A grammar-interpreting parser in a language design laboratory

Elizabeth Bjarnason and Görel Hedin

LU-CS-TR:96-169

To be presented at the poster session of CC'96, International Conference on Compiler Construction, Linköping, Sweden, April 1996.

Department of Computer Science
Lund University

P.O. Box 118
S-221 00 Lund
Sweden

A grammar-interpreting parser in a language design laboratory

Elizabeth Bjarnason and Görel Hedin

Dept. of Computer Science, Lund University
Box 118, S-221 00 Lund, Sweden
e-mail: { Elizabeth.Bjarnason | Gorel.Hedin }@dna.lth.se

Abstract This paper describes the dynamic parser component of a language design laboratory. The laboratory is based on structure editing and supports interactive development of languages defined by grammars. Parsing is done according to the current state of the grammar. To achieve this dynamic behavior, the parser is based on grammar interpretation and uses an object-oriented recursive-descent parsing algorithm.

Keywords structure editing, grammar interpretation, language laboratory, language-based environments, application languages, recursive-descent parsing, object-orientation

1 Introduction

Designing a new language is an iterative process where different language constructs and their syntactic rules are prototyped and tested several times before the desired result is achieved. Having an environment that supports language design in an integrated, interactive way decreases the time and effort spent on this process. We call such an environment a *Language Laboratory*.

Our language lab, APPLAB (*Application Language Laboratory*), supports the interactive development of application-specific languages [1], and environments for these languages [Bja95]. The system supports iterative language design by allowing the user to switch freely between editing the language definition (the grammar) and editing a program in the (changing) new language. All the tools in APPLAB (parser, structure editor, and attribute evaluator) are *dynamic*, i.e., they are designed to automatically adapt to the new language definition as it is changed by the user.

APPLAB uses the dynamic structure editor SbyS [Min90], originally developed for the Mjølner Orm environment [MHM+90], [Mag93]. The SbyS editor differs from most other structure-oriented editors by being based on *grammar interpretation* rather than *generation of source code*. I.e., while systems like MENTOR [DGHKL84], Gandalf [Not85], and the Synthesizer Generator [RT89] generate source code for a language-specific editor from a grammar, the SbyS editor interprets grammars to supply language-specific behavior, thus making compilation and linking of the editor unnecessary after a grammar change. This is the key to providing the dynamic behavior needed in a language lab setting.

This paper describes the dynamic parser GRIP (GRammar-Interpreting Parser) implemented as part of APPLAB. The GRIP parser is used both to allows textual editing as an alternative to structure-oriented editing, and to supports document exchange with other environments via textual representation. Analogous to the dynamic structure editor SbyS, GRIP uses grammar interpretation to supply dynamic language-specific parsing. Other systems which contain dynamic parsers include DOSE [FJS86] and GIPE [Kli91]. (See section 4 for a comparison to these systems.)

The requirements on a language-lab parser differ from those in a traditional text-based environment:

- *Parsing speed.* The speed of the parser is not as important in a language lab as in text-based environments: During editing, very small text-fragments are parsed at a time (often only part of a line), and as long as this can be done in a fraction of a second (so it is

1. An application-specific language embeds application-domain knowledge and programming rules into a language to support simpler and safer programming within that application domain.

unnoticeable to an interactive user), the speed is acceptable. For parsing larger documents produced by other environments, the parsing speed will show, but since this is a comparatively uncommon operation in the language lab, a moderately fast parser may be acceptable.

- *Parser generation speed*. The parser generation speed is very important. It should take at most a few seconds to produce a new parser when the grammar is changed, in order to allow editing/parsing according to the new grammar to proceed immediately after the change.
- *Parser grammar*. The context-free part of a language specification is, typically, centered around an abstract grammar, augmented by a pretty-printing specification describing how to unparse an abstract syntax tree as text. Preferably, the parser should use these existing specifications as far as possible. In particular, we want to avoid having to specify a separate parsing grammar in order to satisfy a particular parsing algorithm.

For GRIP we have chosen an object-oriented recursive-descent parsing algorithm which interprets the grammar while parsing. The object-oriented algorithm supports parsing directly into the structure described by the abstract grammar. To resolve parsing ambiguities, we have extended the grammar specification with precedence- and associativity rules. To achieve a reasonable speed during the grammar-interpreting parsing, a graph representation of the grammar is constructed. This graph construction corresponds to the parser generation, and takes only a couple of seconds.

The rest of this paper is organized as follows. Section 2 describes editing in APPLAB from the user's point of view. Section 3 describes the implementation of APPLAB and the interpretive parser in particular. Section 4 discusses related work, and section 5 concludes the paper.

2 Text- and structure editing in APPLAB

APPLAB supports simultaneous editing of a grammar and an example program following the specified grammar. (The grammar formalisms are specified in meta grammars.) The edited structures are represented internally as abstract syntax trees (ASTs) and are presented on the screen as text by unparsing the ASTs. Any subtree can be selected by the user and edited either structurally or as text. In structure editing, constructs are inserted into the AST by selecting them from a menu which contains all syntactically legal alternatives at the point of insertion. In text editing, the textual representation of the selected subtree is edited by the user, and then parsed into a new subtree which replaces the old selection.

