Some Topics in Parser Generation

CERIEL J. H. JACOBS

Dept. of Mathematics and Computer Science
Vrije Universiteit
de Boelelaan 1081
1081 HV Amsterdam
The Netherlands

SUMMARY

An algorithm has been developed that generates an error-correcting recursive-descent syntax analyzer (parser) with no backtrack from an extended context-free grammar. A program, LLGEN, has been written to implement this algorithm. The paper discusses three aspects of the algorithm: the support of separate compilation, the mechanism for the static or dynamic resolving of conflicts, and the error-correction method used by the generated parsers. The cost of an LLGEN-generated parser as compared to table-driven parsers is discussed. LLGEN, and the parsers it generates, are written in the programming language C.[1] LLGEN is part the Amsterdam Compiler Kit.[2]

KEY WORDS Parser Generator Recursive-descent Syntax-directed error-correction Compiler-compiler

INTRODUCTION

Many computer programs deal with input of some kind. Such a program may either accept or reject its input, thereby defining a language: the input that is accepted belongs to the language, the input that is rejected does not. However, it often works the other way around: a language is defined, usually by specifying a context-free grammar for it, and then a program is written, accepting and processing input that belongs to the defined language. If the program is written properly, it also rejects input that does not belong to the language. The part of a program that either accepts or rejects input is called a *parser*.

Obviously, parsers constitute an important part of many computer programs. This is particularly so in a compiler for a programming language, where the grammar of the language is often used as a framework on which to hang semantic actions. These actions are then attached to the productions of the grammar, and are called by the parser at the appropriate times. This technique, developed by Irons,[3] is known as a *syntax directed translation scheme*.

As parsers are needed often, much effort has been invested in the automatic construction of parsers from context-free grammars. This effort has resulted in the development of many so called *parser generators*, programs that generate a parser for a language defined by a grammar belonging to a certain subclass of context-free grammars (see Meijer and Nijholt,[4] Burgess and James,[5] and Wood[6] for bibliographies). Parser generators are often used in compiler construction.[7,8] Their use has several advantages over writing a parser by hand: the parser generator verifies that the grammar has no ambiguities and causes no conflicts, it produces a parser accepting exactly the language defined by the grammar, and the generated parser usually offers a certain level of error-recovery, that is, it will try to recover from errors in the input, and continue parsing.

Parsing algorithms can be classified into two categories, *top-down* and *bottom-up*. A top-down parser starts with the start symbol of the grammar, and tries to derive the input from the grammar, whereas a bottom-up parser starts with the input, and tries to reduce it to the start symbol. Most bottom-up parsers consist of a set of tables and a (usually language-independent) routine that interprets those tables (i.e. they are *table-driven*). Top-down parsers usually are either table-driven, or use the technique of *recursive-descent*, where the parser has one recursive procedure for each nonterminal.

For our parser generation algorithm, the recursive-descent method was chosen, because of the ease in which actions can be incorporated in the resulting parser. Other recursive-descent parser generators have been discussed by Lewi et. al.,[9] and by Anderson et. al.[10]

This paper discusses three aspects of the parser generation algorithm: support of separate compilation of parts of the resulting parsers, a static and a dynamic method for resolving conflicts, and the error-correction method of the resulting parsers. Before discussing these topics, we give a short overview of LLGEN specifications in the next section. Then, a section is devoted to each of the topics mentioned above, and finally, a section will discuss the cost of an LLGEN-generated parser.

A SHORT OVERVIEW

An LLGEN specification consists of an extended context-free grammar, usually with embedded actions. In this section, a short description of such an extended context-free grammar is presented, along with the way in which actions can be embedded in this grammar.

Extended Context-Free (ECF) grammars

The extensions of an ECF grammar with regard to an ordinary context-free grammar are:

1. An ECF grammar allows *term N* (*N* represents a positive integer) to mean *N* occurrences of *term*.
2. An ECF grammar allows *term* * to mean zero or more occurrences of *term*, and *term *N* (*N* again represents a positive integer) to mean *N* or fewer (including 0) occurrences of *term*.
3. An ECF grammar allows *term* + to mean one or more occurrences of *term*, and *term +N* to mean *N* or fewer (excluding 0) occurrences of *term*.
4. An ECF grammar allows square brackets '[' and ']', which can be used for grouping.
5. An ECF grammar allows *term* ? to mean 0 or 1 occurrences of *term*.

