Automating the Development of Syntax Tree Generators for an Evolving Language

Per Grape
National Defence Research Establishment
S-172 90 Sundbyberg, Sweden
pelleg@atlas.sto.foa.se

Kim Waldén
Enea Data AB
Box 232, S-183 23 Täby, Sweden
kim@enea.se

Abstract

This paper describes an Eiffel system for rapid testing of grammars. Grammars are defined in an extended BNF notation that allows actions on the parse tree nodes to be defined as additional annotations. The actions are high level descriptions (not procedural code) to transform a parse tree into a syntax tree. A parser, producing a syntax tree for a language sentence, can be automatically generated from the annotated grammar as a set of classes.

The object-oriented environment permits a much higher degree of separation between syntax and semantics than is possible with traditional approaches. Structural grammar changes can be made without affecting already developed semantic routines. This gives a great advantage for early compiler implementations, when the language syntax is still evolving.

1 Introduction

When a new language of some complexity is created, this will often force the development of a number of successive compiler versions to support its evolving syntax and semantics. Although automatic parser generators like *yacc* [Joh75] can ease the developer's burden a great deal, they have a number of limitations that make compiler maintenance hard for an evolving language.

Since the syntactic and semantic elements are mixed in a *yacc* specification, a large amount of recoding will often be needed as a result of mere structural changes to the language grammar. The lack of separation between syntax and semantics also makes it hard to write several processors for the same language, such as compiler, static checker, pretty printer etc., without considerable duplication of effort and risk of inconsistencies. The abstraction support of a good object-oriented language is what is needed to overcome some of the limitations above.

This paper describes a parser generator written in Eiffel [Mey92] and a high-level notation for specifying actions to transform a parse tree into a syntax tree. The notation is based on basic graph transformations on trees, such as adding and removing vertices, contracting edges and rearranging vertices. Given a grammar with annotated transformations, the generated parser will recognise a syntactically correct sentence of the language and deliver an abstract syntax tree, which can then be further processed by semantic routines.

If the set of keywords and operators are fairly stable during development of the language, then the generated syntax trees are expected to be at least as stable, regardless of structural grammar changes. This makes it possible to work efficiently on the evaluation scheme before the language has been frozen.

Since the syntax transformation directives are high-level, the task of keeping them consistent with the evolving language constructs becomes easy. This may be contrasted with *yacc*, where the actions to build a syntax tree have to be recoded in C each time the grammar is restructured.

The parser generator was designed to support the development of an analytic query language for geographical databases, called GeoSAL [SZ91]. The work was carried out as part of a joint project between the National Defense Research Establishment, NobelTech Systems AB, and Ericsson Radio Systems AB. The project is part of a national research and development program in information technology.

2 The Eiffel environment

Among the attractive features of the Eiffel distribution from Interactive Software Engineering Inc. (ISE) are substantial class libraries supporting basic data structures, lexical analysis, and parsing [MN90]. Thus for the most part, there is no need for the user to implement common data abstractions, such as lists, hash tables, trees, stacks and queues, since these are directly available and easily tailorable through subclassing.

The lexical library supports grammars of regular expressions, and provides approximately the facilities of *lex* [LS75]. Instead of the preprocessor approach of *lex*, the lexical classes contain operations to generate a lexical scanner from descriptions in a file, and store it in internal object format in another file for subsequent use.

The parsing library classes map the constructs of an arbitrary LL(1) grammar, and provide operations for recursive descent parsing of the corresponding language [HM89]. This is different from *yacc*, which provides bottom up parsing of LALR(1) languages (for an overview of compiling techniques, see for example [FL91]). The family of languages that can be expressed with LL(1) grammars is somewhat smaller than the corresponding family for LALR(1) grammars. On the other hand, most well-designed programming languages can be turned into LL(1) form. Rare exceptions, such as the C/Pascal dangling "else", can usually be taken care of by allowing the grammar to be ambiguous and then let the parser apply disambiguating rules. (This technique is also employed by *yacc* on LALR(1) grammars.)

Moreover, LL(1) parsing has the advantage of much easier error reporting and recovery for the compiler, compared to the LALR(1) technique. So it suited our needs well, since we wanted to reduce the effort of syntax control and spend as much time as possible on the semantics of the language under development.

