Very Fast LR Parsing

Thomas J. Pennello
MetaWare Incorporated
412 Liberty Street
Santa Cruz, CA 95060

Abstract

LR parsers can be made to run 6 to 10 times as fast as the best table-interpretive LR parsers. The resulting parse time is negligible compared to the time required by the remainder of a typical compiler containing the parser.

A parsing speed of 1/2 million lines per minute on a computer similar to a VAX 11/780 was achieved, up from an interpretive speed of 40,000 lines per minute. A speed of 240,000 lines per minute on an Intel 80286 was achieved, up from an interpretive speed of 37,000 lines per minute.

The improvement is obtained by translating the parser's finite state control into assembly language. States become code memory addresses. The current input symbol resides in a register and a quick sequence of register-constant comparisons determines the next state, which is merely jumped to. The parser's push-down stack is implemented directly on a hardware stack. The stack contains code memory addresses rather than the traditional state numbers.

The strongly-connected components of the directed graph induced by the parser's terminal and nonterminal transitions are examined to determine a typically small subset of the states that require parse-time stack-overflow-check code when hardware does not provide the check automatically.

The increase in speed is at the expense of space: a factor of 2 to 4 increase in total table size can be expected, depending upon whether syntactic error recovery is required.

1. Introduction and Background

In 1982 this author's firm consulted with another attempting to replace their old COBOL compiler with a new one. The old one was implemented entirely in assembly language and used recursive-descent compiler technology. The new one sported many new language features and bug fixes, was far easier to maintain, and used modern techniques, such as a machine-generated scanner and LR parser, fully automatic syntactic error recovery, and table-driven code generation.

The new compiler was many times slower than the old one. Although the principal reason was the lumbering code generation algorithms, it was noticed that the old compiler's parser was much faster than the new parser. The old parser's current input symbol was held in a register and the next parsing action was determined by a usually very short series of register-constant comparisons. Procedure call overhead was minimized by using a jump-to-subroutine instruction instead of the high-overhead full procedure call instruction.

The LR parser in the new compiler interpreted tables, as is standard, and it was written in PL/I rather than assembly language. The tables were comprised of seven arrays of numbers, following a design almost identical to that used in YACC [Joh 75]. Shifting a terminal or nonterminal almost always required just three or four array accesses to determine the next state. But although the LR parser was compiled with a reasonable optimizing compiler, it still took 50 machine instructions to effect a terminal transition, compared to an average of perhaps 6 instructions for the recursive-descent assembly-language parser.

This author made a successful effort to obtain the same assembly language efficiency for LR parsers. A speed-up factor of 10 was achieved on the client's processor (which is roughly equivalent to a VAX 11/780), resulting in a parsing rate of nearly 1/2 million lines per minute. A large COBOL grammar was used for the testing.

Recently we have produced similar results on an Intel 80286, obtaining a speed-up factor of 6.5, resulting in a parsing rate of 240,000 lines per minute. This paper reports specific statistics about the 80286 implementation and describes the speed-up technique in enough detail that the reader could implement it on any particular hardware. (Our measurements of parsing time exclude the time taken to read and scan the input being parsed, but include procedure call and similar overhead necessary to obtain the next input symbol.)

The end result is that the time required for LR parsing can be reduced to near insignificance. Since LR parsers are now being used for more than just programming-language phrase-structure checking — e.g., code generation [GG 78] — the result can be used to speed up more than just the syntactic analysis phase of a compiler.

The remainder of this paper is organized as follows. Section 2 explains the standard interpretive LR parsing algorithm used in the experiments, and gives CPU instruction counts for parser shift and reduce actions for purposes of comparison with the assembly-language parser techniques. Section 3 explains how to encode an LR parser in assembly language, using an idealized, fictitious machine in the interest of avoiding unnecessary details. Section 4 shows how to avoid parse stack overflow without checking the stack depth at each state. Section 5 discusses the impact on syntactic error recovery; specifically, we describe how the parser's stack of machine-code addresses can be translated back into state numbers so that traditional error recovery schemes still work. Section 6 details some specific results, and Section 7 summarizes. References follow in Section 8.

2. Table-Interpretive LR Parsing

An LR parser is conventionally implemented using a "driver" routine containing a loop. Each time around the loop a single parse action is made: a terminal shift, a reduction, accepting the input, or signaling an error. To determine the next action, the state on the top of the parse stack and the current input symbol are considered.

The table encoding scheme used in YACC [Joh 75] produces the smallest tables and affords the fastest parsing times we know of, notwithstanding recent results on parse table optimizations [DDH 84].

