An Algorithm for Parsing Flow Graphs

Daniel Carl Brotsky

Artificial Intelligence Laboratory
Massachusetts Institute of Technology

This report is a revised version of a thesis submitted to the Department of Electrical Engineering and Computer Science on February 8, 1983, in partial fulfillment of the requirements for the degree of Master of Science.

Abstract

This report describes research about *flow graphs* – labeled, directed, acyclic graphs which abstract representations used in a variety of Artificial Intelligence applications. Flow graphs may be derived from *flow grammars* much as strings may be derived from string grammars; this derivation process forms a useful model for the stepwise refinement processes used in programming and other engineering domains.

The central result of this report is a parsing algorithm for flow graphs. Given a flow grammar and a flow graph, the algorithm determines whether the grammar generates the graph and, if so, finds all possible derivations for it. The author has implemented the algorithm in LISP.

The intent of this report is to make flow-graph parsing available as an analytic tool for researchers in Artificial Intelligence. The report explores the intuitions behind the parsing algorithm, contains numerous, extensive examples of its behavior, and provides some guidance for those who wish to customize the algorithm to their own uses.

CONTENTS

Acknowledgements vii

1. Introduction 1
- 1.1. Motivation . . . 2
- 1.2. Background . . . 3
- 1.3. Structure of this Report . . . 3

2. Definitions 5
- 2.1. Flow Graphs . . . 5
- 2.2. Flow Grammars . . . 7
- 2.3. Flow Grammar Derivations . . . 10

3. Motivation for the Algorithm 13
- 3.1. Non-Deterministic String Parsers . . . 13
  - 3.1.1. An Example . . . 15
  - 3.1.2. Discussion . . . 19
- 3.2. Simulating the State-based Parser . . . 20
  - 3.2.1. Preliminaries . . . 20
  - 3.2.2. Multiple-Call Collapsing . . . 21
  - 3.2.3. Left-recursion . . . 26
  - 3.2.4. Duplicate-Item Merging . . . 27
  - 3.2.5. The String Algorithm . . . 29
  - 3.2.6. Why is this Earley's Algorithm . . . 31
  - 3.2.7. Using the Algorithm to produce Parse Trees . . . 32

4. The Algorithm 35
- 4.1. Non-Deterministic Graph Parsers . . . 35
  - 4.1.1. Reading a Flow Graph . . . 36
  - 4.1.2. Flow Graph Recognizers . . . 39
  - 4.1.3. States . . . 40
  - 4.1.4. State Transition Functions . . . 41
  - 4.1.5. Linkage Mechanism . . . 46
  - 4.1.6. Flow Graph Parsers . . . 48
- 4.2. The Parsing Algorithm . . . 58

4.2.1. Preliminaries 58
4.2.2. Optimizations 61
4.2.3. Examples 61
4.2.4. Algorithm Description 130

5. Discussion **135**
5.1. Flow Graphs and Grammars 135
5.2. Applicability of the Algorithm 136
5.3. Correctness 138
5.4. Complexity Analysis 138

References **143**

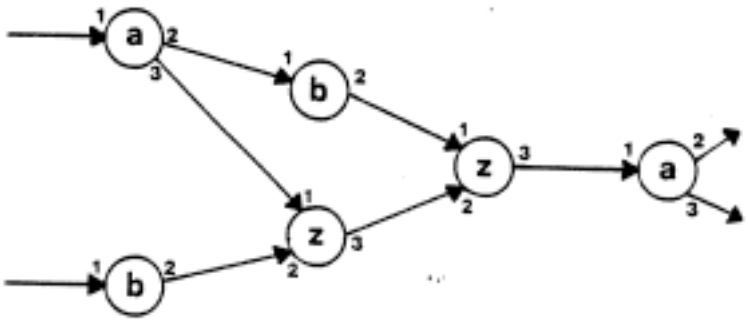
Acknowledgements

This report has been a long, long time in the making, so I have had time to get help from many, many people. Some helped me with my research, some helped me with my life, and some helped me with both. I am grateful to all of them, and especially to Charles Rich, who has stuck with me through a few fast times and many slow ones.

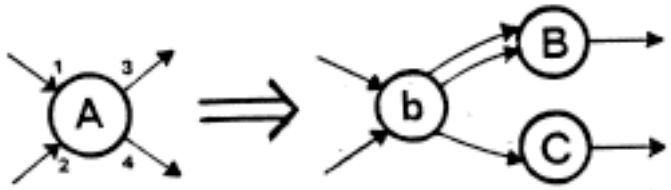
Chapter 1.

Introduction

This report summarizes research about *flow graphs*, a graph-based representation abstracted from those used in a variety of Artificial Intelligence applications. A flow graph is a labeled, directed, acyclic graph whose nodes are annotated with *ports*—positions at which edges enter or leave the node. Here is an example of a flow graph:



We can generate complex flow graphs from simple ones by replacing single nodes with multi-node subgraphs. The obvious analogy between this process and that of string derivation from a context-free grammar gives rise to the notion of a *flow grammar*: a set of rewriting rules which specify how to replace given nodes with pre-specified subgraphs. Here is an example of a rule from a flow grammar:



The central result of this report is a parsing algorithm for flow graphs. Given a flow grammar and a flow graph, the algorithm determines whether the grammar generates the graph and, if so, finds all possible derivations for it. The algorithm runs in time polynomial in the number of nodes in the input graph, with an exponent and constant of proportionality determined by the input grammar. The author has implemented the algorithm in LISP.

1.1. Motivation

The work described here grew out of the author's research into automated program analysis [Brotsky 1981], done as part of the Programmer's Apprentice project at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology [Rich and Waters 1981]. In the work of that group, programs are represented as annotated graphs, called *plans*, whose nodes stand for operations and whose arcs indicate control and data flow between the nodes. (Plans are additionally annotated with a great deal of other information about the program they represent, but the details of these annotations do not concern us here. Interested readers should consult [Rich 1980].)

The author's idea was that the stepwise-refinement process, wherein high-level program operations are implemented as groups of lower-level operations, could naturally be modeled as a plan-rewriting process. Thus, flow graphs were developed as abstractions of plan structure, flow grammars were developed to encode allowable derivation steps, flow-graph derivations were developed as models of plan derivations, and structural program analysis could be effected through parsing.

This program-analysis work is continuing, but does not concern us here. Flow graphs, while developed as *ad hoc* abstractions of plans, are general enough to serve as abstractions of the graphical representations of other

domains. The intent of this report is to make flow graph parsing available as an analytic tool for AI researchers in these other domains.

1.2. Background

The structure of flow graphs and flow grammars has been influenced by early work on *web grammars* [Pfaltz and Rosenfeld 1969; Montanari 1970; Pavlidis 1972], but none of this work was concerned with parsing. The structure of our parsing algorithm arose from careful study of Earley's algorithm [Earley 1969] and Donald E. Knuth's seminal work on LR(k) string grammars [1965].

1.3. Structure of this Report

Chapter 1 of this report is this introduction. Chapter 2 describes flow graphs, flow grammars, and flow-graph derivations in detail. Chapter 3 presents a derivation of Earley's algorithm which differs considerably from those found in standard sources. This derivations is given as background for the very similar derivation of the graphs parsing algorithm presented in chapter 4. Finally, chapter 5 discusses flow graphs, grammars, and the parsing algorithm. This discussion includes a brief complexity analysis of the algorithm, and suggestions for related research.

Chapter 2.

Definitions

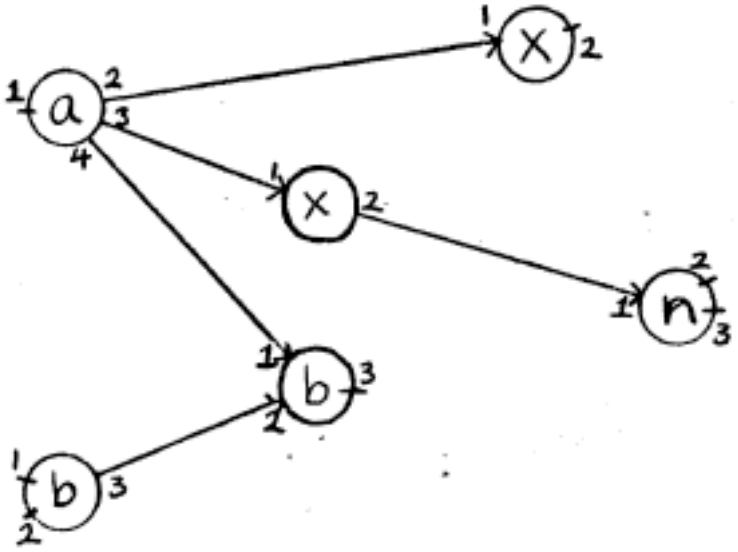
In this chapter we define flow graphs and flow grammars, and give the mechanism by which a grammar derives a graph.

2.1. Flow Graphs

A flow graph is a labeled, acyclic, directed graph whose nodes and edges are restricted in a variety of ways:

- The label of each node is called its *type*.
- Each node has a set of *input ports* and a set of *output ports*. These two sets are disjoint. All nodes with the same type have the same input and output port sets.
- The input and output port sets of flow graph nodes are never empty. That is, all nodes have at least one input and one output port.
- Edges in flow graphs do not run merely from one node to another, but from a particular output port of one node to a particular input port of another. No two edges may enter or exit from the same port, so a node can be adjoined by only as many edges as it has ports.

Intuitively, a flow graph looks like this:

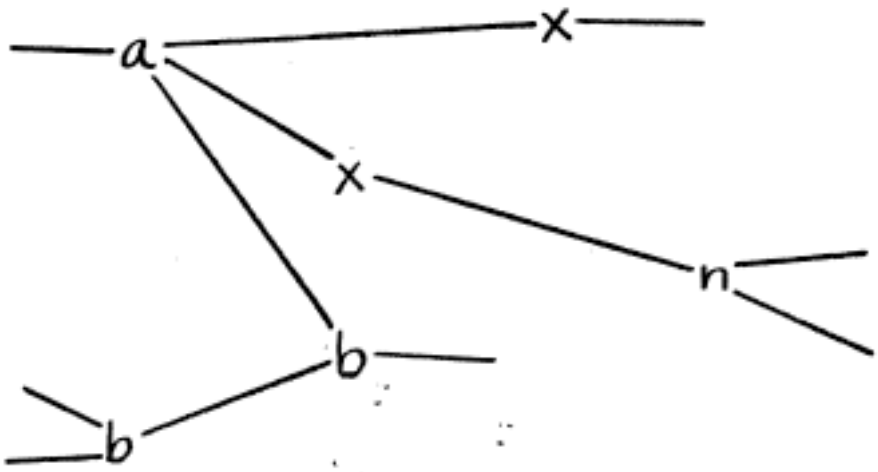


Notice that ports (which are identified by numeric annotations on the nodes) need not have edges adjoining them. Any input (or output) port in a flow graph that does not have an edge running into (or out of) it is called an *input* (or *output*) of that graph.

Notation

We will always direct our flow-graph diagrams from left to right. We will often subscript node types so as to make them into unique labels. (This avoids awkward constructions such as "the third a from the bottom-left.") When we do not care which port an edge adjoins, or if this is made clear from context, we will omit port annotations. If we omit all the ports annotations on a node, we will often omit the circle drawn around the node's label. Finally, we will always emphasize the inputs and outputs of graphs by adjoining them with edge stubs, called the *leading* and *trailing* edges of the graph.

Here is the graph we saw above written using the conventions just described:



We will use this form whenever possible.

Terminology

The *linkage information* for a node in a graph is a set of ⟨port, edge⟩ pairs detailing which edge adjoins each port on that node. For example, figure 2.1 shows a graph whose edges have been labeled for easy reference. The linkage information for nodes a_1 and z_2 in this graph is:

a_1	z_2
$\langle 1, e_1 \rangle$	$\langle 1, e_6 \rangle$
$\langle 2, e_3 \rangle$	$\langle 2, e_7 \rangle$
$\langle 3, e_4 \rangle$	$\langle 3, e_8 \rangle$

In keeping with our left-to-right conventions, that portion of a node's linkage information which involves only input (resp. output) edges is called its *left-linkage* (resp. *right-linkage*) information.

2.2. Flow Grammars

Flow grammars are a generalization of context-free string grammars. Essentially, a flow grammar is a set of rewriting rules, where each rule explains

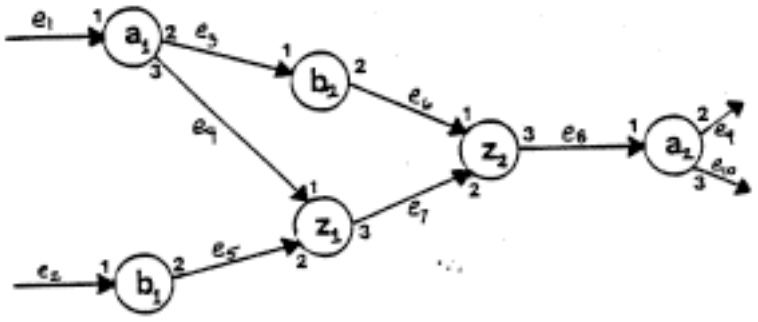


Figure 2.1. A flow graph. The edges of this graph have been labeled for easy reference.

how to replace a node in a graph with a particular sub-graph. Just as a string grammar gradually rewrites a single-element string as a longer and longer string, a flow grammar gradually rewrites a single-node graph as a larger and larger graph.

More precisely, a *flow grammar* G consists of 4 parts: a set P of *productions*, two disjoint sets of types N—the *non-terminals*—and T—the *terminals*, and a distinguished non-terminal type S—the *start type* of G. Each production in P consists of three parts: two flow graphs and a list of port correspondences. The first of the two flow graphs—the production's *left-hand side*—consists of a single node whose type must be from N. The second of the flow graphs—the *right-hand side*—consists of nodes whose types are from $N \cup T$. The left and right-hand sides must have the same number of inputs and outputs, and the list of port correspondences is a 1–1 correspondence between inputs and outputs of the two sides.

A flow grammar is shown in figure 2.2 Each rule maps a single node to a graph. The left-hand side node of each rule must be a non-terminal, that is, of a non-terminal type, while the right-hand side graph can mix types at will. (We will indicate non-terminal types with capital letters, and terminal types with lower case letters.)

The inputs of the left-hand side of a rule correspond one-to-one with the inputs of the right-hand side, as do the outputs. Where clarity is needed, we will indicated this relationship by drawing lines between the edge stubs adjoining corresponding ports, as was done above. Where it's clear, however,

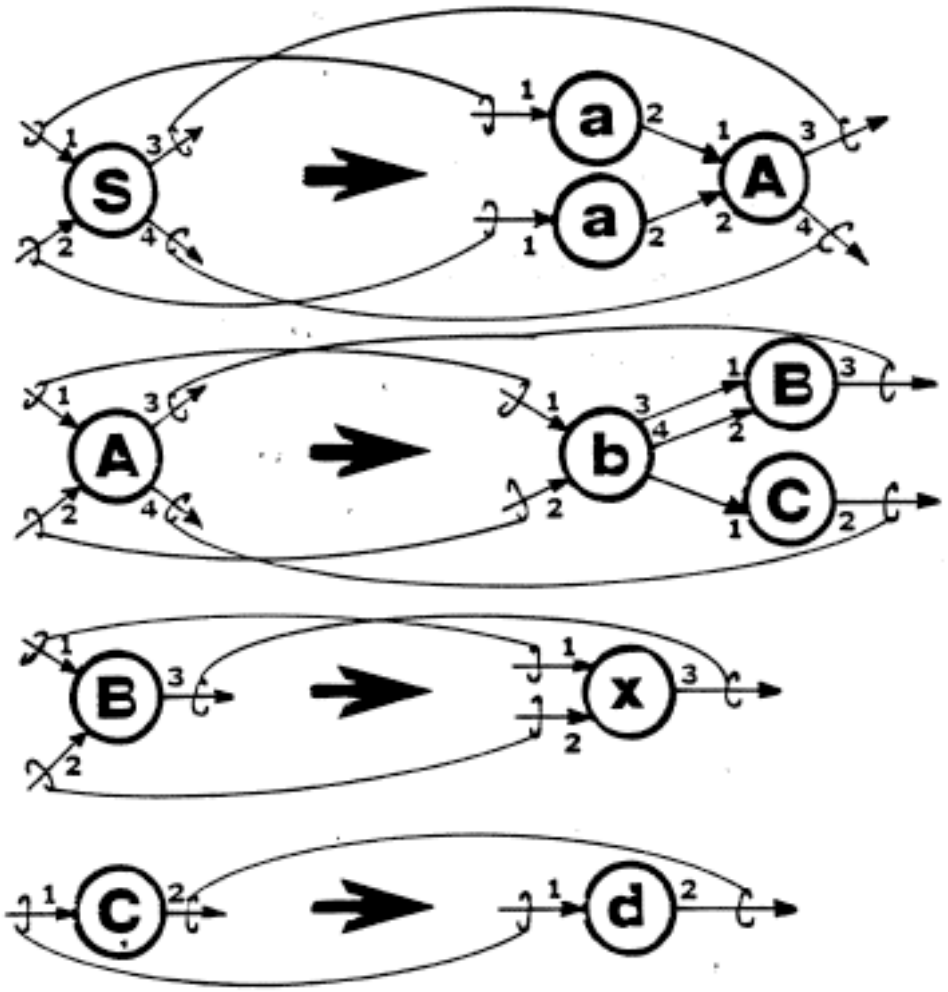
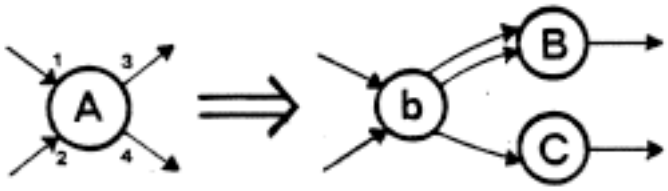


Figure 2.2. A flow grammar.

we will indicate the correspondence simply by mirroring the alignment of left-hand side edge stubs with those of the right-hand side. For example, the second rule in the above grammar could have been written as follows:



Notice that there is no flow-grammar equivalent of an "ϵ-rule" in a string grammar: that is, there are no flow grammar rules whose right-hand sides are empty. This is because it is meaningless to replace a node in a graph with nothing: the edges that were adjoined to that node must go somewhere.

2.3. Flow Grammar Derivations

Flow graphs are derived from flow grammars in the expected way. We start with a graph consisting of a single S-node and then rewrite it with an applicable rule from the grammar. This gives us a flow graph. If there are no non-terminals in the derived graph, the derivation stops. Otherwise, we pick a non-terminal and a rule that derives it, and replace the non-terminal by the right-hand side of the rule. This gives us another graph, and the whole process iterates.

Of course, when we replace a non-terminal by a right-hand side that derives it, we have to do something with the edges that adjoined that non-terminal. This is what the port correspondences in rules are for: if p was a port on the replaced non-terminal, then the edge that adjoined p (if any) is made to adjoin p's corresponding port in the replacement graph. The restrictions on rule formation insure that there is never any question as to how a right-hand side should replace a left-hand side. For example, figure 2.3 shows the derivation of a graph from the grammar given in the last section.

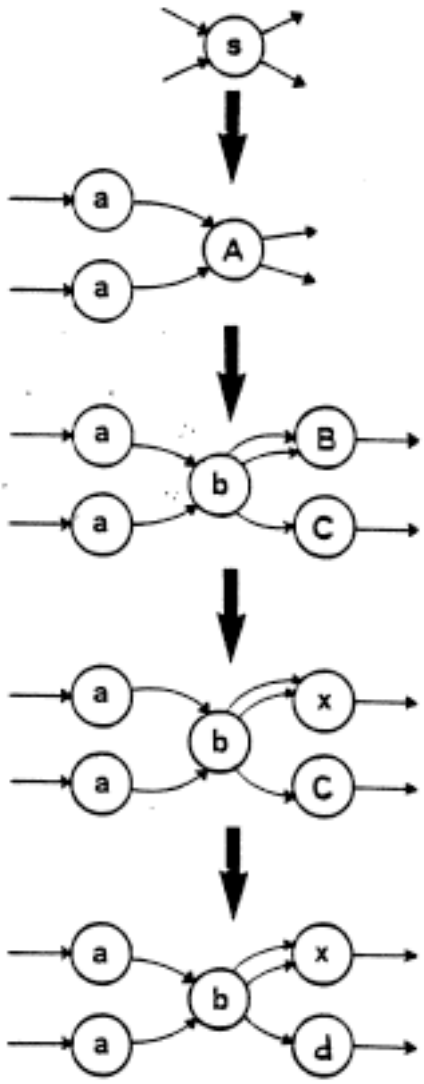


Figure 2.3. Sample Flow Graph Derivation

Chapter 3.

Motivation for the Algorithm

Earley's algorithm is a well-known string parsing algorithm [Earley 1969]. It takes a string grammar and a string as input, and determines all possible derivations of that string from that grammar. The output of the algorithm is a list of representations known as *items*; the acceptability and derivations of the input string are encoded in this list.

This section presents a derivation of Earley's algorithm that differs significantly from those found in standard sources. For a given input grammar and string, we first construct a non-deterministic stack-based parser for the grammar. We then deterministically simulate the behavior of that parser when run on the input string; the representations of the parser's configurations generated in this simulation will be homomorphic to the items produced by Earley's algorithm when run on the same input.

The derivation given here is presented as background for the very similar derivation of our flow graph parsing algorithm given in the next chapter. Much of the complexity inherent in both algorithms arises from optimizations that are employed in the simulation process; since the intuitions underlying these optimizations are the same in both the string and graph cases, we believe that presenting them in the relatively familiar context of string parsing will make their use in graph parsing more comprehensible.

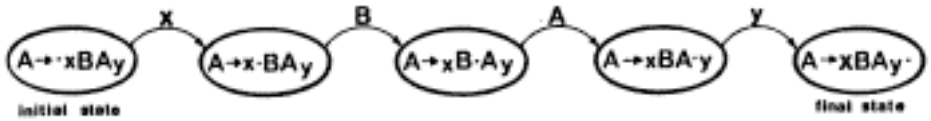
3.1. Non-Deterministic String Parsers

Given a context free grammar G with productions $P_1, \ldots, P_n$ and start symbol S, the following construction yields a non-deterministic stack-based parser for G:

1. Construct a state-machine recognizer for the right-hand side of each P_i. A state in the recognizer R_i constructed for rule P_i will consist of a copy of P_i's right-hand side with a dot placed just to the left or right of one of its symbols; the state set of R_i will consist of all the states formed in this way. The state transition function of R_i will map ⟨state, symbol⟩ pairs to states; each state with a dot to the left of some symbol s will have a transition on s to the state whose dot is just to the right of s. The initial state of R_i will be the state with a dot to the left of the leftmost symbol in P_i's right-hand side: its final (accepting) state will be the state whose dot is to the right of the rightmost symbol in P_i's right-hand side.
   For example, if P_i is the production $A \rightarrow xBAy$, then the recognizer for P_i will have the following five states:

$$[A \rightarrow \cdot xBAy]$$
$$[A \rightarrow x \cdot BAy]$$
$$[A \rightarrow xB \cdot Ay]$$
$$[A \rightarrow xBA \cdot y]$$
$$[A \rightarrow xBAy \cdot]$$

   and the transition diagram for P_i's recognizer would look as follows:



2. Create a state-based machine P whose state space and transition function is the union of all those of the recognizers for the P_i. The initial and final states of P are the initial and final states of the recognizer for S.
3. Convert P to a non-deterministic stack machine by adding a stack and instructions as follows: For each state s which has a transition on a non-terminal input, associated instructions to that state which (i) push the state onto the stack and (ii) put P into the start state of the recognizer for some production which derives that non-terminal. (If the non-terminal on which a state has a transition has n possible derivations, then this step will associate n instructions with that state.)
4. Complete P by adding instructions as follows: To each accepting state of a recognizer for a P_i, add an instruction which (i) pops a state off the top of the stack and (ii) put P into the state which is led to by the popped state's transition on the non-terminal derived by P_i.

The machine P built in this way is a top-down non-deterministic parser for sentences derived from G.[1] It operates by reading symbols one at a time from the input and making appropriate state transitions as it does so. Whenever it enters a state which has associated stack instructions, it chooses one of those instructions and executes it. (The choice involved here is what makes the parser non-deterministic.) We first consider an example of such a parser, and then discuss some implications of the construction technique.

