Initial experience with noncorrecting syntax error handling

Arthur van Deudekom
Dick Grune
Peter Kooiman

Department of Mathematics and Computer Science
Vrije Universiteit
De Boelelaan 1081
1081 HV Amsterdam
Tel: +31 20 548 5778
dick@cs.vu.nl

ABSTRACT

Noncorrecting error handling is a technique which, when implemented with some caution, guarantees fully reliable syntax error messages. This paper describes a simple but efficient implementation and its incorporation in an existing top-down parser generator. Initial experience with the generated parsers is reported. Two points which require special attention are identified and discussed: handling grammars that are not completely context-free, and obtaining informative error message texts. The implementation is general enough to be incorporated in any parser generator.

Additional keywords: general context-free parsers, suffix parsers, syntax error testbeds

1. Introduction

Compiler users and compiler writers take parsers for granted these days, and rightly so. Parser construction is a mature science and a variety of efficient parsers, guaranteed not to loop or crash on any input, can be found in the pertinent literature. The same cannot be said about syntax error handling. Syntax error handling is still very much an art, mostly black, and users of 'professional' compilers such as GNU's *gcc* or Sun's *cc* are still used to see a single syntax error result in streams of error messages. This is surprising, since already in 1985 Richter[1] showed a method that guarantees that each syntax error will result in at most one error message, the so-called *noncorrecting error handling*. More precisely, the first syntax error is always reported and subsequent syntax errors are almost always reported; no spurious error messages occur. His method has drawn little response from the compiler construction community, the work by Cormack[2] being the only reaction known to us.

Noncorrecting parsers are potentially more useful than correcting ones, during program debugging. Correcting parsers use some heuristics to correct the error, which allows them to continue; noncorrecting parsers use a cleaner approach, avoid heuristics and turn out to be less difficult to implement than some correcting methods. The goal of our paper is to evaluate the idea of noncorrecting error handling in a practical setting. The paper has three important contributions. First, it describes an actual implementation of a simple noncorrecting parser, embedded in a widely used parser generator system. Second, it gives insight into the problems that

arise if the parser generator supports grammars that are not really context-free (all real-world parser generators do so). Finally, the paper describes our initial experience with compilers featuring noncorrecting syntax error handling, and compares their error handling capabilities with those of traditional compilers.

We have implemented noncorrecting error handling in the LL(1) parser generator *LLgen* which is part of the Amsterdam Compiler Kit (ACK)[3]. To get a quick impression of the properties and effectiveness of the method, we have tested the resulting parsers on a testbed which generates programs with single token modifications from existing correct programs. An unexpected result was that although the method guarantees in theory at most one error message per syntax error, it can still produce more than one error message if applied naively to 'real-world' grammars.

The remainder of this paper consists of six sections. Section 2 introduces noncorrecting error handling. Section 3 gives an overview of *LLgen*. Section 4 describes the incorporation of noncorrecting error handling in *LLgen*. Section 5 shows the results of the testbed and discusses some problems and their solutions. Section 6 outlines the suffix recognizer needed for the noncorrecting error handling and gives some time measurement. Section 7 contains some preliminary conclusions.

2. Noncorrecting syntax error handling

Richter[1] describes how syntax error handling can be done without making any corrections to the input stream the parser sees. He gives two reasons why error handling that does make corrections is undesirable. There are often many possible corrections, each with different consequences, and it is difficult to choose a reasonable one; and the effects of the choice may propagate arbitrarily far into the rest of the input.

The noncorrecting technique described by Richter can be summarized as follows: When a syntax error has occurred, the input up to and including the erroneous token is discarded. The remainder of the input is processed by a *substring parser* of the input language, a parser that recognizes any substring of a string in the input language. When the substring parser detects a syntax error, the offending token is reported as another error, and the input up to and including the new erroneous token is discarded. The process is repeated with the remaining input, possibly finding other syntax errors, until all the input has been scanned.

