Locating Free Positions in LR(*k*) Grammars*

SHIH-TING OUYANG, PEI-CHI WU+ AND FENG-JIAN WANG++
Sunplus Technology Co., Ltd.
Hsinchu, 300 Taiwan
E-mail: tim_ouyang@sunplus.com.tw
+*Department of Computer Science and Information Engineering*
National Penghu Institute of Technology
Penghu, 880 Taiwan
E-mail: pcwu@npit.edu.tw
++*Department of Computer Science and Information Engineering*
National Chiao Tung University
Hsinchu, 300 Taiwan
E-mail: fjwang@csie.nctu.edu.tw

LR(*k*) is the most general category of linear-time parsing. Before a symbol is recognized in LR parsing, it is difficult to invoke the semantic action associated with the symbol. Adding semantic actions to an LR(*k*) grammar may result in a non-LR(*k*) grammar. There are two straightforward approaches adopted by practitioners of parser generators. The first approach is to delay all semantic actions until the whole parse tree is constructed. The second is to add semantic actions to the grammar by chance. This paper presents an efficient algorithm for finding positions (called *free positions*) that can freely put semantic actions into an LR(*k*) grammar. The speedups of our method range from 2.23 to 15.50 times for the eight tested grammars.

Keywords: parser generators, LR(*k*) grammars, semantic actions, parse tree, free positions

1. INTRODUCTION

Using parser generators [1] is a fast and easy way to construct parsers or compiler front ends [2]. Most parser generators support a category of context-free grammars. LR(*k*) grammars are the most general category for deterministic linear-time parsing. Parser generators such as YACC [3] and Bison [4] support LALR(1), a subcategory of LR(1). These generators have been widely used in the development of compilers, e.g., the parser of GNU C compiler. Recent advances [5, 6] in parser generators can further speed up the efficiency of generated parsers.

Although LR(*k*) has a larger scope than top-down parsing technique, e.g., LL(1), adding semantic actions in LR(*k*) grammars is more difficult than in LL(1). A *position* is a place between grammar symbols in the production of a grammar. For example, the dots in $\mathbf{T} \rightarrow \cdot\, \mathbf{P} \cdot \mathbf{*} \cdot \mathbf{T} \cdot$, indicate all the positions in the production $\mathbf{T} \rightarrow \mathbf{P} \mathbf{*} \mathbf{T}$. In LL(1) parsing semantic actions can be invoked at any position because the production rules that are matched can always be predicted. Before a symbol is recognized in LR parsing, it is difficult to invoke the semantic action associated with the symbol, because not all positions of an LR(*k*) grammar are free for adding these actions. Adding actions to an LR(*k*)

Received June 30, 2000; revised February 15, 2001; accepted May 18, 2001.
Communicated by Jang-Ping Sheu.
*This research was partly supported by National Science Council, Taiwan, R.O.C., under Contract No. NSC 89-2213-E-346-002.

grammar may make the grammar no longer LR(k). Consider the following example:

rule 1: **E** $\rightarrow$ **E** + **id**
rule 2: **E** $\rightarrow$ **id**

The grammar is LR(1). Adding the following action (*#action*) to the grammar makes the grammar no longer LR(k):

rule 1: **E** $\rightarrow$ *#action* [1, 0] **E** + **id.**

This is because adding a new symbol X in position [1, 0] with an ε-production

$X \rightarrow \varepsilon$ {the code to perform *#action*}

makes the grammar no longer LR(k).

There are two straightforward approaches commonly adopted by practitioners of parser generators. The first is to delay all semantic actions until the whole parse tree is constructed [7]. Only the semantic actions that construct parse trees are invoked by the generated parser. The second approach is to add semantic actions to the grammar by chance [8]. When the resulting grammar violates the grammar class, the designer needs to rewrite part of the grammar or put the semantic actions in other positions. Both approaches have disadvantages. The first approach costs more memory storage and computing time because the whole parse tree must be constructed for the semantic analysis. The second is very tedious and error prone. In addition the resulting grammar rules may be far different from the original rules specified in a language reference manual.

