Full LR(1) Parser Generator Hyacc
Theory

Xin Chen, Ph.D.
University of Hawai`i

Tutorial at the International Symposium on Code Generation and Optimization (CGO 2012), San Jose, CA
March 31, 2012

Outline

- Overview
- Knuth Canonical LR(1) Algorithm
- Pager's LR(1) Practical General Method
- Pager's LR(1) Lane-tracing Algorithm
- Pager's Unit Production Elimination Algorithm
- Extension to Pager's Unit Production Elimination Algorithm
- Edge-pushing LR(k) algorithm
- $\mathrm{Thead}_k(\alpha)$: A New $\mathrm{First}_k(\alpha)$ Algorithm
- Summary

I. Overview - Terminology

- Grammar: A grammar for a language L is a 4-tuple G = (N, ∑, P, S).
  - N : nonterminal symbols
  - ∑ : terminal symbols
  - P : productions
  - S : start symobl
- A configuration (= item) : $A \rightarrow \alpha \bullet X \beta, \psi$
  - Marker : $\bullet$
  - Scanned symbol : X
  - Production : $A \rightarrow \alpha \bullet X \beta$
  - Context : ψ = First (β)
  - Transition symbol : X
- State (= item set) : a state contains one or multiple configurations.
- Successor state
- Parsing machine : automata used for parsing, consists of states and transitions.
- Transition successor (configuration) : $A \rightarrow \alpha X \bullet \beta$
- Immediate successor (configuration) : $X \rightarrow \bullet \eta$
- A lane in a parsing machine: a sequence of configurations $\xi_1, \xi_2, \ldots, \xi_n$, where for i = 0 to n-1, ξ_{i+1} is the immediate or transition successor of ξ_i.

I. Overview

- We will go over these for each algorithm
  - Algorithm detail
  - Pros and cons
  - Related work
  - An example

II. Knuth Canonical LR Algorithm

- First described in Knuth's 1965 paper:
  - Donald Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.
- Pros:
  - Grammars handled by LR(k) is a superset of LL, LALR, SLR
- Cons:
  - Computationally expensive in space/time
  - Generated parser also too big to be practical
  - Complexity grows exponentially in theory
- Related implementations
  - Usually limited to k=1: LR(1)
  - Research had focused on improving space/time expense

II. Knuth Canonical LR Algorithm

- An easier to understand specification:
  - In the dragon book (1986, page 232)
    - Alfred V. Aho, Jefferey D. Ullman, and Ravi Sethe. Compilers: Principles, Techniques, and Tools. 1986
  - Two major steps
    - Closure():
      Get the closure of a state.
    - Transition()/Goto():
      Make a transition from a state to one of its successors.

```
function closure(I);
begin
    repeat
        for each item [A → α . B β, a] in I,
            each production B → γ in G',
            and each terminal b in FIRST(β a)
            such that [B → . γ, b] is not in I do
                add [B → . γ, b] to I;
    until no more items can be added to I;
    return I
end;

function goto(I, X);
begin
    let J be the set of items [A → α X . β, a] such that
        [A → α . X β, a] is in I;
    return closure(J)
end;

procedure items(G');
begin
    C := {closure({[S' → .S, $]})};
    repeat
        for each set of items I in C and each grammar symbol X
                such that goto(I, X) is not empty and not in C do
                    add goto(I, X) to C
    until no more sets of items can be added to C
end
```

Fig. 4.38. Sets of LR(1) items construction for grammar G'.

II. Knuth Canonical LR Algorithm

- Implementation tips to improve performance:
  - Closure():
    Incrementally combining new configurations with the same core.

II. Knuth Canonical LR Algorithm

- Implementation tips to improve performance:
  - Closure():
    Incrementally combining new configurations with the same core.
  - Transition()/Goto():
    Search for existing state efficiently

Algorithm 4.11: findExistingState(S)

Input: state S

Output: state number of the found state, or -1 if not found

```
find a state T that is the same as S;
if T is found then
    return the state number of T;
else
    return -1;
```

II. Knuth Canonical LR Algorithm

- Implementation tips to improve performance:
  - Closure():
    Incrementally combining new configurations with the same core.
  - Transition()/Goto():
    Search for existing state efficiently (using a hash table of states).

Algorithm 4.11: findExistingState(S)

Input: state S

Output: state number of the found state, or -1 if not found

find a state T that is the same as S;
if T is found **then**
  return the state number of T;
else
  return -1;

Algorithm 4.12: Hash function for a state S

Input: state S

Output: hash value of the state

$val = 0;$
foreach core configuration r of state S **do**
  $val = (val + r.ruleID * 97 + r.marker * 7 + i) \ \% \ H\_SIZE;$
return $val;$

II. Knuth Canonical LR Algorithm

- Example: Given grammar G1:
  $E \rightarrow E + T \mid T$
  $T \rightarrow T * a \mid a$

II. Knuth Canonical LR Algorithm

- Example: Given grammar G1:
  E → E + T | T
  T → T * a | a

- This can also be written as:
  0) G → E $ → Augmented rule, $ is end marker
  1) E → E + T
  2) E → T
  3) T → T * a
  4) T → a

II. Knuth Canonical LR Algorithm

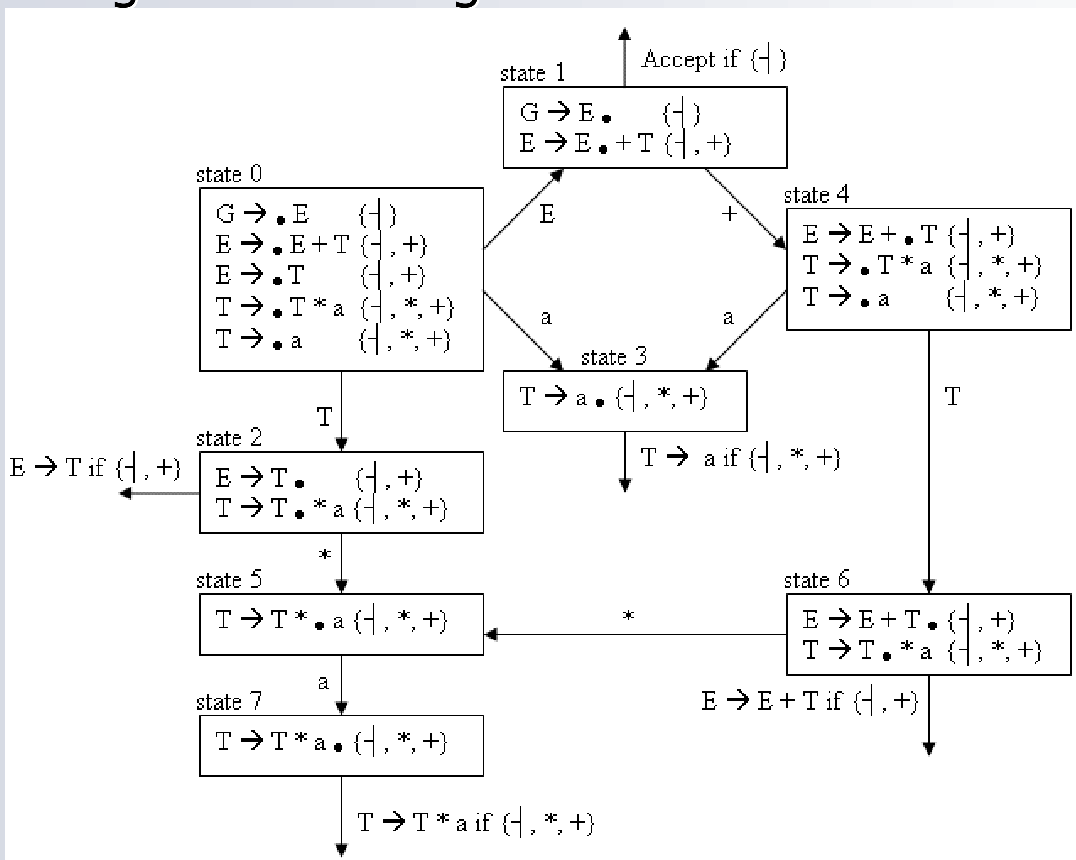
- Parsing table for grammar G1:

TABLE 2. PARSING TABLE FOR GRAMMAR G1

State	$	+	*	*a*	*E*	*T*
0	0	0	0	s3	g1	g2
1	a0	s4	0	0	0	0
2	r2	r2	s5	0	0	0
3	r4	r4	r4	0	0	0
4	0	0	0	s3	0	g6
5	0	0	0	s7	0	0
6	r1	r1	s5	0	0	0
7	r3	r3	r3	0	0	0

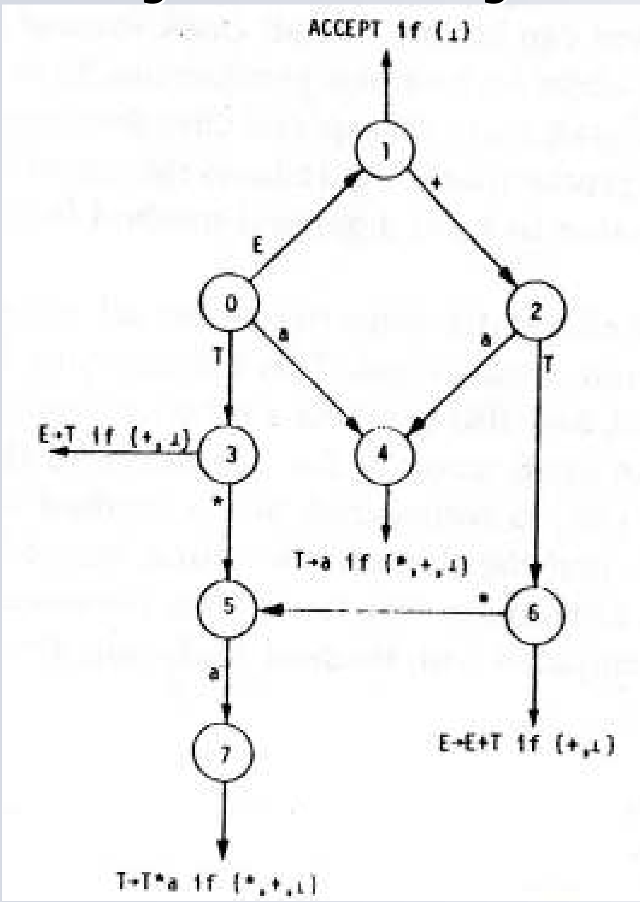
II. Knuth Canonical LR Algorithm

- Parsing machine for grammar G1:



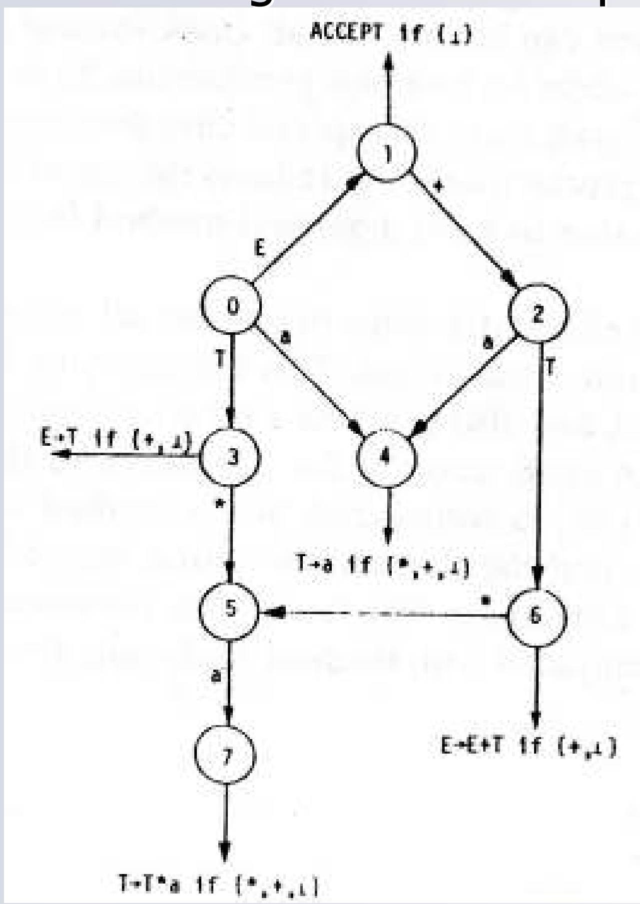
II. Knuth Canonical LR Algorithm

- Parsing machine for grammar G1:



II. Knuth Canonical LR Algorithm

- Parse string “a + a” with parsing machine of G1:



STEP NO.	STACK	REMAINING INPUT
1	$_0$	a+a⊥
2	$_0a_4$	+a⊥
3	$_0T_3$	+a⊥
4	$_0E_1$	+a⊥
5	$_0E_1+_2$	a⊥
6	$_0E_1+_2a_4$	⊥
7	$_0E_1+_2T_6$	⊥
8	$_0E_1$	⊥
9	ACCEPT	

III. Practical general method (PGM)

- Given by Pager
  - David Pager. A practical general method for constructing LR(k) parsers. Acta Informatica, 7:249 – 268, 1977.
  - Merge compatible states
  - Based on weak and strong compatibility
- Pros
  - Weak compatibility is easy to understand and implement
  - Easy addition to Knuth canonical LR(1) algorithm
- Implenetations
  - LR: ANSI standard Fortran 66 (Lawrence Liverpool Lab, 1981)
  - LRSYS: Pascal (Lawrence Liverpool Lab, 1985)
  - LALR: MACRO-11, on a RSX-11 machine (1988)
  - GDT_PC: (Sweden, 1988)
  - Menhir: Objective Caml (France, 2004)
  - The Python Parsing module: Python (2007)

