Chapter 5

Table Optimization and Generation

In general, one of the main objectives that must be achieved in generating parsing tables is that they must be time- and space-efficient. In addition, if the resulting parser must recover from input errors, the tables generated must retain enough important information about the input grammar and its LR(0) automaton to accommodate error recovery. Unfortunately, these two goals are sometimes conflicting. For example, as mentioned earlier, when space-saving optimizations such as default reductions (and others that will be described in this chapter) are used, the resulting parsing tables lose certain information about the automaton. Space-saving optimizations are necessary in order to reduce the parsing tables to an acceptable size but the information they lose (such as the viable candidates in a given state) is usually important to insure good recoveries.

In the past, authors who have studied LR error recovery have proposed that some tradeoffs be made regarding space-efficiency, time-efficiency and quality of recoveries. For example, in [21], it is suggested that default reductions not be used in certain states to prevent premature reductions in these states and that only terminal symbols on which transitions are defined in a given state be considered as simple recovery candidates (this information must be available in the parsing tables). In [37], a deferred parsing technique using two parsers is used to avoid premature reductions and all terminal symbols are considered as possible candidates during simple recovery; in effect, time-efficiency is sacrificed in order to gain space-efficiency and obtain good recoveries.

By contrast, the approach taken in this method is to treat the issue of table compaction and optimization for parsing separately from the issue of table compaction and optimization for error recovery. The data for each activity is aggressively optimized and compacted, separately. The result is that the amount of space used for each of these two sets of tables is very small, (usually, the total amount of space used is much smaller than it would be for a single set of parsing tables in other methods). Nonetheless, these tables are very time-efficient and no useful error recovery information is lost.

5.1 Parsing Tables

As described earlier, an LALR(k) parser with varying-length lookahead strings can be represented by a pair of parsing tables in matrix form: ACTION and GOTO. Each row of a parsing table is labeled with a distinct state of the parser. Each column of ACTION is labeled with a terminal symbol and each column of GOTO is labeled with a nonterminal symbol. The entry in a parsing table for a given row and column is the parsing action associated with the corresponding state-symbol pair. In essence, the matrix is used to store the characteristics of the pushdown *automaton* that obeys the grammatical rules of the language. In terms of speed performance, the matrix is an efficient method for representing an LR parsing table. Unfortunately, such a matrix is usually very sparse, requiring an excessive amount of space relative to the number of useful entries it contains. For example, an LALR(1) parser for the Ada language with 541 states and 403 symbols only contained 4151 useful entries, thus utilizing only 1.9% of the matrix.

An LR automaton can also be represented as a directed graph, where each vertex of the graph represents a state of the automaton and each edge, labeled with a symbol, represents transitions between states. (In this representation, one must be able to differentiate between lookahead states and regular states. In addition, special *reduce states* (called "lookahead states" in [6]) and *shift-reduce states* must also be used for reduce and read-reduce actions, respectively.) This representation has the advantage of compactness since it needs only be as large, proportionally, as the number of significant entries in the corresponding matrices. However, the computation of an action using this representation is slow since it may require that each out-edge connected to a state be explored.

From this discussion, one gathers that a desirable goal in generating LR parsing tables is to find a representation that performs as well as the matrix in terms of time-efficiency but whose space requirement is close to that of the graph representation.

A number of methods are known in the art for the compression of LR parsing tables. The compression can be achieved, for example, by the use of hashing, linear lists [33], row-displacement [33, 30, 23, 15], or graph-coloring [30].

Hashing is a standard technique for storing and searching large sparse tables, but it is seldom used for LR parsing applications because of its poor worst-case performance. It also consumes a large amount of space.

Substantial space savings result when the significant entries in each row of a matrix are stored in a linear list. The list, however, must be searched sequentially when a parse action is needed. Therefore, the time required to determine a parse action is not constant, but depends on the number of significant entries in the state in question. This method, discussed in [33], does save space but at the expense of time.

In the row-displacement method, the rows of a sparse matrix are *overlaid* on each other in a one dimensional table and an auxiliary table is used to retrieve the starting index of each row in the overlay table. Each entry in the overlay table contains a *check* field that is used to verify whether or not that entry corresponds to the original useful entry in the matrix. This method is advocated by Aho, Sethi and Ullman in [33] and by Ziegler in [15], and is discussed in detail in [30] and [23]. It is also used together with list searching in the YACC parser generator, as described in [33]. The row-displacement method does well with respect to time-efficiency, but its space utilization is not always optimal unless the matrix in question has a "harmonic decay" property [23] and a *first-fit decreasing* heuristic is used.

In the GCS (graph-coloring scheme) method proposed by [30], the rows of the original matrix are partitioned into classes of rows that do not have different significant entries in any column position. These rows are then merged to form a shorter matrix. Next, the columns of this new matrix are partitioned and merged, using the same method, to obtain a final reduced matrix. A vector *rowmap* indexable by the row indexes of the original matrix and a vector *columnmap* indexable by the column indexes of the original matrix are used to map the row and column indexes of the original matrix into the row and column indexes of the reduced matrix. In addition, when this method is used to compress the ACTION table of an LR parser, a boolean matrix *sigmap*, indexable by the indexes of the original matrix, is required to validate entries in the reduced matrix. If the boolean values in *sigmap* are contained in a single bit, this method yields relatively small parse tables where actions can be computed with a constant number of operations. However, since accessing a bit element of a matrix is usually a complex operation in most machines and *sigmap* must be accessed for each terminal action, the time-performance of this method may not be so good.

In the method presented here, the compaction of LR parsing tables is performed in three steps. In the first step, a number of transformations are applied on the original parsing tables to significantly reduce the number of rows and useful entries in them. (The resulting tables usually have the harmonic decay property.) In the second step, row-displacement is used to compact the resulting transformed matrices. The final step is a clean-up step where the actions in the compressed table are updated and all auxiliary tables are eliminated. This approach often yields perfectly compacted tables that require a fixed number of primitive operations on integers to compute an action.

5.1.1 The Row-Displacement Scheme

Let A be a sparse matrix with m_1 rows, m_2 columns and n useful entries. Such a matrix A is usually stored in row-major order as a one-dimensional array A' containing $m_1 \times m_2$ elements. In this representation, each element $A(i,j)$ of the matrix corresponds to the element $A'((i-1)*m_1+j)$ of the one-dimensional array. Let *row* be an auxiliary one-dimensional array of m_1 elements, each containing the offset of its corresponding row in the matrix; i.e., $row(i) = ((i-1)*m_1)$, $1 \leq i \leq m_1$. Using *row*, an element $A(i,j)$ corresponds to the element $A'(row(i)+j)$.

The row-displacement scheme is a method for compressing a sparse matrix A into two parallel one-dimensional arrays, CHECK and INFO, with fewer positions than A'. Given a pair (i,j), the CHECK array is used to test whether or not that pair is associated with a useful entry in A. If it is, the relevant information is retrieved from the INFO array.

The compaction is performed by overlapping the rows of A and placing their values in INFO in such a way that no two useful entries end up in the same position. The algorithm can be stated more formally as follows. For each row i in A, a sequential search is performed on the elements of INFO until a set of valid positions is found that can accommodate the useful entries of row i. Next, the ith entry of the *row* vector is initialized with the displacement of row i in INFO; each useful entry $A(i,j)$ is placed in its assigned location, INFO$(row(i)+j)$, and the corresponding CHECK$(row(i)+j)$ element is set to i

	a	b	c	d	e
1	2	3	0	1	0
2	0	3	0	0	4
3	2	0	5	0	0
4	0	0	0	5	0
5	2	0	0	0	4

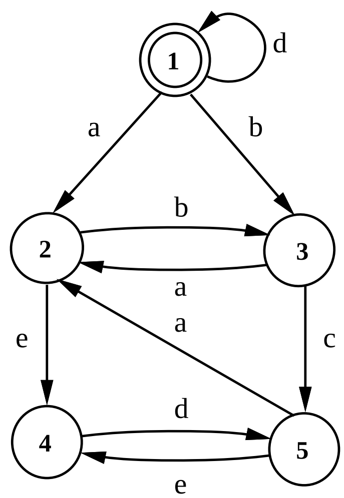
A

1	2	3	0	1	0								
2		0	3	0	0	4							
3					2	0	5	0	0				
4					0	0	0	5	0				
5									2	0	0	0	4

	row
1	1
2	2
3	5
4	5
5	9

INFO	2	3	3	1	2	4	5	5	2	0	0	0	4
CHECK	1	1	2	1	3	2	3	4	5	0	0	0	5
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*

Figure 5.1: A matrix A and its compressed representation



(a)

INFO	2	3	3	1	2	4	5	5	2	0	0	0	4
CHECK	1	1	2	1	3	2	3	4	5	0	0	0	5
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*

Start
1

(b)

Figure 5.2: A finite state automaton and its compressed representation

to indicate that the element $row(i) + j$ of INFO is an element of row i in A. This algorithm is known as the "*first-fit*" method. (It is NP-complete to find a set of displacements that minimizes the size of the compressed vector [23] [16]).