3 Object-oriented parsing

The idea underlying the Eiffel parsing library, which is a direct application of the phrase "syntax-directed compiling", is to model each production of a grammar by a separate class. This object-oriented approach to parsing has several advantages. Encapsulating each syntactic construct as an independent unit makes it easy to build an abstract syntax tree, which can then be traversed and decorated in successive passes. Different semantic actions can be applied to the same syntax tree, thus permitting several tools to share the same syntactic representation. Classes with algorithms for semantic analysis and evaluation can be developed independently, without having to rewrite the code each time superficial changes are made to the language grammar.

The Eiffel parsing library restricts the productions to three kinds of construct: *aggregate*, *choice* and *sequence* (their exact meaning will be described in the next section). Each construct has a library class to inherit from, which implements the parsing details applicable to constructs of that type. All the user needs to do, is fill in the concrete parts in the class text for each grammar construct. Although this is simple enough to do for a few classes, writing classes by hand for the grammar of a realistic language is not feasible, particularly for an evolving grammar where they would constantly have to be rewritten.

This renders the parsing library next to useless, unless complemented by a program that can generate the classes needed directly from a grammar description. Currently, there is no such generator in the Eiffel distribution, but ISE has made one for internal use named *yoocc* (Yes!, an Object-Oriented Compiler Compiler) in homage of *yacc*, which is planned to be a product, but not yet released [HM89]. By courtesy of ISE, we were allowed to reuse a set of classes for Eiffel source code generation previously developed for *yoocc* when building our own parser generator.

4 Syntax notation

We use a straightforward extension of BNF to describe language syntax, which is very close to the notation used for Eiffel in [Mey92], augmented by a special construct that permits tree transformations to be specified for each production. We will simply refer to it as EBNF (*extended* BNF). It is based on the following concepts.

Any syntactically identifiable part of a language text, such as a query in a query language, an expression in an arithmetic language, an if statement in a programming language etc., is called a *component.*

The structure of components of a certain category is described by a *construct.* The corresponding components are called *specimens* of the construct. Every construct has a unique *construct name.*

Every construct is either *terminal* or *non-terminal.*

A terminal has only one specimen and no further syntactical structure. In contrast a non-terminal may have many different specimens, and is defined in terms of other constructs.

Terminals can be defined in two ways, as a string constant or by typing the construct name in capitals. The latter form refers to a token defined by a regular expression. This is described in section 4.4.

Non-terminals are defined by *productions* giving the structure of the specimens of each construct. A production has the form:

```
Construct_name :  Right_hand_side ;
```

In EBNF, every non-terminal construct appears (through its construct name) as the left-hand side of *exactly one* such production. On the other hand, terminal constructs (by definition) only occur at the right-hand side of productions.

The right-hand side of a production consists of an optional *build description* and then either an *aggregate* (concatenation), a *choice*, or a *sequence* (repetition).

The build description specifies what actions are needed to turn the parse tree into a syntax tree. These actions will also depend on the context, that is, whether the construct is an aggregate, a choice or a sequence. They will be described in section 6.

EBNF also contains a "commit" mechanism similar to Prolog's "cut", to guide the parser for better error reporting.

4.1 Aggregate

An *aggregate* construct is a non-empty sequence of constructs (parts), to be concatenated in the given order. Optional parts are written in square brackets. For example

```
a :  b [c] D ;
```

expresses that an `a` consists of a `b`, followed by zero or one `c`, followed by the terminal `D`.

4.2 Choice

A *choice* construct is a list of alternative constructs separated by vertical bars, as in

```
e :  f | g | h ;
```

This means that an `e` is either an `f`, a `g`, or an `h`.

4.3 Sequence

A *sequence* construct describes variable length lists of specimens of a given construct, separated (if more than one) by a given separator. If `base` is the construct and `separator` is the separator, then the sequence construct has one of the following forms:

```
s : { base separator ... } ;
s : { base separator ... }+ ;
```

The first form allows `s` to be empty, while the second form requires at least one specimen of `base`. The separator is optional, and may be a string constant or a construct.

The following three variants are thus valid sequence specifications:

```
var_list : { identifier ... }+ ;
var_list : { identifier "," ... }+ ;
expr     : { integer arith_op ... }+ ;
```

the first one specifying no separator, the second a fixed string separator, and the third a general construct separator.

4.4 Definition of terminals

The words and symbols recognized by a lexical scanner are commonly called *tokens*. Each token has a *token type* and a *token value*. A terminal construct corresponds exactly to the set of tokens of a given type.