3.1.1. An Example

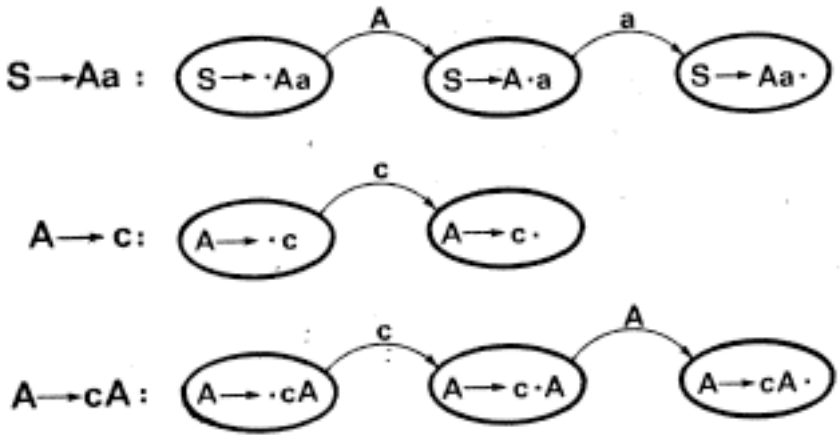
Consider the following grammar G:

$$S \rightarrow Aa$$
$$A \rightarrow c$$
$$A \rightarrow cA$$

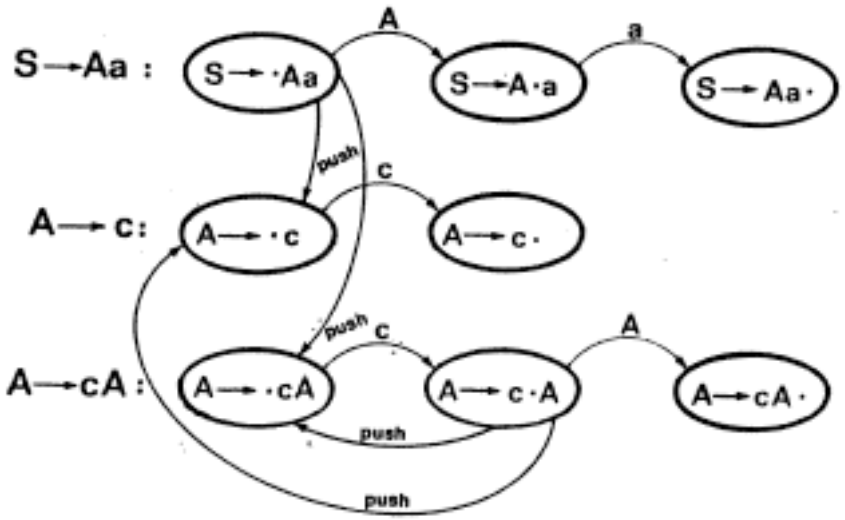
G derives all strings consisting of one or more c's followed by an a. We will carry out the construction described above so as to produce a parser for G, and then run this parser on the input cca.

First, we construct state machines which recognize each of the productions in G. These are as follows:

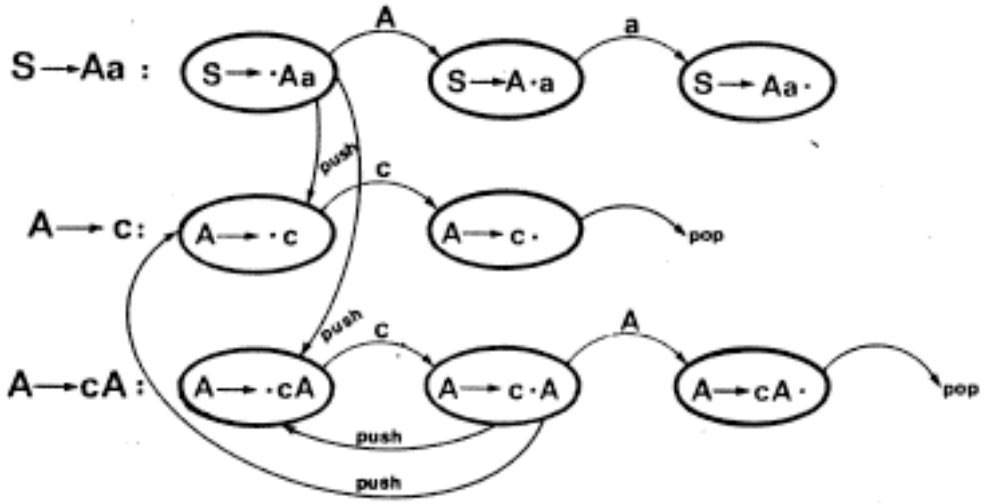
[1] Actually, this machine is merely an acceptor for such sentences. However, if we have each push instruction in P output the non-terminal which gave rise to the push, and we output each input symbol as it is read, then each accepting path through P will output a leftmost derivation for the sentence accepted. Thus, we view P as a parser.



Now we create the union machine and replace non-terminal transitions with pushes:



Finally, we complete the construction by adding stack pops on reductions:



This completes our parser. We will represent a given configuration of the parser as

$$\langle \text{input position} \rangle \quad [\langle \text{state} \rangle; (\langle \text{stack top} \rangle; \ldots : \langle \text{stack bottom} \rangle)]$$

where the states are represented using the dot representation shown above. (Recall that stack entries are just states.) For example, when running this parser on the string cca, it starts in the following configuration:

$$0(\epsilon) \quad [S \to \cdot\, Aa, ()]$$

The state $[S \to \cdot\, Aa]$ has two transitions on *push* instructions. The parser must choose one of the two, leading it into one of these two configurations:

$$\begin{array}{ll} 0(\epsilon) & [A \to \cdot\, c, (S \to \cdot\, Aa)] \\ & [A \to \cdot\, cA, (S \to \cdot\, Aa)] \end{array}$$

At this point, no more state transitions are possible without reading an input symbol. Thus, the parser will read the first c, leading it into one of these configurations:

$$\begin{array}{ll} 1(c) & [A \to c \cdot, (S \to \cdot\, Aa)] \\ & [A \to c \cdot A, (S \to \cdot\, Aa)] \end{array}$$

The first of these two configurations is an accepting state for the rule $A \to c$, and allows a *pop* into the following configuration:

$$1(c) \quad [S \to A \cdot a, ()]$$

while the second configuration is in a state containing *push* transitions to the these configurations:

$$\begin{array}{ll} 1(c) & [A \to \cdot\, c, (A \to c \cdot A; S \to \cdot\, Aa)] \\ & [A \to \cdot\, cA, (A \to c \cdot A; S \to \cdot\, Aa)] \end{array}$$

Once again, no more state transitions are possible without reading another input symbol.

We can summarize all the possible computations so far in the following tabular fashion:

$$\begin{array}{ll} 0(\epsilon) & [S \to A \cdot a, ()] \\ & [A \to \cdot\, c, (S \to \cdot\, Aa)] \\ & [A \to \cdot\, cA, (S \to \cdot\, Aa)] \end{array}$$

$$
\begin{array}{ll}
1(c) & [A \to c\cdot, (S \to \cdot Aa)] \\
 & [S \to A\cdot a, ()] \\
 & [A \to c\cdot A, (S \to \cdot Aa)] \\
 & [A \to \cdot c, (A \to c\cdot A; S \to \cdot Aa)] \\
 & [A \to \cdot cA, (A \to c\cdot A; S \to \cdot Aa)]
\end{array}
$$

We will use this form extensively to summarize actions of these parsers; for example, the remainder of the run of this parser on the string cca goes as follows:

$$
\begin{array}{ll}
2(cc) & [A \to c\cdot, (A \to c\cdot A; S \to \cdot Aa)] \\
 & [A \to cA\cdot, (S \to \cdot Aa)] \\
 & [S \to A\cdot a, ()] \\
 & [A \to c\cdot A, (A \to c\cdot A; S \to \cdot Aa)] \\
 & [A \to \cdot c, (A \to c\cdot A; A \to c\cdot A; S \to \cdot Aa)] \\
 & [A \to \cdot cA, (A \to c\cdot A; A \to c\cdot A; S \to \cdot Aa)] \\
3(cca) & [S \to Aa\cdot, ()]
\end{array}
$$

3.1.2. Discussion

From one point of view, this construction technique produces classic recursive-descent parsers, such as those presented in undergraduate compiler classes. Where a recursive-descent parser would have a subroutine dedicated to the recognition of each rule's right-hand side, these parsers have state-machine recognizers, and these recognizers are linked together via a "subroutine-call" mechanism based on a stack. In what follows, we will often describe the actions of these parsers using terminology suggested by this metaphor.

From another point of view, this construction technique produces classic push-down automatata. The state-based machines constructed for each grammar rule are finite-state recognizers for the right-hand sides of those rules, and the dots in their states indicate the expected position of a read head in the parser's input. In this context, the stack push and pop instructions act as ϵ-transitions between the various recognizers, and the parser appears as a non-deterministic push-down automaton whose finite state control compares substrings of the input against the right-hand side of grammar rules and whose stack monitors the center-embeddedness of the input as a whole. In what follows, we will also use terminology suggested by this metaphor.

3.2. Simulating the State-based Parser

In order to simulate a parser constructed as above, we must perform all the actions which follow from all possible non-deterministic choices. The recursive-descent metaphor suggests that we do this with a sequential approach that employs backtracking, while the automaton metaphor suggests a parallel approach in which one simulator state represents a number of reachable parser states. We shall adopt this latter approach, and keep track of all the ⟨state, stack⟩ pairs reachable by a parser at each step of the input. The result of the simulation will be a sequence of lists of reachable configurations, much like those used in the sample parse above.

3.2.1. Preliminaries

We use here a slightly different representation for the stack segment of a configuration than we did in the sample parse above. In line with our subroutine-call point of view on push operations, we will not keep the whole stack with each configuration. Rather, each time we make a transition to the initial state of a recognizer, we will keep a *return pointer* which indicates the configuration we were in before entering that state.

For example, we presented above the configuration sequence for the parse of cca. If we make the representational changes just described, we obtain the following, more compact representation, in which we have subscripted the configurations for use in return pointers:

$0(\epsilon)$ $[S \to \cdot\, Aa,\ ]_1$
$[A \to \cdot\, c, 1]_2$;expand A from item 1
$[A \to \cdot\, cA, 1]_3$

$1(c)$ $[A \to c\,\cdot, 1]_4$
$[S \to A \cdot a,\ ]_5$;return to item 1
$[A \to c \cdot A, 1]_6$
$[A \to \cdot\, c, 6]_7$
$[A \to \cdot\, cA, 6]_8$

$2(cc)$ $[A \to c\,\cdot, 6]_9$
$[A \to cA\,\cdot, 1]_{10}$
$[S \to A \cdot a,\ ]_{11}$
$[A \to c \cdot A, 6]_{12}$
$[A \to \cdot\, c, 12]_{13}$

$$[A \rightarrow \cdot cA.\ 12]_{14}$$

3(cca) $[S \rightarrow Aa \cdot .\ ]$;accept

We call the [⟨state⟩, ⟨return pointer⟩] pairs used here *items*, to distinguish them from configuration representations that show the complete stack.[2]

3.2.2. Multiple-Call Collapsing

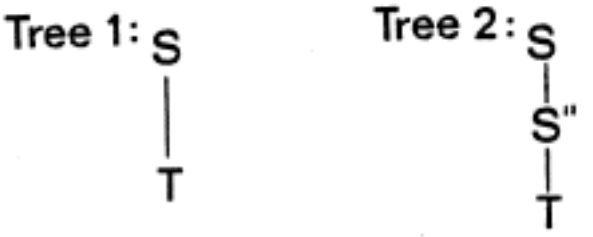
It is convenient to think of this method as simulating, not one, but many non-deterministic parsers at the same time. As these parsers run, they make different decisions at each choice point, and the simulation keeps track of all the different configurations they get into. At any position in the input, the current state of any given parser is contained in some item on the current item list, and the contents of that parser's stack may be computed by following return pointers from that item upwards.

It may happen, however, that two parsers whose stacks differ enter the same state at the same position in the input. For example, consider the following grammar G':

$$\begin{aligned} S &\rightarrow S' \\ S &\rightarrow S'' \\ S'' &\rightarrow S' \\ S' &\rightarrow Aa \\ A &\rightarrow c \\ A &\rightarrow cA \end{aligned}$$

G' derives the same strings as the grammar G given above. However, if G derives a string via derivation tree T, then G' derives it via the following two trees:

[2] Their relationship with Earley items is examined below.



The parallel structure of these trees can be seen clearly in the following simulation of a parse of *cca* under G':

$0(\epsilon)$ $[S \rightarrow \cdot S',\ ]_1$
$[S' \rightarrow \cdot Aa, 1]_2$
$[A \rightarrow \cdot c, 2]_3$
$[A \rightarrow \cdot cA, 2]_4$
$[S \rightarrow \cdot S'',\ ]_5$
$[S'' \rightarrow \cdot S', 5]_6$
$[S' \rightarrow \cdot Aa, 6]_7$;compare with item 2
$[A \rightarrow \cdot c, 7]_8$
$[A \rightarrow \cdot cA, 7]_9$

$1(c)$ $[A \rightarrow c \cdot, 2]_{10}$
$[S' \rightarrow A \cdot a, 1]_{11}$
$[A \rightarrow c \cdot A, 2]_{12}$
$[A \rightarrow \cdot c, 12]_{13}$
$[A \rightarrow \cdot cA, 12]_{14}$
$[A \rightarrow c \cdot, 7]_{15}$;compare items 15–19 with items 10–14
$[S' \rightarrow A \cdot a, 6]_{16}$
$[A \rightarrow c \cdot A, 7]_{17}$
$[A \rightarrow \cdot c, 17]_{18}$
$[A \rightarrow \cdot cA, 17]_{19}$

$2(cc)$ $[A \rightarrow c \cdot, 12]_{20}$
$[A \rightarrow cA \cdot, 2]_{21}$
$[S' \rightarrow A \cdot a, 1]_{22}$
$[A \rightarrow c \cdot A, 12]_{23}$

$[A \rightarrow \cdot c, 23]_{24}$
$[A \rightarrow \cdot cA, 23]_{25}$

$[A \rightarrow c \cdot, 17]_{26}$
$[A \rightarrow cA \cdot, 7]_{27}$
$[S' \rightarrow A \cdot a, 6]_{28}$
$[A \rightarrow c \cdot A, 17]_{29}$
$[A \rightarrow \cdot c, 29]_{30}$
$[A \rightarrow \cdot cA, 29]_{31}$

$3(cca)$ $[S' \rightarrow Aa \cdot, 1]_{32}$
$[S \rightarrow S' \cdot, \ ]_{33}$

$[S' \rightarrow Aa \cdot, 6]_{34}$
$[S'' \rightarrow S' \cdot, 5]_{35}$
$[S \rightarrow S'' \cdot, \ ]_{36}$

The possible configurations obtained upon reading each of the input tokens break cleanly into two groups whose state transitions are identical but whose stack environment is different. Each group can be thought of as containing the configurations of a different parser—one predicting the derivation that starts $S \rightarrow S'$ and the other predicting the derivation that starts $S \rightarrow S'' \rightarrow S'$. The similarity between the two groups is a corollary of the fact that our grammar is context-free. In both cases we are seeing the transitions involved in the leftmost derivation of cca from S'; these transitions must remain the same regardless of the context of S' in the derivation.

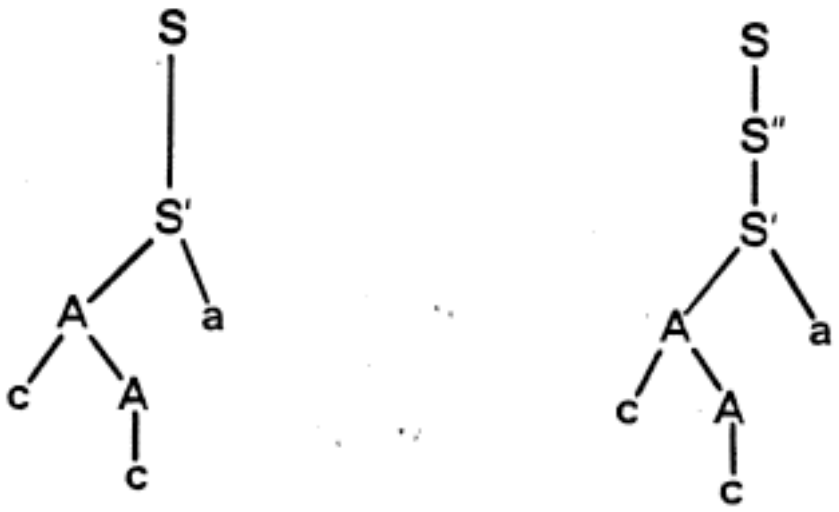
The key observation here is that, when the recognizer for a given rule is called, *the starting position of that recognizer in the input completely determines its behavior.* A particular recognizer may be called from parsers with a variety of stack configurations, but if all the calls occur at the same input position, we need only simulate the state transitions made by that recognizer once: the results can then be used in all the parsers that made the simultaneous calls.

With our representation, this optimization is easily made by turning multiple calls to the same recognizer at the same input position into a single call with multiple returns. This leads to the following parse list for cca under G' (each item now contains a set of return pointers instead of just a single one):

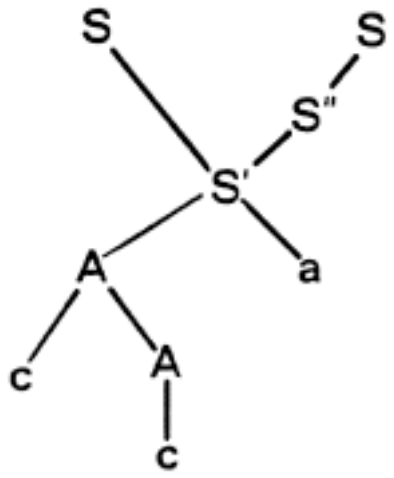
$0(\epsilon)$ $[S \rightarrow \cdot S', \{\}]_1$

	$[S \to \cdot S'', \{\}]_2$	
	$[S'' \to \cdot S', \{2\}]_3$	
	$[S' \to \cdot Aa, \{1,3\}]_4$	;compare with items 2 and 7 above
	$[A \to \cdot c, \{4\}]_5$	
	$[A \to \cdot cA, \{4\}]_6$	
$1(c)$	$[A \to c \cdot, \{4\}]_7$	
	$[S' \to A \cdot a, \{1,3\}]_8$	
	$[A \to c \cdot A, \{4\}]_9$	
	$[A \to \cdot c, \{9\}]_{10}$	
	$[A \to \cdot cA, \{9\}]_{11}$	
$2(cc)$	$[A \to c \cdot, \{9\}]_{12}$	
	$[A \to cA \cdot, \{4\}]_{13}$	
	$[S' \to A \cdot a, \{1,3\}]_{14}$	
	$[A \to c \cdot A, \{9\}]_{15}$	
	$[A \to \cdot c, \{1,5\}]_{16}$	
	$[A \to \cdot cA, \{1,5\}]_{17}$	
$3(cca)$	$[S' \to Aa \cdot, \{1,3\}]_{18}$	
	$[S \to S' \cdot, \{\}]_{19}$	;return to item 1, accept
	$[S'' \to S' \cdot, \{2\}]_{20}$	;return to item 3, ...
	$[S \to S'' \cdot, \{\}]_{21}$	; ...accept

A useful way to conceptualize the optimization performed here is to visualize the parse trees "built" by the pushes and pops of the various parsers being simulated. Before the optimization, the simulator built *both* of the correct derivation trees:



After the optimization, it builds the following hybrid structure:



The latter structure contains the same information about the parse as the two previous trees together.

3.2.3. Left-recursion

An important sub-case in which multiple-call collapsing is applicable is that of left-recursive grammar rules. These rules present well-known difficulties for deterministic recursive-descent parsers, because the parser can not know how many times to invoke the recursive expansion of a non-terminal without looking ahead in the input. For example, consider the following grammar:

$$\begin{aligned} S &\rightarrow Aa \\ A &\rightarrow c \\ A &\rightarrow Ac \end{aligned}$$

This grammar derives exactly the same strings as the right-recursive grammar G given above, but consider the following "parse" of the input string cca (we have not used multiple-call collapsing):

$0(\epsilon)$ $[S \rightarrow \cdot\, Aa,\ ]_1$
$[A \rightarrow \cdot\, c, 1]_2$;expand A from item 1
$[A \rightarrow \cdot\, Ac, 1]_3$;ditto
$[A \rightarrow \cdot\, c, 3]_4$;expand A from item 3 (uh oh)
$[A \rightarrow \cdot\, Ac, 3]_5$
$[A \rightarrow \cdot\, c, 5]_6$;and so on
$[A \rightarrow \cdot\, Ac, 5]_7$
$[A \rightarrow \cdot\, c, 7]_8$;and so on ...
$\vdots$

$1(c)$ $[A \rightarrow c\,\cdot, 1]_{\infty+1}$
$[S \rightarrow A \cdot a,\ ]_{\infty+2}$;return to item 1
$[A \rightarrow c\,\cdot, 3]_{\infty+3}$
$[A \rightarrow A \cdot c, 1]_{\infty+4}$;return to item 3
$[A \rightarrow c\,\cdot, 5]_{\infty+5}$
$[A \rightarrow A \cdot c, 3]_{\infty+6}$;return to item 5 (uh oh)
$[A \rightarrow c\,\cdot, 8]_{\infty+7}$
$[A \rightarrow A \cdot c, 5]_{\infty+8}$;and so on ...
$\vdots$

$2(cc)$ $[A \rightarrow Ac\,\cdot, 1]_{\infty+\infty+1}$
$[S \rightarrow A \cdot a,\ ]_{\infty+\infty+2}$
$[A \rightarrow Ac\,\cdot, 3]_{\infty+\infty+3}$
$[A \rightarrow A \cdot c, 1]_{\infty+\infty+4}$

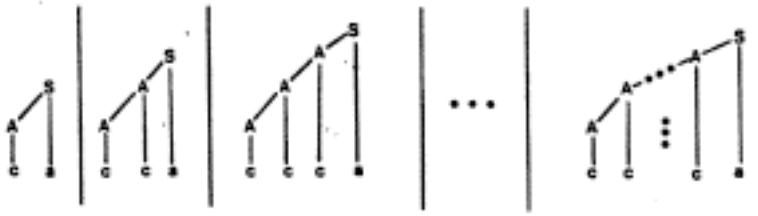
$[A \rightarrow Ac \cdot, 5]_{\infty:\infty:5}$
$[A \rightarrow A \cdot c, 3]_{\infty:\infty:6}$
$\vdots$

$3(cca)$ $[S \rightarrow Aa \cdot, \ ]$

If we perform multiple-call collapsing, however, something very interesting happens:

$0(\epsilon)$	$[S \rightarrow \cdot Aa, \{\}]_1$	
	$[A \rightarrow \cdot c, \{1,3\}]_2$	;expand item from items 1 *and* 3
	$[A \rightarrow \cdot Ac, \{1,3\}]_3$	;note the *self*-recursion here
$1(c)$	$[A \rightarrow c \cdot, \{1,3\}]_4$	
	$[S \rightarrow A \cdot a, \{\}]_5$	;return to item 1 ...
	$[A \rightarrow A \cdot c, \{1,3\}]_6$	; and *again* to item 1!
$2(cc)$	$[A \rightarrow Ac \cdot, \{1,3\}]_7$	
	$[S \rightarrow A \cdot a, \{\}]_8$	
	$[A \rightarrow A \cdot c, \{1,3\}]_9$	
$3(cca)$	$[S \rightarrow Aa \cdot, \{\}]_{10}$	

The subtlety here involves item 3, which serves the same purpose as items 3, 5, 7, 9, ..., in the previous simulation. We are in fact simulating an infinite number of parsers here, one predicting each of the following parse trees:



At any given point past the first c in the input, however, they have all invoked the same recognizer (for $A \rightarrow Aa$) at the same point, so the simulation keeps just one representation for all of them.

3.2.4. Duplicate-Item Merging

Multiple-call collapsing optimizes the case where different parsers invoke the same recognizer at the same point in the input. If we consider only unambiguous grammar, this is the only case in which recognizers invoked by

different parsers are guaranteed to perform identical actions. But consider the following ambiguous grammar fragment:

$$
\begin{aligned}
S &\rightarrow A \ldots \\
A &\rightarrow BC \\
B &\rightarrow b \\
B &\rightarrow bb \\
C &\rightarrow bc \\
C &\rightarrow c
\end{aligned}
$$

This fragment produces the following two derivations of the string fragment bbc:

These derivations are recovered in the following parse:

$0(\epsilon)$ $[A \rightarrow \cdot\, BC, \{\}]_1$
$[B \rightarrow \cdot\, b, \{1\}]_2$
$[B \rightarrow \cdot\, bb, \{1\}]_3$

$1(b)$ $[B \rightarrow b\,\cdot, \{1\}]_4$
$[A \rightarrow B \cdot C, \{\}]_5$
$[C \rightarrow \cdot\, bc, \{5\}]_6$
$[C \rightarrow \cdot\, c, \{5\}]_7$
$[B \rightarrow b \cdot b, \{1\}]_8$

$2(bb)$ $[C \rightarrow b \cdot c, \{5\}]_9$
$[B \rightarrow bb\,\cdot, \{1\}]_{10}$
$[A \rightarrow B \cdot C, \{\}]_{11}$;compare with item 5
$[C \rightarrow \cdot\, bc, \{11\}]_{12}$
$[C \rightarrow \cdot\, c, \{11\}]_{13}$

$3(bbc)$ $[C \rightarrow bc\,\cdot, \{5\}]_{14}$
$[A \rightarrow BC\,\cdot, \{\}]_{15}$
$[C \rightarrow c\,\cdot, \{11\}]_{16}$

$$[A \rightarrow BC \cdot, \{\}]_{17} \quad \text{;compare with item 15}$$

Note that items 15 and 17 are identical. The situation encountered here is quite similar to that in which we invoke multiple-call collapsing, in that recognizers for the same rule invoked simultaneously by different parsers have reached the same state at the same point in the input. In this case that state is not the recognizers' initial state, but the same reasoning shows that both recognizers will perform identical actions until their respective parsers make differing derivation decisions. Thus, as with multiple-call collapsing, we need only keep one item to represent the state of both recognizers.

