Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Maintaining presentation invariants in the Views system

J. Ganzevoort

Computer Science/Department of Algorithmics and Architecture

CS-R9262 1992

Maintaining Presentation Invariants in the Views System

Job Ganzevoort

CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
Email: Job.Ganzevoort@cwi.nl

Abstract

The Views system is an open application integration environment, in which the data layer and user interface layer are supplied by the system. Objects in this environment can be linked through invariants; when the value of either of the objects changes, the others is updated to match the invariant.

The invariant mechanism is also used for the presentations of objects on screen. If an object changes, the presentation is updated, but also the inverse should be true: if the presentation changes, eg. by an edit action, the structure of the object should be updated incrementally to match the presentation.

In this report, an incremental implementation of some functional list operators (map, reduce, filter) is presented, and then, using these operators, a general parsing algorithm is described. The resulting parser is therefore automatically incremental, without any attention being paid to incrementality in the algorithm.

1991 Mathematics Subject Classification: 68N15, 68Q50.

1991 CR Categories: H.1.2, H.5.2, D39, D.3.4, F.4.2, F.3.1., I.1.3.

Keywords and Phrases: user interfaces, ergonomics of computer software, constraint systems, parsing, incrmental techniques.

Contents

1 Introduction **5**
- 1.1 Software ergonomics 5
- 1.2 New trends in user interfaces 6
- 1.3 The VIEWS system 7
  - 1.3.1 The conceptual framework 7

2 Presentation invariants **11**
- 2.1 Tree restructuring 12
- 2.2 Design of presentation invariants 13

3 Incremental List Operators **15**
- 3.1 Lists operators 15
  - 3.1.1 Map 15
  - 3.1.2 Filter 16
  - 3.1.3 Reduce 16
- 3.2 Implementing the list operators 17
  - 3.2.1 Sorting 17
  - 3.2.2 The Structure Watcher 18
  - 3.2.3 Map 19
  - 3.2.4 Reduce 20
  - 3.2.5 Filter 22

4 Parsing **25**
- 4.1 Mapping types on grammars 25
  - 4.1.1 Presentations 25
  - 4.1.2 Creating the grammar 27
  - 4.1.3 Translating a parse tree to an object value 28
  - 4.1.4 Example 29
- 4.2 Incremental lexical analysis 29
- 4.3 Parsing strategies 32
- 4.4 The VIEWS parser 33
  - 4.4.1 The parser's operation 35
  - 4.4.2 Parsing example 36

4.4.3 Local ambiguity packing 38
4.4.4 Lookahead example 39
4.4.5 Handling ϵ-productions 40
4.4.6 Error recovery 41
4.4.7 Locality of changes 42
4.5 Parsing structures 43
4.5.1 Parsing graphical objects 43
4.5.2 Parsing combined presentations 43

5 Conclusions **45**
5.1 General observations 45
5.2 Current Situation 45

Bibliography **47**

Chapter 1

Introduction

1.1 Software ergonomics

The research in software ergonomics has provided many valuable insights concerning factors that influence the productivity in the use of software. In particular, the research has concentrated on the cognitive ergonomic aspects of the user interface (or human-computer interface). One of the lessons learned is that human beings are quite varied in skills. Many user interfaces are unsatisfactory because they are too narrow: they require a certain fixed combination of skills and allow no substitution.
Vital to a good human interface is a human-oriented task analysis. A substantial part of this analysis can be made relatively domain-independent, and thus it is possible to obtain generally valid insights on the design of user interfaces.
Looking at typical present-day products, we find the following kinds of ergonomical shortcomings:

Lack of integration To obtain the desired results, end users often have to combine the working of several applications, that each define their own data format. Once the data of one application is imported into another, it is fixed, and can only be updated by re-importing the changed data, instead of changing it in situ.

Inconsistency & mode confusion When working with interactive programs, users often have to switch context. For example, a user working in the shell starts a news-reading program, replies to an article, so the mail program is started which runs an edit session to create the letter: Four different user interfaces in quick succession. This requires not only knowledge about all these modes, but also at all times awareness of the current application and mode, which is a high cognitive load, and thus a source of errors.

Inflexibility

Arcaneness The exact form of formats required for commands or functions that are not regularly used are often such that it is impossible to remember them when needed, so they have to be dug up from the manuals.

The "Swiss army knife" syndrome Many applications package a number of functions, like, for electronic mail, finding a letter (retrieval), archiving it (storing), composing a letter (editing), etc. Such functions are in fact emasculated versions of much more general functions, applied to one specific context. This can lead to duplication of functionality, inconsistency, and the overall complexity of the user interface.

1.2 New trends in user interfaces

The new trends in user interfaces are the result of the attempts of software designers to overcome some of the problems mentioned above.

WYSIWYG: The term WYSIWYG (What You See Is What You Get) is usually applied to text formatting systems in which a text being edited appears on the screen as it will appear when printed in hardcopy form. This provides the user with an immediate feedback.

Direct Manipulation Rather than giving commands in some more or less arcane command language, the user 'directly executes', for example, the printing of a document by dragging its icon to the icon of the printer. To the user, it is psychologically the same as the real physical execution. There is much less chance of making an error in phrasing the command and the user gets explicit visual clues on how to express the action.

Integrated Information Systems An information system is called integrated if it presents itself to the end user as a collection of functions and tools that are able to cooperate smoothly and that can be handled by the user in an uniform way.