The token value for a given terminal may be fixed, such as a keyword or an operator symbol, or variable, such as a number or an identifier. Token definitions have the following form:

```
Name_1 Regular_expression_1
Name_2 Regular_expression_2
...
```

The Eiffel notation [HM89] is used to specify regular expressions, supporting the usual construction mechanisms for arbitrary character, intervals, grouping, alternation, repetition, concatenation, optional components, and set difference—roughly equivalent to those of *lex* [LS75].

4.5 Example

As an example we give an EBNF description of the EBNF language itself.

```
grammar ebnf ;

lang : "grammar" NON_TERMINAL_ID ";"
    [ebnf_grammar] "end" ";" ;

ebnf_grammar : { statement ...  }+ ;

statement : NON_TERMINAL_ID ":"
    [build_description] production ";" ;

build_description : "%" !  "[" A_STRING
    [build_tail] "]" ;

build_tail : "," !  NON_TERMINAL_ID ;

production : ebnf_choice | ebnf_aggregate
    | ebnf_sequence ;

ebnf_choice : id_or_string "|" choice_tail
    ;

id_or_string : identifier | A_STRING ;

identifier : TERMINAL_ID | NON_TERMINAL_ID
    ;

choice_tail : { id_or_string "|" ...  }+ ;

ebnf_aggregate : { opt_req_com ...  }+ ;

opt_req_com : commit_tag | opt_construct |
    req_construct ;

commit_tag : "!" ;

opt_construct : "[" req_construct "]" ;

req_construct : id_or_string ;

ebnf_sequence : non_empty_seq |
    poss_empty_seq ;

non_empty_seq : "{" id_or_string
    [id_or_string] "..." "}+" ;
```

```
poss_empty_seq : "{" id_or_string
    [id_or_string] "..." "}" ;

end ;
```

The terminals of EBNF, typed in upper case letters above, are defined as follows:

```
  A_string ("\"" -> "\"")
  Non_terminal_id ~('a'..'z')
      *(~('a'..'z') | '_' | ('0'..'9'))
  Terminal_id ~('A'..'Z')
      *(~('A'..'Z') | '_' | ('0'..'9'))
```

The `->` symbol accepts any string up to and including the succeeding string (in this case a single "). The `|` separates alternatives and the `*` is the Kleene closure operator in prefix form (matching zero or more occurrences of the operand). The prefix operator `~` is used to force case sensitivity for an expression.

5 Parse trees and syntax trees

In recursive descent parsing, the parse tree is a trace of the parsing process. Thus all intermediate levels in the grammar are present in the parse tree. These levels cannot in general be removed from the grammar because this may introduce ambiguity, that is, it may become possible to parse an expression in more than one way.

Moreover, it is often desirable to write highly structured grammars that can be easily read and understood by humans, thus also serving as precise documentation of the language syntax. It is therefore unavoidable that the parse tree contains nodes which carry no semantic information.

A syntax tree on the other hand, contains only nodes that are essential for evaluation. In other words, only terminal symbols occur as nodes in a syntax tree (and in some special cases, newly created nodes).

This is best understood by looking at an example.

5.1 Example

Consider the grammar for simple arithmetic expressions.

```
grammar g ;
expression : { term "+" ... }+ ;
term       : { factor "*" ... }+ ;
factor     : UNSIGNED_INT | par_expr ;
par_expr   : "(" expression ")" ;
end ;
```

Here `UNSIGNED_INT` is lexically defined as:

```
Unsigned_int ('1'..'9') *('0'..'9')
```

The parse tree and syntax tree for the expression $1 + 2 * 3$ can be illustrated as in figure 1. Note that a straightforward tree evaluation of the expression is not possible in the parse tree.

The desired syntax tree is smaller than its parse tree. This difference in size obviously depends on the grammar and the expression parsed, but ratios like $1 : 10$ are not unusual.

6 Transforming a parse tree

Building a syntax tree from a parse tree is done in a depth first, post action manner. Thus, at a certain level, all children of a parse tree node have already been transformed into syntax subtrees. This enables us to focus on the current level and forget about lower levels in each step of the transformation.

The optional build description that can be attached to each production specifies the resulting syntax subtree in terms of the corresponding parse tree node and its (already transformed) children. The specification elements applicable to each type of construct will be described in the following sections. But first some general concepts.