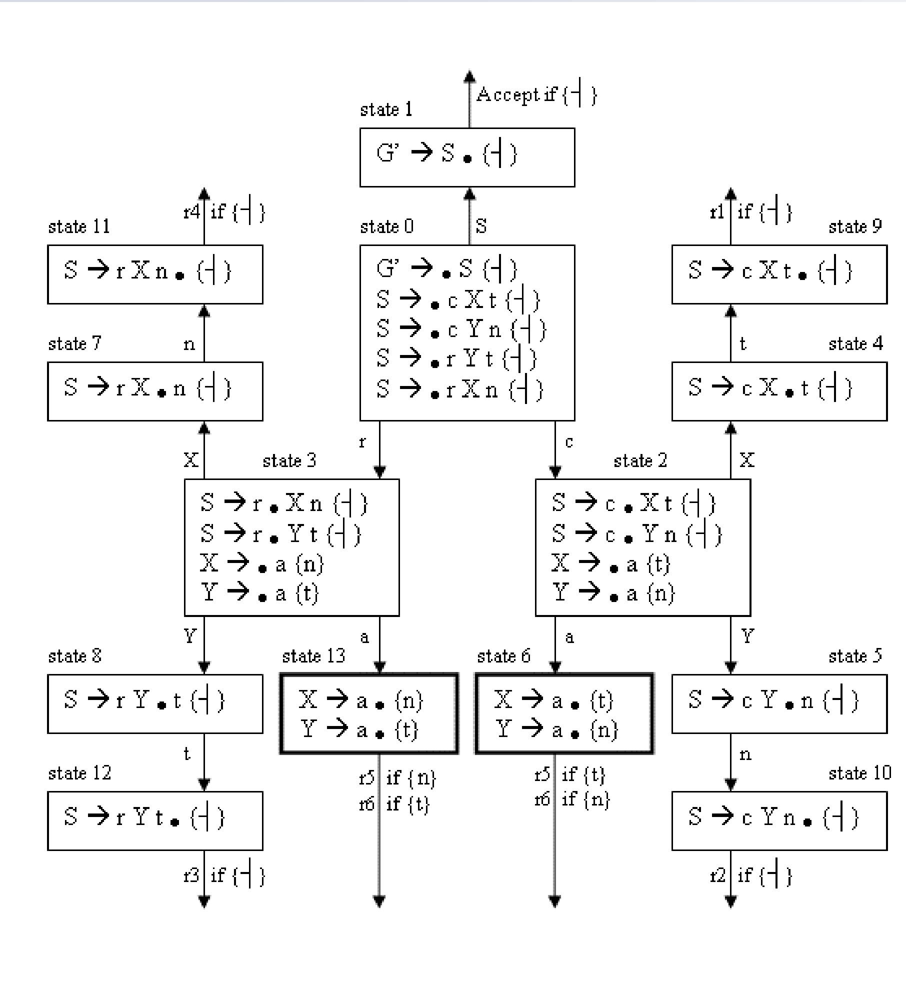
III. Practical general method (PGM)

- Definition of weak compatibility
  - Let S and S′ be two states with a common set of core configurations
  - Let U_r and U_r' be the context of the r-th core configuration of S and S′, respectively
  - S and S′ are weakly compatible iff at least one of the following is true for all i and j, $1 <= i < j <= n$, n is number of core configurations in S and S′.
    - 1) $U_i \cap U_j' = \varnothing$ and $U_i' \cap U_j = \varnothing$
    - 2) $U_i \cap U_j \neq \varnothing$
    - 3) $U_i' \cap U_j' \neq \varnothing$

III. Practical general method (PGM)

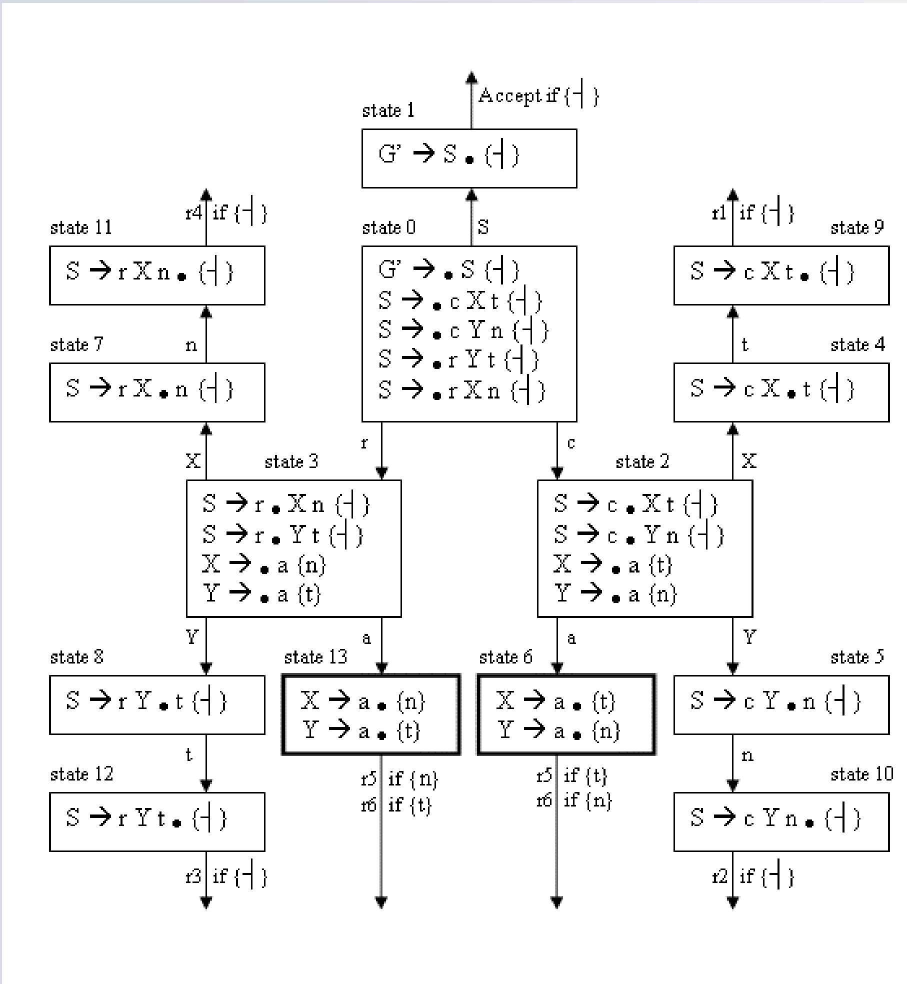
- Example: given grammar G2:
  - S → c X t | c Y n | r Y t | r X n
  - X → a
  - Y → a

III. Practical general method (PGM)



- State machine of G2.
- Can we combine state 13 into 6? (i.e., are states 13 and 6 compatible?)
- State S = 6,
  - U_1 : {n}, U_2 : {t}
- State S′ = 13,
  - U_1' : = {t}, U_2' : = {n}
- Check weak compatibility:
  1) $U_i \cap U_j' = \emptyset$ and $U_i' \cap U_j = \emptyset$?
  2) $U_i \cap U_j \neq \emptyset$?
  3) $U_i' \cap U_j' \neq \emptyset$?

III. Practical general method (PGM)



- State machine of G2.
- Can we combine state 13 into 6? (i.e., are states 13 and 6 compatible?)
- State S = 6,
  - U_1 : {n}, U_2 : {t}
- State S′ = 13,
  - U_1' : = {t}, U_2' : = {n}
- Check weak compatibility:
  1) $U_i \cap U_j' = \emptyset$ and $U_i' \cap U_j = \emptyset$?
  2) $U_i \cap U_j \neq \emptyset$?
  3) $U_i' \cap U_j' \neq \emptyset$?
- 1), 2), 3) are all false: NOT compatible, CAN'T combine.

III. Practical general method (PGM)

- Implemented as addition to the Knuth Algorithm

Algorithm 4.13: findExistingState(S) for the PGM algorithm

Input: state S

Output: the state number of a same or compatible state, or -1 if not found

```
find a state T which is the same as S;
if T is found then
    return the state number of T;
else
    find a compatible state T of S;
    if T is found then
        combineCompatibleStates(T, S);
        return state number of T;
    else
        return -1;
```

III. Practical general method (PGM)

- Implemented as addition to the Knuth Algorithm

Algorithm 4.13: findExistingState(S) for the PGM algorithm

Input: state S

Output: the state number of a same or compatible state, or -1 if not found

```
find a state T which is the same as S;
if T is found then
    return the state number of T;
else
    find a compatible state T of S;
    if T is found then
        combineCompatibleStates(T, S);
        return state number of T;
    else
        return -1;
```

Algorithm 4.11: findExistingState(S)

Input: state S

Output: state number of the found state, or -1 if not found

```
find a state T that is the same as S;
if T is found then
    return the state number of T;
else
    return -1;
```

III. Practical general method (PGM)

- Combine compatible configurations

```
void combineCompatibleConfig(State * s) {
  ...

  for (i = 1; i < s->config_count; i ++) {
    c = s->config[i];
    if (c == NULL) continue;
    for (j = 0; j < i; j ++) {
      if (isCompatibleConfig(c, s->config[j]) == TRUE) {
        combineContext(& s->config[j]->context, & c->context);
        freeConfig(c); // combine config i to j, then remove i.
        s->config[i] = NULL;
        break;
      } // end if
    } // end for
  } // end for

  ...
}


BOOL isCompatibleStates(State * s1, State * s2) {
  int count;

  if (hasCommonCore(s1, s2) == FALSE) return FALSE;

  count = s1->core_config_count;
  if (count == 1) return TRUE;

  // now check context to see if s1 and s2 are compatible.
  if (isCompatibleState_a(s1, s2) == TRUE) return TRUE;
  if (isCompatibleState_bc(s1) == TRUE) return TRUE;
  if (isCompatibleState_bc(s2) == TRUE) return TRUE;

  return FALSE;
}
```

III. Practical general method (PGM)

- Major implementation issue:
  - If the combine state has successor states:
    - Propagate state context change
    - Do this when cycle exists
    - Can do this more efficiently by:
      - Propagate configuration change only where contexts atually change
      - Use a queue, append changed configurations to the end

IV. Lane-tracing algorithm

- Given by Pager
  - David Pager. The lane tracing algorithm for constructing LR(k) parsers. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 172 – 181, Austin, Texas, United States, 1973.
  - David Pager. The lane-tracing algorithm for constructing LR(k) parsers and ways of enhancing its efficiency. *Information Sciences*, 12:19–42, 1977.

IV. Lane-tracing algorithm

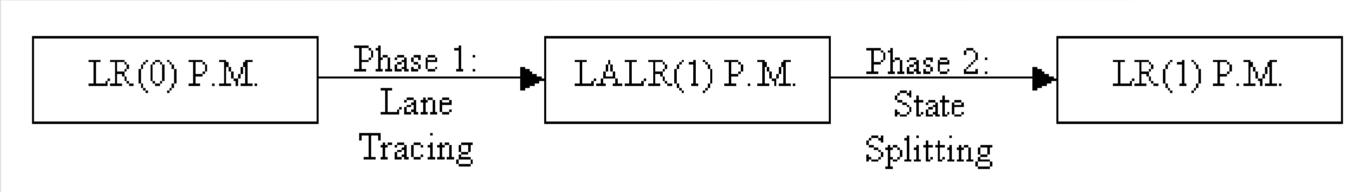
- Lane-tracing algorithm
  - The first practical general method used to create LR(k) parser generators
  - Split states with conflicts
- Pros
  - Good for extending to LR(k)
- Cons
  - Hard to understand and implement
- Relevant implementations
  - In OS 360/65 Assembly (1977), applied to ALGOL grammar

IV. Lane-tracing algorithm

- Outline
  - First generate LR(0) parsing machine
  - Lane tracing phase 1: Lane tracing
    - Trace lanes, generate context, resolve shift/reduce and reduce/reduce conflicts, to result in a LALR(1) parsing machine
    - If no reduce/reduce conflict, stop. Otherwise, go on to phase 2.
  - Lane tracing phase 2: State splitting
    - Split inadequiate states to result in a LR(1) parsing machine
    - If no more reduce/reduce conflict, stop. Otherwise, not LR grammar.

IV. Lane-tracing algorithm

- Outline
  - First generate LR(0) parsing machine
  - Lane tracing phase 1: Lane tracing
    - Trace lanes, generate context, resolve shift/reduce and reduce/reduce conflicts, to result in a LALR(1) parsing machine
    - If no reduce/reduce conflict, stop. Otherwise, go on to phase 2.
  - Lane tracing phase 2: State splitting
    - Split inadequiate states to result in a LR(1) parsing machine
    - If no more reduce/reduce conflict, stop. Otherwise, not LR grammar.



IV. Lane-tracing algorithm

- Example: given grammar G3:

$$
\begin{aligned}
&G \rightarrow xWa \mid xV \mid yWb \mid yV \mid zWr \mid zV \mid uUXd \mid uUYE \\
&W \rightarrow UXC \\
&V \rightarrow UYd \\
&U \rightarrow Ut \mid s \\
&X \rightarrow tUXP \mid t \\
&Y \rightarrow tUYu \mid t \\
&E \rightarrow a \mid b \mid c \mid v \\
&c \rightarrow c \mid w \mid \epsilon \\
&P \rightarrow \epsilon
\end{aligned}
$$

IV. Lane-tracing algorithm

- Step 1. Create LR(0) parsing machine.