In his paper, Richter points out that this process finds the right-most limits of the errors locations rather than their exact locations. To remedy this, he proposes a second, right-to-left scan, which will find the left-most limits of the errors locations, thus providing error location intervals. We found the performance of the normal left-to-right algorithm to be good enough and have not added the complexity of an additional right-to-left scan.

To implement the technique, a parser that parses substrings of the input language is needed. If we confine ourselves to syntax analysis, it is sufficient to construct a substring recognizer. Richter does not give a practical construction, but Cormack[2] describes how an LR substring parser can be constructed which handles BC-LR(1,1) grammars. We have constructed a top-down substring recognizer which can handle any context-free grammar. Furthermore, our recognizer is actually a *suffix recognizer*: it recognizes any suffix (tail) of a string (program) in the input language. Our suffix recognizer has the *correct-prefix property*, which means that it will stop as soon as the input seen so far is no longer a correct prefix of a suffix of a string in the language. As a result, it detects the next syntax error as early as possible in a left-to-right scan of the input. Because the suffix parser has this correct-prefix property, it can be used as a substring parser, as it will detect the first input token that is not part of a substring of the language. Because it is a suffix recognizer, it additionally will detect incomplete input, as in that case the parser will not be in an accepting state at the end of the input.

3. Overview of *LLgen*

LLgen[4] is an extended LL(1) parser generator. It is part of the Amsterdam Compiler Kit (ACK) and has been used in compilers for many languages, such as C, Pascal, Modula-2, Eiffel and Orca. It can actually handle grammars which are not purely LL(1), through the use of conflict resolvers. Semantic actions can occur anywhere in the grammar rules, and they are executed when their position is reached during parsing. A typical *LLgen* rule looks like

S: A { *action* } B;

where the *action* is a piece of C code which will be executed when the parser is using the rule for S and has recognized A. Since the parser is the main loop of the compiler, the actions embody, directly or indirectly, the rest of the compiler. We will call the actions the *semantic level* of the compiler, although it is actually a combination of contextual processing and code generation.

In grammars which are nearly −but not completely− LL(1) conflicts will arise in the two places where parsing decisions are made: the choice of which alternative to start (*alternation conflicts*) and the decision to stop or continue a repeated item (*repetition conflicts*). In order to allow *LLgen* to handle this type of grammar, the user can specify conflict resolvers in those places where conflicts arise. These resolvers are Boolean expressions labelling an alternative, and are evaluated when a conflict arises during parsing. If the expression evaluates to 'true' the labelled alternative will be taken. The Boolean expressions are expressions in C, and can consult any information available at the point at which they occur. As we will see in the next section, this feature makes it necessary for the suffix recognizer to handle grammars which are not LL(1).

LLgen allows more than one grammar rule to be designated as a start symbol of a parser and a separate parser routine is generated for each of them. Multiple start symbols are used, for example, for processing constant expressions in `#if` directives in C and for reading IMPORTed definitions in Modula-2.

Parsers generated by the original *LLgen* use *correcting* syntax error handling, based on a scheme designed by Röhrich[5], which inserts and/or deletes tokens at the point of error detection until correct input results. This means that actions in the parser will always be executed in an order that could also have resulted from syntactically correct input, and most importantly, once a grammar rule is started it is guaranteed to be completed. This in turn means that syntactic errors can never result in inconsistencies as to the actions: actions only have to deal with syntactically correct input. In a nutshell, the original error handling in *LLgen* parsers works as follows. We will first consider the case that the parser is presented with a prefix of a correct program: the input is correct except that it breaks off before the end. The error handling mechanism now provides a *continuation path*, a sequence of tokens chosen in such a way that all active rules are finished as soon as possible. Effectively, the continuation path is the 'shortest way out'. The tokens on this path are called *acceptable*; they always include end-of-file. Furthermore, at each point along the continuation path there can be other tokens that would be correct; these are acceptable as well. Now, when an error occurs and there is input left, input tokens are discarded until an acceptable token appears in the input. The tokens on the continuation path up to but not including the acceptable input token are inserted. The parser first consumes the inserted tokens and then the acceptable token. From then on, normal parsing resumes.

