21. Parser-Independent Compilers

W.M. McKeeman, Shota Aki, and Scot Aurenz
Digital Equipment Corporation
110 Spitbrook Road
Nashua, NH 03062

Abstract

The parse of a text is a sequence of grammar rules applied to source input. The text is brought into the parser as shift actions and the rules are applied as reduce actions. The resulting shift/reduce sequence has some useful properties as an intermediate language for compilers. It is independent of the parsing technology used to produce it. It can be stored in a file. It can be incrementally updated. It can be used to build other intermediate forms such as syntax trees. This paper discusses some techniques for building and optimizing the shift/reduce sequence. One such technique has been applied by the authors to an incremental ANSI C compiler.

Background

The parser is both the best understood and the most central facility of a compiler. The driving loop of the compiler is in the parser—it calls the scanner and drives the generator. For the purposes of this paper, a top-down parser is a set of mutually recursive routines. A bottom-up parser is a table-driven stack automaton. Both examine the input text and report its structure. There are several relatively standard and low-cost ways of building parsers that are error free and efficient [1, 4].

In the process of attempting to reuse a particular C front end, the authors found that the output of the existing parser would not serve the intended new purpose. One alternative was to modify the parser to make a different form of output that would work. Another was to build a new parser. Neither alternative provided reuse—in the end there would be two artifacts to maintain. When we in fact built a new front end, we built output from the new parser that would have been sufficient for both previous uses. It seems as if this result is more generally reusable, and therefore is documented here.
Technical Basis

The use of a context-free grammar (CFG) to describe the phrase structure of programming languages is nearly universal [1]. Nonterminal symbols represent the main structures of the language. Terminal symbols represent the punctuation (operators, reserved words, etc.) and words (identifiers, constants, strings, etc.) of the language. The implementation representation of a terminal symbol is called a token. The effect of the scanner is to produce a sequence of tokens.

For C the output of the scanner is preprocessor tokens which are then input to the preprocessor which produces tokens for the parser. Providing preprocessing is irrelevant for the purposes of this paper. Think of the preprocessor as part of the scanner.

The parser examines the token stream and discovers and reports the phrase structure. The two major contending technologies for implementing parsers are bottom-up and top-down (BU and TD).

A BU parser such as yacc is typically a shift/reduce automaton. There is a parse stack which is initially empty. Each shift action takes one terminal symbol from the input and pushes it on the parse stack. Each reduce action applies a rule from the CFG to the top of the parse stack. The right-hand-side of the rule consists of a sequence of n terminal and nonterminal symbols; the top n symbols of the parse stack must match the right-hand-side. The reduce action pops all n symbols and pushes the left-hand-side of the CFG rule. Parsing terminates when the input is exhausted and the parse stack contains exactly one symbol—the so-called goal symbol of the CFG.

A TD parser is a set of recursive routines, each named for a nonterminal symbol and responsible for choosing and applying the CFG rules defining that nonterminal. The process begins when the routine for the goal symbol of the CFG is called. The process 'descends' through further routine calls and eventually returns to the original routine, which by returning, signifies that the parse is complete. There is no explicit parse stack, but the same sequence of shift and reduce actions implicitly takes place in the call stack.

The effect of the scanner and parser together is completely described by an interleaved sequence of shift and reduce actions. Suppose the rest of the compiler, represented by a module called the generator, receives only the shift and reduce actions via entries:

\[\text{Shift}(t) \quad \text{— report shifting token } t \text{ to the generator}\]
\[\text{Reduce}(r) \quad \text{— report applying rule } r \text{ to the generator}\]

The token is reported as soon as it is accepted and the rule is reported as soon as it is applied. This interface can be satisfied by either a BU or a TD parser.

The generator then has enough information to implement any kind of translation desired without referring to private data in the parser. In particular, the parse tree itself can be constructed, which contains all of the information about the original program (a left-to-right sweep of the leaves of the tree) as
well as complete information on the application of the CFG. This paper proposes the restriction of the post-parser interface to just the two routines, Shift and Reduce, together with access functions for the abstractions that deal with the rules and tokens passed by these two routines to the generator. It is our experience that any differences in compiler performance caused by following the structure recommended here are slight.