Overview

Figure 1 contains a snapshot of an APPLAB session. The grammar currently edited describes ARL, a robot programming language [ABB94]. The windows marked ABSTRACT, CONCRETE, and PARSE contain different *aspects* of the context-free part of the language specification (the context-sensitive parts are not discussed in this paper). The ABSTRACT aspect describes the abstract syntax, i.e. the syntactic structure of the language. The CONCRETE aspect describes the concrete syntax, i.e. the screen layout of different constructs of the language. The PARSE aspect is used to define additional rules needed for parsing. The TARGET window contains an example program following the ARL grammar. Changing, e.g., the concrete syntax in the CONCRETE window causes the TARGET window to be updated to show the new syntax instead of the old one. In the same way, a new language construct, added to the ABSTRACT and CONCRETE windows, is immediately accessible in the TARGET, both for structure and text editing.

Figure 1 shows the different grammar aspects for expressions, and for `mult_exp` in particular. The ABSTRACT aspect gives the BNF AND/OR structure of the grammar. An OR rule corresponds to the declaration of a class hierarchy, where the left-hand side, e.g. `C_exp`[1], is a superclass of the symbols on the right-hand side, e.g. `C_arith`. Similarly, `C_arith` is a superclass of `mult_exp`. An AND rule corresponds to composition where a syntax node object of the left-hand-side class has components according to the right-hand side. For example, a `mult_exp` node has two components (son-nodes) of class `C_exp`.

The CONCRETE aspect gives the concrete text representation for each of the AND rules. For example, when displaying a `mult_exp` node, the token "*" will be shown between the texts representing the two son nodes (referred to as @1 and @2).

The PARSE aspect shows the precedence and associativity rules for different expressions, in order to disambiguate the textual representations for different syntax trees. In the figure, the `mult_exp` has been given higher precedence than `add_exp` (rules on lower lines have precedence over rules on upper lines). This means that the string 'a+b*c' will be parsed as 'a+(b*c)'. Switching the

1. The prefix "C_" used in some rules is used as a primitive structuring devise for the editor menus. It will be removed in future releases of APPLAB.

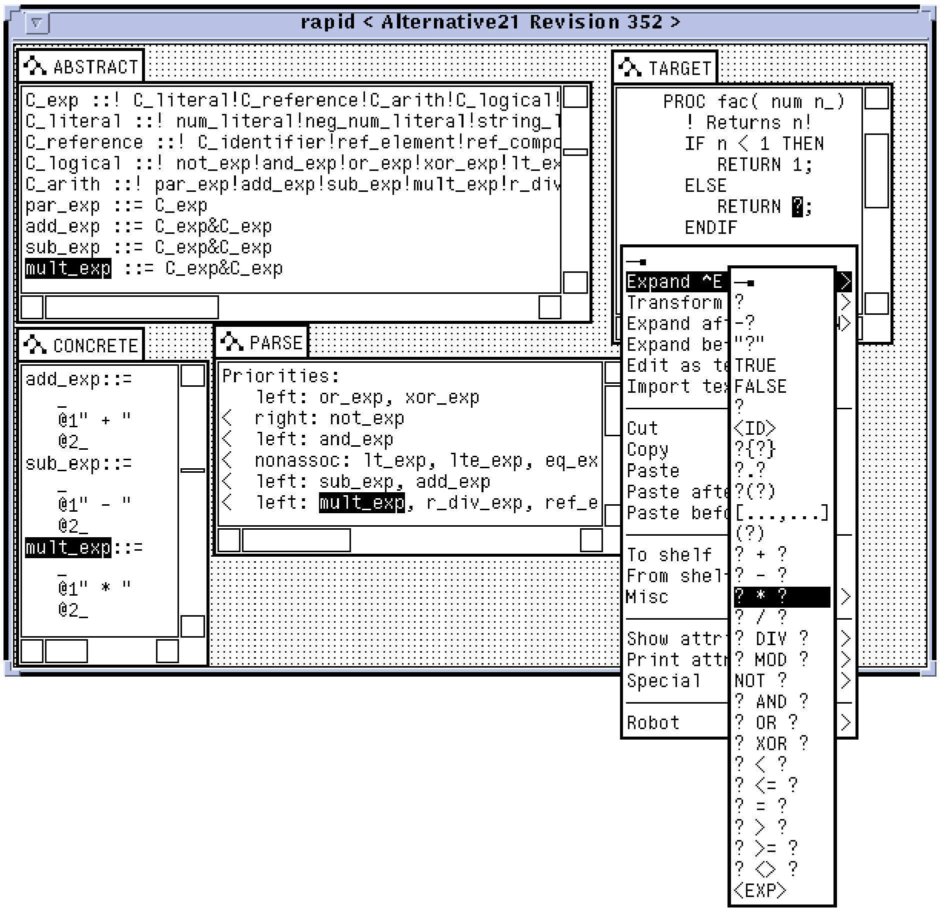

Figure 1 *Structure editing in APPLAB*

precedence levels for mult_exp and add_exp would mean parsing the same string as '(a+b)*c'.