Purdom and Brown [9] proposed a method to distinguish free and forbidden positions in LR(k) grammars. A position is *free* if adding a useless symbol (a symbol defined by ε-production) at the position still results in an LR(k) grammar; otherwise, the position is *forbidden*. Their method uses a *partial state position graph* (*PSPG*) to classify position types.

This paper presents an improvement over Purdom and Brown's method and discusses how to handle grammars containing conflicts. We tested this algorithm on several grammars defined for programming languages. Our results indicate that the speedups of our method range from 2.23 to 15.50 times for the eight tested grammars. Our results also show that about 53.9% to 86.9% of positions are free in these grammars. Our implementation is based on Bison [4], a YACC-compatible parser generator distributed by GNU.

2. BASIC DEFINITIONS AND RELATED WORK

LR(k) Grammars and Free Positions

A *context-free grammar* is a four-tuple $G = (N, T, P, S)$. N and T denote finite sets of nonterminal and terminal symbols respectively, and form the vocabulary $V = N \cup T$, and $N \cap T = \emptyset$. P is a finite set of productions. $S \in N$ is the start symbol, which cannot appear on the righthand side of any production. A production $p \in P$ is denoted by p: $X_0 \rightarrow X_1 ... X_{n_p}$, where $n_p \geq 0$, $X_0 \in N$, and $X_k \in V$, $1 \leq k \leq n_p$. For production p we say that X_0 derives $X_1 ... X_{n_p}$.

A *position* in a grammar G is a pair $[i, j]$, where i is the rule number of production p, $0 \leq j \leq n_p$, and n_p is the number of symbols in the righthand side of p. Position $[i, j]$ represents the location after the j-th symbol in the production i. An *item* of a grammar G is a production with a dot (·) at a position of the righthand side. Positions [0, 0] and $[i, j]$, $j \geq 1$ are *main* positions; $[i, 0]$, $i > 0$ are *derived* positions. Whether an item is main or derived depends on whether the associated position is main or derived. Note that a main item cannot be derived from any item.

S → [0, 0]	**E**						
E → [1, 0]	**E**	[1, 1]	+	[1, 2]	**T**	[1, 3]	
E → [2, 0]	**T**	[2, 1]					
T → [3, 0]	**P**	[3, 1]	*****	[3, 2]	**T**	[3, 3]	
T → [4, 0]	**P**	[4, 1]					
P → [5, 0]	**(**[5, 1]	**E**	[5, 2]**)**	[5, 3]			
P → [6, 0]	**id**	[6, 1]					

Fig. 1. Example grammar.

The example grammar shown in Fig. 1 has positions marked between grammar symbols (boldfaced). For example, **T** → **P** · ***** **T** is an item that corresponds to position [1, 3].

An LR(k) parser scans an input program from left to right with k symbols of lookahead and constructs the rightmost derivation in reverse. It uses a stack of symbols to store the progress of parsing. The parser pushes terminal symbols onto the stack when scanning a program. Whenever the top n elements of the stack can be reduced to a nonterminal symbol X, i.e., X can derive the top n elements of the stack, the top n elements are popped and X is pushed onto the stack. If a program is syntactically correct, the stack contains only the start symbol after the whole program is parsed.

An LR(k) generator constructs parser states and the transitions between these states. A state contains a set of items that represent the set of productions to match. The transitions from state to state, i.e., the parser recognizes a symbol, represent shifts of positions in productions. A state starts with a set of items that shift from items in other states. The other items of the state are added by an *ε-closure* operation [10, Fig. 4.33] on their main items.

A position $[i, j]$ in an LR(k) grammar G is *free* if, after adding a symbol X that derives an empty string at $[i, j]$ of G, the resulting grammar is still LR(k). If adding a symbol X at $[i, j]$ makes the resulting grammar no longer LR(k), then $[i, j]$ is a *forbidden* position.

Fig. 2 presents the complete state transition graph of Fig. 1. The starting item of the *initial state* is the production of the start symbol with the leftmost position.

