Global Data Flow Analysis Problems Arising in Locally Least-Cost Error Recovery

ROLAND BACKHOUSE
Heriot-Watt University

Locally least-cost error recovery is a technique for recovering from syntax errors by editing the input string at the point of error detection. A scheme for its implementation in recursive descent parsers, which in principle embodies a process of passing a parameter to *each* procedure in the parser for *each* terminal symbol in the grammar, has been suggested. For this scheme to be practical it is vital that as much parameterization as possible is eliminated from the recursive descent parser. This optimization problem and how it may be split into three separate global data flow analysis problems—classifying terminal symbols and the so-called min and max follow cost problems—are discussed. The max follow cost problem is a particularly difficult one to solve. The application of Gaussian elimination to its solution is shown by expressing it as a continuous data flow problem, and it is also related to an "idiosyncratic" data flow problem arising in the optimization of very high level languages. Classifying terminal symbols is also difficult since the problem is unsolvable in general. However, for the class of LL(1) grammars, the problem is shown to be expressible as a distributive data flow problem and so may be solved using, say, Gauss–Seidel iteration.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*optimization*; *translator writing systems and compiler generators*; G.2.2 [**Mathematics of Computation**]: Graph Theory—*path and circuit problems*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Code optimization, compiling, error correction, error recovery, error repair, Gaussian elimination, Gauss–Seidel iteration, global flow analysis, lattice, LL grammar, parser generator, path problem, regular algebra, shortest path

1. INTRODUCTION

One of the principal benefits of the introduction of Backus Normal Form (BNF) in the Algol 60 report was to facilitate the development of parser generators. Such tools, which assist greatly the compiler writer's task in the initial stages, are now finding widespread use. Many parser generators, however, provide little assistance with the important requirement of incorporating efficient and effective error recovery into the parser, although this situation is changing (e.g., [11, 18–

This work was supported by a grant from the Science and Engineering Research Council of Great Britain and was completed when the author was a member of the Department of Computer Science, Heriot-Watt University.
Author's Address: Department of Computer Science, University of Essex, Wivenhoe Park, Colchester CO4 3SQ, U.K.

20]). One technique for the automatic inclusion of error recovery, called "locally least-cost error recovery," has been investigated [2–4]. The basis for this technique is to recover from syntax errors by editing the next input symbol at the point of error detection to a syntactically correct string. A table-driven LL(1) parser generator that exploits a similar idea has been developed by Fischer et al. [11], and Backhouse [5] discusses its implementation in recursive descent parsers.

The implementation of locally least-cost repairs, as described by the author in [5], includes a mechanism for parameter passing that generalizes the follow set parameters in the recovery scheme advocated by Wirth [24]. Essentially, the idea is to anticipate the possibility of having to repair any terminal symbol by computing, *before* the call of a recursive descent procedure, the cost of editing the symbol *after* the call of the procedure has been completed. The potential advantage of this mechanism over the searching strategy suggested by Fischer et al. [11] is that locally least-cost repairs can be evaluated using a simple table look-up rather than, effectively, solving a shortest path problem when the error is detected. Unfortunately, a worst-case analysis of the size of the tables required to implement the parameter passing is quite off-putting. Typically, a grammar defining a programming language will have between 50 and 100 terminal symbols and 300 to 400 production positions; the size of the parameter table is the product of these two quantities! Fortunately, an analysis of the values taken by the parameters in practice reveals that the vast majority of them are unnecessary and so can be eliminated. In addition, the terminal symbols can be partitioned into a number of equivalence classes such that the parameter values for symbols in the same class are always equal. Thus the parameter table can be considerably reduced by storing only the values for a representative element of each class. The subject of this paper is the techniques whereby we are able to classify terminal symbols in a grammar and eliminate parameters from the parameter table.

The problems we tackle amount to a global data flow analysis of the recursive descent parsers that would be generated were the scheme described in [5] rigidly adhered to. In fact, determining whether a parameter is strictly necessary splits into two distinct problems, the "min follow cost problem" and the "max follow cost problem." The min follow cost problem is a shortest path problem and is not so interesting; on the other hand, the max follow cost problem is challenging because it can be related to another global data flow problem, which others have found difficult [12, 22]. It is a problem that can be solved using a technique like Gaussian elimination [6, 8, 22] but cannot be solved using an iterative technique like the Gauss–Seidel method [8]. Classifying terminal symbols is an interesting problem for two reasons: First, its usefulness is not restricted to locally least-cost error recovery—it could be applied to improving the space requirements of other recovery schemes, for example, the follow set scheme [24]; second, it is not an easy problem since it is unsolvable in general but, as we show, it can be solved for LL(1) grammars.

Throughout this paper we assume that the reader is familiar with the notation of context-free languages as used in, for example, [1]. In the definitions that follow we refer, without explicit mention, to a context-free grammar $G = (N, T, P, Z)$ with nonterminal set N, terminal set T, production set P, and start symbol Z. We assume that Z does not appear on the right-hand side of any production.

Our results are relevant to the case in which G is LL(1) (in which case a recursive descent parser for G exists) although none of the definitions or lemmas require this fact.

The structure of the remainder of this paper is as follows. Section 2 comprises a number of definitions and lemmas leading up to the definition of a locally optimal repair, the basis of locally least-cost error recovery, and then briefly summarizes its implementation in a recursive descent parser. Section 3 poses and solves the problem of classifying terminal symbols, and Section 4 does likewise for the min and max follow cost problems. These two sections may be read independently, except for Section 3.4, which defines the notion of an open portion graph, which is central to the solution of both problems. Section 5 describes the results we have obtained on effectiveness of these optimizations and Section 6 concludes the paper.

2. BASIC PRINCIPLES OF LOCALLY LEAST-COST ERROR RECOVERY

2.1. Cost Functions and Locally Optimal Repairs

This section contains a summary of the definitions and properties we use in the remainder of the paper. For further motivation and comparison with extant work see [2] and [5]. Note that solely for brevity we have adopted a notation that is different from [5], as well as having made a number of technical changes in our presentation.

Definition 1 (Primitive Edit Costs). Let Λ denote the empty word and $u \rightarrow v$ denote the edit operation of replacing the string u by the string v. Then $\text{cost}(\Lambda \rightarrow t)$, $\text{cost}(t \rightarrow \Lambda)$ and $\text{cost}(t \rightarrow t')$ denote, respectively, the cost of inserting t, the cost of deleting t, and the cost of changing t to t', where $t, t' \in T$. These costs are primitive and assumed given. All insertion costs are strictly positive integers or ∞. All change costs are strictly positive integers, ω, or ∞, except $\text{cost}(t \rightarrow t)$, which is zero for all $t \in T$. An edit cost of ω (unbounded) means that the operation is permissible, but only if it cannot be circumvented by an edit operation or sequence of edit operations of finite cost. An edit cost of ∞ (infinity) means that the edit operation is not permissible. The properties we assume of ω and ∞ are as follows:

$$\left.\begin{array}{l} -\infty < c < \omega < \infty \\ c + \omega = \omega + \omega = \omega \\ c + \infty = \omega + \infty = \infty + \infty = \infty \\ \omega - \infty = -\infty \end{array}\right\} \quad \text{for all (finite) integers } c.$$

Definition 2 (Composite Edit Costs). Let $v = a_1 a_2 \cdots a_n \in T^*$ where $a_i \in T (1 \leq i \leq n)$. Let $t \in T$. Then the cost of inserting v, denoted $I(v)$, and the cost of editing t to a prefix of v, denoted $P(t, v)$, are defined as follows:

$$\begin{aligned} I(v) &= 0 && \text{if } v = \Lambda \text{ (i.e., } n = 0) \\ &= \sum_{i=1}^{n} \text{cost}(\Lambda \rightarrow a_i) && \text{otherwise,} \\ P(t, v) &= \infty && \text{if } v = \Lambda \\ &= \min_{1 \leq i \leq n} \{I(a_1 \cdots a_{i-1}) + \text{cost}(t \rightarrow a_i)\} && \text{otherwise.} \end{aligned}$$

Definition 3 (Edit Costs for Languages). We extend the insert cost function I and the prefix cost function P to languages as follows. Let $t \in T$ and $L \subseteq (N \cup T)^*$. Then $I(L)$ and $P(t, L)$ are defined by

$$I(L) = \min_{\alpha \in L} \left\{ \min_{v \in L_G(\alpha)} \{I(v)\} \right\} \quad \text{and} \quad P(t, L) = \min_{\alpha \in L} \left\{ \min_{v \in L_G(\alpha)} \{P(t, v)\} \right\}$$

where

$$L_G(\alpha) = \{v \mid v \in T^* \text{ and } \alpha \Rightarrow^* v\}.$$

LEMMA 1. $P(t, L_1 \cdot L_2) = min\{P(t, L_1), I(L_1) + P(t, L_2)\}$.

