Diagnostic Parsing in a Software Development Environment

Walter Williams

October 13, 2002

Abstract

This paper explores the use of partial parsing and other NLP techniques in providing diagnostic information about errors in an Interactive Program Development Environment (IPDE). The paper suggests that such techniques can provide better diagnostics than current IDPEs by partial parsing of the input and increased interaction between the lexical, syntactic, and semantic analysis.

Contents

I Introduction 1

1 How Existing Interactive Development Tools Work 2

2 Compiler Terminology 2

2.1 Scanning 3

2.2 Parsing 3

2.3 Semantic analysis 3

3 A note on Grammars, Syntax and Semantics 4

4 Natural Language Processing 4

II Errors Detection and Diagnosis 5

4.1 Error Recovery 5

5 Error Diagnosis 6

III Programming Languages and Grammars 7

6 Languages and Grammars 7

7 Three levels of grammar 7

7.1 Programs 8

7.2 Statements 8

7.3 Expressions 8
8 Practical Diagnosis

9 Modification Metrics
   9.1 Holes – missing lexical elements .............................................. 10
   9.2 Interjections – extraneous lexical elements ................................. 11
   9.3 Substitutions ................................................................. 11

IV Examples of specific errors

10 The Examples
   10.1 Missing input ................................................................. 11
   10.2 Interjection ................................................................. 12
   10.3 Ambiguous missing or extraneous input ................................... 12
   10.4 Multiple Related Holes (missing parenthesis) ............................ 12
   10.5 Multiple interjection or transposition ..................................... 13
   10.6 Lexical Error ................................................................. 13
   10.7 Name resolution/type coercion ............................................. 14
   10.8 Lexical replacement ......................................................... 14
   10.9 Control Flow Error ......................................................... 15
   10.10 Data flow errors ......................................................... 15
   10.11 Unbalanced Delimiters ................................................... 16
   10.12 Garbage ................................................................. 16

11 Comments on Examples ....................................................... 16

12 Input-modification vs. Grammar-Modification ............................ 17
   12.1 Requirements of an Edit-Distance Metric ................................. 17

V Edit Distance and Diagnosis ........................................... 17

13 Granularity of Diagnosis .................................................. 18

14 Distance Metric
   14.1 Height ................................................................. 18
   14.2 Type of Modification ..................................................... 18
   14.3 Symbol Table Edit ......................................................... 18

15 Probabilistic Methods .................................................. 18

16 Free Grammar-Edit Operations:
   16.1 Static Grammar Distance .............................................. 19
   16.2 Complexity ............................................................ 19

VI Related Research .................................................. 19

17 Cox and Clarke .......................................................... 19
18 Aho and Peterson’s Error Correcting Parser  
19 The Cornell Synthesizer Generator  
20 Harmonia, Ensemble and PAN  

VII Conclusion  
21 Summary  
22 Future Research  

VIII Appendices  
A Desirable IPDE Features  
B Notes on Practical Requirements  
  B.1 Practical programs  
  B.2 Editing Focus  
  B.3 Responsiveness  
  B.4 Conclusion
Part I

Introduction

The overall goal of this research is to develop an Interactive Program Development Environment (IPDE) that analyzes a program as it is being developed or modified and makes the results of that analysis available to the user. To achieve this goal, the IPDE must incorporate an editor capable of detecting and diagnosing syntactic (and some semantic) programming errors. Ultimately, the IPDE should provide high-level semantic analysis of data and control-flow for complete programs as well.

The present paper explores the issue of diagnosing errors in programs at edit-time. A program being edited is likely to be in an intermediate or incomplete state much of the time, and while in that state, the program will, more often than not, contain syntax errors. It might appear that such a program cannot be parsed. However, this paper draws on research in Natural Language Processing to show that a program in this state can be parsed and analyzed. The focus on analyzing incorrect programs is crucial, since experience (as well as common sense) indicates that it is exactly when a program contains errors that diagnostic analysis is most useful.

As a program is edited, some changes to the text may cause substantial changes in the structure of the parse tree representing that text which, in turn, may require a considerable amount of computation. This paper does not address this issue directly, but rather focuses on the detection and diagnosis of errors. The issue of computational resource requirements is addressed informally in Appendix B.

The paper draws on parsing algorithms and grammar models developed for Natural Language Processing to achieve robust, fully-incremental edit-time parsing of a program. While parsing is often considered a “dead” issue in compilers, it remains an active area of research in the Natural Language Processing community, where the ability to parse damaged, incomplete, or merely ambiguous text is referred to as robustness and the notion of incremental modifications to a derivation is called fully-incremental parsing.

Unlike most existing interactive program development environments, which trace their lineage to editor and compiler technology first developed in the late 1960’s and early 1970’s, the system envisioned here is intended to provide robust and fully-incremental parsing and to maintain an internal representation (e.g., Abstract Syntax Tree) which accurately reflects the program text while it is being modified.

The design of an IPDE with these capabilities raises a number of challenging questions: How can error information be presented without overwhelming the user? What information is useful at edit-time? How can an incorrect program be analyzed? How can such a program even be adequately parsed? Will the attempt collapse under the weight of its own data structures or time complexity?

These issues are wide ranging in their implications and too diverse to be addressed in a single paper. They will not all be addressed here. Appendix A provides some background and discusses some desirable IPDE features.

User-interface issues will not be discussed either. However, for the purpose of visualizing diagnostic information, one may assume that the IPDE simply highlights errors and provides diagnostic information in a “fly-over” or “tool-tip” window. The objective is to make more information available to the programmer without being too intrusive.

The original intent of this study was limited to parsing—that is, to syntactic analysis. As it has become apparent during the course of the study that the division between syntax and semantics is
often somewhat artificial. Thus, semantic analysis will not be completely ignored.

While the ability to efficiently and correctly update a parse in response to changes in the program text is a practical necessity, the algorithmic issues related to incremental parsing will not be addressed here. Fully incremental parsing has been addressed in the literature [FER94] and will be incorporated in future work.

The parsing and analysis of a program are notorious for their consumption of resources. The approach described here may initially appear to require a prohibitive amount of storage or processing. We do not address that issue in this paper. However, there are several reasons to believe that the goal is indeed achievable; these are presented informally in Appendix B.

The following discussion draws on ideas and terminology from two distinct disciplines, natural language parsers and compilers. Readers may not be entirely familiar with one or the other of these disciplines. The rest of this introduction includes a brief description of how existing ‘syntax-aware’ editors work, along with background material on compilers (for the natural language folks) and on computational linguistics (for the compiler folks).

The remainder of the paper is divided into seven parts, not including the appendices. Part I provides some background. Parts II and III focus on the diagnosis of errors in general and with respect to computer programs respectively. Part IV presents examples of typical programming errors and their diagnoses. Part V explores approaches to defining a diagnostic edit-distance. Part VI surveys related research and Part VII presents a summary and suggests directions for future research.

1 How Existing Interactive Development Tools Work

Program editors such as EMACS[LEWI98] or Microsoft Visual Studio provide text editing environments that have special features for editing programs. Typical features include simple syntax checking, pretty-printing, and the ability to match uses of names with definitions stored in a symbol table or external library. In general, such editors do not attempt to parse an entire program [COX]. Rather, they use simple automata, which can recognize declarations and other simple constructs, to build a list of declared names. In addition, they perform context free analysis sufficient to establish the scope of identifiers and detect unbalanced delimiters. Such editors can detect enough program structure to provide useful diagnostic information without fully parsing the input e.g., indicating a type error if the types of the arguments to a function call do not match the types declared in the function’s signature. However many, if not most, errors are not detected by these approaches; the user discovers those errors only at compile time.