Figure 5.1 depicts a matrix A and a possible compressed representation for that matrix. Non-useful entries in A contain the value 0. The columns of A are labeled with the sequence of letters [a,b,c,d,e] to avoid confusion. These indexes can easily be mapped into the integers [0..4] as the row-displacement scheme assumes that the indexes of the matrix are integers. To look up an element $A(i,j)$, one checks whether or not CHECK$(row(i) + j) = i$. If the test succeeds then INFO$(row(i) + j)$ is the relevant value of $A(i,j)$; otherwise, $A(i,j)$ is not a useful entry and 0 is the relevant value.

Note that if each row of a matrix has a unique displacement in its compressed representation then the column indexes of the matrix can be used as check values instead of the row indexes. This is particularly useful if $m_1 \ll m_2$ (say $m_1 < 256$) and can be stored in a smaller storage unit than m_2. Note also that the lower bound of the compressed vector does not have to be 1, but instead can be an arbitrarily chosen integer.

Assume that the matrix A of Figure 5.1 is a table representation of the transitions of a finite state automaton (Figure 5.2(a)). Assume that state 1 is the root state and as before, the edge labels [a,b,c,d,e] are mapped into the integers [0..4]. In such a case, as was stated earlier, the direct access capability of the matrix does not need to be preserved and the auxiliary displacement vector can be eliminated by replacing each useful entry in INFO by its displacement value. See Figure 5.2(b). Of course, in such a case the displacement of the starting state must still be stored in an auxiliary $Start$ variable.

Ziegler suggested that the rows of a matrix to be compacted with the row-displacement scheme be sorted in decreasing order by the number of useful entries they contain before applying the compaction algorithm. This approach is known as the "*first-fit decreasing*" method. Tarjan and Yao [23] showed that this approach always produces a perfectly compacted vector "if the distribution of nonzeros among the rows are reasonably uniform". Their analysis was stated as follows:

> Our intuition is that rows with only a few nonzeros do not block too many possible displacement values for other rows; it is only the rows with many nonzeros that cause problems. To quantify this phenomenon, we define $n(l)$, for $l \geq 0$, to be the total number of nonzeros in rows with more than l nonzeros. Our first theorem shows that if $n(l)/n$ decreases fast enough as l increases, then the first-fit decreasing method works well.
>
> **Theorem 5.1.1** *Suppose the array A (to be compressed) has the following "harmonic decay" property:*
>
> $$H\text{: For any } l,\ n(l) \leq n/(l+1).$$
>
> *Then every row displacement $row(i)$ computed for A by the first-fit decreasing method satisfies* $0 \leq row(i) \leq n$.
>
> ...If A has harmonic decay, at least half the nonzeros in A must be in rows with only a single nonzero. In addition, no row can have more than $\sqrt{n}$ nonzeros.

5.1.2 Transformations

Given an augmented grammar $G = (N, T, P, S)$ and its LALR(k) parser with varying-length lookahead strings and LR(0) reduce states removed, an entry in the ACTION matrix of such a parser can have one of six values. They are:

- *shift* p
- *reduce* $A \to \omega$
- *shift-reduce* $A \to \omega$
- *la-shift* p'
- *accept*
- *error*

where p is an LR(0) state of the parser, $A \to \omega \in P$ and p' is a lookahead state. The *error* parsing action is a constant value that is associated with each element of the ACTION matrix whose indexes correspond to a state-terminal pair on which no action is defined in the automaton. The parsing action *accept* is also a constant value. When an LR parser is constructed from an augmented grammar, *accept* is associated with a single entry in the matrix. During parsing, *accept* signals the successful completion of a parse.

An entry in the GOTO table can have one of three values. They are:

- *goto* p
- *goto-reduce* $A \to \omega$
- *don't care*

where p and $A \to \omega$ are defined as before. A non-useful entry in the GOTO table is referred to as a *don't care* entry instead of as an *error* entry because such an entry is never consulted during parsing.

5.1.2.1 GOTO Default Actions

Since "don't care" entries in the GOTO table are never accessed, a significant decrease in the number of useful entries in that table can be obtained by introducing default actions on nonterminals [33]. In this method, this is achieved as follows. Let GOTO_DEFAULT be a vector whose elements are indexed by the nonterminals. For each nonterminal A, scan the column of GOTO indexed by A and find the most frequent action, *act*, in that column. Remove all occurrences of *act* found in the A column and set GOTO_DEFAULT(A) = act. Looking at this transformation from the point of view of the directed graph representation, this is equivalent to removing all incoming edges labeled A that point to the state associated with the action *act*. (If *act* is a shift or lookahead-shift then all the incoming edges of its corresponding state are labeled A.) When actions are removed from the GOTO table, these entries and all the "don't care" entries are replaced by error entries. During parsing, if a state-nonterminal pair (p, A) yields *error*, GOTO_DEFAULT(A) is used.

5.1.2.2 Merging of Compatible States and Default Reductions

In chapter 3, the concept of default reduce actions was introduced as a way to optimize space in the ACTION matrix. In this method, the "default reduce actions" optimization is combined with the *merging of compatible states*. Together, these two optimizations result in a significant decrease in the space requirement of the ACTION matrix.

Definition 5.1.1 A state p is said to be compatible with another state q if and only if the following conditions are satisfied:

- p has the same terminal transitions as q; i.e., the rows of the ACTION matrix associated with p and q have the same shift, shift-reduce and lookahead-shift actions defined in their corresponding columns.
- The set of rules on which reduce actions are defined in q is the same as the set of rules on which reduce actions are defined in p.
- For each terminal a on which a reduce action is defined in both p and q, the reduce action is the same.

Two compatible states p and q can be merged into a single state r where the terminal transitions of r are the same as the terminal transitions of p (or q) and the reduce actions of r is the union of the reduce actions in p and the reduce actions in q. During parsing, if state r is entered in a context where p should have been entered and a reduce action that originated solely from q is executed, that action has the same effect as a default reduction. Similarly, if r is entered instead of q and a reduce action of p is executed in r, it can be viewed as a default reduction. Thus, unlike the method of [30], when states are merged based on the above criteria, the resulting merged states retains their error detecting capabilty and no external boolean check matrix is needed.

The set of states in a parser can be partitioned into a set of *compatible classes* based on the concept of compatible states described above. In other words, a compatible class of states is a subset of the set of states where any two states in the class are compatible. In general, finding an optimal partition of compatible classes which minimizes the number of subsets in the partition is NP-complete. However, the practical goal one tries to achieve in merging compatible states is to take advantage of the fact that most languages, especially programming languages, contain some phrases that are used in different contexts, but the actions induced by these phrases in the different contexts are compatible. For example, most procedural languages support arithmetic and relational expressions which are derived from a generic nonterminal expression and used in different statements such as:

$$\begin{array}{lcl} \mathsf{statement} & \rightarrow & \mathsf{variable} := \mathsf{expression}; \\ & | & \textbf{while}\ \mathsf{expression}\ \textbf{loop}\ \ldots \\ & | & \textbf{until}\ \mathsf{expression}\ \textbf{loop}\ \ldots \\ & | & \textbf{for}\ \mathsf{expression}\ \textbf{do}\ \ldots \\ & | & \textbf{if}\ \mathsf{expression}\ \textbf{then}\ \ldots \end{array}$$

Given a set of states of an LR parser whose kernel items are all of the form $[\alpha \cdot A\beta]$, where α and β are arbitrary strings, any two such states satisfy conditions 1 and 2 of the compatibility test. In the above example, the nonterminal expression appears in phrases that are all produced from the same source, statement, and no two of these phrases share the same prefix in that context. The LR parser for a grammar containing such rules will contain a state for each rule whose sole kernel item is the item derived from the right-hand side of the rule in question with the "dot" in front of expression. These states form a compatible class.