Almost always, three or four array accesses suffice to determine the next action: three for each shift, and four for each reduction. Three or four array accesses are needed to determine the next state after a reduction has been made. Only multiply-inconsistent states (those from which more than one reduce action is possible) require list searching to determine the next action; such states are rare, if occurring at all, in programming language grammars (22 states out of 590 for a 329-production grammar for C that we use, where the longest list has six elements). Counted array accesses exclude the access to the parser's stack.

Determining the next shift or reduce action involves roughly the following code:

```
q := Stack[SP];
Next := Comb[Tbase[q] + Input_symbol];
if Accessing_symbol[Next] = Input_symbol then
   ... action is a shift to Next.
else begin
   D := Default[q];
   if D <= Last_production then
      ... action is a reduce by production D.
   else   { rare case }
      ... search a list for one of
      ... several productions to reduce by.
   end;
```

When a reduce action is determined, the next state is determined as follows:

```
LP := Left_part[Production];
SP := SP - Right_part_length[Production];
q  := Stack[SP];
Base := NTbase[LP];
Next := Comb[Base+q];
if Accessing_symbol[Next] <> LP then Next := Comb[Base];
SP := SP+1;
Stack[SP] := Next;
```

Some of the details have been glossed over for simplicity; but the reader can see that the access is efficient. The basic idea behind the YACC technique is to merge the rows of the terminal-transition matrix into a single array, Comb, where the row for a particular state q starts at Tbase[q] in the Comb. The columns of the nonterminal-transition matrix are merged into that same array, after deleting the most popular entry in the column and placing it at the top of the column, extended by one row upwards. The column for a particular nonterminal N starts at NTbase[N].

Validity of an entry is determined by checking that the destination state of a transition on a symbol V is in fact accessed by V (each state in an LR parser can be "accessed" by only a single symbol V — that symbol preceding the LR "dot" in the state's item set [ASU 86]). For a state having at most one reduction possible, the array Default encodes the production to be reduced by; more than one possible reduction requires searching a list, but this is rare.

In our own implementation, 40 machine instructions are necessary to decode and complete a shift transition on an 80286. This includes parse stack access and overflow checking, but excludes the call to the Scanner. Through some parse state sorting, we can sometimes decode a reduce action in 10 instructions, but in the standard YACC table organization 41 instructions are required, and 52 more are needed to complete it, for a total of 93 instructions. This excludes the case requiring the list search.

The point is that even though table access time is usually constant, the number of instructions required for a parser action is high. The next section shows that by careful use of a machine architecture, a complete shift transition or complete reduce transition can be made in far fewer instructions.

3. Tables in Assembly Language

In assembly language, each state n is represented by two code addresses, which we label Q_n and $NTXQ_n$. Pictorially, here is what a state looks like; an explanation follows:

```
Qn:     call Scan   ; Only if accessed by a terminal.
        push address of label NTXQn (below) on machine stack
        ; Make next-action decision:
        ... if shift, jump to destination state Qm ...
        ... if reduce by production p,
                    jump to reduce-handler for p ...
        ... otherwise, jump to error handler ...
NTXQn:  ... determine destination state m on current left part
            and jump to Qm ...
```

Q_n is where entry is made into the state when the symbol V accessing state n is being (conceptually) shifted on the parse stack, whether V is a terminal or nonterminal. This portion of the code is responsible for calling the scanner to obtain the next input symbol, if V is a terminal, and for determining the next action — whether shift or reduce.

A shift action is effected by a simple jump to the destination state m (label Q_m). When a reduction by production p is indicated, a jump is made to a single common routine handling p, since a reduction by p may occur in several states. Control is transferred to $NTXQ_n$ only after the right part of a production has been popped from the stack as a consequence of a reduce action, and the top state is revealed to be n. Code at $NTXQ_n$ is responsible for determining the destination state on the left part of the production. Essentially, at Q_n the next action is determined — *shift or reduce* —, and at $NTXQ_n$ *a reduction is completed.*

In the interpretive scheme of Section 2, the parser's stack is consulted both for determining the next action, and for determining the new state following a reduction. In the assembly-language encoding, the stack is unnecessary for determining the next action, since the location in the code implicitly determines the state (i.e., which Q_n the CPU's instruction pointer is at). Instead, it is used only for determining the destination of a nonterminal transition after a reduction.

Therefore, the stack consists only of the $NTXQ_n$ code addresses. After the right part of a production has been popped, a jump to the location contained at the stack top transfers control to the code that completes the reduction by shifting the left part. The $NTXQ_n$ code address for state n is pushed on the stack when n is *entered at* Q_n.