4. Incorporation of noncorrecting error handling in *LLgen*

An important consideration in incorporating the noncorrecting error handling in *LLgen* was that correct programs should suffer as little as possible in compilation speed. Furthermore, the existing error handling method has the desirable property that rules which are started are guaranteed to finish, thus ensuring that errors in the input text will not cause inconsistencies in the semantic actions. We have implemented the noncorrecting error

handling in such a way that this property is preserved.

We achieved these two goals by including the suffix recognizer as a second recognizer in the generated parser. Correct programs are handled in the usual way by the parser, but if an error occurs the following happens: instead of going to the standard error handling routine, the parser starts executing the noncorrecting error handler. The noncorrecting error handler does not execute any semantic actions; it continues, reporting all errors, until the end of the input text is reached. Then, control is handed back to the standard error handling routine. This routine will now think there is no more input, and thus start inserting tokens so as to construct the continuation path; error reporting is turned off during this phase. The inserted tokens lead the original parser to an accepting state, while the proper semantic actions are being performed. Thus all rules that were started are also finished, and no inconsistencies can occur in the semantic actions.

During noncorrecting error handling, we can no longer rely on the conflict resolvers to guide parsing decisions. The suffix recognizer is only concerned with syntax, and will not execute any semantic actions. As a result, the information that conflict resolvers could use will not be available, because the semantic actions that would build this information have not been executed. *LLgen* only requires its input to be LL(1) when the conflict resolvers are active; it imposes no restrictions on the *underlying grammar*, i.e. the grammar in which the conflict resolvers are ignored. Consequently, the suffix recognizer has to handle any context-free grammar; in particular, it has to be able to handle left recursion.

Ignoring the conflict resolvers will in theory allow some syntax errors after the first one to go unnoticed. Almost all conflict resolvers in an LLgen grammar, however, fall into one of two classes: those implementing a local form of LL(2), for example to distinguish between the labelled statement ‘`a: b`’ and the expression ‘`a = b`’; and those disambiguating ambiguous sections of the grammar, as in the dangling else problem. In both cases the underlying grammar still describes the correct language, so no syntax errors are missed. Syntax errors related to other uses of conflict resolvers may be missed, however, but we expect them to be few.

5. Results

In this section we first give some examples of noncorrecting error handling and then discuss results of an attempt to gain experience in a more systematic way.

5.1. Effect on user programs

The use of noncorrecting error handling has two effects for the compiler user:

1. All syntax error messages are to the point and their number is kept to a minimum.
2. No semantic error messages occur after the first syntax error (since the noncorrecting error handler does not perform semantic actions). Consequently, all semantic error messages are to the point too.

Both phenomena cooperate to clear up the visual appearance of the error message list and induce a high level of user confidence.

Sometimes the beneficial effect of noncorrecting error handling is striking. The program from Figure 1, excerpted from a larger program, contains two syntax errors, a missing semicolon in line 6 and a `9` instead of an open parenthesis in line 15. It produces the following 8 syntax and 3 semantic error messages on the Sun C compiler (version: bundled 4.1.2):

```
 1  struct sgttyb {short sg_flags;};
 2
 3  main () {
 4      if (1) {
 5          explain ('\n');
 6          printf ("")                          <=== missing semicolon
 7      }
 8      set_cooked ();
 9      exit (0);
10  }
11
12  struct sgttyb term;
13
14  set_raw () {
15      if (gtty 90, &term) < 0) {           <=== 9 instead of (
16          printf ("");
17      }
18      term.sg_flags = 1;
19      stty (0, &term);
20  }
21
22  set_cooked () {
23  }
24
25  explain (ch) int ch; {
26  }
```

Figure 1. A reduced sample program with two syntax errors

```
"keyb.c", line 7: syntax error at or near symbol }
"keyb.c", line 12: syntax error at or near word "struct"
"keyb.c", line 14: syntax error at or near symbol {
"keyb.c", line 15: gtty undefined
"keyb.c", line 15: syntax error at or near constant 90
"keyb.c", line 15: term undefined
"keyb.c", line 15: syntax error at or near symbol )
"keyb.c", line 22: syntax error at or near symbol {
"keyb.c", line 25: ch undefined
"keyb.c", line 25: syntax error at or near type word "int"
"keyb.c", line 27: syntax error
```

In effect, the Sun C compiler was completely led astray, and would give a syntax error message on each and every subsequent function declaration.