Open System Architectures Open systems give the vendors a bigger potential market, while offering the user more choice and less vendor dependence. It holds the promise that the kind of integration that is needed can be achieved not only within a single system, but also without much additional effort for whole networks.

Object Oriented and Object Centered Programming The coupling of operations relating to objects to the objects themselves has made it far easier to specify and build flexible environments. For example, there is then no need to provide a print capability that is able, once and for all, to print all conceivable documents, which would effectively limit the set of available types of documents. An application designer introducing a new kind of document has, instead, to specify what it means for such a document to be 'printed'. In combination with an open architecture, this means a large increase in the flexibility of systems. A step beyond this is the new paradigm of object-centered programming, in which objects are no longer necessarily passive most of the time. The functionality of the system then results from the interplay of the objects.

1.3 The VIEWS system

VIEWS addresses these problems by supplying a framework that new applications can be added to, offering a consistent and integrated user-interface across applications [9]. The best way to achieve conceptual simplicity in software systems design is to use the power of mathematical abstraction in the design and description of the underlying concepts. The rest of this chapter is an informal description of the conceptual framework, of which the implementation model is most important for the understanding of this report.

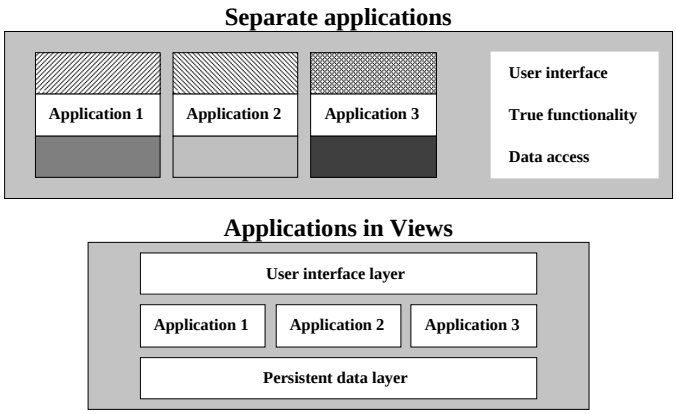
1.3.1 The conceptual framework

For the purpose of this exposition, the terms 'document', 'object' and 'form' will be used interchangeably. The difference is one in emphasis. The term 'document' is most appropriate if we keep the user's logical view of certain data kept in the system in mind. The term 'form' is most appropriate if we want to stress the external appearance of a certain kind of documents. The term 'object' emphasizes a prolonged identity in time during which the value (contents) of an object may change.
The conceptual framework is now simply that the system consists of a collection of objects that are linked by certain 'logical' relationships that hold between their values. This may be described in mathematical terms as a graph or network whose nodes are objects and whose edges are the links connecting objects. Objects may be atomic, but also structured (composed of other objects). In the course of time, new objects and relationships may be created or deleted. There are 'au-

tonomous agents' that may cause objects to change. The user, editing a form, is such an agent, but other agents may, for example, be the clock, or the mail delivery subsystem.
Such a change may, for a fleeting instant, invalidate some of the relationships linking this object to other objects. However, there are 'daemons' guarding these relationships, and whenever the validity of one is impaired, they intercede and restore it by also changing the object at the other object of the link. This may, in turn, invalidate other relationships, which then will also be restored. Thus, a single change may in principle trigger a cascade of changes throughout the network of linked objects, so that the 'events' generated by the autonomous agents drive the semantics of the system.

The data model

VIEWS supplies a data-layer to applications. VIEWS objects are structured, thus containing other objects, and consist only of 'content-full' parts: they contain no detail of formatting or other presentation information, which is added by a separate process when objects are displayed. Each object has a type, which describing the internal structure of the object, and it's external presentation. Any application may import objects from other applications without losing information about their structure and without having to know about files or external storage. The distinction between applications then blurs. For instance, if the user pastes a graphical object into a text document, the graphics is presented in the same way, and is editable in the same way, as before.



Rather than requiring each application to define its own data-formats, file I/O and user interface, VIEWS *supplies a data layer and a user interface layer. The application is only concerned about its true functionality.*

The user's conceptual model

The basic model in VIEWS is: every object in the system is editable, every action is carried out by editing, and you edit the object directly. There is a generic editor which knows about the structure of objects, and allows the user to edit all objects in the same way, regardless of how they are presented.

Implementation model

Types Each object in VIEWS has a type. Types can be primitive, such as numeric and boolean types, or composed from other types using the following type constructors:

- An object of type $\mathit{choice}(\alpha_1, \alpha_2, \ldots, \alpha_n)$ can be of any of the types $\alpha_1, \alpha_2, \ldots, \alpha_n$.
- An object of type $\mathit{compound}(f_1 : \alpha_1, f_2 : \alpha_2, \ldots, f_n : \alpha_n)$ is an n-tuple, where field i has name f_i and type α_i. π_f applied to a compound c selects the field $c.f$.
- The type $\mathit{sequence}(\alpha)$ is the type of lists of elements of type α.
- An object of type $\mathit{reference}(\alpha)$ is a pointer to an object of type α.

Invariants Objects are linked by invariants, that specify the relations that must hold between objects. A link is a vector from some objects $x_1 \ldots x_m$ to some other objects $y_1 \ldots y_n$, labeled with an update operation. If any of the objects $x_1 \ldots x_m$ is changed, the link is outdated, and at some later instant, the update operation is executed, which updates the values of $y_1 \ldots y_n$.
The invariant $y = fx$ can be implemented by the two links $y = fx$ and $x = f^{-1}y$, which generates the bidirectional behavior.