The stack can be implemented using the machine's standard hardware stack, if it has one. If a stack is lacking, a register can be dedicated as a pointer to a memory stack. Stack overflow is treated in Section 4.

The determination of the next action from a state q is done by comparing the current input symbol T with all terminal symbols valid from q. To speed up the testing, T is held in a machine register, and a series of register-constant comparisons followed by a conditional jump suffice.

If no terminal transition is taken, but it is possible to reduce by exactly one production p in q, we need not check that T is in the look-ahead set of p in q, since the reduce action can serve as a default action after all terminal shift transitions have been tested. (This has a minor effect on error recovery; see Section 5.) Call this ignoring of the look-ahead set the *defaulting assumption.*

Similarly, determination of the next state for a reduction is done by comparing the left part, held in a register, with the possible shiftable nonterminals. We can use the fact that a nonterminal transition must always succeed in an LR parser to reduce the number of comparisons.

For example, if only three nonterminals can be shifted by a state, a single compare to the middle of the three can be made, followed by a jump-if-less to the first of the three, a jump-if-equal to the middle, and an unconditional jump to the third. The most advantageous scheme here is to use a binary search, the leaves of which take maximum advantage of using a single comparison to determine one of three destinations.

Here is an example of a state accessed by a terminal symbol and having nine terminal transitions on symbols 'a'..'i', a single optional reduction, and ten nonterminal transitions, on nonterminals A..I.

```
Q23:    call Scan     ;Since accessed by a terminal.
        push address of NTXQ23
        cmp R_T,4     ;If 'a'
        je  Q11         ;Shift to 11 on 'a'
        cmp R_T,5     ;If 'b'
        je  Q12         ;Shift to 12 on 'b'
        cmp R_T,6     ;If 'c'
        je  Q13         ;Shift to 13 on 'c'
        cmp R_T,7     ;If 'd'
        je  Q14         ;Shift to 14 on 'd'
        cmp R_T,8     ;If 'e'
        je  Q15         ;Shift to 15 on 'e'
        cmp R_T,9     ;If 'f'
        je  Q16         ;Shift to 16 on 'f'
        cmp R_T,10    ;If 'g'
        je  Q17         ;Shift to 17 on 'g'
        cmp R_T,11    ;If 'h'
        je  Q18         ;Shift to 18 on 'h'
        cmp R_T,12    ;If 'i'
        je  Q19         ;Shift to 19 on 'i'
        jmp NSQ20     ;Reduce by production 20

NTXQ23: ; Binary search for nonterminal transitions:
        cmp R_NT,19   ;If E
        ja  L1
        cmp R_NT,16   ;If B
        jl  Q33         ;Shift to 33 on A
        je  Q25         ;Shift to 25 on B
        cmp R_NT,18   ;If D
        jl  Q26         ;Shift to 26 on C
        je  Q27         ;Shift to 27 on D
        jmp Q28         ;Shift to 28 on E
L1:     cmp R_NT,21   ;If G
        jl  Q29         ;Shift to 29 on F
        je  Q30         ;Shift to 30 on G
        cmp R_NT,23   ;If I
        jl  Q31         ;Shift to 31 on H
        je  Q32         ;Shift to 32 on I
        jmp Q24         ;Shift to 24 on K
```

Figure 1. Sample state with nine terminal transitions, one default reduction, and 10 nonterminal transitions.

Here, "je" means "jump if equal", "jl" "jump if less", and "cmp" "compare". Notice how a maximum of three comparisons are necessary for the ten nonterminal transitions. Assuming all nonterminals are equally likely, an average of 2.9 comparisons are necessary, and an average of 7.3 instructions executed (some of which are conditional jumps that are not taken) to actually transfer control to the destination state. For the terminal transitions, averages of five comparisons and ten instructions are needed.

The binary search scheme works well for nonterminal transitions, since the assumption of no error reduces the number of comparisons. For the terminal transitions, binary search would require that each leaf of the search tree contain the default action for the state (reduce or error), thus requiring a copy of the code for this action at each leaf; for these reasons linear search is preferable. However, binary search can be employed to search through the set of possible reductions in a multiply-inconsistent state after no terminal shift transition has been found since, under the defaulting assumption, at least one reduction applies.

It is possible, however, that the binary or linear search time can exceed interpretive parse tables decoding time when the number of transitions becomes large. In such infrequent cases, we revert to interpretive parse tables encoding. See **"States with a large number of transitions"** below for more details.