Text editing

When choosing to edit a structure as text the user is given the text representation of the currently selected subtree as the initial text and may edit it character by character. In some situations text editing may be preferred to structure editing, e.g. when editing complex expressions, and when changing an existing construct. For example, changing an existing while-stmt to a for-stmt with the same body. Text editing such constructs is easy, while doing the same editing using the structure editing technique would mean a series of cut-and-paste operations.

If errors are encountered during parsing they are reported to the user in an error window which displays an error message and the position of the error in the edited text. GRIP tries to repair the errors as far as possible. For example, if an error causes the parser to fail in parsing the body of a while-stmt, the parse still results in a while-stmt, but possibly with an empty body structure.

Changing the grammar

When the grammar is changed, subsequent text editing in the TARGET will be parsed according to the new grammar state. The user does not need to give any explicit command for updating the parser. Instead, the parser will be regenerated automatically the next time the user edits a subtree as text.

Some changes to the grammar may cause the existing program in TARGET to become inconsistent with the current grammar. If only the CONCRETE aspect is changed, the TARGET will be automatically reunparsed to reflect the new grammar state. If rules in the ABSTRACT aspect are changed, e.g. adding or deleting components in an AND rule, this will cause existing nodes of these rules in the TARGET to become inconsistent with the grammar. Currently, the inconsistent nodes are simply marked in the TARGET, and the user may explicitly delete them to obtain a consistent program. Future versions of APPLAB may add more sophisticated techniques for updating inconsistent TARGET programs, allowing transformation rules to be stated using, e.g., the techniques suggested in [GKL94].

Grammar errors
When editing the grammar, errors will often be introduced. For example, the misspelling of nonterminals, the use of a nonterminal on the right-hand side of a rule without it appearing on the left-hand side of any rule, and inconsistent aspects of the same grammar rule, e.g. different number of components in the ABSTRACT and CONCRETE aspects of a rule. These errors are reported during the parser generation phase.

More intricate errors can occur as a result of certain combinations of rules, e.g. indirect left-recursion and grammar ambiguities. Indirect left-recursion is detected and reported during parsing, see section 3.5. Grammar ambiguities are not detected, but rules appearing early in the grammar are given precedence over rules appearing later, thus disambiguating the grammar.

Since we can assume that errors in the grammar will be rather common while designing a language it is desirable to have an environment that is as tolerable as possible towards these errors. The environment must not crash as a result of a grammar error, and it should be as supportive and helpful as possible in debugging the language being designed. GRIP has been designed with this in mind and allows parsing even with an incomplete, incorrect grammar. If the erroneous rules are used during the parse, the grammar errors are reported as warnings, and parsing is done according to a repaired version of the grammar, as described in section 3.5.

3 Implementation

3.1 OO recursive-descent parsing

Recalling the requirements on language-lab parsers listed in section 1, it is important that the parser generation speed is high, whereas the actual parsing speed is less crucial, and that the abstract grammar used for structure editing can be used as the basis also for parsing. To meet these demands, we have chosen to base GRIP on OORD, an Object-Oriented Recursive-Descent parsing method [Hed96]. Similar to traditional recursive-descent parsing, OORD follows the grammar closely, and it is therefore simple and fast to generate the parser. The parsing method is termed *object-oriented* because the language specification is given in an object-oriented form, using classes and subclasses, rather than in the traditional form using nonterminals and productions. These classes are called *node classes*, because they classify the nodes occurring in the syntax trees. An important advantage of the OORD method is that it is particularly well suited for parsing directly into abstract syntax trees:

- Precedence and associativity, normally specified using production rules such as term, factor, etc., and common prefix factorization, can be specified using specialization (subclassing), thereby avoiding the generation of non-abstract nodes in the resulting parse tree.
- Left-recursion, as it normally appears for expressions, can be handled directly by the algorithm. In contrast, in traditional recursive-descent parsing one has to eliminate left-recursion from the grammar, thereby distorting the resulting parse tree.

Precedence and associativity
The example below shows how precedence and associativity of arithmetic expressions can be defined in the OORD notation:

```
Exp: ();
Term: Exp ();            # Term is a specialized expr
Factor: Term ();         # Factor is a specialized term
Primary: Factor ();      # Primary is a specialized factor
Add: Term (Term '+' Factor);
          #  Add is a specialized term with two
          #  subcomponents - a term and a factor.
Sub: Term (Term '-' Factor);
Mul: Factor (Factor '*' Primary);
Div: Factor (Factor '/' Primary);
Id: Primary (LexId);
ParExp: Primary ( '(' Exp ')' );
```

As seen from the example, specialization is used to limit what kinds of expressions may occur as subcomponents to different expressions. For example, the specification for Add limits the left component to be a term and the right component to be a factor. Add is thus left-associative, since another instance of Add can only appear as the left subcomponent. The corresponding specification in a conventional nonterminal/production notation would be:

```
add: <Term> ::= <Term> '+' <Factor>
```

where add is the production name, and Term and Factor are nonterminals. An important difference between these notations is that they result in different parse trees: the OORD parse tree is like an abstract syntax tree in that it only contains instances of the most specialized classes: Add, Sub, Mul, Div, Id, and ParExp. In contrast, a parse tree from a conventional grammar contains additional internal nodes for the nonterminals Exp, Term, Factor, and Primary.