IPDEs often integrate several tools (e.g., editor, compiler, debugger, and version management) into a single interface. Usually this integration is only skin deep: the extent of the integration is generally limited to the user interface. The individual tools—particularly the compiler and editor—are separate and independent. The objective of this work is to incorporate the analysis capabilities of the compiler into a modern programming editor.

2 Compiler Terminology

A compiler translates program source code into machine code for a specific (possibly virtual) machine. The transformation is divided into two major parts: one that builds an internal structure from the source code, the front end; and another that turns that structure into efficient machine code, the back end. This paper is concerned only with the first part, which focuses on analyz-
ing and “understanding” the source code. Indeed, the intent is to include the analysis normally
performed by a compiler front end in an editor. Typically, the front end of a compiler analyzes
source code—presented as a text file—in three well-defined phases: scanning, parsing and semantic
analysis.

2.1 Scanning

First, a scanner performs lexical analysis by reading the input and partitioning it into lexical items
called tokens (e.g., integer, string or floating-point literal, keyword, identifier, operator symbol, etc.). A scanner may produce tokens containing a semantic value for literal constants and built
in symbols, but normally does not do so for identifiers. This task is generally performed by finite
automata generated for the purpose. The output of the scanner, a stream of tokens, is then handed
off to the next phase, the parser.

2.2 Parsing

A parser reads the stream of tokens provided by the scanner and builds an internal structure known
as an Abstract Syntax Tree (AST). The most common parsing strategies used in compilers are
LL(k) and LALR(k), although operator precedence is sometimes used for parsing expressions.
Additional information about parsing (and compilers in general) can be found in [AHOS6].

The AST is a form of parse tree in which related syntactic elements are attached to a single
node. For example, ‘(a + b)’ would appear as a single ’+‘-node with two children, rather than as
a tree containing the parenthesized expression and an explicit operator symbol. Despite its name,
the AST actually establishes the operational semantics of the program. For example, the ’+‘-node
corresponds directly to the operation of addition.

The completed AST is passed to the next phase: semantic analysis.

2.3 Semantic analysis

Finally, a semantic analysis phase generates an intermediate code representation of the program,
builds auxiliary data structures and checks the program for type consistency and other possible
errors. The primary data structures include:

- a control-flow graph, which models the flow of control defined by if, while or related state-
  ments;
- a data-flow graph, along with def-use and use-def chains, indicating which assignments to a
  variable can affect particular uses of the variable;
- basic blocks, which contain a small sequence of instructions (generally in the form of inter-
  mediate code) and which form the nodes of the data and control flow graphs;
- a symbol table, which contains the various attributes associated with each identifier in the
  program (indexed by identifier).

It is worth noting that the limitations imposed by use of LL(k) and LALR(k) grammars often
force compiler writers to push what are really syntactic requirements on to the semantic analysis
phase (e.g., most compilers seem to recognize a missing return statement in a function that returns
a value during semantic analysis, even though it could be expressed grammatically and detected
during parsing).

3
The *symbol table*, which contains type and other information for the named entities in the program and thus provides access to context sensitive information. The symbol table is used during the semantic analysis phase (and sometimes parsing or lexical analysis as well) to check non-context-free aspects of the program. For example, type conflicts can be discovered by comparing the declared type of an identifier in the symbol table to the type expected at each use of the identifier.

3 A note on Grammars, Syntax and Semantics

Compilers, and textbooks on compilers, usually make a clear distinction between syntax and semantics. That distinction is something of an illusion. Mark Steedman [STEE00] and others have suggested that in natural language there is a strong “rule-to-rule correspondence” between syntax and semantics. Steedman also remarks, (*ibid.* p.11), “The artificial languages that we design ourselves, such as logics or programming languages, exhibit a very strong form of the rule-to-rule relation between their semantics and the syntax as it is defined in the textbook or reference manual.”

In compiler technology “semantic” frequently refers to anything that cannot be conveniently described by a particular subset of the deterministic CFGs. The use of semantic information in parsing is not uncommon in NLP. In practical compilers, the scanner often makes use of semantic information stored in a symbol table. Type checking, control-flow checking, and so on could be incorporated into the grammar of a programming language (sometimes as attributes sometimes directly), if the grammar were expressive enough. Therefore, the word semantic ought be taken with a grain of salt: perhaps it should be read as “semantic or, given a more descriptive grammar model, syntactic,” at least in the context of this paper.

4 Natural Language Processing

Natural language is highly ambiguous. Lexical ambiguity appears whenever words have multiple meanings (e.g., “Tiger Woods would not take his woods into the woods”) and structural ambiguity appears when a sentence (or constituent) has multiple grammatical parses (e.g., “I saw a man with a telescope”). Thus parsers intended for natural language must be capable of maintaining multiple possible parses and often rely on data other than the grammar and input text to resolve ambiguity (e.g., on a lexicon of words with their associated parts of speech and other semantic information).

Natural language processing often uses *Chart Parsers*, which are a family of parsers based on dynamic programming that maintains a table (or graph) of possible partial parses. The chart contains *edges*, which represent partially satisfied grammar rules and are similar in some respects to LR items (dotted grammar rules). These edges are divided into *incomplete edges*, which represent rules that are missing constituents, and *completed edges*, which represent completely satisfied rules. A parse tree (or trees, in the case of an ambiguous parse) can be built from the completed edges of the chart.

In addition natural language parsers often use probabilistic methods. These resolve ambiguity based on probability of occurrence of various possible structures or word uses.

Another parsing strategy, Unification Parsing, constructs a parse tree by unifying features associated with lexical elements and non-terminals. The algorithm is very much like that used for type inference in functional programming languages such as ML.

A variety of grammatical models have been developed to explain the structure of human natural language. Most such models are essentially context free grammars, although some grammatical models, such as the Mildly Context-Sensitive Grammars(MCSG), are able to directly capture
enough local context to easily describe context sensitive phenomena such as agreement and conjunction. Many of these grammatical models can be lexicalized. Lexicalized grammars are grammars in which the rules of the grammar are contained in the lexicon itself. Greibach normal form is, in a weak sense, a lexicalized CFG, since each rule of the normalized grammar is introduced by a terminal symbol (word). If all the terminal symbols of a programming language are included in a compiler’s symbol table, along with their associated grammatical rules and types, then each symbol table entry becomes the focus of both syntactic and semantic analysis. That makes it easy to use type information to resolve ambiguity and to detect type errors during parsing. Moreover, identifiers that have not been declared can be given polymorphic type. This enables the IPDE to provide edit-time type inference in addition to type checking.

**Part II**

**Errors Detection and Diagnosis**

An error is generally defined as an observed behavior that deviates from the desired (specified) behavior of a system. In the case of software diagnosis, the system is the program text and the desired behavior is that the program conform to the rules of the programming language in which it is written.

Detecting the *existence* of a syntax error in a program is relatively easy, but not as useful to a programmer as is diagnosis of the error. LR and some other parsers have a *valid left-prefix property*, which means that the parser will, on a left-to-right scan of the input, report an error as soon as it determines that the input processed so far cannot be a valid prefix of any program. In practice, it is possible to provide some information about the error—whether it is a lexical, syntactic or semantic error, an attribute conflict, an expected token not found, and so on. But such efforts only report the symptoms that result from the error. In an IPDE, we would like to know the actual cause of the error, so that a reasonable diagnosis can be presented to the user and so that the parser can properly recover from the error.