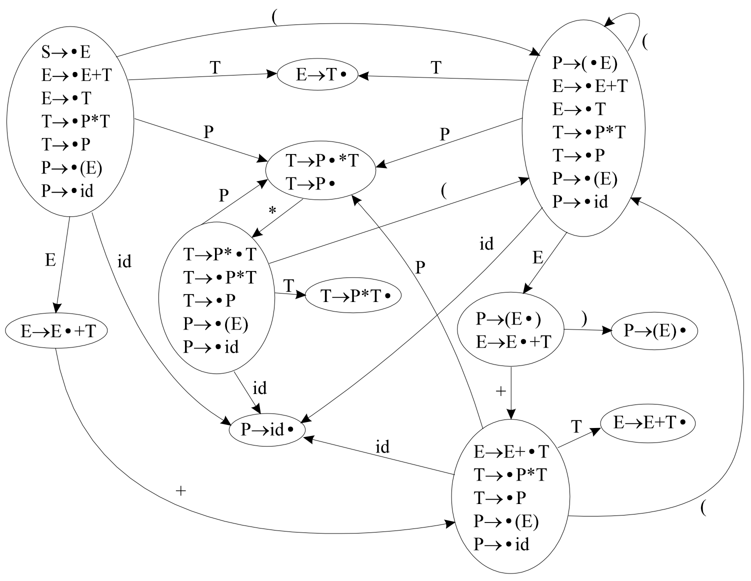


Fig. 2. States and transitions of parser in Fig. 1.

Purdom and Brown's Method

Purdom and Brown [9] proposed a method to distinguish free positions and forbidden positions in LR(k) grammars. In their method a *partial state position graph* (*PSPG*) is constructed to classify the position types. A *partial state* of a parser state contains the following items:

- an item d that shifts a terminal symbol or reduces a production.
- any main item that derives d. Let the set of these main items be m.
- any item that is in *closure*(m) [10, Sec 4.7] and derives d.

There is a PSPG for each partial state where the graph contains an *initial node*, a *final node*, and one intermediate node for each position [i, j] that is associated with an item of the partial state. The graph is a directed graph in which *arcs* represent the order of ε-transitions between items. There are arcs from the initial node to the set of nodes for m and an arc from the node of d to the final node. Each arc of the form $a \rightarrow b$, where a and b are neither an initial node nor final node, indicates that a derives b in one ε-transition. A node in a partial state position graph represents a position in the grammar. Fig. 3 presents one PSPG of the initial state in Fig. 2. The final node is represented as "shift s" or "reduce r" in the following sections.

Let a *dominator* be a node that is included in every path from the initial node to the final node of a PSPG. A dominator is a free position in the grammar [10]. For example, the nodes representing S→•E, E→•T, and P→•id are dominators of the PSPG in Fig. 3. These positions are free. There is an almost linear-time algorithm [11] for finding dominators.

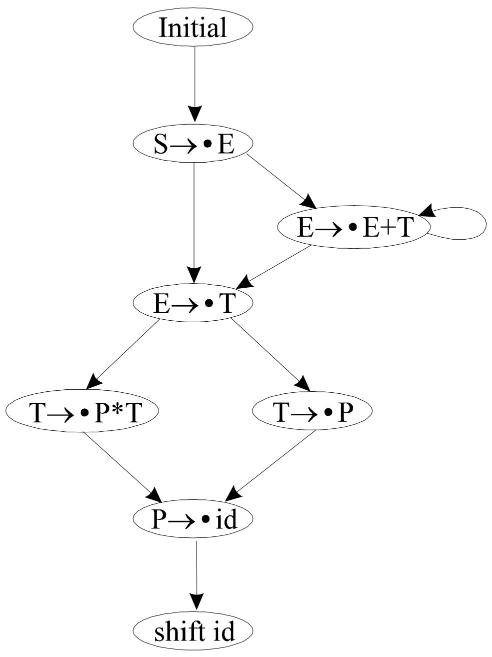


Fig. 3. A partial state position graph (PSPG).

Usually there are several PSPGs for a parser state. A parser state may have more than one final node, and the algorithm generates one PSPG for each pair of initial and final nodes. However, applying a dominator algorithm in these PSPGs separately may result in inconsistent position classification. Consider the following grammar *G'*:

S → A
A → B
A → C
B → t1
C → B t2
C → t3

Fig. 4. The grammar *G'*.