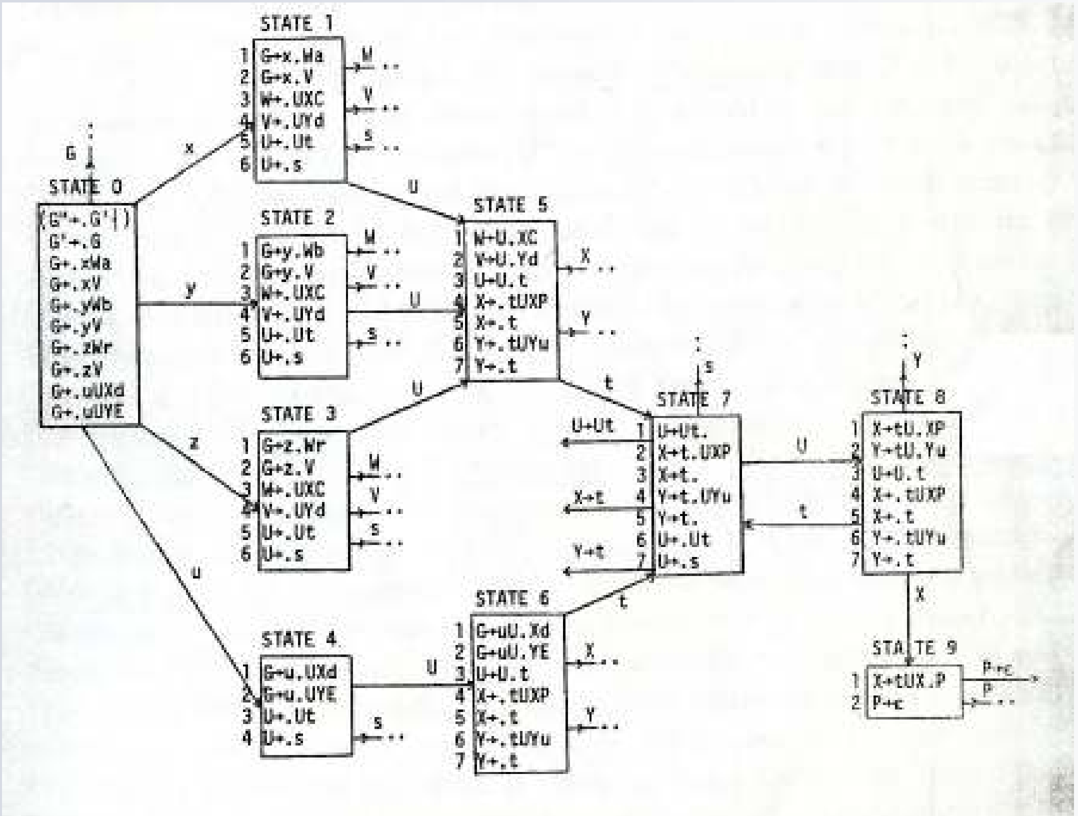


Fig. 5. Part of the $LR(0)$ parsing machine for G_3: $G \to xWa|xV|yWb|yV|zWr|zV|uUXd|uUYE$ $\quad W \to UXC \quad V \to UXd \quad U \to Ut|s \quad X \to tUXP|t \quad Y \to tUYu|t \quad E \to a|b|c|e$ $\quad C \to c|w|e \quad P \to \varepsilon.$

IV. Lane-tracing algorithm

- Step 2. lane-tracing phase 1. Trace relevant lanes leading to conflict.

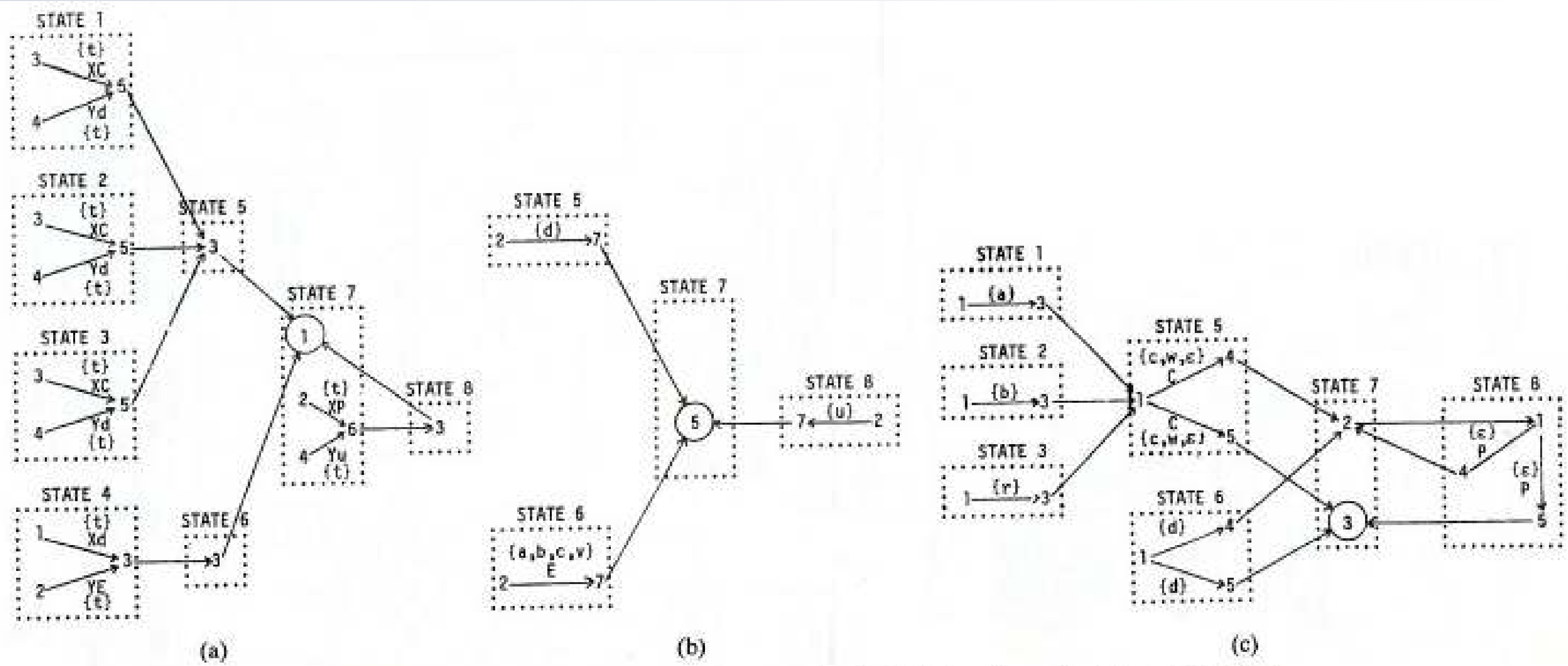


Fig. 6. Lanes leading to the configurations associated with the various reductions at state 7 of Fig. 5. The configuration associated with the reduction is circled in each case. (*a*) Lanes generating contexts for $U \rightarrow Ut$. (*b*) Lanes generating context for $Y \rightarrow t$. (*c*) Lanes generating context for $X \rightarrow t$.

IV. Lane-tracing algorithm

- A practical algorithm for Phase 1
  - The number of possible lanes rises exponentially with the number of configurations
  - We want at most a linear increase:
    - Avoid trace lane repeatedly
    - Add the contexts generated by each lane to an auxiliary set of strings ($CONTEXTS_{\xi}$) associated with each member configuration ξ of the lane.
  - Strategems and heuristics for enhancing efficiency
    - The context adding procedure
    - Determining contexts for the members of loops
    - Making use of context sets already evaluated
    - Heuristic propagation of context sets

IV. Lane-tracing algorithm

- Flow diagram for evaluating the context set for reduction π_i with which configuration ξ_j is associated during phase 1 lane tracing.

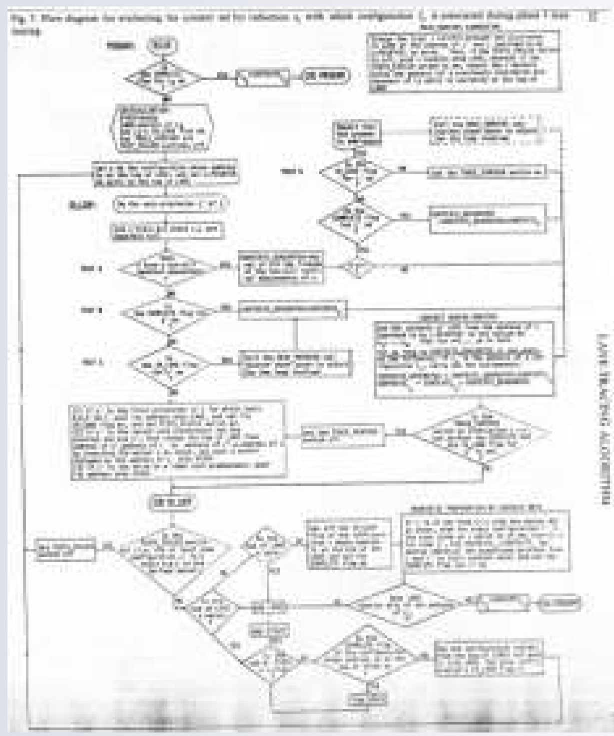
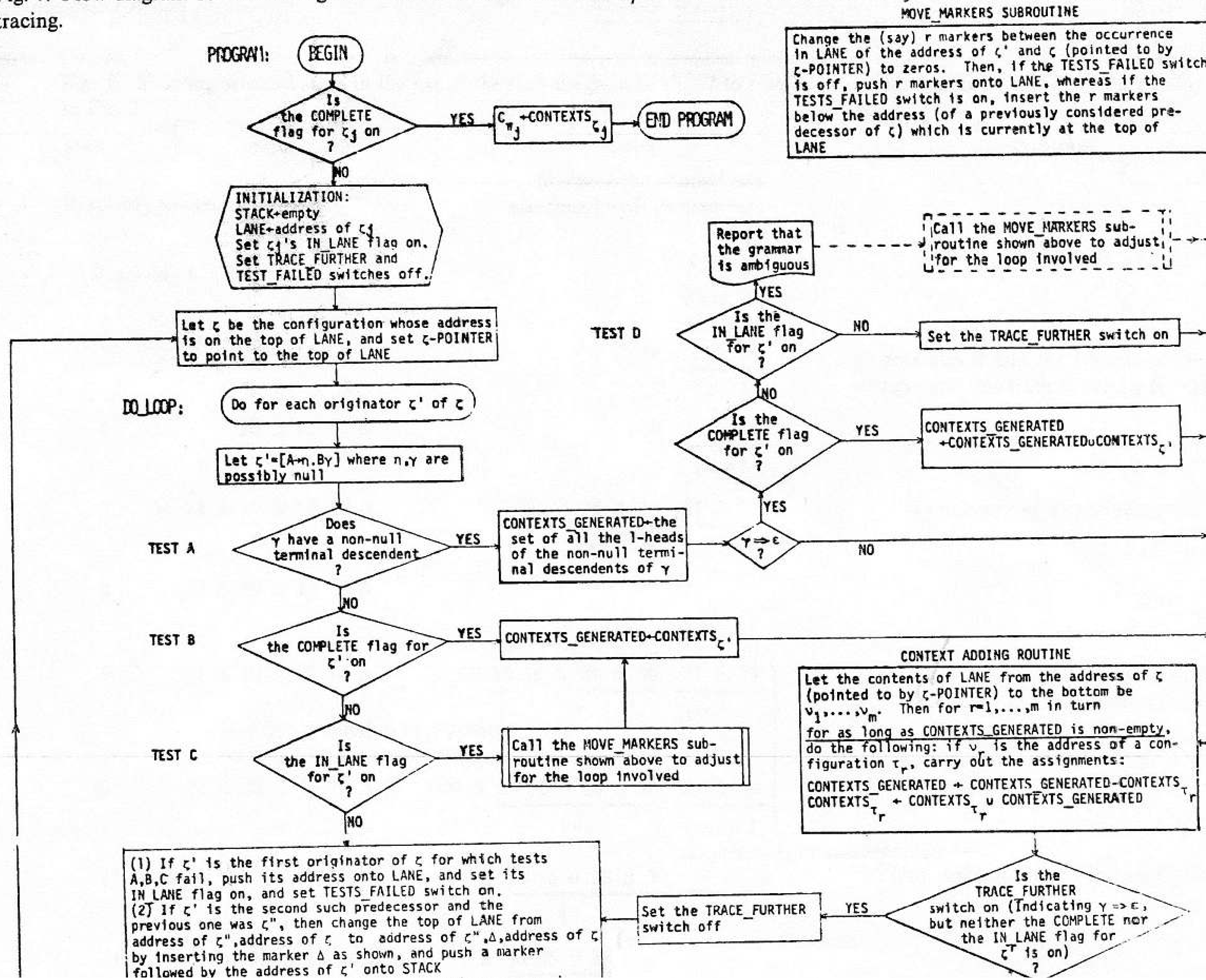
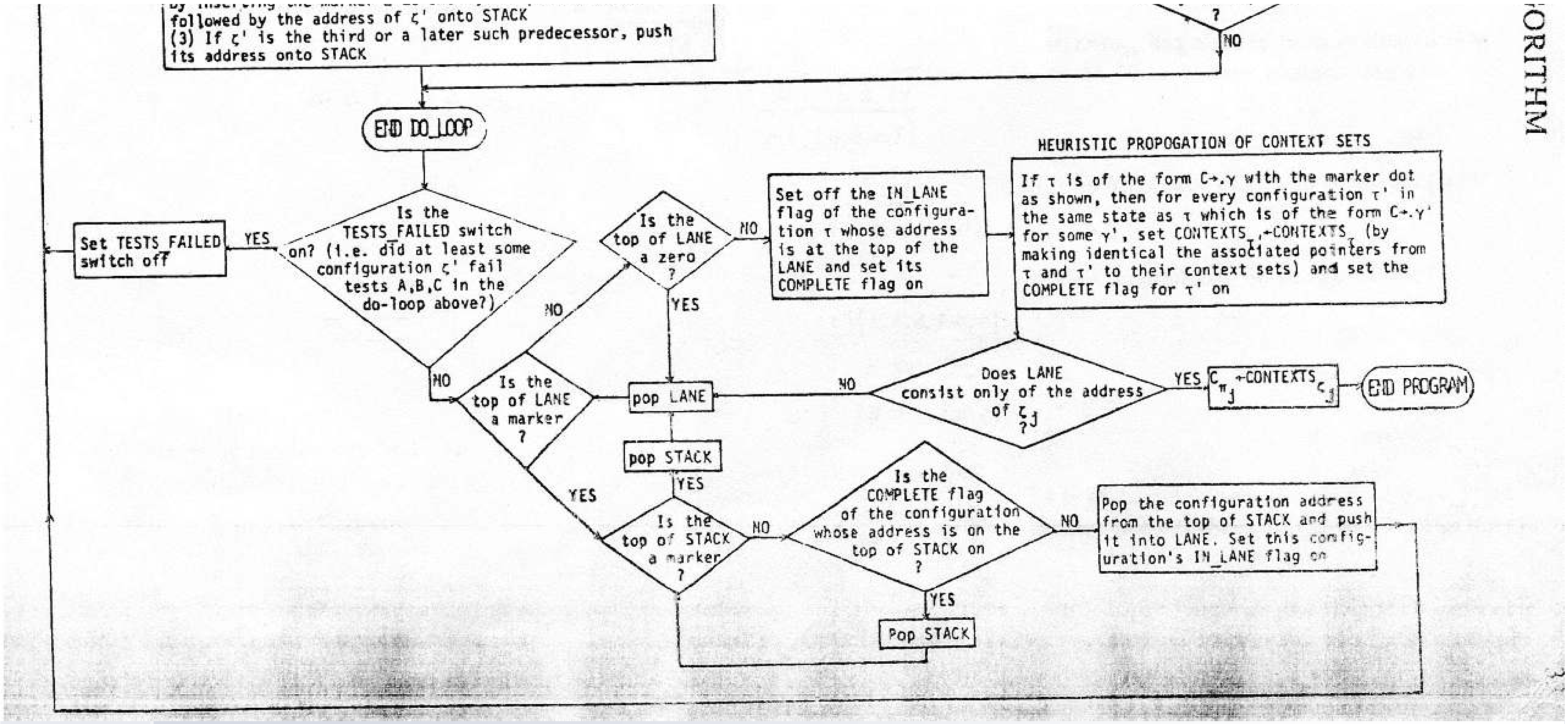