### 4.1 Error Recovery

Error recovery is the ability to continue processing after encountering an error. If the parser is capable of maintaining a “corrected” version of the input, then error recovery is simply a matter of using that version for subsequent analysis (subsequent in time, not necessarily left-to-right). It is important to note that the “corrections” serve only to recover from an error and continue parsing; they are not intended to correct a programmers mistakes. In fact, compilers often resort to “correcting” the input by ignoring text until some recognizable break (such as a statement delimiter, the end of a block of statements or a new declaration) occurs.

In the context of an editor, we cannot expect that the input will actually belong to the language of the parser’s grammar. The major objective of this work is to develop a parsing strategy that can model such input and also identify possible correct variations of the text. This, first of all, requires a robust parsing strategy that can build and maintain a partial parse of the acceptable portions of the text. In addition, it requires a strategy to build and maintain information about portions of the text that are not acceptable.

A partial parse is one that does not reduce to the start symbol of a grammar. An incomplete program may have many partial parses, some of which may overlap if the input is ambiguous. Simultaneous syntactic and semantic analysis can help disambiguate, and thus reject some of those
partial parses as well as providing more useful diagnostic information. Also, partial parses may be extended to (possibly ambiguous) complete parses. Multiple overlapping parses can also be extended to multiple possible complete parses.

5 Error Diagnosis

There are two general approaches to error diagnosis. For a more in-depth survey, see [MOZE92]:

1) Diagnosis from first principles, which generates a diagnosis by comparing the observed behavior of a system to the specified behavior for that system, and;

2) Heuristic approaches, which map symptoms to causes based on expert knowledge and experience.

Diagnostics from first principles is further divided into a consistency-based and abductive approaches.

The consistency-based approach, described in [REIT77], defines a system as a set of components and defines a diagnosis as a minimal subset of components Δ that can explain the difference between the actual output of a system and its intended output. The system uses first order logic to describe the system and generates a diagnosis by finding the smallest set of components whose behavior would have to be abnormal in order to explain the observed behavior. Components, in this case, are the lexical and structural elements of the programming language. The intended output is a derivation. The goal of diagnosis is to find a minimal set of lexical or structural items that explain why the input is not in the specified language. There are likely to be many component sets that explain the error, including a diagnosis that includes every component, which is trivially correct, but useless.

Finding such a minimal set is not always practical and is NP-Complete in the general case. What’s worse, conventional consistency-based diagnosis assumes that the structure of the system and the connections between components are fixed. In the case of a faulty program, a diagnosis may require identifying missing or superfluous components, or restructuring existing components.

The abductive approach [POOL89] does not distinguish correct and incorrect states: it simply defines multiple modes of behavior. An abductive diagnosis is a minimal set of assumptions that produce the observed output from the system description.

Heuristic diagnoses map specific symptoms to probable errors based on expert knowledge and experience. Clearly this approach is not easily applicable to unanticipated system behavior. Nevertheless it is the approach used in many commercial compilers. It can be effective with respect to programming errors, since a few kinds of errors occur quite frequently. Unfortunately it turns the parsing algorithm into a sea of special cases, each dealing with particular kind of error.

The three approaches outlined above are not mutually exclusive.

Certain typographical and programming errors are much more common than others: hitting the key adjacent to the desired key on the keyboard or omitting structural delimiters such as semicolons from a program statement, for example. While heuristic approaches are likely to be useful in diagnosing particular programming errors, they do not provide a general, comprehensive solution to diagnostics. The somewhat ad-hoc nature of heuristic approaches makes it difficult to prove them sound or complete.

Diagnosis from first principles, on the other hand, is based on general mathematical principles. It is therefore amenable to analysis and proof of soundness and completeness. The consistency-based approach focuses on the difference between the observed state and a desired state—between the actual output and correct output. The abductive approach attempts to model, and “explain” every possible state, labeling some as desirable and others as erroneous.
The remainder of this paper describes a consistency-based approach to program diagnostics, which combines aspects of the abductive and first-principle approaches. This approach models a minimal correct version of the input, and presents the difference between that model and the actual input as the diagnosis.

Part III
Programming Languages and Grammars

6 Languages and Grammars

The concept of a program needs to be clarified a bit for the purpose of this paper, since a program may appear in several distinct incarnations. The word “program” is used here to refer to an abstract entity that incorporates information about implementation of an algorithm. Some of this information may have meaning in one context, but not in another. A program may be presented as text in some programming language, as a GUI presentation, as intermediate code, as an AST, or in any number of other ways. None of these need be complete representations of the program, e.g., a text file does not directly expose structural elements, such as data or control flow. An intermediate code representation exposes those structural elements but omits, for example, comments.

According to this view, a program may be expressed in a variety of different languages. The language of interest here, $L$, is the language in which the program appears in a graphical editor. Clearly, $L$ must be closely related to a well defined programming language, but it need not be identical. $L$ can be described by any number of grammars $G_i(L)$. For example, structural units, annotated by delimiters such as semicolons or various kinds of brackets, can be represented graphically by indentation, boxes, or other GUI attributes, without changing the program’s underlying semantics. Additional information about the program, including various types of errors, can be incorporated into $L$, making it a superset of the actual programming language — the union of a programming language and an ”error” language.\footnote{The notion of an editing language, $L$, that is the union of several sub-languages is relevant not only for diagnostics but also for formal program verification and software engineering.}

7 Three levels of grammar

In Programming Languages it is useful (for practical reasons) to distinguish the following three levels of structure:
<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Program</td>
<td><code>#include</code>, <code>import</code>, <code>package</code> keywords, <code>class</code> declarations in Java or function declarations in C.</td>
</tr>
<tr>
<td></td>
<td>The structure of a program—or compilation unit—typically includes linkage information and top-level declarations and block structure.</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>Statement</td>
<td><code>if-then-else</code>, <code>for</code>, <code>while</code>, <code>break</code>, <code>return</code>,...</td>
</tr>
<tr>
<td></td>
<td>Procedural languages include a (relatively small) set of statements, which typically modify control flow. Purly functional languages do not have statements.</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Expression</td>
<td><code>f(a * x + b)</code></td>
</tr>
<tr>
<td></td>
<td>Expressions produce values, and are built up from operations and function application on sub-expressions. The structure of expressions is recursively defined and can be deeply nested.</td>
<td></td>
</tr>
</tbody>
</table>

### 7.1 Programs

Structure at the *program* level depends largely on the kinds of top level declarations permitted by programming language. In a language such as C, program structure is limited to a list of struct, union, variable and function declarations, but does not include the definition. This is the level at which Syntax-directed editors, such as EMACS [LEWI98] work. At this high level, there is something to be said for a structure editor approach in which the editor actually enforces the structure of the grammar by modeling structural elements of the language. The commercial product Mathematica and the Cornell Synthesizer[REPS89] are examples of structure editors for mathematical formulae and programs respectively.

### 7.2 Statements

Structure at the *statement* level is more or less equivalent to a sentence in NL, since both represent a complete syntactic and semantic unit at roughly the same level – i.e., components of a sentence or statement do not usually express a complete thought. Sequences of sentences and statements appear in higher level constructs, such as paragraphs or functions, respectively, but aside from the environment inherited from those higher level constructs, programming language statements are grammatically independent of those higher level entities.

### 7.3 Expressions

Structure at the *expression* level is characterized by operator and function invocations. Expressions are often parsed by an operator-precedence parser in compilers. This is the level at which semantic
(i.e., type) information can have a substantial impact on syntactic analysis.

