Abstract

The design of a parser for Standard C that is independently reusable and incremental is presented. The approach is similar in concept to earlier papers on scanners and preprocessors. Reusability is achieved by separation of function, simple and consistent interfaces, and provision of component-level testing. The parsing algorithm is traditional recursive descent. Incremental capability is achieved by saving shift/reduce sequences corresponding to blocks of source lines together with a superimposed tree structure to manage the increments. When an increment of source is unchanged and the same type definitions as before are needed to parse it, the corresponding saved shift/reduce subsequence is used instead of repeating the parsing.

Introduction

Reusable and incremental compiler components were introduced in earlier papers [5, 6, 7]. The information in those papers is used here without further explanation. As before, the desire to speed up parsing and to enable incremental processing downstream from the parser form the design focus. The fundamental observation driving this work is that source code is usually formatted along syntactic and semantic boundaries—line breaks are deeply significant. We have leveraged this observation to produce a prototype fine-grain incremental vertically integrated programming environment and have applied it to building an incremental front end for Standard C.

This paper proposes a regular interface to a parser for Standard C to achieve reusability. A typical reuse of a parser is for some CASE tool requiring source code analysis, for example lint or a pretty printer. The design of the parser is conceptually consistent with the interface proposal in an earlier paper[8].

This paper extends the interface to provide for incremental parsing. The general approach, which can be applied to any programming language, is to build shift/reduce subsequences for increments and then combine them in a
relatively flat tree structure designed for incremental update. The leaf subsequences correspond to the constructs that do not themselves contain increments (for example, C expression statements). The interior nodes combine other nodes to represent compound constructs (for example, C function definitions). The incremental tree management structure is not a syntax tree—all syntactic information is contained in the shift/reduce entries.

Type definition analysis in C complicates any parser separated from the C declaration analysis. The typedef problem was discussed in an earlier paper[4].

Making a parser incremental requires some changes in its structure. These changes are described here in terms of traditional recursive descent techniques. We believe analogous changes could be made to bottom-up parsers. There is some useful information about the differences and similarities in an earlier paper[8].

Incremental parsing for context free languages has been the subject of many papers[9, 3, 2]. The earliest work in this area required the user to work through an environment-supplied editor that kept the structure of the source correct at the cost of restricting the order in which the user presented changes. Gafter's solution[3] has no such restriction. Its intermediate form for parser output is modeled on a parse tree. Whenever the structure of a parse subtree becomes stale, the subtree is dissolved into a sequence of tokens and then reconstituted by the incremental parser. This is a general and elegant way to deal with incremental update, and does not require constraints on the user input. Gafter's solution is somewhat less efficient than our solution, and neither requires nor utilizes the unconscious help provided by neatly formatted source text. Another speedup trick depends on the fact that most C presented to a parser actually comes from include files—preparing include files is probably as effective a speedup as any other techniques.

The solution presented in this paper differs from previous solutions in the form of its output, the full treatment of Standard C[1], and the incorporation of incremental typedef analysis.

Parsing and the Shift/Reduce Sequence

The C Standard contains a grammar which defines the form of C[1, Appendix A.2]. The published grammar is not a context-free grammar. The abbreviation represented by the opt notation can be mechanically backed out, increasing the number of grammar rules. The missing rule defining enumeration-constant as identifier can be added. These modifications provide a definition of the shift/reduce sequence except that there are two ambiguities, the time-worn if else ambiguity and the typedef-name ambiguity. The former is resolved by matching each else to the nearest unmatched if; the latter is resolved through a special typedef analysis mechanism[4].