In what follows, we take the tree viewpoint (in the graph meaning) when talking about productions. This means we will refer to the *children* of a construct. For example, the construct

```
a :  b [c] D ;
```

has three children. The second child is optional and might not be present in the statement parsed, but is always present in the parse tree in order to keep the natural enumeration of the children independent of optional constructs.

To distinguish parsed children from omitted optional ones, we say that a construct is *complete* if it is completely recognized, that is, all non optional children are complete. A terminal is complete if it matches a token. Thus if `c` in the production above is not present (not complete), and `b` and `D` are complete then `a` is complete.

As defined in section 4.5, a build description consists of a string called *description string* or just *description* and an optional identifier which we will call the *new-name*. A new-name may only be given when the description specifies that a new node is to be inserted at this point, and will then become its label.

We will now explain the specification elements available for each type of construct.

6.1 Transforming an aggregate

Since aggregates can have any fixed number of subconstructs, a flexible notation for specifying rearrangement of syntax tree nodes is needed. For this, we use a pattern which attempts to draw an image of the resulting syntax subtree.

A `*` indicates that a new top node is to be created, and a `-` signifies the start of a subtree. The children of the current node are referred to by number. Parentheses may be used for grouping, and lists of blank separated node references will become siblings. A single node number enclosed in `<>` refers to the list of the corresponding node's children.

Thus, `2-(1 4)` makes the second child the new top node, with the first and fourth child as its children, `*-<3>` creates a new top node (labeled new-name, if given, otherwise given a temporary name) and adopts all the children of the third child, and `3-1-4-2` creates a vertical branch from a permutation of the first four children of the current parse tree node.

An example of a deeply nested transformation description is given in figure 2.

The following semantic rules constrain what patterns are valid:

1. The numbers in a pattern must not be greater than the number of children of the current aggregate, and each number can be used at most once in a pattern.
2. When `f` is to be a father of one or several children then these are always inserted *before* the children that `f` already has (if any).
3. The only time an optional child `c` can be father of another child in a pattern is when `c` occurs first in the pattern and `c` gets just one child. If `c` is not complete then the child of `c` in the pattern acts as father.
4. If an optional child is not present and it is a leaf in a pattern, then it does not generate any node in the syntax tree.

The pattern `*-ALL` can be used as abbreviation to collect all complete children under a new node.

6.2 Transforming a sequence

Sequences need much less flexibility, since they can only contain two construct types, the base element and the separator. A fixed number of alternative specifications is enough:

`*-ALL` Keep the structure of the parse tree. This is done by creating a new node in the syntax tree and make it father of all complete children of the current sequence, including separators. If a new-name was specified, it becomes the label of the new node, otherwise a temporary name is assigned.

`LTREE` Build a binary tree from the sequence of operands and operators (where the separators are interpreted as operators), assuming left-to-right association. The top operator of the binary tree becomes the top node of the syntax tree. If there is only one element in the sequence of children (hence no operators), the only child instead becomes the top node of the syntax tree. (Only applies to sequences with separators.)

`*-LTREE` Like `LTREE`, but a new node is created and becomes the top node, thus adding an extra level. This is sometimes desirable for semantic processing.

`RTREE` Analogous to `LTREE` assuming right-to-left association.

`*-RTREE` Analogous to `*-LTREE` assuming right-to-left association.

`BSEQ` The separator is interpreted as an operator, and becomes the top node having all base elements as children. If there is just one child (namely a base element) then it becomes the top node.

Only applicable for sequences with fixed separator.

`*-BSEQ` Like `BSEQ`, but a new node is created and becomes the top node.

6.3 Transforming a choice

There is only one possible action on complete choices: the retained child becomes the new top node.

6.4 Transforming a terminal

Since terminals have no children, no transformation description is needed. Each parsed terminal becomes a single-node syntax tree (a leaf). Later, this leaf may be moved to become an inner node of the tree. A typical example of this are the operators of figure 1.

6.5 Examples

To visualize the successive transformation of a parse tree into a syntax tree, consider the following conditional expression:

```
if c then x * y / z else f (x, y, z)
```

A simplified parse tree for the expression is shown in figure 3.

Traversing the parse tree left-to-right, depth first, shows the successive transformations taking place at individual nodes according to a plausible specification.