8 Practical Diagnosis

Ideally, the diagnosis provided by an IPDE should be both correct and complete. That is, we would like every diagnosis to be correct, and we would like every error to have a diagnosis. Unfortunately, achieving such a goal would require knowledge of the programmer’s intent; in any case deterministic diagnosis is known to be intractable in the general case.

As a practical matter, it is not necessary that the IPDE always provide a complete and accurate diagnosis. The only absolute requirement is that a correct program have a correct and unambiguous parse. The number of components in a $\Delta$ set can be constrained to a relatively low number, since the diagnosis is ultimately intended to be comprehensible to a human programmer. For example, if an error is so complex that it cannot reasonably be diagnosed, it is likely that it is too complex to be useful to a human. Reporting the component dominating the error as erroneous may be as, or possibly more, useful than a more detailed diagnosis. In the extreme, if no such component exists and none can be inferred, the whole of the input must be considered erroneous. I expect, however, that isolated errors in a mostly correct program can be diagnosed with sufficient accuracy for practical purposes. Some work on error attribution in programming languages has been done [WAND89].

Explaining an error—that is, explaining the difference between observed and expected behavior—requires some idea of what constitutes expected behavior. The system being diagnosed is, of course, the program being edited. The expected behavior of the system is that it conform to the grammar specified by a programming language. Therefore, the diagnosis is the minimal set of modifications of the input in order to produce an acceptible input. Similarly, a diagnosis can be described as the minimal set of modifications to the grammar that are required to produce a grammar accepting the input.

This approach to diagnosis builds a forest of parse trees (or a parse dag) consisting of a set of partial parses. It then inserts or removes components to create a complete parse. If the grammar specifies components in terms of other components (as CFG grammars do) the granularity of the modifications (and components) is critical—deleting the entire program in a single modification not a useful "correction."

The job of diagnosing an error becomes one of building and maintaining a minimal acceptable parse for a given input. That is, inserting, removing, or modifying elements in a derivation in order to maintain an acceptable parse. The job of identifying a minimal acceptable parse has three components:

1. A strategy for creating a partial parse of the input.
2. A strategy for determining a minimal set of modifications: insertions, deletions or restructuring, that produces at least one valid parse. A strategy uses the rules of the grammar to synthesize a complete parse from a set of partial parses.
3. A distance function $D$ that determines the relative “goodness” of each possible parse and selects the one to use as the diagnosis.
9 Modification Metrics

The distance function $D$ provides a quantitative measure of the difference between the actual input and an input that is in the language. There are many possible distance functions. The distance function may measure the number of tokens, symbols, paths or AST nodes that must be added, removed or changed to explain the error, or it could be probabilistic, weighting common explanations more highly than unusual ones.

Given a context-free grammar with the following productions:

\[
S \rightarrow X Y Z \\
X \rightarrow 'x' \\
Y \rightarrow 'y' \\
Z \rightarrow 'z'
\]

consider the following inputs:

1. “xyz”, which is acceptable and produces a normal parse.
2. “xz”, which is not acceptable since it is missing the ‘y’
3. “xyyz”, which is not acceptable since it contains an extra ‘y’

In the last two cases there is a discrepancy between the input and the grammar. Normally, we look at this as an error in the input, but for the purpose of diagnosis, especially w.r.t Abductive Diagnosis, it is simply a discrepancy.

The diagnostic machinery can account for these discrepancies either by generating a minimal set of changes to the input to make it acceptable to the grammar, or by generating a minimal set of changes to the grammar that make the actual input acceptable. In the first approach the string “xz” is considered to have a hole in the input, while in the second approach, it is considered to have an interjection in the grammar. Similarly, the string “xyyz” can be interpreted either as a hole in the grammar or as an interjection in the input. Thus these two approaches can be considered duals of each other.

Parsing that is augmented with interjections or holes can always produce one or more correct derivations by modifying either the input or the grammar as it proceeds. Each derivation may have nodes that are produced by rules in the original grammar (resp. original input) or by rules (resp. input) generated by the diagnostic machinery. The latter are referred to here as phantoms: phantom nodes in the derivation, phantom rules in the grammar, or phantom symbols in the input.

9.1 Holes – missing lexical elements

When a portion of the text is not acceptable and the most appropriate correction is to open a gap in the actual input and fill it with additional symbols, the gap in the input is referred to as a hole. For example, in programming languages that use semicolons to mark the end of a statement, the parser may reach the end of a statement where the only acceptable input is a semicolon, but none is found. Inserting a missing semicolon is an example of filling a (terminal) hole. It may happen that one of several terminal symbols could fill the hole or that more than one symbol is required to fill the hole. In either case the hole can be filled with the nonterminal symbol dominating the possible missing constituents.

If the grammar is being modified rather than the input, a hole is corrected by creating a copy of the appropriate grammar rule in which the constituent not present in the input is removed.
9.2 Interjections – extraneous lexical elements

Where a portion of the text is not acceptable, but an acceptable parse can be obtained by eliminating one or more symbols from the input, the extraneous text is referred to as an interjection.

For example, the statement \( a[,x] \) contains a comma interjection. The situation is dual to the hole case, and deleting the symbol from the input or inserting into the grammar forms a valid diagnosis. In the example, the corrected input would be \( a[x] \) and the corrected grammar would include a rule such as \( ARRAY\_REF \rightarrow IDENTIFIER: [ ",\,VALUE]\)

9.3 Substitutions

Where a portion of the input is not acceptable but an acceptable derivation can be obtained by replacing a lexical element (or non-terminal) with a different lexical element, the change may be applied as a replacement. This is equivalent to the insertion of a hole and removal of an interjection, but it is convenient to consider it as a single operation.

Part IV

Examples of specific errors

This section presents some examples of errors that may occur in programs. Each example is accompanied by one or more grammar rules that partially match the input; followed by examples of grammar and input modifications that diagnose the error.

In what follows I will try to distinguish "modifications", which are changes made by the IPDE, from "edits", which are performed by a human.

10 The Examples

Each example shows a single error within an otherwise correct program and shows how the rules of the original grammar may be modified to produce a grammar accepting the actual input as well as how the input may be modified to correct the error. The language of the examples is roughly C, although some use C++. The examples also assume that changes to the grammar do not affect any other part of the parse.

10.1 Missing input

Actual Input: \( \text{if ( )} \{ \text{foo()} ; \} \)
Grammar Rule: \( \text{STMT} \rightarrow \text{if ( EXPR )} \{ \text{STMT\_LIST} \} \)
Input Edit: \( \text{if( EXPR ) then} \{ \text{foo()} ; \} \)
Grammar edit: \( \text{STMT} \rightarrow \text{if ( )} \{ \text{STMT\_LIST} \} \)

The input matches the rule except for the missing \( \text{EXPR} \). The input can be accepted by the grammar by inserting a phantom \( \text{EXPR} \), which acts as a place-holder for the expected constituent. The grammar can be modified to accept the input by creating a copy of the rule that omits the \( \text{EXPR} \) symbol. Each of these modifications involves inserting or deleting a single non-terminal.
10.2  Interjection

Actual Input:  
if a ( a ) { foo(); }  
Grammar Rule:  
STMT \rightarrow if (EXPR) \{ STMT\_LIST \}  
Input Edit:  
if ( a ) { foo(); }  
Grammar Edit:  
a) STMT \rightarrow if (EXPR) \{ STMT\_LIST \}  
b) STMT \rightarrow if a (EXPR) \{ foo(); \}

The input matches the rule except for the extraneous a, which has wandered into the statement following the if keyword. The input modification simply removes the errant token; the grammar modification creates a new rule that includes the extraneous token.