If p is entered via a terminal transition (Q_p entry), then, like all non-reduce states, p calls the scanner to obtain the next input symbol. Next, we *avoid* the standard action of a non-reduce state in pushing the address of the NTXQ entry point, leaving the parse stack *one short* of the right part length. The left part is loaded into a register and a jump is made to a routine that handles productions of length one less than the length of p. Thus we have optimized away a push and, sometimes, its corresponding pop located in the jumped-to reduce routine (if the right part length is greater than one, a pop is still required to pop the remainder of the right part).

The code at $Reduce_i$, for i = 1, 2, 3, ... that handles reductions is as follows:

```
Reduce_i: add sp,Element_len*i ; Pop the stack.
          jmp @(sp)
          ; Jump to nonterminal transition
          ;    portion of top state.
```

where Element_len is the length of a stack element (two bytes on an 80286) and by "@(sp)" we mean an indirect reference to the location at the top of the stack. We assume the stack grows downward in memory; hence the "**add** sp,..." to pop the stack.

When a production's right part length exceeds four, a separate register is loaded to contain the right-part length (there is nothing magic about the length four; we chose it because longer productions are infrequent in programming-language grammars).

The code necessary to perform a reduction is generated once per production, to conserve space, and is done as follows. First, the states are ordered so that the first P states of the parser are the *reduce states*, where P is the number of productions. The p^{th} reduce state consists only of the LR item for production p with the dot at the end — thus the only action permissible in state p is to reduce by production p. Therefore, all states numbered P+1 and above have either terminal shift transitions or multiple reductions possible.

Pictorially, here is what a reduce state looks like in assembly language; an explanation follows:

```
; Production numbered p;
;     Left part is LP; right part length is L.
Q_p:    call Scan       ;Only if accessed by a terminal.
        mov  R_NT,LP    ;Entry via transition.
        jmp  Reduce_L-1 ;Pop L-1 elements and reduce.
NSQ_p:
        mov  R_NT,LP    ;Entry via reduction.
        jmp  Reduce_L   ;Pop L elements and reduce.
```

Figure 2. General form of a reduce state.

The reduce states can be entered in two ways: first, by a normal terminal or nonterminal shift transition to the state; second, by a jump when a reduction is determined in a non-reduce state. These entries are labeled Q_p for the normal entry and NSQ_p ("NS" = "non-shift") for the reduction entry.

If reduce state p is entered via a reduction action (NSQ_p entry), the left part of p is loaded into a register and a jump is made to a routine to handle the remainder of the reduction. The production length is implied by the particular routine jumped to. A separate routine exists for productions of length 0 through 4, covering the vast majority of productions. For productions longer than that, another register can be loaded with the right part length and a more general routine jumped to.

For example:

```
Q48:    mov R_NT, 66     ;Entry via transition.
        mov R_RPL, 5
        jmp ReduceN      ;Pop 5 elements and reduce.
NSQ48:  mov R_NT, 66     ;Entry via reduction.
        mov R_RPL, 6
        jmp ReduceN      ;Pop 6 elements and reduce.

ReduceN: shl R_RPL, 1    ;Multiply by Element_len,
        add sp, R_RPL    ;   assumed 2.
        jmp @(sp)
```

The left shift scales the right-part length before adding to sp. This scaling could instead be done before the jump to ReduceN. However, in practice, the parse stack is usually moving in parallel with another "semantics stack" or other associated stacks, and the right-part length must be known accurately for the other stacks. In such a case, the optimized jump to $Reduce_{L-1}$ in Figure 2 above is replaced by a jump to a different routine Reduce_minus_1_L so that the right-part length is accurately known as L, thus permitting proper manipulation of other stacks.

States with a large number of transitions. It is possible that the binary or linear search time can exceed interpretive parse tables decoding time when the number of transitions becomes large. In such infrequent cases, a vector is indexed to obtain constant-time look-up for a transition.

First, we treat the case of nonterminal transitions. For each state q with "too many" nonterminal transitions, the transitions are laid out in a row R[1..#Nonterminals] indexed by nonterminal, with R[N] = "blank" where q has no nonterminal transition on N. All such rows are folded into a single vector NT_next, which is initially all "blank". A row can be folded into NT_next at location i iff for every non-blank R[j], NT_next[i+j−1] is blank.

The rows with fewest blank entries are folded first, and those with the most last. This permits the mostly blank rows to "fill in the cracks" in NT_next.

Non-blank state entries in NT_next are represented as the Q_n address of the state, not as the state number. The nonterminal transitions of a state can then be implemented by simple code that transfers control to the destination state:

```
NTXQ23: ; Table look-up for nonterminal transitions:
        add R_NT, Base
        jmp @NT_next[Base]
```

Figure 3. Nonterminal transitions of Figure 1 using table decoding instead of binary search.