Figure 1 gives an ECF grammar of ECF grammar. Terminal symbols are represented by single characters between quotes or by identifiers, here in upper case. The identifier "NUMBER" represents a positive integer, the identifier "LITERAL" represents any character between quotes, and the identifier "IDENTIFIER" represents an identifier.

```
grammar             : grammarrule+ ;
grammarrule         : nonterminal ':' righthandside ';' ;
nonterminal         : IDENTIFIER ;
righthandside       : production [ '|' production ]* ;
production          : term* ;
term                : element repetition_specifier? ;
element             : LITERAL | IDENTIFIER | '[' righthandside ']' ;
repetition_specifier : NUMBER | '*' NUMBER? | '+' NUMBER? | '?' ;
```

Figure 1. The ECF grammar of ECF grammar

Thus, terminal symbols (tokens) are indicated by either a LITERAL or an IDENTIFIER. Tokens may consist of several characters. A *lexical analyzer* must read the input, identify tokens and pass them to the parser. A useful tool for constructing such a lexical analyzer is the *lex* program.[11]

The class of ECF languages is identical with the class of CF languages. However, recursive definitions of language features can now often be replaced by iterative ones. This tends to reduce the number of non-terminals and results in efficient recursive-descent parsers. Also, many specification grammars are written in an ECF-like style, and thus can be mapped easily onto an ECF grammar.

Actions

Usually, a program not only accepts/recognizes input, but also performs some operations on it. Therefore, many parser generators allow the association of *actions* with grammar rules. Each time a grammar rule is recognized, the associated actions will be executed.

As LLGEN-generated parsers use a top-down predictive parsing algorithm (recursive-descent with no backtrack), LLGEN allows arbitrary insertions of actions within the right-hand-side of a grammar rule in the ECF grammar. Since LLGEN generates a parsing routine for each grammar rule, it can just insert the actions on the proper place, without an explicit connection. On the other hand a predictive table-driven parser needs some value in the table (the connection) to designate each action.

Bottom-up parsers can only execute an action after they have recognized a grammar rule. To do actions before recognizing a particular construct, *hooks* are introduced. Hooks are productions created for the sole

purpose of allowing an action to be done before the construct is recognized completely. Some bottom-up parser generators generate such hooks automatically,[12] but this may cause unexpected and obscure conflicts.

An action is specified by some statements in the programming language C,[1] enclosed in the curly brackets '{' and '}'. Declarations may precede the statements in an action, as the opening and closing brackets '{' and '}', which are scope delimiters in C, are copied into the target code along with the action. The scope of those declarations terminates with the closing bracket '}' of the action.

In addition, each parsing routine (= grammar rule) can have local variables and parameters that can be used in the actions. This allows for an easy implementation of L-attributed grammars.[13] It should, however, be noted that LLGEN is not an attribute grammar evaluator: it does not check order dependencies. It does not check the contents of actions at all. The parameter passing mechanism is that of C, call-by-value.

All this gives the grammar writer the feeling that he is programming in a language based on C, but with some added control structures. By allowing arbitrary insertions of actions within the right-hand-side of a production rule and letting nonterminals have parameters, LLGEN supports this view.

SEPARATE COMPILATION SUPPORT

C, like many programming languages, allows programs to be split up into several files. These files can be compiled separately and then linked together, so changing one file does not require recompiling the others. This feature speeds up program development. As LLGEN input can be viewed as a programming language, consisting of a mixture of ECF and actions, and LLGEN itself can be viewed as a compiler, recognizing this language and generating C, the question arises whether LLGEN input can be compiled separately.

Traditional parser generators expect their input on one file. Recompilation after a change in this file requires a complete run of the parser generator and compilation of the resulting parser, which usually is a time consuming process.

Allowing the input for the parser generator to be split up into several files is a first step towards separate compilation. This allows the parser generator to generate for each input file the corresponding part of the parser, on a separate file. However, a change in one input file still requires a complete run of the parser generator and a compilation of each of its output files, because these output files will have been updated.

A next step would be to let the parser generator compile the input separately. Then, only changed parts of the input would have to be processed by the parser generator, only the corresponding parts of the parser would be updated, and only these parts would have to be recompiled. Unfortunately, this is impossible for LLGEN, for two reasons:

1. A change in one part of the grammar may not only affect the corresponding part of the parser, but also require changes in other parts. For instance, one part of the grammar might contain the rule

   n_1 : n_2 ['x' | 'y'] ;

   and another part might contain

   n_2 : 'u' | 'v' | ;

   Now suppose the first rule is changed into

   n_1 : n_2 ['x' | 'y' | 'z'] ;

   This change will not only affect the code generated for the nonterminal n_1, but also the code for n_2, because its last alternative must now also be chosen on the token 'z'.

2. LLGEN needs the whole grammar, even for generating only part of the parser. For instance, one part of the grammar might contain the rule

   statement : if_statement | assignment_statement ;

   and the rules for *if_statement* and *assignment_statement* might be declared in another part. LLGEN needs those rules to generate code for *statement*, because it must for instance decide which tokens must cause the generated parser to invoke the rule for *if_statement*. LLGEN also needs those rules to detect possible conflicts.