Here t1, t2, and t3 are terminal symbols. Two PSPGs of the state containing the start symbol are shown in Fig. 5. The node number is denoted at each node's upper-right corner. By definition nodes 1, 5, 6, 7, and 8 are dominators and thereby are free positions. Nodes 2, 3, 4 are forbidden positions. Note that nodes 3 and 7 represent the same position in *G'*, while node 7 is free but node 3 is forbidden. When such an inconsistency occurs, the position is forbidden. Only the intersection of dominators of corresponding PSPGs are free positions.

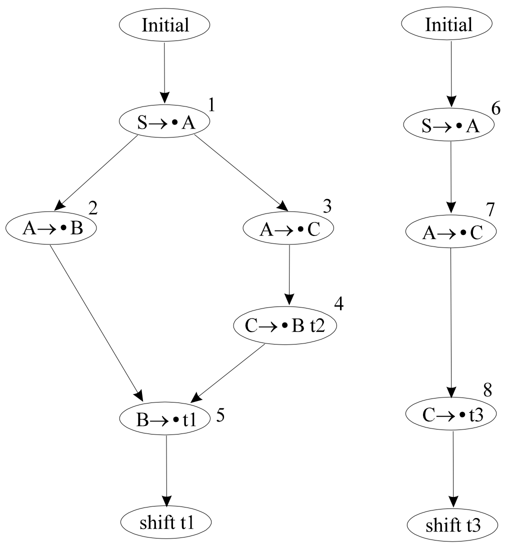


Fig. 5. Two partial state position graphs (PSPGs).

3. AN IMPROVEMENT IN FINDING FREE POSITIONS

One deficiency in Purdom and Brown's method is that it may construct more than one PSPG for a parser state. The dominator finding algorithm is applied for each PSPG. Here we present an improved method, which constructs one graph for each parser state and finds the dominators on each graph. We call our graph the *state position graph* (SPG), which is the "merged" PSPGs. Lemma 1 shows that the set of dominators in an SPG are the intersection of the dominators of the corresponding PSPGs. Thus, the dominators in an SPG are free positions.

Lemma 1: A node *n* is a dominator in an SPG if and only if *n* is a dominator in all of the corresponding PSPGs.

Proof: If node *n* is a dominator in the SPG, then *n* is a dominator in all of the subgraphs of the SPG, which include all of the corresponding PSPGs.

Conversely, suppose that *n* is a dominator in all of the corresponding PSPGs. Assume that *n* is not a dominator in the SPG. There are at least two paths going from the initial to a final node *f* that include node *n*. Since both paths contains the final node *f*, they are in the same corresponding PSPG for node *f*. Thus, node *n* is not a dominator in the PSPG for node *f*. A contradition.

Q. E. D.

Fig. 6 shows the SPG for the corresponding PSPGs in Fig. 5. Nodes for S $\rightarrow \bullet$ A (nodes 1 and 6 in Fig. 5) are combined into one node. Nodes for A $\rightarrow \bullet$ C (nodes 3 and 7) are also combined. The corresponding arcs of these nodes remain unchanged. The only dominator in this SPG is the node for S $\rightarrow \bullet$ A.

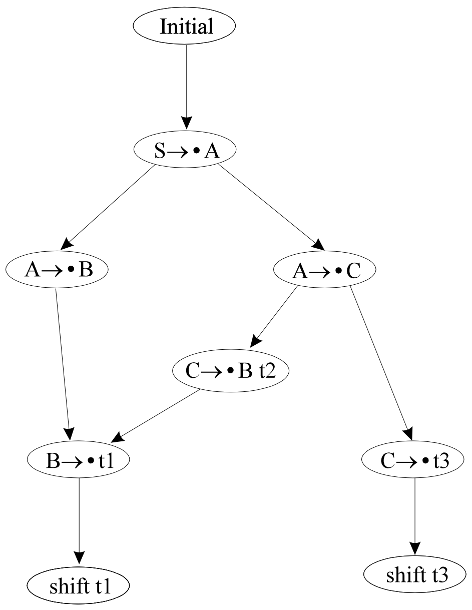


Fig. 6. A state position graph (SPG).