Experiments have shown that, in most cases, a set of states that meets conditions 1 and 2 of the compatibility test above is made up of states with kernel items having identical dot symbols. This is an important observation because it is condition 3 that renders the problem of finding the smallest partition of compatible classes NP-complete (see Appendix B for proof). To take advantage of this, a practical algorithm for merging states can be implemented in three steps based on the three conditions of the compatibility steps as follows:

step1: Construct a coarse partition based only on condition 1; i.e., states are grouped together if their terminal transitions are identical.

step2: Each partition class obtained from step 1 is broken down into smaller classes by applying condition 2.

step3: Finally, each class C obtained from step 2 whose states contain reduce actions by more than one rule is further fragmented by repeating the following heuristic process until C is empty:

> Remove an arbitrary state from C and use it to initialize a new singleton class C'. Next, each state p in C is considered, in turn. If p is compatible with all the states in C', it is removed from C and added to C'.

Once the states of an LR parser have been partitioned into compatible classes, each class of states is merged into a single merged state. The new merged states obtained by this process are used to form a new ACTION matrix. An extra column indexed by a special *default symbol* is appended to the front of ACTION. A default reduce action is computed for each merged row and that value is entered in the default column position of the row in question. The default reduce action of a given row, as described before, is the most frequently occurring reduce action in that row if there is any, otherwise it is the error action.

Now, since the terminal transition entries in the new ACTION matrix still refer to the original state numbers and having merged the rows of the ACTION matrix independently of the GOTO table, one must be able to relate each original state of the parser (still associated with a row of GOTO) with its merged counterpart in ACTION. This can be achieved by adding a new column to the GOTO table, each element of which points to its associated merged row in the new ACTION matrix. From now on, ACTION will denote the new matrix obtained after compatible states have been merged and default reductions computed.

5.1.3 Compaction of the Parsing Tables

Recall that in order to execute a reduce action by a given rule, the length of the right-hand side and the left-hand side nonterminal of the rule must be known. Therefore, to accommodate these actions, a vector, RHS_SIZE, indexable by rule numbers is initialized with the length of the right-hand side of each rule and a vector, LHS, also indexable by rule numbers is initialized with the left-hand side symbol of each rule.

LR parsers have two important properties which will be used as the basis of some important optimizations to improve both space utilization and the time performance of the compacted tables. They are reviewed here:

1. Since each time a parsing action is executed it determines the next state that will be entered, the direct access property of the matrix does not have to be preserved. In other words, since the parser can only enter a state p if a read transition places it in p or a read transition had previously placed p in the state stack and a reduce action re-exposed p on top of the stack by popping states corresponding to the right-hand side of a rule, only actions that are read transitions to state p need to be able to access p.

2. Another automaton, isomorphic to the original automaton, that recognizes the same language may be constructed by permuting and/or relabeling the rows and columns of the parsing tables, and changing the read transition entries accordingly. Viewed as a graph, permuting the rows constitutes a relabeling of the vertices (states) of the graph; and likewise, permuting the columns constitutes a relabeling of the edges of the graph or a renaming of the symbols.

5.1.3.1 The GOTO Matrix

Usually, the GOTO matrix contains many rows with no useful entries in them. After nonterminal actions are removed from GOTO by default, the usually sparse matrix ends up being even sparser. With the addition of the column that identifies the corresponding rows in ACTION as the first column in GOTO, many rows that previously had no action end up with a single action and the resulting matrix almost always has harmonic decay. Before compacting the GOTO matrix, its columns are sorted in decreasing order by the number of useful entries they contain. The GOTO matrix is then compacted into two one-dimensional arrays, GOTO_CHECK and GOTO_INFO, with lower bound $|P| + 1$.

Since each element of the identification column contains a useful entry, this column will be the first column in GOTO after the columns have been sorted. It follows that each row of GOTO will have a unique displacement in its compressed representation and the column indexes (nonterminal symbols) of GOTO can be used as check symbols in GOTO_CHECK. Furthermore, since the application of default actions to GOTO usually removes all useful entries in some columns, the sorting places these empty columns in the upper range of the column indexes - minimizing the index values that may appear in GOTO_CHECK. Finally, observe that the elements of GOTO_CHECK that correspond to a state identification entry in GOTO_INFO can be used for any purpose since they are not associated with any parsing action. One only has to ensure that a value placed in these locations does not fall in the range of the column indexes of GOTO.

5.1.3.2 Encoding of the Parsing Actions

Since the *goto* actions in GOTO_INFO represent the transitions of a finite state automaton, each such state entry can be replaced by its corresponding row displacement. This mapping eliminates the need for a displacement vector. Furthermore, it allows a convenient encoding of the different kinds of values that may appear in the parsing tables. Let m be the highest displacement associated with a state, any integer r such that $1 \leq r \leq |P|$ represents a rule; any integer q such that $|P| < q \leq m$ can potentially represent a state; the *accept* action is represented by the constant $m+1$ and the *error* action is represented by $m+2$; an integer sr such that $error < sr \leq error + |P|$ represents a shift-reduce action; finally, an integer ls such that $error + |P| < ls$ potentially represents a lookahead-shift action.

With this encoding, one can easily identify what kind of action a given entry in a compressed parsing table represents. Reduce and goto-reduce actions are represented by the number r associated with the rule in question. Shift and goto actions are represented by the number q associated with the state in question. Shift-reduce actions are encoded as described above and the constant *error* must be subtracted from these values to obtain the rule in question. Similarly, the constant $error + |P|$ must be subtracted from the value of a lookahead-shift action to obtain the lookahead state in question.

After the GOTO table has been compressed, enough information is known to encode all parsing actions in both matrices except the lookahead-shift actions since these actions are transitions to lookahead states that are associated with rows in ACTION that have no counterpart in GOTO. The initialization of the state identification entries in GOTO_INFO must also be deferred until after ACTION has been compacted. Thus, a clean-up phase is required after the compaction of ACTION has been performed. This will be discussed later.

5.1.3.3 The Action Matrix

A typical LR parser usually contains many states (but less than half the total number of states) on which only a single terminal action is defined. The removal of many reduce actions by default further increases the number states with a single action as well as the sparseness of the ACTION matrix. If an optimized ACTION matrix is considered without its default reduction column, it usually has harmonic decay. Therefore, a good theoretical approach for compacting ACTION is to remove the default reduction column from it, store the default reduce action associated with each state in the unused entry in GOTO_CHECK associated with that state, and compress ACTION without the default reduction column. Note that when this approach is used, the rows of ACTION will not automatically have unique displacements. From a practical point of view, it may be beneficial to enforce this restriction because a typical grammar contains fewer than 255 terminals but a parser for such a grammar usually contains several hundred states. Therefore, if the column indexes of ACTION can be used as check values they can be accommodated in a single *8-bit byte*. On the other hand, the number of states may not only exceed 255 but, moreover, when states are replaced by their displacements, this further increases their range.

A second approach is to assume that each entry in the default reduction column is useful and sort the columns of ACTION in decreasing order by the number of useful entries in them. As in the case of the GOTO matrix, this guarantees that a column containing only useful entries will be the first column in ACTION and each row of ACTION will have a unique displacement in the compressed representation. ACTION is compressed into two one-dimensional arrays, ACTION_CHECK and ACTION_INFO, with lower bound 1.

If the input grammar used to generate an LR parser does not contain useless nonterminals (nonterminals that do not produce any terminal string), each state of the parser will have at least one terminal action. In that case, when the entries of the default reduction column are considered to be useful entries, each row of ACTION has at least two useful entries and thus, ACTION does not have harmonic decay. However, experiments have shown that when the columns of ACTION are sorted in decreasing order prior to applying the first-fit decreasing method, this rearrangement tends to produce good compaction. This second approach for compacting ACTION is the one that is used in this method. For practical reasons which will become clearer later, it turns out that in most cases, this approach saves space and, in any case, it produces tables that give a slightly faster time performance.

5.1.3.4 Compressed Tables Update

In this section, the final clean-up phase of the compaction procedure is described. Let d be the displacement (in ACTION_INFO) of a row associated with a lookahead state. Each entry in ACTION_INFO that is a transition into that lookahead state is replaced by the value $error + |P| + d$. Thus, when a lookahead-shift action is decoded, it is the displacement of the relevant lookahead state in ACTION_INFO that is obtained. Next, the displacement element in GOTO_INFO associated with each state is updated with the displacement of its corresponding ACTION row in ACTION_INFO. The encoding of all parsing actions in the compressed tables is now complete.

In order to avoid having to check for boundary conditions during parsing, one must be able to check whether or not a useful entry is defined for any state-symbol pair. This implies that the GOTO_CHECK vector must contain at least $m + |N|$ elements, where m is the highest displacement associated with a state. Similarly, the ACTION_CHECK vector must be extended to contain $m' + |T|$ elements, where m' is the highest displacement associated with an ACTION row.