PROOF. Straightforward. □

Definition 4 (Continuations). The string $u \in T^*$ is said to be a *valid prefix* (of Z) if and only if $\exists v \in T^*$ such that $Z \Rightarrow^* uv$. The set of (*valid*) *continuations of* u, denoted $C(u)$, is defined by

$$C(u) = \{v \mid v \in T^* \text{ and } uv \text{ is a valid prefix}\}.$$

Definition 5 (Locally Optimal Repairs). Let u be a valid prefix and $t \in T$. A *locally optimal repair of* t *following* u is defined to be $t \rightarrow w$ where $w \in T^*$ is such that

$$w \in C(u),$$

and either

(a) $w = \Lambda$, $\text{cost}(t \rightarrow \Lambda) \leq P(t, w')$ for all $w' \in C(u)$ and $\text{cost}(t \rightarrow \Lambda) < \infty$, or
(b) $w \neq \Lambda$, $P(t, w) \leq P(t, w')$ for all $w' \in C(u)$ and $P(t, w) < \infty$.

If no such w exists, then a locally optimal repair of t following u is undefined and the syntax analysis must be aborted.

2.2 Parsing LL(1) Grammars

Incorporating locally least-cost error recovery into LL(1) parsers is straighforward. The feature of LL(1) grammars that makes this so is that each valid prefix u defines a *unique* leftmost derivation sequence. More specifically, it is possible to identify a unique *open portion* $\gamma \in (N \cup T)^*$ and a unique production position $A \rightarrow \alpha \cdot \beta$ (consisting of a production $A \rightarrow \alpha\beta$ and a marker dot) such that

(1) $Z \Rightarrow_l^* vA\gamma \Rightarrow_l v\alpha\beta\gamma \Rightarrow_l^* vw\beta\gamma = u\beta\gamma$ for some $v, w \in T^*$;
(2) $Z \Rightarrow^* ux$ implies $x \in L_G(\beta\gamma) = \{x \mid \beta\gamma \Rightarrow^* x\}$.

Now, given that we can identify these two quantities, we can reexpress the cost of a locally optimal repair of t following u, using Lemma 1, as

$$\min\{\text{cost}(t \rightarrow \Lambda), P(t, C(u))\} = \min\{\min\{\text{cost}(t \rightarrow \Lambda), P(t, \beta)\}, I(\beta) + P(t, \gamma)\}.$$

The latter equation is important because it makes the choice of a locally optimal repair of t following u a two-way decision—either we choose to repair t to a prefix of β (which choice is represented by the term $\min\{\text{cost}(t \rightarrow \Lambda), P(t, \beta)\}$), or we choose to insert β and then repair t to a prefix of the open portion γ (which choice is represented by the term $I(\beta) + P(t, \gamma)$). The associated recovery actions are called, respectively, the *nonreturn action* and the *return action* because in a

recursive descent parser the effect of making the repair is, respectively, not to return and to return from the procedure which recognizes A.

To make the choice between the nonreturn and return actions, it suffices to compute and compare the following quantities:

(1) $B(A \rightarrow \alpha \cdot \beta, t) = P(t, \beta) - I(\beta)$,
(2) $P(t, \gamma)$.

The first of these quantities is called the *boundary cost* associated with t at the position $A \rightarrow \alpha \cdot \beta$. Note that it is independent of u (it depends only on t and the production position) and so can be precomputed and stored as part of the driving tables of the parser. The second quantity could depend on u and is called the *parameter passed to A corresponding to t.*

Computing parameters to be passed to each recursive descent procedure is one of the more significant differences between our scheme and that suggested by Fischer et al. [11]. It makes no difference to the effectiveness or appropriateness of the repairs; its advantage is that it is asymptotically faster since it guarantees linear-time parsing for deterministic context-free grammars [2]. The mechanism for evaluating the parameters is quite simple. Initially, on the very first call to the procedure recognizing Z, the parameter passed is ∞ for each $t \in T$. Subsequently, suppose the production $A \rightarrow \alpha B \beta$ is being recognized and the procedure for A calls the procedure for B. Suppose At is the parameter passed to A corresponding to t at this juncture. Then the parameter Bt passed to B is

$$Bt = \min\{P(t, \beta), I(\beta) + At\}.$$

Once again we remark that the quantities

(3) $P(t, \beta)$, and
(4) $I(\beta)$

are independent of u and so can be included in the parser tables.

Although locally least-cost error recovery is conceptually elegant, it would appear at first sight to be hopelessly inefficient. Consider, for instance, the programming language Pascal. A typical grammar for Pascal will have about 50–60 terminal symbols and about 400 production positions. Now the quantities (1) and (3), which we blithely said should be included in the parser tables, are defined for each combination of terminal symbol and production position. Thus there are between 20,000 and 24,000 of them! If this were truly necessary, it would not only create a huge storage problem but also slow the parsing process substantially, since 50–60 parameters have to be evaluated and passed every time a procedure is called. Fortunately, by exploiting the computations we are about to discuss, it is possible to trim the entries to about 1000. (The actual size depends significantly on the primitive costs; the figure we quote is drawn from our experience. We have, in fact, generated parsers for Pascal requiring as few as 300 and as many as 2000 entries.)

3. CLASSIFYING TERMINAL SYMBOLS

3.1 Problem Statement and Motivation

The first step toward reducing the size of the parser tables is to classify the terminal symbols. More specifically, given a context-free grammar $G = (N, T, P, Z)$ we wish to find the *equivalence classes* of T, where $t, t' \in T$ are *equivalent* if

and only if the following condition holds:

(C1) $(\forall u \in T^*)((\exists v \in T^*)(Z \Rightarrow^* utv)$ iff $(\exists v' \in T^*)(Z \Rightarrow^* ut'v'))$.

More intuitively, t and t' are equivalent if and only if, in a left-to-right parse of G, whenever t is a valid next symbol, then so is t' also, and vice versa.

Typical equivalence classes of terminal symbols in the programming language Pascal are

{**to, downto**}
{<, >, <=, >=, =}
{**if, goto, while, repeat, for, with**}
{*, /, **div, mod, or, and**}
{**begin**}

Another way of expressing condition (C1), which will help to motivate the problem, is

(C2) $$(\forall u \in T^*)(\{v \mid Z \Rightarrow^* uvtw \text{ for some } w \in T^*\} = \{v \mid Z \Rightarrow^* uvt'w \text{ for some } w \in T^*\}).$$

Condition (C2) can be paraphrased in the following way. Suppose in a left-to-right parse of sentences in G the string u has been successfully parsed. Consider the possibility that the next symbol is t or t'. Then t and t' are equivalent with respect to u if and only if the set of strings v that must prefix t before t may be successfully parsed is equal to the set of strings that must prefix t' before t' may be successfully parsed. The symbols t and t' are equivalent if they are equivalent with respect to all strings u.

A straightforward consequence of (C2), which is the ultimate motivation for our definition of equivalence, is that if t and t' are equivalent and the deletion costs and change costs associated with t and t' are equal, then, for all valid prefixes u, a locally optimal repair of t following u is also a locally optimal repair of t' following u, and vice versa. Indeed, if these conditions are met, then the parameter values and boundary costs associated with t and t' will always be equal in any state of the parse. So instead of computing these values for every terminal symbol in the grammar, we only need to do so for a representative element of each equivalence class (or, more precisely, for each equivalence class after the partition defined by (C1) has been refined according to the given edit costs). For Pascal the grammar we used had 63 terminal symbols, which were classified into 38 classes, thus giving an immediate reduction of 40 percent in the size of the tables.

Note that the definition of equivalence contains no elements specifically related to locally least-cost error recovery. Indeed, the concept may be exploited to improve the efficiency of other recovery techniques. For example, in follow set error recovery [24] under almost all reasonable methods of computing follow sets, if t and t' are equivalent, they will be both in or both out of any follow set passed as a parameter to a procedure. Thus instead of passing sets of symbols, it would suffice to pass sets of classes as parameters to the procedures. However, in this case the resulting saving of space is not likely to be worthwhile.

3.2 Difficulty of the Problem

The ease with which one can detect equivalent symbols in a grammar for, say, Pascal is very variable. It is trivial to see for example that "**to**" and "**downto**" are equivalent because their only appearance is in an extended BNF production of the form

$$\text{for\_statement} = \ldots (\text{"}\textbf{to}\text{"} \mid \text{"}\textbf{downto}\text{"}) \ldots$$

Less easy to see is that "**if**", "**goto**", "**while**", "**repeat**", "**for**", and "**with**" are all equivalent, since their only appearance is at the beginning of statements, but they are not equivalent to "**case**" (which can appear in a **record**) or "**begin**". Much harder to see from the grammar is that "*", "**div**", "**mod**", "**and**", and "**or**" are all equivalent yet not equivalent to "+" or "−".