Here NT_next[Base+1] is where state 23's R[1] has been folded; Base is known at assembly-language generation time and hence is a constant in the code. The base is added to the nonterminal, and the result is used to perform an indirect jump (@) through the appropriately indexed NT_next vector. Since there is no possibility of error for a nonterminal transition, the indexing must always extract a valid destination state.

(On the 80286 the code is more involved. There, a register is loaded with the value Base and a jump is made to a common routine that does the array index and indirect jump. The decision to use a routine is, of course, entirely architecture-dependent.)

This scheme is so efficient that it could perhaps always replace binary search. However, the nonterminal transitions do not pack well at all by row, so that the speed is gained only by yet more sacrifice in space. Furthermore, the array indexing and memory access can be more expensive than a state with very few nonterminal transitions — especially a state with a single nonterminal transition, whose code at $NTXQ_n$ is simply "**jmp** ...".

Turning to the terminal transitions, the situation is more complicated due to the need to check for error. Here, a T_next and a T_check array are used, both of the same size. For each state q with "too many" terminal transitions, the transitions are laid out in a row R[1..#Terminals] indexed by terminal, with R[T] = "blank" where q has no terminal transition on T. All such rows are folded into T_next, which is initially all "blank". A row can be folded into T_next at location i iff for every non-blank R[j], T_next[i+j−1] is blank.

For each state q in T_next, T_check contains the accessing terminal for q; otherwise the T_check entries are some value that never compares equal to any terminal. The terminal shift transitions of a state are replaced by a call to a routine that, if it finds a valid terminal transition, does not return but instead jumps to the appropriate state. If the routine finds no valid terminal transition, it returns to the calling state where the default action is taken. For example:

```
Q23:    call Scan    ;Accessed by a terminal.
        push address of NTXQ23
        mov  Base_reg, Base
        call Find_T_transition
        ; Return here if no T transition found.
        jmp  NSQ20   ;Reduce by production 20.
```

Figure 4. Terminal transitions of Figure 1 using table decoding instead of binary search.

The routine Find_T_transition is as follows:

```
Find_T_transition:
        add  Base_reg, R_T
        cmp  T_check[Base_reg], R_T
        jne  No_match
        pop  Scratch_reg ; Discard return address.
        jmp  @T_next[Base_reg]
No_match: return
```

Here, if Find_T_transition finds a valid transition, the return address of the calling state is discarded, effectively turning the original call into a jump instruction. Then, an indirect jump is made through T_next to the destination state.

Find_T_transition is involved enough that the normal linear search is often faster. The break-even point is of course architecture-dependent. On the 80286, we use table look-up when the number of terminal transitions exceeds nine, and when the number of nonterminal transitions exceeds twenty-four.

Span-dependent instructions. Conditional jumps on many architectures have a limited range. On the 80286, for example, that range is −128..+127 bytes from the byte following the jump. Some assemblers automatically convert a conditional jump whose target T is too far away into a conditional jump around a non-conditional, full-span jump:

```
je Qn   =>      jne Next
                jmp Qn
          Next:
```

Relying on this feature of an assembler makes the generation of the assembly language easy, but can potentially degrade the performance of the parser if a taken conditional jump takes more time than a non-taken conditional jump, which is usually the case (over twice as long on the 80286). It is then preferable to instead issue a conditional jump to a nearby full-span jump. The terminal transitions of Figure 1 become

```
        cmp  R_T,4   ;If 'a'
        je   L11       ;Shift to 11 on 'a'
        cmp  R_T,5   ;If 'b'
        je   L12       ;Shift to 12 on 'b'
        cmp  R_T,6   ;If 'c'
        je   L13       ;Shift to 13 on 'c'
        cmp  R_T,7   ;If 'd'
        je   L14       ;Shift to 14 on 'd'
        cmp  R_T,8   ;If 'e'
        je   L15       ;Shift to 15 on 'e'
        cmp  R_T,9   ;If 'f'
        je   L16       ;Shift to 16 on 'f'
        cmp  R_T,10  ;If 'g'
        je   L17       ;Shift to 17 on 'g'
        cmp  R_T,11  ;If 'h'
        je   L18       ;Shift to 18 on 'h'
        cmp  R_T,12  ;If 'i'
        je   L19       ;Shift to 19 on 'i'
        jmp  NSQ20   ;Reduce by production 20
L11:    jmp  Q11
L12:    jmp  Q12
L13:    jmp  Q13
L14:    jmp  Q14
L15:    jmp  Q15
L16:    jmp  Q16
L17:    jmp  Q17
L18:    jmp  Q18
L19:    jmp  Q19
```

Figure 5. Terminal transitions of Figure 1 with single jumps expanded to jump pairs.

