SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2001; **31**:803–814 (DOI: 10.1002/spe.390)

Schrödinger's token

John Aycock*,† and R. Nigel Horspool

Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, B.C., V8W 3P6, Canada

SUMMARY

A common problem when writing compilers for programming languages or little, domain-specific languages is that an input token may have several interpretations, depending on context. Solutions to this problem demand programmer intervention, obfuscate the language's grammar, and may introduce subtle bugs. We present a technique which is simple and without the above drawbacks—allowing a token to simultaneously have different types—and show how it can be applied to areas such as little language processing and fuzzy parsing. We also describe ways that compiler tools can support this technique. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: parsing; lexical analysis; syntactic analysis; superposition

INTRODUCTION

Compilers begin processing an input file by performing lexical analysis and syntactic analysis. Lexical analyzers, or scanners, effectively break the input into a sequence of words which are examined for syntactic correctness by a parser. As each word from the scanner is representative of one or more characters in the input, they are typically referred to as tokens.

Usually, the type of a token—the class of words that the token describes—is clear-cut. Problems arise, however, when a token's type depends upon context information that the scanner is not privy to. An example of this problem comes from PL/I, where statements like the following are legal [1,2]:

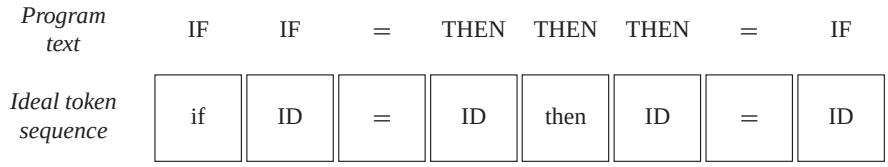
$$\text{IF IF} = \text{THEN THEN THEN} = \text{IF}$$

Ideally, the scanner should divine which uses of 'if' and 'then' correspond to keywords, and which correspond to identifiers; this would allow the parser to use grammar rules which directly reflect the structure of the language (PL/I, in this case). This ideal token sequence and grammar are shown

*Correspondence to: J. Aycock, Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, B.C., V8W 3P6, Canada.
†E-mail: aycock@csc.uvic.ca

Contract/grant sponsor: National Science and Engineering Research Council of Canada

Received 19 June 2000
Revised 16 January 2001
Accepted 16 January 2001



stmt → ifstmt | asgnstmt
ifstmt → 'if' expr 'then' stmt
asgnstmt → 'ID' '=' expr
expr → 'ID' '=' 'ID'

Figure 1. Ideal token sequence and (simplified) grammar.

in Figure 1. Unfortunately, the necessary context information to produce this token sequence is not directly available to the scanner.

In PL/I, this problem arises because keywords like 'if' are not reserved, and can be used in other contexts; most modern languages have avoided this particular difficulty. While compilation of PL/I is no longer a hot topic, this same problem still arises when compiling little, or domain-specific languages [3].

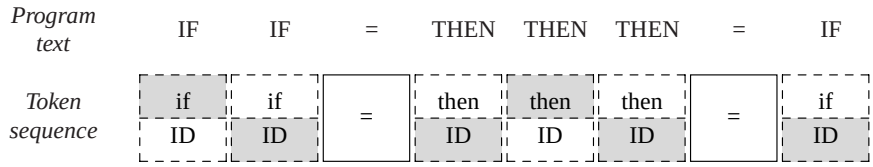
More generally, we do not restrict ourselves to keywords, and consider *any* situation where a token's type is context-dependent. One language may be embedded within another, as SQL can be embedded within C, C++, or COBOL; the interpretation of tokens may vary greatly between an SQL construct and the host language! There may also be many dialects of a single language which software re-engineering tools are obliged to recognize [4].

The following sections discuss a simple technique for handling context-dependent tokens, its implementation, and some applications. For comparison purposes, alternative techniques are also presented.