Finally, since each element of GOTO_CHECK whose location corresponds to the location of a state identification entry in GOTO_INFO is still unused, the original row index (negated to differentiate it from check values) associated with the state in question can be stored at that location in GOTO_CHECK. This information is not needed for parsing (even though it might be useful for debugging purposes), but it provides a convenient remapping of each displacement into its original state number which is useful for storing the error recovery maps *t_symbols* and *nt_symbols*. The domain of these two maps is the set of states of the parser. This remapping allows the states to be represented by the row indexes of GOTO instead of the displacements in GOTO_INFO which fall in the larger range $|P| + 1..accept - 1$.

Note that when this remapping of the states is used with the first approach for compacting ACTION discussed in the previous section, an additional DEFAULT vector, indexable by the states, must be used to store the default reduce action associated with each state. This arrangement not only requires extra space for the new vector but it also requires

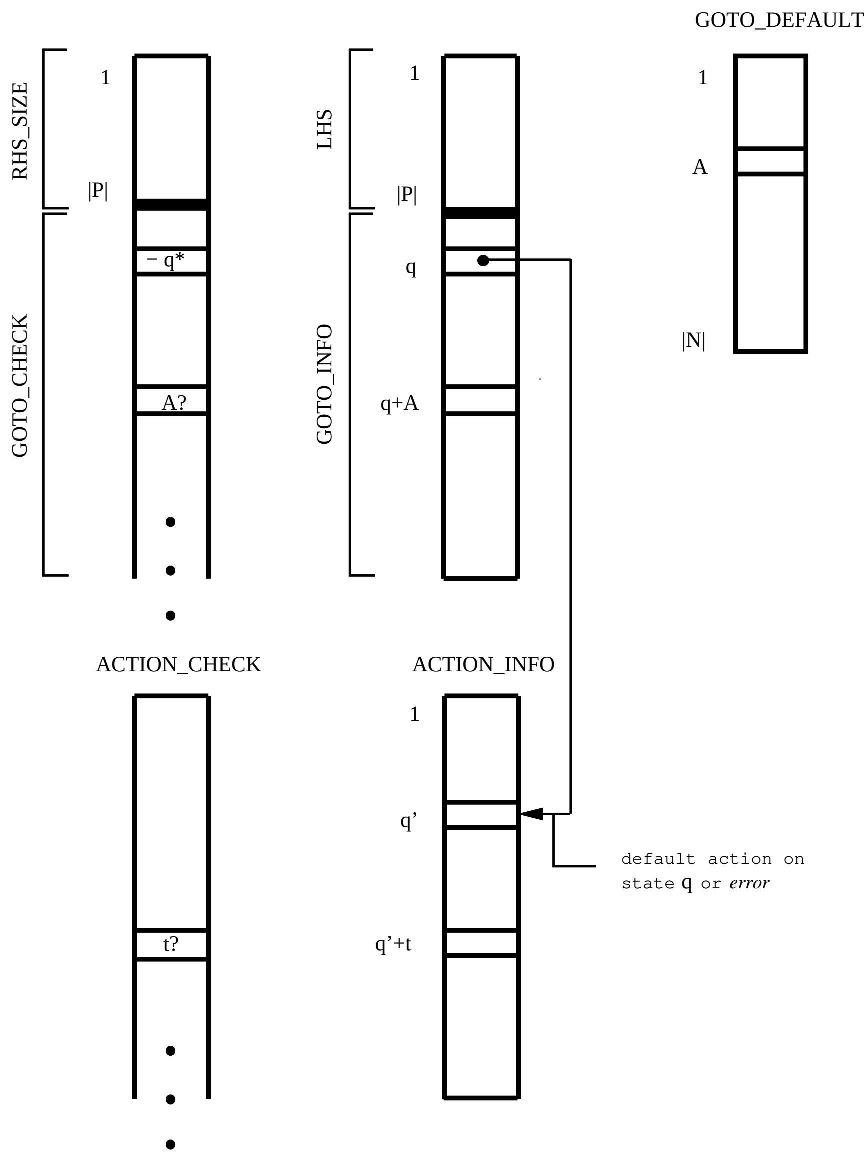


Figure 5.3: Compressed parsing tables

-- *Assume* NUM_RULES $= |P|$ *and* LA_OFFSET $=$ *error* $+$ NUM_RULES.

```
function ACTION(q, t1..tk);
    q' := GOTO_INFO(q);
    if ACTION_CHECK(q' + t1) = t1 then
        act := ACTION_INFO(q' + t1);
    else act := ACTION_INFO(q');
    end if;
    i := 1;
    while(act > LA_OFFSET) loop
        i := i + 1;
        q' := act - LA_OFFSET;
        if ACTION_CHECK(q' + ti) = ti then
            act := ACTION_INFO(q' + ti);
        else act := ACTION_INFO(q');
        end if;
    end loop;
    return act;
end ACTION;

function GOTO(q, A);
    if GOTO_CHECK(q + A) = A then
        return GOTO_INFO(q + A);
    else return GOTO_DEFAULT(A);
    end if;
end GOTO;
```

Figure 5.4: Parsing functions

an additional indexing operation to compute a default reduce action. Experiments have shown that the number of useful entries in an optimized ACTION matrix is usually so small that the extra space lost by not obtaining a perfect compaction using the second approach is exceeded by the space required to store the DEFAULT vector. The computation of a terminal action with the second approach always requires the same number of operations.

5.1.3.5 Computing Parsing Actions on the Compacted Tables

Figure 5.3 shows a graphical representation of the compacted tables. ($q*$ denotes the row index of GOTO with displacement q.) Given such a representation the ACTION and GOTO functions of Figure 5.4 compute the action associated with a state-terminal or state-nonterminal pair, respectively. Note that if the tables were constructed from an LALR(1) parser, the code segment in lines 7 through 15 of the ACTION function can be omitted. The value returned by the ACTION function is either a reduce action if it is less than or equal to NUM_RULES, a shift action if it is greater than NUM_RULES but less than *accept*, or a shift-reduce action if it is greater than *error*. In the case of a shift-reduce action, *error* is subtracted from the value returned by ACTION to obtain the rule in question. The value returned by the GOTO function is either a goto-reduce action if it is less than or equal to NUM_RULES or a goto action, otherwise.

5.2 Error Recovery Tables

The error recovery method proposed in the previous chapter requires that some information be precomputed from the input grammar and its automaton. This information includes the scopes, discussed in section 4.3.3.1, the map *t_symbols* from each state q into a relevant subset of viable terminal error recovery candidates for q, the map *nt_symbols* from each state q into a relevant subset of viable nonterminal error recovery candidates for q and the *names* associated with each viable error recovery candidate.

5.2.1 Optimization of Candidates

Consider the case of a secondary substitution in which a recovery goal A must be inserted into the input stream. In such a case, every nonterminal candidate in the recovery state q_i in question is a potential reduction goal. However, an implementation that checks all potential candidates for each error phrase would be prohibitively slow.

Two optimizations are applied to the set of nonterminal candidates in a given state to obtain, in most cases, a substantially reduced subset of *relevant reduction goals*.

In [27], the following concept is presented: a reduction goal A of error phrase $\beta|x$ in error configuration $\alpha\beta|xy$ is *important* if $\beta|x$ has no reduction goal B such that $B \rightarrow^+ A$. In this method, a more restricted concept of an important symbol is used. The new concept takes into consideration the full context of the error phrase:

Definition 5.2.1 A nonterminal A on which a transition is defined in a state q_i is said to be *important* if A does not appear in a single item of the form $B \rightarrow \cdot A$ in q_i.

$$
\begin{array}{ll}
E \rightarrow \cdot E + T & E \rightarrow \cdot T \\
T \rightarrow \cdot T * F & T \rightarrow \cdot F \\
F \rightarrow \cdot F \uparrow P & F \rightarrow \cdot P \\
P \rightarrow \cdot id & P \rightarrow \cdot (E)
\end{array}
$$

Figure 5.5: Items in a state q_i

For example, assume a recovery state q_i contains the set of items shown in Figure 5.5. By the definition of [27], the only important reduction goal in such a state is E, since T, F and P can be derived from E via a chain of unit productions. By the more restricted definition of this method, T and F would also be considered important symbols since they appear immediately to the right of the dot in more than one item. To understand the importance of T and F, assume that the rules from which the items of Figure 5.5 are derived are all the productions of a grammar and consider the following erroneous input strings:

`( ) ) ( * id + id`
`( ) ) ( ↑ id + id`

If E is the only important symbol considered, then the best secondary repair that is achievable is the replacement of "`( ) ) ( * id`" by E in the first sentence and "`( ) ) ( ↑ id`" by E in the second sentence. However, it is clear from the grammar that replacing "`( ) ) (`" by T in the first sentence and by F in the second sentence would be preferable.