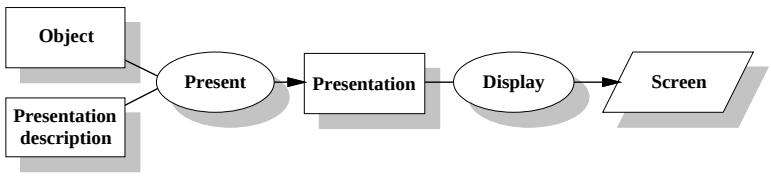
Functions A function can be *built-in* or generated from other functions. One method of generating a new function is currying; If f is a function taking two arguments, (fx) is a function in one argument. Applying this function to y is equivalent to applying f to x and y. For example, $(+1)$ is the increment function. Functions in VIEWS can be partially parameterized, where the omitted arguments of the functions are denoted by the symbol $\Box$. Currying can be expressed using partial parameterization: $(\mathit{curry} fx) = f(x,\Box)$, where f has type $\alpha \times \beta \rightarrow \gamma$, x has type α, and $f(x,\Box)$ has type $\beta \rightarrow \gamma$.

Sharers In some cases, it is useful to distinguish between the object and its contents. The contents can be shared with other objects, guaranteeing the sharers' values to be equal. This may be regarded as multiple instances of the same object, or as an optimization of equality links. Some details of the implementation, especially of the filter-invariant (see 3.2.5), rely on the concept of sharers.

Chapter 2

Presentation invariants

Each object in the VIEWS system has a type, which describes the internal structure of the object, and its external presentation. When an object is displayed, the description of its external presentation is accessed and used to determine how the object should look. The invariant mechanism is used very generally throughout the system, so that presenting objects on the screen is done by the application of the invariant "The presentation on the screen matches the object": if the object changes, then the screen gets updated.



Presentation of documents on the screen is done using the invariant mechanism. When a document is 'visited', a new presentation object is created, linked by an invariant to the object to be visited and the description of how the object should be displayed.

The type of a clock might be:

```
Type clock = compound(h: hours, m: minutes, s: seconds)
presented row[h, ":", m, ":", s]
```

What follows the word *presented* is any expression yielding a 'presentation'; in this case, the function *row* takes a list of objects and their presentations, and joins them together horizontally. So a clock might look like '12:21:30', depending on the presentation of hours, minutes and seconds.
Even though the presentation of an object is conceptualized as a single invariant between object and presentation, the system can actually

break them down into lower-level invariants, which has an extra advantage of automatic screen update optimization: if only the seconds field changes, only the seconds part of the presentation needs to be updated.
In the editor, a mixed style between free textual editing and rigid syntax directed editing is desired. That means that it in some situations, it should also be possible to focus on an object's presentation, allowing the text to be edited. In some situations, the user's conceptual model of the object relates to the appearance of his document, rather than to the structure. An expression displayed as $2 + 3 * 5$ may be regarded as linear (presentation) rather than hierarchical (structure). In a word processor, where words are separated by spaces, a user might select a space and remove it, in order to concatenate two words. In these cases, it should be possible to move the focus through the presentation of the object. Selecting an object's presentation is not always meaningful however (see [4, ch. 3]).
The presentation invariant thus needs to work two-way; the presentation should be updated whenever the object changes, but if the presentation is edited, the contents of the object should be updated to match that presentation. The presentation invariant thus needs to be composed of the functions, one that generates a presentation for an object, and the inverse function that generates an object value given the presentation.

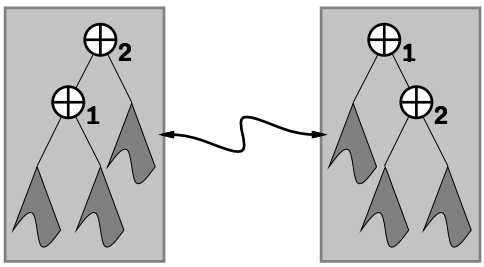
2.1 Tree restructuring

As an example of the kind of problems expected for the presentation invariants, the following problem is investigated: If a user is editing an expression, a parse tree will be maintained. For $a + b + c$ that would be the tree $((a + b) + c)$. If the second operator is changed into a $*$, the expression's presentation will be $a + b * c$ and the parse tree $((a + b) * c)$ would not match the presentation. This problem can be solved by two totally different approaches:

1. Assume the parse tree is correct, and use brackets where necessary in the presentation. This solution is chosen in more classical syntax-directed editors, eg. the editors generated by the Synthesizer Generator [11].

2. Assume the presentation is correct and adjust the parse tree. We believe this fits better in the VIEWS model, since the parse tree is a function of the presentation and vice versa. Therefore, this approach is preferred.

An experiment was used to clarify the expected behavior: An invariant was created between the operators, with knowledge of the precedences, that can restructure $((a + b) * c)$ to $(a + (b * c))$ as in:



Operator tree restructuring

Based on the grammar given below, an expression was created, which was displayed both structurally as $(3+(5*7))$ and flattened as $3+5*7$. Changing the operators in the flattened display had the desired effect of restructuring the parse tree.

Exp → Exp Op Exp | (Exp) | Num
Op → + | - | * | /

Expression grammar

2.2 Design of presentation invariants

Two approaches to the design of presentation functions and the inverse were recognized. These approaches are not totally different, but represent different viewpoints that should be integrated.