So LLGEN itself cannot compile its input separately and the question arises whether the files containing the generated parser can be compiled separately. The answer is affirmative, because the parser is written in C, and C supports separate compilation. But nothing is gained if all parts of the parser are updated each

time the parser generator is run. Therefore, the following solution has been adopted: the source text of the grammar may be kept in several files, but these files must all be offered to LLGEN. For each of these files, LLGEN will first generate a temporary file containing the corresponding part of the parser, then compare the temporary with the previous version of this part, and update this part only if there is a difference. This allows the re-compilation of only the updated versions. Thus the grammar-to-C step is required for all the files when any change is made, but the C-to-object step is only needed for those files that have actually been affected by the changes to the grammar files.

The UNIX† environment features the *make*[14] program, which allows the user to specify which files depend on which others, and to associate commands with each dependency. When invoked, it will update one or more targets with a minimum amount of work. Make is a useful tool for maintaining and developing programs whose source consists of more than one file. It can be made to invoke LLGEN, find out which parts of the generated parser are updated, and recompile the updated parts only.

This support of separate generation has proven to be a useful feature during the development of parsers and compilers. Although a small change in the grammar may change the whole parser, and thus require a complete re-compilation of the parser, experience has shown that this does not happen often. Also, changing an action in a part of the grammar does not change the grammar, and thus only affects the corresponding part of the parser, so that only this part must be recompiled. The inability of LLGEN to compile separately turned out not to be much of a nuisance, because LLGEN generates parsers particularly fast. The time spent in generating a parser is almost negligible compared to the time spent in compiling the parser.

AMBIGUITIES AND CONFLICTS

As an LLGEN-generated parser is recursive-descent with no backtrack, and has only a one token look-ahead, it must always be able to decide what to do, based on the current input token. Therefore, the grammar must be LL(1). Unfortunately, not all ECF grammars are LL(1), and given an arbitrary ECF grammar, it is undecidable whether there exists a LL(1) grammar for the language defined by the former grammar.[15] This section discusses conflicts, and what can be done about them.

Conflicts

Two kinds of conflicts can occur in an ECF grammar:

1. a grammar rule is of the form 'S_1 | S_2', and the parser cannot decide which production to choose. We call this an *alternation conflict*. For instance, consider the following grammar:

   n_1 : n_2 | n_3 ;
   n_2 : 'x' n_3 | 'y' ;
   n_3 : 'y' n_2 | 'z' ;

   The first grammar rule contains an alternation conflict: when the parser sees "y", it cannot tell whether to start n_2 or n_3.

2. a grammar rule contains the form 'C^X' where X specifies a variable repetition count (including a "?"), and the parser cannot decide whether to choose C (once more), or to continue with the rest of the rule. This we call a *repetition conflict*. For instance, consider the following grammar:

   n_1 : n_2* 'x' ;
   n_2 : 'x' 'y' | 'z' ;

   The first grammar rule contains a repetition conflict, because when the parser sees "x", it cannot tell whether it has to choose n_2 once more, or to continue with the rest of the rule for n_1.

There are two causes of conflicts: the grammar is ambiguous, or it requires a more complex parser than LLGEN can construct. LLGEN offers a facility for printing the relevant sets of tokens, allowing the user to analyze a conflict.

Conflicts can be resolved in several ways. Often, although not always, it is possible to rewrite the grammar in a new form, the new grammar being LL(1). Two techniques for doing so are

1. The elimination of left recursion.
   Left recursive rules always cause conflicts and can always be eliminated.[15]

† UNIX is a Registered Trademark of AT&T Bell Laboratories.

2. Factorization and substitution.
   For instance, a rule

   ’a’ B | ’a’ C

   could be rewritten as

   ’a’ [B | C]

   thereby eliminating the conflict on the token ’a’.

Discussions on this subject are given by Foster[16] and by Griffiths.[17]

Conflict resolvers

Rewriting of a grammar may lead to unnatural and large grammars. Also, it is not always possible to rewrite a grammar to an LL(1) form. Therefore, LLGEN offers a mechanism for dealing with conflicts, using static or dynamic *conflict resolvers*.

The mechanism of dynamic conflict resolvers is based on the attributed parsing of Milton.[18] It works as follows: when the parser arrives at a point where it cannot choose based on the current input token, it will base its choice on the evaluation of a user-supplied conflict resolver. This conflict resolver may inspect some attributes of the current token, or it may cause the lexical analyzer to look further ahead. Anything is possible, because the conflict resolver is just a C-expression.