The application of the modified Standard C grammar to source text yields a sequence of rule applications, ending with a rule for translation-unit. At each
typedef struct {private layout} Sr

s = SrFirst() — iterator functions
s = SrNext(s) — start at head of the input
s = SrPrev(s) — step to next action

— step back to previous action

— attribute functions

c = SrIsPp(s) — shift or reduce discriminant
t = SrAsPp(s) — token representation of shift
r = SrAsRule(s) — handle for applied rule

Table 1: Parser Output Interface

step, the parser examines the current token to make decisions. Once a token
has been accepted, it is placed in the shift/reduce sequence and the parser turns
its attention to the next token. The placement can be done by either the token
consumer (as part of the discard action) or the token producer (as part of the
next-token action). The shift/reduce sequence is sufficient to build any post-
parser intermediate form. The correctness of a particular shift/reduce sequence
can be simply checked by reference to the grammar.

The Shift/Reduce Input and Output Interface

Suppose a preprocessor for Standard C is implemented (in C) in module pp.c
with public interface pp.h[6] and a parser is implemented in module sr.c with
public interface sr.h. The token output of the preprocessor is input to the
parser. Used in a C translator, the shift/reduce output of the parser is input to
the generator action routines. The principal data type produced by the parser
is Sr. The values of type Sr are action handles—all uses of Sr in other modules
except for declarations, assignments and parameter passing are forbidden. The
internal details are private to the parser. The parser provides access functions
that work via the handles. The choice of type name Sr emphasizes that reusab
modules need names that are tied to what they produce rather than some
particular consumer.

The form of the parser interface for the consumer is similar to that for the
scanner and preprocessor. The entries are given in Table 1.

The iterator functions provide the primitive navigation operations out of
which a shift/reduce jacket can be built by the consumer. The SrPrev() op-
eration is not needed for traditional compiler use. The information carried
by tokens is available through the preprocessor abstraction. For example, the
Implementing Recursive Descent

The implementation of a recursive descent parser is usually straightforward; the technique is well documented in introductory compiler texts and results in one recursive routine for each non-terminal in the grammar. It is a good idea to pass a grammar through an LR parser generator such as yacc before starting to write the recursive routines, just to confirm the civility of the grammar.

The LR technology automatically handles long prefixes of locally ambiguous text so long as the grammatical description of the prefix is common to all parses; long common prefixes are harder for recursive descent. An example of this kind of problem for recursive descent occurs in the C grammar rule for parameter-declaration where declarator and abstract-declarator have arbitrarily long common prefixes; the recursive routine for parameter-declaration has no easy way to find out which of the two routines to call. The long lookahead solution is to add a new recursive routine

\[\text{declarator-or-abstract-declarator}\]

which forges ahead. By the time the parameter-declaration is completely processed, the local ambiguity is resolved, and the “or” routine can report the correct reduce action.

Making Recursive Descent Incremental

For the moment we ignore the typedef problem.

The incremental version of the parser differs in only a small number of places from the non-incremental textbook form for recursive descent. Recall that the input from the preprocessor comes in increments each containing a sequence of post-pp tokens. Usually (because programmers format their programs neatly) the input increment reduces via the grammar to a single non-terminal, and often that non-terminal is from a small set of potential atomic increments: e.g.,

\[\text{declaration, statement, or external-declaration}\]

Initially all source must be parsed. For each input increment there is an output shift/reduce subsequence. The catenation of all these subsequences for a whole translation-unit is the shift/reduce sequence for the translation-unit. For some such subsequences the final rule (root non-terminal) represents a permitted shift/reduce increment. When permitted, a tree-node collecting all of the constituent subsequences is built and that tree-node takes the place of the shift/reduce subsequence in subsequent increment building. The purpose of the tree-node is to make incremental delete and insert convenient—it is entirely transparent otherwise.

At file scope level, a syntactically correct C module is a sequence of external declarations. Restricting potential increments to external declarations is essentially the same as choosing function definition as the incremental granularity.
Desiring finer granularity, we started with the observation that within all statements except expression-statement and jump-statement, more statements and declarations may be nested. Building increments at this level of granularity results in an increment tree that is much flatter than a conventional parse tree or abstract syntax tree. We chose the statement/declaration level of granularity because it is the finest granularity that is likely to match user source line breaks.