Left-recursion
As seen from the conventional specification of add, the production is left-recursive (the first component 'Term' is the same nonterminal as the left-hand side 'Term'). In a standard recursive-descent parser , such left-recursion must be eliminated by rewriting the grammar rules in a way which distorts the parse tree and adds more nodes to it. In the OORD method, such rewriting is unnecessary; the parser can deal with the left-recursion and construct the abstract syntax tree directly .

Common prefix factorization
If two productions with the same left-hand side nonterminal have a common prefix, a standard recursive-descent parser would, in general, need an unbounded lookahead. The usual technique to avoid this, in order to use a lookahead of fix length, is to factor out the common prefix into a new nonterminal, thereby changing the structure of the parse tree and adding more nodes to it. Using the OORD notation, the factorization can be done by specialization rather than by adding new nonterminals, and this way the resulting parse tree is unaffected by the factorization. The example below shows an example grammar where two variants of the if-statement start in the same way , i.e. they have a common prefix.

```
Stmt: ();
IfThenElseStmt: Stmt ('if' Exp 'then' Stmt 'else' Stmt);
IfThenStmt: Stmt ('if' Exp 'then' Stmt);
```

The common prefix can be factored out by adding a new class IfThenStartingStmt which contains the common prefix. The new class is inserted in the class hierarchy between the general class Stmt and the specialized If-classes:

```
Stmt: ();
IfThenStartingStmt: Stmt ('if' Exp 'then' Stmt);
IfThenElseStmt: IfThenStartingStmt ('else' Stmt);
IfThenStmt: IfThenStartingStmt ();
```

Since we have only changed the class hierarchy , and not the structure of instances of the concrete most specialized classes, this factorization does not affect the structure of the parse tree.

In the case of left-recursive classes, like Add and Sub above, the common prefix does not have to be factored out because the OORD method for handling the left-recursion will automatically take care of the common prefix as well.

3.2 Parser generation

When the grammar has been changed, the parser needs to be re-generated. For GRIP , being an interpretive parser , this re-generation means updating only the graph structure interpreted by the parser . The core of the parser is generic and does not need to be updated. The following steps are carried out:

- Check the grammar to report possible errors in it.
- Construct a graph representation of the grammar according to the information in the ABSTRACT and CONCRETE grammar aspects.
- Transform the graph representation of the grammar to deal with precedence and associativity according to the information in the PARSE aspect.
- Transform the graph representation by doing common-prefix factorization.
- Generate a lexer .

The graph construction and transformation steps are done even if the grammar contains errors. It is then possible to parse successfully , at least as long as none of the erroneous parts of the grammar are involved. The transformation steps are described in sections 3.3 and 3.4. Section 3.5 discusses error handling and in section 3.6 the generation of the lexer is described.

The graph is an internal representation of the grammar , structured according to the OORD notation. There is one vertex per node class, and related vertices are connected to each other in order to allow efficient interpretation: each vertex has a set of edges to its immediate subclasses, and to its component classes. Node classes which have components also have an edge to the corresponding rule in the CONCRETE grammar aspect, where the keywords occurring in the rule can be found. The grammar transformations which are described in sections 3.3 and 3.4, are done only on the graph representation of the grammar , and do thus not interfere with the structure-oriented editor which interprets the original grammar .

3.3 Transformation of precedence and associativity

The precedence and associativity specification given in the P ARSE grammar aspect is used to transform the grammar graph to include this information by inserting additional node classes. As an example, consider arithmetic expressions where the specification in the ABSTRACT and CONCRETE grammar aspects correspond to the following (ambiguous) OORD grammar:

```
Exp: ();
AddExp: Exp (Exp '+' Exp);
SubExp: Exp (Exp '-' Exp);
MulExp: Exp (Exp '*' Exp);
DivExp: Exp (Exp '/' Exp);
ParExp: Exp ( '(' Exp ')' )
```

These rules are disambiguated by the following PARSE aspect specification:

```
<  left: AddExp, SubExp        precedence level 1
<  left: MulExp, DivExp        precedence level 2
```

The grammar graph is transformed to include these precedence and associativity rules by inserting node classes, yielding the following OORD grammar:

```
Exp: ();
Exp_1: Exp ();
Exp_2: Exp_1 ();
Exp_3: Exp_2 ();
AddExp: Exp_1 (Exp_1 '+' Exp_2);
SubExp: Exp_1 (Exp_1 '-' Exp_2);
MulExp: Exp_2(Exp_2, '*', Exp_3);
DivExp: Exp_2(Exp_2, '-', Exp_3);
ParExp: Exp_3 ( '(' Exp ')' );
```

We consider the transformation corresponding to a precedence level specification for subclasses to a base class B. Each of the listed subclasses should be a unary or binary operator with subcomponents qualified by B.