We may appreciate more fully the difficulties involved by examining a specific example. Consider the grammar $G = (\{E, F, R, T, U\}, \{a, (,), +, *\}, P, E)$ whose production set P is

$$\begin{array}{ll} E \to TR & \\ R \to \Lambda & R \to +ER \\ T \to FU & \\ U \to \Lambda & U \to *TU \\ F \to a & F \to (E) \end{array}$$

This is a well-known LL(1) grammar defining arithmetic expressions involving "+", and "*", the identifier "a", and parentheses "(" and ")". Now if we think about the *language* of arithmetic expressions, it is clear that the equivalence classes of terminal symbols are $\{)\}$, $\{a, (\}$, $\{+, *\}$. Thus wherever "a" may appear, we may also have "(", and vice versa. Also, in any arithmetic expression all instances of "+" may be changed to "*", and vice versa. It is also clear from the *grammar* that "a" and "(" are equivalent, since their only appearance is as the first right-hand-side symbol of productions with the same left-hand side. But it is certainly not clear from the *grammar* that "+" and "*" are equivalent, since they appear on the right-hand side of quite distinct productions. We shall return to this example later to illustrate our algorithm.

Further evidence for the claim that the problem is by no means trivial is given by the following theorem, which states that it is undecidable for general context-free grammars!

THEOREM 2. *Given a context-free grammar $G = (N, T, P, Z)$ it is in general undecidable whether t is equivalent to t' for given $t, t' \in T$.*

PROOF. It is well known that the problem of deciding whether two context-free grammars generate the same language is unsolvable [14]. So take any two grammars $G_1 = (N_1, T, P_1, S_1)$, $G_2 = (N_2, T, P_2, S_2)$ defined over the same alphabet T but with distinct nonterminal alphabets N_1 and N_2. Let t, t', S be distinct symbols not in $N_1 \cup N_2 \cup T$. Let $G = (N_1 \cup N_2 \cup \{S\}, T \cup \{t, t'\}, P_1 \cup P_2 \cup \{S \to S_1 t, S \to S_2 t'\}, S)$. Then t is equivalent to t' in G if and only if the languages generated by G_1 and G_2 are equal. □

Since the problem is undecidable in general, we shall restrict ourselves to the class of LL(1) grammars, the class for which we have an immediate application.

We shall show that in this case the problem is decidable and that our algorithm can be applied "safely" to other grammars in the sense that it will always produce a refinement of the partition of T defined by our equivalence relation. However, our results do not extend to giving an exact solution for LR(1) grammars, and the decidability of the problem in this case remains open.

3.3 Solution Plan

The key to the solution of the classification problem for LL(1) grammars is the observation made in Section 2.2 that each valid prefix u defines a unique open portion, which we denote by γ_u. This allows us to rephrase the definition of equivalence as stated in Lemma 2. First, however, we need another definition.

Definition 6. Let $G = (N, T, P, Z)$ be a context-free grammar and let $\gamma \in (N \cup T)^*$. Then FIRST(γ) is defined to be $\{t \mid t \in T \text{ and } \gamma \Rightarrow^* tv \text{ for some } v \in T^*\}$.

LEMMA 3. *Suppose $G = (N, T, P, Z)$ is an LL(1) grammar. Then the symbols t and t' are equivalent if and only if for all valid prefixes u, either $\{t, t'\} \subseteq FIRST(\gamma_u)$ or $\{t, t'\} \cap FIRST(\gamma_u) = \emptyset$.*

PROOF. Obvious from property (2) of an open portion. □

The reason that Lemma 3 is important is that the set of all open portions forms a *regular* set. Moreover, FIRST(γ) is always finite (since it is a subset of T), so $\{\text{FIRST}(\gamma_u) \mid u \text{ is a valid prefix}\}$ is a *finite* set of subsets of T and its computation may be expressed as a *path problem* on a directed graph. Our plan of action is therefore as follows.

(1) Construct a graph $\mathscr{G}$ from the given grammar G, paths through which define the set of open portions of G.
(2) Identify the so-called FIRST_SETS problem. FIRST_SETS is a set of subsets of T, each element of which is FIRST(γ) for some open portion γ. Use the FIRST_SETS to define a partition on T which is the set of equivalence classes of T.
(3) Express the FIRST_SETS problem as a path problem on $\mathscr{G}$. More particularly, we shall express the problem as a distributive data flow problem [5, 8–10]. We shall also outline its solution as a continuous data flow problem [12] and explain why this alternative has not been adopted.

3.4 The Solution

3.4.1 *The Open-Portion Graph $\mathscr{G}$.* We shall begin by describing the construction of a graph representing the set of open portions of G. Since our definition of a distributive data flow problem differs slightly from other definitions we had better make clear what we mean by a graph.

Definition 7. A *labeled graph* $\mathscr{G} = (\mathscr{N}, \mathscr{A}, \mathscr{V}, s, f)$ consists of a set of *nodes* $\mathscr{N}$, a set of *arcs* $\mathscr{A}$, an *alphabet* $\mathscr{V}$, and distinguished *start* and *final* nodes s and f. Associated with each arc $a \in \mathscr{A}$ are three items—its *from component* from(a), its *into component* into(a), and its *label* $l(a)$. The from and into components are nodes, the label is an element of $\mathscr{V}^*$. A *path from node x to node y* in $\mathscr{G}$ is a sequence of arcs $a_1, a_2, \ldots, a_n$ where either $n = 0$ and $x = y$, or $n > 0$, from(a_1)

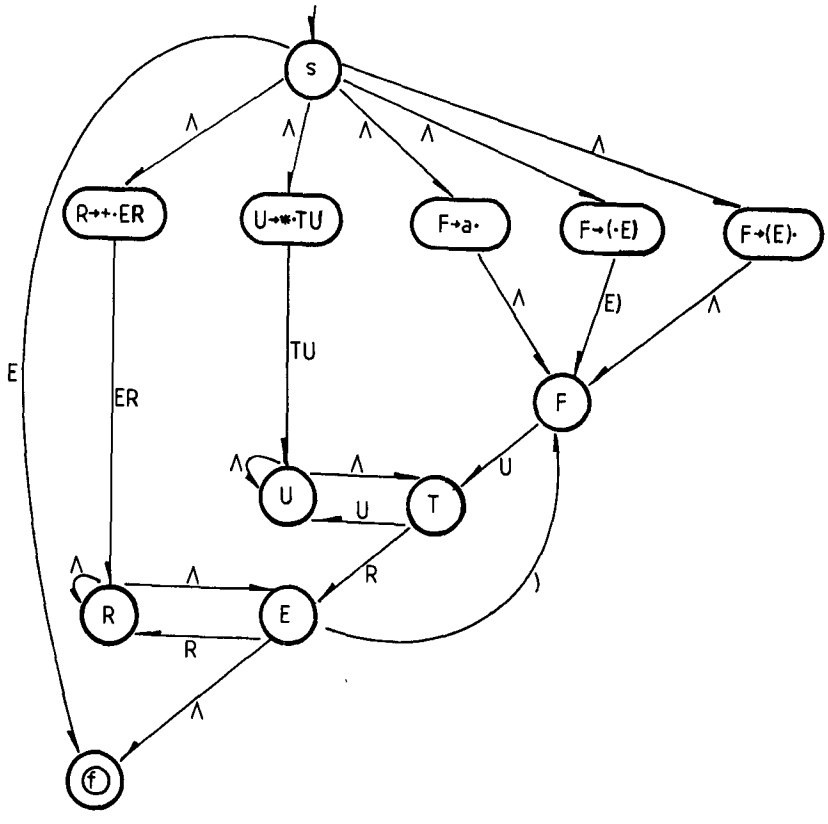

Figure 1

$= x$, into$(a_n) = y$, and into$(a_i) =$ from(a_{i+1}) for all i, $1 \leq i < n$. Such a path *spells* $\gamma \in (N \cup T)^*$ iff $\gamma = \Lambda$ (the empty word) and $n = 0$, or $n > 0$ and $\gamma = l(a_1)l(a_2) \cdots l(a_n)$.

Now suppose $G = (N, T, P, Z)$ is a context-free grammar. By a *production position* of G we mean a string of the form $A \rightarrow \alpha \cdot \beta$ where $A \rightarrow \alpha\beta$ is in P and the marker "$\cdot$" is any symbol not in $N \cup T$. From G we construct a labeled graph $\mathscr{G} = (\mathscr{N}, \mathscr{A}, \mathscr{V}, s, f)$ as follows. The nodes $\mathscr{N}$ are divided into three sets. First, there is a node corresponding to every nonterminal in G. Second, there is a node for every production position of the form $A \rightarrow \alpha t \cdot \beta$ where $t \in T$. Finally, there are two nodes s and f, the start and final nodes of $\mathscr{G}$, respectively. The alphabet $\mathscr{V}$ of $\mathscr{G}$ is $N \cup T$ (so the arc labels are elements of $(N \cup T)^*$) and the arcs are constructed as follows. There is an arc labeled Z from s to f and an arc labeled Λ from s to each node corresponding to a production position. There is an arc labeled Λ from the node Z to the node f and an arc labeled β from the node $A \rightarrow \alpha t \cdot \beta$ to the node A. Last of all there is an arc labeled β from A to B if there is a right-hand-side occurrence of A of the form $B \rightarrow \alpha A \beta$.

Figure 1 shows the graph constructed in this way for the grammar given in Section 3.2.

LEMMA 4. *There is a path spelling γ from s to f in the open portion graph of the LL(1) grammar G if and only if $\gamma = \gamma_u$ for some valid prefix u.*

PROOF. First, note that the path of arc length 1 spelling Z from s to f is the open portion of Λ. The remaining paths are the open portions of valid prefixes $u \in T^+$, which claim is proved as follows.