On a reparse, if a sequence of input increments corresponds to a single output increment, and none of the inputs have changed, then the output is unchanged and need not be recomputed. The typical case is a single line statement or declaration. An unchanged source increment is the most frequent situation when translating a program that is undergoing small changes during development.

If, on the other hand, some input has changed (has been modified or is new), the parsing is redone and, if the same non-terminal is reached, the new tree node can be inserted for the old one without changing the neighboring shift/reduce subsequences. This is the second most frequent situation.

Finally, if a parse increment is redone and the same non-terminal is not reached, the following increment now has the wrong left context and must therefore be treated as invalid. This situation occurs when a change involves the nested statement-level structure of the language.

The repair of the tree after any change happens because each containing level is reparsed, rebuilding the connective rules, but not needing to reparse any unchanged previously parsed increments. The repair of the increment tree is simple because it is so flat.

The typedef Problem

The C identifier is used for many purposes. Reserved words are dealt with by the scanner. All other uses can be disambiguated by local context except a typedef-name. As detailed in an earlier paper[4], the solution for a conventional parser is to implement a conventional symbol table in the parser, capable of answering only one question: is the current identifier token a typedef-name?

It is the nature of C that symbol table contents at some point during a parse depends on everything prior to that point—evaluating the is-typedef-name question is an action-at-a-distance problem. The preprocessor has a similar problem because #define and #undef directives determine the state of the macro table[7]. The simplest way to look at the action-at-a-distance problem is that certain calls to the parser symbol table were made during the original parsing. If the input is clean, and the symbol table calls are repeated, and give the same answers as before, then reparsing will only duplicate the contents of the output increment, and therefore the older version may be retained without further computation.

Our solution is to attach journals of parser symbol table activity to each shift/reduce increment. A shift/reduce increment can be reused if, in addition
typedef struct {private layout} Sr
void SrEnterScope(void);
void SrExitScope(void);
void SrReenterScope(void); /* undo premature Exit */
void SrEnterTypedef(Token t);
void SrObscureTypedef(Token t);
bool SrInTypedef(Token t);

void SrUnEnterScope(void);
void SrUnExitScope(void);
void SrUnReenterScope(void);
void SrUnEnterTypedef(Token t);
void SrUnObscureTypedef(Token t);

Table 2: Parser Symbol Table Interface

to not being stale with respect to the corresponding input increments, the calls
to the symbol table that were used to create give again the previously journaled
results. When an increment passes both tests, the reparse can be skipped.

The parser symbol table journal playout can affect the state of the symbol
table, in particular for entering new names, and entering or leaving scope. This
causes a problem when playing out a journal entry causes a table change, and a
later journal entry for the same increment causes the journal to report failure.
The journal-caused change must be backed out before the reparse, which will
cause its own change as a matter of course. This problem was carefully avoided
in the incremental preprocessor by never combining a \#define action into a
larger increment. That choice is not available in parsing, where the increments
are determined by the nesting structure of the language.

The six entries for the parse symbol table are extended by adding comple-
mentary back-out routines where necessary[4] (see Table 2).

The journal playout function recurs as it follows and executes the journal.
If it successfully reaches the end of the journal, it returns from the recursions
without further action. If it fails at some point before reaching the end of the
journal, it returns from the recursion, applying the Un versions of the functions
before each return.

The parse symbol table routines may generate user diagnostics but it is not
necessary. Specifically, if one name is declared twice in a scope (a C error),
the second declaration can be allowed to override for parsing purposes. If type
analysis follows the parse pass, the duplicate declaration will be rediscovered
there.
typedef struct {private layout} SrInc

w = SrIncFirst(w)  // iterator functions
w = SrIncNext(w)  // start at head of current file
w = SrIncPrev(w)  // step to next increment

SrIncPpIncDelete(j)  // list manipulation functions
SrIncPpIncInsert(j, a)  // ppinc j has been deleted
                        // ppinc j inserted ahead of ppinc a

w = SrInc2Sr(w)  // attribute functions
w = SrIncOfSr(t)  // first shift/reduce in srinc w
k = SrIncMade(w)  // srinc associated with shift/reduce t
b = SrIncJournal(w)  // time of construction of srinc w
                      // value of journal for srinc w

Table 3: Parser Increment Interface

Skipping

The requirement to skip some parsing leads to primitives for navigating and managing the shift/reduce increments (see Table 3).