SCHRÖDINGER'S TOKEN

In 1935, Erwin Schrödinger published a paper on quantum mechanics in which he posed the exaggerated case of the now-famous 'Schrödinger's Cat' [5]. A cat and a flask of lethal acid share accommodation in a steel chamber, along with some radioactive material. If an atom in the radioactive material decays—there is a 50% chance of this happening—the flask is broken, and the global cat population is decreased by one. However, the decay of the atom, and the fate of the cat, are unknown until the chamber is opened; effectively, the cat exists in a superposition of 'alive' and 'dead' states until that time.

This idea of superposition is a faithful model of the token interpretation problem. In lieu of context information, a token does *not* have a unique type but instead has a superposition of types: it is all of its possible types simultaneously, until the parser peeks inside the chamber and resolves the token's



stmt → ifstmt | asgnstmt
ifstmt → 'if' expr 'then' stmt
asgnstmt → 'ID' '=' expr
expr → 'ID' '=' 'ID'

Figure 2. Token sequence and grammar, using Schrödinger's tokens. The shaded token types indicate which interpretation is eventually used by the parser.

type with context information. We use the term 'Schrödinger's token' to refer to a token which has a superposition of types.

Returning to our PL/I example, Figure 2 shows the new token sequence from the scanner. 'If' and 'then' are now Schrödinger's tokens, since they can be either a keyword or an identifier; the '=' token still has a unique type[‡]. Notice that the grammar is unchanged from the 'ideal' grammar in Figure 1—the use of Schrödinger's tokens is thus programmer-friendly, in that no modifications to the grammar are required, and therein lies a problem.

Physical superposition embodies[§] non-determinism, something which current computers are not particularly adept at. Parsing algorithms commonly used for compilers, such as the LALR(1) algorithm in Yacc [6,7] are deterministic and cannot generally cope with the fact that input involving a Schrödinger token may (temporarily) not have a unique parse. Instead, we use more general parsing techniques such as generalized LR parsing [8] or Earley's algorithm [9,10], which effectively simulate non-determinism if necessary to handle ambiguity in the grammar. In the past, these types of parsers have not been widely used in compilers due to efficiency concerns: all other things being equal, the more general parsing algorithms tend to be slower due to extra overhead [11]. However, this is becoming less of a concern with increases in processor speed and memory capacity. There are now a number of parser generators and other tools using these general algorithms [12–15], as well as approaches to making the algorithms faster [16–19].

We note that the Schrödinger token technique has been used in the past by computational linguists [20,21], who are accustomed to using more general parsing techniques. Specifically, the technique was used in natural language parsing to model the case where a single word can belong to many different parts of speech. To the best of our knowledge, computational linguists have no name for this technique, nor has it been applied outside the area of natural language.

[‡]One could argue that '=' also has a dual interpretation, as a comparison operator or an assignment operator.
[§]Or, in the cat's case, possibly disembodies . . .

ALTERNATIVE TECHNIQUES

There are a number of other techniques[¶] for dealing with context-dependent tokens. We present them here in a parser-independent manner but, in practice, many of them involve more programmer effort than is initially apparent. This is because techniques requiring grammar modification often trigger an orgy of grammar rewriting in order to make a modified grammar palatable to a particular parsing engine.

Lexical feedback

The scanner can be made aware of context if the parser and scanner share some state. The parser uses this shared state to communicate context information to the scanner; this is referred to as lexical feedback [6,7].

Lexical feedback has a number of problems in practice. It couples the scanner and parser tightly: not only do they share state, but the parser and scanner must operate in lock-step. The scanner cannot, for example, tokenize the entire input in a tight loop, or operate in a separate thread of execution, because at any moment the parser may direct it to change how it interprets the input. Additionally, the programmer must fully understand how and when the parser handles tokens, otherwise subtle bugs may be introduced.

Enumeration of cases

If the scanner insists upon returning the wrong token type in the wrong context, the grammar can be modified such that the 'wrong' token type is accepted as well as the right token type: the programmer enumerates all the valid cases in the grammar [22,23]. In our PL/I example, we would explicitly note in the grammar that the scanner may erroneously return an 'if' or 'then' token in place of an 'ID' token. Besides these grammar modifications, the correct addition of a new keyword requires changes to logically unrelated grammar rules.