Precedence is handled by inserting $n+1$ new node classes P_k, $k=1..n+1$, where n is the number of precedence levels, and where super(P_1) = B and super(P_k) = P_{k-1} (for $k>1$). Node classes at precedence level k are moved in the class hierarchy by making them subclasses to P_k, and the qualifications of their components are also changed to P_k, or to the more specialized class P_{k+1}, depending on associativity , as described below . For instance, MulExp on level 2 is made subclass to Exp_2, and its component qualifications are changed to Exp_2 and Exp_3 respectively .

Associativity of binary operators is handled by specializing either or both of the subcomponents to P_{k+1}. For left-associative operators, the second component is specialized. This restricts an additional instance of an operator on the same level to appear only to the left of the operator . For instance, in MulExp, the second component is specialized to Exp_3. An additional instance of MulExp or DivExp can thus only appear as the first component. Analogously for right-associative operators, the first component is specialized. For non-associative operators, both components are specialized (disallowing adjacent instances of the operator altogether). Unary prefix (or suffix) operators are treated like binary operators with an empty first (or second) subcomponent.

Subclasses to the base class B whose right-hand side neither starts nor ends with B, for example ParExp, should not be listed in the precedence level specification since they cannot take part in the grammar ambiguities addressed by precedence and associativity . These classes are made subclasses to P_{n+1} and the qualifications of their components are left unchanged. For instance, ParExp is made subclass to Exp_3, but its component is still qualified by Exp.

The transformation algorithm for precedence and associativity is given below . In the algorithm we assume that all classes listed in the precedence specification are direct subclasses of the base class. Some of our grammars, however , make use of intermediate classes, such as C_arith, as shown in figure 1. These intermediate classes merely have the role of grouping similar expressions in order to give more structure to the abstract grammar , and they are removed in the parser version of the grammar before doing the transformation.

```
Algorithm PrecAndAssocRewrite(B)
  where B is a base class with subclasses specified in n
  precedence levels.

  Let SUB_k be the subclasses to B listed at level k.
  Let NONLISTED be the subclasses to B not listed on any
  level.

  Add a new class P_1; super(P_1) := B;
  for k := 1 to n do
     Add a new class P_k+1; super(P_k+1) := P_k;
     for each class X in SUB_k do
        super(X) := P_k;
        Let s_1 and s_2 be the subcomponents of X
        case associativity of level k:
           left: qual(s_1) := P_k; qual(s_2) := P_k+1;
           right: qual(s_1) := P_k+1; qual(s_2) := P_k;
           non: qual(s_1) := P_k+1; qual(s_2) := P_k+1;
        esac;
     od;
  od;

  for each class X in NONLISTED do
     super(X) := P_n+1;
  od;
```

3.4 Transformation of common prefixes

Common prefixes of sibling classes (classes with the same superclass) are factored out by introducing an additional class containing the common prefix, as was described in section 3.1. The algorithm is given in detail below . In the case of left-recursive classes, however , the OORD method handles the common prefix automatically , and no factorization is needed.

```
Algorithm CommonPrefixFactorize(C, SUB)
   where C is a class and SUB is a set of subclasses of C.

   if |SUB| > 1 then
      let p be the longest common prefix in SUB.
      I.e., p is the longest common prefix of the sequences
         RHS(X), X ∈ SUB,
      where RHS(X) is the sequence of right-hand side
      symbols (keywords and nonterminals) of the class X.
      if |p| = 0 then
         Split SUB into subsets SUB_k starting with the same
         symbol. I.e.,
            ∀X ∈ SUB_k, head(RHS(X))=sym_k.
               sym_i ≠ sym_j, i ≠ j
         for each set SUB_k do
            CommonPrefixFactorize(C, SUB_k);
         od
      else
         Construct a new class N for the common prefix p
         and insert it between C and SUB in the class
         hierarchy, removing the prefix p from the classes
         in SUB.
      fi
   fi
```

3.5 Error handling

When doing interpretive parsing there are two sources of errors: errors in the parsed text and errors in the grammar. Errors in the grammar are reported at parser generation time, but since APPLAB allows parsing even if there are errors in the grammar, these errors may be reported at parsing as well.

Errors in the parsed text

GRIP performs *Phrase-level Error Recovery* [ASU86], if the parsed text does not match the grammar. I.e., it performs local correction on the remaining input in the following manner. At a token mismatch, the parser first tries to skip the current token and continue parsing. If there still is a token mismatch, the parser inserts the expected token instead and continues parsing, possibly encountering more errors.

If there is more than one possible expected token, i.e., if the parser is parsing a node class with subclasses, and the current token matches neither of the FIRST symbols for the subclasses, the parser inserts a placeholder for the expected node class and continues parsing.

Errors in the grammar

Most grammar errors, e.g. reference to undefined symbols, etc., are reported at parser generation time. If the grammar contains errors, the incomplete or inconsistent rules are repaired in the graph representation and parsing is allowed all the same. If a repaired rule is used during parsing, the parser reports a warning. This is useful in the experimental language lab setting where the grammar is often changed; one can continue to edit the target even if the grammar temporarily contains errors.