3.2.5. The String Algorithm

We are now ready to state our string parsing algorithm. The algorithm takes as input a grammar G and a string a, and determines whether G generates s. The output of the algorithm is a sequence of item lists—one for each symbol in a—which represent all the configurations reachable by a non-deterministic string parser for G operating on s. The algorithm does not construct a parse tree for the input, but we show below how it can easily be modified to construct all possible ones.

The algorithm operates by using a list of items I_i to keep track of all the configurations a parser might be in after reading the i-th input symbol. Given lists $I_0, \ldots, I_{i-1}$, the algorithm constructs list I_i by using three operations:[3] a *scanner* operation, a *predictor* operation, and a *completer* operation. We first describe the nature of these operations, and then how the algorithm uses them to construct the lists $I_0, I_1, \ldots, I_n$.

The Scanner

The scanner operation takes as input an item i from list I_{j-1} and the j-th input symbol a_j. Let s be the state part of i and r its set of return pointers. If s has no transition on a_j, then the scanner does nothing. Otherwise, s has a transition on a_j to some state s', and the scanner creates an item i' on list I_j whose state part is s' and whose list of return pointers is r.

We can abbreviate the scanner operation as follows: Let $[A \rightarrow \alpha \cdot t\beta, r]$ be an item from I_{j-1}. If t is the j-th symbol of the input string, then the scanner adds the item $[A \rightarrow \alpha t \cdot \beta, r]$ to I_j.

---

[3] The names of these operations are taken from [Earley 1969].

The Predictor

The predictor operation takes as input an item i from list I_j. If the state part s of i does not have a transition on a non-terminal node, then the predictor operation does nothing. Otherwise, let A be the non-terminal on which s has a transition, and let $s_1, s_2, \ldots, s_m$ be the initial states of all the recognizers for rules which derive A. For each s_i, the predictor operation checks to see if there is an item with state part s_i on list I_j. If so, the predictor adds i to the return set of that item. If not, the predictor creates an item with state part s_i and return set $\{i\}$ and adds it to I_j.

We can abbreviate the predictor operation as follows: Let $[A \rightarrow \alpha \cdot B\gamma, r]_i$ be an item on I_j. For all rules $B \rightarrow \beta$ in G, the predictor operation searches I_j for an item of the form $[B \rightarrow \cdot\beta, r]$. If it finds one, it adds i to r. Otherwise, it adds an item $[B \rightarrow \cdot\beta, \{i\}]$ to I_j.

The Completer

The completer operation takes as input an item i on list I_j. If the state part of i is not the accepting state of a recognizer for some rule of G, the completer operation does nothing. Otherwise, let A be the non-terminal derived by the accepted rule, and let $i_1, \ldots, i_m$ be the members of the return set of i. The state part of each i_i must have a transition on A; let $s_1, \ldots, s_m$ be the states led to by those transitions. For each i_i, the completer looks for an item on I_j whose state part is s_i and whose return set is that of i_i. If it finds one, it does nothing, otherwise it adds such an item to I_j.

The completer operation may be abbreviated as follows: Let $[A \rightarrow \gamma \cdot, r_1]$ be an item in I_j. For each item $[B \rightarrow \alpha \cdot A\beta, r_2]_i$ such that $i \in r_1$, add $[B \rightarrow \alpha A \cdot \beta, r_2]$ to I_j if it is not already there.

The Algorithm

First, we construct I_0 as follows:

1. Let $s_1, \ldots, s_m$ be the initial states of recognizers for the rules in G which derive S. For each s_i, add an item to I_0 whose state is s_i and whose return set is empty.
2. Complete I_0 by running the predictor on every item in it. If new items are added to it, run the predictor on them, and repeat this until no new items are added.

Next, we successively construct $I_1, \ldots, I_n$. Given $I_0, \ldots, I_{j-1}$, we construct I_j as follows:

3. Run the scanner over every item in I_{j-1}.

4. Run the completer over every item in I_j. If this adds new items to I_j, run the completer over them, and repeat this until no new items are added.

5. Run the predictor over every item in I_j. If this adds new items to I_j, run the predictor over them, and repeat this until no new items are added.

A little thought will convince the reader that this is indeed the algorithm used to produce the lists shown above. A string is accepted by this algorithm if I_n contains an item whose return set is empty and whose state part is the accepting state of a recognizer for a rule deriving S.

3.2.6. Why is this Earley's Algorithm

The algorithm described above does not appear, *prima facia*, to be Earley's algorithm. The apparent difference is due to a couple of factors, both of which we examine here.

Abbreviation of Return Pointers

Our algorithm uses items of the form $[A \rightarrow \alpha \cdot \beta, r]$, where r is a set of return pointers. Earley's items have the form $[A \rightarrow \alpha \cdot \beta, i]$, where I_i is the number of input symbols read when the A recognizer was first invoked. (Of course, at that time, the recognizer was represented by an item of the form $[A \rightarrow \cdot \alpha\beta, i]$.)

These representations seem unrelated; however, some thought reveals that we can encode our representation in Earley's form. An item of the form $[A \rightarrow \cdot \alpha, r]$, when added to list I_i, represents a call on one of A's recognizers at point i in the input. Thus, the callers of such an item—the members of r—must be all the items for recognizers which expect to see an A at point i in the input. But these items are exactly all those on I_i which have an A to the right of their dot. Thus, if an item of the form $[A \rightarrow \cdot \alpha, r]$ appears on I_i, r must consist of exactly those items on I_i that have an A to the right of their dot, so we can encode r with the integer i.

Handling of ϵ-rules

Earley's algorithm handles grammars with productions of the form $A \rightarrow \epsilon$.[4] This involves running the completer on I_0, and alternating the repeated application of steps 4 and 5 (instead of applying one repeatedly and then the other).

If these steps were added to our algorithm, and if the representation were changed as mentioned above, our algorithm description would agree exactly with the description given by Earley in [Aho and Ullman 1972].

3.2.7. Using the Algorithm to produce Parse Trees

The algorithm we have presented here is actually an acceptor, not a parser. That is, while its output indicates immediately whether or not the input string is in the language of the input grammar, it does not provide a parse tree.

Algorithms are available from a variety of sources (*e.g.*, [Aho and Ullman 1972]) which produce a parse tree from the parse lists output by our algorithm. In addition, consider the following definitions of the scanner and completer operations:

The Completer

The completer operation takes as input an item i on list I_j. If the state part of i is not the accepting state of a recognizer for some rule of G, the completer operation does nothing. Otherwise, let A be the non-terminal derived by the accepted rule, and let $i_1, \ldots, i_m$ be the members of the return set of i. The state part of each i_i must have a transition on A; let $s_1, \ldots, s_m$ be the states led to by those transitions. For each i_i, the predictor looks for an item on I_j whose state part is s_i and whose return set is that of i_i. If it finds one, it adds to it a pointer to i and a pointer to I_i, otherwise it adds such an item (including these pointers) to I_j.

The Scanner

The scanner operation takes as input an item i from list I_{j-1} and the j-th

[4] Our algorithm need not handle these productions, since we are interested only in generalizing it to graph grammars (in which such productions can not occur).

input symbol a_j. Let s be the state part of i and r its set of return pointers. If s has no transition on t, then the scanner does nothing. Otherwise, s has a transition on t to some state s', and the scanner creates an item i' on list I_j whose state part is s', whose list of return pointers is r, and which contains the same completer-added pointers as i (if any).

If the algorithm uses these definitions, each item of the form $[A \rightarrow \alpha \cdot, r]$ in the constructed lists will be the root of a pointer structure giving all the derivation trees for that instance of A in the input. In particular, if a sentence is accepted by the algorithm, the items of the form $[S \rightarrow \alpha \cdot, \{\}]$ on I_n will be the roots of all the derivation trees for that sentence.

Chapter 4.

The Algorithm

In this chapter we present our flow graph parsing algorithm. The inputs to the algorithm are a flow grammar and flow graph; its output is a sequence of lists similar to the item lists produced by Earley's algorithm.

As in the last chapter, we will produce the algorithm by developing a non-deterministic parser and then simulating its behavior deterministically. Both the parser and the simulation technique generalize those we used for strings: the resulting algorithm is a generalization in that, when it is run on a string graph, it performs a superset of the actions performed by our string algorithm.

4.1. Non-Deterministic Graph Parsers

The method we used to construct a parser for a string grammar consisted essentially of two steps:

1. Construct recognizers for the right-hand sides of each of the grammar's productions.
2. Construct a stack-based machine out of these recognizers by replacing their non-terminal recognition steps with "subroutine calls" on other recognizers.

We will apply this same method to flow grammars in order to construct flow graph parsers. The nature of this construction is determined by our generalizations of (i) the mechanism used to read the parser's input, (ii) the recognizers used for the right-hand sides of grammar rules, and (iii) the linkage mechanism used to interconnect recognizers. Each of these general-

izations preserves intuitions that arise in the string case, but phrases these intuitions so as to make them applicable to graphs as well as strings.

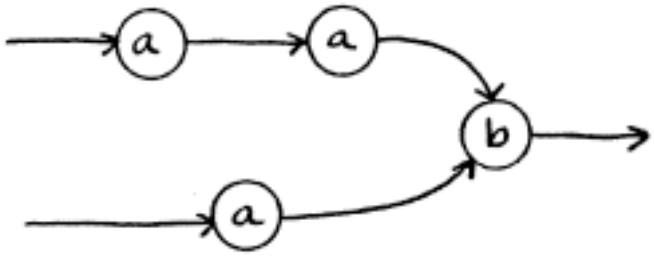
4.1.1. Reading a Flow Graph

Our string parser constructed a parse for its input while reading it once from left to right, and so will our graph parser. 'Once' means that the parser will look at each node in the input only one time. 'From left to right' means that the parser will consider nodes in the partial order imposed by the input graph: that is, a node in the input will be looked at by the parser only when it has already looked at all that node's predecessors.

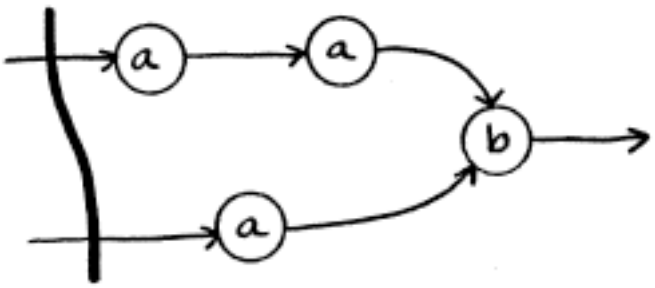
As mentioned in the last chapter, it is natural to think of our string parser as an automaton using a read head to examine its input. This head moves from left to right over the input, passing the symbols read on to the state-transition functions of the parser's active recognizers.

Our graph parsers will examine their input graphs as if they, too, had read heads. These heads should be thought of as "multi-track" heads which can be positioned over more than one node at a time. They start at the left edge of the input, read nodes one at a time from left to right, and pass information about these nodes on to the state transition functions of the parser's recognizers.

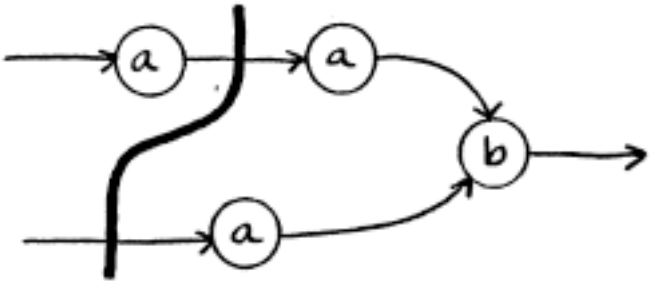
For example, consider the following graph:



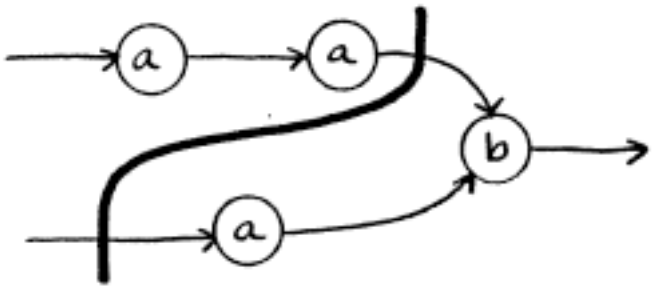
A parser reading this graph would start off with its read head positioned to the left of the graph's two minimal nodes, like this:



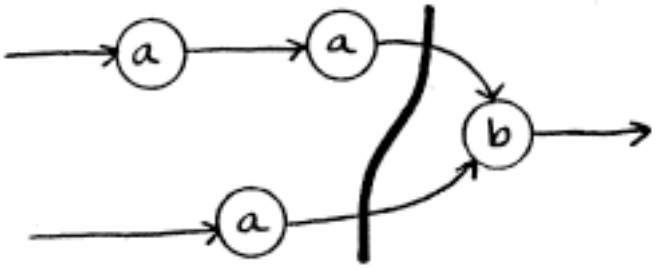
It would then select one of the two minimal nodes to be read next—we don't care which. Let us say it chooses the upper one; this would leave its read head in the following position:



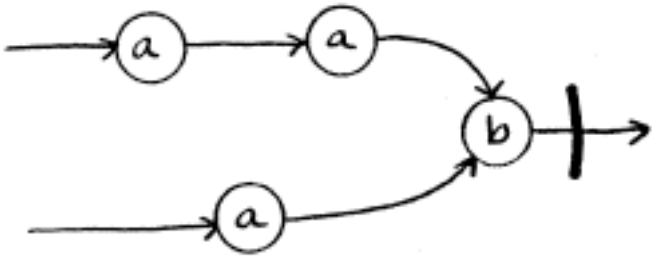
Here the parser must again choose which node to read; let us say it again chooses the upper one. The read head would move over this node to give the following position:



At this point, there is only one node — the lower one — available for reading. Intuitively, the read head is not "just to the left" of the graph's maximal node: there is still a preceding node which must be read first. Thus, the next read head position is as follows:



Finally, after the last node is read, we have:



and the read head stops.

We have indicated the position of the read head at each stage by denoting the unique set of edges (possibly leading or trailing edges) all of which follow all the nodes already read and precede all the nodes yet to be read. We call these edge sets *head positions*, and we precisely characterize the order in which graph parsers examine the nodes of their input as follows:

1. Each parser is considered to have a read head. The initial head position of the read head in the input consists of all the input's leading edges.
2. The parser can examine any node all of whose incoming edges are in the current head position. (Such a node is said to be in the *right fringe* of the head position.)

3. When a parser whose read head is in position p examines a node n, its read head moves to a new position p' calculated by taking p and replacing n's incoming edges by its outgoing ones. (We call p' the *n-successor* of p. The node n, all of whose outgoing edges are now in p', is said to be in the *left fringe* of p'.)

4. The parser examines nodes one at a time until it reaches a head position with no nodes in its right fringe.

The reader can verify that the example given above meets the above conditions. Some thought will also show that (i) a node is never read until all its predecessors have been read, and (ii) this method, when applied to any flow graph, eventually reads all the nodes in that graph.

It is worth noting that this method, while phrased so as to apply to all flow graphs, describes exactly the motion of our string parser's read head through its input "string graph." The string case simply makes no use of the non-determinism inherent in step (2).

Each time a graph parser examines a node, it passes three pieces of information to the state transition functions of its active recognizers: the type of the node read, its left-linkage information (a set of port-edge pairs), and its right-linkage information (another set). As with our read-head motion rules, it is worth noting that this list describes in a general manner the exact information read by the head of a string parser. In the string case, however, the left-linkage and right-linkage information are both trivial: it is always the case that the only edge in the old head position went into the node's only input port, and the only edge in the new head position came out of the node's only output port.

4.1.2. Flow Graph Recognizers

The right-hand sides of flow-grammar rules are flow graphs; thus, the recognizers from which we build our parser will be flow graph recognizers. These recognizers will receive type and linkage information about the input from the parser, and compare this information with that found in their *target graph*—the right-hand side they are recognizing. Their structure and function will be generalizations of those of their string counterparts; that is, they will be state machines which make transitions based on the input read.

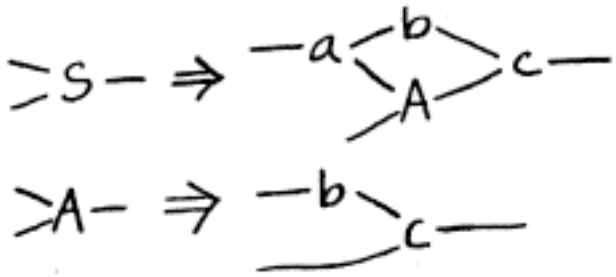


Figure 4.1. A grammar.

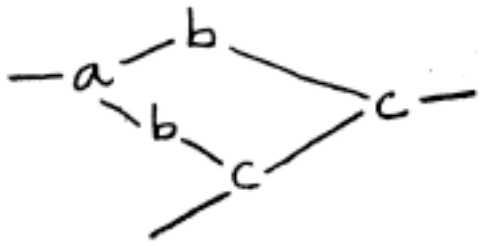


Figure 4.2. A graph generated by the grammar of figure 4.1.

4.1.3. States

A state in a flow-graph recognizer consists of pairs matching edges in the recognizer's target graph with edges in the parser's current head position. For example, consider the grammar of figure 4.1, and the graph generated by that grammar shown in figure 4.2. At some point in the parse of this graph, the recognizer for the right-hand side of the A-rule might reach the following state:

We have indicated the parser's head position as in the last section; the labels on the input-graph and target-graph edges indicate the pairing which is the state. The target-graph edge paired with a given input edge is called the *target image* of that edge; the latter is the *input image* of the former.

It is convenient (albeit redundant) to think of a state as having two parts: (i) a set of edges from the target graph, and (ii) a 1-1 correspondence between that set and some of the edges in the parser's head position. In this view, it becomes clearer that the states of our string recognizers had the same composition: their edge set was the edge denoted by their Earley dot, and their correspondence was always the trivial one sending that edge into the single edge in the parser's current head position. The triviality of this correspondence allowed us to ignore it and "pretend" that the states of our string recognizers were completely determined by their dot position. We do not have this luxury in the graph case: for example, examine the two states shown in figure 4.3, and consider which of these states should begin a transition sequence leading to an accepting state.

4.1.4. State Transition Functions

The state transition functions of our graph recognizers take as inputs a recognizer state and the type and linkage information of an input node; they produce a new recognizer state as output. Recognizers operate in the expected manner: they apply their state transition functions to their current state and the information returned by the parser's read head, and then make a transition to the new state returned by the transition function (if any).

The state transition function of a graph recognizer is best thought of as an algorithm that proceeds in two steps: it first determines whether a transition exists from the given state on the given input; if so, it then determines the state that the transition leads to. In other words, the algorithm first determines the acceptability of the input, and then it determines the correct

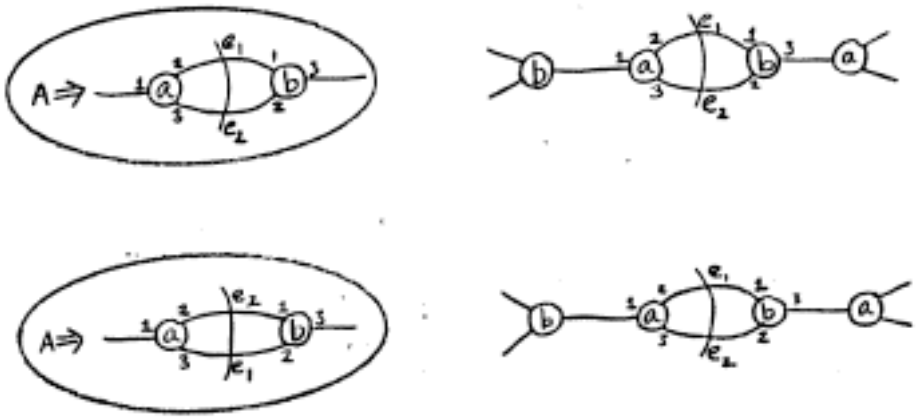


Figure 4.3. Two states which differ only in their correspondence part.

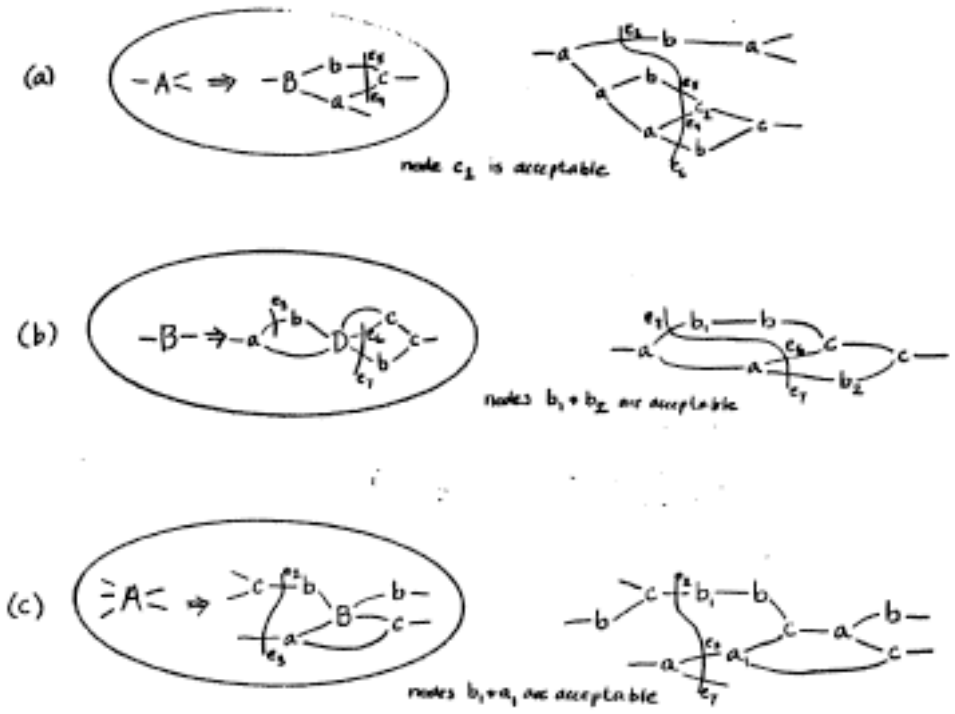


Figure 4.4. Some acceptable ⟨state, input⟩ pairs.

next state for its recognizer.

Acceptability is determined by comparing the type and left-linkage information of the input node read with that of the target graph node which corresponds to it. More precisely, let s be the the current state, let n be the input node read, let L be the set of input edges of n, and let L' be the set of target images of L under s. If L' consists of all the input edges of some target graph node n', if the type of n' is the same as the type of n, and if the port adjoined on n' by each edge in L' is the same as the port adjoined by its input image (in L), then n is said to be *acceptable* and n' is said to be its *target image*. Figure 4.4 shows examples of acceptable input situations; figure 4.5 shows some unacceptable ones.

Once the acceptability of an input node has been determined, the new state to move to is computed by matching its right-linkage information against that of its target image. More precisely, let s, n, n', L, and L'

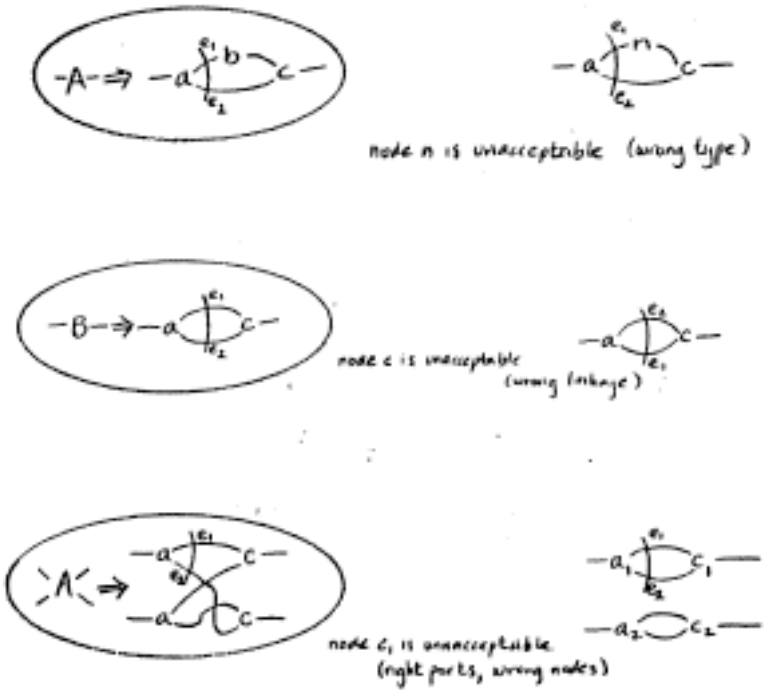


Figure 4.5. Some unacceptable (state, input) pairs.