Tradition

The traditional BU and TD parsers each differ from the proposed solution in the manner in which they provide storage for the generator. The parse stack, an internal artifact of the BU parser, is a convenient structure to elaborate and exploit to save intermediate generator information [2]. The call stack of the recursive routines in a TD parser provides local variables, which are the corresponding place to save intermediate information. If either traditional storage technique is used, the parsers are incompatible. The alternative is to provide a general state saving mechanism in the generator, replacing the traditional use by the generator of the BU parse stack or TD call stack.

The description of the technique of parser-independent compilation requires some detailed discussion of functions provided across the compiler interfaces. There are many ways such interfaces can be defined, and many names by which the functions can be called. The interface presented here is picked to make the presentation easy to read. There is no implication that either the names or the specific choice of functions is optimal.

It is sometimes necessary to go beyond strictly grammatical means of constructing a parse. The variety of such ad-hoc solutions (backtracking for resolving ambiguity, feedback from declarations to the scanner for typedef, etc.) is beyond the scope of this paper. One can observe that the more regular the parser, the easier it is to make ad-hoc modifications.

The Scan/Parse Interface

Each compiler has a scanner that delivers up the source text of the program as a sequence of tokens. As a practical matter, the scanner needs three entries. Suppose they have names as follows:

\[
\begin{align*}
\text{Scan()} & \quad \text{— steps ahead in the input} \\
\text{t} = \text{CurrentToken()} & \quad \text{— provides the current token} \\
\text{t} = \text{LookAheadToken()} & \quad \text{— provides the lookahead token}
\end{align*}
\]

It happens that each call of Scan is always followed by a call of CurrentToken. (Otherwise, why bother to step ahead?) There is no reason not to implement an entry into the scanner combining Scan and CurrentToken, but doing so does not simplify this presentation.
Each token carries some required information: a lexical code (a small integer identifying which terminal symbol it represents), a textual representation (a character string), and perhaps also some other less often used information, such as the line and column in the input where the token begins. Access to the information is provided by routines acting on the value $t$ of function `CurrentToken`:

- $c = \text{LexCode0f}(t)$ — small integer identifying token $t$
- $v = \text{Text0f}(t)$ — string representation of token $t$
- $f = \text{File0f}(t)$ — name of source file containing token $t$
- $n = \text{Line0f}(t)$ — source file line for start of token $t$
- $n = \text{Col0f}(t)$ — source file column for start of token $t$

and so on. The value $t$ itself is a unique representation for the token upon which no operations are allowed except assignment and those supplied by the scanner. The frequency of use of function `LexCode0f` is high, indicating that its implementation should be particularly efficient. Both `LexCode0f` and `CurrentToken` may in fact be macros and/or use hidden local variables to improve performance.

The parser calls through these entries to the scanner. Excepting nonstandard situations (such as caused by C's `typedef`), the parsing decisions require only `LexCode0f(t)` for each token $t$. A TD parser has numerous calls to `Scan` and `CurrentToken` scattered over a number of recursive procedures. A BU parser needs just one call to `Scan` and also just one call to `CurrentToken` to implement the read-state processing of the automaton it implements. In both cases the parser may be unable to make some decisions without looking ahead. As before, a TD parser may have many scattered calls to `LookAheadToken` where a BU parser has exactly one call to `LookAheadToken` to implement the reduce-state processing of the automaton. The important point is that the scan/parse interface is the same for both TD and BU parsers.

**The Parse/Generate Interface**

The principal action of a parser is the application of a CFG rule to reduce the input. A sequence of such actions constitutes the canonical parse.