The GNU C compiler (Version 2.4.5) gave 5 syntax and 1 semantic error messages:

```
keyb.c: In function ‘main’:
keyb.c:7: parse error before ‘}’
keyb.c:12: parse error before ‘struct’
keyb.c:14: parse error before ‘{’
keyb.c: At top level:
keyb.c:18: parse error before ‘.’
keyb.c:19: parse error before ‘0’
keyb.c:19: warning: data definition has no type or storage class
```

The ACK C compiler using noncorrecting error handling produced:

"keyb.c", line 7: } syntactically not acceptable here
"keyb.c", line 15: integer syntactically not acceptable here

which does indeed pinpoint the errors. We observe, however, that the contents of the noncorrecting error messages are very meagre: the offending token is named, and that is all. The cause of this and possible remedies for it are discussed in Section 6.

Similar results have been observed for the ACK Modula-2 compiler.

5.2. The testbed

The above examples give incidental impressions only; acquiring extensive field experience in error handling is a slow process, however. To gain quick though artificial experience, we tested the C compiler and to a lesser extent the Modula-2 compiler on an error-generating testbed. This testbed is parametrized with very simple language and compiler descriptions, and starts from a correct program supplied by the user. From this program, it produces other programs with a single modification, compiles each of them, and collects data about compiler behaviour. Specifically it counts the total number of error messages, the number of syntax error messages and the number of compiler crashes; syntax error message patterns must be supplied in the compiler description. For each token position in the original program, the testbed generates $1{+}N$ test programs, where N is the number of language-specific tokens supplied in the language description. For each token position P, the first test program is a copy of the original program with the P-th token removed and the other N test programs each have one of the N language tokens inserted after position P. A special case provides insertion *before* the first program token.

This process mimics single deletion and insertion errors. In the present test no keywords were inserted, since the chance of one being inserted by accident is quite small, although such insertions may occur with programmers inexperienced in the language. The C program contained 258 tokens, and 37 different tokens were inserted in each position, resulting in 9841 tests (258 + 259×37).