Alternation conflicts

To resolve an alternation conflict, a conflict resolver, consisting of the keyword ‘*%if*’, followed by a C-expression between parentheses, is put in front of the first conflicting alternative. The parser will evaluate the expression whenever a token is met at this point on which there is a conflict. If the expression evaluates to non-zero (*true*, although C has no booleans), the first conflicting alternation is chosen, otherwise a remaining one is chosen. Thus, the conflict is resolved dynamically, by the parser. Figure 2 gives a small example in a case where the conflict could also have been resolved by factorization.

```
statement : %if (is_procedure(current_token))
                IDENTIFIER parameterlist?
          | %if (is_array(current_token))
                IDENTIFIER ’[’ expressionlist ’]’ ASSIGN expression
          |     IDENTIFIER ASSIGN expression
;
```

Figure 2. An example of a dynamic conflict resolver

A special case of a conflict resolver is a constant expression, either always evaluating to non-zero, or always evaluating to zero. In such a case, the conflict can be resolved statically, by LLGEN. This may be indicated by the keywords ‘*%prefer*’ and ‘*%avoid*’, equivalent in behavior to ‘*%if* (1)’ and ‘*%if* (0)’ respectively. The difference is that LLGEN will recognize the former as static conflict resolvers and produce more efficient code for them.

A classical example of a conflict that can be resolved statically is a fragment from a programming language involving an ‘if-then-else’ construction:

```
statement:      IF condition THEN statement [ ELSE statement | ]
```

This rule forms an ambiguous grammar, because the statement

IF *condition*$_1$ THEN IF *condition*$_2$ THEN *statement*$_1$ ELSE *statement*$_2$

can be parsed in two ways. The ELSE could be paired to either the first or the second IF. Let us suppose that the language definition further states that an ELSE must be paired with the closest IF that is not already paired with an ELSE. This would lead us to the following grammar fragment:

```
statement:      IF condition THEN statement [ %prefer ELSE statement | ]
```

Repetition conflicts

To resolve a repetition conflict a conflict resolver, consisting of the keyword ‘*%while*’ followed again by a C-expression between parentheses, is put right after the opening square bracket of the repeating term. Again, the parser will evaluate the expression whenever a token is met at this point, on which there is a

conflict. If the expression evaluates to non-zero the repeating part is chosen, otherwise the parser continues with the rest of the grammar rule.

An example

Figure 3 gives another example of dynamic conflict resolvers. It consists of part of an LLGEN specification of a parser for arithmetic expressions. This parser accepts arithmetic expressions made up of numbers and operators, taking the operator priorities into account. The ECF grammar used is highly ambiguous. Every expression can be parsed in infinitely many different ways.

```
/*
 * This is a comment. LLGEN accepts C-like comments.
 */

/*
 * Tokens that are given a name must be declared. This way a mistyped nonterminal
 * will not be mistakenly accepted as a token name.
 */
%token  OPERATOR,               /* any operator */
        NUMBER;                 /* an integer */

start:
    expression(1)               /* invoke the rule for expression with priority 1 */
;

expression (int level;) :       /* a rule with one parameter */
    %if (level <= MAXPRIO)
    expression(MAXPRIO + 1)     /* A left-recursive grammar rule.
                                 * Of course a conflict resolver is needed.
                                 */
    [
      %while (priority(current_token) >= level)
                                /* Eat operators as long as their priority is higher than or
                                 * equal to the priority level of the current invocation.
                                 */
      OPERATOR expression(priority(current_token) + 1)
                                /* It is assumed that the operators are left associative.
                                 * For right associative operators the call should be :
                                 *     expression(priority(current_token)).
                                 */
    ]*
|
    '(' expression(1) ')'
|
    NUMBER
;
```

Figure 3. An example of the use of conflict resolvers

The example demonstrates the use of conflict resolvers, in a case where the conflicts could also have been resolved by an extensive rewrite of the grammar. This rewrite, however, would have to take into account the number of operator priorities. In an ordinary LL(1) grammar there would be one nonterminal for each operator priority. The example shows how it can all be done with one grammar rule, no matter how many operator priorities there are.

ERROR CORRECTION

Input submitted to a parser often contains syntactic errors. A good parser not only detects such errors, but also recovers from them. This is difficult, even more so because the parser may have invoked semantic actions. It usually is not acceptable to just restart the parser at another point in the grammar, because the

actions might have allocated a chunk of memory, which was expected to be initialized in a later action, and now the latter action will not be executed. This obviously will not contribute to the reliability of the parser.

So, LLGEN-generated parsers do error-correction. The input string is automatically edited at the point of error-detection, deleting or inserting symbols, until a syntactically correct input results. An important property of this method is that, once a grammar rule is started, it will be finished. This means that all actions in a grammar rule will be executed in an order that could also occur because of a syntactically correct input. So, actions only have to deal with syntactically correct input and thus, syntax errors in the input will cause no inconsistencies for the actions.

An informal discussion of our error-correction technique

The error-correction technique used by LLGEN is a modification of the one presented by Röhrich.[19] and belongs to the class of follow-set error-recovery schemes.[20] The central idea of a follow-set error-recovery scheme is as follows. At any moment, the parser is in a certain "state". With each "state" q a set fol(q) of tokens is associated. This set contains the tokens that are not to be skipped in this "state". When an error occurs, an error message is emitted and tokens are skipped until a token $t \in$ fol(q) is reached. Next, the parser will advance to a state q_1, in which the token t is accepted.