One further notices that using F as a reduction goal in the first sentence would have worked just as well, since after a transition on F, with the symbol "*" as lookahead, a reduction by the rule "$T \rightarrow F$" would be applied. Similarly, P could have been used as a suitable reduction goal in both sentences. This leads to the following concept, on which the second optimization is based:

Definition 5.2.2 *A nonterminal element C of a set of nonterminal candidates S in an LR state q is said to be* relevant *with respect to S if there does not exist a nonterminal D, such that $D \in S$, $D \neq C$, and D can be successfully substituted for C as a reduction goal for any error phrase with q as the recovery state.*

Given a set S of nonterminal candidates for a given state, the objective is to find the largest subset $S' \subseteq S$ such that S' contains only relevant reduction goals.

Lemma 5.2.1 *Let $S = \{B_1, \ldots, B_k\}$; for $1 \leq i \leq k$, $B_i \in S$ is relevant iff $\nexists B_j$, $j \neq i$, such that $B_i \Rightarrow^+_{rm} B_j$.*

The proof follows directly from the definition of an LR parser. If a nonterminal B can be substituted for an error phrase, then the recovery symbol t in question must be a valid lookahead symbol for any rule derivable from B. In particular, if $B_i \Rightarrow^+_{rm} B_j$ and B_j is substituted for an error phase where B_i is known to be a valid reduction goal, the recovery symbol will cause $B_j$to be reduced to B_i.

For each state q in an LR automaton, the set *nt_symbols*(q) is obtained as follows. Starting with the set of nonterminal symbols on which an action is defined in q, remove all unimportant symbols from that set, and reduce the resulting set further by removing all irrelevant reduction goals from it. For example, consider the state q_i of Figure 5.5. State q_i contains nonterminal transitions on the symbols E, T, F and P. The only unimportant symbol in that set is P. After P is removed, the irrelevant symbols E and T are removed from the subset $\{E, T, F\}$ leaving F as the only relevant reduction goal in q_i.

The notion of an important symbol can also be extended to terminal candidates in the *t_symbols* sets. Once again, consider the state q_i of Figure 5.5. This state contains a single terminal action on the symbol *id*, but, since *id* appears only in the item $P \to \cdot id$, it is not an important candidate in q_i. The removal of unimportant terminals improves the time performance of the primary recovery and saves space. However, it may suppress some opportunities for merging and misspelling corrections.

In section 5.2.3, an algorithm is presented that can be used to further reduce the space used by *t_symbols* and *nt_symbols*.

5.2.2 Optimization of Names

In order for the error recovery to issue meaningful diagnoses, the names of the symbols used in the grammar should be saved. In fact, it is very helpful to allow the user to map terse symbol names used in his grammar into more descriptive names. For example, a nonterminal "relop" might be mapped into the string "relational operator"; the end-of-file terminal in an Ada grammar might be mapped into the string "End of Ada source", etc.

Depending on the nature of an input grammar, some nonterminals may never be used in issuing a diagnosis. Therefore, the names associated with these symbols can be removed. Two kinds of nonterminals fall in this category: nullable nonterminals and nonterminals that are always subsumed by another nonterminal. The processing of nullable nonterminals in issuing diagnostic messages was discussed in section 4.4.3. A nonterminal B is said to be always subsumed by another nonterminal if whenever B appears in the right-hand side of a rule, the rule in question is a single-production of the form $A \to B$, for some nonterminal A. In such a case, B (actually, the name associated with B) will never appear in a diagnostic message since the left-hand side symbol A in question would be a higher symbol (see section 4.4.3) than B.

5.2.3 Optimization of Finite Subsets of a Small Universe

Observe that most of the information generated for error recovery consists of finite subsets of a small universe. If the only operation performed on these subsets is iteration, a sequential representation of these sets will yield a good time-performance. For example, *t_symbols* and *nt_symbols* are mappings from states into subsets of terminals and nonterminals, respectively. During error recovery, the only operation performed on these sets of candidates is iteration.

In practice, many of the subsets involved in these applications are identical and can share the same space. In addition, the resulting collection of unique subsets can be further optimized with a new algorithm, called Optimal Partition [40], that takes advantage of the fact that some subsets in that collection are proper subsets of others. Optimal Partition is used to optimize the subsets in the range of *t_symbols* and *nt_symbols* as well as the sets of states associated with the scopes. The operation performed on the set of states associated with a scope is a lookup. However, such a set is usually so small that, in practice, a look up on a sequential representation of it is acceptable.

5.2.3.1 The General Problem

Let Σ be an alphabet and let C be a collection of unique subsets of Σ:

$$C = \{\Sigma_1, \Sigma_2, \ldots, \Sigma_n\}.$$

Find a sequential representation of C that minimizes the storage space required such that the elements of each Σ_i, $1 \leq i \leq n$, occur in consecutive locations.

A set C is said to have the *consecutive retrieval property* with respect to a universe Σ if there exists an organization of Σ (without duplication of any symbol) such that for every $\Sigma_i \in C$ all relevant symbols of Σ_i can be stored in consecutive storage locations. A polynomial time algorithm for finding such an arrangement, if it exists, was presented by Fulkerson and Gross [2]. However, in most practical cases, a pair (Σ, C) does not have the consecutive retrieval property. Hence, duplication of symbols must be allowed so that pertinent symbols corresponding to any subset in C are always stored consecutively. This problem can be stated more formally as follows:

> Is there a string $w \in \Sigma^*$ with $|w| \leq K$ such that for each i, the elements of Σ_i occur in a consecutive block of $|\Sigma_i|$ symbols of w?

This problem known as Consecutive Sets(CS) was posed by Kou in [1] and proven to be NP-complete. Now, consider the problem of finding a string w, of minimum length, required to store the subsets of C such that the elements of each subset occur in a consecutive block of symbols of w and if any two subsets Σ_i and Σ_j share common elements then one is a proper subset of the other. This problem can be stated more formally as follows:

> Is there a partition P of C such that for a given integer K:
>
> - $\forall p_k \in P$, $\forall (x, y) \in p_k^2$, either $x \subset y$ or $y \subset x$.
> - Let $\hat{\Sigma}_k$ represent the largest subset in $p_k \in P$, then $\sum_{p_k \in P} |\hat{\Sigma}_k| \leq K$

This problem is the Optimal Partition problem, OP for short. In Appendix C, OP is proved to be solvable in polynomial-time by reducing it to the maximum weighted bipartite matching problem [40]. In the remainder of this section, a algorithm based on OP is used to optimize the space required to store a set of sets C sequentially, followed by the description of a time-efficient heuristic for OP which, in most cases, finds a partition that is very close to being optimal.

5.2.3.2 Application of OP

With OP, one can take advantage of the fact that there is no overlap of subsets in the elements of the partition P to construct a string w such that each subset can be identified with only one index indicating its starting point. Assuming # is a special marker symbol, $\# \notin \Sigma$, one proceeds as follows:

step0: initialize w to the *empty string.*

step1: **for** each $p_k \in P$ **do** steps 1.1-1.3

step1.1: construct $[\Sigma_{k_1}, \Sigma_{k_2}, ..., \Sigma_{k_m}, \Sigma_{k_{m+1}}]$, an ordered list of the elements of p_k, where $|p_k| = m$, $\hat{\Sigma}_k = \Sigma_{k_1}$, $\Sigma_{k_{m+1}} = \emptyset$, and $\Sigma_{k_i} \supset \Sigma_{k_{i+1}}$, $1 \leq i \leq m$. It is known that such a linear ordering of the elements of p_k is possible by the definition of OP.

step1.2: **for** i **in** $1..m$, append to w all symbols that are in Σ_{k_i} but not in $\Sigma_{k_{i+1}}$.

step1.3: append to w the symbol #.

5.2.3.3 OP Heuristic

Even though OP can be computed by a polynomial-time algorithm, the algorithm in question is not very efficient and in certain environments may prove to be impractical. A simple heuristic for OP will be described below. In most cases, this approach works very well and can be computed in $O(n^2)$ time.

