 4 in the first LR recognizer have split to become pairs of states in the second recognizer. Unfortunately, there does not appear to be a simple test for determining when such state splitting is required {or, indeed, for determining what extra elements must be added to aset of items.

C. SLR (1) Look ahead Sets

The SLR (1) method uses a slightly more sophisticated definition for the look-ahead sets. The sets are computed using a function called FOLLOW. But computation of FOLLOW is facilitated if the set of all null able symbols, NULL, and a function called START are also computed.
The set of null able symbols is formally dened as

NULL = f X j X) g

Using NULL, we can determine the null ability of any sentential form.
The set of nullable symbols after a rule addition is closely related to that set beforehand.
Continuing with the convention that primes refer to the grammar G0, the relationship is:
NULL0 = (NULL [fBjB)
Lg if 8 i (1 _ i _ n) :Ri 2 NULL
NULL otherwise where the rule L!R1 : : :Rn is added to the grammar G to create G0. A simple iterative algorithm based on a work list of non-terminal symbols that need to be re-checked for null ability can be used to obtain NULL0 from NULL efficiently.
The START function yields the set of starter symbols for a grammar symbol. It is formally defined as

START(X) = f Y j X _) Y _ g

The FOLLOW function yields the set of symbols that may legally follow a grammar symbol in a sentential form. It is defined as

FOLLOW(X) = f Y j S _XY _ g

Methods for computing the START and FOLLOW functions can be found in most bookson compiler construction. The approach given here is based on [10]. The START function

can be found by constructing the Immediate Starters relation for the grammar and then forming the transitive closure of that relation. If we choose to represent that relation by a matrix IS, where IS[X; Y] = true means that symbol X has Y as one of its immediate starters, and then the matrix is defined as follows. An entry IS[X; Y] has the value true i_ the grammar contains a rule
X!Z1Z2 : : :ZkY _ such that 8i (1 _ i _ k) : Zi 2 NULL. All other entries have the value false.
The transitive closure of IS is written as IS_ and is defined by IS_[X;Z] = IS[X;Z] or (9Y1; Y2; :::Yn : IS[X; Y1]&IS[Y1; Y2]&:::&IS[Yn; Z])
There are well-known algorithms for computing the transitive closure of a relation [1, 22]. Efficient incremental algorithms for transitive closure also exist [23] and would be particularly suitable for use here. If we form the transitive closure IS_ using any of the standard methods, we have
START(X) = f Y j IS_[X; Y] = true and Y 2 VT g
Similarly, the FOLLOW function can be computed by first constructing an Immediate Followers relation and then taking its transitive closure. We define the IF matrix so that IF[X; Y] has the value true if either the grammar contains a rule Z!_Y Z1Z2 : : :ZkW_ and 8i (1 _ i _k) : Zi 2 NULL and X 2 START(W), or the grammar contains a rule X!_Y Z1Z2 : : :Zk and 8i (1 _ i _ k) : Zi 2 NULL.
The transitive closure of IF can be computed, again using standard techniques. We can then obtain FOLLOW from IF_ as follows.
FOLLOW(X) = f Y j IF_[X; Y] = true and Y 2 VT g
The formulation of START and FOLLOW in terms of IS_ and IF_ demonstrates the monotonic nature of the problem. When a new rule L→R1R2 : : :Rn is added to the grammar, we must change entries from false to true in the IS and IF matrices. Changes in the reverse direction cannot occur. For example, if n > 0, the entry IS[L;R1] would be set to true. In turn, this implies changing entries from false to true in transitive closure, IS_, and thus we would have computed START0. Having computed START0, we can update IF. For example, when we add the rule
L→ R1R2 : : :Rn
and where n > 1, the entries IF[R1; x] for x 2 START0(R2) would be set to true.
An algorithm which works well is to update IS and IF as suggested, and then use an iterative, worklist-based, approach for updating the transitive closures.
Once we have computed the IF_ relation (and thus the FOLLOW function), we can determine the look ahead sets. According to the SLR (1) approach, we use:
LA (q; [A!_ _]) = FOLLOW(A)
And, if we define LA for non-reduce items in the same way as for LR(0) parsers, conflict checkingcan also be performed in the same way.