Fig. 7. Flow diagram for evaluating the context set for reduction π_j with which configuration ζ_j is associated during phase 1 lane tracing.




followed by the address of ζ' onto STACK
(3) If ζ' is the third or a later such predecessor, push its address onto STACK
NO
ORITHM
END DO_LOOP
HEURISTIC PROPOGATION OF CONTEXT SETS
If τ is of the form C→.γ with the marker dot as shown, then for every configuration τ' in the same state as τ which is of the form C→.γ' for some γ', set CONTEXTS τ'←CONTEXTS τ (by making identical the associated pointers from τ and τ' to their context sets) and set the COMPLETE flag for τ' on
Set off the IN_LANE flag of the configuration τ whose address is at the top of the LANE and set its COMPLETE flag on
Set TESTS_FAILED switch off
YES
Is the TESTS_FAILED switch on? (i.e. did at least some configuration ζ' fail tests A,B,C in the do-loop above?)
Is the top of LANE a zero ?
NO
YES
NO
Does LANE consist only of the address of ζj ?
YES
C πj ←CONTEXTS ζj
END PROGRAM
NO
Is the top of LANE a marker ?
pop LANE
pop STACK
YES
YES
Is the top of STACK a marker ?
NO
Is the COMPLETE flag of the configuration whose address is on the top of STACK on ?
NO
Pop the configuration address from the top of STACK and push it into LANE. Set this configuration's IN_LANE flag on
YES
Pop STACK

IV. Lane-tracing algorithm

- Example: use the flow diagram to resolve inadequate states.

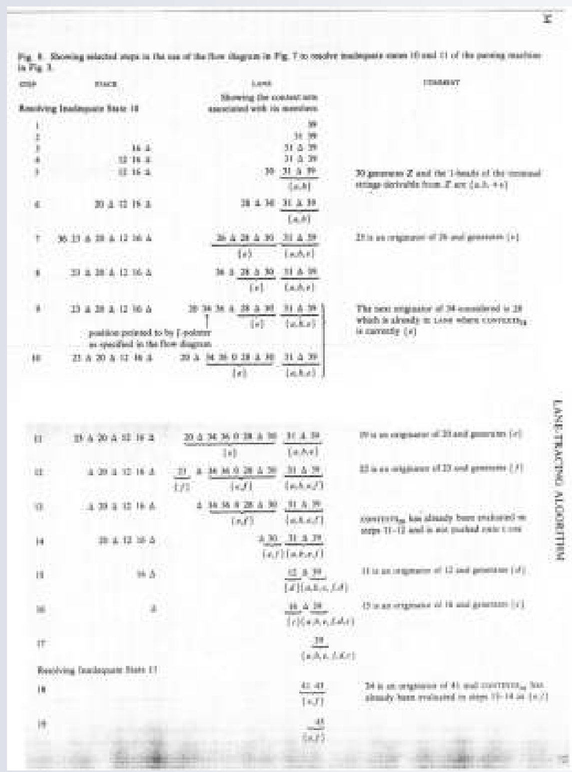

LANE-TRACING ALGORITHM

Fig. 8. Showing selected steps in the use of the flow diagram in Fig. 7 to resolve inadequate states 10 and 11 of the parsing machine in Fig. 3.

STEP	STACK	LANE	COMMENT
Resolving Inadequate State 10		Showing the context sets associated with its members	
1		39	
2		31 39	
3	16 Δ	31 Δ 39	
4	12 16 Δ	31 Δ 39	
5	12 16 Δ	30 $\underbrace{31\ \Delta\ 39}_{\{a,b\}}$	30 generates Z and the 1-heads of the terminal strings derivable from Z are $\{a, b, +\varepsilon\}$
6	20 Δ 12 16 Δ	28 Δ 30 $\underbrace{31\ \Delta\ 39}_{\{a,b\}}$	
7	36 23 Δ 20 Δ 12 16 Δ	$\underbrace{26\ \Delta\ 28\ \Delta\ 30}_{\{e\}}\ \underbrace{31\ \Delta\ 39}_{\{a,b,e\}}$	25 is an originator of 26 and generates $\{e\}$
8	23 Δ 20 Δ 12 16 Δ	36 Δ $\underbrace{28\ \Delta\ 30}_{\{e\}}\ \underbrace{31\ \Delta\ 39}_{\{a,b,e\}}$	
9	23 Δ 20 Δ 12 16 Δ	20 34 36 Δ $\underbrace{28\ \Delta\ 30}_{\{e\}}\ \underbrace{31\ \Delta\ 39}_{\{a,b,e\}}$ ↑ position pointed to by ζ-pointer as specified in the flow diagram	The next originator of 34 considered is 28 which is already in LANE where CONTEXTS_{28} is currently $\{e\}$
10	23 Δ 20 Δ 12 16 Δ	20 Δ $\underbrace{34\ 36\ 0\ 28\ \Delta\ 30}_{\{e\}}\ \underbrace{31\ \Delta\ 39}_{\{a,b,e\}}$	

11	23 Δ 20 Δ 12 16 Δ	$\underbrace{20\ \Delta\ 34\ 36\ 0\ 28\ \Delta\ 30}_{\{e\}}\ \underbrace{31\ \Delta\ 39}_{\{a,b,e\}}$	19 is an originator of 20 and generates $\{e\}$
12	Δ 20 Δ 12 16 Δ	$\underbrace{23}_{\{f\}}\ \Delta\ \underbrace{34\ 36\ 0\ 28\ \Delta\ 30}_{\{e,f\}}\ \underbrace{31\ \Delta\ 39}_{\{a,b,e,f\}}$	22 is an originator of 23 and generates $\{f\}$
13	Δ 20 Δ 12 16 Δ	$\Delta\ \underbrace{34\ 36\ 0\ 28\ \Delta\ 30}_{\{e,f\}}\ \underbrace{31\ \Delta\ 39}_{\{a,b,e,f\}}$	CONTEXTS$_{20}$ has already been evaluated in steps 11–12 and is not pushed onto LANE
14	20 Δ 12 16 Δ	$\Delta\ \underbrace{30}_{\{e,f\}}\underbrace{31\ \Delta\ 39}_{\{a,b,e,f\}}$	
15	16 Δ	$\underbrace{12}_{\{d\}}\ \Delta\ \underbrace{39}_{\{a,b,e,f,d\}}$	11 is an originator of 12 and generates $\{d\}$
16	Δ	$\underbrace{16}_{\{c\}}\ \Delta\ \underbrace{39}_{\{a,b,e,f,d,c\}}$	15 is an originator of 16 and generates $\{c\}$
17		$\underbrace{39}_{\{a,b,e,f,d,c\}}$	
Resolving Inadequate State 11			
18		$\underbrace{41\ 43}_{\{e,f\}}$	34 is an originator of 41 and CONTEXTS$_{34}$ has already been evaluated in steps 13–14 as $\{e,f\}$
19		$\underbrace{43}_{\{e,f\}}$	

IV. Lane-tracing algorithm

- Lane-tracing Phase 2: State-splitting phase.
  - Start from lanehead states (where conflict lanes originates), regenerate successor states.
  - Two approaches:

    1) using the Practical General Method (PGM)

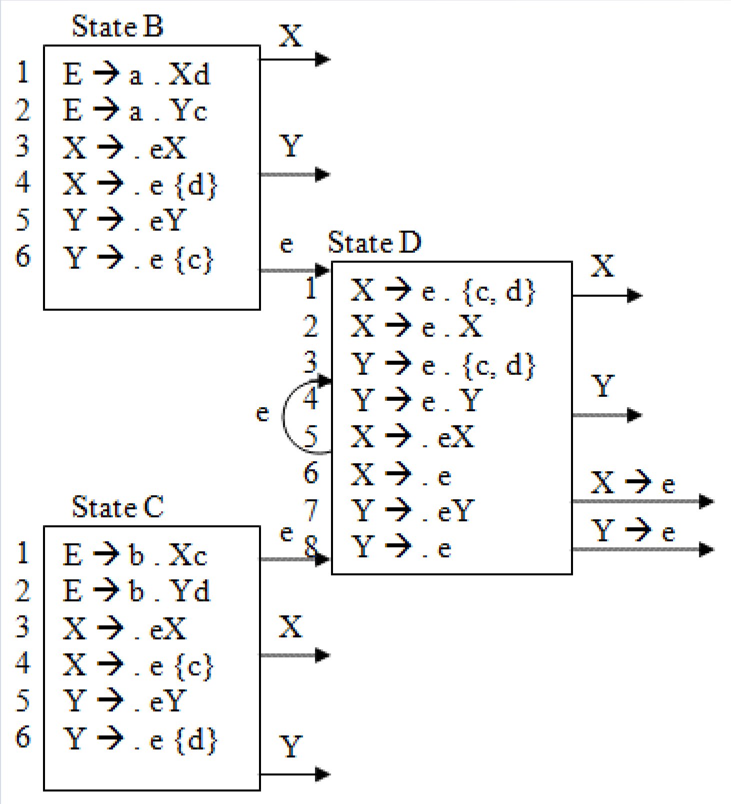
    2) using the Lane Table method.