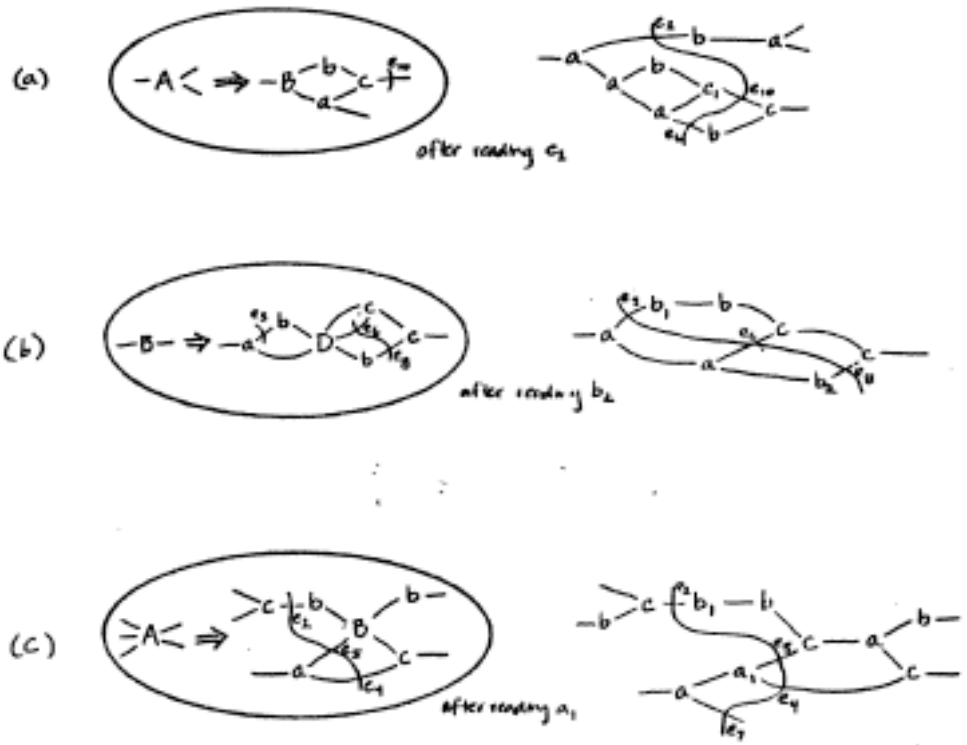


Figure 4.6. New ⟨state, input⟩ pairs computed by the state-transition algorithm from the pairs of figure 4.4.

be as above, let R be the output edges of n, and let R' be the output edges of n'. The new state s' is computed by (i) deleting from s all pairs involving edges in E, and (ii) adding a new pair for each edge in R. In step (ii), the pair added for an edge e which leaves n from a port p pairs it with the target graph edge e' in R' which leaves n' from p. (Since n and n' have the same type and thus the same port sets, this operation is well-defined.) Figure 4.6 shows the ⟨new-state, new-input⟩ pairs computed from the pairs of figure 4.4 by this procedure. Notice that state pairs not involving input edges to the input node read are unaffected.

As the reader may have noticed, this procedure agrees with that used to determine the state transition functions of our string recognizer. In fact, if we take into account both the edge mapping implicit in our string recognizer states, and the linkage information implicitly read by the string parser read

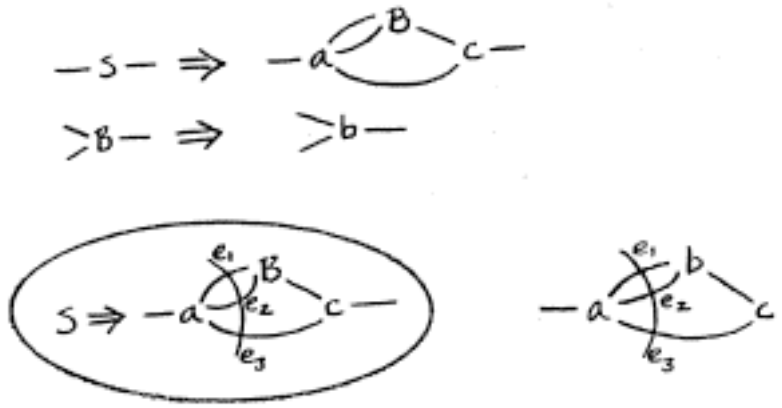


Figure 4.7. A grammar and a state in which a sub-recognizer should be invoked.

Figure 4.8. The initial state of the sub-recognizer invoked from the state of figure 4.7.

head, this *is* the procedure used to compute the state transition functions in our string parser.

4.1.5. Linkage Mechanism

Whenever a recognizer moves into a state whose edge set contains inputs to a non-terminal, the parser will invoke a sub-recognizer for that non-terminal. For example, consider the grammar and state shown in figure 4.7. Two of the target-graph edges in the state of the S-recognizer are inputs to the non-terminal B, so the parser calls a recognizer for B, giving it the initial state shown in figure 4.8.

The initial state of the B recognizer has followed by "transitivity" from the port-correspondence information in the grammar rule for B. In general, suppose recognizer state s contains target edges e'_j—target images of edges e_j—which are inputs to a non-terminal node n'. The parser deletes any edge pairs from s which involve the e'_j, chooses a production P which derives the

Figure 4.9. An accepting state for the recognizer invoked in figure 4.8. This rule should be reduced.

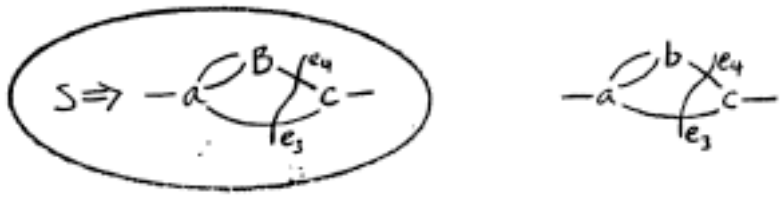


Figure 4.10. The result of the reduction invoked in figure 4.9.

type of n', and invokes a recognizer for the right-hand side of P. The initial state s' of this recognizer will contain one pair for each edge e'_j as follows: Suppose e'_j enters n' at port p'_j, and suppose port p'_j is mapped by P to port p_j on port n (in P's right-hand side). Then s' will pair the leading (target) edge entering n with e_j (the input image of e'_j). The reader is encouraged to verify that this procedure produces the state of figure 4.8.

The operation dual to invocation of a sub-recognizer is the return of that sub-recognizer. In the example given above, the state of the B-recognizer after the parser reads node n will be that given in figure 4.9. The edge set of this state contains a trailing edge, so the parser will reduce the recognized rule and move the calling S-recognizer into the state shown in figure 4.10. In general, whenever a state contains a target-graph trailing edge, the parser will perform a reduction by adding edge pairs to the caller's state in a procedure which reverses that used at invocation time.

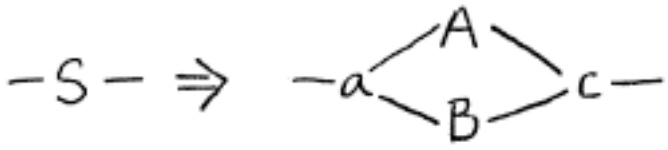
The reader must by now be expecting the following claim: this linkage mechanism is a general phrasing of the exact mechanism used by the string parser. It simply make explicit the manipulations of the target-graph/input-graph edge correspondence that were left implicit in the string case.

4.1.6. Flow Graph Parsers

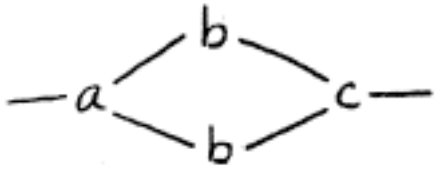
We have now introduced the three basic components of our parser construction technique: a mechanism for reading the input, the recognizers for individual rules, and the linkage mechanism used to interconnect recognizers for different rules. Rather than state a complete construction mechanism (as we did in the previous chapter), we will instead consider two examples of grammar/graph pairs and the behavior of the parser constructed for them. These examples will expose some details of the construction and behavior of the resulting parsers that have not been considered thus far; in addition, they will introduce a representation that forms the basis for that used by our simulation algorithm.

A Simple Example

Let us start by considering the behavior of a parser constructed for the following simple grammar:



when run on the following graph:

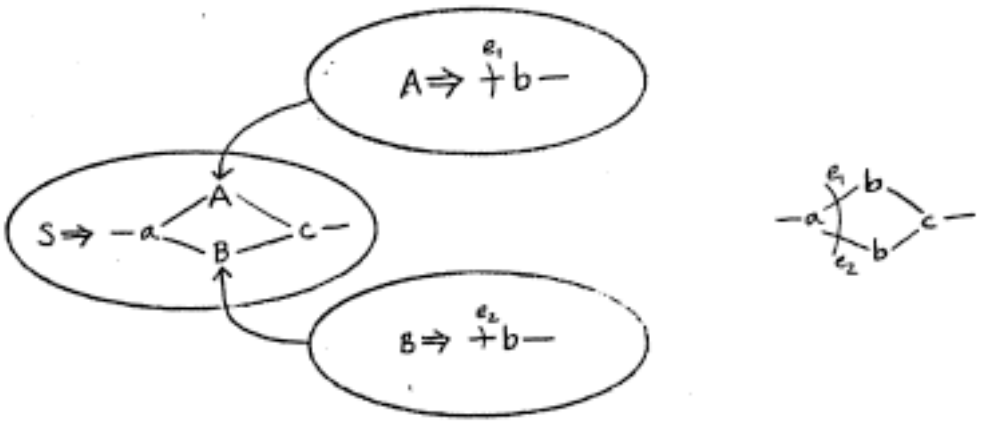


This parser starts off by calling the recognizer for the right-hand side of the rule deriving S. The parser stack is empty, and its read head is over the leading edge of the input. Since there is only one such edge, the parser has no choice in determining the S-recognizer's initial state: that is, there is only one possible correspondence between the leading edges of the input and those of the S-recognizer's target graph. (We consider below how to make this choice in general.)

The initial configuration of the parser is as follows:

At this point, only one node in the input is readable. As the parser's read head reads and moves over it, its type and linkage information is used to make a state transition in the S-recognizer. Of course, if the state-transition algorithm determined the input to be unacceptable, the parser would stop and reject the input. In this case, however, the parser moves into the following configuration:

This state contains target edges which are inputs to the non-terminals A and B. The parser thus invokes sub-recognizers for these nodes, moving into the following configuration:

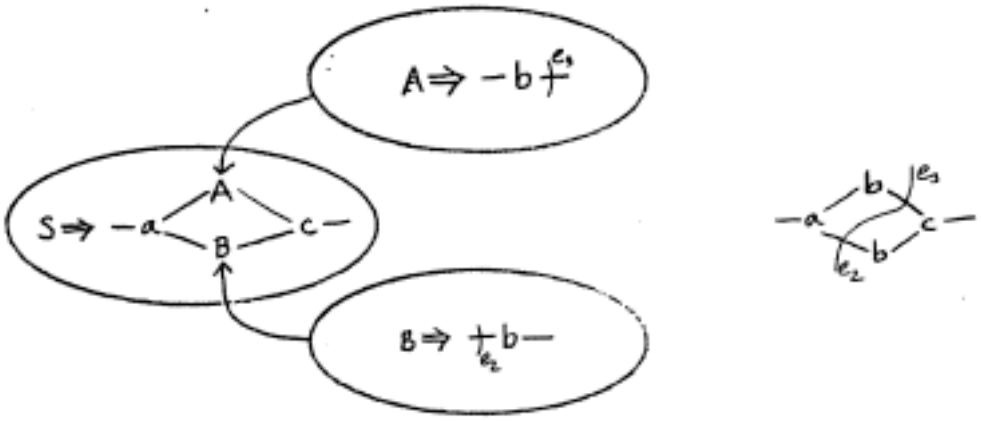


The following points are worth noting:

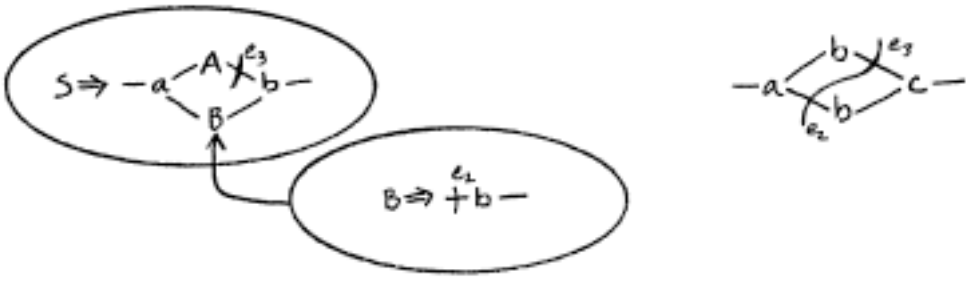
- We are no longer using a simple stack to keep track of sub-recognizer calls. Because multiple calls may be made from a single state, we use a "tree-shaped stack" that keeps track, for each call made, of both the calling state and the particular node being recognized in the caller's target graph.

- This behavior appears different from that of the string recognizer, which left an "Earley dot" in front of the node being derived. In fact, this dot served to identify the node being derived—a function now handled by information kept on the stack--not as a state marker.

The parser is now ready to read another node. Let us say it reads a; this leaves it in the following configuration:

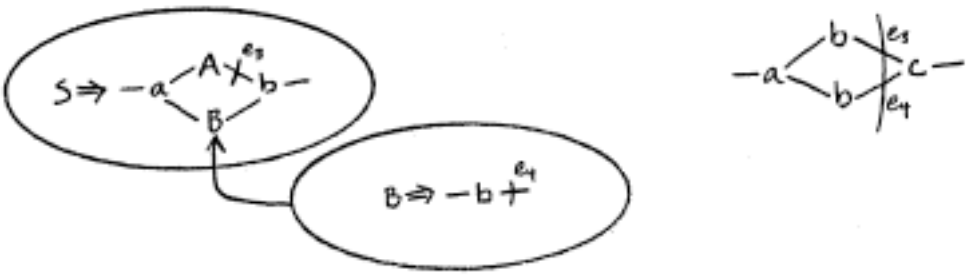


The recognizer for A has now moved into an accepting state, so the parser reduces the production involved by moving into this configuration:



Note that the S-recognizer has changed state while its call to the B-recognizer is outstanding. This could never happen in the string parser. To emphasize this mutability of the state information stored in a graph parser's stack, we think of the stored information as *state objects* rather than states.

Next, the b-node is read, and the B-recognizer changes state accordingly:



The B-recognizer has now moved into an accepting state, so the parser reduces its rule and moves into the following configuration:

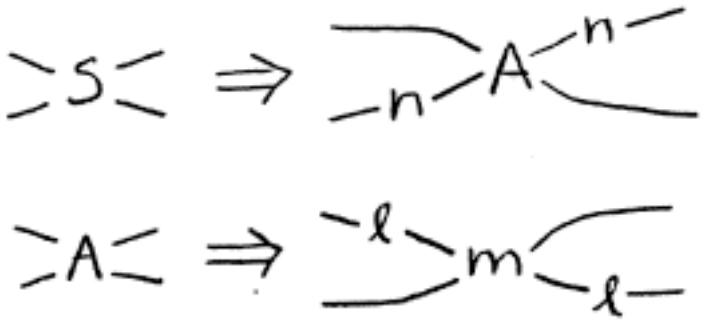
Notice that the S-recognizer state-pairs derived from the reduction procedure are *added* to those of its prior state. This additivity, together with the tree-shaped stack, allows multiple simultaneous calls to sub-recognizers.

Finally, the parser read the m-node and moves into the following configuration:

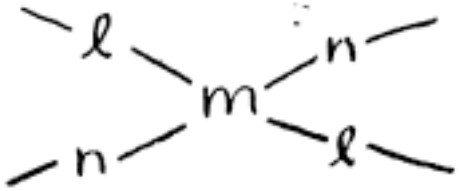
The parser's input has been completely read, and its trailing edges are in correspondence with all the trailing edges of the S-recognizer's target graph. The parser accepts.

A Complex Example

The following example demonstrates the behavior of a parser whose grammar and read-head motion combine to produce "staggered invocations and reductions." Consider the following grammar:



which derives the following graph:

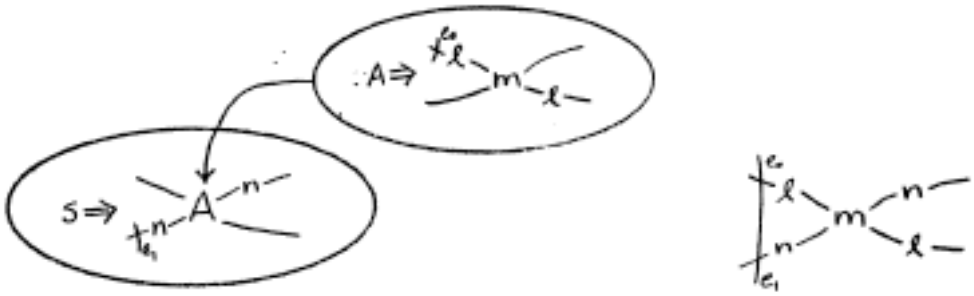


Unlike the grammar considered in the last example, the start symbol for this grammar has two inputs, so the parser constructed from it must make some determination as to which of the input graph's inputs correspond to which of the start symbol's inputs. In general, there is no way (short of trying each possibility) that a parser can determine which choice of correspondences, if any, allows a parse. Thus, in our description of this parser (and in our simulation algorithm), we will assume that the input itself contains a specification of one such correspondence, and the constructed parser will use that one.[1]

[1]Since the simulation algorithm takes both grammar and graph as input, the terms for such a specification are readily at hand.

Let us assume, then, that the parser for the above grammar starts in the following configuration on the above graph:

The S-recognizer's state contains an input edge of the non-terminal A, so the parser activates a recognizer for A and moves into the following configuration:



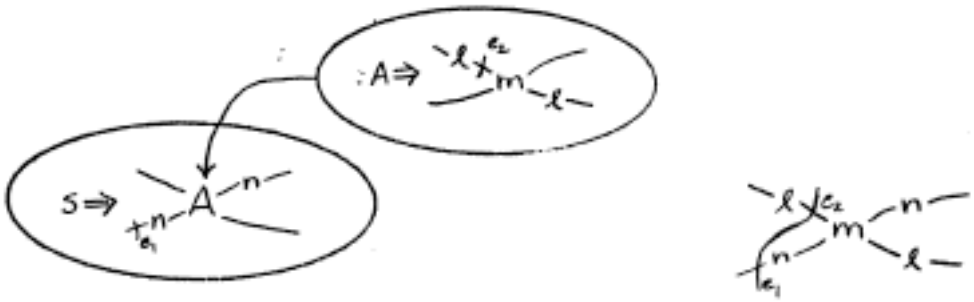
The following points are worth noting:

- The parser has started the A-recognizer before it can determine an input-edge correspondence for all of A's inputs. When node n_1 is read, and the parser determines a correspondence for A's other input, the new input will be added to the recognizer's (then-current) state. This process is called *staggered invocation*.
- Only those pairs involving input edges of A have been deleted from S's state. This "subtractivity" is dual to the additivity of the reduction process.
- Configurations which involve partial states, such as this one, will always result from situations in which the head image of a recognizer's state contains some but not all of a non-terminal's input edges. In these situations, the parser will invoke a sub-recognizer for the non-terminal involved even

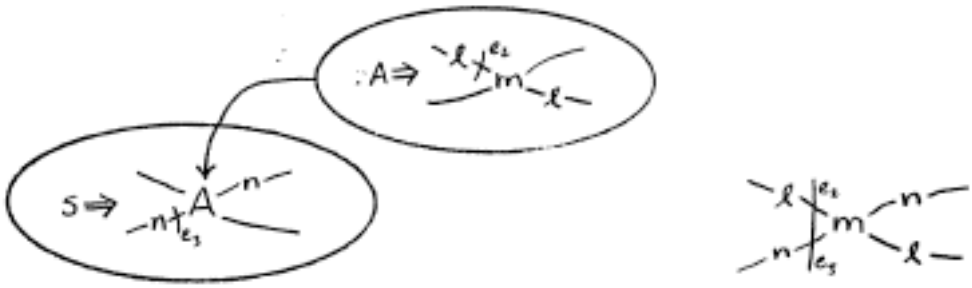
if the input edges corresponding to the non-terminal's inputs are not inputs to a node in the right fringe of the parser's read head position. For example, in this configuration:

a parser would invoke a recognizer for B even though the node b is not yet eligible for reading.

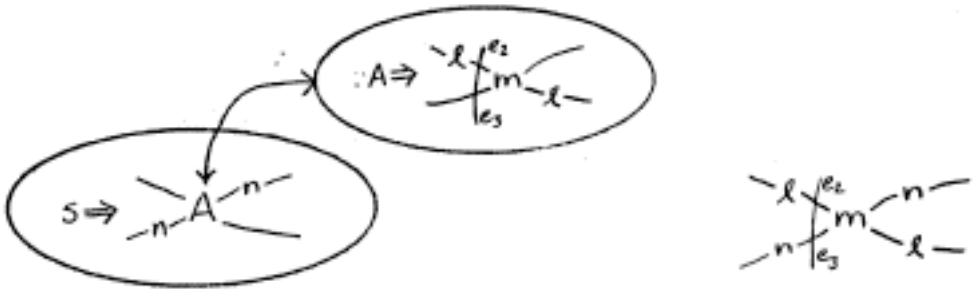
The parser is now ready to read a node; let us say it reads the l node. It would move into the following configuration:



in which only the n node is readable, giving:

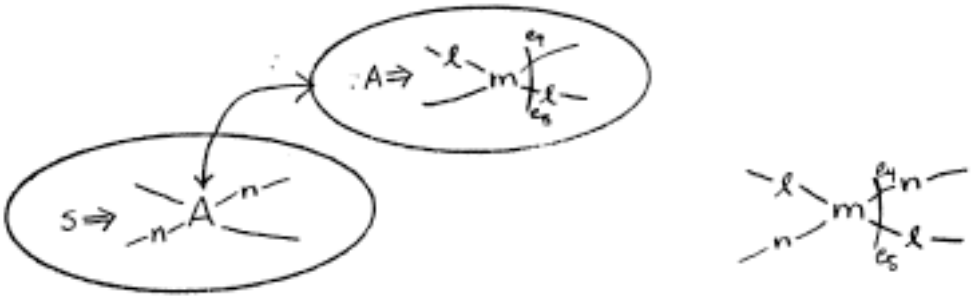


The other input to the previously-invoked A-recognizer has arrived, so the parser adds it to that recognizer's state:

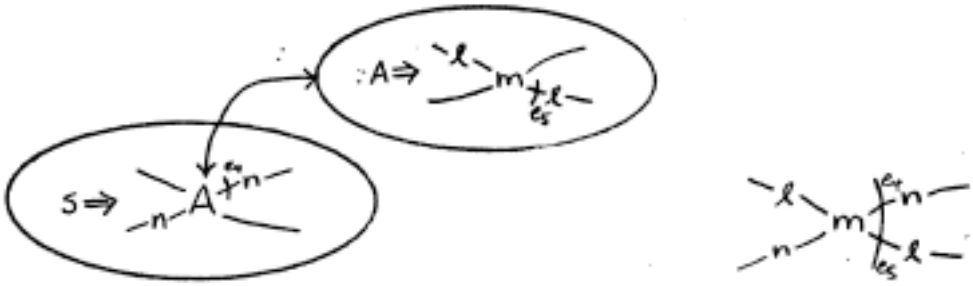


A careful reader may have noticed that we have drawn a two-way arrow between the A-recognizer and the node it was called for. This is to remind us that the parser, in order to do this staggered invocation, must keep track not only of which node a recognizer was called for, but also any recognizer that has already been called for a given node. That is, in addition to keeping "return pointers" with active recognizers, the parser must keep "call pointers" with non-terminal nodes that have sub-recognizers active for them.