We begin by observing that there is a path spelling β from the node A to the final node f if and only if $Z \Rightarrow_l^* vA\gamma$ for some $\gamma \in (N \cup T)^*$. The "if" part of this claim involves a straightforward induction on the length of the leftmost derivation sequence and the "only if" part an induction on the arc length of the path. The basis for both inductions is that $Z \Rightarrow_l^* Z$, which explains the Λ arc from Z to f. The induction step relies on the fact that $Z \Rightarrow_l^* vA\gamma$, where $\gamma \neq \Lambda$, if and only if there is some right-hand-side occurrence of A of the form $B \rightarrow \alpha A\beta$ such that $Z \Rightarrow_l^* wB\delta \Rightarrow_l w\alpha A\beta\delta \Rightarrow_l^* vA\gamma$, for some w, δ, and $\gamma = \beta\delta$.

Now suppose $\gamma = \gamma_u$ for some $u \in T^+$. Let $u = vt$ where $t \in T$. Consider the leftmost derivation sequence $Z \Rightarrow_l^* u\gamma$. Since γ minimizes the length of this derivation sequence, we must have $Z \Rightarrow_l^* wA\delta \Rightarrow_l w\alpha t\beta\delta \Rightarrow_l^* vt\beta\delta = u\gamma$ for some production $A \rightarrow \alpha t\beta$ and some $w \in T^*$, $\delta \in (N \cup T)^*$. Thus $\gamma = \beta\delta$ where $A \rightarrow \alpha t \cdot \beta$ is a production position in G and δ spells a path from A to f in the graph $\mathscr{G}$. By the construction of the arcs to and from production positions, we may conclude that every open portion γ_u spells a path from s to f.

The converse proceeds similarly. Each path of arc length greater than 1 from s to f spells $\beta\delta$ for some β, δ where there is a production position $A \rightarrow \alpha t \cdot \beta$ and $Z \Rightarrow_l^* vA\delta$ for some $v \in T^*$. Pick any $w \in L_G(\alpha)$ and let $u = vwt$. Then $Z \Rightarrow_l^* vA\delta \Rightarrow_l v\alpha t\beta\delta \Rightarrow_l^* vwt\beta\delta = u\gamma$ is a minimal length leftmost derivation of $u\gamma$; that is, γ is the open portion of u. □

3.4.2 *The FIRST SETS.* Looking once again at Lemma 3 we see that our primary interest is in FIRST(γ) for each open portion γ. Since we now have a representation for the set of all open portions, we also have a representation for the set of all FIRST sets of open portions. More specifically, we have the following definition and lemmas.

Definition 8. Let $G = (N, T, P, Z)$ be an LL(1) grammar. Then FIRST_SETS(G) is a subset of the set of subsets of T defined by

$$x \in \text{FIRST\_SETS}(G) \quad \text{iff} \quad x = \text{FIRST}(\gamma) \text{ for some open portion } \gamma \text{ of } G.$$

LEMMA 5. *FIRST_SETS(G) = $\{x \mid x = FIRST(\gamma)$ where γ spells a path from s to f in $G\}$*

PROOF. Trivial from Lemma 4. □

LEMMA 6. *The terminal symbols t and t' are equivalent in G if and only if $\{t, t'\} \subseteq x$ or $\{t, t'\} \cap x = \varnothing$ for all $x \in FIRST\_SETS(G)$.*

PROOF. Immediate from the definition of FIRST_SETS and Lemma 3. □

In order to apply these lemmas by hand to our example grammar (see Section 3.2), we need to note that

$$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{a, (\},$$

$$\text{FIRST}(U) = \{*\},$$

$$\text{FIRST}(R) = \{+\},$$

and

$$\text{NULLABLE}(U), \text{NULLABLE}(R),$$

whereas **not** NULLABLE(E), **not** NULLABLE(T), and **not** NULLABLE(F), where NULLABLE(α) is true if and only if $\alpha \Rightarrow^* \Lambda$. Then by inspection of the graph in Figure 1 the reader may verify that FIRST_SETS for our example grammar is

$$\{\text{FIRST}(E), \text{FIRST}(T), \text{FIRST}(U) \cup \text{FIRST}(R), \\ \text{FIRST}(U) \cup \text{FIRST}(R) \cup \{)\}\}.$$

(Here we have observed that all paths spelling $E\ldots$ have the same FIRST set, all paths spelling $T\ldots$ have the same FIRST set, and so on.) Thus FIRST_SETS evaluates to

$$\{\{a, (\}, \{+, *\}, \{+, *,)\}\}$$

and the partition on T defined by it is

$$\{\{a, (\}, \{+, *\}, \{)\}\}.$$

Lemma 6 reduces our problem to finding FIRST_SETS(G), which, we note, is a *finite* set of *finite* sets. Lemma 5 is almost all the way to expressing the computation of the FIRST_SETS as the "meet over all paths solution" to a path-finding problem [15, 16, 22]. Completing this task is our next step.

3.4.3 *A Distributive Data Flow Problem.* One of the most significant outcomes of work on "optimizing" compilers has been the abstract, algebraic formulation of general circumstances in which a number of path-finding algorithms can be applied [5, 8–12]. Two approaches to the solution of a path problem have been identified. First, an iterative technique like the well-known Gauss–Seidel method may be applied if the problem can be shown to be a "distributive data flow problem" [8, 10]. Second, elimination techniques like the equally well-known Gaussian elimination method may be applied if the problem can be shown to be a "continuous data flow problem" [12]. The classification problem succumbs to both approaches, but we only consider the former in any detail. In contrast, Section 4 considers the so-called max follow cost problem, which is an example of a problem that cannot be solved by an iterative technique but can be solved by an elimination technique. To give precise meaning to our concept of a distributive data flow problem (which differs slightly from other formulations), we need a number of definitions

Definition 9. A *semilattice* is a pair $(\mathcal{S}, \wedge)$ where $\mathcal{S}$ is a set and $\wedge$ is an associative, commutative, and idempotent binary operation on $\mathcal{S}$. The set $\mathcal{S}$ is assumed to have a zero element 0 and a unit element 1 such that $0 \wedge a = 0$ and $1 \wedge a = a$ for all $a \in \mathcal{S}$.

Definition 10. *A distributive data flow framework* (L, F) consists of a semilattice $L = (\mathcal{S}, \wedge)$ and a set of functions $F: L \to L$ such that

(D1) Each $g \in F$ is distributive, that is,

$$g(x \wedge y) = g(x) \wedge g(y) \qquad \text{for all} \quad x, y \in \mathcal{S}.$$

(D2) There is an identity function i in F such that

$$i(x) = x \qquad \text{for all} \quad x \text{ in } \mathcal{S}.$$

(D3) F is closed under composition, that is, $g, h \in F$ implies $g \cdot h \in F$.

(D4) $\mathcal{S}$ is equal to the closure of $\{0\}$ under the meet operation and application of functions in F.

Definition 11. *A distributive data flow problem* is a triple $(\mathcal{G}, (L, F), \mathcal{f})$ where $\mathcal{G} = (\mathcal{N}, \mathcal{A}, \mathcal{V}, s, f)$ is a labeled graph, (L, F) is a distributive data flow framework, and $\mathcal{f}$ is a mapping from $\mathcal{V}^*$ to F satisfying the property

(D5) $$\mathcal{f}(\alpha) \cdot \mathcal{f}(\beta) = \mathcal{f}(\alpha\beta) \qquad \text{for all} \quad \alpha, \beta \in \mathcal{V}^*.$$

The *meet over all paths* (MOP) *solution* to this problem is *mop*(s) where the mapping *mop* from $\mathcal{N}$ to $\mathcal{S}$ is given by $mop(v) = \bigwedge \{\mathcal{f}(\gamma)(0) \mid \gamma \text{ spells a path from } v \text{ to } f \text{ in } \mathcal{G}\}$.

THEOREM 7. *Given a distributive data flow problem* $(G, (L, F), \mathcal{f})$ *in which the semilattice* L *is finite, the following algorithm will compute the MOP solution to the problem.*

Algorithm 1. Basic Iterative Algorithm

```
{Input: Labeled graph G = (N, A, V, s, f), semilattice L = (S, ∧), set of functions F,
        and mapping f: V* → F}
for each v ∈ N do m(v) := 1; m(f) := 0;
repeat change := false;
  for each arc a in A do
      begin temp := m(from(a));
      m(from(a)) := f(α(a))(m(into(a)))
      if temp ≠ m(from(a)) then change := true
      end
until not change;
{m(v) = mop(v) for all v ∈ N}
```

We shall not prove Theorem 7 since our formulation differs from others only in the requirements that the graph $\mathcal{G}$ have labeled arcs and the property (D5). It is not difficult therefore to amend the proofs given in, say, [8] accordingly.

Let us now show how to express the computation of FIRST_SETS(G) for a given LL(1) grammar G as a distributive data flow problem.