The mechanics of skipping are achieved by a slight modification of a few of the recursive parsing routines, using routines SrIncValid() and SrIncReUse() which are constructed out of the increment navigation functions and journal playouts. For example, Tables 4 and 5 contain the recursive parser for the non-terminal statement. This parsing function follows the Standard C grammar definition for statement:

```
statement
    labeled-statement
    compound-statement
    expression-statement
    selection-statement
    iteration-statement
    jump-statement
```

There is a small bit of code at the function head to reuse shift/reduce increments and skip the recursion. If the current increment is invalid, it is discarded. Even when a shift/reduce sequence is discarded, valid increments nested within it will be reused. If the recursion is not skipped, a new empty increment bucket
is built, filled with a newly constructed shift/reduce sequence, and inserted in the shift/reduce sequence data structure.

Syntax Errors

A parser is expected to detect every syntax error. Parser variability comes in (1) the quality of the issued diagnostic, (2) the possibility of the parse continuing beyond the error point, (3) the possibility of later phases of the translation, up to and including test execution, using parser output despite syntax errors.

The incremental parser invites rethinking these questions. Our philosophy is that elaborateness of a diagnostic message is best determined by the subtlety of the error and the experience level of the user. Trivial errors deserve trivial diagnostics. Elaboration should be available, but not the default behavior.

The location of a syntax error is the unacceptable token. Because in C that token can have come from a source file, the command line (-D flag), a macro parameter, body, or token creating operator (# or #), the locator mechanism must be fairly general. The trivial locator is either the file position of the token or the file position of the outmost macro invocation giving rise to the token. More sophisticated locators can trace the creation history of the token back through macro expansions to its source origins. As discussed in the earlier papers, the source file line is not an invariant of an increment—earlier insertions and deletions can change it. Our solution was to keep enough relative information in the increments so as to compute line numbers afresh each time they were needed.

We implemented four levels of error diagnostics: trivial, elaborate, reference, and help. All located the problem point in the input source. Trivial diagnostics were limited to a single line of diagnostic text. Elaborate diagnostics often included snippets of source text in the message. Reference diagnostics gave pointers to the published literature. Help diagnostics guessed the root cause, and gave one or more bits of advice on what the user might do to solve the problem.

At the detection of first error, the elaborate diagnostic is limited to the standard parser response, "While trying to do this, the parser detected that; only one of these would have been acceptable". The reference message might display a few grammar rules from the C Standard, and give page references to a C textbook. The help message might, for example, note that nearby the error point there was an identifier spelled external and guess that perhaps the user intended extern.

Continuing parsing beyond the first error is accepted practice. The immediate motivation is to supply more information to the user—multiple diagnostics are more useful than rerunning after fixing the first error. The tradeoff is that the user must look at occasional irrelevant messages from cascaded errors but need not wait for a second compile.