As described above, the OORD algorithm handles direct left-recursion which appears, e.g., for normal left-associative binary expressions. However, the algorithm does not handle indirect left-recursion. According to [DM82, p. 62], indirect left-recursion hardly ever appears in practical grammars. Nevertheless, endless recursion must be avoided, should the user happen to specify a grammar with indirect left-recursion, by mistake or on purpose. To handle this, a check is done while parsing. When this check detects an indirect left-recursion in the current parse, an error is reported, and the current parse is cancelled.

It would be possible to check indirect left-recursion at parser generation time as well, but by checking it at parsing time it is possible to parse strings which do not involve the indirect left-recursion.

3.6 The lexer

The lexer uses a table of keywords constructed from the CONCRETE grammar aspect. The table is constructed during parser generation. Currently, the definitions of other lexical items are hand-coded to fit the ARL language, including identifiers, numbers, strings, and comments (for the latter two, start- and end-tokens can be specified in the PARSE aspect). These definitions are fairly standard and are sufficient for many experimental languages. However, future releases of APPLAB will include a fully grammar-interpretive lexer which allows complete specification of the lexical level. Analogously to the parser, a graph representation of the lexical specification will be constructed and interpreted during lexical analysis. This will allow full use of the interpreting parser for all languages defined in APPLAB.

4 Related work

Editing programs in a structure-oriented editor is fundamentally different from using a traditional text editor since the editing is done in terms of language constructs rather than characters. Many voices have been raised to say that text editing should not be banned altogether from structure-oriented editors, e.g. [VG92, Wat82]. On the other hand, it has been argued that the structure-oriented approach has not yet been fully explored and evaluated, and that the usual arguments given in favor of text editing are not valid in structure-based environments [Min92]. Whatever viewpoint is taken towards text editing in structure-oriented editors, there is a need for parsing in such systems if one wishes to integrate it and exchange documents with other (text-based) systems.

Syntax-directed versus syntax-recognizing editors

Among the existing language-based environments there are two fundamentally different approaches to combining text and structure-oriented editing: *syntax-directed* editors and *syntax-recognizing* editors. In the syntax-directed editors, the syntax tree of the document is presented to the user, and editing operations are performed directly on the tree. Adding text editing to such editors is done as in APPLAB, by letting the user select a subtree and edit it as text. The edited text cannot affect any parts of the syntax tree outside the structural selection. Other environments supporting this kind of editing include the Synthesizer Generator [RT89], Centaur [BCD+88], Gandalf [Not85], GIPE (ASF/SDF) [Kli91] and DOSE [FJKS85].

The selection can be parsed either *exhaustively*, i.e. parsing all the text of the selection, or *incrementally*, i.e. parsing only the changed parts of the text and reusing the old syntax subtree in order to construct the new subtree for the selection. Incremental parsing can speed up the parsing in case the selections are large. It depends on how the environment is used if large selections are edited as text or not. It is common to edit only small structures, e.g. expressions, as text, whereas larger structures, e.g. procedures or control structures, are edited using structure editing. In this case an exhaustive parser is usually sufficiently fast. APPLAB, the Synthesizer Generator, and Centaur use exhaustive parsing. The Gandalf system uses a technique for incremental parsing where a series of tree transformations are performed as tokens are inserted and deleted [KK85]. DOSE uses a combination of techniques, where assignments and expressions are parsed using a method based on the same incremental parsing algorithm used in the Gandalf system.

A different approach in combining text and structure is taken in *syntax-recognizing editors*, e.g. Pan [BGV90]. In these editors, the user edits the document as text and the system maintains an internal syntax tree which is hidden from the user. This type of system allows the user to continue working in the same way as in a traditional text editor, while the internal syntax tree can be taken advantage of in the same way as in the syntax-directed editors, e.g. for cross-referencing or type-checking. The text-editing in the syntax-recognizing editors thus corresponds to text editing in a syntax-directed editor where the selection is always the complete document. The "selection" is thus usually quite large, and incremental parsing is therefore very useful in these systems. Since it is not feasible to perform incremental lexical and syntactic analysis for each character-edit operation, a design decision has to be made for such editors as to when the incremental lexical analysis and parsing is to be invoked. In Pan, the different levels of analysis maintain a summary of increments to be used by the next analysis level. I.e. the incremental LALR(1) parser revises the internal structure in response to increments produced by the lexical analyzer.

Language lab parsers

When implementing a programming environment or structure editor for a specific language, the easiest way to produce a parser component is to generate one by using a general-purpose *parser generator* such as `yacc`[Joh79]. For example, both Centaur and the Synthesizer Generator use `yacc` to generate the parser component. However, `yacc` and other parser generators produce program code as output (C in the case of `yacc`) which has to be compiled and linked. For a language lab, this approach is not suitable since it would take too long time to generate a new parser, definitely more than a few seconds as was required according to our list in section 1. Furthermore, with current standard compilation and linking technology, such a parser could not be an exchangeable part of the language lab process, but would need to run as a separate process, adding interprocess communication overhead during parsing.