A BU parser naturally calls `Reduce(r)` when each rule $r$ is applied. The proposed TD parsers must do exactly the same. It is not difficult to arrange for the TD parser to cause the same sequence of `Reduce` calls that the BU parser causes. The resulting recursive routines are more regular since all non-parsing detail is removed from them. The details of the rules may be built into the generator or may be available through a grammar abstraction. For example, the generator may have access to functions in addition to those accessing tokens, to simplify the process of interpreting the rules. For example:
\( c = \text{RuleCodeOf}(r) \) – small integer identifying rule \( r \)
\( s = \text{LhsOf}(r) \) – left side of rule \( r \)
\( n = \text{LengthOf}(r) \) – number of symbols on right side of rule \( r \)
\( s = \text{RhsOf}(r, i) \) – \( i \)th symbol on right side

The generator uses the rule and token information to build the intermediate representation of the program. The intermediate representation is typically some form of prefix notation, linear pseudo-code, or abstract syntax tree.

The traditional path for the token is from the scanner through the parser to the generator. For traditional TD compilers, a call to `Shift` immediately precedes each call to `Scan` because that is the moment of acceptance.

\[
\text{Shift}(\text{CurrentToken}()); \quad \text{– send token along}
\]
\[
\text{Scan}(); \quad \text{– discard token}
\]

For traditional BU compilers the tokens are already in the (private) parse stack. Rather than send the token to the generator by calling `Shift`, the information may be kept in the parse stack and delivered up to the generator on demand (a pull by the generator from the parser, instead of push by the parser into the generator). Some generators contain private knowledge of the layout of the parse stack data structure. Others use a procedural interface to get at the information, keyed on the match of the top of the parse stack and the right-hand-side of the rule. For example:

\[ t = \text{ParseStack}(2) \]

might retrieve the token positioned two below the top of the parse stack, and so on. This technique is not to be used with the parser structure proposed here. To match the activity of the TD parser, the proposed BU parser must also call `Shift` (so it too does a push into the generator).

To summarize, the traditional BU parser calls `Reduce`. The traditional TD parser calls `Shift`. They both use ad-hoc methods for communicating additional intermediate information to the generator (parse stack versus local variables in the call stack). The proposed BU and TD parsers must limit their interactions with the generator to calling only the two routines `Shift` and `Reduce`.

There are two convenient places for a parser to add the calls to `Shift`. Each call to `Scan` can be preceded by a call to `Shift`, as noted above. Or the scanner itself can call `Shift` immediately upon entry to `Scan`, just before updating the value of `CurrentToken`. It is obvious that the effect is the same. When it is difficult to modify the parser it may be necessary to have the scanner call `Shift`. 
A Parse Tree Generator

The parse tree is too voluminous for practical use as an intermediate language. However, the following example shows how a very general generator is constructed using the two-routine interface described above.

Suppose that the only effect of \texttt{Shift(t)} is to stack the token \( t \) on a local stack in the generator. And suppose the only effect of \texttt{Reduce(r)} is to build an \( n \)-ary node (where \( n \) is the length of rule \( r \)), pop \( n \) things off the generator stack, place them in the the node, and push the node on the generator stack. At the end of parsing, the generator stack will have a single entry—the root of the parse tree itself.

The important point is that the parse tree is built without reference to any information saved in the parser. This shows the sufficiency of the two-function interface.

Filtering

Some tokens and some rules have no semantic significance. That is, they result in no action in the rest of the compiler. While it can be said that tokens carrying semantic information, such as identifiers and constants, and rules corresponding to semantic actions, such as arithmetic and branching, are surely significant, there is no corresponding concept of ‘surely insignificant.’ Only the language implementor knows for sure.

Without loss of generality one can say that the compiler filters the sequence of tokens and rules, discarding insignificant items. The filter may be placed on either the sending or receiving end, much as the call to \texttt{Shift} is placed before or within the call to \texttt{Scan}. At the receiving end, the generator may provide filtering by ignoring \texttt{Shift} and \texttt{Reduce} when insignificant information arrives. This is in fact how things end up if nothing special is done.