A set $\hat{\Sigma}_k$ that is not included in any other set in a given partition is called a *base* set. The following heuristic is essentially a greedy algorithm that constructs the classes of the partition one at a time and tries to minimize the length of the final string by always including the largest subset it can find. The partition P is constructed as a set of ordered lists of subsets, where each such list make up a class of the partition and the order of the elements in the list is exactly the proper linear ordering for the subsets:

step 0: $P = \emptyset$;

step 1: *Construct a partition* $P' = \{p_i \neq \emptyset \ : \ i \in [1..|\Sigma|]\}$ *of* C, *where:*

$$p_i = \{\Sigma_k : \Sigma_k \in C \text{ and } |\Sigma_k| = i\}$$

step 2: *do* steps 2.1 *and* 2.2

step 2.1: *Let* p_k *be the class in* P' *with the largest subsets. Let* x *be an arbitrary subset of* p_k, *remove* x *from* p_k *and initialize* $list := [x]$;

step 2.2: **if** $p_k = \emptyset$ **then** *remove* p_k *from* P'; **end if**;

step 3: **for** i **in** $[k-1, k-2..1] \ | \ p_i \in P'$ **loop**
  if $\exists\ y \in p_i \mid y \subset x$ **then**
    remove y *from* p_i
    $list := list + [y]$;
    $x := y$
    if $p_i = \emptyset$ **then** *remove* p_i *from* P'; **end if**;
  end if;
 end loop;

step 4: $P = P \cup \{list\}$;

step 5: **if** $P' \neq \emptyset$ **then goto** step 2; **else stop**; **end if**;

Using a bucket sort, step 1 can be computed in $O(|\Sigma| + n)$ time. For steps 2-5, the worst case occurs when no subset is a superset of another and each p_i contains exactly one element. In that case, each subset is removed, in turn, from $p_i \in P'$ and tested against the element in each p_j, $1 \leq j < i$. The total number of subset checks in that case is $n(n-1)/2$, which gives us a total running time of $O(n^2)$. Experimental results using both OP and this heuristic algorithm are presented in Figure C.1 in section C.3.

5.3 Remarks

When parsing tables are compressed with this method, each parsing action can be computed with a fixed number of primitive operations on integers and arrays of integers. For an LALR(1) parser, the computation of a terminal action costs exactly 3 indexing operations, 1 addition and 1 comparison. For an LALR(k) parser, in addition to the initial LALR(1) action, $k-1$ extra lookahead transitions may be required. Each lookahead transition costs 2 additions, 1 subtraction, 2 comparisons and 2 indexing operations. The computation of a nonterminal action always costs 2 indexing operations, 1 addition and 1 comparison.

Figure D.1 of Appendix D shows the space requirement of the parsing and error recovery tables of 6 programming language grammars.

Conclusion

A new method has been presented for constructing practical and efficient LALR(k) parsers with automatic error recovery. Significant contributions and improvements over the current state-of-the-art were made in three areas: parser generation, error recovery, table optimization and table compaction.

In the parser generator described in this thesis, only the minimum amount of lookahead information necessary (up to k levels) to disambiguate inconsistent states in the LR(0) automaton is computed. Consequently, this parser generator is efficient since, in practice, few states require more than one lookahead. Furthermore, the generated parsers are efficient, because at run-time the parser consults extra lookahead symbols only when necessary.

The LR(k) error diagnosis and recovery method presented here is completely language- and machine-independent and more efficient than other known methods. It features the following innovations and techniques:

- a new deferred driver that always detects an error at the earliest possible point;
- a generalized simple recovery method that uses both terminal and nonterminal symbols;
- a phrase-level recovery that is an efficient (and completely automatic) generalization of the error production method;
- a new, completely automatic method for scope recovery.

A number of innovative space optimization ideas are introduced in this thesis. The number of useful entries in the parsing tables and the error recovery maps are reduced and space is shared among similar sets. Following optimization, the parsing tables and the error recovery maps are compacted separately. This allows complete and accurate information to be retained. The resulting compacted parsing tables are the most time- and space-efficient known to date.

Acknowledgements

There are many people without whose help I would not have been able to reach this goal. I would like to give a very special thank you to Fran Allen for her encouragement, support and for giving me the opportunity to go back to school and finish my degree. I thank the IBM Corporation for its financial support. I give a special thanks to my advisor, Edmond Schonberg, for his guidance and insightful suggestions and to Gerry Fisher, my M.S. advisor, who got me interested in this field and with whom I developed the table compression algorithm. I thank the members of the PTRAN group: Mike Burke, Paul Carini, Ron Cytron, Jong Deok Choi, Jeanne Ferrante, Sue Flynn, Emily Plachy, Edith Schonberg and David Shields. Mike and I have been friends for many years. I am particularly thankful to him for helping me with my writing and for many useful technical discussions on error recovery and parsing in general. The error recovery method described in this thesis was inspired by previous work done by Mike and Gerry. Ron and I joined IBM around the same time and we have been friends ever since. Ron has always been very supportive and helpful. Jeanne reviewed some of my papers and made some helpful comments.

Many people have contributed some useful ideas to this work: Laurent Pautet was a co-author of the Optimal Partition solution; Franco Gasperoni proved the NP-completeness of the string compatibility problem; Fritz Henglein helped me on many occasions with some theoretical problems; Matthew Smosna made many worthwhile suggestions which found their way into this work; Yvonne Lesesne translated the original PL/I code to C; Kirk Snyder helped me uncover some bugs and offered many valuable suggestions. Some ideas presented in this work were developed as a result of requests made from the LPG user community at IBM - in particular, Kile Reeves and Ira Sheftman made some helpful suggestions.

Many thanks to my family, especially my mother. Special thanks to Andrea Erstling who has been very supportive and helped me edit this work. Thank you to all my friends, especially Elysée, Fritz, and Selena. I am very grateful to Professor Malcolm Brown of Brooklyn College, Mr Sherman Lee, my undergraduate advisor, and Professor Harry Hamilton of the State University of New York at Albany for their positive influence during my undergraduate years.

This work is dedicated to my aunt, Ms. Jeanne Leonard.

Appendix A

Experimental Results with the LALR Algorithms

In this appendix, experimental results obtained from six grammars of different sizes and complexity are presented. The table of Figure A.1 summarizes the characteristics of each of the grammars used. The Pascal1 and Pascal2 grammars are LALR(2) and the others are LALR(1). Pascal2 is the grammar of Appendix F. Ada is the grammar of [31]. Note that for the Pascal1 and Pascal2 grammars, under the "states" column two numbers are reported. The first is the number of LR(0) states and the second is the number of lookahead states. The automaton for these grammars also contains a lookahead-shift action for each lookahead state.

Grammar	$\|T\|$	$\|N\|$	$\|P\|$	#items	#states	#shifts	#shift-reduces	#gotos	#goto-reduces
Pascal	63	110	213	626	193	396	329	336	574
Pascal1	63	111	215	625	189/1	393	324	332	569
Pascal2	63	111	215	627	191/5	393	326	334	569
C	85	98	266	786	206	349	987	975	322
Ada	96	304	523	1527	516	593	700	884	1779
Sedl	123	362	746	2398	824	1987	5755	2250	4360

Figure A.1: Grammar information

The first three columns in the table of Figure A.1 indicate the number of terminals, nonterminals and productions, respectively, in each grammar; the $\#items$ column indicates the total number of items in the grammar, i.e., let rhs_i represent the list of symbols on the right-hand side of production i, then $\#items = \sum_{i=1}^{|P|} |rhs_i| + 1$; the $\#states$ column indicates the number of states in the LR(0) automaton; the $\#shifts$ and $\#gotos$ columns indicate, respectively, the number of shift and goto transitions in the LR(0) automaton. Figure D.1 gives more information about these grammars and their automata.

The main goal of the experiments was to compare the time performance of each method in terms of the number of union operations it required to compute the LALR(1) lookahead sets for each grammar. Note that lookahead sets were computed only when necessary to resolve conflicts. In particular, lookahead sets associated with LR(0) reduce states were not computed. The KM, DP and PCC method were implemented. It was not necessary to implement the BL method since its time performance, in terms of number of union operations performed, is identical to the DP method.

The KM algorithm was implemented as described in section 2.2.1. The number of union operations required is reported under the $\#unions$ column. When the improvement suggested in section 2.3.1 is added, the number of union operations required is calculated as the number of visits ($\#visits$) made to TRANS. Recall that in such a case a preliminary pass is required to compute the DR sets for each state. The total number of elements in these sets is equal to $\#shifts$. The data from the KM experiments is shown in figure A.2.

KM	$\#visits$	$\#unions$
Pascal	1176	3382
C	13651	36829
Ada	5861	12038
Sedl	30770	67479

Figure A.2: Results of KM experiments

The table of figure A.3 shows the number of union operations required to compute the necessary FIRST sets. The $N - unions$ column indicates the number of union operations required to construct the FIRST sets for nonterminals. The $S - unions$ column indicates the additional operations required to compute the FIRST sets for suffixes that follow a nonterminal in a production of P. The cost of computing these maps was incurred by the DP and PCC algorithms.