As anticipated in Section 3.4.2, the graph $\mathcal{G}$ is the open-portion graph of G.

The semilattice L is $(\mathcal{S}, \cup)$ where $\mathcal{S}$ is the power set of the power set of T. (That is, each element of $\mathcal{S}$ is a set of subsets of T.) The zero element is $\{\emptyset\}$ and the unit element is 2^T, the power set of T (the set containing all subsets of T).

The elements of the function space F are pairs $\langle x, b\rangle$ where $x \subseteq T$ and b is a Boolean. Function application is defined by

$$\langle x, b\rangle(X) = \bigcup_{y \in X} \{x \cup \textbf{if } b \textbf{ then } y \textbf{ else } \varnothing\}.$$

Function composition is defined by

$$\langle x, b\rangle \cdot \langle y, c\rangle = \langle x \cup \textbf{if } b \textbf{ then } y \textbf{ else } \varnothing,\ b \textbf{ and } c\rangle.$$

Finally the mapping $\not\ell$: $(N \cup T)^* \to F$ is given by

$$\not\ell(\alpha) = \langle \mathrm{FIRST}(\alpha), \mathrm{NULLABLE}(\alpha)\rangle.$$

The definitions of the function space F and the mapping $\not\ell$ are a little curious, but there is a simple explanation. Our principle objective has been to define $\not\ell$ so that $\not\ell(\gamma)(\{\varnothing\}) = \{\mathrm{FIRST}(\gamma)\}$. We also require that $\not\ell(\alpha)\cdot\not\ell(\beta)(\{\varnothing\}) = \not\ell(\alpha\cdot\beta)(\{\varnothing\})$. But $\mathrm{FIRST}(\alpha\cdot\beta) = \mathrm{FIRST}(\alpha) \cup \textbf{if } \mathrm{NULLABLE}(\alpha) \textbf{ then } \mathrm{FIRST}(\beta)$. Hence the two components in $\not\ell(\alpha)$—$\mathrm{FIRST}(\alpha)$ and $\mathrm{NULLABLE}(\alpha)$.

LEMMA 8. *(L, F) is a distributive data flow framework.*

PROOF. L is obviously a semilattice and we can verify that F satisfies conditions D1–D4 as follows.

(D1) Let $X, Y \in 2^T$. Then

$$\begin{aligned}\langle x, b\rangle(X \cup Y) &= \bigcup_{y \in X \cup Y} \{x \cup \textbf{if } b \textbf{ then } y \textbf{ else } \varnothing\} \\ &= \bigcup_{y \in X} \{x \cup \textbf{if } b \textbf{ then } y \textbf{ else } \varnothing\} \\ &\quad \cup \bigcup_{y \in Y} \{x \cup \textbf{if } b \textbf{ then } y \textbf{ else } \varnothing\} \\ &= \langle x, b\rangle(X) \cup \langle x, b\rangle(Y).\end{aligned}$$

(D2) $\langle \varnothing, \textbf{true}\rangle$ is the identity function.

(D3) $\langle x, b\rangle(\langle y, c\rangle(X))$

$$\begin{aligned}&= \langle x, b\rangle\Big(\bigcup_{z \in X} \{y \cup \textbf{if } c \textbf{ then } z \textbf{ else } \varnothing\}\Big) \\ &= \bigcup_{z \in X} \{x \cup \textbf{if } b \textbf{ then } (y \cup \textbf{if } c \textbf{ then } z \textbf{ else } \varnothing) \textbf{ else } \varnothing\} \\ &= \bigcup_{z \in X} \{x \cup (\textbf{if } b \textbf{ then } y \textbf{ else } \varnothing) \cup (\textbf{if } b \textbf{ and } c \textbf{ then } z \textbf{ else } \varnothing)\} \\ &= \langle x \cup \textbf{if } b \textbf{ then } y \textbf{ else } \varnothing,\ b \textbf{ and } c\rangle\ (X) \\ &= (\langle x, b\rangle \cdot \langle y, c\rangle)(X).\end{aligned}$$

(D4) Let $X \in 2^T$. Then $X = \bigcup_{x \in X} \langle x, \textbf{false}\rangle(\varnothing)$. □

THEOREM 9. *If G is an $LL(1)$ grammar, then FIRST_SETS(G) is the MOP solution to a distributive data flow problem.*

PROOF. Noting that $\not\ell(\gamma)(\{\varnothing\}) = \text{FIRST}(\gamma) \cup \textbf{if } \text{NULLABLE}(\gamma) \textbf{ then } \varnothing \textbf{ else } \varnothing = \text{FIRST}(\gamma)$, we have by Lemma 5, $\text{FIRST\_SETS}(G) = mop(s) = \cup\{\not\ell(\gamma)(\{\varnothing\}) \mid \gamma \text{ spells a path from } s \text{ to } f \text{ in } \mathcal{G}\}$ where $\mathcal{G}$ is the open portion graph. This together with Lemma 8 leaves only the proof of condition (D5), which is proved as follows:

$$\begin{aligned}(\text{D5}) \quad \not\ell(\alpha\cdot\beta) &= \langle \text{FIRST}(\alpha\cdot\beta), \text{NULLABLE}(\alpha\cdot\beta)\rangle \\ &= \langle \text{FIRST}(\alpha) \cup \textbf{if } \text{NULLABLE}(\alpha) \textbf{ then } \text{FIRST}(\beta), \\ &\qquad\qquad \text{NULLABLE}(\alpha) \textbf{ and } \text{NULLABLE}(\beta)\rangle \\ &= \not\ell(\alpha)\cdot\not\ell(\beta). \end{aligned}$$

□

COROLLARY. *If G is an $LL(1)$ grammar, then FIRST_SETS(G) may be computed using Algorithm* 1.

4. MIN AND MAX FOLLOW COSTS

4.1 The Problems and Their Solution

Classifying the terminal symbols offers a partial solution to reducing the size of the parameter and boundary cost tables, but the improvement (from 24,000 to 15,000 entries for Pascal) is insufficient for practical purposes. This section describes a further analysis of the parameter values, which leads to a much bigger reduction in the table sizes.

We remarked earlier that the boundary costs are independent of the valid prefix. Now all parameters are strictly positive, so if perchance a boundary cost is negative we know immediately that it is unnecessary—we do not need to compare it with the parameter to know that it is smaller, and we know in advance what the chosen recovery action will be. Conversely, if the boundary cost is ∞, it is again unnecessary because we know in advance that the parameter value will always be less than or equal to it. Thus in these two cases we can reduce our storage requirements and the time taken to evaluate the recovery action.

Unfortunately, the latter analysis is not sufficient to reduce the parameterization to an acceptable level. We can, however, complicate the analysis by including some knowledge of the values taken by the parameter. Suppose we consider the recovery action given input symbol t at production position $A \to \alpha\cdot\beta$. Consider the parameter At passed to A at this juncture. Suppose we precompute the minimum and maximum values that may be taken by At. Let these be denoted by $m(A, t)$ and $M(A, t)$, respectively. Then evaluation of At and the boundary cost is unnecessary at this position if either

$$B(A \to \alpha\cdot\beta, t) \le m(A, t) \qquad \text{or} \qquad B(A \to \alpha\cdot\beta, t) \ge M(A, t).$$

In the former case we know that the nonreturn action should always be chosen; conversely, in the latter case the return action should always be chosen.

Another way of expressing this is that the boundary cost and parameter value are *necessary* if there are valid prefixes u_1 and u_2, both of which define the production position $A \to \alpha\cdot\beta$, but for which the chosen recovery action when t is input is in one case to return from and in the other case not to return from the

call of the procedure recognizing A, that is, when

$$m(A, t) < B(A \to \alpha \cdot \beta, t) < M(A, t).$$

The functions m and M are called the minimum and maximum follow costs and evaluating them is a global flow analysis problem. As we shall see, m is straightforward to evaluate, but evaluating M is the difficult problem to which we referred in the introduction.

Let us now formulate precisely the definitions of m and M. Both depend on the "follow set" of A, which we define first.

Definition 12 (Follow Sets). Let $A \in N$. The *follow set of* A, denoted FOL(A), is a subset of $(N \cup T)^*$ defined by

$$\text{FOL}(A) = \{\gamma \mid \exists \upsilon \in T^* : Z \Rightarrow_l^* \upsilon A \gamma\}.$$

Definition 13 (Min Follow Costs). Let $A \in N$ and $t \in T$. The *minimum cost of repairing* t *following* A, denoted $m(A, t)$, is defined by

$$m(A, t) = \min\{P(t, \gamma) \mid \gamma \in \text{FOL}(A)\}$$

The definition of M is essentially the same as for m, but with max replacing min. However we need first to define the maximum of an infinite set of integers. Since we shall exploit it later, it is useful for us to go one step further and define a semilattice L at this point.