10.3  Ambiguous missing or extraneous input

Actual Input:  
a++b  
Grammar Rules:  
a)EXPR \rightarrow EXPR ++  
b)EXPR \rightarrow ++ EXPR  
c)EXPR \rightarrow EXPR + EXPR  
Input Edits:  
a) BIN\_OPER ++ b  
b)a ++ BIN\_OPER b  
c)a + b  
Grammar Edits:  
a)EXPR ++ EXPR  
b)EXPR +  
c)+ EXPR

The input is unacceptable and the intent is unclear. Does the programmer intend a post-increment of a, a pre-increment of b, or is one of the + symbols an interjection? The input can be modified to remedy the situation by inserting any binary operator symbol either preceding or following the ++, or by removing one of the + symbols.

Since the ++ is a single token, replacing it with a + symbol could be considered a lexical modification rather than a grammar modification. Thus it is not clear that the grammar should be edited at all. Nevertheless, there are grammar modifications that can resolve the problem.

10.4  Multiple Related Holes (missing parenthesis)

Actual Input:  
if a == b { x = 0 }  
Grammar Rule:  
a) STMT \rightarrow if (EXPR) \{ STMT\_LIST \}  
b1) STMT \rightarrow if PARENEXPR \{ STMT\_LIST \}  
b2) PARENEXPR \rightarrow (EXPR)  
Input Edit:  
if (a == b) { x = 0 }  
Grammar Edit:  
STMT \rightarrow if EXPR \{ STMT\_LIST \}

The correction using grammar rule a is to insert the missing parentheses, or to modify the grammar as shown. The question is how to describe the distance of this modification: Is it one insertion or two? Using the (admittedly contrived) grammar rules b1 and b2 raises a slightly different question. The statement has a hole (the missing PARENEXPR with respect to rule b1) and a partial parse (the expression a == b). The correction is to insert a new node in the parse tree that glues the two together. If the parse tree is in fact an AST, the terminal symbols are irrelevant, only the intermediate non-terminal is required.
10.5 Multiple interjection or transposition

Actual Input: ( x y + )
Grammar Rule: a)EXPR → EXPR + EXPR
           b)EXPR → IDENTIFIER
Input Edits: a) ( x )
           b) ( y )
           c) ( x + y )
           d) ( x Bin OPER y )
Grammar edit: a)EXPR → IDENTIFIER IDENTIFIER BIN OPER
             b)EXPR → x IDENTIFIER *
             c)EXPR → x y +

A variety of edits are possible here, for example: ignore the interjection of the '+' and one of the variables, transpose the '+' and the y, or insert a non-terminal representing a binary operator between the x and y and ignore the plus symbol. Each of these options requires two edits: two deletions, a deletion and an insertion, and an insertion and deletion respectively.

This example indicates that a simple error can give rise to a number of possible corrections and that selecting the appropriate diagnosis may require information that goes significantly beyond the grammar of the programming language.

10.6 Lexical Error

Actual Input: int f oo();
Grammar Rule: FNDEF → TYPENAME IDENTIFIER ( PARAMETER_LIST )
Input Edits: a) int foo();
           b) int f();
           c) int oo();
Grammar edits: a) IDENTIFIER → f oo
              b) FNDEF → TYPENAME IDENTIFIER IDENTIFIER ( PARAMETER LIST )

While the intended meaning of this input is ambiguous, if it happens that 'foo' is in the lexicon (i.e., symbol table) and neither 'f' nor 'oo' is, then 'foo()' is the most likely meaning and therefore the most appropriate diagnosis. The error in this case is the interjected space between the 'f' and 'o'. It is also possible that the 'f' is intended as a type modifier, as in int * oo(); .

Input edits include deleting the space between the two tokens (thus merging the two identifiers into one), and deleting one or the other of the identifiers. Each of these edits involves eliminating one token (or whitespace, which is not normally tokenized). Alternatively, the first identifier 'f' could be replaced with another token, say, 'a*'. (If the language permitted, 'f' could even be added to the symbol table as a new type modifier.)

The grammar can be modified by either inserting a rule that accounts for the two identifiers by treating them as a single identifier, or by modifying the grammar rule for function declaration by permitting the additional identifier.

This error could also be corrected at the lexical level by combining the two unknown identifiers to produce one acceptable identifier. Unfortunately, there is no lexical error; both f and oo are valid tokens. This case illustrates a grammatical error for which the most likely diagnosis is lexical. This kind of interplay between lexical and grammatical levels argues for a tight coupling between lexical and grammatical analysis.
10.7 Name resolution/type coercion

Actual Input: \( x = b.\text{foo}(0) \)
Correction: \( x = b.\text{foo}((\text{char }*) 0) \)

Assuming \( b.\text{foo}() \) is overloaded for both int and pointers (as occurs in C++, for example), the input may be semantically ambiguous. Most compilers do not even look for type errors in a program that contains syntax errors. But if type information is incorporated into the grammar this ambiguity can be resolved during parsing. Since natural languages are so ambiguous, Natural Language parsing commonly incorporates type information (Part of Speech Tagging, for example) during parsing in order to avoid creating derivations that will ultimately be fruitless.

If, however, the argument ‘0’ has a type feature representing the principal typing of ‘0’, and the function \( b.\text{foo}() \) has a similar feature, this can be resolved in the parser by unifying the type features. This example illustrates the value of incorporating type information into the grammar.

The type systems of real-world programming languages are complex: including coercion, overloading, inheritance, dynamic dispatch and other forms of ad-hoc polymorphism. C++’s templates add a primitive form of parametric polymorphism (and the next version of Java will support true parametric polymorphism).

For example, if the grammar is lexicalized such that the two categories \( b.\text{foo}(<\text{integer}_\text{value}>) \) and \( b.\text{foo}(<\text{char*}>) \) are both visible in the symbol table, then an actual argument of type \( \text{integer}_\text{value} \) will unambiguously match only the first category.

10.8 Lexical replacement

Actual Input: \( \text{Int \ foo}(); \)
Grammar Rule: \( \text{FUNC.DECL} \rightarrow \text{TYPE.NAME IDENTIFIER ( PARAM.LIST )}; \)
Input Edit: \( \text{int \ foo}(); \)
Grammar Edit: \( \text{FUNC.DECL} \rightarrow \text{Int IDENTIFIER ( PARAM.LIST )}; \)

If no type name "Int" exists, it is reasonable to replace the token with a lexically similar token, which invites the use of heuristic distance measures. This is not a grammatical error, so it’s inappropriate to resolve it by grammar editing. On the other hand it is the grammatical context that provides the crucial information for diagnosis, i.e., that the name must refer to a type.
10.9 Control Flow Error

Actual Input:
```c
bool f()
{
    return true;
    return false;
}
bool f(bool c) {
    if (c)
        return true;
    else
        puts("c");
}
```
Correction:
```c
bool f()
{
    return BOOL_VALUE;
}
bool f(bool c) {
    if (c)
        return true;
    else
        puts("c");
    return BOOL_VALUE;
}
```

Grammar:
```
FUNCTION → TYPE_NAME NAME { RETURN_STMT LIST }
PROCEDURE → void NAME ( PARAM_LIST ) {_STMT LIST }
RETURN_STMT → if ( EXPR ) { RETURN_STMT_LIST }
RETURN_STMT_LIST → return EXPR OR_STMT LIST return EXPR
```

Control flow errors are normally handled during the semantic analysis phase of a compiler.

Some control flow requirements, such as the required return statement in non-void functions, could be specified by CFG rules. Usually, they are not; apparently the only obstacle to doing so is the increase in the complexity of the (LR or LL) grammar. Presumably a more expressive grammar could more easily incorporate this and similar cases.