With incremental techniques, turnaround is so quick that waiting is not a
static void SrStatement(void) {
    if (SrIncValid()) {
        /* is clean */
        SrIncReUse();
        /* skip ahead */
        return;
        /* do not recur */
    }
    SrIncNew();
    /* new bucket */
    switch (PpClass(PpNext())) {
    case caseTOKEN: case defaultTOKEN:
        SrLabeledStatement();
        /* recur */
        SrRulePut(statement.1);
        /* into s/r incr */
        break;
    case identifierTOKEN:
        if (PpClass(PpLookahead()) == colonTOKEN) {
            SrLabeledStatement();
            /* recur */
            SrRulePut(statement.1);
            /* into s/r incr */
        } else {
            SrExpressionStatement();
            /* recur */
            SrRulePut(statement.3);
            /* into s/r incr */
        }
        break;
    default:
        SrExpressionStatement();
        /* blindly recur */
        SrRulePut(statement.3);
        /* into s/r incr */
        break;
    case leftbraceTOKEN:
        SrEnterScope();
        /* typedef table */
        SrCompoundStatement();
        /* recur */
        SrRulePut(statement.2);
        /* into s/r incr */
        break;
    case ifTOKEN: case switchTOKEN:
        SrSelectionStatement();
        /* recur */
        SrRulePut(statement.4);
        /* into s/r incr */
        break;
    case doTOKEN: case whileTOKEN: case forTOKEN:
        SrIterationStatement();
        /* recur */
        SrRulePut(statement.5);
        /* into s/r incr */
        break;
    }
}

Table 4: Recursive Parser for statement, Part 1 of 2
Table 5: Recursive Parser for statement, Part 2 of 2

problem. Continuing the parse causes a new technical problem. The previous execution saved shift/reduce sequences beyond the error point. Each diagnostic issued necessarily invalidates the containing shift/reduce sequence. The result of issuing multiple diagnostics is increased invalidation, therefore increased re-parsing, therefore increased turnaround time. We chose to stop parsing after the first syntax error. We marked the error point in the shift/reduce sequence, allowing partial processing by later passes, most particularly the type checking mechanisms.

Testing

Testing a parser means running it over a challenging set of source texts and verifying that the correct shift/reduce sequence is produced. There is some payoff in separating the test input into many small tests, each of which exercises a small part of the input language. The minimal criterion for test input is full coverage of the recursive parser, including calls to all diagnostic routines. It is assumed that the parser will have firewalls for all resource limitations—they too should be executed. There will be several hundred small tests of this kind to test a C parser. Using the standalone testing methodology, we tested our parser in the absence of scanner and preprocessor by providing a stub scanner in the test jig.

It is beyond human patience to eye-verify the correctness of a large number of shift/reduce sequences. Our solution is to build an unparsert tool to interpret the shift/reduce sequence, verify its internal consistency, and reproduce the original input. The token abstractions described in earlier papers provide the needed information for recreating the textual form of the source. This reduces checking the correctness of a parse to file differencing—the reconstituted input should be exactly what came in. Details such as comments and formatting can be dealt with by re-inserting the original whitespace or indenting the output to the same style standards as the input. Introducing an additional source of programming errors from an incorrect unparsert tool causes no trouble in practice—the chance of compensating errors is very small, therefore one just debugs parser and unparsert
together.

Testing an incremental parser is harder than testing a parser. Grinding through a standard test suite only means the non-incremental parsing passes. For each such one-time test, one can envision a random change to the input, a reparse, a restoration of the input, a reparse and then a check that the shift/reduce sequence before the two compensating changes is the same as the shift/reduce sequence after. For our C parser, for each test, we systematically and automatically deleted each line, reparsed, reinserted the line, reparsed, and checked. This trick fit well with our incremental algorithm. Obviously, this is still a weak test strategy yet already very expensive.

Summary

We implemented two fine-grain incremental parsers. The prototype handled the Dijkstra 12-21 language extended to implement functions in separately compiled modules. Our industrial-strength C parser treated Standard C and all popular extensions, and parsed was very fast. It is difficult to directly compare our parser performance to other parsers which exist only within their containing compilers, but we feel safe in predicting a speedup factor of 30. The corresponding cost is a similar factor in increased memory use—you avoid recreating state by saving a lot of it. While virtual memory provided the required storage, and careful engineering avoided page thrashing, we found many rarely touched system limits left us uncertain whether our solution would scale to large programs (over 100KLOC).

The fundamental idea, that of attaching a journal of external access activity to an increment and replaying it to solve the action at a distance problem, is applicable in many problem domains.

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. His current technical interests are studying and improving compile speed and responsiveness and the application of Software Engineering techniques to small programming projects. 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.