IV. Lane-tracing algorithm

- Example: Lane-tracing Phase 2 using the PGM method.
- Given grammar G4:
  - E → a X d | b X c | b Y d
  - X → e X | e
  - Y → e Y | e

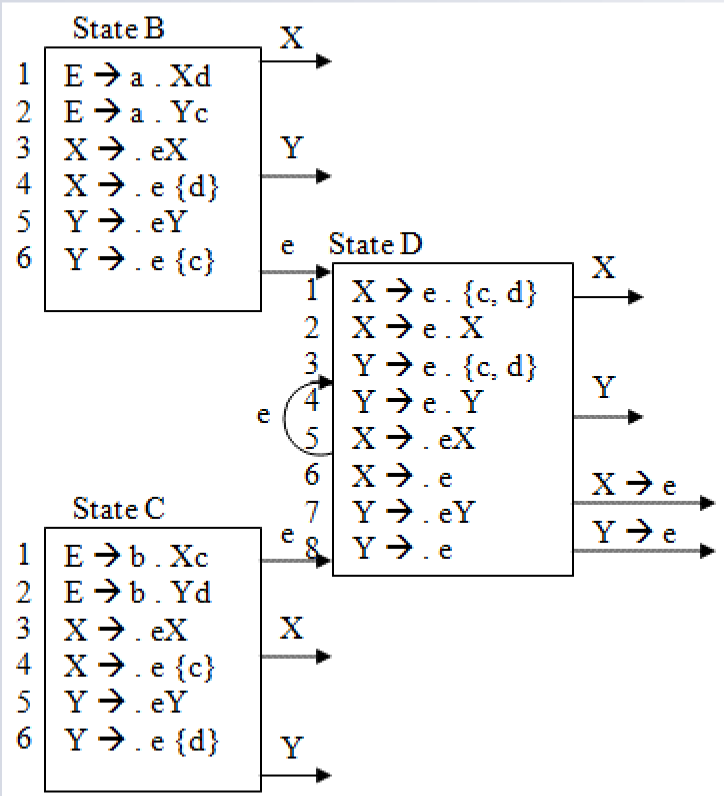
IV. Lane-tracing algorithm

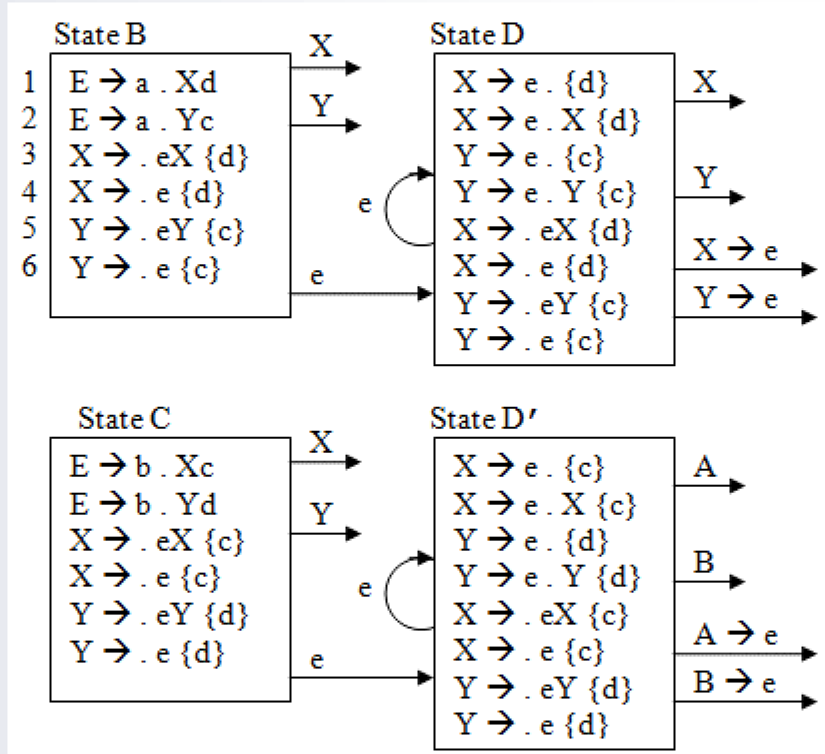
- LALR(1) parsing machine of G4 (relevant part)



IV. Lane-tracing algorithm

- LALR(1) parsing machine of G4 (relevant part): obtained by splitting state D.





V. Unit Production Elimination (UPE)

- Unit production: X → Y
- Unit production:
  - Does nothing but reduce
  - A number of unit productions can form single production chain:

    U → V, V → W, W → X, X → Y, Y → Z   form this chain:

    U → V → W → X → Y → Z
  - Increase parsing table size, waste parsing time. Sometimes by 20-30%.

V. Unit Production Elimination (UPE)

- Pager's Unit production algorithm:

Algorithm 1.

1. For each state S of the machine in turn (including the new states added to the machine in step 2), do step 2 for each leaf x, if any, such that the x-successor of S has a unit reduction. When all these iterations of step 2 are complete, go on to steps 3, 4, 5.

2. Let $x_1, \ldots, x_n$ be the symbols (which will include x) for which actions are defined at S such that $x_i \Rightarrow x$ and, for $1 \leqq i \leqq n$, let the x_i-successor of S be T_i. If any state R is, or at a previous stage of the Algorithm has been, a combination of states $T_1, \ldots, T_n$[6], make R the new x-successor of S; otherwise set up a new state T as the x-successor of S, where T is a combination of states $T_1, \ldots, T_n$[7].

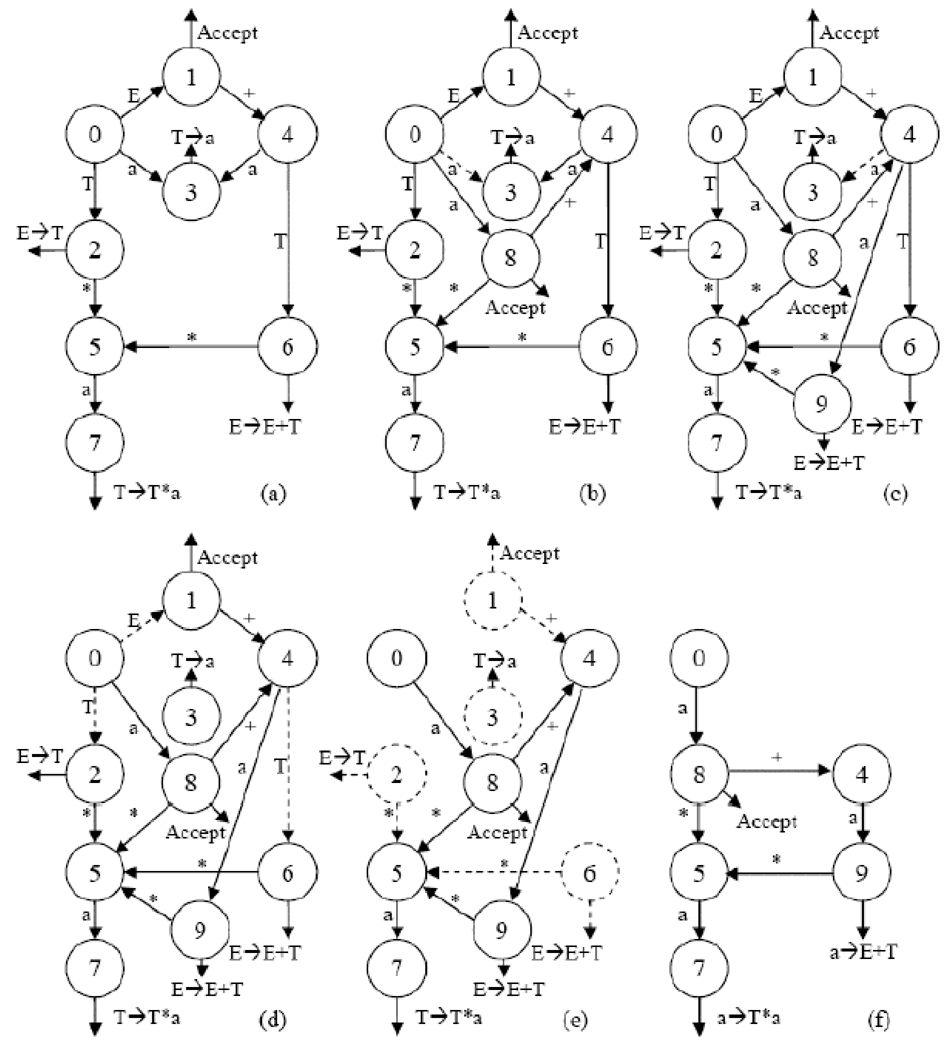
3. Delete all transitions with respect to left-hand sides of unit productions.

4. Delete all states which at this stage are not reachable from state 0.

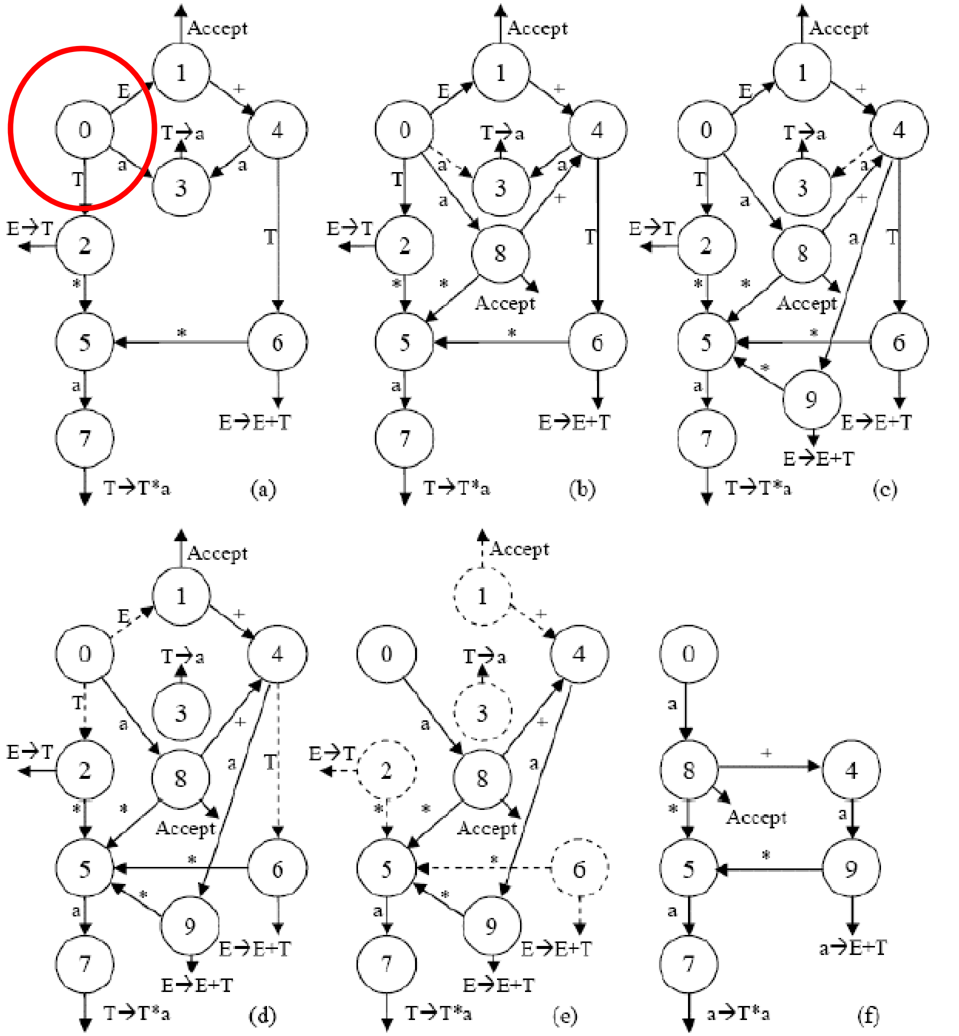
5. Replace every reduce action $y \rightarrow \omega$, where y is a left-hand side of a unit production, by $x \rightarrow \omega$, where x is an arbitrarily selected leaf such that $y \Rightarrow x$[8].

V. Unit Production Elimination (UPE)

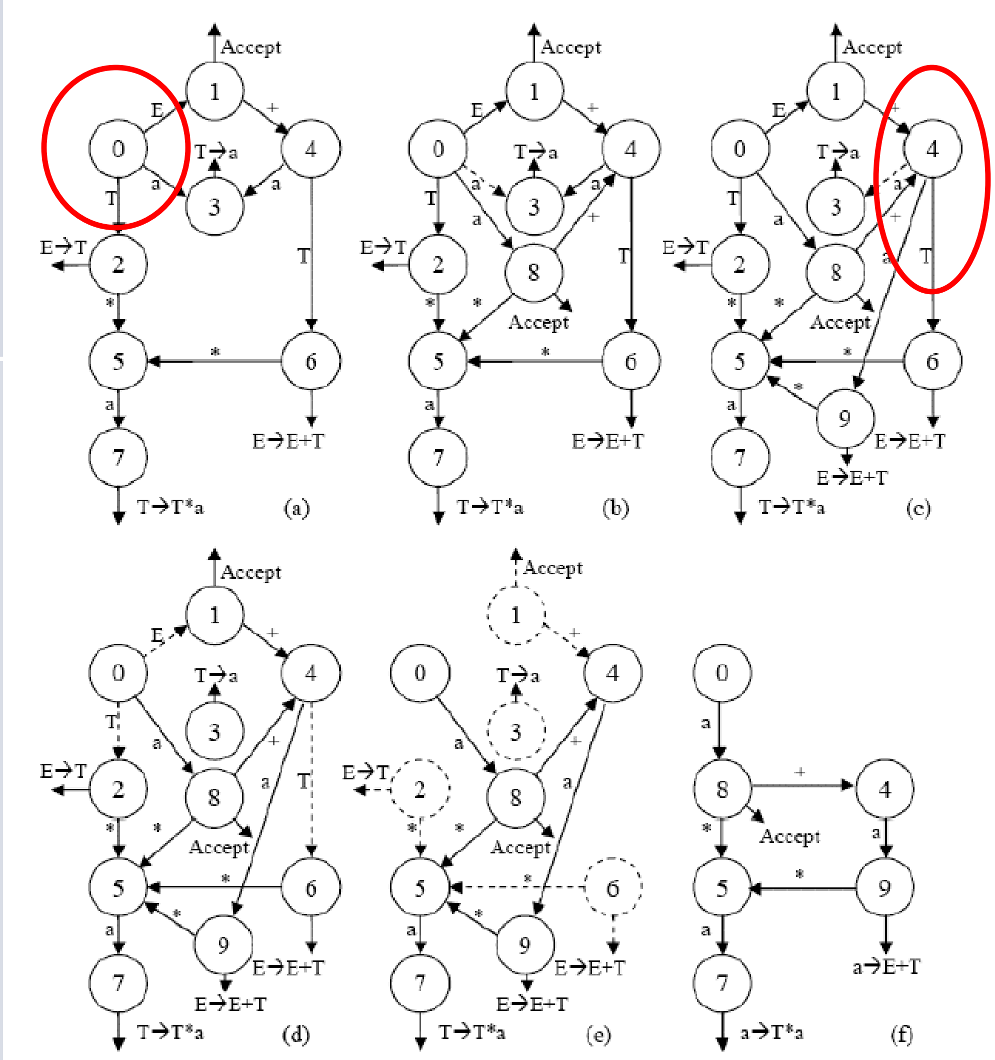
- Example: Given grammar G1:
  - E → E + T | T
  - T → T * a | a
- Two unit productions exist:
  - E → T, T → a
- Multi-rooted tree:
  - E → T → a
  - “a” is the only leaf



(a) Original parsing machine. (b) Combine states 1, 2 and 3 to state 8. Remove link 0 → 3 because there can be only one a-successor for state 0. (c) Combine states 3 and 6 to state 9. Remove link 4 → 3 because there can be only one a-successor for state 2. (d) Remove transitions corresponding to LHS of unit production: E, T. (e) Remove all states unreachable from state 0, and remove their associated action links. (f) Replace LHS of reductions to corresponding leaf.