Structure-directed presentation To construct the presentation of an object, the presentations of the sub-objects are glued together. As an example, the *map* operator (see 3.1.1) on lists applies a function to all elements of the list. This can be used to merge the presentations to all of the elements to form the presentation of the entire list. In this approach, generally applicable strategies to update the structure of the object if the presentation changes, as for example the tree restructuring above, are needed. One way to generalize the method of this example is to create mechanisms (for example structure watchers, see 3.2.2 and 5) to guard the restrictions imposed on a structure.

Parsing The inverse process of the presentation is parsing, to which all of chapter 4 is devoted. Parsing graphical presentations of objects is hard though. Therefore, the structure of the presentation should resemble the structure of the object, hinting the parser

how to (re)construct an object from the presentation. Section 4.5 elaborates on this, and related problems.

Chapter 3

Incremental List Operators

Expressions in a functional language can be constructed, manipulated and reasoned about, like any other kind of mathematical expression, using more or less familiar algebraic laws. This provides a conceptual framework for programming which is simple, concise, flexible and powerful. For an introduction to functional programming, see [3]. A textbook on the implementation of functional languages is [6].

Lists are finite sequences of values of the same type. We use the notation $[\alpha]$ to describe the type of lists whose elements have type α. Lists will be denoted using square brackets; for example $[1, 2, 1]$ is a list of three numbers. The notation used is adopted from [2].

In this chapter, three useful operations on lists are introduced, followed by their implementations. These operations will be used very generally in the presentation invariants. A vital aspect of the implementation of the operators is that they work in an incremental manner; a slight change in the input list should update the result, and the costs of the updates should be minimal. For example, the map operator can be used to build the presentation of a list from the presentations of the elements. If one element changes, only the presentation of that element has to be updated, and not for the entire list.

3.1 Lists operators

3.1.1 Map

The operator $*$ (pronounced "map") takes a function and yields a function that operates on each element of a list. We have

$$f * [a_1, a_2, \ldots, a_n] = [fa_1, fa_2, \ldots, fa_n]$$

The type of $*$ is given by

$$* : (\alpha \to \beta) \to [\alpha] \to [\beta]$$

3.1.2 Filter

The operator $\triangleleft$ (pronounced "filter") takes a predicate p and returns a function that selects all elements of a list that satisfy p. For example,

$$prime \triangleleft [1..10] = [2, 3, 5, 7]$$

The type of $\triangleleft$ is given by

$$\triangleleft : (\alpha \to Boolean) \to [\alpha] \to [\alpha]$$

3.1.3 Reduce

The operator / (pronounced "reduce") takes an operator $\oplus$ and yields a function that works on a list. Its effect is to insert $\oplus$ between adjacent elements of the list. Thus:

$$\oplus/[a_1, a_2, \ldots, a_n] = a_1 \oplus a_2 \oplus \cdots a_n$$

For the right-hand side of this equation to be unambiguous in the absence of brackets, the operator $\oplus$ must be associative. In fact, the form $\oplus/x$ is only permitted when $\oplus$ is associative, so the grouping of terms on the right is irrelevant.
For a singleton list $[a]$, we have from the informal description of $\oplus/$ that $\oplus/[a] = a$. For the empty list, if $\oplus$ has an identity element e, $\oplus/[] = e$; otherwise $\oplus/[]$ is not defined. This means that $+/[] = 0$, $\times/[] = 1$, but $\uparrow/[]$ is undefined. This preserves the law

$$\oplus/(x +\!\!+ y) = (\oplus/x) \oplus (\oplus/y)$$

The symbol $+\!\!+$ is the concatenation operator: $[a_1..a_n] +\!\!+ [b_1..b_n] = [a_1..a_n, b_1..b_n]$.
The type of / is given by

$$/ : (\alpha \times \alpha \to \alpha) \to [\alpha] \to \alpha$$

Thus, in the combination $\oplus/x$, the operator $\oplus$ has a type of the form $\alpha \times \alpha \to \alpha$ and x has a type of the form $[\alpha]$. The combination will then have type α.

Reduce variants The 'left reduce' $\oplus \not\rightarrow_e [a_1, a_2, \ldots, a_n]$, which evaluates $(((e \oplus a_1) \oplus a_2) \cdots \oplus a_n)$, and 'right reduce' $\oplus \not\leftarrow_e [a_1, a_2, \ldots, a_n]$, which evaluates $(a_1 \oplus (a_2 \oplus \cdots (a_n \oplus e)))$, can be used if the operator $\oplus$ is not associative. These have the types:

$$\not\rightarrow : \beta \rightarrow (\beta \times \alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow \beta$$
$$\not\leftarrow : \beta \rightarrow (\alpha \times \beta \rightarrow \beta) \rightarrow [\alpha] \rightarrow \beta$$

In $\oplus \not\rightarrow_e x$, the unit e has type β, the operator $\oplus$ has type $(\beta \times \alpha \rightarrow \beta)$, x has type $[\alpha]$ and the result has type β.