Our algorithm takes the following steps:

1. Label positions that can derive themselves as forbidden.
2. Find all dominators in the state position graph.
3. Label all unlabelled positions as free.

The comparison between our method and Purdom and Brown's is as follows. The time complexity of both algorithms is proportional to the number of nodes constructed: for ours, the nodes in SPGs, and for Purdom and Brown's, the nodes in PSPGs. Assume that both algorithms are implemented with the same constant factor. The speedup is thus

{number of nodes in PSPGs} / {number of nodes in SPG}

Let n_f be the average number of final nodes in a parser state, n_d be the average number of duplicate nodes of PSPGs, and n_t be the average number of nodes in a PSPG. The total number of nodes in PSPGs of a parser state is $n_t \cdot n_f$. The number of nodes in a SPG is $n_t \cdot n_f - n_d(n_f - 1) - (n_f - 1) = n_t \cdot n_f - (n_d + 1)(n_f - 1)$. The speedup is thus

$$n_t \cdot n_f / (n_t \cdot n_f - (n_d + 1)(n_f - 1)).$$

This speedup is very large when $n_t \cdot n_f \cong (n_d + 1)(n_f - 1)$, i.e., most nodes are duplicated and there are many final nodes.

4. HANDLING LR(*k*) GRAMMARS THAT CONTAIN CONFLICTS

Purdom and Brown's method assumes that there are no conflicts in an input grammar. The LR(*k*) grammars that contain conflicts are difficult to understand; however, some of them have been well accepted in practice due to their simplicity. Most LR parser generators have built-in conflict resolvers [1, p. 219]. This section discusses how to generate free positions for such grammars.

LR Conflicts

There are two types of LR conflicts: shift/reduce and reduce/reduce. An example of a shift/reduce conflict is a grammar having *if-then* and *if-then-else* statements with a pair of rules like this:

if_stmt → IF expr THEN stmt
if_stmt → IF expr THEN stmt ELSE stmt .
stmt → if_stmt
expr → ...

Here IF, THEN, and ELSE are terminal symbols; if_stmt, expr, and stmt are nonterminal symbols. The following is a piece of input program:

`if` **x** `then if` **y** `then` **callX**(); `else` **callY**();

There would be a parser state containing

if_stmt → IF expr THEN stmt •
if_stmt → IF expr THEN stmt • ELSE stmt .

Since the stmt may be an if_stmt, the parser cannot determine whether the *else* part belongs to the first `if` or the second `if` statement. That is, the parser cannot determine whether to reduce the production p_1 or shift the ELSE token on the production p_2. Parser generators such as YACC [3] and Bison [4] give a warning message and choose to shift rather than reduce.

A reduce/reduce conflict occurs if there are two or more productions that apply to the same sequence of input. This usually indicates a serious error in the grammar. Consider the following grammar:

S → DECL
DECL → type VAR
DECL → type F_NAME
VAR → id
F_NAME → id

For a "type id" sequence, there are two derivations:

$S \Rightarrow DECL \Rightarrow$ type $VAR \Rightarrow$ type id
$S \Rightarrow DECL \Rightarrow$ type $F\_NAME \Rightarrow$ type id.

In the above derivations, both VAR $\rightarrow$ id$\bullet$ and F_NAME $\rightarrow$ id$\bullet$ can be reduced. Although the program is syntactically correct, its semantic meaning is ambiguous. Parser generators usually choose to reduce the production that appears first in the grammar. In this case VAR $\rightarrow$ id$\bullet$ is reduced.

Generate Free Positions for LR Conflicts

If a reduce/reduce conflict occurs, the parser state looks like

p_1: A $\rightarrow$ C$\bullet$
p_2: B $\rightarrow$ C$\bullet$

These positions are free. Adding semantic actions at the end of both productions neither transforms the conflict nor changes the reduction priority of productions.