Language superset

An alternative approach to handling context-dependent tokens is for the scanner to *never* attempt to distinguish between them, and to always return the most generic token type. The grammar can then be rewritten to accept a superset of the original language.

Eventually, however, it must be verified that the input conforms to the original grammar. This can be done in actions associated with the grammar rules, or deferred to later semantic processing. In either case, the programmer must perform work that the parser was supposed to do in the first place! Furthermore, the intent of the grammar rules is now hidden, making maintenance difficult.

[¶]Some of these techniques are folklore. Consequently, not all the citations in this section refer to original sources.

Oracles

When in doubt about a token's type, the scanner could invoke an oracle. This oracle would look forwards or backwards in the input stream, looking for patterns with which to glean context information. An oracle was used in a PL/I compiler [24], not by the scanner but by the parser, to compensate for the use of a weak parsing method. In practice, oracles may not always be feasible due to the performance impact of reprocessing input and the complexity of constructing a correct oracle.

Scannerless parsing

Clearly, if the central problem behind context-dependent tokens is a communication barrier between the scanner and parser, then merge them together to remove the barrier! This is the promise of scannerless parsing [25,26]. The programmer supplies a single grammar which describes both lexical and syntactic rules, making it easy to express the context of tokens.

There are tradeoffs, however. The programmer must supply a grammar which specifies the legal placement of whitespace and comments. Besides being error-prone, this does not decrease the grammar size nor increase its readability. The other concern is efficiency, because scannerless parsing uses more powerful pushdown automata to handle individual characters rather than finite-state automata. This latter point is hard to judge, as scannerless parsing tools are not available, and no timing results have been published.

Discussion of alternatives

Use of Schrödinger's tokens is closest in spirit to enumeration of cases. The two methods accomplish the same goal, albeit with different tradeoffs: enumeration of cases leaves the scanner unchanged and modifies the grammar, the opposite of Schrödinger's tokens. More broadly, Schrödinger's tokens suffer from none of the problems of alternative techniques, such as extensive grammar modification and limiting assumptions placed on the scanner and parser's operation.

IMPLEMENTATION

Implementation of Schrödinger's tokens can be divided into two logical parts: support required in parsing tools, and support by the user of those parsing tools, the programmer.

Programmer support

Given a parser generator which supports Schrödinger's tokens, the programmer need only supply a scanner which produces said tokens. This can be done in a hand-crafted scanner as well as using scanner generator tools.

Figure 3 gives a partial Lex [7,27] specification for our running example. A distinct token type, 'SCHRODINGER,' is returned to the parser to signal the appearance of a Schrödinger token. The set of possible types is stored in a global array for use by the parser; for pedagogical reasons, a maximum

```
%%

=               return '=';
if              return schrodinger(IF, ID);
then            return schrodinger(THEN, ID);
[a-zA-Z]+       return ID;

%%

extern int types[2];

int schrodinger(int type1, int type2) {
    types[0] = type1;
    types[1] = type2;
    return SCHRODINGER;
}
```

Figure 3. Partial Lex specification using Schrödinger's tokens.

of two superpositioned types is imposed. The usual disambiguation rules for preferring the longest match are applied.

How is the overlap between token definitions determined? In our experience, this can usually be done by inspection without much difficulty in the typical case where a single lexeme corresponds to multiple token types. However, it is possible to construct a scanner generator tool which would look for such cases automatically.

Parser tool support

Parser generators must have appropriate support for Schrödinger's tokens. We emphasize that this is a one-time tool modification, invisible to the end user/programmer.