The reduce operators can be used very generally to express some form of iteration, where the order is ($\not\rightarrow$, $\not\leftarrow$) or is not ($/$) relevant. For example, the special symbol $\sum$ corresponds to $+/$, as in:

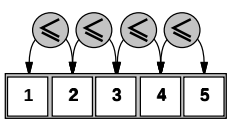
$$\sum_{i=1}^{n} f_i = +/f * [1..n]$$

3.2 Implementing the list operators

The list operators were implemented using the invariant mechanism. This implied that an application can impose an invariant structure on the sequence, which may have to be adjusted as a side-effect of inserting an element into the sequence, or removing one from it. This can be demonstrated with a simple bubble-sort application.

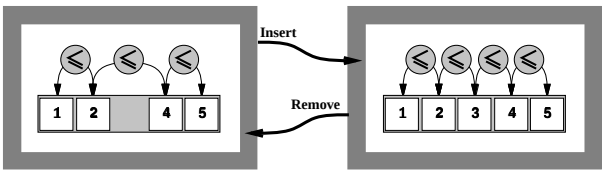
3.2.1 Sorting

A sequence of integers was sorted by joining each two consecutive elements with an invariant $x \leq y$. If the invariant is invalidated, it is restored by swapping x and y.



Bubblesort approach to sorting

This solution sorted the sequence, but when an element is inserted into the sequence, two invariants have to be created and one deleted. For deletes the inverse is needed.



Restructuring the sort-links

This was solved by using a *structure watcher* (see next section). The side-effect of an insertion into, or removal from the sequence should be the restructuring of the invariants.

3.2.2 The Structure Watcher

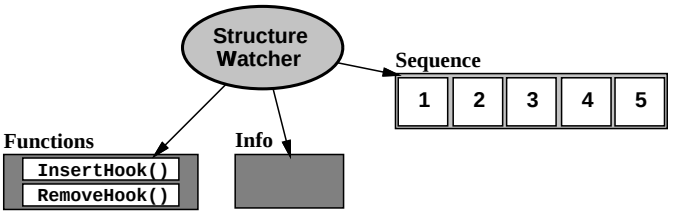
The structure watcher is an aid to the application to maintain the invariant structure imposed on a sequence.

Design principles

The solution chosen was to define the structure watcher as a mechanism for attaching two callback-functions to a sequence. Whenever an element is inserted into, or removed from the sequence, the corresponding function is called. The update and the following callback function call form an atomic action. The application's callback-functions are responsible for adjusting the invariant structure. Using this philosophy, the structure watcher need not have any knowledge of the actual structure the application imposes on the sequence.

Implementation

The structure watcher is an administrative link between the sequence, a compound of two function objects, and an *Info* object. The sequence-update operations search for a structure watcher link, and call the insert or remove callback-function with the inserted or removed child and the *Info* object as parameters. The insert callback-function is called after the update, whereas remove callback-function is called before, thereby making the siblings and parent available to the callback-function, since they may be needed in the restructuring.
The *Info* object can be of any type. It can be used by the update functions for any extra context information needed. Since in general, the *Info* object is not visible, it is not necessary to react to changes made to it by the user's editing.



The structure watcher link

3.2.3 Map

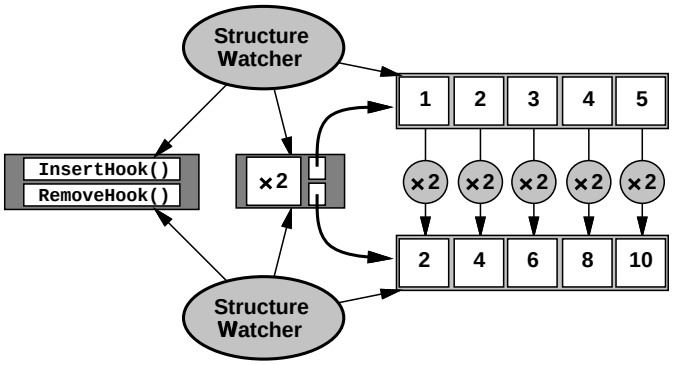
If the invariant $Y = f * X$ is created, then any subsequent change in either X or Y should affect the other to re-establish the invariant incrementally. For the sake of discussion, let's specify that $X = [1, 3, 5]$ and $Y = (\times 2) * X$, so Y will be the list $[2, 6, 10]$. Two types of changes can be distinguished:

Contents changes If an element is changed, the corresponding element in the other should be updated to match. Changing the 3 in X to 4 should change the 6 in Y to 8. Note that invariants are two-way, so changing 10 in Y to 100 should change 5 in X to 50. This type of changes can be catered for incrementally by creating, for each pair of corresponding elements, the invariant $y_i = 2 \times x_i$.

Structural changes If an element is removed from, or inserted into either X or Y, the corresponding element in the other sequence should also be removed or inserted: deleting 1 from X should remove 2 from Y. Appending 8 to Y should append 4 to X. These changes are best handled with the help of a structure watcher.

Implementation

For the implementation of the invariant $Y = f * X$, structure watchers are attached to both X and Y. The information object *Info* used by the callback-functions contains the function object f, and references to X and Y. For each pair (x_i, y_i) the invariant $y_i = fx_i$ is maintained. The *Info* object and the callback-functions are shared by the structure watchers. In the following description, it is assumed, without loss of generality that the removed or inserted element is x_i, an element of X. Its left and right siblings are x_{i-1} and x_{i+1}.



Map implementation

RemoveHook(x_i, *Info*) This is the removal callback-function for the sequences. It locates the corresponding element y_i in the other sequence by looking at the invariants of x_i, deletes the invariant and then removes y_i. If y_i cannot be found, that must be because removing y_i induced removing x_i, so no more work needs to be done.