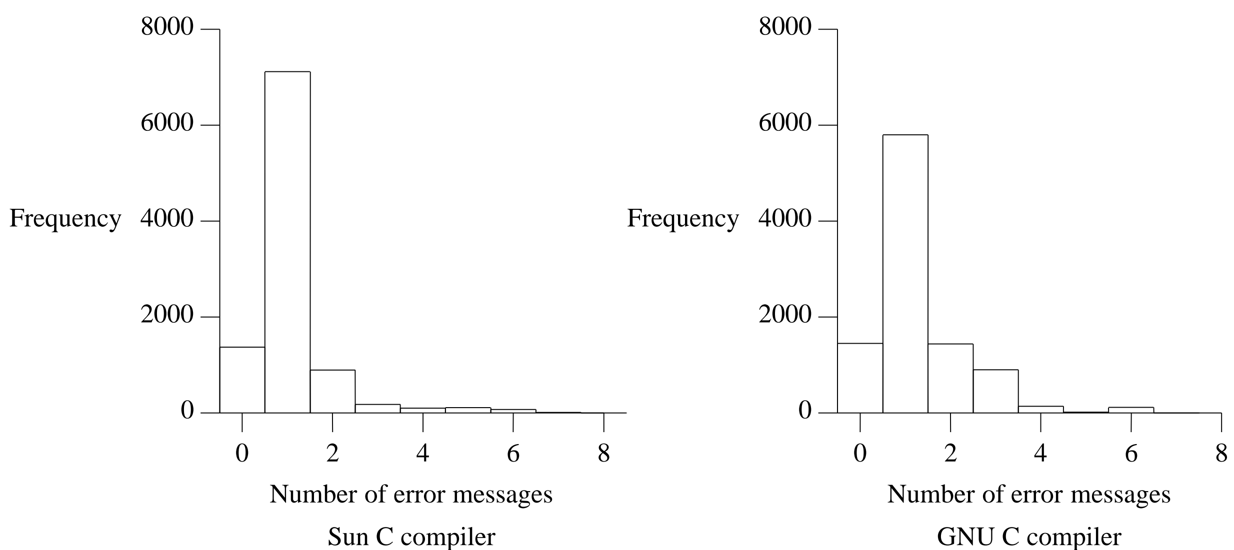


Figure 2

The results for the Sun and GNU C compilers are given in Figure 2. The first thing that strikes us is that about 14% (Sun) / 15% (GNU) of the modified programs exhibit no syntax errors at all. (The numbers differ slightly between the tested compilers due to differences in interpretation of the C manual. Also, some compilers prefer to handle certain syntax errors on the semantic level, in order to give a more user-friendly error message.)

And indeed, many modifications to a C program do not affect the syntactic correctness. For instance, in `a(3);` both the `a` and the `3` may be left out without causing a syntax error; also, `a:(3);` is syntactically correct. Likewise, before `*p=0;` the tokens `1` or `a` may be inserted with syntactic impunity.

The remaining 86% (Sun) / 85% (GNU) do have one or more syntax errors; of these, 16% (Sun) / 31% (GNU) show more than one syntax error message.

When noncorrecting error handling was applied directly to the original LLgen grammar of the ACK C compiler, the result was that some of the tests produced more than one error message. The reason was that the grammar of C is not completely context-free: identifiers, though normally used for variables, functions, etc., can be defined as type identifiers, which have syntactically different properties. The ACK C compiler handles this by having the lexical analyzer ask the semantic module if the identifier just read is a normal or a type identifier. During noncorrecting error handling, no further semantic actions are performed, however, and no further type identifiers are recognized. Consequently all legal uses of unrecognized type identifiers will show up as syntax errors.

We solved this problem by introducing a new keyword `%illegal` in *LLgen*, which identifies an alternative that is illegal during normal parsing but must be considered legal during noncorrecting error handling. After having replaced all occurrences of ‘`TYPE_IDENTIFIER` by ‘`TYPE_IDENTIFIER | %illegal IDENTIFIER`’ and all those of ‘`IDENTIFIER`’ by ‘`IDENTIFIER | %illegal TYPE_IDENTIFIER`’ in the grammar, no more multiple syntax error messages occurred in the test. The results are shown in Figure 3, together with the results of the original ACK C compiler. Again, this relaxation on the input requirements may allow some syntax errors after the first to be missed.