(a) Original parsing machine. (b) Combine states 1, 2 and 3 to state 8. Remove link 0 → 3 because there can be only one a-successor for state 0. (c) Combine states 3 and 6 to state 9. Remove link 4 → 3 because there can be only one a-successor for state 2. (d) Remove transitions corresponding to LHS of unit production: E, T. (e) Remove all states unreachable from state 0, and remove their associated action links. (f) Replace LHS of reductions to corresponding leaf.



(a) Original parsing machine. (b) Combine states 1, 2 and 3 to state 8. Remove link 0 → 3 because there can be only one a-successor for state 0. (c) Combine states 3 and 6 to state 9. Remove link 4 → 3 because there can be only one a-successor for state 2. (d) Remove transitions corresponding to LHS of unit production: E, T. (e) Remove all states unreachable from state 0, and remove their associated action links. (f) Replace LHS of reductions to corresponding leaf.

VI. UPE Extension

- The LR(1) parsing machine obtained from UPE may contain redundant states.
- An extension is used to remove these redundant states.

VI. UPE Extension – Algorithm

Algorithm 1 (UPExt):

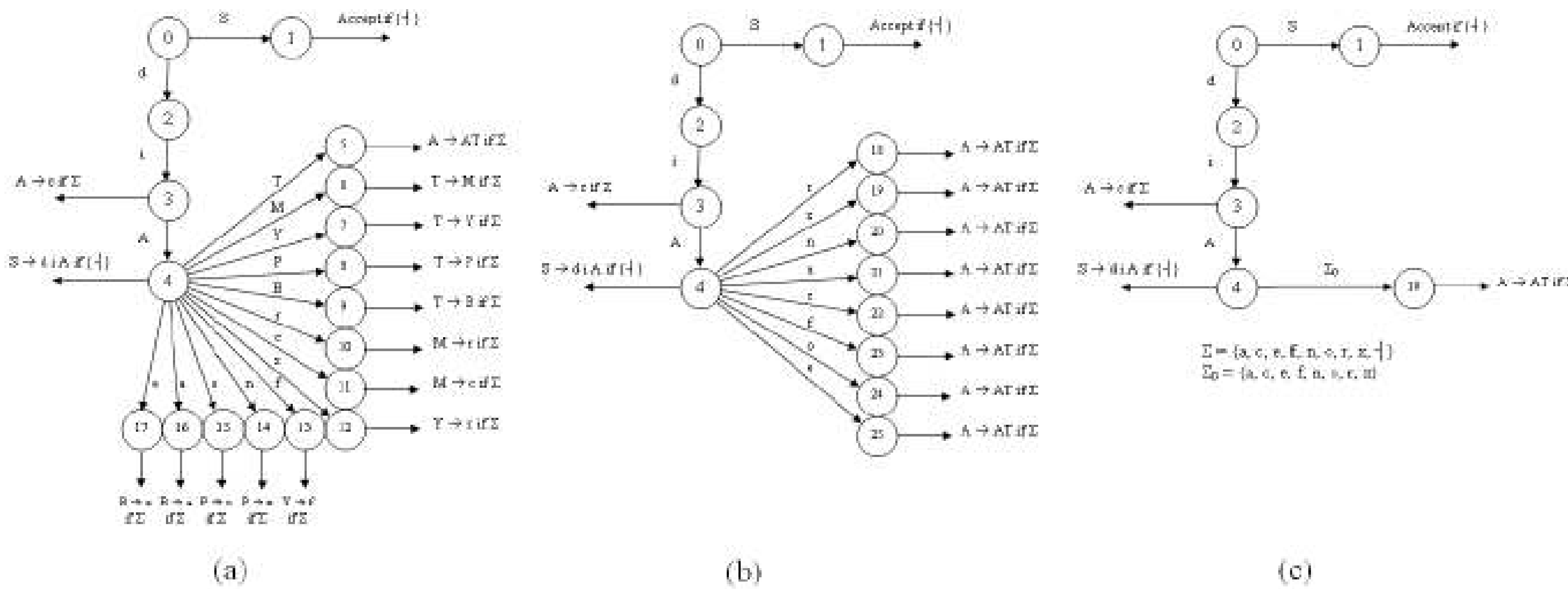
```
Input: Parsing Machine M
Output: A parsing machine M′ where all the
        equivalent states in M are removed;

let Shift(X, k) → Y be a Shift transition from
state X to state Y on token k;
foreach state S in M do
    find the set ∑ of all the equivalent states
    of state S;
    foreach state S′ in ∑ do
        foreach Shift(R, k) → S′ in M do
            replace it by Shift(R, k) → S;
        end
    end
    remove ∑ from M;
end
```

VI. UPE Extension - Example

- Given grammar G4:
  - E → E + T | T
  - T → a | n | (E)

VI. UPE Extension - Example



(a) LR(1) parsing machine, (b) after applying UPE, (c) after applying UPE extension

VII. Other LR(1) algorithms

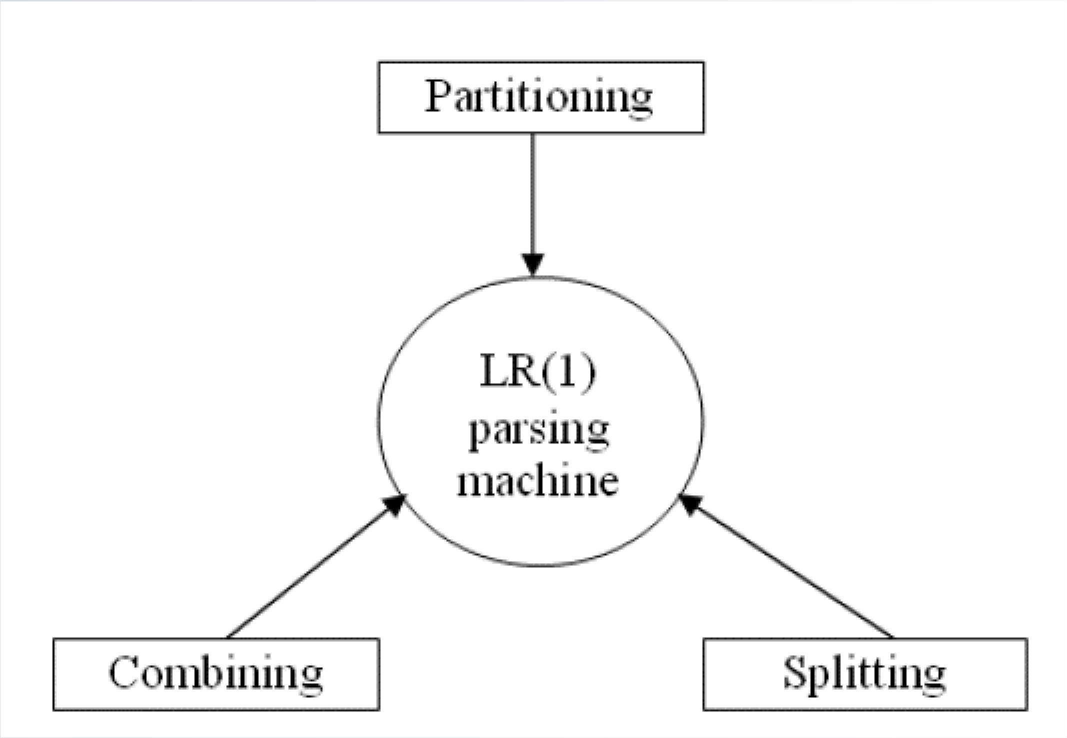
- The Splitting Algorithm of Spector (1981, 1988)
  - Similar to the lane-tracing algorithm in concept
  - No math proof on correctness
  - Was implemented in the Muskox parser generator by Boris Burshteyn
    - "it takes 400Kbytes for each of the C, F77, and MUSKOX grammars", which seems fairly good. Boris also mentioned that "The only interesting pathological case I know about is a COBOL grammar from PCYACC (TM of ABRAXAS SOFTWARE). There, MUSKOX algorithm reached the limit of 90Mbytes and then stopped since there were no virtual memory left."

VII. Other LR(1) algorithms

- The Partitioning Algorithm of Korenjak (1969) [30]
  - Partition a large grammar into small parts, check each to see whether it is LR(1), generate LR(1) parsing table, and then combine these into a larger LR(1) parsing table for the original grammar.
  - Problem: how to partition? How many levels to partition? Parition heuristic?
  - Is a framework, can use different LR(1) algorthm for the partitions.

VII. Other LR(1) algorithms

- Summary on approaches to LR(1) parsing generation



VIII. Edge-pushing LR(k) algorithm

- Motivation
  - LR(1) issue solved
    - Reduced space LR(1) algorithms exist
    - Has good performance in both time and space
    - Implemented in Hyacc
  - LR(k): the next step from LR(1)
  - Use of LR(k)
    - Programming languages
    - NLP

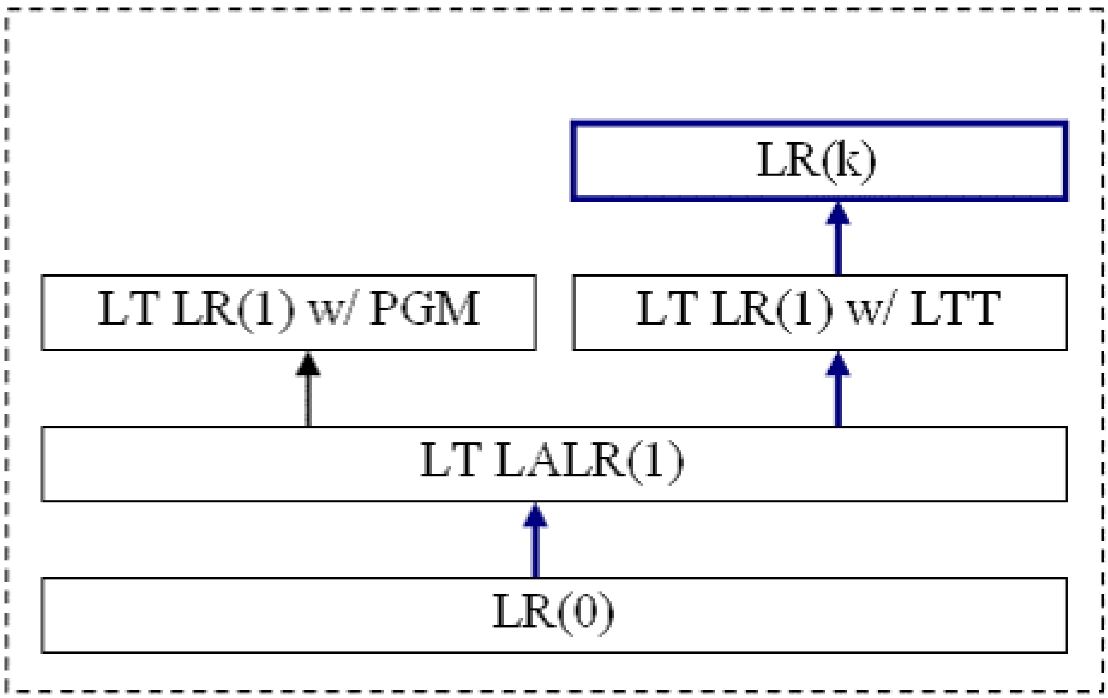
Practical work related to LR(k)

- Ancona et al. (1980s – 1990s)
  - Proposed a method in which non-terminals are not expanded to terminals in context until absolutely necessary.
  - No public tool available.
- Terrence Parr. (1993)
  - PhD thesis: Obtaining practical variants of LL(k) and LR(k) for k > 1 by splitting the atomic k-tuple.
  - ANTLR.
  - Concept applicable to both LL and LR, but only worked on LL(k) in ANTLR
- Vladimir Makarov.
  - MSTA. (1999)
  - The only claimed practical tool. But details not available.
- Other attempts: Will Donahue, Josef Grasch, Bob Buckley, Karsten Nyblad, Chris Clark, Ralph Boland, Ron Newman, Adrian Johnstone etc.

LR(k) algorithm

- Theoretically widely studied, but not practically solved.
- The challenge
  - Two sources of the exponential behavior of LR(k)
    - Number of states
    - Number of lookahead tuples
  - Three issues in a practical LR(k) solution
    - LR(k) Algorithm
    - Storage of LR(k) Parsing Table
    - LR(k) Parse Engine

LR(k): the Lane-tracing path



PGM: Practical General Method. LT: Lane-Tracing. LTT: Lane-Tracing Table.