First, we encounter the construct `cond` where no specification has been given. The default behavior, when there is only one child, is to simply lift the child up to current level as in figure 4. For sizable grammars, particularly highly structured ones, the majority of tranformations in a parse tree will in fact often be this basic "pruning" of the tree. This is not surprising, since the result is often a size reduction by an order of magnitude.

Next, we find the construct `term` which transforms its sequence into a binary operator tree through the pattern `LTREE` (figure 5). The result replaces, in the next step, the `expr` construct of the then-part through default behavior.

We then proceed to construct `alist`, where figure 6 shows how a `BSEQ` specification collects the comma-separated argument list.

The parentheses are removed by `2` (figure 7), and the arguments are collected directly under the function identifier through pattern `1-<2>`, lifting the sublist one level (figure 8). This identifier then replaces the `expr` construct of the else-part.

Finally, the keywords are removed from the conditional expression using the selection `1-(2 4 6)`, and we are left with a pure syntax tree, ready for semantic checking and evaluation, as shown in figure 9.

We conclude by showing the grammar G of section 5.1 with build descriptions added to do the transformation depicted in figure 1:

```
grammar g ;
expression : %["LTREE"]
             { term add_op ... }+ ;
term       : %["LTREE"]
             { factor mul_op ... }+ ;
factor     : UNSIGNED_INT | par_expr ;
par_expr   : %["2"]
             "(" expression ")" ;
add_op     : "+" ;
mul_op     : "*" ;
end ;
```

7 The parser generator

The parser generator PG is an Eiffel program which will read a language definition L in EBNF, and generate an Eiffel program that correctly parses L.

PG and the generated parsers make heavy use of the ISE Eiffel class libraries mentioned in sections 2–3 for high level data structures, recursive descent parsing and lexical analysis.

We have made some modifications to the parsing and lexical libraries. For example, a general construct is now allowed as separator in a sequence (only string constants were supported), the communication between parser and scanner has been made dynamic, so lexical definitions can be changed without recompilation (before, lexical token type numbers had to be compiled into the parser).

We have also corrected some minor bugs, and made some extensions. The extensions include features for transforming the parse tree into a syntax tree, as well as printing and tracing facilities.

The generated parsers are sets of classes that inherit from the parsing library. The only class that must be programmed by hand is a small root class, which plays the role of main program in object-oriented contexts. The rest is automatically generated by PG.

7.1 How to use PG

PG makes it very easy to change the language while the compiler is being implemented. Initially, the user performs the following steps:

1. Create a *language-definition* file containing an EBNF definition of the language to be parsed.
2. Create a *token-definition* file containing regular expressions defining lexical tokens.
3. Program a root class for the parser to be generated. This is a minimal programming task.
4. Run PG on the language-definition.
5. Compile and run the parser.

If the user is not satisfied with the language, he makes the desired changes in the language-definition and executes steps 4–5 again.

If a different set of token values is needed, then the old token-definition is replaced by a correct one. Unless there was a token name change in the lexical description (thus forcing a grammar change), no recompilation is needed, the parser is just reexecuted.

7.2 About the generated code

The classes generated by PG are placed in two clusters corresponding to the UNIX directories Language and Semantics.

The Language cluster contains the actual parser classes, which are descendants to classes in the parsing library. The Semantics cluster contains skeletons for evaluation classes corresponding to all terminal operators in the grammar, as well as to all new-names given in the build descriptions.

A naming convention is used to distinguish terminals that will not appear in the syntax tree (and thus should not have evaluation classes) from those who play the role of operators.

Finally, a few words about run-time efficiency. The parsers generated with the current Eiffel 2.3 implementation are slow performers (about 1 second per line of query). We have not tried to make any detailed measurements, but it is clear that most of the time is spent generating the (often huge) parse trees.

Partly this may be inherent in the approach (which is then the price to pay for automation), but we suspect that much can be done in terms of more efficient tree structures, improved generated code from the Eiffel compiler etc.

So far, the long compilation times for the generated parsers (around 300 classes for a very expressive query language) and the slow parsing have been somewhat annoying, but has been more than offset by the extreme flexibility of the approach.

With future Eiffel releases, we hope that performance will cease to be a problem, at least for research compilers.

8 Conclusions

This paper has presented a method to augment a BNF language description with annotations describing the simple tree manipulations needed to transform a parse tree for a language sentence into a syntax tree.