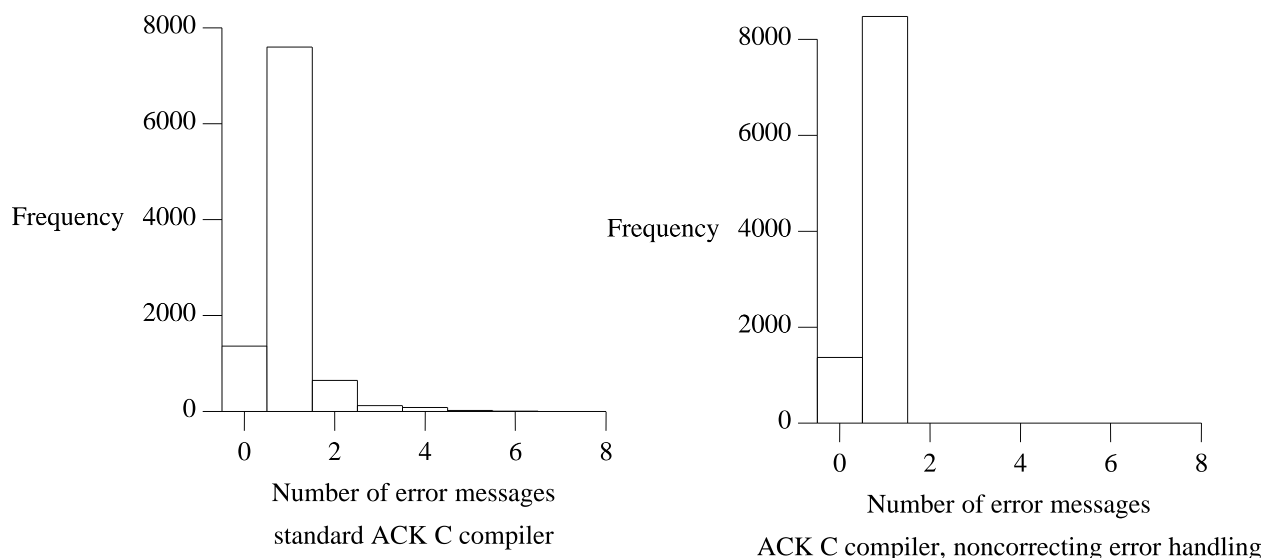


Figure 3

Two more keywords had to be introduced for a smooth operation, `%erroneous` and `%substart`. The keyword `%erroneous` identifies an alternative that is legal during normal parsing but illegal during noncorrecting error handling; it is the counterpart of `%illegal`. This keyword is useful for marking alternatives added by the compiler writer to catch certain syntax errors and handle them explicitly. The ACK Modula-2 compiler, for example, catches the use of ‘`=`’ for ‘`:=`’ and gives a message pointing out the confusion. Since this message is a semantic action, the use of ‘`=`’ for ‘`:=`’ would go unnoticed during noncorrecting error handling. Marking the pertinent alternative as `%erroneous` will make it invisible to the noncorrecting error handling and incorrect use of ‘`=`’ will be caught properly.

The `%substart` keyword has to do with the possibility that a semantic action starts a subparser that reads from the same input. During noncorrecting error handling no semantic actions are performed, the subparser will not be started and the input it should have processed is now recognized using the original grammar. This may, and probably will, cause unjustified error messages. The prefix `%substart X` to an action signals that the action may call a subparser with start symbol `X`. The noncorrecting recognizer will accept at that point a sequence of zero or more terminal productions of `X`.

We see that wherever the original parser deviates from pure context-free parsing to accommodate real-world grammars, we have to take special measures to introduce noncorrecting error handling, since the latter is purely context-free.

6. The suffix recognizer

The suffix recognizer is based on a full context-free top-down recognizer, modified to handle suffixes. In principle any full context-free recognizer would do, for example CYK, Earley or Tomita[6], but since we had no good full context-free recognizer in C available, we decided to stay within the top-down framework as used by *LLgen* and to design and implement a full context-free top-down suffix recognizer. The recognizer, which is simpler than both Earley and Tomita and is probably simpler than CYK, is described here only briefly; it is explained in detail by van Deudekom and Kooiman[7].

As is well known, the basic data structure in a top-down recognizer is the prediction stack, a sentential form which describes the input the recognizer will be willing to accept; in a normal (nonsuffix) recognizer it starts out as the start symbol followed by end-of-input. LL(1) recognizers have only one prediction stack, but full context-free top-down recognizers may need unboundedly many of such stacks. Our recognizer is loosely modelled on a (normal) context-free recognizer described by Greibach as early as 1964[8], which also features an explicit set of prediction stacks.

