Schutz, a programming-language-aware editor.

Rodney Bates

1. Introduction

In the 1980s, there was a lot of research done on editors for programming languages. I was very interested in these, in part because I was tired of constantly reworking code layout when reworking code content. Also, I wanted help with many of what have come to be called refactorings, that are time-consuming and error-prone to carry out.

I was particularly interested in the method of static semantic analysis used the the PSG system. This does incremental analysis on incomplete programs, including type inference in the absence of declarations. Most of the tools I have encountered that provide the user with interactive information work mainly only on complete and syntactically correct code. They start to rapidly lose capability the minute any edit changes are made.

Unfortunately, interest in these died down, primarily, I believe, because they generally used "syntax directed editing", where the user constructs code something like building or altering an abstract syntax tree, while the code is displayed textually. This turns out to be, in many cases, far more difficult to use than just typing text. Interestingly, a variation of this idea has recently resurfaced in tools that call for graphiically abutting puzzle-piece-like blocks of various shapes, each representing a syntactic construct. But apparently it has not been embraced by programmers with even moderate experience.

I did not give up on the idea of a programming-language-aware editor. Instead, I embarked on an attempt to allow user editing in the usual textual way, while internally maintaining a syntactially analyzed form of code, upon which layout, refactoring, semantically-aware browsing, etc. could be performed by the editor. Schutz is the result of a decades long project to that aim. So far, about all it does is support text editing while maintaing an internal representation akin to an abstract syntax tree (AST), plus automated layout of the syntactic portions of code. But it is on the brink of readiness to support more sophisticated functions.

This has turned out to be insanely complicated, but I think the behaviors it shows mostly necessitate the complication. Here is a list of design goals, many of them more or less achieved.

2. Design goals

2.1. Internal AST Representation

The internal representation can be viewed as an AST. This is needed to support language-aware browsing, transformation, etc. Currently, this is used to support batch-mode backend processing of language definition, see Ldl1. Other uses are pending.

2.2. Automatic Layout of Syntactic Mterial

The layout of syntactic content of the code being edited is performed by the editor, according to layout rules provided as part of the definition of the language, see Ldl1. A relatively rich set of conditional layout rules are possible. They are isolated, defined in Ldl1, and can be changed by the user. This requires some thoroughness, so this is not a trivial task.

2.3. User-specified Nonsyntactic Layout

In contrast to syntactic material, the layout of nonsyntactic material such as comments is specified by the user and mostly preserved. There are is some automatic left-right shifting of commets, done to maintain certain kinds of alignment with shifted code.

2.4. Partial Analysis

Scanning, parsing, and semantic analysis can be performed on incomplete and incorrect code. Obviously, there are limitations on what is possible here, but these are minimized.

2.5. Language Independence

Schutz is capable of handling multiple languages. The great majority of Schutz is language-independent. Most of the description of a to-be-edited language is provided by a language definition in a specialized language, creatively named Ldl1, for "language definition language".

Currently, a lexical scanner for a language must be handwritten, conforming to a somewhat unusual interface, to support incremental rescanning. Generation of most or all of a scanner from the language definition would not be difficult, but is not currently implemented.

Schutz does make certain assumptions about the language, in order to allow for incremental scanning and parsing. In particular, scanning must be separable from parsing and both from identifier resolution, thus precluding C and C++. Removing this restriction would require a fundmentally different and, no doubt, even far more complicated design, and would probably preclude much of the current incrementality anyway. Also, Schutz requires that lexical tokens must either end with an explicit closing delimiting character such as a closing quote, or else not contain white space.

2.6. Language Definition Language

The parsing syntax, abstract syntax, formatting and layout syntax of a to-be-edited language are defined in a language definition language named Ldl1, along with a few odd bits. Schutz itself is used to parse Ldl1. It contains a backend translator, which generates several files, some Modula-3 source code, some tables, some prebuilt data structure, etc. These provide Schutz with the necessary information to edit the defined language.

The parsing of Ldl1 is bootstrapped using some batch mode main programs and a more primitive definition language named Ldl0. Eventually, equivalents of handwritten files are regenerated and used in the regular editor.

2.7. Incremental Anaysis

Rescanning, reparsing, and, when implemented, semantic analysis, are performed incrementally, only on portions of the code that have been edited since last time, and on limited portions of nearby code. Systematic reprocessing of a whole file is not necessary.

Incremental analysis does not require O(N) computation, where N is proportional to the entire edited program size. All operations are designed to have O(log N) complexity. However, there are curently one or two bugs that are worked around by an add-on O(N) repair process.

2.8. Compact Internal Representation

Here, "compact" is relative. All internal representations I have been familiar with tend to run around 10 times the size of the pure text file. Schutz was running around 5 times, in the days of 32-bit computers, but undoubtedly has lost some on 64-bit machines. You can’t feasibly make pointers smaller than pointers. Internal representations are written to files using Pickles, which are no doubt a bit smaller, because of pointer replacement.

2.9. Nothing Unasked-for

Schutz does not do anything to what the user types without being asked. Imagine typing code while your editor was reparsing, syntax repairing, and redoing layout after every keystroke, the way some document editors do line filling, through the intermediate states that are syntactically incorrect. Need I say more?

Incremental reanalysis happens only in response to user commands. However, if you do request it, you will then get the suggested repairs (see Syntax Repair), layout adjustment, etc.

2.10. Functional Representation

The internal representation is functional. that is, the various data objects are, after initial construction, immutable, and thus shareable. The O(log N) complexity of incremental operations depends, in large part on this property. Thus, merely keeping a pointer to the root of a representation effectively keeps that entire representation, unchanged. Any changes build new data structure, starting with a new root pointer, but sharing as much as possible of the old. This means undo is a simple as reusing the older pointer.

There are minor exceptions that do not alter any user-visible properties of the edited code. There is a less minor exception in the data structure for textual display on the screen, which uses some doubly-linked lists that get mutated. Since this is entirely derived from the main data structure, it can be recomputed at any time.

2.11. Syntax repair

Schutz has a syntax error repair scheme that is, well, elaborate. How well its behavior compares to others is unknown, but it was the most thorough I found in the literature, at the time of implementation. Since the representation has an AST at its core, with various additions and exceptions, parsing must come up with a syntactically correct repair.

A typical compiler would just quietly forge ahead translating a repaired program, or issue messages and refuse to go further. In keeping with Nothing unasked-for, Schutz inserts exceptions to the AST that, in effect, undo the repair actions. These are always shown on the screen, as inserted or crossed out tokens. Thus, although the layout will likely change, the token string is as the user typed, pre-repair. The user can explicitly accept repairs, en mass, or individually. Reparsing of unaccepted repairs treats them as originally typed.

2.12. Crash Recovery

With the exception of a brief period of vulnerability during screen repaint (see Functional Representation, Schutz catches assertion failures and runtime errors and opens a dialog allowing the user options, such as save the current data structure, undo the failing command, create a dump, etc.