In a free position where a shift/reduce conflict occurs, adding a semantic action in the position transforms the conflict into a reduce/reduce conflict. Consider the following parser state with a shift/reduce conflict:

p_1: A $\rightarrow$ C$\bullet$
p_2: B $\rightarrow$ C$\bullet$D

The dotted position in p_1 is a free position since it is the right-most position of p_1. Because the parser shifts D on production p_2 by default, production p_1 will not be matched. If we add a semantic action in the dotted position of p_2, the parser replaces the semantic action with a new nonterminal symbol:

p_1: A $\rightarrow$ C$\bullet$
p_2: B $\rightarrow$ C$\bullet$XD
p_3: X $\rightarrow$ $\bullet$

The resulting grammar contains a reduce/reduce conflict on p_1 and p_3. The parser reduces on production p_1 by default because production p_1 is the first of all conflicting productions in the grammar. Now the semantic action at the end production p_1 will be executed. Assigning this position to be free changes the semantics, so this position is forbidden.

5. IMPLEMENTATION AND EXPERIMENTAL RESULTS

Our implementation is based on Bison [4], which is a part of the GNU project. Bison's implementation first generates LR(0) parser states internally. These states are then translated into LALR states. When generating LR(0) parser states, it does not store the transitions between items of a parser state. We replace Bison's code that generates parser states with the code that constructs state position graphs.

We have tested the grammars of several programming languages, including grammars for Ada, C, C++, Fortran, and Pascal, as shown in Table 1. Table 2 shows the statistics of free positions in these grammars. Our results show that there are about 53.9% to 86.9% of free positions in these grammars. The free positions removed due to LR conflicts are minor. The only exception is in the C++ grammar, where 6.9% of positions are forbidden due to LR conflicts.

Table 1. Grammars used in the experiment.

Grammar	Description	Author
Ada	A parser for Ada83	Herman Fischer (HFisher@eclb.arpa)
C1	A parser for ANSI C	Jeff Lee (CSNet@CSNet-Relay.ARPA)
C2	Another C grammar	-
C3	An ANSI C conformant grammar	Jim Roskind (jar@ileaf.com)
C++	A grammar concludes compliance with C++ 2.0	Jim Roskind (jar@ileaf.com)
F77s	A grammar for a subset of FORTRAN 77	John Levine (Levine@yale.edu)
Pascal1	A grammar for the Pascal language	-
Pascal2	A grammar for ISO Level 0 Pascal, plus some simple extensions	Arnold Robbins

Table 2. Statistics of free positions in the tested grammars.

	Parser States	Total positions	Free positions	Forbidden positions	% of free positions	S/R conflicts	R/R conflicts	Positions "forbidden" by conflicts
Ada	858	1421	1081	340	76.1%	0	0	0
C1	366	698	483	215	69.2%	1	0	1
C2	494	900	613	287	68.1%	1	0	1
C3	509	944	628	316	66.5%	1	0	1
C++	1233	2172	1170	1002	53.9%	24	18	149
F77s	178	315	247	68	78.4%	0	0	0
Pascal1	319	534	464	70	86.9%	1	0	1
Pascal2	407	763	548	215	71.8%	0	0	0

S/R = shift/reduce; R/R = reduce/reduce.

Table 3 shows the number of nodes in our algorithm compared with Purdom and Brown's. Our experimental results show that the reduction ratios of our method range from 55.1% to 93.5% for the eight tested grammars. The time complexity of both algorithms is proportional to the number of nodes constructed. The reduction in the number of nodes thus contributes to the speedups in our algorithm, which range from 2.23 to 15.50 times for the tested grammars.

Table 3. Speedups of our algorithm compared with Purdom and Brown's.

	Total nodes of PSPGs (1)	Total nodes of SPGs (2)	Reduction ratio 1- (2)/(1)	Speedup (1) / (2)
Ada	27153	7439	72.6%	3.65
C1	72848	8708	88.0%	8.37
C2	92401	11603	87.4%	7.96
C3	100346	11388	88.7%	8.81
C++	675630	43584	93.5%	15.50
F77s	4320	1938	55.1%	2.23
Pascal1	6629	2808	57.6%	2.36
Pascal2	10201	4050	60.3%	2.52

6. CONCLUSIONS

In this paper, we have presented an efficient method to determine whether a position is free or forbidden in LR(k) grammars. We have also discussed the interactions of LR conflicts and free positions. Positions with shift/reduce conflicts are not free; positions with reduce/reduce conflicts are free. Our experimental results show that the speedups of our algorithm range from 2.23 to 15.50 times for the eight tested grammars. Our results also show that about 53.9% to 86.9% of positions are free. The free positions removed due to LR conflicts are quite minor.