For expository purposes, we begin with Yacc [6,7]. Yacc uses an LALR(1) parsing algorithm, a flavor of shift/reduce parsing. In shift/reduce parsing, the parser shifts values onto a stack until a valid right-hand side of some grammar rule is seen atop the stack, at which time the parser reduces, popping values from the stack. The decision as to what parsing action to take is controlled by a deterministic automaton, whose transitions Yacc encodes in a table. At its core, Yacc's parsing algorithm is a loop, indexing into a table based on the topmost stack value and the current input symbol, and performing the action specified in the table entry. Figure 4(a) shows this algorithm in pseudocode form.

As shown in Figure 4(b), supporting Schrödinger's tokens is a matter of performing an action for each of the superpositioned token types. Unfortunately, this is where LALR(1) parsing and Schrödinger's tokens part ways. Even a simple example such as the one in Figure 2 would require the parser to be in two distinct automaton states simultaneously, which is not possible in a deterministic automaton.

This is where general parsing algorithms come in. Where Yacc's deterministic LALR(1) algorithm would fail, an algorithm like generalized LR (GLR) parsing [8] simulates non-determinism, effectively


```
a = input()
while (true) {
    s = top_of_stack()
    switch (action[s, a]) {
        case shift s′:
            ...
        case reduce A → α:
            ...
        case accept:
            ...
        case error:
            ...
    }
}
```

(a)

```
a = input()
while (true) {
    s = top_of_stack()
    foreach t ∈ a.types {
        switch (action[s, t]) {
            case shift s′:
                ...
            case reduce A → α:
                ...
            case accept:
                ...
            case error:
                ...
        }
    }
}
```

(b)

Figure 4. Pseudocode for (a) the LALR(1) parsing algorithm, and (b) conceptual modifications for Schrödinger's token support.

running multiple parsers in parallel. In essence, a GLR parser employs the algorithm in Figure 4(a). However, instead of the single parsing stack that LALR(1) parsers have, a GLR parser has a set of parsing stacks: conceptually, one stack for each derivation currently being considered. A GLR parser must thus use the LALR(1) algorithm for each of its stack tops.

Regardless of the general parsing algorithm used, the required change is the same as outlined above. Any operation that depends on a token's type must be repeated for all the superpositioned types in a Schrödinger token. In the Earley and GLR parsers we have examined, very few lines of code need modification.

Depending on the parser's design, no changes at all may be required to support Schrödinger's tokens. In an object-based parser such as the Earley parser we use in our Scanning, Parsing, and Rewriting Kit (SPARK) [12], tokens are black boxes. The parser invokes a comparison method within each token to discern information about its type. In this parsing model, a one-line change to the token's comparison method is sufficient for Schrödinger's tokens to work.

APPLICATIONS

Schrödinger's tokens have many uses. We present three application areas: domain-specific languages, fuzzy parsing, and whitespace-optional languages.

Domain-specific languages

Domain-specific languages are languages tailored to particular application areas; they need not be fully general programming languages, so long as they are able to express information about their intended

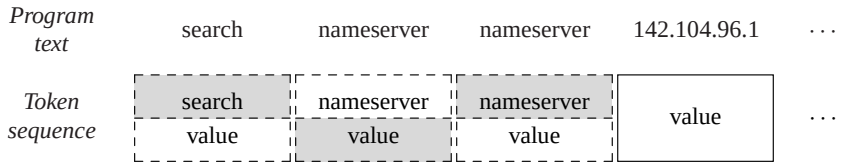


Figure 5. Schrödinger's tokens for parsing key-value pairs.

domain. These languages are often designed and implemented in an *ad hoc* fashion, making the use of traditional compiler techniques difficult, in part due to context-dependent tokens.