Definition 14 (Semilattice L). The semilattice L consists of the set $\mathbb{N} \cup \{\omega, \infty\}$ (where $\mathbb{N}$ is the set of natural numbers), together with meet operation $\wedge$ defined by

$$\left.\begin{aligned} a \wedge b &= \max\{a, b\} \\ a \wedge \omega &= \omega = \omega \wedge \omega \\ a \wedge \infty &= \infty = \omega \wedge \infty = \infty \wedge \infty \end{aligned}\right\} \text{ for all } a, b \in \mathbb{N}$$

$$\begin{aligned} \textstyle\bigwedge X &= \infty && \text{if } \infty \in X \\ &= \omega && \text{if } X \subset \mathbb{N} \cup \{\omega\} \text{ and } \omega \in X \\ &= \omega && \text{if } X \text{ is an infinite subset of } \mathbb{N} \\ &= \max\{a_1, a_2, \ldots, a_n\} && \text{if } X = \{a_1, \ldots, a_n\} \text{ is a finite subset of } \mathbb{N}. \end{aligned}$$

The zero element of L is ∞.

The zero element of L is ∞.

Definition 15 (Max Follow Costs). Let $A \in N$ and $t \in T$. The maximum cost of repairing t following A, denoted $M(A, t)$, is defined by

$$M(A, t) = \textstyle\bigwedge \{P(t, \gamma) \mid \gamma \in \text{FOL}(A)\}.$$

Note that $M(A, t)$ may be ω even though $P(t, \gamma)$ is finite for each $\gamma \in \text{FOL}(A)$. A simple example would be a grammar with productions

$$Z \to S \dashv \quad S \to (S) \quad S \to a.$$

Since $Z \Rightarrow_l^* ({}^n S)^n \dashv$ for all $n \geq 0$, $M(S, \dashv) = \omega$ provided only that $\dashv$ may not be changed to) at finite cost.

The key to evaluating m and M is the observation that FOL(A) is a *regular* subset of $(N \cup T)^*$ represented by a set of paths through the open-portion graph. To be precise we have the following lemma.

LEMMA 10. *There is a path spelling γ from A to Z in the open-portion graph $\mathcal{G}$ if and only if $Z \Rightarrow_l^* vA\gamma$ for some $v \in T^*$.*

PROOF. Elementary. □

Now the evaluation of m amounts to a shortest path problem. For, applying Lemma 1, m is the solution to the following system of simultaneous equations:

$$m(Z, t) = \infty$$

$$m(A, t) = \min_{B \to \alpha \cdot A\beta \in R(A)} \{\min\{P(t, \beta), I(\beta) + m(B, t)\}\}$$

for all $A \neq Z \in N$, where $R(A)$, the right-hand-side occurrences of A, is $\{B \to \alpha \cdot A\beta \mid B \to \alpha A\beta$ is a production of $G\}$.

Indeed, it is clear from its definition that $m(A, t) = P(t, \text{FOL}(A))$ and FOL(A) is a regular subset of $(N \cup T)^*$. So evaluating m is an application of the least cost repair of a regular language [5, chap. 5] in the particular case in which the input string is of length 1. Algorithms applicable to its solution therefore include Dijkstra's shortest path algorithm [10], Gauss–Seidel iteration [8], or Gaussian elimination [6, 8, 22].

Evaluating M is not so straightforward since it is not susceptible to an iterative technique but requires the use of an elimination technique, one of the principal reasons being that the lattice L is not finite. (Note, though, that finiteness is not a necessary condition for the applicability of algorithm 1.) It is an instance of a *continuous data flow problem* [22], which is not a distributive data flow problem.

A continuous data flow problem is defined almost identically to a distributive data flow problem (sec. 3.4.3). The differences are that finiteness of L and the distributivity condition (D1) are abandoned in favor of the following:

(D1′) Each function $f \in F$ is *continuous*, that is, for any nonempty $X \subseteq L$,

$$f(\wedge X) = \wedge\{f(x) \mid x \in X\}.$$

(D3′) F is closed under meet and * (in addition to composition) where

(i) $(f \wedge g)(x) = f(x) \wedge g(x)$;
(ii) $f^*(x) = \wedge\{f^i(x) \mid i \geq 0\}$.

Note that condition (D1′) is stronger than (D1): A continuous function is also distributive. In fact, in our application the function space (see Definition 16) satisfies (D1).

Definitions 7, 14, and the following additional definitions express the evaluation of $M(A, t)$, for fixed $t \in T$ and all $A \in N$, as a continuous data flow problem:

Definition 16 (Function Space F). A *valid cost pair* is a pair $\langle a, b\rangle$, such that $a, b \in \mathbb{N} \cup \{\omega, \infty\}$ and $a > 0$ or $b > 0$. An element of F is an ordered sequence $f = \langle a_1, b_1\rangle, \langle a_2, b_2\rangle, \ldots, \langle a_n, b_n\rangle$ of valid cost pairs such that $n \geq 1$, $a_i < a_{i+1}$, and $b_i > b_{i+1} (1 \leq i \leq n - 1)$.

Each element of F is a function from $\mathbb{N} \cup \{\omega, \infty\}$ into itself given by

$$(\langle a_1, b_1\rangle, \langle a_2, b_2\rangle, \ldots, \langle a_n b_n\rangle)(x)$$
$$= \max\{\min\{a_1 + x, b_1\}, \min\{a_2 + x, b_2\}, \ldots, \min\{a_n + x, b_n\}\}.$$

Definition 17 (Operations on Functions). Letting λ denote the empty sequence of valid cost pairs, we define the meet, product, and star operations on F as follows.

(a) Function meet:

$$(\langle a, b\rangle, f) \wedge (\langle c, d\rangle, g)$$
$$\begin{array}{lll} = f \wedge (\langle a, b\rangle, \langle c, d\rangle, g) & \text{if} & a < c \quad \text{and} \quad b > d, \\ = f \wedge (\langle c, d\rangle, (\langle a, b\rangle \wedge g)) & \text{if} & a > c \quad \text{and} \quad d > b, \\ = f \wedge (\langle \max\{a, c\}, \max\{b, d\}\rangle \wedge g) & \text{otherwise,} & \end{array}$$

where f, g denote elements of F or λ , and

$$\lambda \wedge f = f = f \wedge \lambda \quad \text{for all} \quad f \in F.$$

(Note: Clearly $\langle a, b\rangle, \langle c, d\rangle, g$ conforms to the definition of an element of F when $a < c$ and $b > d$. It is also straightforward to show by induction on the length of g that $\langle c, d\rangle, (\langle a, b\rangle \wedge g)$ conforms to the definition when $c < a$ and $d > b$.)

(b) Function products:

$$(\langle a, b\rangle, f) \cdot (\langle c, d\rangle, g)$$
$$= \langle a + c, \min\{b, a + d\}\rangle \wedge (\langle a, b\rangle \cdot g) \wedge (f \cdot \langle c, d\rangle) \wedge f \cdot g$$

where f, g are elements of F or λ, and

$$\lambda \cdot f = f = f \cdot \lambda \qquad \text{for all} \quad f \in F.$$

(c) Function stars:

$$(\langle 0, b\rangle, f)^* = f^*$$
$$(\langle c, d\rangle, f)^* = \langle 0, \infty\rangle \wedge \langle \max\{c, \omega\}, d\rangle$$

where f is an element of F or λ and

$$\lambda^* = \langle 0, \infty\rangle.$$

Definition 18 (Mapping f_t). Let $t \in T$. Then the mapping f_t from $(N \cup T)^*$ into F is given by

$$f_t(\beta) = \langle I(\beta), P(t, \beta)\rangle \qquad \text{for all} \quad \beta \in (N \cup T)^*.$$

The following theorem enables us to claim that any algorithm that computes regular expressions denoting FOL(A) for each $A \in N$ can be modified into an algorithm for computing $M(A, t)$ for each $A \in N$.

THEOREM 11. *The triple* $(\mathcal{G}, (L, F), f_t)$, *where* $\mathcal{G}$ *is the open-portion graph of* G *and* L, F, *and* f_t *are as in definitions 14, 16, 17, and 18, is a continuous data flow problem. Moreover, the MOP solution to this problem is the max follow cost function for* t (*i.e.*, $mop(A) = M(A, t)$ *for all* $A \in N$).

The proof of Theorem 11 is long and tedious, but it is the entire justification for the spate of definitions just given. We shall, however, give some explanation for our definitions of function meet, product, and star sufficient for the reader to construct the full proof.

Consider first function product (Definition 17(b)). Here our requirement is that $f_t(\beta\cdot\gamma) = f_t(\beta)\cdot f_t(\gamma)$. Now

$$\begin{aligned} f_t(\beta\cdot\gamma)(x) &= \min\{I(\beta\gamma) + x,\ P(t,\ \beta\gamma)\} \\ &= \min\{I(\beta) + I(\gamma) + x,\ \min\{P(t,\ \beta),\ I(\beta) + P(t,\ \gamma)\}\}. \end{aligned}$$

Thus, letting $a = I(\beta)$, $b = P(t, \beta)$, $c = I(\gamma)$, $d = P(t, \gamma)$, it suffices to define $f_t(\beta)\cdot f_t(\gamma) = \langle a, b\rangle\cdot\langle c, d\rangle$ as

$$\langle a, b\rangle\cdot\langle c, d\rangle = \langle a + c,\ \min\{b,\ a + d\}\rangle.$$

The remainder of the definition of function product simply ensures that every cost pair defining f is composed with every cost pair defining g and the results joined together with the $\wedge$ operation.