Follow-set error-recovery schemes differ in their definition of fol(q), and in the method used to advance from state q to a state q_1 in which a non-skipped token is acceptable. The rest of this section discusses our scheme, which is based on so called *default choices*, informally. In particular, our definition of fol(q), and our method of advancing from one state to another in case of an error are described. More formal definitions, based on the formalism of Stirling,[20] are given in an appendix.

At every point in the grammar where there is a choice, a *default choice* is introduced. In an alternation, one alternative is marked as a default choice; in a term with a variable repetition a default choice is made between doing the term (once more) and continuing with the rest of the grammar rule in which the term appears. These default choices are explicitly or implicitly specified in the grammar. By default, the default choice in an alternation is the one with the shortest possible terminal production. However, the grammar may specify another alternative as default choice. The default choice for

$C*$ and $C?$

is to continue with the rest of the grammar rule, and

$C+$

is treated as

$C\ C*$

The default choices will be used to describe the set fol(q) for each state q, but we must first give an account of what a state of a recursive-descent parser is. Such a state can be described with a grammatical representation of the part of an input string that has yet to be recognized. Consider for instance the following grammar:

n_1 : n_2 'e' 'f' ;
n_2 : 'a' 'b' n_3 ;
n_3 : 'c' 'd' | 'u' 'v' 'w' ;

After having scanned an 'a', the parser would be in a state q_1 represented by

'b' n_3 'e' 'f'

Now suppose the next token is a 'u'. This is an error, because 'b' is expected. Now, a default continuation is computed from the representation of state q_1, along the default choices. Here, the default continuation consists of the tokens

'b' 'c' 'd' 'e' 'f'

Informally, but not quite accurately, fol(q) is the set of all tokens that either occur in a default continuation from the state q, or may legally start a deviation from this default continuation. Here:

fol(q_1) = {'b', 'c', 'u', 'd', 'e', 'f'}

When the parser, in state q, detects an error, it computes the set fol(q), and then discards zero or more tokens from its input, until a token $t \in$ fol(q) is found. Next, it continues parsing, following the default

choices, inserting tokens along the way, until the token *t* is legal. The advancing from a state *q* to a state q_1 in case of an error is thus performed by following default choices. So, in the example, the parser will find that 'u' ∈ fol(q_1), so that 'u' will not be skipped. Instead, the token 'b' is inserted and the parsing will be continued with the second alternative of nonterminal n_3.

As the computation of a default continuation must be a finite process, and the parser must not loop, inserting tokens forever, LLGEN imposes the restriction on the grammar, that default choices may not invoke themselves recursively. LLGEN detects and disallows recursive default choices.

Terms with a variable repetition count sometimes call for a more persistent treatment. Consider for instance the grammar rule

```
statementlist : statement [ ';' statement ]* ;
```

When a syntax error comes up at the start of the repeating term, with the parser in a state *q*, tokens will be skipped until either ';' or a token that may legally follow "statementlist" is encountered. Thus, probably a full 'statement' will be skipped, just because a ';' was missing. In such a case, the term can be marked *persistent*, by changing the grammar rule into

```
statementlist : statement [ %persistent ';' statement ]* ;
```

This has no effect on the default continuation, but with regard to fol(*q*), the term will now be treated as if it occurred on the default continuation. So, tokens that are an element of fol(*q*) because of the persistent term will not be skipped. The default choice is determined by the parser, depending on the token at hand. For instance, if "statement" may start with the token "IDENTIFIER", and this token is offered to the parser in state *q*, it will not be skipped. Instead, a ';' will be inserted.

Dealing with empty productions

Allowing grammar rules to produce the empty string may cause some complications. For instance, consider the following grammar:

```
n1 : 'x' n2 | 'u' n2 'z' ;
n2 : %default 'y' 'z' |  ;
```

and consider the behavior of the resulting parser on the input 'zuz', using the informal definition of fol(*q*) given above. The parser detects an error on input 'z'; let's say it is in state *q* when detecting it. The parser then proceeds by computing the default continuation and the set fol(*q*). Here,

fol(*q*) = { 'x', 'y', 'z', 'u'},

so the token 'z' will not be skipped. Then the parser will follow the default choices, inserting the token 'x'. Then, the nonterminal n_2 is called for. Now, the second alternative of n_2 will be taken, because 'z' may legally follow n_2, and this second alternative can produce the empty string. So, the parser is finished with n_2, without having parsed 'z'. It will now proceed by skipping 'z', 'u' and 'z'.

Although the parser has corrected the error, it has done so badly. The problem is, that at first 'z' was not skipped because it was expected to be parsed by n_2. Unfortunately it was not, so it was skipped after all, but much to late. To prevent this problem, care has to be taken that tokens that are a member of fol(*q*) because of a certain grammar rule, **will** be parsed at this grammar rule. In other words, the tokens on which this problem could occur must be removed from fol(*q*). This is reflected in the formal definition of fol(*q*), as given in the appendix.