Two jumps taken when a match is found are cheaper than taking a jump for each comparison that fails.

Summary. Below is a pictorial summary of the various strategies.

Non-reduce state n:

Q_n: call Scan ; Only if accessed by a terminal.

---

push address of label $NTXQ_n$ on machine stack

---

linear search for terminal transition
or
mov Base_reg,Base
call Find_T_transition

---

jump to default reduce state
or
linear search for default reduction (rare)
or
binary search for default reduction (rare)
or
jump to error handler if no reductions possible

$NTXQ_n$: binary search for default reduction
or
in-line code to extract transition from NT_next

Reduce state for production p:

Q_p: call Scan ; Only if accessed by a terminal.

---

mov R_NT,Left_part

---

jmp $Reduce_{L-1}$
or
jmp $Reduce\_minus\_1_L$
or
mov R_RPL,L−1
jmp ReduceN
or
mov R_RPL,L
jmp Reduce_minus_1N

NSQ_p: mov R_NT,Left_part

---

jmp $Reduce_L$
or
mov R_RPL,L
jmp ReduceN

Figure 6. Possible forms of states in the assembly-language parser.

4. Stack-Overflow Checking

A interpretive LR parser typically checks for stack overflow each time a state is pushed on the stack. In an assembly-language parser, the run-time stack must likewise be guarded. Some architectures provide this checking, but, e.g., an 80286 running in non-protected mode does not. Furthermore, there may be other non-hardware stacks (such as a so-called "semantics stack") running in parallel with the parse stack that might need to be expanded as the parse stack deepens.

Therefore it may be necessary to generate stack-checking code in the assembly-language parser. However, unlike the interpretive parser, it is prohibitively expensive in code space to insert such a check at each state (Q_n entry). Furthermore, checking the stack at each state wastes time. What is preferred is a way to check the stack only at certain key states, in such a way that the stack always has room enough for the parse to proceed until the next state with a check is encountered.

In general, an LR parser's stack can be arbitrarily deep (if one can prove that its size is bounded by a constant, the language being parsed is regular and does not require a context-free parser!). The unlimited depth comes from loops in a directed graph G induced by the transitions, whose vertices are the parser states and whose edges are labeled with terminals and nonterminals.

A set of key states can be determined by discovering the strongly-connected components (SCCs) of G. An SSC is a maximal subset of vertices such that there exists a path from any vertex in the subset to any other.

For each SCC S, choose a subset of the vertices of S such that when that subset is removed, S becomes acyclic. The union U of these subsets for all such S, when removed from G, makes G acyclic. A stack check — implemented by a call to a single stack-check routine S_Check — is then inserted at the Q_n entry of each state in U, just before the address of $NTXQ_n$ is pushed. Call states that have stack checks *checking states*.

Once this has been done, the parse stack can grow at most a finite amount D before another stack check is imposed. D is the maximum distance between two vertices in the now-acyclic G, and can be determined when the assembly-language tables are generated. It is sufficient, then, that S_Check ensure that D stack cells are available for future use. When fewer than D cells are available, the stack can potentially overflow in the future, and S_Check should terminate processing, or expand the stack.

In practice D is small — perhaps no more than 20. See Section 6 for specific values of D and numbers of SCCs for some grammars tested. For computing SCCs we use an algorithm given in a paper by DeRemer and Pennello [DeP 82].

It is also possible to fix D *a priori* for all parsers — say, to 50 — by further breaking any path in G longer than 50. S_Check is then independent of the particular tables, always checking for 50 cells remaining.

In choosing a subset S' of the vertices of an SCC S such that S−S' is acyclic, a simple — although not optimal — algorithm consists in removing one of the vertices and its connecting edges from S, and applying the same SCC analysis and loop-removal process to the result, breaking SCCs wholly contained within it. Repeat the process until no SCCs remain. When finished, the

original SCC S is entirely "broken" — sufficient vertices have been removed from it to make it acyclic. At each removed vertex a stack check is placed.

An non-linear but polynomial time analysis of the SCC's vertices can of course yield the smallest set of vertices that when removed make S acyclic, but we used the simple algorithm in our tests. (Open problem: Is there a linear-time algorithm?)

5. Error Recovery

Error recovery techniques for LR parsers normally assume, of course, access to the parser's state stack. Unfortunately, the state stack of the assembly-language parser does not contain state numbers directly useful for recovery.