For a language lab, the reasonable approach seems to be to use some kind of generic parser interpreting a data structure derived from the grammar. The language-lab environments DOSE, GIPE, and our APPLAB, all use some variation of this approach. There are a few different techniques, or combinations of techniques, that are used to generate the parser sufficiently fast:

- Incremental parser generation. I.e., update the parser data structure according to the grammar changes.
- Choose a top-down parsing algorithm, since they require less pre-processing than bottom-up algorithms.
- Lazy parser generation. I.e., generate the parser data structure piece by piece as it is needed during parsing.

E.g., APPLAB uses a top-down parser, while GIPE uses a bottom-up parser and both incremental and lazy parser generation. DOSE uses an algorithm that combines top-down and bottom-up parsing. The different language-lab parsers are discussed in more detail below.

DOSE Parsing is performed in DOSE by a combination of parsing techniques. Expressions and assignment statements are parsed using a method based on the incremental expression parsing algorithm developed by Kaiser and Kant [KK85], while other language constructs are parsed top-down by recognizing the first keyword of a construct, like IF and WHILE. Just as in APPLAB the same language description that is used for the structure-editor is also used by the parser. While APPLAB presents the subtree to be text edited in a separate window, DOSE allows the user to edit text within its context on the screen.

GIPE The parser generator of the GIPE environment, IPG [HKR90, Rek92], is implemented as a lazy, incremental LR(0) parse table generator. The parse tables are incrementally updated by need while parsing input. The syntax specification of GIPE is similar to the one in APPLAB in that the abstract syntax is augmented with disambiguating precedence and associativity rules. The text editing facilities of GIPE are more smoothly integrated into the structure-oriented editor. While APPLAB lets the user text edit a selected part of a program in a separate window, GIPE allows text editing of the program fragment currently in focus, to be done within its context in the structure-oriented editor. Parsing of the edited text is performed when the user moves the focus to another part of the program. If any syntax errors are encountered the user is required to correct them before moving the focus. In APPLAB, should any syntax errors be detected, they are reported and, as far as possible, corrected by the parser and the resulting AST is inserted into the edited program. If the user so wishes the latest text editing operation can be undone by re-inserting the old (unedited) AST. GIPE and APPLAB both allow the user to select any part of the edited program for text editing.

TaLE The TaLE environment [JKP91, JK93] is a specialized, partly graphical, editor for designing textual languages. It allows the user to edit, reuse and refine language features in a controlled fashion. TaLE is suited for rapid prototyping of application-specific languages but is not a language lab in our sense since it does not allow the user to freely switch between designing and testing the language. Instead, TaLE generates Eiffel-classes (which may be manually edited) corresponding to the language constructs which are then compiled and result in a traditional source-code compiler rather than a structure-based editor.

TaLE uses incremental parser generation where the generated Eiffel classes are incrementally updated as the user changes the language description. The parsing algorithm used in TaLE is a lazy recursive descent algorithm modified to fit object-oriented context-free grammars [KV92]. The parsers in TaLE and APPLAB have certain similarities. They are both recursive-descent parsers, and they both draw on the principles of object orientation in distributing the information needed for parsing into objects representing the nonterminals of the grammar. In TaLE, this is utilized to support incremental (and lazy) parser generation. For example, changing the production for a nonterminal only affects the corresponding object in the parser.

5 Conclusion

We have presented the dynamic parser GRIP and its integration in the language design laboratory APPLAB. GRIP is a generic parser which interprets a grammar graph in order to parse according to a given grammar. Any text selected in APPLAB's structure-oriented editor can be edited as text and parsed by GRIP. In case the grammar has been changed since the last text edit operation, a new grammar graph is generated automatically, causing GRIP to always use the current version of the grammar. The system has been tested on several languages, ranging from small meta-grammars of around 10 productions to grammars for real-sized programming languages with over 200 productions (the grammar for ARL, which is a Pascal-like language with some special constructs for programming robots).

APPLAB and GRIP are implemented in the object-oriented language Simula [Sta87] and runs on SUN Sparc workstations. An evaluation of GRIP compared to the requirements listed in section 1 yields the following conclusions:

- *Parsing speed*. GRIP parses around 40 tokens/second on a Sparc Station 5 (1 10 MHz). This corresponds to around 350 lines per minute. This is sufficient for the main requirement, namely that parsing of small texts, such as short expressions, should be unnoticeable by the user. However, faster parsing would certainly be desirable for document exchange. There are many details in the implementation which could be optimized and we therefore have good hope of increasing the parsing speed substantially.
- *Parser generation speed*. A new parser is generated in 1.5 seconds for a real-sized language with over 200 productions. We find this well within the acceptable delay.
- *Parser grammar*. As required, GRIP runs without the need of a special parsing grammar. The abstract and concrete grammars augmented with disambiguating precedence and associativity rules is sufficient information to generate the parser.

The object-oriented recursive-descent (OORD) algorithm turned out to be very suitable for implementation of the language lab parser. Being a recursive-descent algorithm it is straight-forward to implement. Furthermore, the possibility to express precedence, associativity, and common prefix factorization using specialization, allowed us to transform the original abstract grammar to a grammar fit for the OORD algorithm, yet without changing the structure of the resulting parse tree. I.e., the tree resulting from parsing is also a derivation tree of the abstract grammar.