Now, let us analyze the same problem again, using the formal definition of fol(*q*). Here, 'z' ∉ fol(*q*), so 'z' will be skipped. On token 'u', the second alternative of nonterminal n_1 will be chosen, 'u' will be parsed, the second alternative of nonterminal n_2 will be invoked on the second 'z', and finally the second 'z' will be parsed. The parser thus has corrected the error by deleting the first 'z'.

A more realistic example

Figure 4 gives the output of an LLGEN-generated Pascal-parser for an erroneous Pascal program. The correction of the procedure-header demonstrates the backtrack-free nature of the correction method. Once a token is accepted, nothing can be done about it; also, nothing can be changed in front of this token. This is also shown by the deletion of the keyword "**then**". A backtracking error-correction method might have inserted an "**if**" in front of the erroneous statement.

```
1       program p(input,output);
2       var i: array 1..2 1..2] of integer;
***                   ^ '[' inserted
***                      ^ ',' inserted
3       procedure p (k : integer;);
***                                ^ 'identifier' ':' 'identifier' inserted
4       begin
5               i[k,1] then k
***                     ^ 'then' deleted
***                          ^ ':=' inserted
6       end
7       begin p begin
***     ^ ';' inserted
***               ^ ';' inserted
8                 writeln('----------')
9                 p(1)
***               ^ ';' inserted
10      end.
***         ^ 'end' inserted
```

Figure 4. An example of error-correction in a Pascal program

COST OF AN LLGEN-GENERATED PARSER

Generated parsers often consist of a set of tables and a routine interpreting them. A recent paper by Waite and Carter[21] discusses the cost of such a generated parser with regard to a (hand-written) recursive-descent parser. To summarize, they explain that there appear to be four important determinants in the cost of executing the resulting compiler:

1. The mechanism for allocating attribute storage.
   In a recursive-descent parser, storage for most attributes is allocated on the stack as part of the activation record of the parsing procedure, whereas the table-driven parser must allocate (and reallocate) this storage explicitly, sometimes circumventing the type mechanism of the implementation language. The latter often is more expensive in time and space.
2. The method by which a semantic action is determined.
   In a recursive-descent parser, the semantic actions are directly embedded in the parsing procedures, whereas in a table-driven parser a connection is needed between the actions and the parser, bringing about at least some overhead.
3. Using semantic information to guide the parsing process.
   This facility permits smaller grammars to be used and is often used in recursive-descent parsers to resolve conflicts.
4. Interpretation of parse tables versus direct execution of recursive-descent procedures.
   This tends to make recursive-descent parsers faster and larger than generated parsers. The more efficient attribute storage in recursive-descent parsers may compensate a little for the extra space the parser itself takes.

All these cost determinants, except maybe the fourth, are in favor of a recursive-descent parser, and also apply to a recursive-descent parser generated by LLGEN:

1. LLGEN-generated parsers use the same method for attribute storage allocation as a hand-written recursive-descent parser does.
2. In an LLGEN-generated parser, actions are directly embedded in the parsing routines.
3. LLGEN allows semantic information to guide the parsing process; see the section on conflicts and ambiguities.
4. LLGEN generates parsing procedures that are executed directly.

Some measurements

We have compared an LLGEN-generated parser with a parser generated by Yacc.[12] Yacc is a popular LALR(1) (bottom-up) parser generator. Fortunately, grammars of the C programming language were available for both parser generators: for Yacc the grammar of the Portable C compiler[22] was used; for

LLGEN the grammar of the new C compiler[23] for the Amsterdam Compiler Kit[2] was used. All actions in both grammars were deleted.

To avoid influences of different lexical analyzers, for both parsers a file was created, containing the tokens as integers in the machines internal representation. Each of these files were generated from a C program containing about 10,000 tokens. The work of the lexical analyzer was thus reduced to reading the next integer.

Tables I, II, and III show the results on different machines. The time used by the lexical analyzer is also given. Times are given in seconds; sizes are given in bytes. The sizes given are the sizes of the parser objects only. Sizes of the lexical analyzer and an error-reporting routine are not counted.

Table I. Cost comparison on a DEC VAX 11/750 running 4.1 BSD Unix

	LLGEN-parser	Yacc-parser	lexical analyzer
Execution time	2.6	3.5	0.8
Memory	11648	6316	
Code space	10728	712	
Data space	920	5604	

Table II. Cost comparison on a MC68010-based system running Unix System V

	LLGEN-parser	Yacc-parser	lexical analyzer
Execution time	1.4	1.9	0.4
Memory	14548	6348	
Code space	13628	756	
Data space	920	5592	

Table III. Cost comparison on a DEC PDP 11/44 running Version 7 Unix

	LLGEN-parser	Yacc-parser	lexical analyzer
Execution time	2.3	3.0	0.9
Memory	10112	6170	
Code space	7904	574	
Data space	2208	5596	

Subtracting the time used by the lexical analyzer from the total execution time gives us the time used by the parsers. The tables show what was expected: the LLGEN-generated parser is much faster, but also much larger than the Yacc-generated one. As memory becomes cheaper and cheaper, and even personal computers usually have at least ½ a megabyte nowadays, we regard faster execution as more important than saving a few thousand bytes of memory.