Some instances of unreachable code can be also be detected syntactically,

10.10 Data flow errors

Actual Input:
```
int x;
++x;
```
Correction:
```
int x;
x = 0;
```
Grammar example:
```
NAME_INIT → NAME = EXPR
EXPR → NAME_INIT;
```

In this example, the variable x is used before it is initialized and it is assigned without having been used since the previous assignment (++x is an assignment). This seems to be context sensitive, but
I'm not so sure—it may be feasible to extend a grammar to distinguish initialized and uninitialized variables.

Some data and control flow errors are difficult to detect. The following example contains both a control flow and data flow error:

```c
int f(bool c) {
    if (c)
        return 0;
    else if (c)
        return 1;
    else return 2;
}
```

The unreachable code in this example is more difficult to detect and, in fact, compiles without error on commercial compilers. (A combination of common subexpression, data flow and control flow analysis could presumably detect it.) It is possible, however, that by performing error diagnosis concurrently with editing, a sufficient number of idle cycles will be available to attempt more aggressive error analysis.

### 10.11 Unbalanced Delimiters

**Actual Input:**

```c
z = x * ( z + 1);
```

**Correction:**

```c
z = x * ( z + 1);
```

**Grammar edits:**

```
EXPR → EXPR )
```

Either insert a left parenthesis or remove a right parenthesis

### 10.12 Garbage

**Input:**

```c
) + & fee [fie ; fum
```

**Correction:**

```c
( EXPR ) + &fee[fie]; fum;
```

**Grammar edits:**

```
EXPR → )
ARRAY_REFERENCE → fee[fie
```

It may happen that a surprisingly simple diagnosis can be found for complete garbage. This may indicate that aggressive diagnosis is a bad idea, and that beyond some (relatively low) distance threshold, the diagnosis should simply be an error in the node dominating the problem.

### 11 Comments on Examples

These examples indicate that the interplay among lexical, syntactic and semantic levels is likely to be a significant factor in providing a good diagnosis. Natural Language researchers have long recognized the importance of cooperation among the three levels in resolving ambiguities. Therefore, one objective of future research should be to more closely couple the analysis of these three levels; perhaps using the same grammar model to describe—and the same mechanism to analyze—all three levels.
These examples also indicate that a single error may have alternative corrections, each of which require the same number of edits (insertions, deletions, etc.). Thus it is essential to develop a “goodness” metric that can select the best of several possible diagnoses and it is likely that such a metric will have to rely on semantic and/or statistical knowledge about programs and common programming errors.

12 Input-modification vs. Grammar-Modification

The diagnosis determined by either the Input-modification or Grammar-modification approaches depends on the actual distance metric used to select a phantom parse. Thus the question of whether the two approaches produce the same result reduces to whether an input-edit metric and grammar-edit metric are equivalent (according to some, as yet undefined, equivalence class). One difference between the two approaches lies in the fact that the input-edit approach strongly associates each error with a specific location in the input, while the grammar-edit approach produces a diagnostic that need not be tied to any single location.

For example, consider the use of an undeclared identifier. The grammar-edit approach may insert a single rule that serves for all references to that identifier. Nothing requires that the edit be associated with a specific position in the text, although additional data structures that map errors to text positions can be easily provided. Similarly, the input-edit approach evaluates each appearance of the identifier as a separate error. (Again, auxiliary data structures can be used to link the separate instances together).

12.1 Requirements of an Edit-Distance Metric

The preceding examples and observations indicate that the burden of robust diagnosis falls squarely on the distance metric used to select the best of alternative diagnoses. That is, Reiter's minimum-component set \( \Delta \), is the set of edits with the minimal edit-distance.

The edit-distance function (or simply distance function) \( \delta \), provides a numeric value that indicates the “distance” between the actual input and an a “corrected” input, or, in the case of grammar modifications, between the actual grammar and a corrected grammar. That is, \( \delta(A, B) \) must calculate a value \( \phi \), such that the most likely rule will have the lowest value of \( \phi \). In the case of syntactically correct input, \( \phi = 0 \). A and B are two input texts (the actual input and a modified input) or two grammars (the actual grammar and an modified grammar).

Since \( \phi \) is a metric \( \delta \) it must satisfy the following properties:

1. \( \delta(A, A) = 0 \)
2. \( \delta(A, B) \geq 0 \)
3. \( \delta(A, B) = \delta(B, A) \)
4. \( \delta(A, C) \leq \delta(A, B)) + \delta(B, C) \) (where + is an associative additive function).
Part V
Edit Distance and Diagnosis

13 Granularity of Diagnosis

String edit distance usually refers to editing characters within a string; but in the context of a program editor that may not be the ideal unit of input with which to operate.

For lexical errors, which are often simple typing errors, individual characters are clearly the appropriate level of granularity. Terminal symbols (lexical tokens) and non-terminal structures (such as chart or tree nodes) are the fundamental elements of syntactic diagnosis.

14 Distance Metric

A distance function evaluates the “goodness” of a proposed diagnosis, but just what is a “good” diagnosis? A function that always correctly guessed the programmers intention would be ideal, but such a function is not likely to be found anytime soon. This section explores a few variables by which a distance function could be parameterized in order to produce a metric that reflects our intuition about a good diagnosis.

14.1 Height

The importance of each node within a parse tree is not necessarily uniform. That is, replacing a node near the top of the tree could be a more weighty edit than changing a node near the bottom of the tree, since the higher node covers a greater span of the input and may subsume a number of lower level edits.

14.2 Type of Modification

The basic modification functions are insertion, deletion and replacement (of characters in a string or nodes in a tree). It may be useful to provide other basic operations, such as transposing adjacent lexical symbols, splitting a token or merging two tokens. Each of these operations can be weighted so that those operations that tend to provide better diagnoses are applied in preference to others, all else being equal.

14.3 Symbol Table Edit

Each identifier in a program is stored as a new entry in a symbol table when it is first encountered. Unfortunately, misspelled identifiers are also stored as a new entry in the symbol table, even if they only appear once (with that misspelling) in the program. Since typographical errors are very common, it may be worthwhile consider operations on the symbol table in computing the distance function.

15 Probabilistic Methods

It is certainly possible to use statistical methods for parsing, and to select a diagnosis based on previous errors. But that is beyond the scope of this study.
16 Free Grammar-Edit Operations:

It editing a grammar, the edit distance of changes to the grammar itself may not be indicative of the consequent changes to the language accepted by the grammar. Small changes may cause profound changes in the language accepted by a grammar, and substantial changes may have little effect. In fact many modifications have no effect whatever. For example, given a grammar G, duplicating a rule φ does not change L(G). Similarly, substituting the right hand side of a rule for its left hand side in a rule does not change L(G). The distance metric for grammar-edit diagnosis is perhaps better viewed as an edit-distance on the languages accepted by those grammars.

16.1 Static Grammar Distance

Calculating general edit distance can be intractable unless the distance metric is constrained as in k-edit distance over strings. Fortunately, that limit is already implicit in the grammar. Given two symbols, A and B, in a CFG, there is a static edit distance between those two symbols D(A, B), which is strictly a function of the grammar. D(A, B) is finite since the grammar is finite. The largest static edit distance between the start symbol of the grammar and any other symbol is, essentially, the height of the grammar. This static distance places a limit on the maximum edit distance that will be investigated by the distance function. This limit is useful for implementation, since it bounds the maximum error and prevents the parser from investigating a possibly exponential number of alternatives.