$Grammar$	$N - unions$	$S - unions$
Pascal	137	10
Pascal1	137	11
Pascal2	137	13
C	166	0
Ada	372	5
Sedl	508	10

Figure A.3: Unions required for FIRST maps

The DP method was implemented with the improvements suggested in section 2.3.2. The data from the DP experiments is shown in the table of figure A.4. The #READ column indicates the number of union operations required to construct the READ sets from FIRST sets. The #FOLLOW column indicates the number of union operations required to construct the FOLLOW sets from READ sets. The #LA column indicates the number of union operations required to construct the LA sets from FOLLOW sets. The *Total* column includes the union operations incurred by the construction of the FIRST sets.

The data from the PCC experiments is shown in the table of figure A.5. The #PATH column indicates the number of union operations required to construct the PATH sets. The #FOLLOW and #LA columns indicate the number of union operations required to construct the FOLLOW and LA sets, respectively. The *Total* column includes the union operations incurred by the construction of the FIRST sets.

Finally, the table of figure A.6 shows how the different methods performed. Each entry in the table indicates the total number of union operations that were required by the particular method that labels the column and the grammar that labels the row.

DP/Charles	#READ	#FOLLOW	#LA	*Total*
Pascal	93	769	384	1393
Pascal1	90	764	381	1383
Pascal2	91	764	415	1420
C	106	1684	1212	3168
Ada	132	1868	750	3127
Sedl	281	7083	3154	11036

Figure A.4: Results of DP experiments

PCC	#PATH	#FOLLOW	#LA	*Total*
Pascal	853	759	805	2564
C	1888	2114	2317	6485
Ada	3978	1824	1354	7533
Sedl	6886	7926	6154	21484

Figure A.5: Results of PCC experiments

Grammar	KM	DP/Ch	PCC
Pascal	1176	1393	2564
Pascal1		1383	
Pascal2		1420	
C	13651	3168	6485
Ada	5861	3127	7533
Sedl	30770	11036	21484

Figure A.6: Comparison of the methods

Appendix B

String Compatibility

Let Σ be an alphabet containing a special blank character, denoted $\flat$. Consider a set S of n strings of Σ characters, each of length m:

$$S := \{s_1, s_2, \ldots, s_n\}.$$

A string $s_i \in S$ is said to be compatible with another string $s_j \in S$ if and only if for $1 \leq k \leq m$, either $s_i(k) = s_j(k)$ or $s_i(k) = \flat$ or $s_j(k) = \flat$. This compatibility relation is reflexive and symmetric, but not transitive.

The String Compatibility problem is: given some positive integer K, can S be partitioned into $k \leq K$ *compatible classes*. A *compatible class* of S is a subset of S whose elements are pair-wise compatible. The String Compatibility problem is NP-complete. This will be shown by reducing *clique cover* [8] (also known as *Partition into Cliques* [22]) to String Compatibility [44]. Recall the definition of Partition into Cliques:

Instance: Graph $G = (V, E)$, positive integer $K \leq |V|$.

Question: Can the vertices of the graph G be partitioned into $k \leq K$ disjoint sets $V_1, \ldots, V_k$ such that for $1 \leq i \leq k$, the subgraph induced by v_i is a complete graph (also called a clique).

Reduction

Given an instance of "Partition into Cliques", construct a *negative* graph $G' = (V, E')$, where $E' = V \times V - E$. Let each $e' \in E'$ be represented by an integer in the range $1..|E'|$ and let $\Sigma = V \cup \{\flat\}$. The set S is constructed by mapping each vertex v_i into a string s_i of length $|E'|$ and adding s_i to S. The mapping is done as follows. If $e' = (v_i, v_j) \in E'$ then $s_i(e') = v_j$; otherwise $s_i(e') = \flat$. This transformation can clearly be computed in polynomial-time.

Claim: a subset of the strings in S form a compatible class if and only if the corresponding vertices in G form a clique.

If: Let C be a subset of S that forms a compatible class, then for each pair of strings s_i and s_j in C, $(v_i, v_j) \notin E'$. Therefore, $(v_i, v_j) \in E$. It follows that the vertices that correspond to the strings in C induce a clique in G.

Only if: Conversely, given a clique in G and an edge (v_i, v_j) in that clique, the corresponding strings s_i and s_j are compatible. In fact, if s_i is not compatible with s_j then there exists an edge $e' \in E'$ such that $s_i(e') = v_j$ and $s_j(e') = v_i$. But, this implies that $(v_i, v_j) \in E'$ which is a contradiction.

Thus, G can be partitioned into k cliques if and only if S can be partitioned into k compatible classes. $\Box$

Appendix C

Optimal Partition

C.1 Reduction to Maximum Weighted Bipartite Matching Problem

Given an alphabet Σ and a collection $C = \{\Sigma_1, \Sigma_2, \ldots, \Sigma_n\}$, where for each $\Sigma_i \in C$ and $\Sigma_i \subseteq \Sigma$, an effective method is presented to construct a bipartite graph $G = (V, V', E)$ with a weight function W defined on E such that a maximum weighted matching obtained from G defines an Optimal Partition for C with the elements of each class in the partition linearly ordered.

C.1.1 Bipartite Graph

The bipartite graph $G = (V, V', E)$ is constructed fron the collection C as follows. Firstly, each subset Σ_i is associated with a unique node $v_i \in V$ and a unique node $v'_i \in V'$. Next, two bijective functions f and f' are used to make the respective associations:

$$f : C \rightarrow V$$

$$f' : C \rightarrow V'$$

$$V = \{f(\Sigma_i) : \Sigma_i \in C\}$$

$$V' = \{f'(\Sigma_i) : \Sigma_i \in C\}$$

The node yielded by $f(\Sigma_i)$ is denoted v_i and the node yielded by $f'(\Sigma_i)$ is denoted v'_i. Two nodes $v_i \in V$ and $v'_i \in V'$ corresponding to the same subset Σ_i are said to be *twins*.

Let the operator $\subset$ ($\supset$) denote proper inclusion, i.e., $A \subset B$ ($A \supset B$) means that A is a subset (superset) of B, but $A \neq B$. A binary relation $R \subseteq C \times C$ is defined as follows:

$$R = \{(\Sigma_i, \Sigma_j) \mid \Sigma_i \supset \Sigma_j\}$$

R is transitive, non-reflexive, and non-symmetric. The set of edges $E \subseteq V \times V'$ in G is constructed as follows:

$$E = \{(v_i, v'_j) : v_i \in V, v'_j \in V' \mid (\Sigma_i, \Sigma_j) \in R\}$$

Given an edge $(v_i, v'_j) \in E$, v'_j is said to be a *successor* of v_i and v_i is a *predecessor* of v'_j. Let $M \subseteq E$ be a bipartite matching, not necessarily maximal, on G then lemmas 1 and 2 follow:

Lemma C.1.1 *Each $v_i \in V$ is connected to at most one successor in M*

Lemma C.1.2 *Each $v'_j \in V'$ is connected to at most one predecessor in M*

Proof: definition of a bipartite matching.

A node $v_i \in V$ having no successor is called a *terminal* node. A node $v_i \in V$ whose twin has no predecessor is called a *base* node. Let β be the set of base nodes induced by M and let γ be the set of terminal nodes induced by M,

Lemma C.1.3 $\beta \neq \emptyset$

Proof: Assume that $\beta = \emptyset$. Then every node in V is matched to some node in V'. This implies that for each subset $\Sigma_j \in C$, $\exists\ \Sigma_i$ and Σ_k in C such that $\Sigma_i \subset \Sigma_j \subset \Sigma_k$; but, this implies that the relation R is circular, which is a contradiction.

Corollary C.1.1 $\gamma \neq \emptyset$ *and* $|\gamma| = |\beta|$

Proof: By the same argument used for lemma C.1.3 above, $\gamma \neq \emptyset$. By definition of bipartite matching, it follows that the number of unmatched nodes in V must be equal to the number of unmatched nodes in V'.

Consider an arbitrary matching M and the sets β and γ induced by it. For every $v_i \in \beta$, let $T(v_i)$ be the set of nodes v_k such that there is a sequence $v_{k_1}, v_{k_2}, \ldots, v_{k_m}$ where $v_i = v_{k_1}$, $v_{k_m} \in \gamma$, and $(v_{k_i}, v'_{k_{i+1}}) \in M, 1 \leq i \leq m-1$. A function τ is defined on each element $\Sigma_i \in C$ where $v_i \in \beta$ as follows:

$$\tau(\Sigma_i) = \{f^{-1}(v_k) : v_k \in T(v_i)\}$$

A subset Σ_i corresponding to a base node v_i is referred to as a *base subset* and Σ_i is said to define the set $\tau(\Sigma_i)$.