CONCLUSION

This paper has offered a discussion of some advanced topics of yet another parser generator, LLGEN. Its support of separate compilation is undoubtedly the most unusual feature. Yet, it was simple to implement, and has proven to save much time during the development of parsers. The conflict resolvers provide a method for dealing with conflicts, much as one would deal with them when writing a recursive-descent parser by hand. The error-correction by the resulting parser is not only useful for its eventual users, it is also helpful in debugging the grammar.

LLGEN has been used to develop a new C compiler[23] for the Amsterdam Compiler Kit.[2] Also, LLGEN's input analyzer is LLGEN-generated.

ACKNOWLEDGEMENTS

I am indebted to Dick Grune for his advice, his comments, and his [*%persistent* nagging]+ during the development of LLGEN, and to Dick Grune, Bruce McKenzie and Andy Tanenbaum for their comments on an earlier draft of this paper.

APPENDIX

In this appendix we will present some definitions that will lead to a formal definition of the fol(q) set.

Let G be an ECF grammar, N be the set of nonterminals and T the set of tokens of G. Let Λ represent the empty string. We will now define recursively two infinite sets: the set of 'ECF terms' ($ECT(G)$) and the set of 'ECF strings' ($ECS(G)$). A definition much like this one is also given by Stirling.[20] The one given here reflects our ECF notation.

Definition

(i) $(T \cup N) \subset ECT(G)$
(ii) if $C_1,..., C_n \in ECT(G)$ then their concatenation $C_1 \cdots C_n \in ECS(G)$
(iii) $\Lambda \in ECS(G)$
(iv) if $S \in ECS(G)$ then $[S] \in ECT(G)$
(v) if $C \in ECT(G)$ then
$C^X \in ECS(G)$, where X stands for one of ?, *, *N, +, +N, N
(vi) if S_1 , $S_2 \in ECS(G)$ then $S_1 \mid S_2 \in ECS(G)$

Intuitively $ECT(G)$ is the set of all possible elements in the grammar, and $ECS(G)$ the set of all possible right-hand-sides. Note that $ECT(G)$ and $ECS(G)$ are independent of the production rules in G and depend on N and T only.

The productions of the grammar G induce a relation $\Rightarrow$ on $ECS(G)$, defined by $S_1 \Rightarrow S_2$ iff there are $U,P,Q,V \in ECS(G)$ with $S_1 = UPV$, $S_2 = UQV$, and $P : Q$ is a production of the grammar. $\Rightarrow^*$ denotes the transitive closure of $\Rightarrow$.

Next, we define two set-valued functions: $FIRST(S)$, where $S \in ECS(G)$, is the set of all tokens which may be the first symbol in a string derived from S; $FOLLOW(S)$, where S is (part of) a right-hand-side of a grammar rule, represents the set of all tokens that form the start of an element of the set of all possible strings which can follow S.

Definition

$$FIRST(S) = \{t \in T : S \Rightarrow^* tW, \text{ where } W \in ECS(G)\}$$

Definition

(i) if $S \in N$ then
$FOLLOW(S) = \{t \in T : A \Rightarrow^* VStW$, where $V, W \in ECS(G)\}$,
where A is the startsymbol of G
$FOLLOW$(right-hand-side of S) = $FOLLOW(S)$
(ii) if $S = C_1 \cdots C_n$ then
if $S_n \Rightarrow^* \Lambda$ then
$FOLLOW(C_1 \cdots C_{n-1}) = FIRST(C_n) \cup FOLLOW(S)$
otherwise
$FOLLOW(C_1 \cdots C_{n-1}) = FIRST(C_n)$
(iii) if $S = [S_1]$ then $FOLLOW(S_1) = FOLLOW(S)$
(iv) if $S = C^X$ where X stands for one of N, +N, +, *N, * then
$FOLLOW(C) = FIRST(C) \cup FOLLOW(S)$
if $S = C$? then $FOLLOW(C) = FOLLOW(S)$
(v) if $S = S_1 \mid S_2$ then
$FOLLOW(S_1) = FOLLOW(S)$
$FOLLOW(S_2) = FOLLOW(S)$

Now we are ready to give a formal definition of fol(q). To simplify the definition a little, terms with repetitions containing + or N are disregarded, as they do not contribute to the clarity, and their use gives no extra problems. The definition of $ECS(G)$ can be used to represent a state of a recursive-descent parser. Such a state q can be represented by a $S \in ECS(G)$. This S is a grammatical representation of that part of an input string that has yet to be recognized. The set fol(q) can now be defined as a function from the set $ECS(G)$ into subsets of T. Suppose $S \in ECS(G)$.