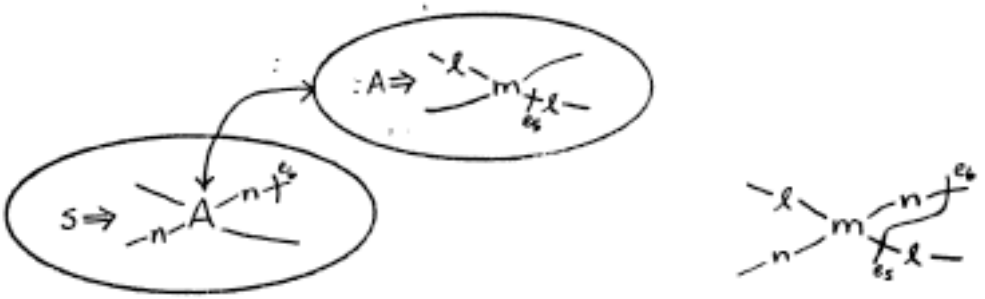
The parser now read the m node, leading to the following configuration:



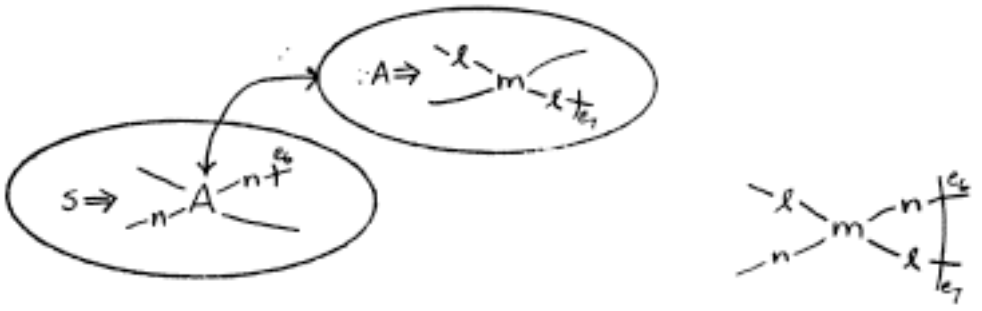
This situation is the converse of one encountered earlier: instead of one (but not all) of the A-recognizer's inputs having been reached, one (but not all) of its outputs have. The parser performs a *staggered reduction* similar to the staggered invocation it performed earlier, and reaches the following configuration:



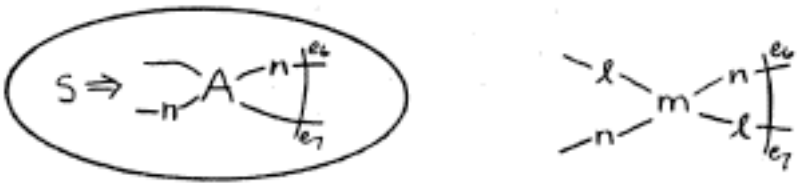
The parser is now ready to read either the n- or the l-node. Let us say it chooses n, this leads to the following configuration:



Finally, the parser reads the l-node and moves into this configuration:



This configuration allows the completion of the staggered reduction started earlier, so the parser terminates the A recognizer and adds to the state of the S-recognizer:



This is an accepting configuration.

4.2. The Parsing Algorithm

We are now ready to present our flow graph parsing algorithm. The algorithm simulates the behavior of a non-deterministic graph parser, exploring simultaneously all the parser's reachable configurations. As with the string algorithm of the last chapter, it will be useful to think of the algorithm as simultaneously simulating a large number of graph parsers, each of which eventually makes different guesses as to the derivation tree of the input.

4.2.1. Preliminaries

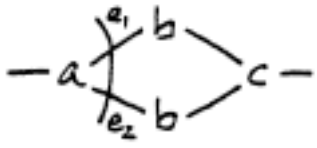
We introduce here an item-based notation for the configurations of a graph parser. We will simulate a given graph parser by constructing item lists, similar to those of the last chapter, which show the parser's configuration at each step of the input.

The basic unit of the notation is a representation of a state object called a *state item* (or just *item*). Items are composed of three parts: a state of a graph recognizer, a list of pending calls to other items, and a list of items to return to. In addition, items will sometimes be annotated as *dead*, and they will sometimes be marked with an *R-flag*. (We say more about these annotations below.) As before, we represent the pointers in call lists and return sets with integers, and we subscript items with integers, yielding a representation like this:

$$[\langle \text{state} \rangle, \langle \text{call list} \rangle, \langle \text{return pointer} \rangle]_{\text{id}}$$

Using this representation, we can represent the configuration of a graph parser by showing its read head position in the input and items for the

state objects of all its recognizers. The correspondence part of each state representation will be indicated by labeling the target and input edges involved. For example, in the first sample run shown above, the configuration obtained just after the invocation of the A- and B-recognizers can be given as follows:



$$[S \Rightarrow -a\overset{A}{\underset{B}{\langle}}c-\ ,\ ((A\ 2)(B\ 3))\ ,\ \phi]_1$$

$$[A \Rightarrow \overset{e_1}{\dagger} b-\ ,\ nil\ ,\ \{1\}]_2$$

$$[B \Rightarrow \overset{e_2}{\dagger} b-\ ,\ nil\ ,\ \{1\}]_3$$

(The subscripts used here are, of course, arbitrary.)

We say that the parser's *active recognizers* (or *active items*) are those whose states are non-empty. In the example above, only the A and B recognizers are active; the S recognizer is said to be *suspended*.

When we wish to show the entire run of a graph parser on a given input, we can compress the space used by showing, after each read operation, only those items are active or which made a state transition. For example, the run of a graph parser on the grammar and input graph shown above is shown in figure 4.11.

While this depiction of a parse run looks a lot like the parse lists of Earley's algorithm, there are some important differences. First, not all active items in a given list change state when the next node is read. Second, more than one active item in a given list can represent recognizers invoked by a

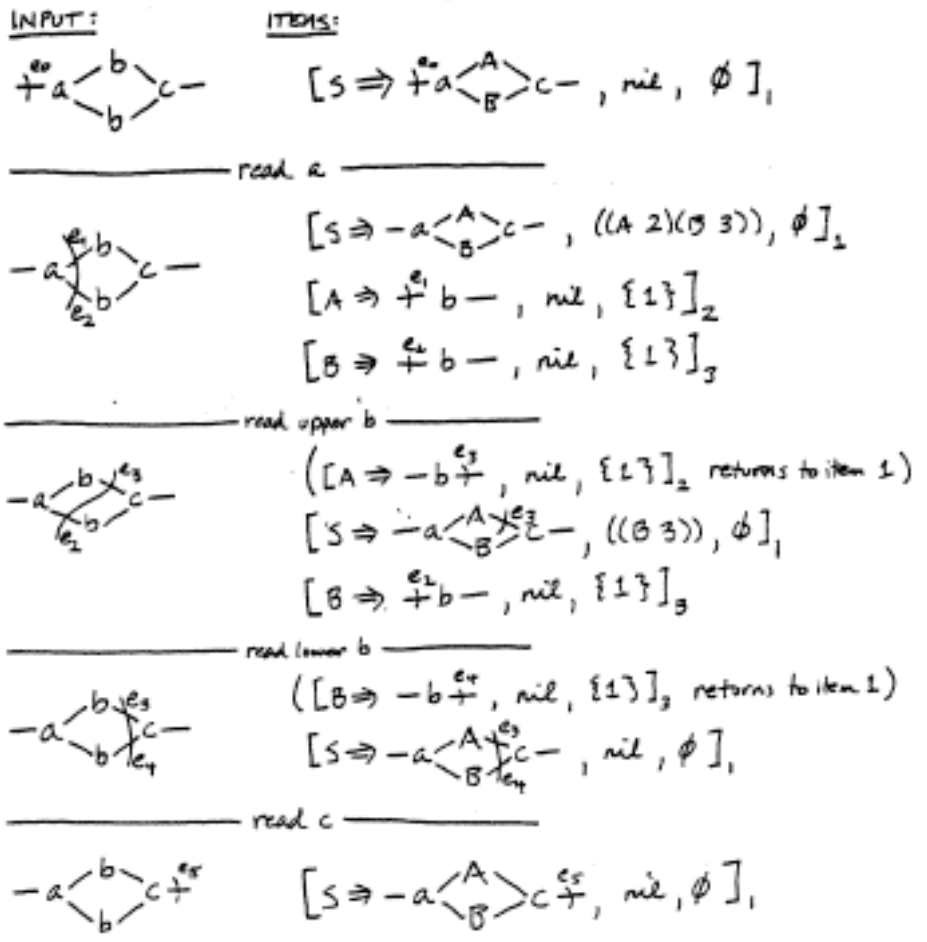


Figure 4.11. Run of graph parser done in first example.

single parser. In Earley's algorithm, each active item represented the *only* recognizer currently active for at least one parser.

4.2.2. Optimizations

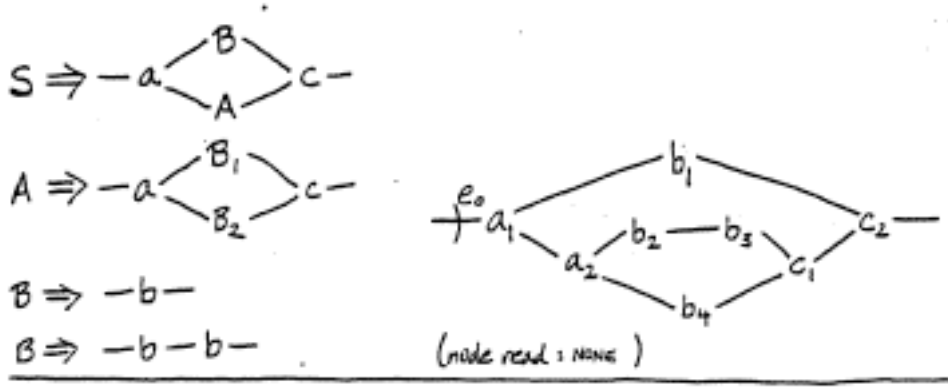
As we did with Earley's algorithm, we will coalesce the items representing recognizers for the same rule which were started at the same input position and are in the same state; one item will be used to represent the state object of all such recognizers. In order to accomplish this, we will be using the two optimizations—multiple-call collapsing and duplicate-item merging—that were introduced in the last chapter.

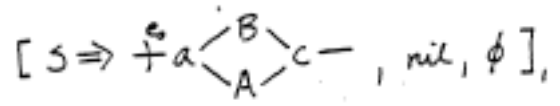
Because the linkage mechanisms of graph parsers are more complex than those of string parsers, we must be more careful in applying optimizations. In particular, the presence of staggered invocations and reductions, and the fact that calling items can change state while their callees are still active, lead to cases that require a very good understanding of the intent of the optimizations. Thus, rather than proceed directly with the statement of our parsing algorithm, we will first consider some examples of its behavior.

4.2.3. Examples

In this section we run the parsing algorithm on five different grammar/graph pairs. Each sample run exposes a different facet of the algorithm's behavior; between them these examples cover all the cases that the algorithm handles specially. Having once understood all five of these examples, readers should have little difficulty understanding the complete statement of the algorithm which follows them.

For each example, we show all of the item lists constructed by the algorithm. Each such list is followed by a some explanation of how it was constructed.



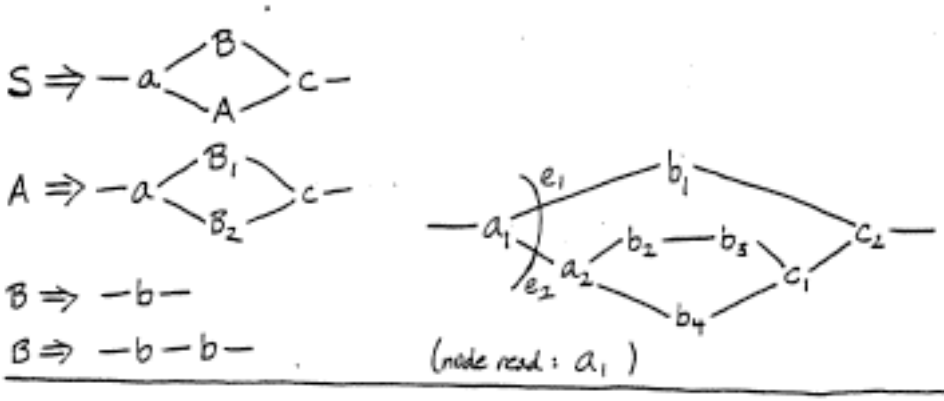


Example 1, list 0.

Example 1.

This example considers the effects of state change in the caller while a callee is still active. It introduces the *c-split* operation, an operation invoked at completion time which splits a single calling item into two.

The previous page shows a grammar, the input graph derived by that grammar, and the item list constructed by the simulation before any nodes have been read. No non-terminals have been predicted at this point, so only one item is necessary.



$$\left[S \Rightarrow -a \overset{[e_1]}{\underset{[e_2]}{\langle}} B \; A \rangle c- \;,\; ((B\ 2\ 3)(A\ 4)),\; \phi \right]_1$$

$$\left[B \Rightarrow \overset{e_1}{+} b- \;,\; nil,\; \{1\}\right]_2$$

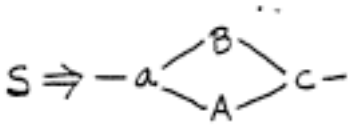
$$\left[B \Rightarrow \overset{e_1}{+} b-b-,\; nil,\; \{1\}\right]_2$$

$$\left[A \Rightarrow \overset{e_2}{+} a \langle B_1 \; B_2 \rangle c- \;,\; nil,\; \{1\}\right]_4$$

Example 1, list 1.

Here is the list after reading node a_1. Item 1 was *active on the node read*; that is, its state contained inputs of that node. The node was acceptable (as per section 4.1.4), and the resulting transition has led to a state containing inputs to the non-terminals B and A. (The '[]' symbols around the edges e_1 and e_2 in item 1 indicate that these edges were present in that item's state immediately after the input node was scanned, but were then deleted as a result of the sub-recognizer call mechanism.)

Only one derivation decision is possible for the A-node, but two are possible for the B-node. Intuitively, item 1 represents the S-recognizer for two parsers. Each has made a different derivation decision for B, but both remain in the same state and were started at the same input position. Thus, we represent both with a single item and use the call list for B in that item to keep track of both outstanding calls.



(node read : a_2)

$[A \Rightarrow -a\overset{[e_3]}{\underset{[e_4]}{\langle}} B_1 \; B_2 \rangle c-,\ ((B_1\ 5\ 6)(B_2\ 7\ 8),\ \{1\}]_4$

$[B \Rightarrow \overset{e_3}{+} b-,\ nil,\ \{4\}]_5$

$[B \Rightarrow \overset{e_3}{+} b-b-,\ nil,\ \{4\}]_6$

$[B \Rightarrow \overset{e_4}{+} b-,\ nil,\ \{4\}]_7$

$[B \Rightarrow \overset{e_4}{+} b-b-,\ nil,\ \{4\}]_8$

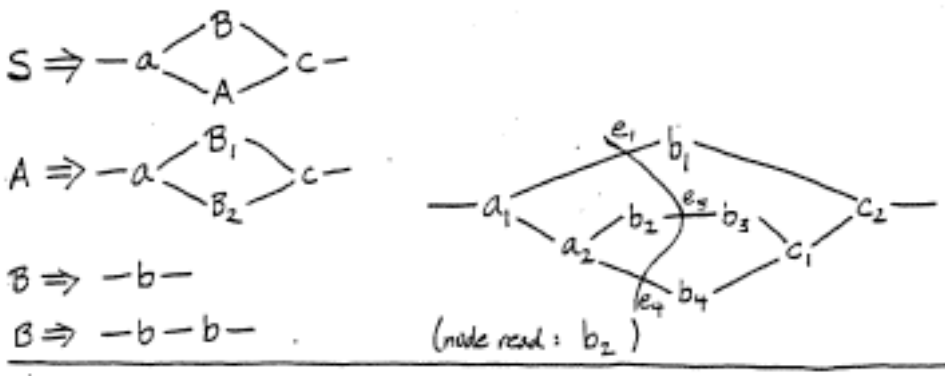
$[B \Rightarrow \overset{e_1}{+} b-,\ nil,\ \{1\}]_2$

$[B \Rightarrow \overset{e_1}{+} b-b-,\ nil,\ \{1\}]_3$

Example 1, list 2.

Now we read node a_2. Item 4 was the only one active on this node, so it makes an appropriate transition. Its new state necessitates predicting the two B-nodes by invoking appropriate sub-recognizers: this happens in a process exactly like that seen in the last list. Items 5, 6, 7, and 8 are the result; notice that although the recognizers for the upper and lower B-nodes are being started on the same list, they are not started at the same input position for the purposes of multiple-call collapsing.

Items 2 and 3 are on this list because, while they were not active on the node read, they are active on a node eligible to be read. This copying-forward of active items insures that, each time a node is read, all the items active on that node will be present on the previous list. Thus, we never have to search back through prior lists for active items.



$[B \Rightarrow -b^{[e_5]}_{+}, nil, \{4,7\}]_5$

$[A \Rightarrow -a\langle {}^{B_1}_{B_2}\rangle c-, ((B_1\ 6)(B_2\ 7\ 8)), \{1\}]_4$

$[A \Rightarrow -a\langle {}^{B_1 \times e_5}_{B_2}\rangle c-, ((B_2\ 7\ 8)), \{1\}]_4$

$[B \Rightarrow -b \overset{e_5}{+} b-, nil, \{9\}]_6$

$[B \Rightarrow \overset{e_4}{+} b-, nil, \{4,9\}]_7$

$[B \Rightarrow \overset{e_4}{+} b-b-, nil, \{4,9\}]_8$

$[S \Rightarrow -a\langle {}^{B}_{A}\rangle c-, ((B\ 2\ 3)(A\ 4\ 9)), \phi]_1$

$[B \Rightarrow \overset{e_1}{+} b-, nil, \{1\}]_2$

$[B \Rightarrow \overset{e_1}{+} b-b-, nil, \{1\}]_3$

Example 1. list 3.

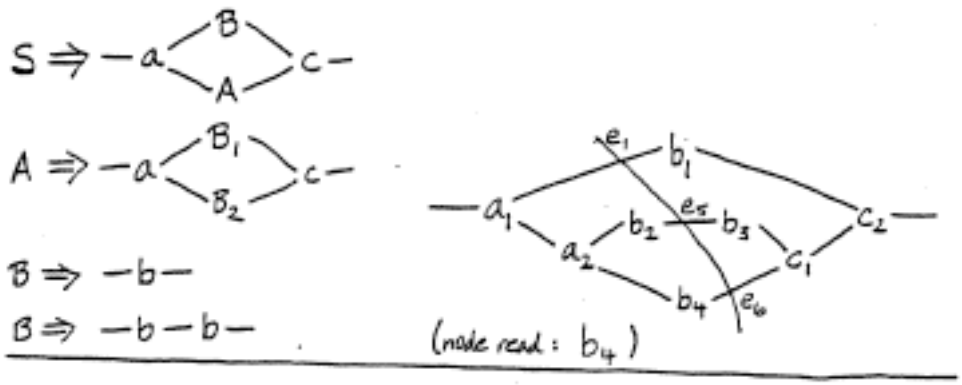
Now the fun begins. We read node b_2 and item 5 moves into an accepting state. We can now complete node B_1 of item 4 by letting item 5 return. But recall that item 4 represents two recognizers: one which predicted B_1 via item 5 and one which predicted B_1 via item 6. If we make the state transition in item 4 called for by the return of item 5, we will have made a spurious state transition in the recognizer which called item 6.

The solution is to *c-split* item 4 into item 9. This process separates into two items the representation of the two recognizers where were merged in item 4. It works by:

1. Copying item 4 to the new item 9. (This gives us two items each of which represent both of the recognizers merged in item 4.)
2. Removing the call to item 5 from item 9, removing the call to item 6 from item 4, and removing the return to item 4 by item 6. (This makes each of item 4 and item 9 represent only one of the two recognizers that were merged in item 4.)
3. Going through all the callees of item 9 and informing them about their new caller. (This keeps the call list of item 9 and the return sets of its callees consistent.)
4. Going through all the callers of item 9 and informing them of their new callee. (This keeps the return set of item 9 and the call lists of its callers consistent.)

The effects of the c-split operation can be seen in items 1, 4, 6, 7, 8, and 9 of the current list; once the c-split is complete, item 5 returns to item 4 as described in section 4.1.5.

Note that items 2 and 3, which have been brought forward to this list because they are active, are unaffected by the c-split. In general, none of the 'siblings' of a c-split item are affected by that split; only its 'parents' (callers) and 'children' (callees) are affected.



$[B \Rightarrow -b \overset{[e_6]}{+}, \text{nil}, \{4, 9\}]_7$

$[A \Rightarrow -a \langle B_1 \overset{e_5}{+} c-, B_2 \rangle, ((B_2\ 8)), \{1\}]_{10}$

$[A \Rightarrow -a \langle B_1 \overset{e_5}{+}, B_2 \overset{e_6}{+} \rangle c-, \text{nil}, \{1\}]_4$

$[A \Rightarrow -a \langle B_1, B_2 \rangle c-, ((B_1\ 6)(B_2\ 8)), \{1\}]_{11}$

$[A \Rightarrow -a \langle B_1, B_2 \overset{e_6}{+} \rangle c-, ((B_1\ 6)), \{1\}]_9$

$[B \Rightarrow -b \overset{e_6}{+} b-, \text{nil}, \{10, 11\}]_8$

$[B \Rightarrow -b \overset{e_5}{+} b-, \text{nil}, \{9, 11\}]_6$

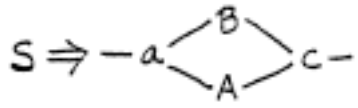
$[S \Rightarrow -a \langle B, A \rangle c-, ((B\ 2\ 3)(A\ 4\ 5\ 10\ 11)), \phi]_1$

$[B \Rightarrow \overset{e_1}{+} b-, \text{nil}, \{1\}]_2$

$[B \Rightarrow \overset{e_1}{+} b-b-, \text{nil}, \{1\}]_3$

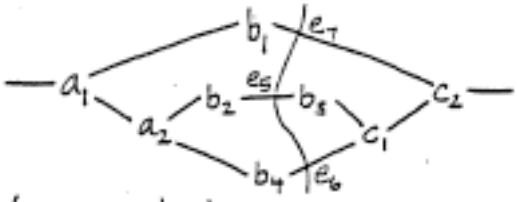
Example 1, list 4.

Reading node b_4 puts item 7 in an accepting state. This c-splits item 4 into item 10 and item 9 into item 11 for the same reasons as item 5 c-split item 4 in the last list, only this time it is the B_2 node that has other derivations pending.



B ⇒ −b−

B ⇒ −b−b−



(node read: b_1)

$[B \Rightarrow -b\overset{[e_7]}{+}, nil, \{1\}]_2$

$[S \Rightarrow -a\langle B\ A\rangle c-, ((B\ 3)(A\ 4\ 9\ 10\ 11)), \phi]_{12}$

$[S \Rightarrow -a\langle B\overset{e_7}{\times} A\rangle c-, ((A\ 4\ 9\ 10\ 11)), \phi]_1$

$[B \Rightarrow -b\overset{e_7}{+}b-, nil, \{12\}]_3$

$[A \Rightarrow -a\langle B_1\overset{e_5}{\times} B_2\rangle c-, ((B_2\ 8)), \{1, 12\}]_{10}$

$[A \Rightarrow -a\langle B_1\ B_2\rangle c-, ((B_1\ 6)(B_2\ 8)), \{1, 12\}]_{11}$

$[A \Rightarrow -a\langle B_1\overset{e_5}{+} B_2\overset{e_6}{+}\rangle c-, nil, \{1, 12\}]_4$

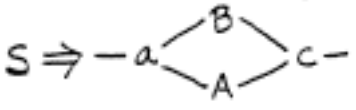
$[A \Rightarrow -a\langle B_1\ B_2\rangle c-, ((B_1\ 6)), \{1, 12\}]_9$

$[B \Rightarrow -b\overset{e_5}{+}b-, nil, \{9, 11\}]_6$

$[B \Rightarrow -b\overset{e_6}{+}b-, nil, \{10, 11\}]_8$

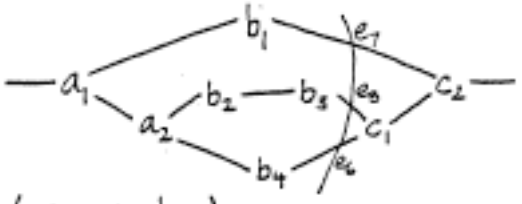
Example 1, list 5.

Now we read node b_1. Items 2 and 3 change state, and item 2's return c-splits item 1 into item 12. Items 4, 9, 10, and 11 are affected by this split.