Acknowledgment

We would like to thank the anonymous referees for valuable comments on a draft of this paper. This work was supported by NUTEK, the Swedish National Board for Industrial and Technical Development.

References

[ABB94] ABB Robotics. *ARL Reference Manual*, 1st edition, February 1994.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BCD+88] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The System. In *Proc. of the ACM SIGPLAN Conference on Practical Software Development Environments*, pages 14–24, 1988.

[BGV90] R. A. Ballance, S. L. Graham, and M. L. Van De Vanter. The Pan Language-Based Editing System For Integrated Development Environments. *SIGSOFT, Software Engineering Notes*, 15(6), December 1990.

[Bja95] E. Bjarnason. APPLAB: User's Guide. TR LU-CS-IR:95-2, Department of Computer Science, Lund University, Sweden, September 1995.

[DGHKL84] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. *Programming Environments Based On Structured Editors: The MENTOR Experience*, chapter 7, pages 128–140. In Barstow et al., editors. *Interactive Programming Environments*. McGraw-Hill Book Company, 1983.

[DM82] A. J. T. Davie and R. Morrison. *Recursive Descent Compiling*. Ellis Horwood Limited Publishing, 1982.

[FJKS85] P. H. Feiler, F.Jalili, G. E. Kaiser, and J. H Schlichter. ISDS: A Retrospective of the DOSE System. Technical report, Siemens Research and Technology Laboratories, 105 College Road East, Princeton, NJ 08540, May 1985.

[FJS86] P. H Feiler, F. Jalili, and J. H Schlichter. An Interactive Prototyping Environment for Language Design. In *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, volume II, pages 106–116. IEEE, 1986.

[GKL94] D. Garlan, C. W. Krueger, and B. Staudt Lerner. TransformGen: Automating the Maintenance of Structure-Oriented Environments. *ACM TOPLAS*, 16(3):727–774, May 1994.

[Hed96] G. Hedin. A Recursive-Descent Parser Based On Object-Oriented Abstract Syntax Tree Grammars. Draft, 1996.

[HKR90] J. Heering, P. Klint, and J. Rekers. Incremental Generation of Parsers. *IEEE Transactions on Software Engineering*, SE-16(12), December 1990. Also in Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, (June 1989), pp. 179-191.

[JK93] E. Järnvall and K. Koskimies. Computer-Aided Language Implementation with TaLE. Technical Report A-1993-4, Department of Computer Science, University of Tampere, P.O. Box 607, SF-33101 Tampere, Finland, July 1993.

[JKP91] E. Järnvall, K. Koskimies, and J. Paakki. The Design Of The Tampere Language Editor (TaLE). Technical Report A-1991-10, University Of Tampere, Department of Computer Science, December 1991.

[Joh79] S.C. Johnson. Yacc: Yet Another Compiler Compiler. Unix Programmer's Manual, 7th Edition, Bell Telephone Laboratories, January 1979.

[KK85] G. E. Kaiser and E. Kant. Incremental Parsing Without a Parser. *The Journal of Systems and Software*, 5(2):121–144, May 1985.

[Kli91] P. Klint. *A Meta-Environment For Generating Programming Environments*, pages 105–124. In Bergstra and Feijs, editors. *Algebraic Methods II: Theory, Tools and Applications*. Springer-Verlag New York Inc, 1991.

[KV92] K. Koskimies and J. Vihavainen. Incremental Parser Construction With Metaobjects. Technical Report A-1992-5, Department of Computer Science, University of Tampere, Finland, P.O.Box 607, SF-33101 Tampere. Finland, November 1992.

[Mag93] B. Magnusson. *The Mjölner Orm System*, chapter 1, pages 11–23. In Knudsen et al., editors, *Object-Oriented Environments The Mjölner Approach*. Prentice Hall, 1993.

[MHM+] B. Magnusson, G. Hedin, S. Minör, et al. An overview of the Mjölner/Orm environment. In J. Bezivin et al., editors, *Proceedings of the 2nd International Conference TOOLS (Technology of Object-Oriented Languages and Systems)*, pp 635-646, Paris, June 1990. Angkor.

[Min90] S. Minör. *On Structure-Oriented Editing*, PhD thesis, Lund University, Lund, Sweden, 1990.

[Min92] S. Minör. Interacting with Structure-Oriented Editors. *International Journal of Man-Machine Studies*, 37(4), October 1992.

[Not85] D. Notkin. The GANDALF Project. *The Journal of Systems and Software*, 5(2):91–105, May 1985.

[Rek92] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[RT89] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator A System for Constructing Language-Based Editors*. Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[Sta87] SIS Standariseringsgrupp. *Databehandling Programspråk-SIMULA, Svensk standard SS 63 61 14*. SIS, 1987.

[VG92] M. L. Van De Vanter and S. L. Graham. Coherent User Interfaces For Language-Based Editing Systems. *International Journal Of Man-Machine Studies*, 37(4), October 1992.

[Wat82] R. C. Waters. Program Editors Should Not Abandon Text Oriented Commands. *SIGPLAN Notices*, 17(7), July 1982.