Definition

(i) if $S \in T$ then fol(S) = {S}
if $S \in N$ then fol(S) = fol(right-hand-side of S)
(ii) if $S = C_1 \cdots C_n$ then fol(S) = fol(C_1) $\cup$ ... $\cup$ fol(C_n)
(iii) if $S = \Lambda$ then fol(S) = {}
(iv) if C = [S] then fol(C) = fol(S)
(v) if $S = C^*$ or $S = C?$ then
if the term is persistent then
fol(S) = fol(C) - (*FOLLOW* (S) - *FIRST* (C))
otherwise fol(S) = *FIRST* (C)
(vi) if $S = S_1 \mid S_2$ then
if S_1 contains a default choice then
if $S \Rightarrow^* \Lambda$ then
fol(S) = (fol(S_1) $\cup$ *FIRST* (S_2)) - (*FOLLOW* (S) - *FIRST* (S))
otherwise fol(S) = fol(S_1) $\cup$ *FIRST* (S_2)
otherwise fol(S) = *FIRST* (S_1) $\cup$ fol(S_2)

REFERENCES

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1978).
2. A. S. Tanenbaum, H. van Staveren, E. G. Keizer, and J. W. Stevenson, ‘‘A Practical Toolkit for Making Portable Compilers,’’ *Communications of the ACM* **26**(9), pp. 654-660 (September 1983).
3. E. T. Irons, ‘‘A syntax directed compiler for ALGOL 60,’’ *Communications of the ACM* **6**(11), pp. 669-673 (November 1961).
4. H. Meijer and A. Nijholt, ‘‘Translator writing tools since 1970: a selective bibliography,’’ *SIGPLAN Notices* **17**(10), pp. 62-72 (October 1982).
5. C. Burgess and L. James, ‘‘An indexed bibliography for LR grammars and parsers,’’ *SIGPLAN Notices* **16**(8), pp. 14-26 (August 1981).
6. D. Wood, ‘‘A bibliography of top-down parsing,’’ *SIGPLAN Notices* **13**(2), pp. 71-76 (February 1978).
7. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Massachusetts (1977).
8. W. M. Waite and G. Goos, *Compiler Construction*, Springer-Verlag, New York, NY (1984).
9. J. Lewi, K. De Vlaminck, J. Huens, and M. Huybrechts, ‘‘The ELL(1) Parser Generator and the Error Recovery Mechanism,’’ *Acta Informatica* **10**, pp. 209-228 (1978).
10. S. O. Anderson, R. C. Backhouse, E. H. Brugge, and C. P. Stirling, ‘‘An Assessment of Locally Least-Cost Error Recovery,’’ *Computer Journal* **26**(1) (1983).
11. M. E. Lesk, ‘‘Lex - A Lexical Analyser Generator,’’ Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murrey Hill, New Jersey (October 1975).
12. S. C. Johnson, ‘‘Yacc: Yet Another Compiler Compiler,’’ Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey (1975).
13. P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns, ‘‘Attributed Translations,’’ *Journal of Computer and System Sciences* **9**, pp. 279-307 (1974).
14. S. I. Feldman, ‘‘Make - A Program for Maintaining Computer Programs,’’ *Software - Practice and Experience* **10**(8), pp. 255-265 (August 1979).
15. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Massachusetts (1979).
16. J. M. Foster, ‘‘A syntax improving program,’’ *Computer Journal* **11**(1), pp. 31-34 (May 1968).
17. M. Griffiths, ‘‘LL(1) Grammars and Analysers,’’ in *Compiler Construction, An Advanced Course*, ed. F. L. Bauer and J. Eickel, Springer-Verlag, New York, N.Y. (1974).
18. D. R. Milton, L. W. Kirchhoff, and B. R. Rowland, ‘‘An ALL(1) Compiler Generator,’’ *SIGPLAN Notices* **14**(8), pp. 152-157 (August 1979).
19. J. Röhrich, ‘‘Methods for the Automatic Construction of Error Correcting Parsers,’’ *Acta Informatica* **13**, pp. 115-139 (1980).
20. C. Stirling, ‘‘Follow Set Error Recovery,’’ *Software - Practice and Experience* **15**(8), pp. 239-257

(March 1985).
21. W. M. Waite and L. R. Carter, ‘‘The Cost of a Generated Parser,’’ *Software - Practice and Experience* **15**(8), pp. 221-237 (1985).
22. S. C. Johnson, ‘‘A Tour Through the Portable C Compiler,’’ in *Unix programmer’s manual, Seventh Edition*, Bell Laboratories, Murrey Hill, New Jersey (January 1979).
23. E. H. Baalbergen, D. Grune, and M. Waage, ‘‘The CEM compiler,’’ IM-4, Vrije Universiteit, Amsterdam, The Netherlands (1985).