B ⇒ −b−

B ⇒ −b−b−



(node read : b_3)

$[\text{dead}]_{10}$

$[\text{dead}]_{4}$

$[B \Rightarrow -b-b \overset{[e_3]}{\dagger}, \text{nil}, \{9, 11\}]_{6}$

$[A \Rightarrow -a \langle B_1 \not{e_3}, B_2 \not{e_4} \rangle c-, \text{nil}, \{1, 12\}]_{9}$

$[A \Rightarrow -a \langle B_1, B_2 \rangle c-, ((B_2\ B)), \{1, 12\}]_{11}$

$[B \Rightarrow -b \overset{e_4}{\dagger} b-, \text{nil}, \{10, 11\}]_{8}$

$[B \Rightarrow -b \overset{e_7}{\dagger} b-, \text{nil}, \{12\}]_{3}$

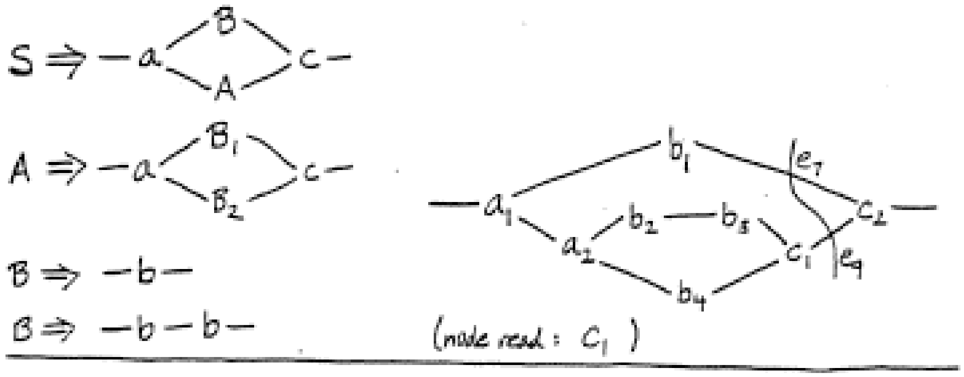
$[S \Rightarrow -a \langle B, A \rangle c-, ((A\ 4\ 9\ 10\ 11)), \phi]_{1}$

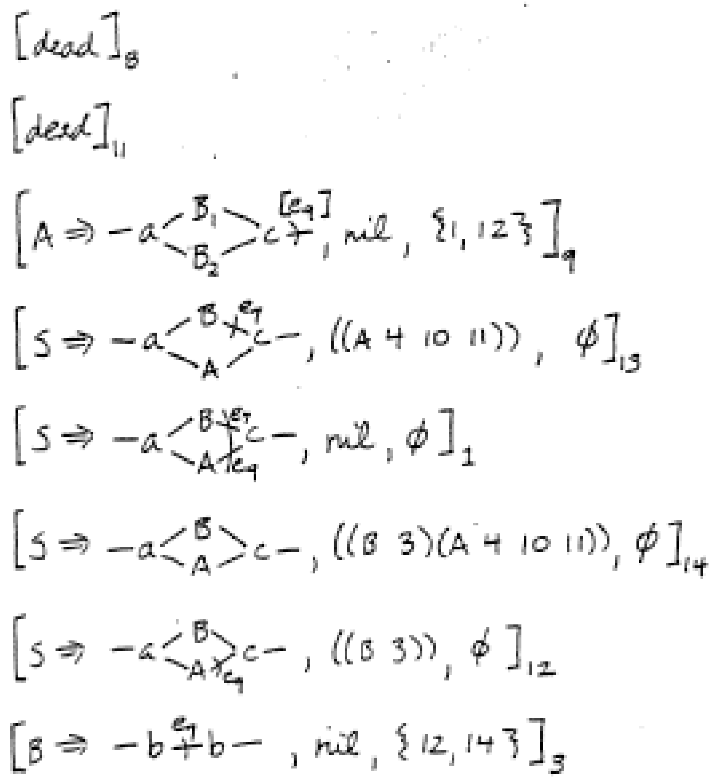
Example 1, list 6.

We read b_3, and the shakeout of spurious parses begins. Items 4 and 10 are active on the node read but it is not acceptable, so their recognizers reject. We indicate this by marking these items as *dead*, a step which was not necessary in the string case. The point here is that, although the recognizers represented by these items have rejected, they may have pending calls to other recognizers. If these other recognizers were allowed to return to their dead callers, these returns might cause state transitions which lead to spurious parses. By marking the items for rejecting recognizers with *dead*, we will know to suppress future returns to those items.

Reading b_3 also puts item 6 into an accepting state, but its return to items 9 and 11 does *not* cause them to c-split because they have no other recognizers pending for the B_1 node that item 6 completed.

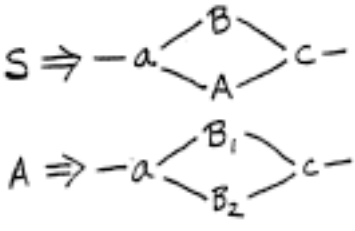
Items 1, 3, and 8 are copied unchanged from the previous list because they are active. Note that item 8, although it contains item 10 in its return list, is unchanged by the fact that item 10 has died. If item 8 were ever to move into an accepting state, its return to item 10 would simply be ignored.

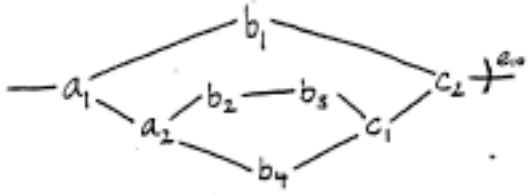




Example 1, list 7.

We read node c_1. Items 8 and 11 die. Item 9 moves into an accepting state, causing items 1 and 12 to c-split into items 13 and 14. This split takes place even though none of the pending calls for the A-node completed by item 9 are to live items; the decision whether or not to c-split is made solely on the basis of number of pending calls. On the other hand, because items 4, 10, and 11 are dead, the updating of their return sets normally done by the c-splits of items 1 and 12 are suppressed.





(node read : C_2)

$[dead]_{13}$

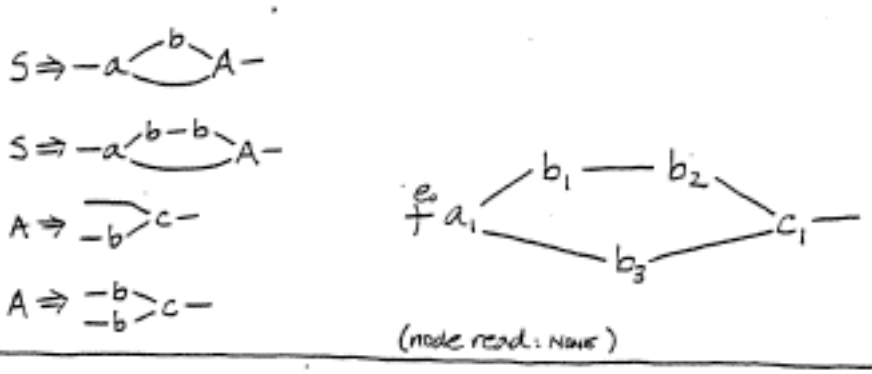
$[dead]_{12}$

$[dead]_3$

$[S \Rightarrow -a \begin{matrix} B \\ A \end{matrix} c \overset{e_{10}}{+}, nil, \phi]_1$

Example 1, list 8.

We read node c_2. Item 1, which is a top-level item, moves into an accepting state that includes all of its trailing edges: the input is accepted.



$$[S \Rightarrow \overset{e}{+}a\overset{b}{\frown}A-,\ nil,\ \phi]_1$$

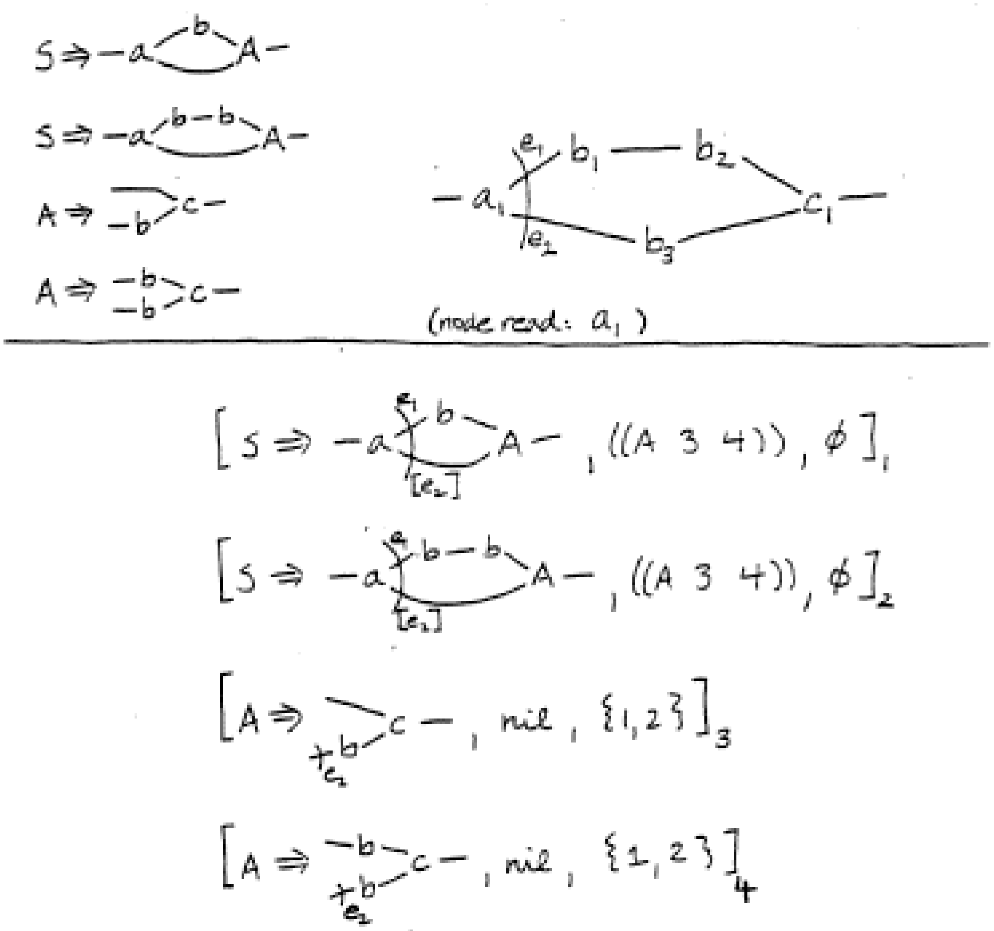
$$[S \Rightarrow \overset{e_0}{+}a\overset{b-b}{\frown}A-,\ nil,\ \phi]_2$$

Example 2, list 0.

Example 2

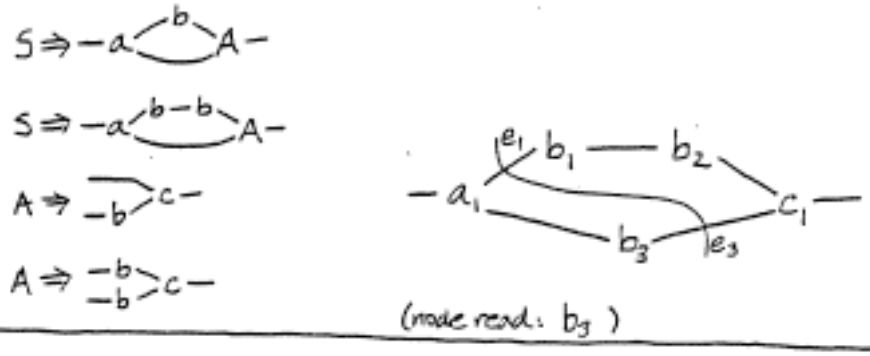
This example explores the interaction of multiple-call collapsing and staggered invocation. It introduces the *p-split* operation, in which an item is split into two items as the result of a prediction operation.

We start off with just two items, one for each derivation of S.



Example 2, list 1.

Upon reading node a_1, both S-recognizers make transitions into a state containing an input of A. Since the two recognizers agree as to which input edge is input to which port of A, multiple-call collapsing takes place and the A-recognizers invoked will each return to both S-recognizers.



$[A \Rightarrow \overset{}{-b}\overset{e_3}{\times}c-\ ,\ nil\ ,\ \{1,2\}]_3$

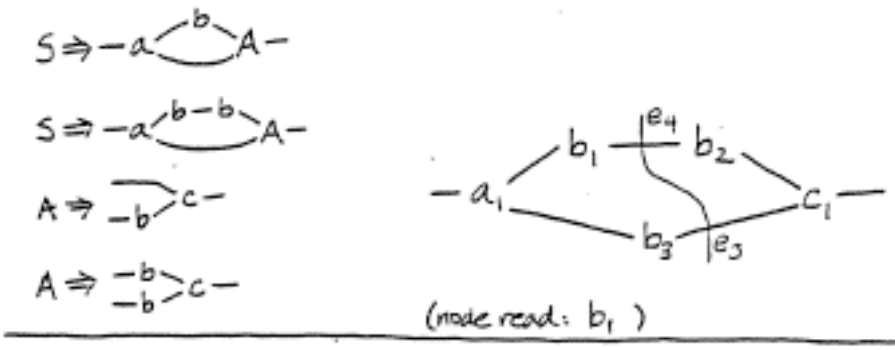
$[A \Rightarrow \overset{-b}{-b}\overset{e_3}{\times}c-\ ;\ nil\ ,\ \{1,2\}]_4$

$[S \Rightarrow -a\overset{e_1}{\times}b\ A-\ ,\ ((A\ 3\ 4))\ ,\ \emptyset]_1$

$[S \Rightarrow -a\overset{e_1}{\times}b-b\ A-\ ,\ ((A\ 3\ 4))\ ,\ \emptyset]_2$

Example 2, list 2.

We read node b_3. and both A-recognizers make appropriate transitions. The items for the S-recognizers are active and unchanged.



$[S \Rightarrow -a\,b\,[e_4]\,A-\ ,\ ((A\ 3\ 4))\ ,\ \phi]_1$

$[A \Rightarrow -b\,c-\ ,\ nil\ ,\ \{2\}]_5$ (b $\not e_3$)

$[A \Rightarrow -b\,c-\ ,\ nil\ ,\ \{1\}]_3$ (e_4, b $\not e_3$)

$[A \Rightarrow -b\,c-\ ,\ nil\ ,\ \{2\}]_6$ (b $\not e_3$)

$[A \Rightarrow -b\,c-\ ,\ nil\ ,\ \{1\}]_4$ (e_4 b, b $\not e_3$)

$[S \Rightarrow -a\,b\,e_4\,b\,A-\ ,\ ((A\ 5\ 6))\ ,\ \phi]_2$

Example 2, list 3.

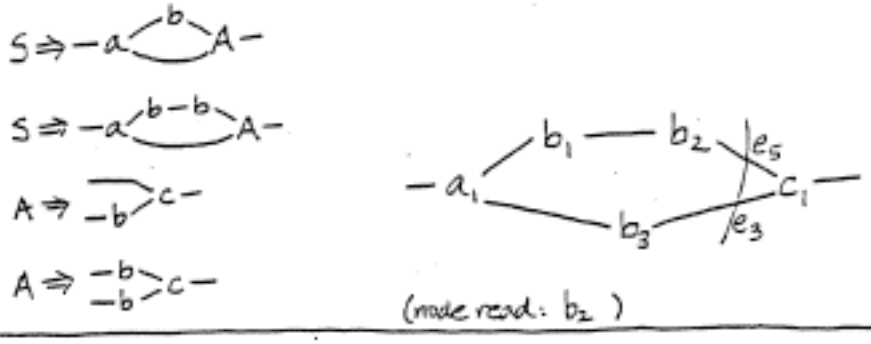
When we read node b_1, the S-recognizer of item 1 moves into a state containing the other input of A. This recognizer must pass this new input down to the A-recognizers of items 3 and 4, but these items also represent recognizers invoked by item 2, which does not want to pass down this second input.

This situation is complementary to that in which we c-split a caller, and the solution is also complementary: we p-split the callee. By this we mean we split the representations of the two recognizers merged in item 3 among two items, and we do the same with item 4. Each of these p-splits is accomplished similarly, for item 3 we do it by:

1. Copying item 3 to item 5. (This gives us two items, each representing a recognizer invoked by two parsers.)
2. Removing item 3's return to item 2, item 5's return to item 1, and item 2's call of item 3. (This makes each of the items represent a recognizer invoked by just one of the two parsers.)

In the general case, items 3 and 4 might have had outstanding calls, in which case we would have made their callees return to both them and their p-splits.

The result of the p-split is that we can now pass down the new input from item 1 to items 3 and 4 without hurting the recognizers invoked by item 2. Thus, we are ready to read the next node.



[dead]$_3$

[A⇒ -b e5 c- / -b e3 , nil , {1}]$_4$

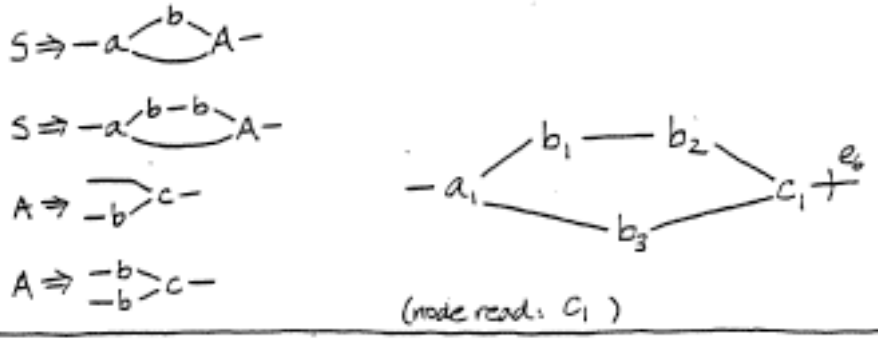
[S⇒ -a b-b [e5] A- , ((A 5 6)) , ϕ]$_2$

[A⇒ e5 c- / -b e3 , nil , {2}]$_5$

[A⇒ e5 b-c- / -b e3 , nil , {2}]$_6$

Example 2, list 4.

We read node b_2. Item 3 dies expecting a c, and item 4 makes a normal state transition. Item 2 moves into a state containing its A-node's other input: this input can be passed down to items 5 and 6 without p-splitting them because they have no other callers.



$[dead]_6$

$[A \Rightarrow -b-c\overset{[e_6]}{+}, nil, \{1\}]_4$

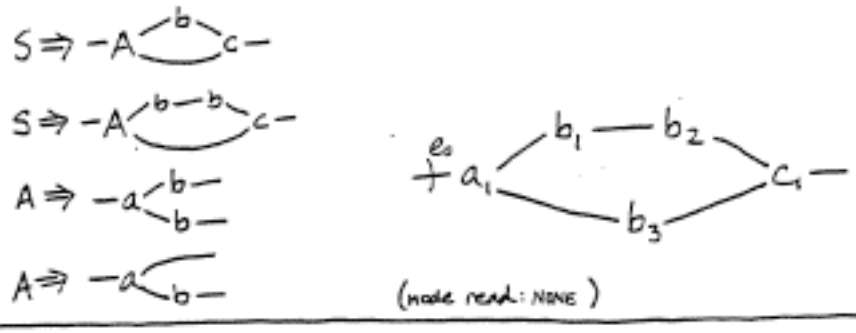
$[S \Rightarrow -a-b-A\overset{e_6}{+}, nil, \phi]_1$

$[A \Rightarrow -b-c\overset{[e_7]}{+}, nil, \{2\}]_5$

$[S \Rightarrow -a-b-b-A\overset{e_6}{+}, nil, \phi]_2$

Example 2, list 5.

We read the final c. Item 6 dies expecting a b, while items 4 and 5 move into accepting states. They then return to items 1 and 2, both of which accept.



$[S \Rightarrow {}^{[e_0]}_{\dagger} A\langle b\ c\rangle -,\ ((A\ 3\ 4)),\ \phi]_1$

$[S \Rightarrow {}^{[e_0]}_{\dagger} A\langle b\text{-}b\ c\rangle -,\ ((A\ 3\ 4)),\ \phi]_2$

$[A \Rightarrow {}^{e_0}_{\dagger} a\langle b-,\ b-\rangle,\ nil,\ \{1,2\}]_3$

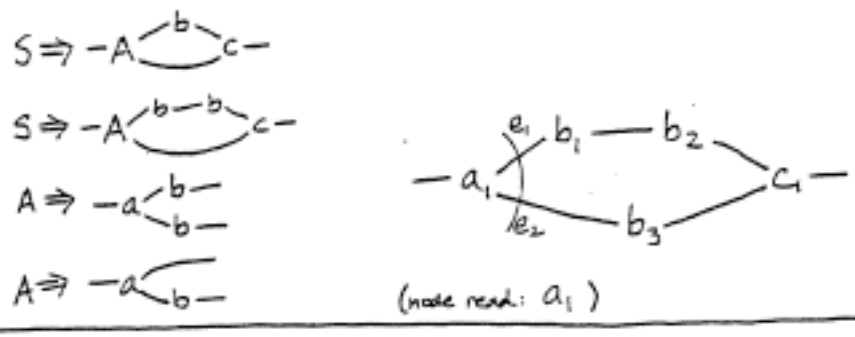
$[A \Rightarrow {}^{e_0}_{\dagger} a\langle -,\ b-\rangle,\ nil,\ \{1,2\}]_4$

Example 3, list 0.

Example 3.

This example demonstrates staggered reduction. No new operations are introduced.

The first list starts with items for two alternate S-recognizers. Both of these recognizers expect an A at the same point in the input, so two A-recognizers are invoked and both will return to both S-recognizers.



$[A \Rightarrow -a\overset{[e_1]}{\not}\underset{e_2}{b}-\cdot\,,\ nil,\ \{1,2\}]_4$

$[S \Rightarrow -A\overset{b}{\frown}c-\,,\ ((A\ 3)),\ \phi]_5$

$[S \Rightarrow -A\overset{e_1}{\not}b\frown c-\,,\ ((A\ 4)),\ \phi]_1$

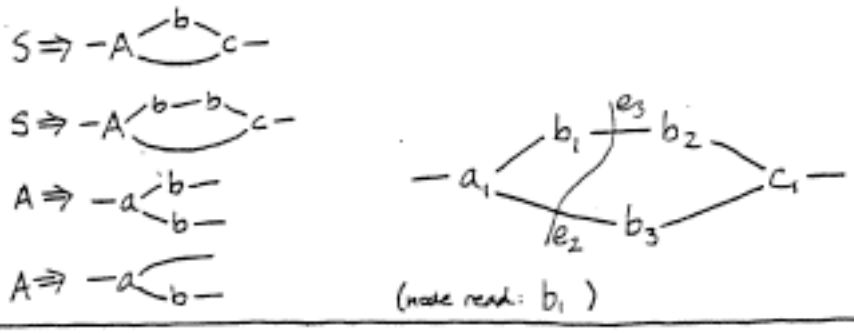
$[S \Rightarrow -A\overset{b-b}{\frown}c-\,,\ ((A\ 3)),\ \phi]_6$

$[S \Rightarrow -A\overset{e_1}{\not}b-b\frown c-\,,\ ((A\ 4)),\ \phi]_2$

$[A \Rightarrow -a\overset{e_1}{\not}\underset{e_2}{b}-\,,\ nil,\ \{5,6\}]_3$

Example 3, list 1.

Node a_1 is read, leading item 4 into an accepting state for the edge e_1. Its return causes items 1 and 2 to c-split into items 5 and 6. Notice that, although item 4 has returned one of its outputs, it remains in the call lists of its callers. It will be removed only when it returns all of its outputs.



$[A \Rightarrow -a\ b\ \overset{[e_3]}{\times}\ \underset{e_2}{b}-\ ,\ nil\ ,\ \{5,6\}]_3$

$[S \Rightarrow -A\ \overset{e_3}{+}\ b\ c-\ ,\ ((A\ 3))\ ,\ \phi]_5$

$[S \Rightarrow -A\ \overset{e_3}{+}\ b-b\ c-\ ,\ ((A\ 3))\ ,\ \phi]_6$

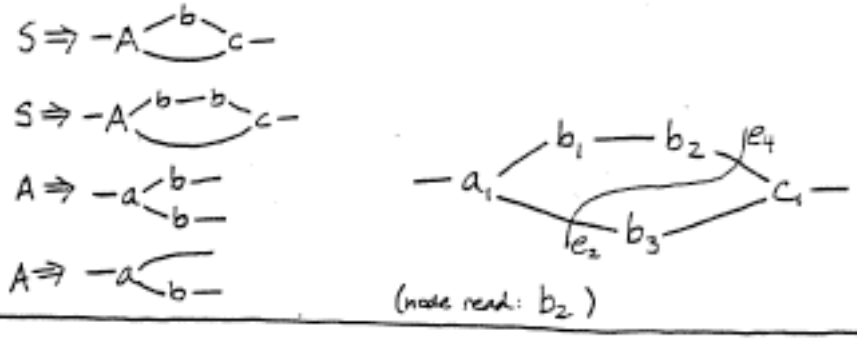
$[S \Rightarrow -A\ b\ \overset{e_3}{\times}\ c-\ ,\ ((A\ 4))\ ,\ \phi]_1$

$[S \Rightarrow -A\ b\ \overset{e_3}{+}\ b\ c-\ ,\ ((A\ 4))\ ,\ \phi]_2$

$[A \Rightarrow -a\ \underset{e_2}{+}\ b-\ ,\ nil\ ,\ \{1,2\}]_4$

Example 3, list 2.

Node b_1 is read, leading item 3 into an accepting state of the edge e_3. Items 5 and 6 need not be c-split since they have no other calls pending for their A-node. Item 4 is active and unchanged.