Lemma C.1.4 *Let* $P = \{\tau(\Sigma_i) \mid \Sigma_i = f^{-1}(v_i) \text{ where } v_i \in \beta\}$ *then P is a partition of C and the subsets contained in each element $p \in P$ can be linearly ordered by the relation R.*

1. Given an arbitrary subset $\Sigma_k \in C$, it is included in one of the (*set*) elements of P.

   Proof: If Σ_k is a base subset then it is clearly included in $\tau(\Sigma_k) \in P$. If Σ_k is not a base subset then its corresponding node v'_k has a predecessor in M, say v_j, corresponding to a subset Σ_j. Since R is transitive and non-circular then so is M. Therefore, there exists a base subset Σ_i such that $(\Sigma_i, \Sigma_k) \in M^+$. Hence, Σ_k is contained in an element of P.

2. Given two base subsets Σ_i and Σ_k, if $i \neq k$ then $\tau(\Sigma_i) \cap \tau(\Sigma_k) = \emptyset$.

   Proof: Lemma C.1.1 and C.1.2 and definition of τ.

3. The total order of the elements of $\tau(\Sigma_i)$ can be determined from the ordering of the corresponding sequence of $T(v_i)$.

(1), (2) and (3) above imply that P is a partition of C and the subsets contained in each $p \in P$ can be linearly ordered according to R.

Lemma C.1.5 *Let P be a partition of C such that the elements of each $p \in P$ can be linearly ordered, then P corresponds to a matching in G.*

Proof: Consider each ordered sequence $p = [\Sigma_{k_1}, \Sigma_{k_2}, \ldots, \Sigma_{k_m}]$ in turn. Include the set of edges $\{(v_{k_i}, v'_{k_{i+1}}) \,:\, 1 \leq i \leq m-1\}$ in the matching.

Theorem C.1.1 *P is a partition of C whose elements can be linearly ordered based on the relation R if and only if it is induced by a matching on the corresponding graph G.*

The proof follows from lemmas C.1.4 and C.1.5.

C.1.2 Weight Function

A weight function W is defined on edges of G as follows:

$$W(v_i, v'_j) = |\Sigma_j|$$

Applying a maximum weighted bipartite matching algorithm to G yields a matching M that induces a set of base nodes: β, such that the total weight of M is:

$$\sum_{(v_i, v'_j) \in M} |\Sigma_j| = \sum_{v_j \notin \beta} |\Sigma_j|$$

Let σ be the total sum of the lengths of the elements of C:

$$\sigma = \sum_{i=1}^{|C|} |\Sigma_i|$$

Let κ be the sum of the lengths of the base subsets. κ can be computed given σ and the weight of the matching M, as follows:

$$\kappa = \sigma - \sum_{v_j \notin \beta} |\Sigma_j|$$

Recall that the elements of the base subsets are concatenated to construct the string w. Therefore, since σ is constant and the sum of the weights of the *non-base* subsets is maximized by the maximum weighted bipartite matching algorithm, it follows that κ is minimized. Hence, the final result which is restated as follows:

Theorem C.1.2 *$min\, \kappa \iff min\, |w|$*

The proof follows from Theorem 1 and maximum weighted bipartite matching.

C.2 Complexity

Assume that one can test whether or not a given set is a proper subset of another set in unit time. This is a reasonable assumption if the length of Σ is not too large and bit-strings are used to represent the subsets. With that assumption, the construction of the graph G from the collection C requires $O(n^2)$ operations.

Tarjan in [7] showed that the maximum weighted bipartite matching problem can be reduced to the max-flow/min-cost problem with no negative cost cycles. He also presented an algorithm for finding a minimum cost maximum flow f^* in a network with n nodes and m edges in $O(m|f^*|\log_{(2+m/n)} n)$ time. In this case, the cost of constructing the graph is dominated by the max-flow/min-cost algorithm and $|f^*| = n/2$. Hence, the total running time is:

$$O(nm \log_{(2+m/n)} n) \quad \square$$

C.3 Experimental Results with OP

Experimental results with the *t_symbols* maps of the six programming language grammars mentioned in Appendix A are presented in figure C.1.

Grammar	$\lvert\Sigma\rvert$	*t_symbols*	merged	Heuristic	OP
Pascal	80	1051/193	425/73	173/29	172/28
Pascal1	63	1036/189	423/72	172/29	171/28
Pascal2	63	1040/191	425/73	172/29	171/28
C	85	2598/206	1153/88	478/32	451/30
Ada	96	2847/516	1229/180	545/67	538/62
Sedl	123	11855/824	3465/266	1465/91	1429/84

Figure C.1: Experimental results

In the column headed *t_symbols*, two numbers are specified for each grammar. The first number indicates the sum of the sizes of the sets in the range of *t_symbols* and the second number indicates the number of elements in the map. Given a *t_symbols* map, before OP is applied to it, the sets in its range that are identical are merged. In the column headed "merged", the first number specified for each element indicates the sum of the sizes of the sets in the merged map and the second number indicates the number of merged sets. The column headed "Heuristic" shows the length of the resulting string and the number of bases in the partition obtained using the heuristic algorithm described in section 5.2.3.3. The column headed "OP" shows the length of the resulting string and the number of bases obtained using OP.

Appendix D

Statistics

The table of Figure D.1 shows some statistics about the 6 grammars described earlier and their LALR(k) parser constructed with this method. The information in the table is self-explanatory. In specifying storage values, an 8-bit byte machine is assumed. That is, integer values less than 255 are assumed to be stored in a single byte; integer values that are greater than 255 are assumed to be stored in two bytes.

	Pascal	Pascal1	Pascal2	C	Ada	Sedl
Terminals	63	63	63	86	96	124
Nonterminals	110	111	111	98	304	362
Productions	213	215	215	266	523	746
Storage for Rules	**424**	**428**	**428**	**530**	**1566**	**2235**
Items	626	625	627	786	1527	2398
Scopes	13	13	13	30	25	93
States	193	189	191	206	516	824
Look-ahead states	0	1	5	0	0	0
Shift actions	396	393	393	349	593	1987
Goto actions	336	332	334	975	884	2250
Shift/Reduce actions	329	324	326	987	700	5755
Goto/Reduce actions	574	569	569	322	1779	4360
Reduce actions	326	320	390	1262	1554	4113
Goto entries removed by default	245	244	245	915	705	1763
Goto/Reduce entries removed by default	484	481	481	270	1592	3958
Non-terminals eliminated by default	60	62	61	52	216	206
Length of GOTO_CHECK Table	485	477	480	417	1187	2076
Length of GOTO_INFO Table	374	365	368	318	882	1713
Useful entries in GOTO_INFO	374	365	368	318	882	2076
Storage for compressed GOTO	1455	1431	1440	1251	3561	1713
Terminal states after merging	130	128	133	159	356	513
Shift actions removed by merging	456	450	451	541	679	5484
Reduce actions removed by merging	30	47	48	109	579	1450
Reductions removed by default	263	261	326	1143	908	2544
Length of ACTION_CHECK Table	475	471	490	1076	1133	3014
Length of ACTION_INFO Table	428	420	436	1049	1037	2951
Useful entries in ACTION_INFO	412	420	417	964	1037	2890
Storage for compressed ACTION	1331	407	1362	3174	3207	8916
Storage for Parsing Tables	**2786**	**2742**	**2802**	**4425**	**6768**	**15144**

Actions in Compressed Tables:

Shifts + Shift/Reduces + LA Shifts	269	268	273	795	614	2258
Gotos + Goto/Reduces	181	176	177	112	366	889
Reduces	55	53	62	90	212	343

Storage for Error maps:

t_symbols map	399	392	394	928	1651	3188
nt_symbols map	328	323	327	346	1636	2578
SCOPE map	282	280	281	442	780	2843
Required storage for Error maps	**1009**	**995**	**1002**	**1716**	**4067**	**8609**
Unoptimized NAME map	2721	2728	2728	2786	8899	9491
Optimized NAME map	**1832**	**1821**	**1821**	**2477**	**3859**	**6365**

Total Storage Requirement for Parsing and Error Recovery Tables

(does not include NAME map)	**4704**	**4642**	**4712**	**7088**	**13588**	**28064**

Figure D.1: Parser Statistics