Configuration files are one example. Although they vary greatly in complexity, a simple format involves only key-value pairs, as in the `/etc/resolv.conf` file from one of our workstations:

```
search      csc.uvic.ca
nameserver  142.104.96.1
nameserver  142.104.6.1
```

On every line, the first word is the key, and the remaining words on the line are the value. If we were to process this using compiler tools, it would be reasonable to consider making each key a reserved word. This way, the grammar would reflect the structure of the language, and appropriate actions could easily be associated with each different key:

$$\begin{aligned} \text{searchstmt} &\rightarrow \text{`search'}\ \text{`value'} && \{\ \textit{action for search}\ \} \\ \text{nameserverstmt} &\rightarrow \text{`nameserver'}\ \text{`value'} && \{\ \textit{action for nameserver}\ \} \\ &\ldots \end{aligned}$$

However, nothing except good taste prevents us from re-using a key's name as a value, perhaps changing the first input line to 'search nameserver.' Figure 5 shows how the scanner can create Schrödinger's tokens to easily handle this case.

Some other domain-specific languages, and examples of how Schrödinger's tokens apply to their implementation, are as follows.

(i) Command-line arguments. On UNIX[‖] systems, commands like `find` and `expr` allow complicated expressions as command-line arguments. Keywords are not reserved and may appear as arguments.

(ii) Text-based network protocols. A number of network protocols, such as FTP [28], HTTP [29], and SMTP [30], use little languages for client–server communication. Again, keywords are not reserved.

(iii) Programming languages. Some modern domain-specific programming languages have context-dependent tokens. The first author experienced this when re-implementing Guide [31] using

[‖] UNIX is a registered trademark of The Open Group in the United States and other countries.

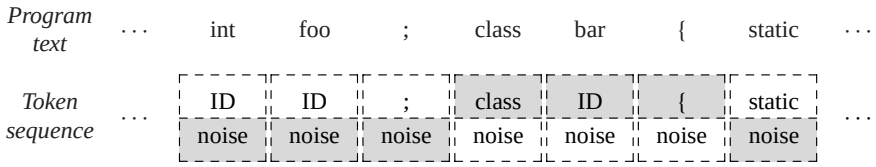


Figure 6. Fuzzy parsing of C++ using Schrödinger's tokens.

compiler tools; the initial implementation was an *ad hoc* Perl script. The data description language ASN.1 [32] also makes provision for non-reserved keywords that are distinguished by their context.

Fuzzy parsing

'Fuzzy parsing' is parsing which only recognizes part of an input [33]. Fuzzy parsers are useful for software re-engineering and tools which only look for certain features in their input. A fuzzy parser could extract all the data structure definitions from a program, for instance, or find all public class members in a Java program.

A fuzzy parser operates by skipping tokens until it sees a specified 'anchor symbol,' a sentinel token that indicates the start of the input sequence that the fuzzy parser recognizes. After parsing this input sequence, the fuzzy parser reverts to skipping tokens again.

From an engineering perspective, a fuzzy parser is looking for a signal amidst noise. Figure 6 shows how this idea can be applied to create a fuzzy parser that locates class definitions in C++ programs, for the purposes of discovering the class hierarchy (an example from Reference [33]). The scanner makes *every* token a Schrödinger token with a superposition of two types: the token's actual type that would be returned normally, and 'noise.' The grammar can then skip uninteresting input tokens by interpreting them as noise.

The scanner, once configured to return Schrödinger's tokens in this manner, can be re-used without modification for any fuzzy parsing of C++. Fuzzy parsers constructed with Schrödinger's tokens and a general parser are more powerful than previous fuzzy parsers, because the 'signal' they look for need not begin with an anchor symbol.

Whitespace-optional languages

Schrödinger's tokens were envisaged as a means of representing one piece of text that may have multiple token types. However, there are some languages where even locating token boundaries is a Herculean task. The classic example is Fortran, where whitespace is optional, and scanning is tricky [34,35]. For example, the partial input 'DO57I=' may correspond to two or four tokens, depending on the context. (Two abutted identifiers or integers are assumed to be invalid.)