The method has enabled us to develop an object-oriented parser generator, whose generated parsers will produce a syntax tree directly from a language sentence. The syntax tree will only contain terminal operators and a (in most cases small) number of separately specified nodes.

The idea is that the set of terminal operators and new nodes reflects the semantic core of the language, and is normally much more stable than the set of grammar constructs for a language under development.

So by confining the implementation of semantic checking and evaluation to classes corresponding to these nodes, significant syntactical language changes can be made without affecting any hand written code at all, whereas in more traditional environments this would mean constant revolutions.

We feel that the idea of modeling each syntactic construct as a separate abstraction (class) is indeed a strong one, making it very easy to successively decorate an abstract syntax tree and add various semantic actions to it. We were able to implement the scheme described in this paper with very little effort in our Eiffel environment, whereas in a traditional setting we believe the cost would have been to high.

The practical productivity gain using the approach has been great indeed, and we firmly believe that object-oriented techniques have the potential to carry the art of automatic compiler generation several steps further.

The transformation technique described in this paper can be used as a small building block in different language prototyping contexts. For a much more ambitious effort, aiming at creating an integrated incremental language development environment, see [MB+90].

Acknowledgements

The authors are grateful to Per Svensson, National Defence Research Establishment, for valuable discussions.

References

[FL91] Charles N. Fisher and Richard J. LeBlanc, Jr. *Crafting a Compiler with C.* Benjamin/Cummings, 1991.

[HM89] Philip Hucklesby and Bertrand Meyer. The Eiffel Object-Oriented Parsing Library. In *Technology of Object-Oriented Languages and Systems (TOOLS 1)*, pages 501–507, Paris, France, Nov 1989. Société des Outils du Logiciel.

[Joh75] Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler. Comp. Sci. Tech, Rep. 32, Bell Laboratories, Murray Hill, NJ, July 1975.

[LS75] Mike E. Lesk and Eric Schmidt. Lex - A Lexical Analyzer Generator. Comp. Sci. Tech, Rep. 39, Bell Laboratories, Murray Hill, NJ, Oct 1975.

[MB+90] Boris Magnusson, Mats Bengtsson, Lars-Ove Dahlin, Göran Fries, Anders Gustavsson, Görel Hedin, Sten Minör, Dan Oscarsson, and Magnus Taube. An Overview of The Mjølner/ORM Environment: Incremental Language and Software Development. In *Technology of Object-Oriented Languages and Systems (TOOLS 2)*, pages 635–646, Paris, France, March 1990. Prentice-Hall.

[Mey92] Bertrand Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

[MN90] Bertrand Meyer and Jean-Marc Nerson. *Eiffel: The Libraries.* Interactive Software Engineering Inc., Oct 1990. Version 2.3.

[SZ91] Per Svensson and Huang Zhexue. GeoSAL: A query language for spatial data analysis. In *Advances in Spatial Databases*, pages 119–140. Lecture Notes in Computer Science Vol. 525, Springer-Verlag, 1991.

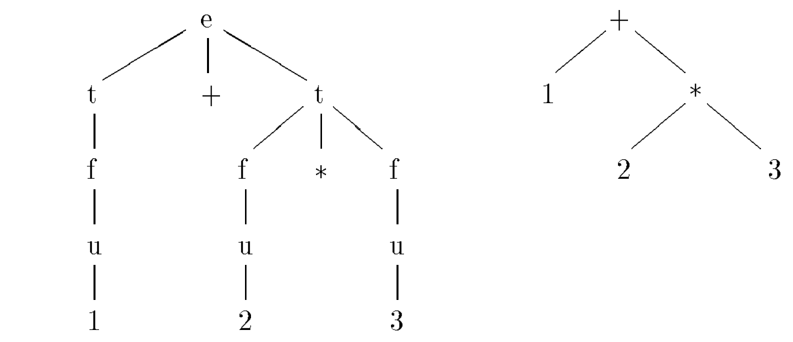

Figure 1: Parse tree to the left and syntax tree to the right.
Legend: e = expression, t = term, f = factor and u = unsigned_int