D. LALR (1) Lookahead Sets

Several methods for computing LALR (1) look ahead sets have been published. Of these, an iterative algorithm due to Aho and Ull man, described in [3, algorithm 4.13] and [8, figure 6.25], appears to be best suited for conversion to use in an incremental setting. We give a modified version of this algorithm below. If the existence of some item, I1, in some state implies the existence of another item, I2, either in the same state (through the addition of completion items) or in some other state (through the state completion process), then the look ahead function applied to I2 yields a set which may contain symbols determined by I1. This is called spontaneous generation of look ahead symbols. In addition, it is possible that the set of look ahead symbols for I2 must include the entire set of look ahead symbols for I1. In this case, the symbols are said to propagate from I1 to I2. The rules for spontaneous generation of symbols and propagation of symbols in the two possible settings are as follows:

Case 1:
Completion Items
Suppose that state q contains an item I1 where the marker appears to the left of a non-terminal symbol. That is, I1 has the form [A!_ _ X_]. The state must also contain one or more completion items with the form [X! _]. Let I2 be one such item. The symbols which can follow the right hand-side of I2 must include the symbols which follow X in item I1, and the symbols which can follow X in that item must include FIRST (_), where FIRST is defined below. In other words, a 2 FIRST (_) implies a 2 LA(q; I2). Inthe terminology of [3], the symbol is spontaneously generated (by I1) and must appearin the lookahead set of I2 [14].

Case 2:
Kernel Items
Suppose that state q1 contains an item I1 with the form [A!_ _ X_]. There must necessarily be another state q2 reached by a transition on symbol X from q1, where q2 contains a kernel item with the form [A!_X]. In such a case, if a 2 LA(q1; I1) then a 2 LA(q2; I2). This is another example of propagation, where symbol ‘a’ propagates from I1 to I2.
The FIRST function is a simple extension of START to the domain of sentential forms.
FIRST(_) = f x j x_; x 2 VT g
An alternative de_nition which shows how to derive FIRST from START is
FIRST(X1 : : :Xk) = (START(X1) [FIRST(X2 : : :Xk) if X1 2 NULL
START(X1) otherwise FIRST(_) = ;
A simple algorithm to determine the look ahead sets can start by initializing all look ahead sets to empty. Then it can make repeated passes over all items in all states adding spontaneously generated symbols and propagated symbols to the sets. This iterative procedure can halt when a pass fails to add any new symbols to any set. (A faster version, and the method used in ilalr, would use a work list so that only items whose look ahead sets have changed participate in the next pass. Entries in the work list consist of < state; item > pairs.)

V. WORST CASE COMPLEXITY

A natural question to ask is how long can it take to update the parser tables after the addition or deletion of one production rule using the algorithms described in this paper?" We will demonstrate that the problem has exponential worst-case time complexity and therefore no algorithm that is efficient in the worst-case can exist. The demonstration is based on the following grammar1 which contains 2n + 2 productions.

S0 → a S0 j b S0
S0 → c S1
S1 → a S2 j b S2
S2 → a S3 j b S3
S3 → a S4 j b S4
: : :
Sn → a Sn j b Sn
Sn → d

The LR(0) construction algorithm applied to this grammar generates a recognizer with 4n + 6states.
Suppose, now, that the production rule S0 → a S1is added to the grammar. The extra rule causes the number of states in the LR(0) recognizer to be increased to 2n + 4n + 6 states.
Therefore, the unfortunate implication is that any algorithm for incrementally updating the LR parse tables must be prepared to create an exponential number of states when processing a single rule addition. Therefore both the time and space cost are exponential in the size of the grammar. (The amended grammar also illustrates that even a non-incremental LR (0) parser generator has exponential worst-case time and space complexity.) The example grammar is constructed in such a way that it illustrates another interesting property. Suppose that we add yet one more production

S0 → b S1

Surprisingly, this addition causes the number of states in the LR(0) recognizer to be reduced to6n +7. Now, consider what happens if this last rule is deleted. We would observe the number of LR (0) states to increase from 6n+7 to 2n+4n+6. In other words, this example demonstrates that an incremental algorithm for rule deletion must also have exponential time and space complexity.
The grammar was constructed by Alan Demers.
[18].

GRAMMAR	PL/0	Pascal	XPL	C	Oberon	Ada
Total CPU time	0.3	2.0	2.7	10.9	2.1	23.6
Average time per rule	0.005	0.009	0.02	0.04	0.008	0.05
Maximum time for one rule	0.02	0.05	0.16	3.72	0.04	0.8
Total CPU time used by yacc	0.3	1.2	1.0	3.8	1.4	11.5