[dead]$_1$

[S ⇒ -A b-b×e4 c- , ((A 4)) , ∅]$_2$

[S ⇒ -A b×e4 c- , ((A 3)) , ∅]$_5$

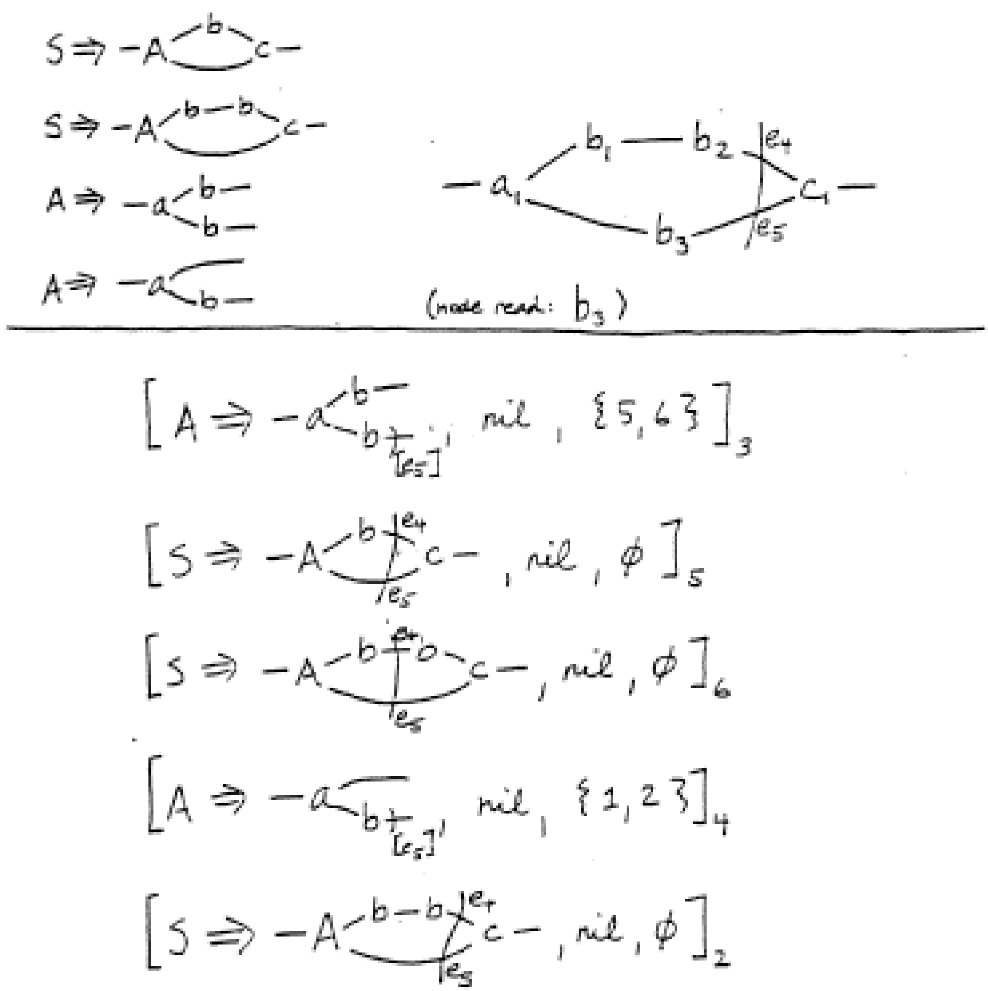
[S ⇒ -A b e4 b c- , ((A 3)) , ∅]$_6$

[A ⇒ -a b- , nil , {5, 6}]$_3$
e2 b-

[A ⇒ -a , nil , {1, 2}]$_4$
e2 b-

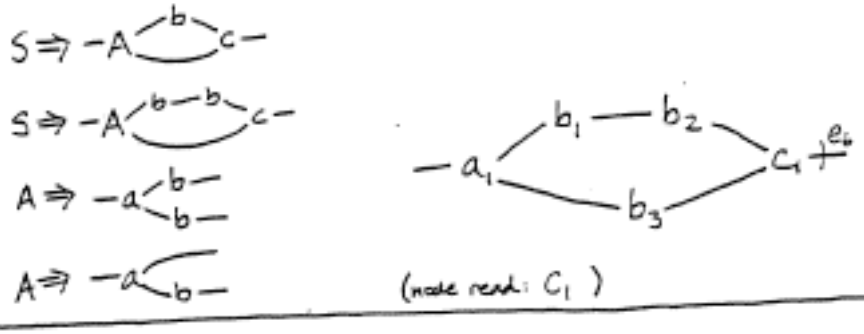
Example 3, list 3.

Node b_2 is read, killing item 1 which was expecting a c. Items 2, 5, and 6 all make state transitions; items 3 and 4 are unchanged.



Example 3, list 4.

Node b_3 is read, leading both items 3 and 4 into accepting states. Their returns are uneventful.



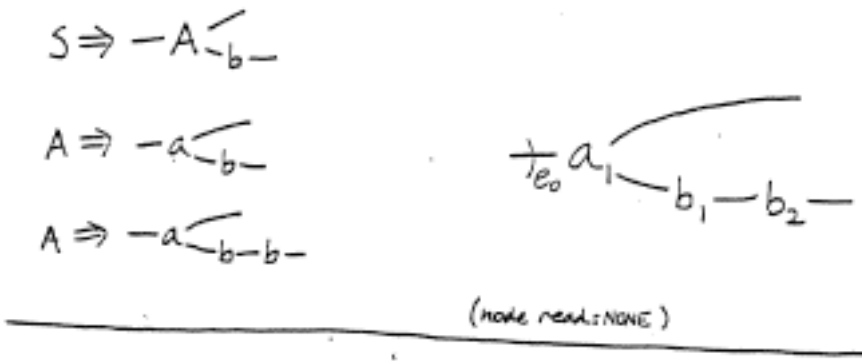
$[\, S \Rightarrow -A\overset{b}{\frown}c\overset{e_0}{+}\,,\ nil,\ \phi\,]_5$

$[\, dead\,]_6$

$[\, S \Rightarrow -A\overset{b-b}{\frown}c-\,,\ nil,\ \phi\,]_2$

Example 3, list 5.

Node c_1 is read. The results are straightforward.

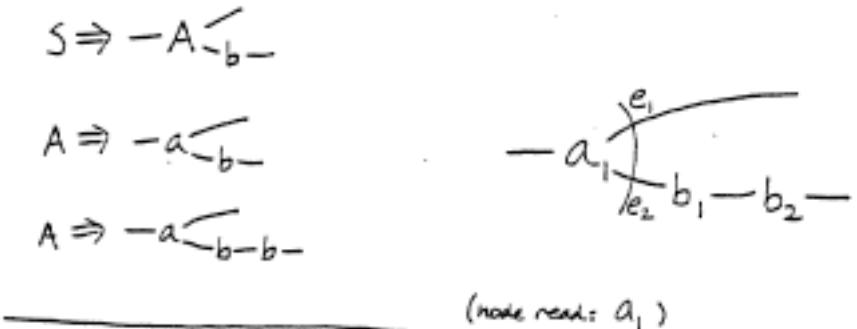


Example 4, list 0.

Example 4

This example explores an interaction between staggered reduction and duplicate-item merging which the reader may have anticipated after seeing the previous example. No new operations are introduced.

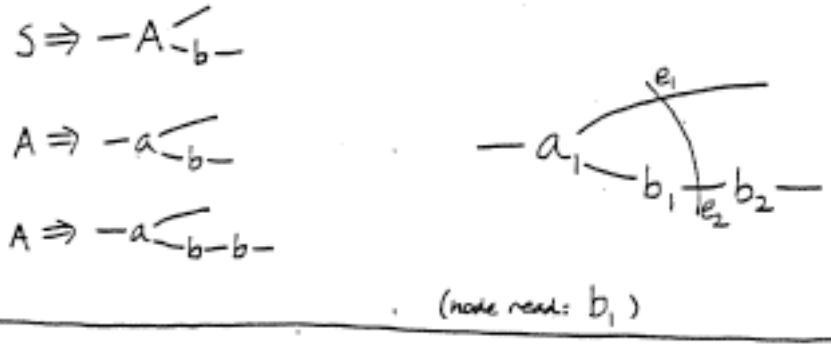
We start by predicting the A-node in the S-recognizer in all possible ways.



Example 4, list 1.

Upon reading node a_1, both the a-recognizers are ready to accept edge e_1. The return of item 2 causes item 1 to c-split into item 4, but then the return of item 3 moves item 4 into the same state as item 1, so it is merged into that item.

A point worth noting here is that only items which were originally c-splits of each other can ever be merged. This is because c-splitting is the only way to get two items which represent the same recognizer started at the same input position.



$$[A \Rightarrow -a\overset{b\ \dagger_{[e_3]}}{\frown}, \text{nil}, \{1\}]_2$$

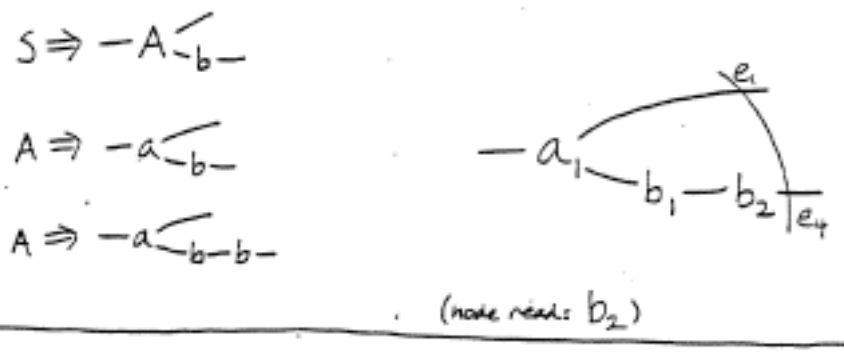
$$[S \Rightarrow -A\overset{\dagger_{e_1}}{\frown}b-, ((A\ 3)), \phi]_5$$

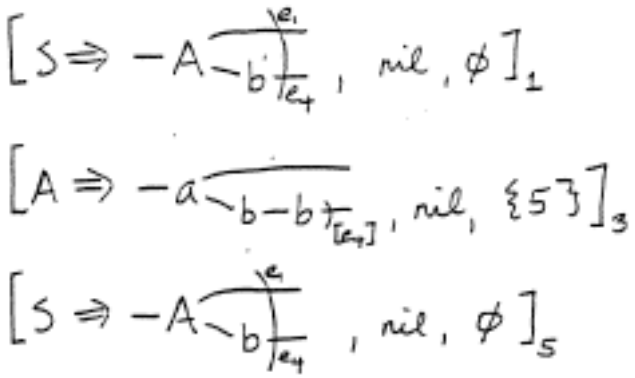
$$[S \Rightarrow -A\overset{\dagger_{e_1}}{\underset{e_3}{\dagger}}b-, \text{nil}, \phi]_1$$

$$[A \Rightarrow -a\overset{b\ \dagger_{e_3}b-}{\frown}, \text{nil}, \{5\}]_3$$

Example 4, list 2.

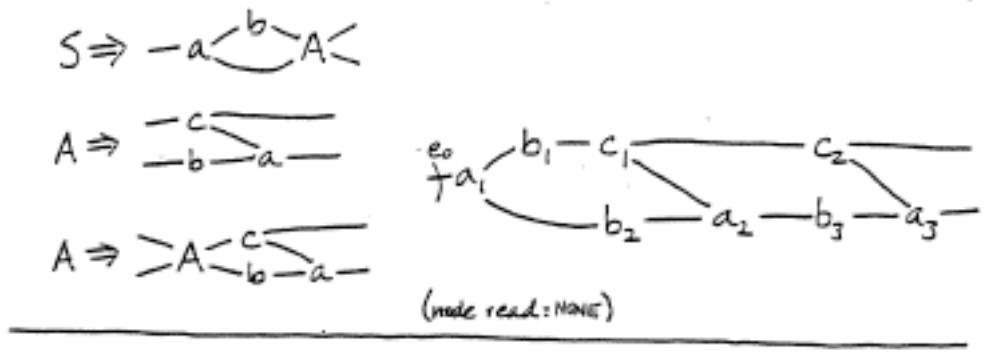
Reading node b_1 moves item 2 (but not item 3) into a state accepting e_3. This causes item 1 to be e-split once again, this time into item 5.





Example 4. list 3.

Node b_2 is read. The results are straightforward.

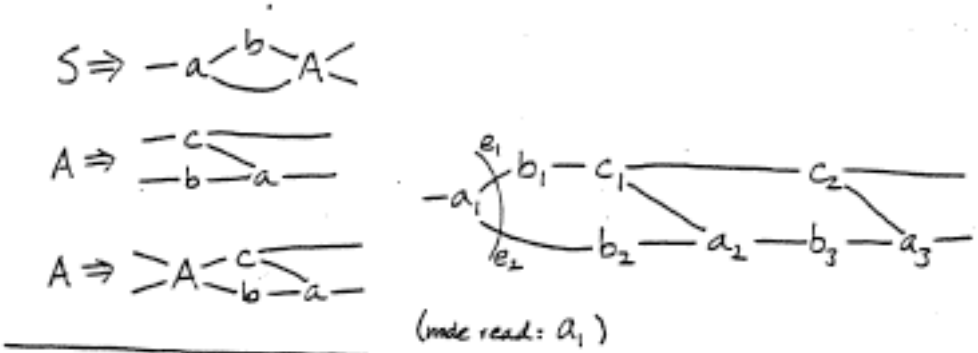


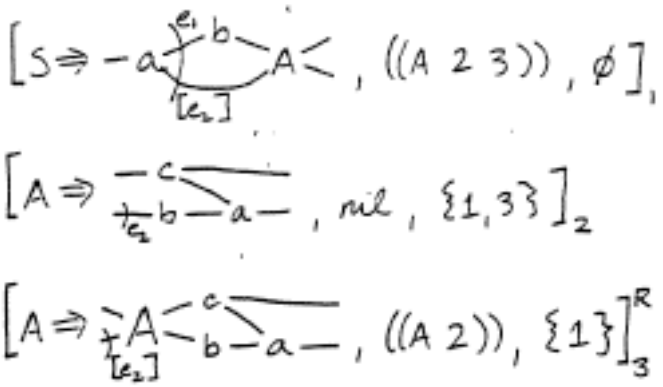
Example 5, list 0.

Example 5

Our last example is our most complex one. It demonstrates left recursion, and introduces the *r-split* operation. Staggered invocations and reductions are thrown in for good measure.

Things start out simply enough, as we invoke a recognizer for s.





Example 5, list 1.

Reading node a_1 moves the S-recognizer into a state where it must predict one of the S-node's inputs. Multiple-call collapsing happens as it did in the string case, with both items 1 and 3 calling item 2 for the non-recursive expansion of A. Unlike the string case, however, item 3 does not explicitly call itself recursively; instead, it is marked with the *R-flag*.

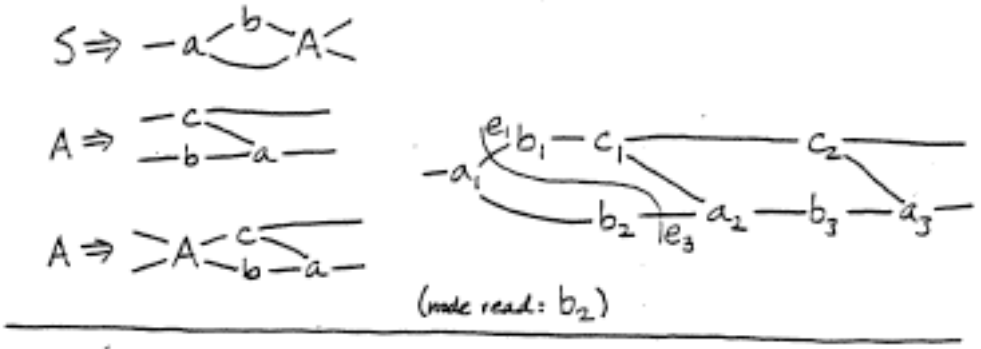
Here are the intuitions behind this marker: If we encode by 'S' an S-recognizer, by A_n a non-recursive A-recognizer, and by A_r a recursive A-recognizer, then the simulation is currently representing an infinite number of parsers, each with one of the following call structures:

$$\begin{aligned}
&S \rightarrow A_n \\
&S \rightarrow A_r \rightarrow A_n \\
&S \rightarrow A_r \rightarrow A_r \rightarrow A_n \\
&\vdots \\
&S \rightarrow A_r \rightarrow \ldots \rightarrow A_r \rightarrow A_n. \\
&\vdots
\end{aligned}$$

The simulation does this by keeping just the two structures

$$\begin{aligned}
&S \rightarrow A_n \\
&S \rightarrow A_r \rightarrow A_n
\end{aligned}$$

(and it shares the A and A_n between the two). The the R-flag on item 3 indicates that it is the recursive A-recognizer which is being used to encode the infinite sequence of such recognizers present in the unoptimized case.

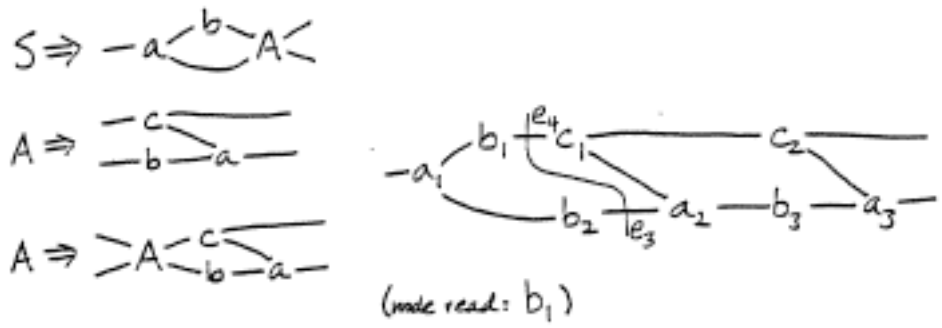


$$\left[A \Rightarrow \begin{array}{l} c \\ b \; e_3 \; a \end{array},\ nil,\ \{1,3\}\right]_2$$

$$\left[S \Rightarrow -a \overset{e_1}{\times} b\ A,\ ((A\ 2\ 3)),\ \emptyset\right]_1$$

Example 5, list 2.

Node b_3 is read, causing the non-recursive A-recognizer to make a state transition.



$[S \Rightarrow -a\overset{b}{\frown}\overset{[e_1]}{\times}A<\ ,\ ((A\ 2\ 3))\ ,\ \emptyset\ ]_1$

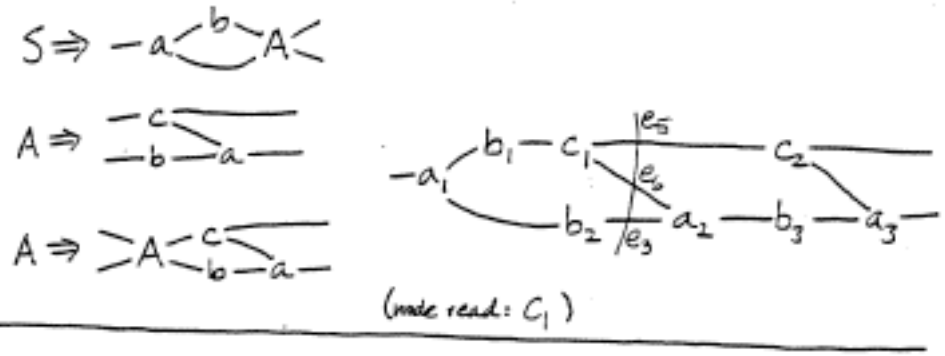
$[$ dead (multiple-call collapsing) $]_4$

$[A \Rightarrow \overset{e_4}{\frown}c\overset{\ }{\frown}\ -b\overset{e_3}{\frown}a-\ ,\ nil\ ,\ \{1,3\}\ ]_2$

$[A \Rightarrow \overset{[e_4]}{\times}A\overset{c}{\frown}\ -b-a-\ ,\ ((A\ 2))\ ,\ \{1\}\ ]^R_3$

Example 5, list 3.

This is a cute one. Node b_1 is read, allowing item 1 to pass down the other input of its *A*-node to items 2 and 3. This first p-splits item 2 into item 4, because item 2 is also called by item 3 and the decision to p-split is made solely on the basis of number of callers, not on whether they agree as to added inputs. But when item 1 passes the new input to item 3 and item 3 goes to pass it on to item 4, such an action would make item 4 and item 2 calls on the same recognizer at the same point, so multiple-call collapsing acts by redirecting item 3's call to item 2 instead of item 4. This in turn removes the last return pointer from item 4, so it is marked as dead. The net effect is to (correctly) keep item 1 and item 3 both calling item 2 for the non-recursive expansion of their *A*-nodes.



$$\left[A \Rightarrow \; , \; \text{nil} \; , \; \{1,3\}\right]_1$$

$$\left[S \Rightarrow \; , \; ((A\ 3\ 6)) \; , \; \phi\right]_5$$

$$\left[S \Rightarrow \; , \; ((A\ 2)) \; , \; \phi\right]_1$$

$$\left[A \Rightarrow \; , \; ((A\ 3)) \; , \; \{5\}\right]^R_6$$

$$\left[A \Rightarrow \; , \; ((A\ 2)) \; , \; \{5,6\}\right]_3$$

Example 5, list 4.

Reading node c_1 moves the non-recursive recognizer for A (item 2) into an accepting state. This causes item 1 to c-split as usual into item 5. Item 3, however, does an *r-split* into item 6, which means it does a normal c-split but then loses its r-flag and picks up its split image as a caller.

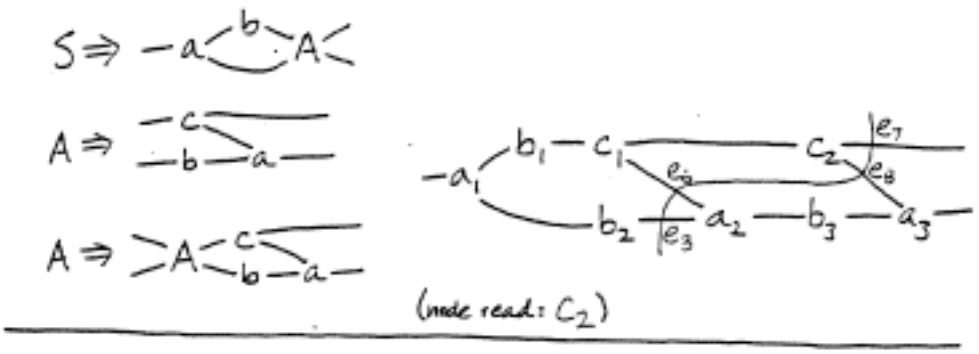
Once again, let us look at the intuition underlying this action. If we flag a recognizer that has returned with an asterisk, and a recognizer that has made a state transition as the result of a return with a prime, then the parsers being simulated now have one of the following call structures:

$$
\begin{aligned}
&S' \rightarrow A_n^* \\
&S \rightarrow A_r' \rightarrow A_n^* \\
&S \rightarrow A_r \rightarrow A_r' \rightarrow A_n^* \\
&\vdots \\
&S \rightarrow A_r \rightarrow \ldots \rightarrow A_r \rightarrow A_r' \rightarrow A_n^*. \\
&\vdots
\end{aligned}
$$

The simulator now represents these structures using the following three structures

$$
\begin{aligned}
&S' \rightarrow A_n^* \\
&S \rightarrow A_r' \rightarrow A_n^* \\
&S \rightarrow A_r \rightarrow A_r' \rightarrow A_n^*
\end{aligned}
$$

where item 1 is S', item 2 is A_n^*, item 5 is S, item 3 is A_r', and item 6—the r-split of item 3—is the A_r recognizer flagged as the representative of the infinite chain. A simple c-split of item 3 would have been insufficient to build this new structure because it would have made items 3 and 6 siblings, instead of child and parent. In addition, it would not have correctly marked item 6 as the A_r recognizer instead of item 3.



$[dead]_1$

$[A \Rightarrow \;>A\!-\!c\!-\!b\!-\!a\!-\;,\; ((A\ 2)),\; \{5,6\}]_3$

$[S \Rightarrow -a\!<\!b\!>\!A\!<\;,\; ((A\ 6\ 8)),\; \phi]_7$

$[S \Rightarrow -a\!<\!b\!>\!A\!<\;,\; ((A\ 3)),\; \phi]_5$

$[A \Rightarrow \;>A\!-\!c\!-\!b\!-\!a\!-\;,\; ((A\ 6)),\; \{7\}]_8^R$

$[A \Rightarrow \;>A\!-\!c\!-\!b\!-\!a\!-\;,\; ((A\ 3)),\; \{7,8\}]_6$

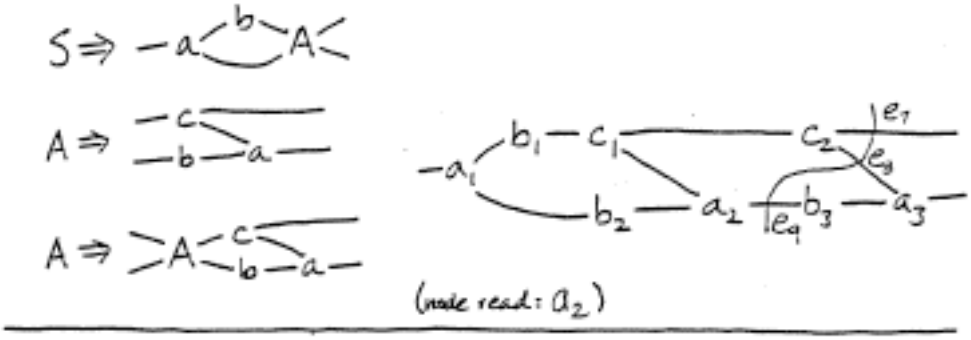
$[A \Rightarrow -c\!-\!b\!-\!a\!-\;,\; nil,\; \{1,3\}]_2$

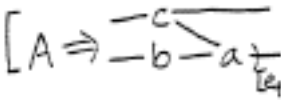
Example 5, list 5.

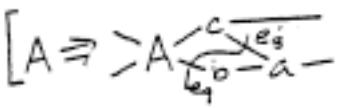
After reading the next c-node, the whole r-split process happens again. The simulation is now keeping the following structure:

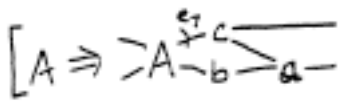
$$
\begin{aligned}
&S' \to A_n^* \\
&S' \to A_r'^* \to A_n^* \\
&S \to A_r' \to A_r'^* \to A_n^* \\
&S \to A_r \to A_r' \to A_r'^* \to A_n^*.
\end{aligned}
$$

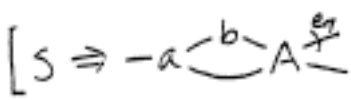
The reader should be sure he understands how this structure was attained!