InsertHook(x_i, *Info*) This is the insertion callback-function. It identifies the corresponding elements y_{i-1} and y_{i+1} of its siblings. If these are siblings, the corresponding element y_i is created and inserted between y_{i-1} and y_{i+1}. If these are not siblings, the element between them has to be y_i.

If x_i is the first element of the sequence, y_{i+1} is expected to be the first of the other sequence. If so, y_i is created and prepended to the other sequence, if not, the first has to be y_i. Mutatis mutandis the same if x_i is the last.

If the invariant between x_i and y_i did not exist yet, it is created now. In the *Info* object are references to X and Y. If the sequence x_i is in, is X, the invariant to be created is $y_i = f\, x_i$. If x_i is in Y, the invariant is $x_i = f\, y_i$.

This mechanism correctly handles recursion and —if allowed— circularities like $Y = f * X; Z = g * Y; X = h * Z$.

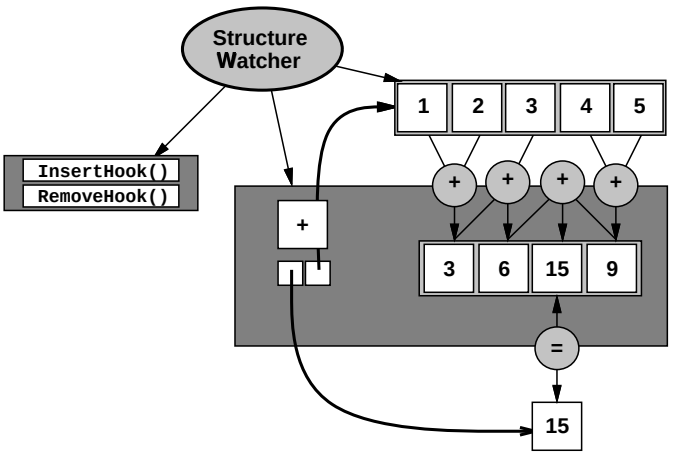
3.2.4 Reduce

For the left and right reduce invariants $r = \oplus \not\rightarrow_e [a_1, a_2, \ldots, a_n]$ and $r = \oplus \not\leftarrow_e [a_1, a_2, \ldots, a_n]$, a linear implementation is needed. For the operation $r = \oplus/[a_1, a_2, \ldots, a_n]$, the evaluation order is not specified. In a tree, the number of invariants that may have to be re-established to update the result, which is the length of the path to the top node,

is logarithmic in the number of elements of the sequence, instead of linear, so the evaluation structure chosen is a tree.

Implementation

A structure watcher link is attached to the sequence. The *Info* object contains the operator, references to the sequence and the result, and a sequence of subresults of the evaluation. In the evaluation tree, a node z is joined to its children x and y by an invariant $z = x \oplus y$. The result r is joined to the top-node t by the invariant $r = t$.



Reduce implementation

RemoveHook(x_i, *Info*) This is the removal callback-function for the reduce. The element has to be removed from the evaluation tree. At this point, some tree balancing could be done, but this is omitted in the current implementation. If x_i is the top-node, the result is unset, and this function terminates. If not, an invariant $z = x_i \oplus y$ or $z = y \oplus x_i$ is searched. y will replace the node z in the tree. Therefore, the invariant $z' = z \oplus y'$ or $z' = y' \oplus z$ is searched and replaced by $z' = y \oplus y'$ or $z' = y' \oplus y$. If z is the top-node, z' is the result, so the invariant $z' = z$ is replaced by $z' = y$. Eventually, z is disposed.

InsertHook(x_i, *Info*) This is the insertion callback-function. The element x_i has to be inserted into the tree. If x_j is a sibling of x_i and z is the parent of x_j, a new subresult z' is created. The invariant $z = x_j \oplus y$ is then replaced by $z = z' \oplus y$ and $z' = x_i \oplus x_j$.

Note that, in general, $z = x \oplus y$ is not equivalent to $z = y \oplus x$ since the operator is not required to be commutative. The same care to preserve the order should be taken as before, but a uniform treatment of left/right cases is notationally convenient.

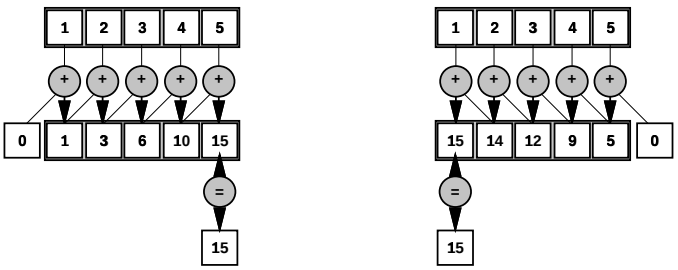
If x_i is the first or last element of the sequence, the choice (left or right sibling) is trivial, but if x_i has two siblings, both are valid. To make any sensible choice in order to obtain the best balance (shortest average path to the result), extra information is needed in the tree nodes. In the current implementation, a random choice is made.