VI. CONCLUSION

Practical algorithms for incremental analysis of grammars and for incremental generation of LR(0), SLR(1) and LALR(1) recognizers have been presented. Compiler construction tools based on these algorithms would help compiler writers develop suitable grammars and might also permit the incremental construction of compilers. Although the worst-case execution times of the algorithms are poor, practical experience shows that they work well. A possibility that might reduce expected execution time requirements even further would be to incorporate an efficient incremental transitive closure algorithm [23]. There is, of course, no hope for such an algorithm improving the worst-case time complexity of the problem. The transitive closure algorithms are efficient only for the case when a single edge is added or deleted from a graph. As section 6 in this paper demonstrates, the number of edges that must be added or deleted from the state graph of the LR(0) recognizer can be exponential in the size of the grammar. It would be nice to give a proper comparison between the speeds of our implementation and Fischer's implementation [9], but only a meaningless comparison seems possible. Using our implementation, the time needed to process the 109 productions of the XPL grammar is 2.7 CPU seconds on a SUN SparcStation I workstation, for an average of 0.024 CPU seconds per production. Fischer's implementation applied to the same grammar used an average of 1.8 CPU seconds per production on a Siemens 7.748 computer (which he describes as being nearly half as fast as an IBM 370/158).

REFERENCES

[1] A.V. Aho, J. Hopcroft and J.D. Ullman.The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974. 22
[2] Aho, A.V., Johnson, S.C. LR Parsing. ACM Computing Surveys, vol. 6, no. 2, pp. 99-124, 1974.
[3] Aho, A.V., Sethi, R., Ullman, J.D. Compilers: Principles, Techniques and Tools. Addison-Wesley, Reading, MA (1986).
[4] Aho, A.V., Ullman, J.D. The Theory of Parsing, Translation and Compiling; vol. 1, Parsing. Prentice Hall, Englewood Cli_s, NJ (1972).
[5] Dencker, P., D□urre, K., Heuft, J. Optimization of Parser Tables for Portable Compilers. ACM Trans. on Prog. Lang. and Sys., 6,4, 546-572 (1984).
[6] DeRemer, F., Pennello, T. E_cient Computation of LALR (1) Look-Ahead Sets. ACM Trans. onProg. Lang. and Sys., 4,4, 615-649 (1982).
[7] Ehre, R. Die Generierungeines Multiple-Entry Parsers und eininkrementeller LALR (1)- Parser generator. Diploma thesis, Department of Mathematics and Computer Science, University of Saarbr□ucken, Federal Republic of Germany (1986).
[8] Fischer, C.N., LeBlanc Jr., R.J. Crafting a Compiler. Benjamin/Cummings, Menlo Park, CA (1988).
[9] Fischer, G. Incremental LR(1) Parser Construction as an Aid to Syntactical Extensibility. PhD Dissertation, Tech. Report 102, Department of Computer Science, University of Dortmund, Federal Republic of Germany (1980).
[10] Gri_ths, M. LL(1) Grammars and Analysers in Compiler Construction: An Advanced Course { second edition, (Lecture Notes in Computer Science vol. 21), F.L. Bauer and J. Eickel, Eds., Springer-Verlag, Berlin (1976).
[11] Heering, J., Klint, P., Rekers, J. Incremental Generation of Parsers. Report CS-R8822, Centre for Mathematics and Computer Science, Amsterdam (1988).

[12] Heering, J., Klint, P., Rekers, J. Incremental Generation of Parsers. Proceedings of Sigplan '89 Conference on Programming Language Design and Implementation. ACM Sigplan Notices 24, 7, 179-191 (1989).
[13] Heindel, L.E., Roberto, J.T. LANGPAK {An Interactive Language Design System. American Elsevier, New York, NY (1979).
[14] Horspool, R.N., Levy, M.R. MkScan{An Interactive Scanner Generator. Software {Pract. &Exper.17, 6, 369-378 (1987).
[15] Johnson, S.C. YACC { Yet Another Compiler-Compiler. Bell Laboratories, Murray Hill, NJ, Rep. CSTR 32 (1974).
[16] Kam, J.B., Ullman, J.D. Monotone Data Flow Analysis Frameworks. ActaInformatica 7, 3, 305-318 (1977).
[17] Lesk, M.E., Schmidt, E. LEX { A Lexical Analyzer Generator. Bell Laboratories, Murray Hill, NJ, Rep. CSTR 39 (1975). 23
[18] Pager, D. A Practical General Method for Constructing LR(k) Parsers. ActaInformatica 9249-268 (1977).
[19]] Spector, D. E_cient Full LR(1) Parser Generation. ACM Sigplan Notices 23, 12 143-150 (1988).
[20] Tremblay, J.-P., Sorenson, P.G. The Theory and Practice of Compiler Writing. McGraw-Hill, New York, NY (1985).
[21] Vidart, J. Extensions Syntactiquesdansune Context eLL(1). Doctoral dissertation, University of Grenoble (1974).
[22] Warshall, S. A Theorem on Boolean Matrices. J. ACM 9,1, 11-12 (1962).
[23] Yellin, D. A Dynamic Transitive Closure Algorithm. IBM T.J. Watson Research Center, Yorktown Heights, NY, report RC 13535 (1988).