However, there is a close correspondence between the assembly-language parser's stack of code addresses and the traditional LR parser's state stack, and the conversion from one to another can (almost) be made by the inclusion of a single additional assembly-language table T. Using T, the assembly parser's stack is first converted into the traditional stack, error recovery is invoked to perform a repair, then the (potentially) modified stack is converted back into state addresses.

Let "AP" stand for assembly parser and "state-numbered stack" for the traditional LR parser's stack in what follows.

The AP stack consists entirely of NTXQ code addresses. To convert the AP stack to a state-number stack, T need only be a table of these addresses, indexed by state number.

To save space we can store the differences between adjacent NTXQ labels rather than the addresses themselves; typically the difference is less than 256 bytes. Even if it exceeds that amount, only two bytes per T entry would be needed rather than the four bytes occupied by a code address on 32-bit computers.

Since the first state on the AP stack is known to be that of the (fixed) start state S, the state number corresponding to arbitrary AP stack value V is the state q = S + I such that V is the NTXQ address for S plus the first I entries in the difference table T. Although this can involving scanning all of T to convert a single address, efficiency is not an issue for presumably infrequent syntactic error recovery.

After error recovery is through modifying the state-number stack, the conversion back to the AP stack is not as simple. All states in the state-number stack correspond to the AP NTXQ code addresses, except possibly the top state. The reason the top state is different is that while the state-number stack parser uses states below the top state for nonterminal transitions only, it uses the top state for both terminal and nonterminal transitions. The AP, however, uses the AP stack only for nonterminal transitions.

If error recovery expects the parse to resume by considering the transitions on the input terminal from the top state, the AP stack is insufficient. If the state-numbered stack has S states, the first S−1 can be converted to NTXQ addresses using T, and the last must be converted to the terminal shift (Q_n) entry for that state. Processing continues by jumping to that Q_n address with the AP stack containing the S−1 addresses.

To obtain the terminal shift entry, a separate table T' is needed that records differences between the Q_n addresses. In general, T and T' look like this:

T:

$NTXQ_{n+2}$	- $NTXQ_{n+1}$	where n+1 is the start state
$NTXQ_{n+3}$	- $NTXQ_{n+2}$	
...		

T':

Q_2	- Q_1	
Q_3	- Q_2	
...		
Q_n	- Q_{n-1}	where there are n reduce states
Q_{n+1}	- Q_n	
Q_{n+2}	- Q_{n+1}	
...		

The two tables could in fact be combined, giving a better chance that the entries will be only a byte, by storing the successive differences between a Q_n, $NTXQ_n$, and Q_{n+1} label, rather than storing the difference between Q_n and Q_{n+1}, and separately the difference between $NTXQ_n$ and $NTXQ_{n+1}$. The parse stack conversion is made only a little more complicated.

T and T' combined:

Q_2	- Q_1	
Q_3	- Q_2	
...		
Q_n	- Q_{n-1}	where there are n reduce states
Q_{n+1}	- Q_n	
$NTXQ_{n+1}$	- Q_{n+1}	
Q_{n+2}	- $NTXQ_{n+1}$	
$NTXQ_{n+2}$	- Q_{n+2}	
...		

Here, each reduce state requires one difference and each non-reduce state two.

T' is unnecessary only if error recovery finishes by inserting a nonterminal at the beginning of the input, expecting the parse to proceed by shifting the nonterminal, but with both T and T', any error recovery scheme works.

Parse tables needed by error recovery. Some error recovery schemes do not need direct access to the standard parse tables, but instead try the parser on a set of repairs. The assembly language parser could be used for this at the (considerable) expense of constantly converting parse stack representations.

Instead, if the full parse tables are available, the standard LR parser could be used. It is likely anyway that the error recovery scheme needs access to the parse tables to invent repairs. Therefore the parse tables cannot be gotten rid of, so that the assembly language parser is *in addition* to the standard parse tables.

The effect of default reductions. Since the assembly language parser ignores the look-ahead set of a production p when reducing by p is the only possible reduction in a state, the stack may be somewhat "over-reduced" when an error occurs. That is, checking the look-ahead might have prevented reductions at the point of the error and possibly allowed a better recovery. While the effect on error recovery does show up in practice, it is not significant enough to warrant slowing down the assembly language parser with additional checks.

6. Specific Results

In 1982 tests were made on the proprietary processor mentioned in Section 1 with a large COBOL grammar, and a speed-up factor of 10 was achieved in parsing a COBOL program whose size was in the thousands of lines, with a tripling of the parse table size. Unfortunately, we do not have access to the processor today and cannot precisely document the results.