The Edge-Pushing Algorithm

- Idea
  - LR(1) cannot resolved all reduce/reduce conflicts, then do lane-tracing recursively, get longer contexts to resolve the conflicts
  - If a conflict is
    - Reduction 1: {`ab', 'ac'}, reduction 2: {`ab', `ad'}
    - Since only `ab' causes the conflict, do lane-tracing on the configurations (the "cutting edge") that generate `ab' only.

The Edge-Pushing Algorithm

- Two key procedures
  - Lane-tracing
  - Computation of theads(α, k) (i.e. $First_k(\alpha)$)

The Edge-Pushing Algorithm

- Performance: solution to exponential behavior
  - Number of states: solved by reduced-space LR(1)
  - Number of lookahead tuples: solved by breaking k-tuple. E.g. 1) ANTLR, 2) Hyacc edge-pushing

The Edge-Pushing Algorithm: Storage of LR(k) Parsing Table

- Use a series of tables:
  - LR(1) parsing table
  - LR(2) parsing table
  - …
  - LR(k) parsing table
- Label conflict with a special symbol
- In case of LR(i) conflict, consult the LR(i+1) parsing table

The Edge-Pushing Algorithm: LR(k) Parse Engine

- An addition to LR(1) parse engine

Table 2. The hyaccpark LR(k) parse engine algorithm

Algorithm 2: The hyaccpar LR(k) parse engine algorithm.

```
Initialization:
push state 0 onto state_stack;

while next token is not EOF do {
  S ← current state;
  L ← next token/lookahead;
  A ← action of(S, L) in parsing table;
  if A is shift then {
    push target state on state_stack,
      pop lookahead symbol;
    update S and L;
  } else if A is reduce then {
    output code for this reduction;
    r1 ← LHS symbol of reduction A;
    r2 ← RHS symbol count of A;
    pop r2 states from state_stack,
      update current state S;
    A_tmp ← action for (S, r1);
    push target goto state A_tmp to state_stack;
  } else if A is reduce/reduce conflict then {
    let k = 2;
    while true do
       L_next ← next lookahead;
       if L_next == EOF then
           Report error and exit;
       else
           In LR(k) parsing table, find entry A_next ← ((S, L), L_next);
           if A_next is reduce/reduce conflict then
               L ← L_next;
               k ← k + 1;
           else
               do reduce;
               break out of while loop;
  } else if A is accept then {
    if next token is EOF then {
      is valid accept, exit;
    } else {
      is error, error recovery or exit;
    }
  } else {
    is error, do error recovery;
  }
}
```

The Edge-Pushing Algorithm: An example

- LR(5) Grammar:

  sentence -> male_the male_subject is male_a male_adjective male_object |
  female_the female_subject is female_a female_adjective female_object ;

  male_the -> the ;

  female_the -> the ;

  male_subject -> student | friend ;

  female_subject -> student | friend ;

  male_a -> a ;

  female_a -> a ;

  male_adjective -> tall | short ;

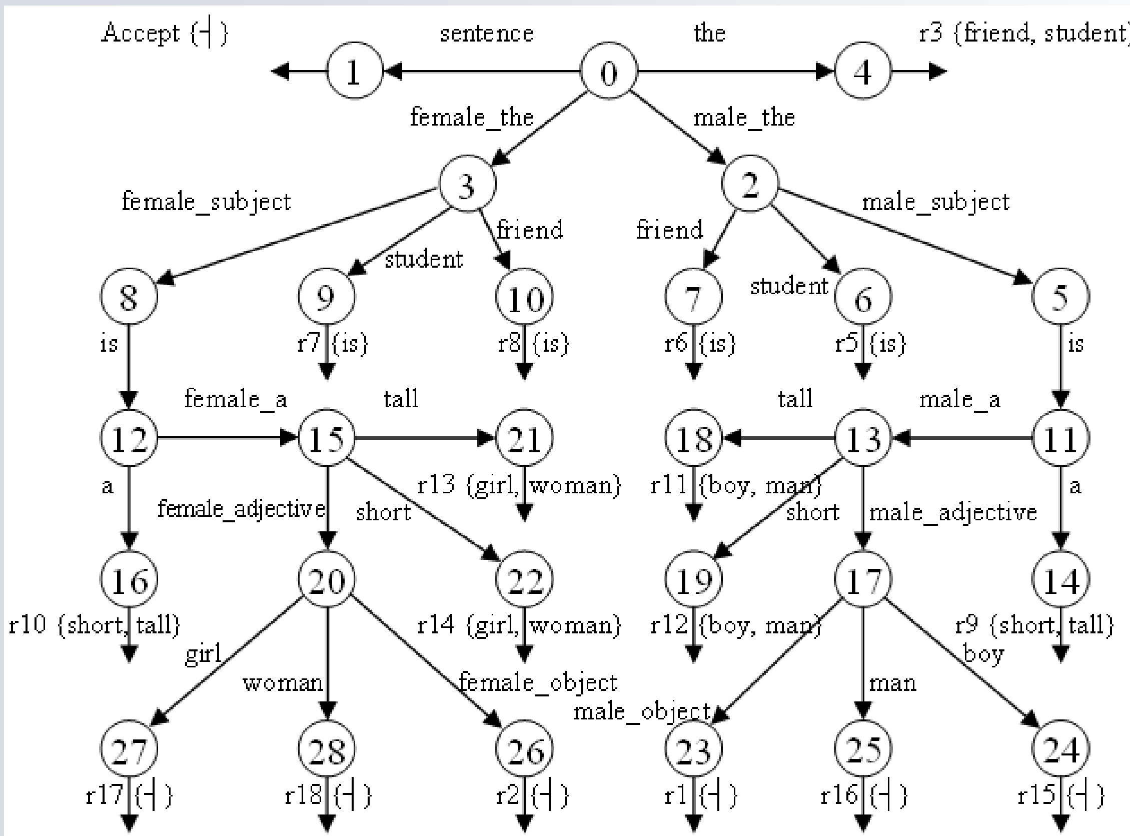
  female_adjective -> tall | short ;

  male_object -> boy | man ;

  female_object -> girl | woman ;

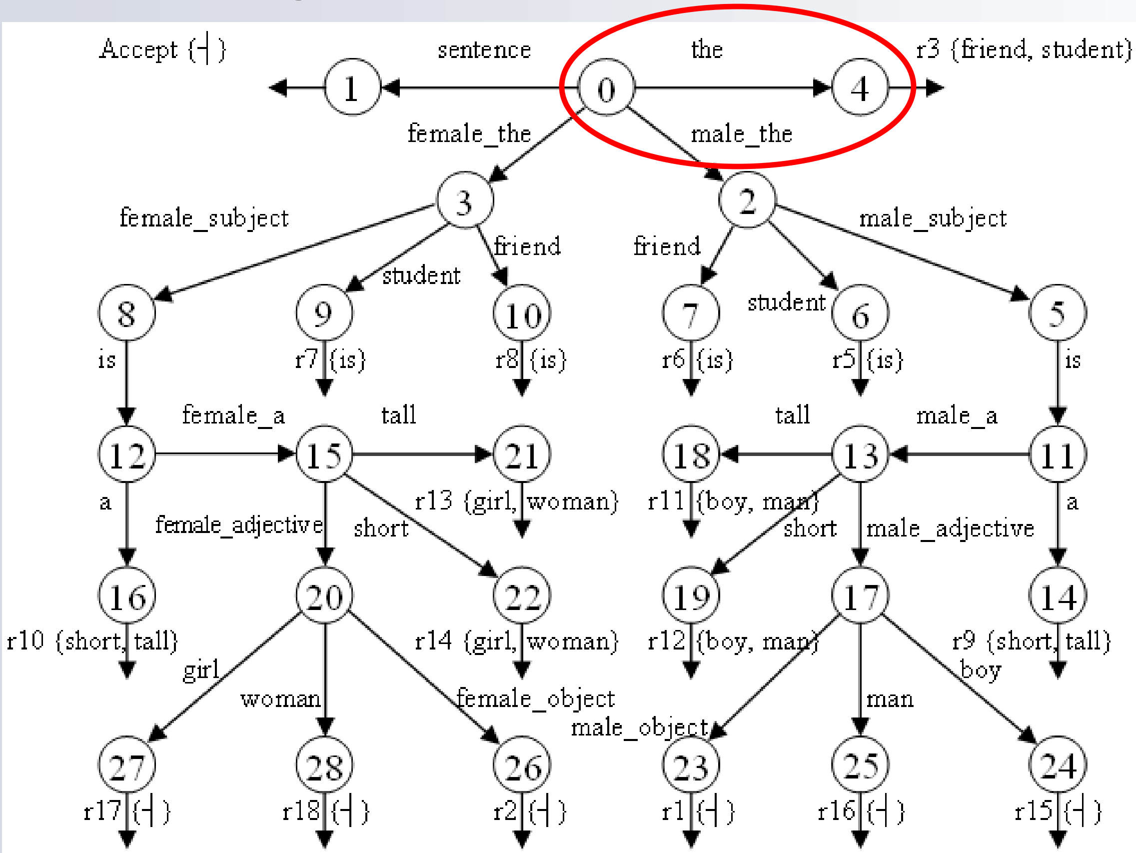
The Edge-Pushing Algorithm: An example

- LR(1) parsing machine



The Edge-Pushing Algorithm: An example

- LR(1) parsing machine



The Edge-Pushing Algorithm: An example

- The part involved in reduce/reduce conflict:

state 0

1 G → • sentence {┤}
2 Sentence → • male_the male_subject is male_a male_adjective male_object {┤}
3 Sentence → • female_the female_subject is female_a female_adjective female_object {┤}
4 Male_the → • the {friend, student}
5 Female_the → • the {friend, student}

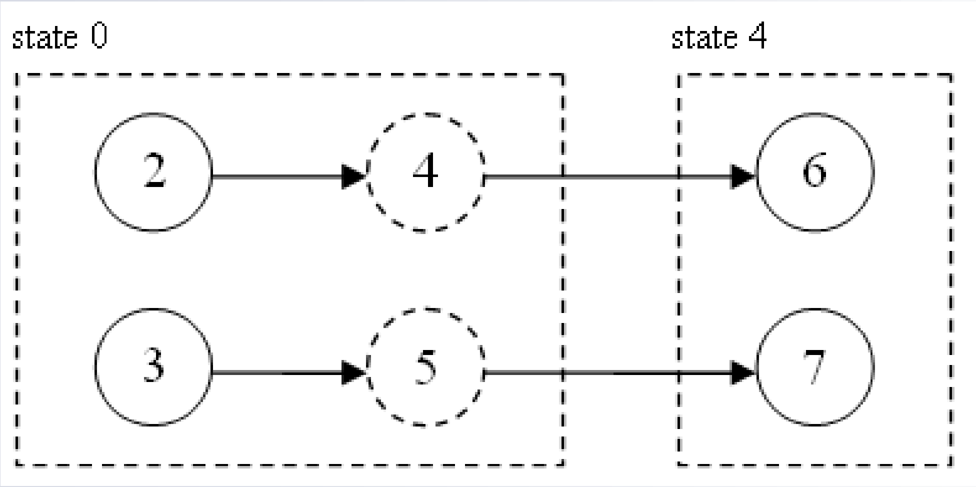
$\xrightarrow{\text{the}}$

state 4

6 male_the → the • {friend, student}
7 female_the → the • {friend, student}

The Edge-Pushing Algorithm: An example

- Lane-tracing result:
  - Configurations 2 and 3 generate non-empty contexts: {friend, student}.



The Edge-Pushing Algorithm: An example

- Edge-pushing Step **1**:
  - lane-tracing
  - theads("male_subject is male_a male_adjective male_object ", **1**)
    - R1: {**friend/student**}
  - theads("female_subject is female_a female_adjective female_object ", **1**)
    - R2: {**friend/student**}

The Edge-Pushing Algorithm: An example

- Edge-pushing Step **2**:
  - theads(“male_subject is male_a male_adjective male_object ”, **2**)
    - R1: {**friend/student is**}
  - theads(“female_subject is female_a female_adjective female_object ”, **2**)
    - R2: {**friend/student is**}

The Edge-Pushing Algorithm: An example

- Edge-pushing Step **3**:
  - theads("male_subject is male_a male_adjective male_object ", **3**)
    - R1: {**friend/student is a**}
  - theads("female_subject is female_a female_adjective female_object ", **3**)
    - R2: {**friend/student is a**}

The Edge-Pushing Algorithm: An example

- Edge-pushing Step **4**:
  - theads(“male_subject is male_a male_adjective male_object ”, **4**)
    - R1: {**friend/student is a tall/short**}
  - theads(“female_subject is female_a female_adjective female_object ”, **4**)
    - R2: {**friend/student is a tall/short**}

The Edge-Pushing Algorithm: An example

- Edge-pushing Step **5**:
  - theads(“male_subject is male_a male_adjective male_object ”, **5**)
    - R1: {**friend/student is a tall/short boy/man**}
  - theads(“female_subject is female_a female_adjective female_object ”, **5**)
    - R2: {**friend/student is a tall/short girl/woman**}

The Edge-Pushing Algorithm: Generated LR(k) Parsing Tables

LR(1) parsing table:

state/token	...	friend	student	...
...				
4		⊗	⊗	
...				