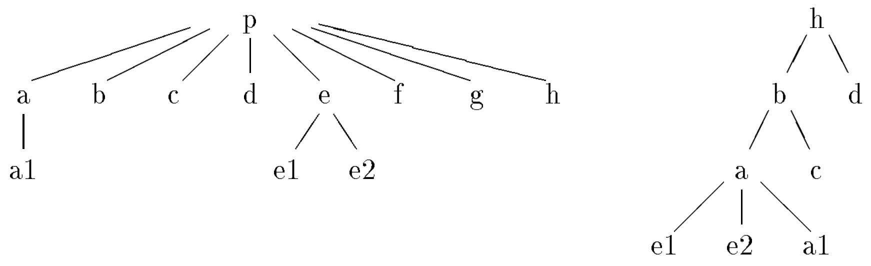

Figure 2: Complex aggregate transformation "8-(2-(1-<5> 3) 4)"

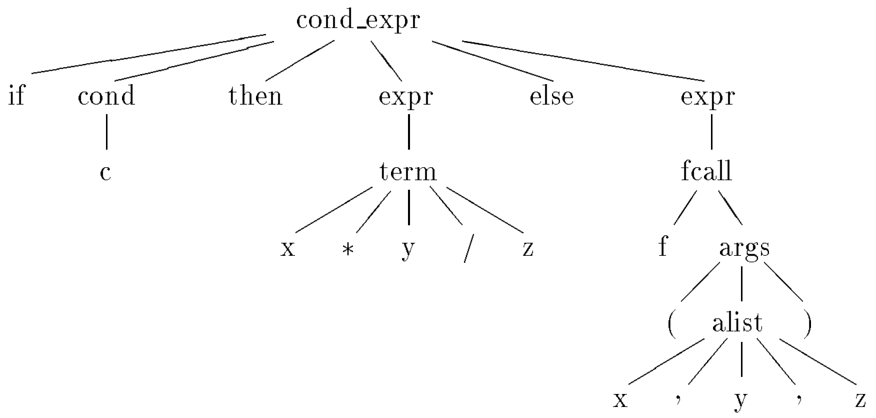

Figure 3: Simplified parse tree for conditional expression

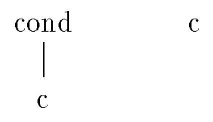

Figure 4: Default transformation

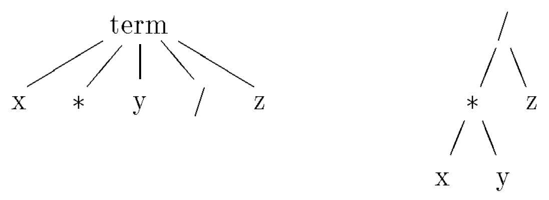

Figure 5: Sequence transformation "LTREE"

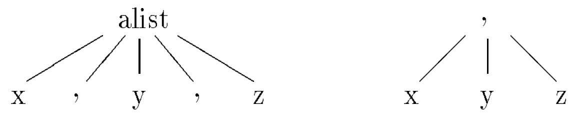

Figure 6: Sequence transformation "BSEQ"

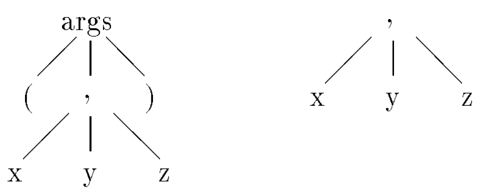

Figure 7: Aggregate transformation "2"

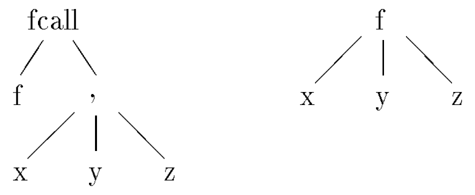

Figure 8: Aggregate transformation "1-<2>"

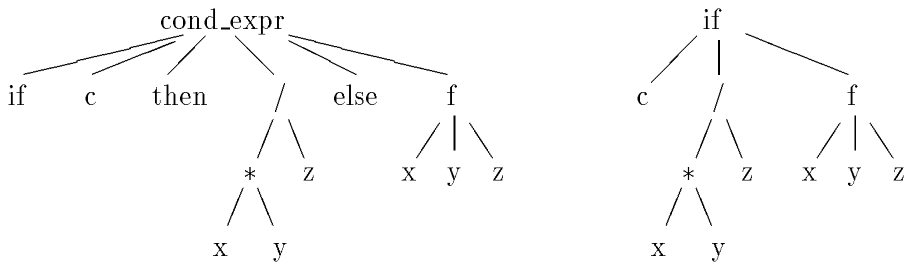

Figure 9: Aggregate transformation "1-(2 4 6)"