We made detailed tests on an IBM PC AT containing an 80286 processor running at the standard 6 MHz. A medium-sized grammar was used for testing; it describes the syntax of phrase-

structure affix grammars as accepted by the MetaWare Translator Writing System's [DeP 81] parser generator. The grammar has 45 terminals, 35 nonterminals, and 126 productions. The LALR(1) parser has 192 states, 305 terminal transitions, and 226 nonterminal transitions. The interpretive parse tables occupy 2022 bytes, excluding 931 bytes of tables used for error recovery and for communicating to the scanner the parser's vocabulary of symbols.

Six strongly-connected components were detected in the graph induced by the parser's transitions, and 27 states (out of the 66 non-reduce states) required stack checking. (This high percentage suggests that a more optimal way of breaking up the graph's SCCs may be useful, but we did not analyze in detail the structure of the SCCs to determine if fewer stack checks were possible.) Each stack check ensured room for 9 more states on the stack.

The assembly-language parser occupies a total of 5602 bytes, including 397 bytes of common routines that prefix the generated assembly language and whose purpose is to interface to the scanner and perform reductions. The tables include the table of state entry point differences discussed in Section 5; each table entry is only a byte.

Stripping out the common routines, this represents a factor of (5602−397 +2022)/2022 = 3.6 increase over the original table size. The 2022 is added in the numerator because the regular parse tables are needed for error recovery. If error recovery were unnecessary (as for code generation grammars, where a syntactic error is a code generator failure), the increase would instead be (5602−397−258)/2022 = 2.4 (the 258 subtracts out the difference table, needed only for recovery).

The assembly-language parser was compared in speed to an interpretive LR parser written in Pascal and compiled by a good optimizing Pascal compiler that eliminates common subexpressions within basic blocks. For a 2200 line input program, the Pascal-based parser took 3.57 seconds, parsing 37,000 lines per minute. The assembly-language parser took 0.55 seconds, parsing at 240,000 lines per minute, for a speed-up factor of 6.5. The PC AT has a timing resolution of 0.05 seconds and multiple tests with the same input yielded identical timings. The entire assembly-language parser for the test is 2870 lines long, including 570 lines of the common prefix routines.

Large grammars could not be tested on the 80286 due to the severe limitations of the Microsoft 3.0 Assembler [Mic 84] that was used to assemble the parser. (Apparently only 64K of the 640K memory available could be used by the assembler.) A 158-production grammar for standard Pascal with a few minor extensions yielded a 351-state parser and a 5326-line assembly file, unfortunately well beyond the capabilities of the assembler. 13 strongly-connected components were detected and 39 states out of the 193 non-reduce states had attached stack checks; each check ensured that the stack had room for 17 more elements.

7. Summary

Trying to speed up an LR parser may seem like beating a dead horse, since LR parsers are already reasonably fast. They are traditionally used only for context-free analysis of programs, and thus take up little time in the overall compilation scheme.

But with increased use of LR parsers, such as in code generation, speeding them up can pay off enough to justify the effort to do so, and on today's machines with virtual memory or lots of real memory, the space increase is justified. We have presented a means whereby a speed-up of 6 to 10 can be achieved, at a cost of a factor of from 2 to 4 more in space. With such a speed-up, parsing time is almost negligible.

8. References

[ASU 86] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1985.

[DDH 84] Dencker, Peter, Karl Durre, and Johannes Heuft. Optimization of Parser Tables for Portable Compilers. *ACM Transactions on Programming Languages and Systems 6*, 4 (Oct. 1984), 546—572.

[DeP81] DeRemer, Frank and Thomas Pennello. *MetaWare TWS User's Manual.* MetaWare Incorporated, Santa Cruz, CA, May 1981.

[DeP82] DeRemer, Frank and Thomas Pennello. Efficient Computation of LALR(1) Look-Ahead Sets. *ACM Transactions on Programming Languages and Systems 4*, 4 (Oct. 1982), 615—649.

[GG 78] Glanville, R. and Susan Graham. A new method for compiler code generation. In *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages* (Tucson, AZ, Jan. 23—25), ACM, New York, 231—240.

[Joh 75] Johnson, S.C. Yacc — yet another compiler compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[Mic 84] *Microsoft Macro Assembler User's Guide for the MS-DOS Operating System.* Microsoft Corporation, Bellevue, WA, 1984.

Acknowledgment

Thanks to Frank DeRemer for a careful reading of the manuscript and for valuable input.