LR(2) parsing table:

(state, LR(1) lookahead)/token	is
(4, friend)	⊗
(4, student)	⊗

LR(3) parsing table:

(state, LR(2) lookahead)/token	a
(4, is)	⊗

LR(4) parsing table:

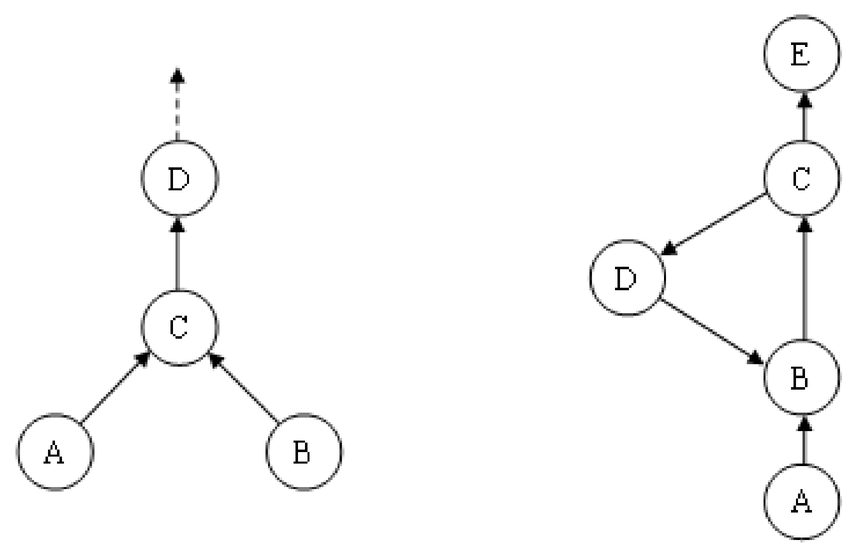
(state, LR(3) lookahead)/token	tall	short
(4, a)	⊗	⊗

LR(5) parsing table:

(state, LR(4) lookahead)/token	man	boy	woman	girl
(4, tall)	r1	r1	r2	r2
(4, short)	r1	r1	r2	r2

The Edge-Pushing Algorithm: Current Problem

- Cycle and branch conditions upon increasing k
  - Need to avoid infinite cycle
  - Can use a cache to solve the problem



(a) Joint (b) Cycle

Edge-Pushing LR(k) Alg. Summary

- Motivation
  - LR(k) has advantage over non-LR(k) algorithms
  - LR(k) is not practically solved due to exponential behavior
- Current research
  - A new LR(k) edge-pushing algorithm is designed
- Future work
  - Branch and cycle conditions need to be solved
  - Other issues to study

$Thead_k(\alpha)$: A New $First_k(\alpha)$ Algorithm

- Overview
  - The $First_k(\alpha)$ Algorithm
    - Evaluate the terminal heads of length k of a string in a grammar
    - Used in LALR, LL and LR parser generations
    - K = 1 is used often.
      - E.g., in Yacc, Bison: LALR(1)
    - k > 1 is rarely used.
      - E.g., in ANTLR: LL(k)

$Thead_k(\alpha)$: A New $First_k(\alpha)$ Algorithm

- Overview
  - The $First_k(\alpha)$ Algorithm
    - Example:

      Given grammar G1: S → MN, N → st, M → bc.
      Evaluate $First_k(\alpha)$, where α = stbc.
      - $First_1(\alpha)$ = {s}
      - $First_2(\alpha)$ = {st}
      - $First_3(\alpha)$ = {stb}
      - $First_4(\alpha)$ = {stbc}

$Thead_k(\alpha)$: A New $First_k(\alpha)$ Algorithm

- Related Work
  - Early work
    - E.g., DeRemer and Ponnello, Kristensen and Madsen
    - Typically vague and imprecise
  - Method of Parr
    - Used in LL(k) computation of ANTLR
    - By walk through a GLA (Grammar Lookahead Automata)

$Thead_k(\alpha)$: A New $First_k(\alpha)$ Algorithm

- Related Work
  - Method of Aho and Ullman (1972)
    - Define set operation $\oplus_k$ (plus k):

      Given 2 sets A and B, S = A $\oplus_k$B is the set of all strings formed from the ordered concatenation of string pairs (a, b), a A, b B, and length of strings in S is <= k.
    - Given a grammar G and a string α in G, $\alpha = X_1X_2...X_n$, then $FIRST_k(\alpha) = FIRST_k(X_1) \oplus_k FIRST_k(X_2) \oplus_k ... \oplus_k FIRST_k(X_n)$
    - $FIRST_k(X_i)$ is built bottom up from $FIRST_0(X_i)$, $FIRST_1(X_i)$, … $FIRST_k(X_i)$

Aho & Ullman. *The Theory of Parsing, Translation, and Compiling*. (1972)

The New $Thead_k(\alpha)$ Algorithm

- The $Thead_k(\alpha)$ Algorithm
  - Evaluate the terminal heads of length k of a string in a grammar
  - A new alternative of $First_k(\alpha)$
  - Overall:
    - Top down (v.s. bottom up of $First_k(\alpha)$)
    - Derives a closure of the initial string

The New Th

- The Thead$_k(\alpha)$ Algorithm in pseudo code

ALGORITHM 1. $THEAD_k(\alpha)$

INPUT: STRING $\alpha = X_1X_2\ldots X_n$; Integer k: length of theads.
OUTPUT: SET H – CONTAINS K-THEADS OF α, AND (OPTIONALLY) SET S – CONTAINS M-THEADS OF α, M < K.

```
H ← Ø
S ← Ø
L ← { h_v(α, k) }
for i = 1 to k do
    foreach string β in L do
      φ = prod(β, i)
       foreach string γ in φ do
         L ← L ∪ { h_v(γ, k) }
       end foreach
    end foreach
    foreach string β in L do
      if β[i] ε NT then L ← L – { β }
    end foreach
    foreach string β in L do
      if h(β, k) ε T^k then
         L ← L – { β }
         H ← H ∪ { h(β, k) }
      end if
    end foreach
    foreach string β in L do
      if |β| < k AND β ε T^|β| then
         L ← L – { β }
         S ← S ∪ { β }
      end if
    end foreach
    if L == Ø then stop
end for
```

The New Thead$_k(\alpha)$ Algorithm

- The Thead$_k(\alpha)$ Algorithm in plain English:

for 1≤ i ≤n and while L is not empty, do:

for j = 1, 2, … until the last member of L is the ith one, and no new members are added to L by the following procedure, do:

Add to the end of L the result of applying all possible productions to the i-th symbol in the j-th member of L, omitting strings that are already in L, and truncating all members added which have k or more symbols that do not vanish, by deleting the part of the string following the k-th symbol that does not vanish;

next i

Remove from L all strings whose i-th symbol is a non-terminal;
Remove from L all strings which possess k-heads consisting entirely of terminals, and add the k-heads involved to the set H;
Remove from the list all strings of length < k which consist entirely of Terminal and add these to the set S;

next j

The New $Thead_k(\alpha)$ Algorithm

- Comparison with other algorithms
  - $Thead_k(\alpha)$ and $First_k(\alpha)$ are standalone methods, other methods depend on parsing machine to work.
  - $First_k(\alpha)$ is a bottom up approach that builds and caches a table of $First_k(X_i)$ for each symbol X_i first, which can be a overhead.
  - $Thead_k(\alpha)$ is top down, only involves symbols in the given string.
  - $First_k(\alpha)$ keeps track of intermediate terminal strings of length < k, $Thead_k(\alpha)$ does not need to.

$Thead_k(\alpha)$ Algorithm - Example

- Given grammar G3: $X \rightarrow Y \mid x \mid \varepsilon$, $Y \rightarrow Z \mid y \mid \varepsilon$, $Z \rightarrow X \mid z \mid \varepsilon$, $U \rightarrow u$. Find the 2-theads of string $\alpha = XYZU$.
  – Solve this using $Thead_k(\alpha)$ and $First_k(\alpha)$

1) By Thead$_k$(α)

Table 1. First round of operation for i = 1

i	j	String added to L	String Sequence Number
1	1	XYZU	1
		YYZU	2
		xYZU	3
		YZU	4
	2	ZYZU	5
		yYZU	6
	3		
	4	ZZU	7
		yZU	8
		ZU	9
	5	zYZU	10
	6		
	7	XZU	11
		zZU	12
	8		
	9	XU	13
		zU	14
		U	15
	10		
	11	xZU	16
	12		
	13	YU	17
		xU	18
	14		
	15	u	19
	16		
	17	yU	20
	18		
	19		
	20		

Table 2. The second round where i = 2

i	j	String added to L	String Sequence Number
2		xYZU	1
		yYZU	2
		yZU	3
		zYZU	4
		zZU	5
		zU	6
		xZU	7
		xU	8
		yU	9
	1	xZZU	10
		xy	11
	2	yZZU	12
		yy	13
	3	yXU	14
		yz	15
	4	zZZU	16
		zy	17
	5	zXU	18
		zz	19
	6	zu	20
	7	xXU	21
		xz	22
	8	xu	23
	9	yu	24
	10	xXZU	25
	11		
	12	yXZU	26
	13		
	14	yYu	27
		yx	28
	15		
	16	zXZU	29
	17		
	18	zYU	30
		zx	31
	19		
	20		
	21	xYU	32
		xx	33
	22		
		…	
	32		
	33		

2) By $First_k(\alpha)$

$F_i(p) = \{p\}$, for all $p \in \{x, y, z, u, \varepsilon\}$, and $i \geq 0$.
$F_0(X) = \{x, \varepsilon\}$
$F_0(Y) = \{y, \varepsilon\}$
$F_0(Z) = \{z, \varepsilon\}$
$F_0(U) = \{u\}$
$F_1(X) = \{x, y, \varepsilon\}$
$F_1(Y) = \{y, z, \varepsilon\}$
$F_1(Z) = \{z, x, \varepsilon\}$
$F_1(U) = \{u\}$
$F_2(X) = \{x, y, z, \varepsilon\}$
$F_2(Y) = \{x, y, z, \varepsilon\}$
$F_2(Z) = \{x, y, z, \varepsilon\}$
$F_2(U) = \{u\}$

From this point on $F_i(S) = F_2(S)$ for $i \geq 3$, $S = X, Y, Z, U$. It converges here. Therefore:
$FIRST_2(X) = F_2(X) = \{x, y, z, \varepsilon\}$
$FIRST_2(Y) = F_2(Y) = \{x, y, z, \varepsilon\}$
$FIRST_2(Z) = F_2(Z) = \{x, y, z, \varepsilon\}$
$FIRST_2(U) = F_2(U) = \{u\}$

Finally, we can calculate $FIRST_k(\alpha) = FIRST_2(XYZU)$
$= FIRST_2(X) \oplus_2 FIRST_2(Y) \oplus_2 FIRST_2(Z) \oplus_2 FIRST_2(U)$
$= \{x, y, z, \varepsilon\} \oplus_2 \{x, y, z, \varepsilon\} \oplus_2 \{x, y, z, \varepsilon\} \oplus_2 \{u\}$
$= \{xx, xy, xz, xu, yx, yy, yz, yu, zx, zy, zz, zu, u\}$

$Thead_k(\alpha)$ - Performance

- Summary
  - $Thead_k(\alpha)$ performs better than $First_k(\alpha)$ when averaged over large number of inputs
  - Let n be the number of non-vanishable symbols in the input string, if $n >> k$, then $Thead_k(\alpha)$ performs much better

Summary

- We reviewed these algorithms related to LR:
  - Knuth Canonical LR(1) Algorithm
  - Pager's LR(1) Practical General Method
  - Pager's LR(1) Lane-tracing Algorithm
  - Pager's Unit Production Elimination Algorithm
  - Extension to Pager's Unit Production Elimination Algorithm
  - Edge-pushing LR(k) algorithm

Surfing - Hawaii

Wrap Up

- We reviewed **Hyacc** today:
  - **Introduction:**
    Overview - Problem and Motivation - Literature Review - Approach and Results - Related work - Contributions and Limitations
  - **Implementation:**
    Architecture - Parse engine - Storing the parsing table - Handling precedence and associativity - Error handling - Data structures - Future work
  - **Usage:**
    The Project website - What is Hyacc, and why it is special - Features - Download - Readme.PDF - Hyaccmanpage.html - License - History - References - Example
  - **Theory:**
    Knuth Canonical LR(1) Algorithm - Pager's LR(1) Practical General Method - Pager's LR(1) Lane-tracing Algorithm - Pager's Unit Production Elimination Algorithm & Extension - Edge-pushing LR(k) algorithm

Wrap Up: Contributions

- Contribution
  - Implemented an efficient, practical parser generator: Hyacc
  - Evaluated different LR(1) algorithms
  - Showed LR(1) parser generation is practical
  - Potential influence on the industry
    - LR(1) to replace LALR(1)

Wrap Up: Limitations

- Limitations
  - Hyacc feature needs enrichment
    E.g. Several more directives in Yacc still need implementation:
  - Parse engine available in C only
  - LR(k) still incomplete

Wrap Up: Summary

- Motivation
  - LR(1) has advantage over non-LR(1) algorithms
  - LR(1) problem: performance
  - Reduced space LR(1) algorithms exist but need study
  - Industry need

Wrap Up: Summary

- Motivation
  - LR(1) has advantage over non-LR(1) algorithms
  - LR(1) problem: performance
  - Reduced space LR(1) algorithms exist but need study
  - Industry need
- Current work
  - Measured and evaluated algorithms
  - Released LR(0)/LALR(1)/LR(1)/partial LR(k) parser generator Hyacc

Wrap Up: Summary

- Motivation
  - LR(1) has advantage over non-LR(1) algorithms
  - LR(1) problem: performance
  - Reduced space LR(1) algorithms exist but need study
  - Industry need
- Current work
  - Measured and evaluated algorithms
  - Released LR(0)/LALR(1)/LR(1)/partial LR(k) parser generator Hyacc
- Hyacc Future work
  - More features
  - Support more languages in parse engine
  - LR(k)


Questions?
Secrete Beach @ Kauai - Hawaii