Left and right reduce variants

The reduce variants $\oplus \not\rightarrow_e[a_1, a_2, \ldots, a_n]$ and $\oplus \not\leftarrow_e[a_1, a_2, \ldots, a_n]$ are usually implemented as:

$$\begin{aligned}
\oplus \not\rightarrow_e[] &= e \\
\oplus \not\rightarrow_e[a_1, a_2, \ldots, a_n] &= \oplus \not\rightarrow_{(e \oplus a_1)}[a_2, \ldots, a_n] \\
\oplus \not\leftarrow_e[] &= e \\
\oplus \not\leftarrow_e[a_1, a_2, \ldots, a_n] &= a_1 \oplus (\oplus \not\leftarrow_e[a_2, \ldots, a_n])
\end{aligned}$$

The implementation (that will be) used here transforms the recursion into a sequence of subresults $[r_1, \ldots, r_n]$, with the invariants $r_i = r_{i-1} \oplus a_i$, $r_0 = e$ and $result = r_n$ for the left reduce, and $r_i = a_i \oplus r_{i+1}$, $r_{n+1} = e$ and $result = r_1$ for the right reduce:



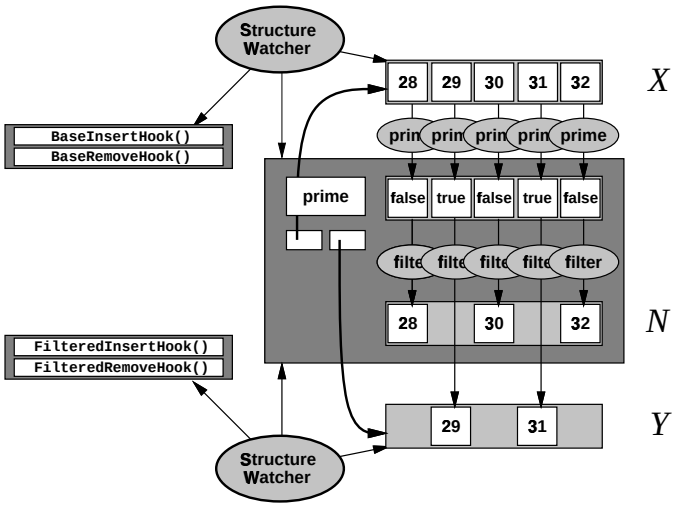
Implementation of $\oplus \not\rightarrow_e[a_1, a_2, \ldots, a_n]$ *and* $\oplus \not\leftarrow_e[a_1, a_2, \ldots, a_n]$

3.2.5 Filter

The implementation of an invariant $Y = p \triangleleft X$ is slightly more complex, since the structure of the result depends on the contents of the sequence elements. If an element of X is changed, the predicate has to be re-applied to it, and if the result has changed, the result sequence has to be updated. If we define the function $split(p, X) = (Y_1, Y_2)$, where Y_1 consists of all elements of X that satisfy p, and Y_2 consists of all elements of X that don't, then the filter function can be defined as $\triangleleft = \pi_1 \cdot split$.

Implementation

Let's say that $Y = p \triangleleft X$, or $(Y, N) = split(p, X)$. Then for each element x in X, a boolean x_b is maintained with a predicate invariant $x_b = px$. A sharer x' is created with a special filter invariant from x_b to x', specifying that if x_b is true, then x' should be in Y, else x' should be in N.



Filter implementation

The *Info* object The *Info* object contains the predicate function, the sequence of booleans X_b, the sequence of filtered-out elements N and references to X and Y.

FilterLink(x_b, x') This is the function implementing the filter invariant. If x_b changes to true, move x' to Y, If x_b changes to false, move x' to N. Since the order in Y and N[1] is the same as in X, the correct position for x' has to be found by finding the last y_b that precedes x_b, such that y' is in the correct sequence (Y or N). x' is then inserted after y'.

The quintessence of finding the position for x' is a UNION-FIND problem with deletes. The (linear) approach used is the simplest. If X_b is implemented as a collection of trees however, a logarithmic algorithm can be used[2]. This will only be noticeable at (very) large segments of filtered-out elements.

BaseRemoveHook(*Me, Info*) Find Me_b and Me', remove the links and then the objects.

[1] as N is an internal object, not meant to be visible, keeping it in the same order (or even as a sequence) is not strictly necessary.

[2] The deletes prevent path-compression

BaseInsertHook(*New, Info*) If New_b and New' do not exist yet, create them. Let New_b be false, and insert New' into N. If it should be in Y, the filterlink will bring it there anyway.

FilteredRemoveHook(*Me', Info*) Find Me_b and Me, remove the links and then the objects, but only if Me_b is true. This test is needed to prevent an object from being removed from X as a side-effect of moving its sharer from Y to N.

FilteredInsertHook(*New', Info*) If New_b and New do not exist yet, create them. Let New_b be true, since New' is in Y. If New does not satisfy p, the predicate link will set New_b to false, and the filterlink will then move New' to N. This means that the newly inserted element is also inserted into X, and then immediately removed again from Y. This seems to be the most sensible reaction to inserting an element into a sequence where it should not be, since if, at a later instant, the elements value changes such that it satisfies p, it is automatically re-inserted into Y.

The implementation $Y = +\!\!+/(f * X)$, where $fx = \mathit{if}\ px\ \mathit{then}\ [x]\ \mathit{else}\ []$, and $+\!\!+$ is the concatenation operator, would also be valid for $Y = p \triangleleft X$. The current implementation is much more direct, and more efficient in terms of time and number of objects needed to maintain the invariant. Besides, the invariants $L = L_1 +\!\!+ L_2$ and $x = \mathit{if}\ c\ \mathit{then}\ x_1\ \mathit{else}\ x_2$ are not implemented yet. The concatenation would be implemented using structure watchers, the *if*-invariant is implemented trivially.