16.2 Complexity

Several variations of LR parsers that can handle ambiguous grammars have been described in the literature, the best known is probably [TOMI86], although Aho and Peterson describe a parser [AHO72], which is capable of minimal distance error correction. These parsers operate in worst case O(n^3) time, but can operate in Θ(n) time on unambiguous input.

Grammars that are not context free have higher time complexity. Parsens for the mildy context sensitive grammars generally operate in O(n^6) time. Unification parser can take exponential time, but usually performs much better in practice.

While polynomial complexity may be prohibitive for a batch oriented parser, in an (incremental) partial parser such complexity may not be a problem since the work is interleaved with human editing of the input.

Part VI
Related Research

17 Cox and Clarke

Cox and Clarke [COX] use sets of hierarchically related regular expressions to analyze a stream of tokens and to build as complete an Abstract Syntax Tree as possible. Their analyzer consists of five such sets (not counting tokenization) where each set is applied in sequence. The first step categorizes tokens by assigning them to equivalence classes such as ‘type’ or ‘assignop’. The second step identifies and marks declarations. The second and third steps locate expression components, and the last step recognizes statements. The entire process is supported by a symbol table.
While Cox and Clarke describe their analyzer as 'lexical', the presence of a stack and the use of nested regular expressions forms a kind of recursive transition network with the power to recognize context-free languages. The presence of a symbol table implies the ability to handle context sensitive constructs as well. Their parser is probably better described as a partial parser. It is, in any case, likely to be a more robust builder of ASTs than traditional programming language parsers.

Their analyzer also incorporates heuristics. For example, recognizing the C statement \texttt{index (x)}; in which \texttt{index} may be either a function name or type, as a function call since redundant parentheses around a variable in a declaration are rare.

They compare the accuracy of their approach to that of the PAN analysis tool, which they assume correctly identifies program structure, using syntactically correct C programs. (The PAN tool assumes syntactically correct input and uses a traditional shift-reduce parser). The two approaches perform comparably at the level of statements and declarations, but Cox and Clarke's approach performed poorly in analyzing expressions. Nevertheless, they conclude that "lexical analysis is an effective technique for generating a sufficiently detailed approximate model".

Cox and Clarke's motivation is essentially the same as mine, but they do not address the issue of partial parsing directly, nor do they provide any indication of how well their tool works with incomplete or erroneous text. The difficulty of their approach lies precisely in a lack of lexical information available for disambiguation. Providing lexicalized semantic and syntactic information (including polymorphic type) would probably improve the accuracy of the analysis.

Cox and Clarke do not provide examples of the regular expressions they use. It would be interesting to see whether there is any correspondence between the structures represented by these hierarchical regular expressions and the structures represented by initial TAG trees, or CCG categories.

18 Aho and Peterson's Error Correcting Parser

Aho and Peterson [AHO72] describe an error correcting parser that uses a simple character edit cost metric. They adapt a CFG to include additional, non-deterministic "error" rules so that the resulting grammar accepts \( \Sigma^* \). The grammar is parsed by a modified Earley parser that maintains a cost feature, which keeps track of the number of error productions used. A separate program extracts a minimum cost parse tree from the completed chart. They only address syntax errors in an LR grammar. Their program operates in \( O(n^3) \) time overall.

19 The Cornell Synthesizer Generator

Thomas Reps and Tim Teitelbaum [REPS89] developed the Cornell Synthesizer Generator (CSG), which builds a language specific structure-editor from an attribute grammar specifying editing rules for the language. The resulting editor defines templates for various constructs, for example:

\texttt{while ( expr ) stat ; \{expr=boolean\}}

and permits the insertion and deletion of a template as an atomic operation. For example, a 'while' editor command produces the following

\texttt{while ( [expr:bool] ) [stat] ;}

and allows the programmer to fill in the placeholders [ ] with other template structures or values. (Mathematica is an example of a structured editor). The effect is to ensure that the structure of the input is always valid. The editor can also perform type-checking, however it does not appear to attempt any error diagnosis.
The authors note that the editor may permit free-text entry, but once a template has been filled, (i.e. a constituent parsed) subsequent editing must conform to the traditional structure based editing.

The authors also note that their use of an attribute grammar may not be scalable, since it can require traversal of large portions of a parse tree.

The proposed scheme differs from the CSG system primarily in that it places no editing restrictions on the user. The scalability problem is not as acute, since this scheme intends to distribute the computation incrementally over the entire time the program is being edited.

20 Harmonia, Ensemble and PAN

A group led by Susan Graham at U. C. Berkeley has created a software development environment with objectives similar to those described in this paper. Their current work, Harmonia is a framework for the development of interactive tools and is an extension of their previous work, ensemble which provides an incremental lexer, parser and semantic analyzer. Ensemble was derived in turn from their work on PAN. Harmonia is intended as a language independent workbench. It relies on the Ensemble parser, which can be customized to support a particular programming language. Ensemble uses a Generalized LR parser, based on Tomita’s, to (re)parse the region of text spanned by any rule that is affected by the change. Harmonia also incorporates versioning of edited text and thus can fall back to previous versions as a means of error correction.


Part VII
Conclusion

The purpose of this study has been to explore the issues associated with error recovery and analysis in the context of a program editor. The study has perhaps raised more questions than it has answered, which indicates that the development of a program editing environment that can interactively diagnose many errors is an interesting and viable research topic.

21 Summary

1. The errors that appear in programming are not neatly divided between lexical, syntactic and semantic layers. Combining these—allowing the parser to contribute to lexical analysis and allowing semantic information to play a role in parsing—could greatly improve a program editor’s ability to analyze a program for errors.

2. The Natural Language Processing community has developed grammar models that are far more expressive than the restricted context free grammars used by compilers. Given a more expressive grammar most, if not all, ‘semantic’ issues can be folded into the syntactic description of the language.

3. Uniting partial parses into a single tree requires exploration of many possible parses. Thus it is important that the parser be able to abandon possible ambiguous parses, perhaps using stochastic means.

21
4. Simple typographical errors are quite common, thus it is important that the program editor provide diagnosis for such errors.

22 Future Research

Among the possibilities for additional research are:

1. The development of a lexicalized grammar (and parser) for a real-world programming language.
2. The implementation of an editor and empirical evaluation of distance metrics.
3. Analysis of control flow errors, data flow errors, type error attribution, etc.

References


[REIT77] Reiter, R., A theory of Diagnosis from First principles, *Artificial Intelligence 32* (197?) p. 57


[STEE00] Steedman, Mark; *The Syntactic Process*, MIT Press Cambridge, MA 2000


Part VIII
Appendices

A Desirable IPDE Features

This appendix lists possible features of the proposed IPDE. The intent is to justify the value of performing incremental analysis of the program at edit-time. An environment that provides such analysis could:

- Detect and Diagnose Syntax errors by actually parsing the text within the editor, syntax errors can be detected and corrected immediately, when they occur, rather than at some later time when the program is compiled.

- Provide Type Inference
  Type inference uses unification to determine the type of a variable automatically from the way in which it is used. Type inference is fundamental to functional programming languages. It is also useful in languages that require explicit type declaration, since it is not uncommon to introduce names before they are defined. The expected benefits of this feature are that it helps a programmer to better understand an unfamiliar program and to avoid introducing type errors. When type inference finds no acceptable type for an identifier, or if the inferred type conflicts with the declared type, the IPDE can highlight all occurrences of the identifier and provide details of the error, e.g., in a fly-over window.

- Diagnose Type Errors
  Even without type inference the IPDE can invoke a more appropriate recovery strategy and thus use a corrected type for subsequent analysis.