Figure 7 suggests how to address this using Schrödinger's tokens. There are two distinct token sequences; the shorter of the two sequences is padded out with a special 'null' type. These null tokens

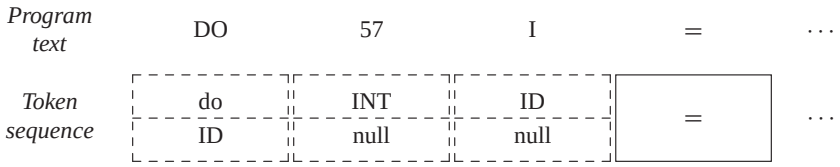


Figure 7. Schrödinger's tokens for parsing Fortran. The token interpretation is not shown due to lack of sufficient context.

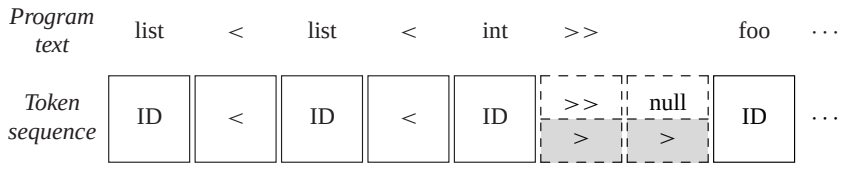


Figure 8. Schrödinger's tokens for parsing C++ template syntax.

must be ignored by the parser, which can be accomplished by grammar modifications that permit an 'ID' to be followed by zero or more null tokens.

This idea may be used to handle lexical ambiguity between subranges and real number representations in some languages, when it is unclear if '1..2' denotes a range of values or two adjacent real numbers. As shown in Figure 8, this idea also makes it straightforward to resolve the ambiguity in C++ between the right shift operator '>>' and nested template parameters (C++ template parameter lists are terminated by the '>' symbol) [36].

CONCLUSION

A superposition of token types—a Schrödinger token—represents situations where a token's interpretation is dependent on context. Using general parsing algorithms, Schrödinger's tokens have five major advantages compared to other methods for addressing the same problem.

1. No grammar modification is required; the grammar can accurately reflect the language being parsed, enhancing readability and maintenance.
2. Robustness: use of Schrödinger's tokens does not require the programmer to understand the parser's internal mechanisms. As well, no assumptions are made as to when the parser performs any actions that are associated with grammar rules.
3. Accurate modeling of context-dependent tokens, capturing the scanner's uncertainty with respect to token types.
4. No tight coupling between scanner and parser. This again simplifies maintenance, and allows scanners and parsers to be seen as interchangeable software components.

5. Support for Schrödinger's tokens may require as little as a one-line code change.

The Schrödinger token technique is a useful addition to the language implementor's toolbox.

ACKNOWLEDGEMENTS

Thanks to Shannon Jaeger for commenting on a draft of this paper. Also thanks to Susanne Reul and Mike Zastre for verifying part of the translation of Schrödinger's paper. The anonymous referees made many suggestions which improved the presentation.