Another way to filter is for the implementor of the compiler to tabulate the significant tokens and rules so that \texttt{Shift} and \texttt{Reduce} omit sending the insignificant items to the generator. It is slightly more efficient to eliminate them at the source rather than ignore them later at the destination. The augmented interface to the generator becomes:

\[
\begin{align*}
\text{if (SignificantToken(t)) } & \text{ Shift(t)} \\
\text{if (SignificantRule(r)) } & \text{ Reduce(r)}
\end{align*}
\]

For TD parsers, the test on \( r \) above is often computable at the time the parser itself is compiled. If filtering is on the sending end, the receiving generator needs to compensate by not looking for the missing information.
An Abstract Syntax Tree Generator

Using the filtered sequences, one can build an abstract syntax tree generator analogous to the parse tree generator described above. Because only significant tokens get stacked, the size of the n-ary node is reduced to the number of significant tokens and non-terminals in the rule. Nodes are built when rules are reported—and therefore fewer nodes are built for the abstract syntax tree because insignificant rules are not reported. At the end of parsing, the generator's parse stack has a single entry—the root of the abstract syntax tree itself. If the compiler requires a different, or more elaborate, intermediate language, all of the building activity can be isolated in the generator. This frees the generator from conforming to structures of the parser, and leaves the parser function and implementation unchanged.

Syntax Error Behavior

In addition to correctly parsing correct programs, the parser must respond constructively to syntax errors. There are two issues: how useful is the diagnostic message and what happens after the error is detected? The proposed solution makes detection, diagnosis and continuation private to the parser. The parser is responsible for terminating the compilation, or alternatively guaranteeing that the reported Shift and Reduce values are consistent with some correct program. The point is that the rest of the compiler is spared the extra engineering required to deal with invalid input from the scanner/parser.

Recovery from syntax errors is simpler with BU parsers because the entire state of the parse is available for manipulation by the error routine. In TD parsers, the state is wrapped up in the call stack. Typically TD parsers written in C resort to the setjmp/longjmp functions as a relatively clumsy way to control the contents of the call stack. Recovery can also be better with BU parsers because there is a well-developed technology for gathering right-context (the so-called forward move algorithm [3]).

Scanning and parsing diagnostics are inherently limited to the model of "an X was seen in the context of trying to do Y, and only one of Z1, Z2, or Z3 is acceptable." Diagnostics that go beyond this formula are guessing what might have been intended. Such guesses are often helpful, but also often misleading. We prefer to stick to the known facts. In addition to stating what happened, the diagnostic should provide the location of the offending X. During scanning, the current file, line, and column are apparent to the scanner. During parsing, the token abstraction carries the necessary information so that the parser can report the location to the diagnostic mechanisms (recall functions LineOf(t), etc.).

If a parsing error occurs at a token that resulted from macro expansion, the reasonable location to report is the outer macro invocation. The compiler can, in addition, list the stack of active macros, although it takes some preplanning beyond just the information available in tokens.
Later in the compilation process, the diagnostics report inconsistencies between two sources of information (for example, declaration and use). The tokens in the shift/reduce sequence provide the basic signposts upon which to establish the locations, although it can happen that an otherwise insignificant, and therefore filtered, token is significant to the diagnostic process. The filter must then pass it.

Conclusion

There are many reasons behind choosing a parsing technique. The point of this note is not to make the choice, but rather to remove one set of reasons often cited for making the choice. The proposed solution rules out any criterion based on the rest of the compiler since the rest of the compiler is independent of the choice. The proposed solution is also of comparable efficiency to traditional solutions. In any case, the authors believe parsing cost is small compared to the rest of compilation.

References


William McKeeman is a Senior Consulting Engineer for Digital. He has co-authored several books and has published papers in the areas of compilers, programming language design, and programming methodology. He can be reached at mckeeman@tle.dec.com.

Shota Aki is a Principal Software Engineer at Digital. He was a principal developer of the VAX APL interpreter. His current interests are in the areas of CASE, Analysis and Design Methods, Compilers and Tools. He can be reached at aki@tle.dec.com.

Scot Aurenz is a Principal Software Engineer at Digital and is currently working on the Language Sensitive Editor project. He can be reached at aurenz@tle.dec.com.