When the suffix recognizer starts, no prediction is available, since nothing is known about the input except that it is (or should be) a suffix of a correct program, and that it begins with a certain token, say `t`. To catch all possibilities, we take the set of predictions to be all suffixes of all right-hand sides in the grammar which start with `t`. Suppose our grammar is

$$S \rightarrow aSb \mid ab$$

and the first token after the error is ‘a’ (the token causing the error is skipped). Our set of predictions will then be:

a → S → b → [S]

a → b → [S]

where the stacks are represented as linked lists and [S] means that if our prediction has come true so far, we have recognized a suffix of S. We now run the normal predict-match cycle of a top-down parser on each of the predictions. During the match phase only those predictions will be kept which start with the input token, and the token will be removed from the input and from all surviving predictions. During the predict phase, a nonterminal on the top of a prediction will be expanded into all of its alternatives, creating one or more new predictions. The next cycle, for example, matches the ‘a’:

S → b → [S]

b → [S]

and then predicts two alternatives for the S which is now on the top of a prediction stack:

a → S → b → [S] → b → [S]

a → b → [S] → b → [S]

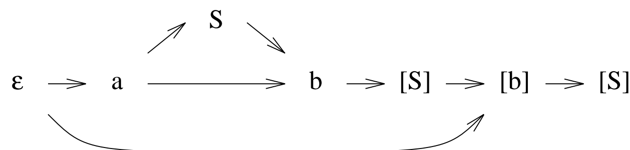
b → [S]

If there are no predictions left, we have found another syntax error. When this happens, the recognizer leaves us with very little knowledge about the nature of the error. All data structures that could suggest acceptable tokens have just been discarded, and all we know is the offending token. To give a better error message than 'token X not acceptable', we need to save information about acceptable tokens and/or suffixes being recognized, just before doing the match phase. We can then use this information to produce a better error message, should an error follow. This feature has not yet been implemented in *LLgen*.

Three aspects have to be paid attention to, to make the above scheme work: what to do when we have exhausted a prediction stack, how to achieve good efficiency, and what to do about left recursion. We shall discuss each of them briefly here; full details are given in the technical report by van Deudekom and Kooiman[7].

When a prediction becomes exhausted, its final entry tells us of which rule we have just seen a suffix; say this is rule *R*. We then extend our set of predictions by all suffixes of right-hand sides in the grammar which start with *R*, just as we did with the initial token `t`, and match the *R*.

If the prediction sets are implemented naively (as described above), the algorithm has exponential complexity. A major gain in efficiency is achieved by combining all prediction stacks in a Tomita-like directed acyclic graph[9]. For example, the actual representation of the above set of predictions is



where the ε entry on the left points to all tops of stacks. Although the resulting algorithm in principle still has exponential complexity for certain pathological grammars, it is next to linear for almost all programming language grammars. Some time measurements are given in Figure 4. In each case a syntax error was introduced before the first program token, to make the noncorrecting error handling process the entire file. For an average program of say a thousand tokens, the noncorrecting error handling takes 1 to 2 seconds on a Sparcstation 1+; the maximum size of the prediction graph is then about 5 kilobytes.

The stack combination process is also helpful in handling left recursion. Suppose our grammar contains the left-recursive rule E → E + x | x. Then a prediction

E → P → Q → ...

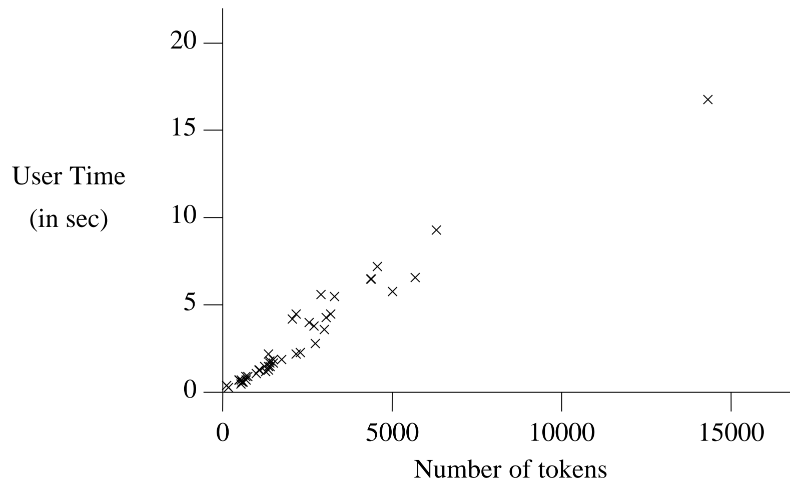
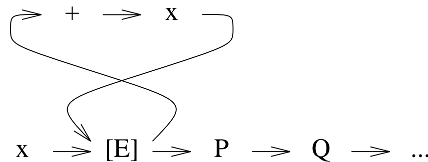


Figure 4. Time measurements of noncorrecting error handling of C programs

where P Q ... stands for the rest of the prediction, is developed to



during the prediction phase. Now the directed graph is no longer acyclic, but very little needs to be changed in the recognition algorithm. Indirect and hidden left recursion are handled similarly (both turn out to require hardly any code).