ACKNOWLEDGMENTS

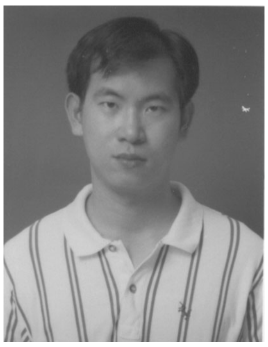
The author would like to thank the referees, whose comments helped improve the overall presentation. This research was partly supported by National Science Council, Taiwan, R.O.C., under Contract No. NSC 89-2213-E-346-002.

REFERENCES

1. D. Grune and C. J. H. Jacobs, *Parsing Techniques: A Practical Guide*, Ellis Horwood, 1990.
2. C. A. Brown and P. W. Purdom, "Methodology and notation for compiler front end design," *Software – Practice and Experience*, Vol. 14, 1984, pp. 335-346.
3. J. R. Levine, T. Mason, and D. Brown, *Lex & Yacc*, O'Reilly & Associates, Inc. 1990.
4. C. Donnelly and R. Stallman, *Bison: The YACC-Compatible Parser Generator*, Free Software Foundation, 1992.
5. A. Bhamidipaty and T. A. Proebsting, "Very fast YACC-compatible parsers (for very little effort)," *Software-Practice and Experience*, Vol. 28, 1998, pp. 181-190.
6. M. Fuketa, K. Morita, S. Lee, and J. Aoe, "Efficient controlling of parsing-stack operation for LR parsers," *Information Sciences*, Vol. 118, 1999, pp. 145-157.

7. H. Mossenboch, “A convenient way to incorporate semantic actions in two-pass compiling schemes,” *Software-Practice and Experience*, Vol. 18, 1988, pp. 691-700.
8. G. D. Finn, “Extended use of null productions in LR(1) parser applications,” *Communications of the ACM*, Vol. 28, 1985, pp. 961-972.
9. P. Purdom and C. A. Brown, “Semantic routines and LR(k) parsers,” *Acta Informatica*, Vol. 14, 1980, pp. 299-315.
10. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, 1986.
11. T. Lengauer and R. E. Tarjan, “A fast algorithm for finding dominators in a flow graph,” *ACM Transactions on Programming Languages and Systems*, Vol. 1, 1979, pp. 121-141.

Shih-Ting Ouyang (歐陽士庭) was born on April 29, 1970, in Taipei, Taiwan. He received B.S. of Electronic Engineering in 1993 and M.S. of Computer Science and Information Engineering in 1995, both from National Chiao Tung University. He now works for Sunplus Technology Co., Ltd. to port compilers for micro-controllers and digital signal processors. His research interests include programming languages, compiler design, operating systems, embedded systems and software engineering.



Pei-Chi Wu (吳培基) was born on March 11, 1967, in Hsinchu, Taiwan, the Republic of China. He received B.S., M.S., and Ph.D. from National Chiao Tung University, in 1989, 1991, and 1995, respectively, all in Computer Science and Information Engineering. He became the member of ACM and IEEE computer Society since 1996. He has been with Department of Computer Science and Information Engineering, National Penghu Institute of Technology, Taiwan, since August 1998. His research interests include multilingual systems, extensible markup language, object-oriented programming, compiler design, random number generators, and high-performance distributed computing.

Feng-Jian Wang (王豐堅) got his B.S. degree from National Taiwan University, 1980, and M.S. and Ph.D. degree from Northwestern University, U.S.A.,1986 and 1988 respectively. He is currently a professor in the Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu. His research interests include software engineering, OO & related techniques, the Internet, workflow and agent applications, and distributed software systems.