Having so defined function product it is easy to see from Lemma 10 that

$$\begin{aligned} mop(A) &= \bigwedge\{f_t(\gamma)(\infty) \mid \exists\text{path } \gamma \text{ from } A \text{ to } Z \text{ in } \mathscr{G}\} \\ &= \bigwedge\{f_t(\gamma)(\infty) \mid Z \Rightarrow_t^* uA\gamma \text{ for some } u \in T^*\} \\ &= M(A, t). \end{aligned}$$

Now consider function meet. The objective here is to define $f \wedge g$ so that $(f \wedge g)(x) = f(x) \wedge g(x)$. This objective is realized by our definition since, essentially, $(\langle a_1, b_1\rangle, \ldots, \langle a_n, b_n\rangle \wedge \langle c_1, d_1\rangle, \ldots, \langle c_m, d_m\rangle)(x)$ is defined to be $\langle a_1, b_1\rangle(x) \wedge \cdots \wedge \langle a_n, b_n\rangle(x) \wedge \langle c_1, d_1\rangle(x) \wedge \cdots \wedge \langle c_m, d_m\rangle(x)$. The complication in the definition is in the formation of $\langle a, b\rangle \wedge \langle c, d\rangle$ when $a \geq c$ and $b \geq d$ or, symmetrically, when $c \geq a$ and $d \geq b$. This is easily explained for

$$\begin{aligned} \langle a, b\rangle(x) \wedge \langle c, d\rangle(x) &= \max\{\min\{a + x, b\}, \min\{c + x, d\}\} \\ &= \min\{a + x, b\} = \langle a, b\rangle(x) \quad \text{when} \quad a \geq c \quad \text{and} \quad b \geq d, \end{aligned}$$

but when $a < c$ and $b > d$ (or, symmetrically, $c < a$ and $d < b$), no simplification can be made to $\langle a, b\rangle(x) \wedge \langle c, d\rangle(x)$. Hence our definition of function meet.

The definition of function meet and the observation that $f(\bigwedge X) = \bigwedge f(X)$ for any subset X of $\mathbb{N} \cup \{\omega, \infty\}$ (which is easily proved) enables one to establish that

$$mop(A) = M(A, t) = (\wedge\{f_t(\gamma) \mid \exists\text{path } \gamma \text{ from } A \text{ to } Z \text{ in } \mathscr{G}\})(\infty).$$

The peculiar ordering property on sequences of valid cost pairs facilitates greatly the evaluation of function stars. Suppose there are cycles in the open-portion graph beginning and ending on A. Suppose some subset of these spell the strings $\beta_1, \beta_2, \ldots, \beta_n$ (i.e., $A \Rightarrow_t^* u_iA\beta_i$ for some $u_i \in T^*$ and all i, $1 \leq i \leq n$). By our earlier analysis of function meet we may assume without loss of generality that $I(\beta_1) < I(\beta_2) < \cdots < I(\beta_n)$ and $P(t, \beta_1) > P(t, \beta_2) > \cdots > P(t, \beta_n)$. Let $Q = \{\beta_1, \ldots, \beta_n\}$, a_i denote $I(\beta_i)$, b_i denote $P(t, \beta_i)(1 \leq i \leq n)$ and f denote $\langle a_1, b_1\rangle, \ldots, \langle a_n, b_n\rangle$. Clearly, $A \Rightarrow_t^* uA\gamma$ for some $u \in T^*$, for all $\gamma \in Q^*$. So our

requirement is that

$$f^*(x) = \Lambda \{\langle I(\gamma), P(t, \gamma)\rangle(x) \mid \gamma \in Q^*\}.$$

Now, by the ordering property, $\beta_i \Rightarrow^* \Lambda$ if and only if $i = 1$ and $a_1 = I(\beta_1) = 0$. Moreover, if indeed $\beta_1 \Rightarrow^* \Lambda$ it is easily proved that $f_t(\alpha\beta_1\delta) \leq f_t(\alpha\delta)$ for all $\alpha, \delta \in (N \cup T)^*$ and hence

$$\Lambda \{\langle I(\gamma), P(t, \gamma)\rangle(x) \mid \gamma \in Q^*\} = \Lambda \{\langle I(\gamma), P(t, \gamma)\rangle(x) \mid \gamma \in (Q - \{\beta_1\})^*\}.$$

So the first pair in the sequence defining f can be ignored when $a_1 = 0$. This gives us the first equation in the definition of function stars.

Suppose, therefore, that $n > 1$ or $n = 1$ and $a_1 > 0$. Let $\langle c, d\rangle = \langle a_1, b_1\rangle$ if $a_1 > 0$ and $\langle c, d\rangle = \langle a_2, b_2\rangle$ otherwise. Let $R = \{\beta_2, \ldots, \beta_n\} \cup$ **if** $a_1 = 0$ **then** ϕ **else** $\{\beta_1\}$, $\beta = \beta_2$ if $a_1 = 0$ and $\beta = \beta_2$ otherwise. Note that $c > 0$ and $d = \Lambda \{P(t, \beta_i) \mid \beta_i \in R\}$. More especially $d = \Lambda \{P(t, \gamma) \mid \gamma \in R^+\}$. Thus d is an upper bound on $f \cdot f^*(x)$. Now suppose $d = \infty$. Then $f^*(x)$ is ∞ if $c = \infty$, but otherwise it is at least ω. (For $f^*(x) \geq I(\beta^m) + x = mc + x$ for all $m \geq 0$.) In other words, $(\langle c, d\rangle, g)^* = \langle 0, \infty\rangle \wedge \langle \max\{c, \omega\}, d\rangle$ which is the last clause in the definition of function star.

4.2 Max Follow Costs and an Idiosyncratic Flow Problem

One of the interesting features of the min and max follow cost problems is that they can be used to model other problems in global data flow analysis. This is exemplified in this section by modeling an "idiosyncratic" flow problem [12, 22] as a max follow cost problem.

The problem is this. Suppose $\mathcal{G}$ is the flow graph of a program that contains occurrences of an expression a. Let E be the arc set, V the node set, and S the start node of $\mathcal{G}$. With each arc e of E is associated an *effect*, which has one of four values depending upon what flow of control through arc e does to the value of a.

$$\text{effect}(e) = \left\{\begin{array}{l}\text{gen}\\ \text{kill}\\ \text{injure}\\ \text{trans}\end{array}\right\} \text{ if } \left\{\begin{array}{l}\text{the program recomputes } a,\\ \text{the program makes a large change to } a\\ \text{the program makes a small change to } a\\ \text{the program does not affect } a.\end{array}\right.$$

For any node A, we say a is *implicitly available on entry to* A if there is a positive bound b such that, for every path $p = e_1, e_2, \ldots, e_k$ from S to A, there is an i such that (i) $\text{effect}(e_i) = \text{gen}$, (ii) $\text{effect}(e_j) \neq \text{kill}$ for $i < j \leq k$, and (iii) the number of values of j such that $i < j \leq k$ and $\text{effect}(e_j) = \text{injure}$ is bounded by b. The problem is to determine from $\{\text{effect}(e) \mid e \in E\}$ the nodes at which a is implicitly available.

To model this as a max follow cost problem we construct from the flow graph $\mathcal{G}$ a grammar G as follows. The nonterminal alphabet is V and the terminal alphabet is $E \cup \{a, k, i\}$ (where we assume $E \cap \{a, k, i\} = \varnothing$). The productions are constructed from the arcs E as follows. Suppose e is an arc from A to B. Then

introduce the production

$$A \to eBa \quad \text{if} \quad \text{effect}(e) = \text{gen},$$
$$A \to eBk \quad \text{if} \quad \text{effect}(e) = \text{kill},$$
$$A \to eBi \quad \text{if} \quad \text{effect}(e) = \text{injure},$$
$$A \to eB \quad \text{if} \quad \text{effect}(e) = \text{trans}.$$

Introduce also the production $A \to \Lambda$ for each $A \in N$. Finally define the primitive edit costs as follows.

$$\text{cost}(\Lambda \to k) = \infty, \quad \text{cost}(\Lambda \to i) = 1,$$
$$\text{cost}(a \to a) = 0, \quad \text{cost}(a \to k) = \infty, \quad \text{cost}(a \to i) = \infty.$$

All other costs are arbitrary

The claim is that $M(A, a) = \infty$ if and only if a is not implicitly available at node A in $\mathcal{G}$. This is easy to prove because $S \Rightarrow^* e_1e_2\cdots e_nAw$ if and only if there is a path $e_1, e_2, \ldots, e_n$ from S to A in $\mathcal{G}$ and $w = t_nt_{n-1}\cdots t_1$ where t_k is a if effect(e) = gen, t_k is k if effect(e) = kill, t_k is i if effect(e) = injure and t_k is Λ if effect(e) = trans.