REFERENCES

1. Aho AV, Sethi R, Ullman JD. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.
2. Fischer CN, LeBlanc RJ Jr. *Crafting a Compiler*. Benjamin/Cummings, 1988.
3. Bentley J. Little languages. *More Programming Pearls*. Addison-Wesley, 1988; 83–100.
4. van den Brand M, Sellink A, Verhoef C. Current parsing techniques in software renovation considered harmful. *International Workshop on Program Comprehension*, June 1988; 108–117.
5. Schrödinger E. Die gegenwärtige Situation in der Quantenmechanik. *Die Naturwissenschaften* 1935; **23**:807–812, 823–828, 844–849. Translation in Trimmer [37].
6. Johnson SC. Yacc: Yet another compiler-compiler. *Unix Programmer's Manual* (7th edn), vol. 2B. Bell Laboratories, 1978.
7. Levine JR, Mason T, Brown D. *Lex & Yacc* (2nd edn). O'Reilly & Associates, 1992.
8. Tomita M. *Efficient Parsing for Natural Languages*. Kluwer Academic, 1986.
9. Earley J. An efficient context-free parsing algorithm. *PhD Thesis*, Carnegie-Mellon University, 1968.
10. Earley J. An efficient context-free parsing algorithm. *Communications of the ACM* 1970; **13**(2):94–102.
11. Grune D, Jacobs CJH. *Parsing Techniques: A Practical Guide*. Ellis Horwood, 1990.
12. Aycock J. Compiling little languages in Python. *Proceedings of the 7th International Python Conference*, November 1998; 69–77. Software available at http://csr.uvic.ca/~aycock/python [6 June 2000].
13. Schröer FW. The ACCENT compiler, introduction and reference. *GMD Report 101*, German National Research Center for Information Technology, June 2000. http://www.combo.org/accent/doc.ps [6 June 2000].
14. van Deursen A. Introducing ASF+SDF using the λ-calculus as example. *Executable Language Definitions. PhD Thesis*, University of Amsterdam, 1994.
15. Wagner T, Graham SL. Incremental analysis of real programming languages. *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1997; 31–43.
16. Aycock J, Horspool N. Faster generalized LR parsing. *Proceedings of the 8th International Conference on Compiler Construction*, March 1999; 32–46.
17. Aycock J, Horspool N, Janoušek J, Melichar B. Even faster generalized LR parsing. 2000. Submitted for publication.
18. McLean P, Horspool RN. A faster Earley parser. *Proceedings of the 6th International Conference on Compiler Construction*, April 1996; 281–293.
19. Graham SL, Harrison AM, Ruzzo WL. An improved context-free recognizer. *ACM Transactions of Programming Languages and Systems* 1980; **2**(3):415–462.
20. Winograd T. *Understanding Natural Language*. Academic Press, 1972.
21. Milne R. Lexical ambiguity resolution in a deterministic parser. *Lexical Ambiguity Resolution*, Small SI, Cottrell GW, Tanenhaus MK (eds.). Morgan Kaufmann, 1988; 45–71.
22. Fisher GA Jr, Weber M. LALR(1) parsing for languages without reserved words. *ACM SIGPLAN* 1979; **14**(11):26–30.
23. Clark C. Keywords: Special identifier idioms. *ACM SIGPLAN* 1999; **34**(12):18–23.
24. Abrahams PW. The CIMS PL/I compiler. *Proceedings of the SIGPLAN Symposium on Compiler Construction*, August 1979; 107–116.
25. Salomon DJ, Cormack GV. Scannerless NSLR(1) parsing of programming languages. *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, June 1989; 170–178.
26. Visser E. Scannerless generalized-LR parsing. *Report P9707*, University of Amsterdam Programming Research Group, August 1997.
27. Lesk ME, Schmidt E. Lex—a lexical analyzer generator. *Unix Programmer's Manual* (7th edn), vol. 2B. Bell Laboratories, 1975.
28. Postel J, Reynolds J. File transfer protocol (FTP). *RFC 959*. October 1985.

29. Berners-Lee T, Fielding R, Frystyk H. Hypertext transfer protocol—HTTP/1.0. *RFC 1945*. May 1996.
30. Postel JB. Simple mail transfer protocol. *RFC 821*. August 1982.
31. Levy MR. Web programming in Guide. *Software—Practice and Experience* 1998; **28**(11):1581–1603.
32. International Telecommunication Union. *Specification of Abstract Syntax Notation One (ASN.1)*. 1993.
33. Koppler R. A systematic approach to fuzzy parsing. *Software—Practice and Experience* 1997; **27**(6):637–649.
34. Sale AHJ. The classification of FORTRAN statements. *Computer Journal* 1971; **14**(1):10–12.
35. Feldman SI. Implementation of a portable Fortran 77 compiler using modern tools. *Proceedings of the SIGPLAN Symposium on Compiler Construction*. August 1979; 98–106.
36. Stroustrup B. *The C++ Programming Language* (3rd edn). Addison-Wesley, 1997.
37. Trimmer JD. The present situation in quantum mechanics: A translation of Schrödinger's 'cat paradox' paper. *Proceedings of the American Philosophical Society* 1980; **124**(5):323–338.