7. Conclusions

Noncorrecting error handling aims to give only correct and reliable error messages, in order to raise user confidence. A trivial way to achieve this is to stop after the first error. Noncorrecting error handling improves on this by continuing to find syntax errors after the first one. If the original grammar is not completely context-free, it may miss some subsequent errors.

Noncorrecting error handling works by discarding all previous syntactic information when a syntax error occurs. The rest of the input is then parsed with a grammar which produces all suffixes (tails) of programs in the language. When we find that the input is not even a suffix, we have located another syntax error. This process continues until the input is exhausted. To be practically applicable without severely restricting the input grammar, the method requires a reasonably efficient general context-free suffix recognizer.

We have shown that such a recognizer can be constructed in a simple way. Its construction does not depend in any way on properties of the original parser or grammar. It can therefore easily be incorporated in any parser generator.

We have incorporated noncorrecting error handling in *LLgen*, the LL(1) parser generator of the Amsterdam Compiler Kit, and have done some simple comparisons with existing compilers and some extensive though artificial tests. These have resulted in two observations.

The first is that if the original parser generator deviates from pure context-free parsing to accommodate

real-world grammars, these deviations will require attention when noncorrecting error handling is being added. Ignoring the deviations will result in small inaccuracies in the error messages, which may or may not be tolerable. For *LLgen*, we have introduced three new constructions, to cater for the fact that *LLgen* allows contextual information to influence parsing.

The second is that, upon error, the suffix recognizer presented here only identifies the offending token. This restricts the quality of the error message texts. To give more informative error messages, additional information will need to be remembered during recognition.

8. Acknowledgements

The authors thank Henri Bal, Philip Homburg, Ceriel Jacobs, Koen Langendoen, Lily Ossendrijver and Tim Ruhl, for their constructive comments.

9. Literature references

1. Helmut Richter, ''Noncorrecting syntax error recovery,'' *ACM Trans. Prog. Lang. Sys.* **7**(3), pp. 478-489 (July 1985).

2. Gordon V. Cormack, ''An LR substring parser for noncorrecting syntax error recovery,'' *ACM SIGPLAN Notices* **24**(7), pp. 161-169 (July 1989).

3. Andrew S. Tanenbaum, Hans van Staveren, E.G. Keizer, and Johan W. Stevenson, ''A practical toolkit for making portable compilers,'' *Commun. ACM* **26**(9), pp. 654-660 (Sept. 1983).

4. Dick Grune and Ceriel J.H. Jacobs, ''A programmer friendly LL(1) parser generator,'' *Softw. Pract. Exper.* **18**(1), pp. 29-38 (Jan. 1988).

5. Johannes Röhrich, ''Methods for the automatic construction of error correcting parsers,'' *Acta Inform.* **13**(2), pp. 115-139 (Feb. 1980).

6. Dick Grune and Ceriel J.H. Jacobs, *Parsing Techniques, A Practical Guide,* Ellis Horwood, Chichester, England (1990), p. 322.

7. Arthur van Deudekom and Peter Kooiman, ''Top-down Non-Correcting Error Recovery in LLgen,'' IR-338, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam (Aug. 1993).

8. Sheila A. Greibach, ''Formal parsing systems,'' *Commun. ACM* **7**(8), pp. 499-504 (Aug. 1964).

9. Masaru Tomita, *Efficient Parsing for Natural Language,* Kluwer Academic Publishers, Boston (1986), p. 210.