- Diagnose Control Flow Errors
  Construction of a control flow graph at edit time enables detection and diagnosis of control flow errors such as mismatched if-else or switch-case statements, incomplete case statements, unreachable code, missing return statements, etc.

- Provide Control Flow Information
  For example, the IPDE can provide a way for the programmer to highlight all control statements that affect whether a selected statement is executed or not; or, highlight all statements whose execution is determined by a selected control statement.

- Diagnose Data Flow Errors
  Given a control flow graph, it is possible to detect and diagnose data flow errors, such as Use of uninitialized variables, useless assignments, etc. This can be augmented by additional dependency notations, as, for example, in ADA Spark[BAR97].

- Provide Data Flow (use/def) Information
  For example, the IPDE can provide a way for the programmer to highlight all uses of the selected assignment to a variable, or highlight all assignments reaching the selected use of a variable.

- Provide Slices
  Slices show all program points that may affect a selected statement or variable. The IPDE
can also list entities in other modules that may affect or be affected by the selected statement or variable.

- **Provide Dices**
  Dices are similar to slices, but show the program points that may be affected by an error.

- **Interactive Execution**
  Once semantic checks have been performed, any valid portion of the AST can be compiled or interpreted. In the context of an editor, it is reasonable to execute a portion of a program, since the user can interactively provide values for undefined variables or functions.

- **Source-Level display of optimized code**
  As one progresses from source code to executable it becomes increasingly difficult to relate portions of the intermediate representations back to specific locations in the source code. Various optimizations (e.g., code-motion, loop optimizations, tail-recursion elimination) can alter the code considerably. Consequently, using a source level debugger to debug optimized code can be somewhat confusing. However, many of the optimizations can be performed on the source code directly; thereby enabling the programmer to see the source code corresponding to the optimized code. Allowing the programmer to see the optimizations in the source language might clarify the actual operation of the program.

- **Incremental Compilation**
  The subject of incremental compilation has already received considerable attention. While I'm not interested in the topic *per se*, it seems that having built the data structures necessary for semantic analysis, it is not a big step to generate code.
B Notes on Practical Requirements

The following is a rumination on the question of whether the space and time requirements for
edit-time parsing of a program are realistic.

The key to making edit-time parsing practicable lies in the development of an efficient, robust,
and incremental parser. That is, when a change is made to the program, the parser processes
only the portion of the derivation tree that must be changed—and does so whether the change is
grammatical or not.

Locality of reference is more important than the number of operations. In part, this reflects the
fact that CPU speed no longer dominates overall processing time for most programs—a single cache
miss can result in hundreds of lost CPU cycles. But more significantly, by building and maintaining
all of the data structures incrementally the time requirements are effectively amortized over the
duration of the editing session.

There are several data structures associated with parsing:

1. A character string representation of the program, which is very compact since it contains no
   information about derivation.

2. A parse chart, which contains information about all possible parse structures, is a dynamic
   programming solution for the problem of examining a potentially exponential number of
   possible parses. This data structure can become very large, since it contains many partial
derivations that may never produce a complete derivation.

3. An Abstract Syntax Tree (AST), which represents a complete derivation. In the case of
   ambiguous derivations, the tree may be a dag. Each node of the tree represents a complete
   semantic entity. ²

   The parse chart is, of course, generated from the character string representation and the AST
   is extracted from the chart—that’s what the parser does. It is equally obvious that the string
   representation can be produced from either the AST or the chart (and a symbol table). This leads
to the following observation:

   Observation: Any one of these structures can be generated from the other.

1. The normal parsing process generates a parse chart from a string and an AST from a parse
   chart.

2. A string representation of the program string can be generated from an AST by a traversal of
   the tree. The exact string from which the chart and AST were generated can be reproduced
   by storing additional nodes, such as white-space and comment nodes in the AST.

3. A parse chart can be generated from an AST by simply reproducing the original program
   string and re-parsing the string.

Thus any of these three representations of a program can be produced from any other. In
fact, within a single program some portions of the program can be stored in the editor using one
representation, while other portions of the program are stored using other representations. Only

²Semantic in the sense of denotational semantics
those portions of a program that are actually being modified need be represented as a parse chart. A portion of a program that has been fully parsed, for example, may be stored in the editor as an AST until the user begins to edit it. There is no need to keep the chart for that portion of the program. The existence of an AST implies the existence of a complete (though not necessarily error free) parse. If that portion of the program is changed, only the portion of the chart related to the the change need be re-created. For those parts of the program that are semantically correct, the chart is not necessary, since the AST contains the one correct derivation (programming languages are always semantically unambiguous).

B.1 Practical programs

Most program development and maintenance efforts these days are not well organized\(^3\).

Experience suggests that programs developed in such an environment tend to be large, poorly structured, and difficult to understand. An individual compilation unit may contain thousands, even tens of thousands, of lines of code. Typical compilation units generally contain several hundred or a few thousand lines of code.

Consider a program containing 10K lines of source code. Based on an informal inspection of a few C programs that translates to about 300KB of source code, and roughly 60K tokens. Making the wild assumption that an AST for that program might contain, say 60K nodes, and each node occupied 200 bytes of memory, the AST would take up 12 megabytes. While that is a substantial amount of memory, a typical programmer’s workstation contains hundreds of megabytes of memory. So the size of the AST is practicable, even for a rather large source file.

B.2 Editing Focus

The space requirement for a parse chart is even greater than that of the AST. A CYK style parser contains \(n(n-1)/2\) cells, thus a program containing 60K tokens would produce a chart of some \(36 \times 10^8\) (3.6 billion) cells. Such a chart would require gigabytes of storage and would be too large to fit in the memory of current workstations.

But since the chart is only required for the portion of the program actively being edited, the entire chart need never be present. Programmers typically write (and re-write) no more than a few hundred lines of code a day and tend to focus on a only small portion of a program at one time. A chart that covers, say, 20 lines of code would thus require only 200 cells, which is quite manageable.

B.3 Responsiveness

If one types at a speed of 60 words per minute (which is rather fast) the duration between keystrokes will be about 166ms (on a 1GHz machine, that’s about \(10^8\) cycles). Programmers do not generally type at such speeds for any sustained length of time. They occasionally have to stop and think.

The time required to parse the changes and update the data structures for a few hundred lines of code over the course of a day is clearly within the capabilities of a typical programmer’s workstation. (Indeed, if the scope of the change is small (i.e., if \(n\) is small), it may well be possible to consider \(O(n^3), O(n^6),\) or even \(O(2^n)\) algorithms).

The issue in an interactive environment is responsiveness, rather than CPU time. Each keystroke does not entail the same amount of processing. In pathological cases, single keystroke may entail

\(^3\)The vast majority of organizations developing software are at or below CMM level 1, which is usually described as “ad-hoc” or “chaotic”. See, for example, *Software Assessments, Benchmarks, and Best Practices*, by Capers Jones; Addison-Wesley Pub Co, 2000.
re-parsing of the entire program. In practice, I expect that sort of thing to be rare, since large programs tend to have rather flat derivations resulting from the declaration of many separate classes or functions. In any case, it is easy enough to modify the chart and AST in a low-priority thread or to use a sort of systolic4 approach to updating those structures.

B.4 Conclusion

The amount of work that must be performed by an incremental parser may be substantial, and the time and space complexity of an appropriate incremental parsing algorithm may be demanding. However, I believe it is reasonable to expect that, with careful implementation, it is computationally practicable.

\footnote{i.e., force the parser thread to yield after each step of a BFS update of the derivation.}