$[A \Rightarrow$  $, nil, \{1,3\}]_2$

$[A \Rightarrow$  $, nil, \{5,6\}]_3$

$[A \Rightarrow$  $, ((A\ 3)), \{7,8\}]_6$

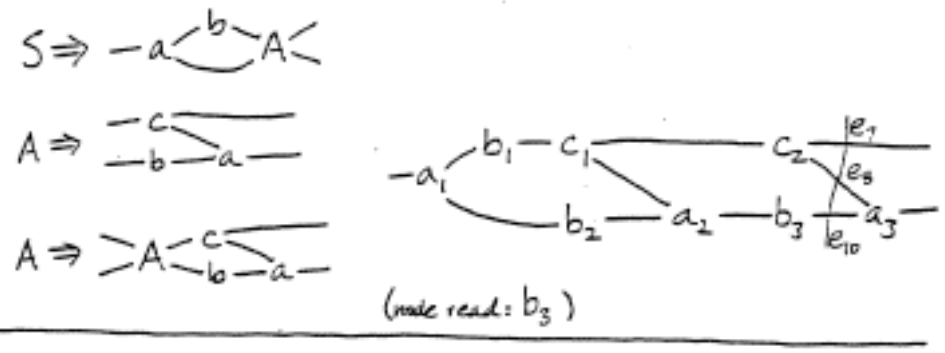
$[S \Rightarrow$  $, ((A\ 3)), \phi]_5$

Example 5, list 6.

With the reading of node a_2, the bottom finally begins to come off the stack. The simulation now contains the following structure

$$\begin{aligned} S' &\to A_\tau'^* \\ S &\to A_\tau' \to A_\tau'^* \\ S &\to A_\tau \to A_\tau' \to A_\tau'^*. \end{aligned}$$

which differs from that of the last list in that all the non-recursive A-recognizers have gone away. The reader should consider what the structure would have been if we had read another c node and started another level of recursion.



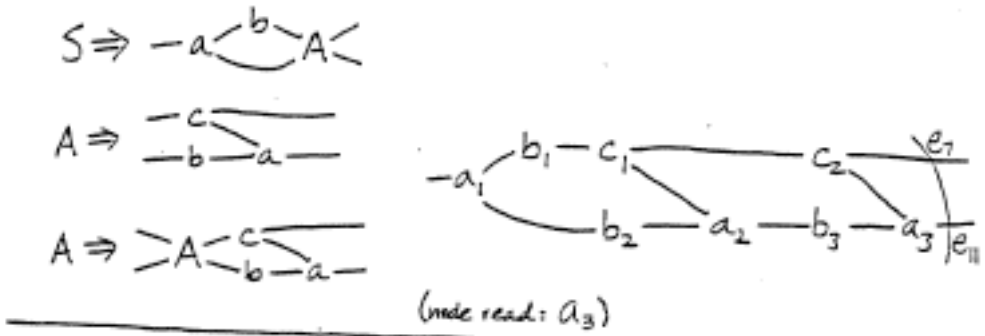
$[A \Rightarrow >A\ c\ a\ b\ ,\ nil,\ \{5,6\}]_3$

$[A \Rightarrow >A\ c\ a\ b\ ,\ ((A\ 3)),\ \{7,8\}]_6$

$[S \Rightarrow -a\ b\ A\ ,\ ((A\ 3)),\ \phi]_5$

Example 5, list 7.

The A_r^{t*} recognizer on the bottom of the stack makes a transition on b_3.



$$[A \Rightarrow >A\langle\, {}^{c}_{b\,a}\,,\ [a_{11}]\,,\ nil\,,\ \{5,6\}]_3$$

$$[S \Rightarrow -a\langle b\rangle A\,,\ e_7\,,\ e_{11}\,,\ nil\,,\ \phi\,]_5$$

$$[A \Rightarrow >A\,,\ e_7\,,\ e_{11}\,,\ c\,,\ b\,,\ a\,,\ nil\,,\ \{7,8\}]_6$$

Example 5, list 8.

The stack unwinds another level. The final configuration kept by the simulator is:

$$
\begin{array}{lll}
S'' & & \\
S & \rightarrow A'_r & \text{(item 7} \rightarrow \text{item 6)} \\
S & \rightarrow A_r \rightarrow A'_r & \text{(item 7} \rightarrow \text{item 8} \rightarrow \text{item 6).}
\end{array}
$$

4.2.4. Algorithm Description

We are now ready to state our flow graph parsing algorithm. The algorithm takes as input a flow grammar and a flow graph: it determines whether the graph is generated by the grammar. As with our version of Earley's algorithm, the output of the algorithm is a sequence of item lists—one for each node in the input—which represent all the possible configurations of a non-deterministic graph parser when run on that input. The algorithm does not output a parse tree for the input, although it can be modified to do so in a manner similar to that presented in the last chapter.

The algorithm operates by using a list of items I_i to keep track of all the configurations a parser might be in after reading the first i nodes it chooses to read. Given lists $I_0, \ldots, I_{i-1}$, the algorithm constructs list I_i by using three operations: a *scanner* operation, a *predictor* operation, and a *completer* operation. These operations in turn use three sub-operations: the *p-split*, the *c-split*, and the *r-split*. We first describe the nature of all these operations, and then how the algorithm uses them to construct the lists $I_0, I_1, \ldots, I_n$.

The P-Split Operation

The p-split operation takes as input an item i, a non-terminal node n which i is deriving, a item c which has called i, and a list I_j. It performs the following actions:

1. It creates a new item i' whose state part is that of i, whose call list is that of i, and whose return set is that of i except that c is removed.
2. It adds i' to I_j.
3. It goes through the live callees of i and adds i' to each of their return sets.
4. It goes through all the live callers of i except c and replaces their calls to i by calls to i'.
5. It changes i's return set to be the singleton $\{c\}$.

Items i and i' are said to be p_j-*splits* of each other. This relation is transitive—any other p_j-splits of i are p_j-splits of i' (and vice versa)—but does not persist across lists: a p_j-split of i is not a p_k-split of i for $i \neq j$.

The C-Split Operation

The c-split operation takes as input an item i, a non-terminal node n in i's target graph, an item c which i has called to derive n, and a list I_j. It performs the following actions:

1. It creates a new item i' whose state part is that of i, whose return list is that of i, and whose call list that of i except that c is removed from the callees for n.
2. It adds i' to I_j.
3. It goes through the live callers of i and adds i' to any call list on which i appears.
4. It goes through all the live callees that i has pending for nodes other than n and adds i' to each of their return sets.
5. It goes through all the live callees other than c that i has pending for n and replaces i by i' in their return sets.
6. It makes c the only callee for n in i.

The items i and i' are said to be *c-splits* of each other. This relation is transitive—any other c-splits of i are c-splits of i' and vice versa—and it persists across lists: a c-split of i created on a list I_j a c-split of all those created on other lists.

The R-Split Operation

The r-split operation takes the same inputs i, n, c, I_j as the c-split. It first performs a c-split operation to produce an item i' on list I_j, and then takes the following actions:

1. It marks i' with the R-flag.
2. It removes the R-flag from i.
3. It adds i to the call list of n in i', and adds i' to the return set of i.

The Scanner Operation

The scanner operation takes as input an item i from list I_{j-1} and the j-th input node to be read n_j. Let s be the state part of i. If s is empty (i is suspended), the scanner does nothing. If s is non-empty, but none of its pairs contain edges which are inputs of n_j, then the scanner adds i unchanged to

list I_j. If s does contain inputs of n_j, but n_j is determined to be *unacceptable* as described in section 4.1.4, the scanner marks i as *dead* and adds it to list I_j. Finally, if n_j is acceptable to s, the scanner (i) computes a new state s' as described in section 4.1.4, changes the state part of i to s', and adds i to list I_j.

The Predictor

The *predictor* operation takes as input an item i from list I_j. Let s be the state part of i. If none of the target edges of s are inputs to non-terminals, the predictor does nothing. Otherwise, for each non-terminal node n which has one or more input edges in s, the predictor distinguishes two cases:

Case (i) The item i already has calls pending for n.

Let $i_1, \ldots, i_m$ be the live items i has called for n. Then for each i_k, the predictor does the following:

1. If i_k has callers besides i, the predictor invokes the p-split operation on i_k, n, i, and I_j.
2. Let s_k be the state gotten by augmenting i_k's current state in accordance with the calling conventions of section 4.1.5. If the predictor has already added a p_j-split i' of i_k with state-part s_k to I_j, then the predictor adds i to the call list of i' and replaces i_k by i' on the call list for n in i. Otherwise, the predictor changes the state of i_k to be s_k and adds it to list I_j.

   (*N.B.* The item i' referred to above may not still be in state s_k when its return list is modified, since predictions occurring between this one and the one that added i' may have modified the state of i'. In such a case, the predictor still reuses i' rather than create a new item.)

Case (ii) The item i has no calls pending for n.

Let N be the type of n, let $R_1^n, R_2^n, \ldots, R_m^n$ be all the recognizers for grammar rules which derive N, and let $s_1^n, s_2^n, \ldots, s_m^n$ be the initial states of those recognizers computed as per section 4.1.5. For each s_k^n, if the predictor has already added a new item i' with state part s_k^n to list I_j, the predictor adds i' to the call list for n in i and adds i to the return set of i'. Otherwise, the predictor creates an item with state part s_k^n, empty call list, and return set $\{i\}$; if this item is for a left-recursive rule the predictor marks it with the R-flag. The predictor then adds this item to

I_j and to the call list for n in i.
(*N.B.* The point made above about possible changes in the state of the items i' also applies in this case.)

The Completer

The completer operation takes as input an item i on list I_j. Let s be the state part of i. If none of the target edges in s are trailing edges, the completer operation does nothing. Otherwise, let $i_1, \ldots, i_m$ be the live members of i's return set. Let n_k be the non-terminal in i_k whose call list contains i, and let c_k be that call list. Finally, let s_k be the state of i_k induced by i's return as per section 4.1.5. The completer performs the following actions for each i_k:

1. If i_k is marked with the R-flag, the completer invokes the r-split operation on i_k, n, i, and I_j.
2. If i_k has calls other than the one to i pending for node n, the completer invokes the c-split operation on i_k, n, i, and I_j.
3. If the completer has already added to I_k a c-split i' of i_k whose state part is s_k and who has calls outstanding for the same nodes as i_k does,[2] the completer marks i_k as dead (from a merge) and adds it to I_j. Otherwise, the completer changes the state of i_k to be s_k and adds it to I_j.
(*N.B.* As in the predictor, the item i' mentioned above may not still be in state s_k at the time of the current completion. Intervening completions may have changed i''s state, but the completer still marks i_k as dead.)

The Algorithm

First, we construct I_0 as follows:

1. For each rule P_k in the grammar which derives S, let s_k be the initial state of a recognizer for that rule computed in accordance with section 4.1.5 and the initial linkage information specified in the input. Add an item

[2] This restriction on calls merely reflects the fact that, unlike the state of string items, the state of graph items does not reflect which non-terminals are being derived. Thus, when considering whether to merge graph items, this part of their 'state' must be checked separately. Note that only which nodes have outstanding calls matters here, not the items to which those calls were made.

to I_0 whose state part is s_k, whose call list is empty, and whose return set is empty.

2. Run the predictor on every item in I_0. If this adds new items to I_0, run the predictor on them, and repeat this process until no new items are added.

Next, we successively construct $I_1, \ldots, I_n$. Given $I_0, \ldots, I_{j-1}$, we construct I_j as follows:

3. Choose an input node n_j to read next. (This node must be in the right fringe of the current read head.)
4. Run the scanner over every item in I_{j-1}.
5. Run the completer over every item in I_j. If this adds new items to I_j, run the completer over them, and repeat this until no new items are added.
6. Run the predictor over every item in I_j. If this adds new items to I_j, run the predictor over them, and repeat this until no new items are added.

A little thought will convince the reader that this algorithm produces the lists given in the examples above. A graph is accepted by this algorithm if I_n contains an item whose call lists and return set are empty and whose state part is an accepting state of a recognizer for a rule deriving S.

This algorithm can be converted to produce a parse using techniques exactly like those presented in the last chapter.

Chapter 5.

Discussion

In this chapter, we discuss a variety of issues related to flow graphs, flow grammars, and our parsing algorithm. We include a complexity analysis of the algorithm and some suggestions for related research.

5.1. Flow Graphs and Grammars

As was mentioned in the introduction, flow graphs were abstracted from program descriptions called *plans*. The goal of the abstraction was to preserve two structural features of plans: (i) the partial ordering of operations, and (ii) the naming of inputs and outputs. There were a number of structural features of plans that were left out of flow graphs, most notably "fan out"—the use of an operation's single output as the input of more than one other operation. The criteria used to determine which features of plans would be preserved in flow graphs grew out of the author's work in program analysis and are not germane here; however, we say more below about how graphic representations which include other features may be usefully manipulated by our parsing algorithm.

It should be quite clear from the above that the structure of flow graphs and flow grammars were developed without much regard for graph-theoretic concerns. This does not necessarily mean, however, that they are devoid of theoretical interest. If we view flow grammars as generalizations of string grammars which generate partially- as opposed to totally-ordered sentences, then the following sorts of questions naturally arise:

- Is there a natural definition of a "finite-state flow-graph automaton"? Is it possible to develop a hierarchy of such automata analogous to the

string-automata hierarchy? What is the relationship between this hierarchy and the string hierarchy?

- Is is possible to develop a hierarchy of flow grammars analogous to Chomsky's type 0–3 string-grammar hierarchy? In particular, are there canonical-form results for flow grammars, and do the sets of languages generated by such grammars satisfy any interesting closure properties?
- If the answers to the above questions are affirmative, can we relate the automaton and grammar hierarchies in a manner reminiscent of the string case?

Our research into flow-graph parsing, as might be surmised from the references mentioned in chapter 1, was initially concerned with questions such as these. We hoped initially that flow-graph parsing might be reducible to context-free string parsing. While we eventually gave up on achieving such a direct connection between the two—we now believe that flow grammars have a strictly greater generative capacity than string grammars—the intuitions we developed in the course of this research seem to indicated strongly that the answers to the questions given above are generally affirmative.

5.2. Applicability of the Algorithm

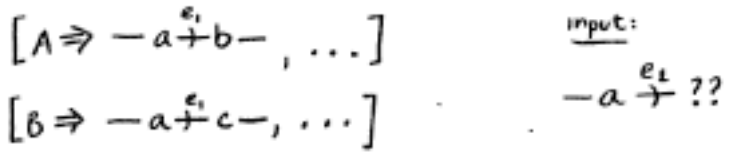
One of the most interesting features of our parsing algorithm is its amenability to 'advice' from outside sources of knowledge. Since the algorithm's control mechanism works by consulting and updating and explicit agenda—the current item list—it is a relatively easy matter for external agents to control or influence the algorithm's behavior: they need only make alterations in this list.

For example, let us consider the fan-out problem mentioned above. Figure 5.1 shows a grammar fragment and a pseudo-flow graph which contains fan-out. This figure represents the result of a typical program optimization and may be read as follows:

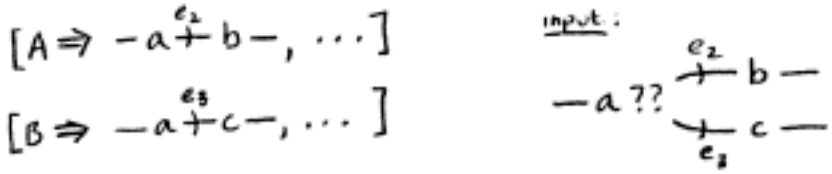
> *A* and *B* are high-level operations. *A* may be implemented by operation *a* followed by operation *b*, while *B* may be implemented by operation *a* followed by operation *c*. If I have a program which performs both *A* and *B*, I can optimize this program by performing *a* just once and then using its output as the input of both *b* and *c*.

Figure 5.1. Flow grammar fragment and pseudo-flow graph. The graph displays fan-out and can be considered an optimization of the two flow graphs generated by the grammar fragment.

We cannot recover this analysis simply by parsing, because the flow graph which represents the optimized program can not be read all the way through by our read-head mechanism. But consider the state of the parser at the point where its read head encounters the fan-out. The grammar fragment shown in figure 5.1 will have given rise to the following two item skeletons:



A fan-out handler invoked at this point, basing its actions on a theory of program optimization through shared operations, might (i) replace e_1 in the parser's head position by e_2 and e_3, and (ii) change these item fragments to read:



These actions would allow the parser, using the rules of figure 5.1, to recover both the desired analyses for the input.

Readers may be dismayed by the seeming informality of such a solution.[1] The point here is that our parsing algorithm embeds a notion of input structure which does not take into account anomalies due to sharing. A domain expert which understands such structural anomalies can show the parser how to proceed in these cases by altering its state in the aforementioned way; the operations themselves may seem low-level but the theory underlying them is not.

To put it another way, it is important here not to confuse "representation-level" with "low-level". Because the algorithm's representations fairly directly reflect the state of the parsers being simulated, a wide range of grammar- and graph-theoretic operations can be implemented directly via simple representation-level operations. In fact, it is precisely this closeness between the theoretic and representational levels that makes this algorithm so easy to advise.

5.3. Correctness

We have done no substantive work on a correctness proof for our parsing algorithm. For one thing, such a proof would ideally require definitions of flow graphs, flow grammars, and the graph-derivation process which are a great deal more rigorous than those presented here. For another, the algorithm itself would have to be stated quite a bit more precisely than we have stated it. To readers who might be interested in constructing a correctness proof, we recommend the proof of correctness of Earley's algorithm contained in [Aho and Ullman 1972]. The structure of that proof would probably serve as a good model for a correctness proof in this case.

5.4. Complexity Analysis

We conclude by considering the running time of our simulation algorithm. In our analysis, we will be concerned entirely with obtaining a loose upper bound on worst-case time complexity as a function of the number of nodes and edges in the input graph, paying only cursory attention to space complexity or time/space complexity as a function of the input grammar. The

[1] In fact, we expect that readers are screaming "Ouch! What a hack!"

point of this focus is twofold: (i) we are interested primarily in exposing intuitions about the relative cost of the algorithm's operations, and (ii) we will be satisfied to show that the algorithm displays polynomial rather than exponential time growth with the size of the input.

The algorithm spends all its time, except for a constant amount at the beginning, running the scanner, predictor, and completer over each constructed item. Thus, the total running time of the algorithm is the product of the number of constructed items and the sum of the times needed to run each of these three operations on an item. We will consider each factor separately.[2]

In what follows, we will be using the following definitions:

p = the number of rules in the grammar,
w = the maximum number of inputs to a grammar rule,
t = the maximum number of pairs in the state of a recognizer,
h = the maximum number of edges in a head position,
e = the number of edges in the input graph, and
n = the number of nodes in the input graph.

In addition, we will often be making an ordered choice of k items from among m items. There are

$$m \times m-1 \times \ldots \times m-(k-1) = k! \times \binom{m}{k}$$

ways of making such a choice; we will denote this number by $\left[\begin{smallmatrix} k \\ m \end{smallmatrix}\right]$.

The algorithm constructs $n+1$ lists. On each list, any two items must differ in one of (i) the rule they were invoked for, (ii) where in the input they were invoked, or (iii) their state. Each start position is an ordered choice of at most w edges from the e in the input, and each state is determined by an ordered choice of at most t edges from the current head position; thus, the number of items a single list is bounded above by

$$p \times \begin{bmatrix} w \\ e \end{bmatrix} \times \begin{bmatrix} t \\ h \end{bmatrix}$$

[2]This analysis is patterned directly after that given by Earley in [1969].

Both the predictor and the completer need to know whether they have already added a particular item to the current list. If we are concerned only with running time (and not space requirements), we can optimize this operation to take constant time by keeping a table of all such items which is indexed by the the three factors mentioned above.[3] We will assume that such an optimization is used in the following analysis.

Now we wish to bound the time it takes to run each of the three basic operations on a given item i from list I_j. In this analysis, we will use an augmented predictor operation that also attaches to each created item a unique integer identifier. When the c-split operation splits such an item, this identifier will be copied to the split item, allowing the predictor to tell in constant time whether two items are c-splits of each other.

The scanner either copies i or simulates a state transition for the recognizer represented by i. This takes constant time. In addition, the scanner has to check each of up to t pairs in the item's state part to determine whether the item is active on the node read. This also requires time independent of the size of the input graph.

When the predictor considers an item i on I_j whose state contains inputs to target non-terminal(s), it tries to add to I_j up to tp items: one for each rule which derives each of the non-terminals whose inputs have been reached. In addition, it may p-split the up to $\left[\frac{t-1}{phw}\right]$ members of its call list: a figure derived below. Given the optimization described above, checking whether each of the resulting items has already been added to the list takes constant time, so the predictor takes up to $\left[\frac{t-1}{phw}\right]$ time on each item.

When the completer considers an item i on I_j whose state part contains trailing edges of its target graph, it must process every item in i's return set. Thus, the amount of time taken by the completer is the product of the maximum number of items in a return set and the amount of time it takes to process each such item.

Return pointers in an item arise from two sources. Originally, an item has one return pointer for each of its calling items. However, as the parser runs and its callers split, an item may contain several return pointers to different splits of one original caller. We consider first how many original callers an item can have, and then how many splits each can turn into.

---

[3] Such a table, of course, will tend to be very sparse; the author's actual implementation of the algorithm used a more compact, time-consuming representation.

A given sub-recognizer R can only have been originally called from recognizers at one of w head positions (one for each of its inputs). This, its original calling recognizers must have been in one of at most $\begin{bmatrix} t \\ wh \end{bmatrix}$ states, so there can have been at most

$$p \times \begin{bmatrix} t \\ wh \end{bmatrix} \times \begin{bmatrix} w \\ e \end{bmatrix}$$

original calling items.

Suppose an instance of a recognizer r (which was called at some specific input position) calls an item i to derive a node n. Every state transition on a node other than n that the calling recognizer takes while i is running might split the caller and add up to p return pointers to i. Since at least one input of n can not appear in the state led to by such a transition, there are at most $\begin{bmatrix} t-1 \\ e \end{bmatrix}$ possible states which contribute splits, leading to a total return set membership of

$$\left(p \times \begin{bmatrix} t \\ wh \end{bmatrix} \times \begin{bmatrix} w \\ e \end{bmatrix}\right) \times \left(1 + \begin{bmatrix} t-1 \\ e \end{bmatrix}\right)$$

When a calling item i c- or r-splits as the result of a callee's return, we must add the split-off item to the call lists of i's callers and the return lists of i's callees. This will take as much time as there are callers and callees. We saw above that how may callers there are; a similar argument (utilizing the symmetry of creation of call and return pointers) shows that there are up to $\begin{bmatrix} t-1 \\ phw \end{bmatrix}$ callees. Thus, splitting an item may take up to the sum of these figures, leading to a total cost for the completer of

$$\left[\left(p \times \begin{bmatrix} t \\ wh \end{bmatrix} \times \begin{bmatrix} w \\ e \end{bmatrix}\right) \times \left(1 + \begin{bmatrix} t-1 \\ e \end{bmatrix}\right)\right] \times \left[\left(p \times \begin{bmatrix} t \\ wh \end{bmatrix} \times \begin{bmatrix} w \\ e \end{bmatrix}\right) \times \left(1 + \begin{bmatrix} t-1 \\ e \end{bmatrix}\right) + \begin{bmatrix} t-1 \\ e \end{bmatrix}\right]$$

When we add the costs of the three operations together, we find that the completer dominates the cost of the other operations, so the cost of the entire algorithm is bounded by the product of the completer cost and the total number of items. This product is polynomial in e; in the string case it reduces to $e^3 = n^3$ which is the cost of Earley's algorithm.

References

Aho, Alfred V., and Jeffrey D. Ullman, *The Theory of Parsing, Translation, and Compiling*, volume 1, Prentice Hall, 1972.

Brotsky, Daniel C., "Program Understanding through Cliché Recognition," (M.S. Proposal), MIT-AI-WP-224, December 1981.

Earley, J., "An Efficient Context-Free Parsing Algorithm," (Ph.D. Thesis), Computer Science Department, Carnegie Mellon University, 1968.

Knuth, Donald E., "On the Translation of Languages from Left to Right," *Information and Control* 8:12, December 1965, pp. 607–639.

Montanari, U.G., "Separable Graphs, Planar Graphs, and Web Grammars," *Information and Control* 16:3, March 1970, pp. 243–267.

Pavlidis, T., "Linear and Context-Free Graph Grammars," *Journal of the ACM* 19:1, January 1972, pp. 11–23.

Pfaltz, J.L., and A. Rosenfeld, "Web Grammars," *Proceedings of the IJCAI*, Washington, D.C., 1969, pp. 609–619.

Rich, Charles, "Inspection Methods in Programming," (Ph.D. Thesis), MIT-AI-TR-604, December 1980.

Rich, Charles, and Richard C. Waters, "Abstraction, Inspection, and Debugging in Programming," MIT-AIM-634, June 1981.