In terms of locally least-cost error recovery, the implicit-availability problem is this. Suppose the path $p = e_1e_2\cdots e_n$ has been parsed when a is encountered (i.e., the input is $e_1e_2\cdots e_na\cdots$). Then there are three possibilities:

(a) a is OK;
(b) recovery can be achieved by inserting a finite number of i's;
(c) no repair of a is possible according to the given costs and the analysis must be aborted.

An implication of this result is that the continuous data flow framework we constructed in Section 4 is an alternative framework (albeit a less efficient one) to that proposed by Tarjan [22] for the solution of this problem.

5. EXPERIMENTAL RESULTS

The objective of this paper has been to show how a number of optimization problems arising in the practical implementation of locally least-cost error recovery can be expressed as global data flow analysis problems; elsewhere [4] we have given a detailed analysis of the efficiency and effectiveness of the technique vis à vis the follow set technique advocated by Wirth [24]. In this section we present a brief summary of the results we have obtained using our parser generator where they pertain to reducing the size of the parameter and boundary cost tables. All these results relate to a Pascal grammar having 69 terminal symbols and 368 production positions; for complete details see Bugge [7].

We have already mentioned that classifying terminal symbols effectively reduced their number to 38, that is, by approximately 40 percent. In theory the resulting reduction in the size of the boundary cost and parameter tables may

Table I. Table Sizes for Five Pascal Parsers

Constraints on edit costs	Number of entries	
	Boundary costs	Parameters
All infinite	44	187
All finite	693	1164
High delete costs made infinite	528	1073
Delete costs and change costs either infinite or 1	184	669
Infinite delete costs made unbounded	236	621

not be so great; in practice we found that the reduction was between 30 and 40 percent.

The effect of the min and max follow cost calculations on the table sizes depends to some extent on the primitive edit costs. We generated five parsers using different assignments to the edit costs. The resulting sizes of the parameter and boundary cost tables is shown in Table I. The parser having the fewest table entries is the first; for this parser all costs were set to infinity and therefore the parser aborts from all syntax errors. (The effect of the parameterization is to transform the recursive descent parser from a strong LL(1) parser into an LL(1) parser, i.e., from a parser that may take parsing decisions before announcing an error into one that announces errors at the earliest possible opportunity.) The remaining parsers were all generated with the aim of producing the best possible error recovery within the specified constraints on the edit costs.

The maximum possible size of the boundary cost table is $198 \times 69 = 13662$ entries (the number of production positions excluding the first in each production $\times$ the number of terminals), and so the worst improvement we obtained was a reduction to 5 percent of the maximum table size. The maximum possible size of the parameter table is $205 \times 69 = 14007$ entries (the number of right-hand-side occurrences of nonterminals $\times$ the number of terminals), and so the worst improvement we obtained was a reduction to less than 10 percent of the maximum table size. The best improvements we obtained, excluding the first parser, were to table sizes of less than 2 and 5 percent of the maximum possible sizes of the boundary cost and parameter tables, respectively. Nevertheless, the absolute table sizes are still significantly large and are a drawback to the recovery scheme.

The main conclusions of [4] were that locally least-cost error recovery is more effective than the follow set scheme, but because they have the same inherent limitations and because locally least-cost error recovery requires substantially more storage space than the follow set scheme, the follow set scheme is to be recommended in conventional programming environments. It was noted though that the parameters (in locally least-cost error recovery) most often perform a simple Boolean function (e.g., in the infinite-cost parser they indicate whether a terminal symbol is OK or not) and so, by replacing integer entries by Booleans, further improvements in the table sizes may tip the balance in favor of locally least-cost error recovery. However, algorithms for detecting when such replacements may be made have yet to be developed and remain as open problems.

6. CONCLUSIONS

Originally global data flow analysis was developed for use in "optimizing" compilers for improving code compiled from a hand-written program. The application described here is novel in the sense that the code being improved is itself automatically generated. It is an interesting application, first, because without it the scheme described in [5] would simply not be viable and second, because it seems to be a harder application than others studied previously. We have reason to believe that parameter minimization in implementing locally least-cost error recovery in LR parsers [3] may be yet more difficult. The solution to our optimization problems has nevertheless been relatively straightforward and this must be attributed to earlier theoretical work on identifying a most general framework for global flow analysis problems [9, 13, 15–17, 21–23].

Solving min and max follow cost problems may prove to be an important paradigm for evaluating path-finding algorithms, particularly elimination techniques. The reason for this is that we must solve about 40 different problems (the max follow cost problem for each terminal class in the grammar) defined on the same flow graph (the open portion graph). Thus the advantages of using an algorithm that exploits the structure of the graph over, say, the iterative techniques, which do not, will be multiplied and so should be readily apparent. It would be interesting to see, for instance, whether notions of reducibility used in conventional global data flow problems are of value to the follow cost problems presented here.

ACKNOWLEDGMENTS

Many thanks to Stuart Anderson for his careful reading of the manuscript, to Edle Bugge for the results quoted in Section 5, and to the referees for their critical comments.

REFERENCES

1. AHO, A.V., AND ULLMAN, J.D. *The Theory of Parsing, Translation and Compiling.* Vol. 1, *Parsing.* Prentice-Hall, Englewood Cliffs, N.J., 1972.
2. ANDERSON, S.O., AND BACKHOUSE, R.C. Locally least-cost error recovery in Earley's algorithm. *ACM Trans. Program. Lang. Syst. 3*, 3 (July 1981), 318–347.
3. ANDERSON, S.O., AND BACKHOUSE, R.C. Locally least-cost error recovery in LR parsers: A basis. Tech. Rep., Dept. of Computer Science, Heriot-Watt Univ., Edinburgh, Scotland, 1981.
4. ANDERSON, S.O., BACKHOUSE, R.C., BUGGE, E.H., AND STIRLING, C.P. An assessment of locally least-cost error recovery. *Comput. J. 26*, 1 (1983), 15–24.
5. BACKHOUSE, R.C. *Syntax of Programming Languages: Theory and Practice.* Prentice-Hall Int., London, 1979.
6. BACKHOUSE, R.C., AND CARRÉ, B.A. Regular algebra applied to path-finding problems. *J. Inst. Math. Appl. 15* (1975), 161–186.
7. BUGGE, E.H. Implementing and assessing locally least-cost error recovery for Pascal. Master's thesis, Dept. of Computer Science, Heriot-Watt Univ., Edinburgh, Scotland, 1982.
8. CARRÉ, B.A. An algebra for network routing problems. *J. Inst. Math. Appl. 7* (1971), 273–294.
9. COUSOT, P., AND COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages.* (Los Angeles, Calif., Jan. 17–19, 1977). ACM, New York, 1977, pp. 238–252.
10. DIJKSTRA, E.W. A note on two problems in connection with graphs. *Numer. Math. 1* (1959), 269–271.

11. FISCHER, C.N., MILTON, D.R., AND QUIRING, S.B. An efficient insertion-only error-corrector for LL(1) parsers. *Acta Inf. 13* (1980), 141–154.
12. FONG, A.C. Generalized common subexpressions in very high level languages. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages.* (Los Angeles, Calif., Jan. 17–19, 1977). ACM, New York, 1977, pp. 48–57.
13. FONG, A.C., KAM, J.B., AND ULLMAN, J.D. Application of lattice algebra to loop optimizations. In *Conference Record of the 2nd ACM Symposium on Principles of Programming Languages.* (Palo Alto, Calif., Jan. 20–22, 1975). ACM, New York, 1975, pp. 1–9.
14. HOPCROFT, J.E., AND ULLMAN, J.D. *Formal Languages And Their Relation to Automata.* Addison-Wesley, Reading, Mass., 1969.
15. KAM, J.B., AND ULLMAN, J.D. Global data flow analysis and iterative algorithms. *J. ACM 23*, 1 (Jan. 1976), 158–171.
16. KAM, J.B., AND ULLMAN, J.D. Monotone data flow analysis frameworks. *Acta Inf. 7* (1977), 305–317.
17. KILDALL, G.A. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages.* (Boston, Mass., Oct. 1–3, 1973). ACM, New York, 1973, pp. 194–206.
18. LEWI, J., DEVLAMINCK, K., HUENS, J., AND HUYBRECHTS, M. The ELL(1) parser generator and the error recovery mechanism. *Acta Inf. 10* (1978), 209–228.
19. PAI, A.B., AND KIEBURTZ, R.B. Global context recovery: A new strategy for syntactic error recovery by table-driven parsers. *ACM Trans. Program. Lang. Syst. 2*, 1 (Jan. 1980), 18–41.
20. RÖHRICH, J. Methods for the automatic construction of error-correcting parsers. *Acta Inf. 13* (1980), 115–139.
21. ROSEN, B.K. Monoids for rapid data flow analysis. *SIAM J. Comput. 9* (1980), 159–196.
22. TARJAN, R.E. A unified approach to path problems. *J. ACM 28*, 3 (July 1981), 577–593.
23. TARJAN, R.E. Fast algorithms for solving path problems. *J. ACM 28*, 3 (July 1981), 594–614.
24. WIRTH, N. *Algorithms + Data Structures = Programs.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

Received September 1981; revised August 1982; accepted June 1983