Chapter 4

Parsing

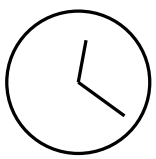
4.1 Mapping types on grammars

Since the presentation information in the type of the object is used, the grammar for the parser has to be generated from the type system. A type definition gives a name, a structure and a presentation, as in the type definitions below. A number of standard types, like numbers and text, are provided, and 5 ways to build new types: compounds, sequences, choices, references, and likenesses. Likenesses are a way of specifying a synonym of a type, possibly with a different presentation, as in:

Type clock = compound(h: hours, m: minutes, s: seconds)
presented row[h, ":", m, ":", s]

Type gclock = like clock
presented circled(combine[mhand, hhand])
    mhand = line(20) rotated(m×6)
    hhand = line(15) rotated((h mod 12)×30 + m/2)

12:21:30
Presentation of an object of type clock



and of type gclock

For the presentation invariant of an object, the grammar of the object's type is created to enable parsing of the presentation. The parse tree can then be translated to object values.

4.1.1 Presentations

The presentation of an object can be built using the following rules:

- *vline*(*length*), *hline*(*length*), *circle*(*diameter*), *square*(*width*) and *box*(*width, height*) are basic presentations.
- A presentation p can be manipulated to form a new presentation in the following ways: *rotated*(*p, degrees*), *at*(p, *pos*(x, y)), which

moves the picture to a new origin, and *circled*(p), *squared*(p) and *boxed*(p), which draw a circle, square or rectangle around p.

- *literal*(x) is just the string x.
- *std*(x) is the standard presentation for a type in a certain class; *std*(s) presents text-like types, *std*(d) reals and *std*(i) presents integer-like types. In the example type above, the type *hours* would probably be *like*$(integer)$, presented *std*(i). *std*(x) for other values of x can be for multi-media objects, types, certain pictures, etcetera.
- *field*(f_i), where the type is a *compound*$(f_1 : t_1, f_2 : t_2, \ldots, f_k : t_k)$, presents field f_i with the presentation for t_i.
- *alt*() is a presentation for choices. The object is presented using the *chosen* type; if the object x is of type *choice*$(t_1, t_2, \ldots, t_k)$, then *chosen*(x) is of type t_i for some $i \in [1..k]$.
- *loop*(p) and *seploop*(p, s) are presentations for sequences. If type $T =$ *sequence*(t), *loop*(p) will apply the presentation p to each of the elements of the sequence. *seploop*(p, s) also applies the presentation p to each of the elements, but separates the elements with the presentation s. Typically, p will be *self*() and s will be a *literal*(`", "`).
- *self*() is the presentation of the entire object. Displaying an object of type $T = S$ presented *self*() will therefore loop. If however *self*() is enclosed in a *loop*(p), it refers to the child to be displayed. It will then follow the presentation of the type of the children, which is often precisely what is desired.
- If $p_1, p_2, \ldots, p_k$ are presentations,
  - *row*$[p_1, p_2, \ldots, p_k]$ is a presentation putting $p_1, p_2, \ldots, p_k$ adjacent to each other.
  - *hang*$[p_1, p_2, \ldots, p_k]$ is a presentation stacking $p_1, p_2, \ldots, p_k$ on top of each other (p_1 on top).
  - *pile*$[p_1, p_2, \ldots, p_k]$ is a presentation stacking $p_1, p_2, \ldots, p_k$ on top of each other (p_k on top).
  - *combine*$[p_1, p_2, \ldots, p_k]$ is a presentation putting $p_1, p_2, \ldots, p_k$ on the same spot.

  If p_1 and p_2 are presentations yielding  and  , then *row*$[p_1, p_2]$ yields , *hang*$[p_1, p_2]$ yields , *pile*$[p_1, p_2]$ yields , *combine*$[p_1, p_2]$ yields .

4.1.2 Creating the grammar

In the generated grammar, the symbols are not text strings. Instead, the nonterminals are types, and the terminals are primitive presentations, *literal*(*string*) or *std*(*string*). This does not make the task of the parser generator any more difficult, while reducing some problems of the grammar generator. In the discussion of the parser, textual names for the nonterminals are used.

To generate a grammar from a type:

- Create an empty grammar.
- If *type* is "Type *name* = *structure* presented *presentation*", add the production "*type* → *build-rhs*(*presentation, structure*)" to the grammar.

The function *build-rhs*(*presentation, structure*) is defined as:

- If *presentation* is a *row*(*list*) or a *hang*(*list*), ...
  Return flatten map *build-rhs*(□, *structure*) *list*.
- If *presentation* is a *pile*(*list*), ...
  Return *build-rhs*(*hang*(*reverse*(*list*)), *structure*), since the presentation $pile[a_1, a_2, \ldots, a_n]$ is equivalent to $hang[a_n, \ldots, a_2, a_1]$.
- If *presentation* is a *literal*(*string*), ...
  Return a list containing this literal.
- If *presentation* is a *std*(*string*), ...
  Add the token with the regular expression for *string* to the current grammar and return a list containing the token.

  The regular expression for *std*(*i*), the standard presentation for integers, could be $[0\text{-}9]^+$, but for *std*(*s*), the standard presentation for strings, a sensible choice is harder to make. If however the structure remains available in the presentation level, *std*(*s*) could recognize just the word it presents, which suffices.
- If *presentation* is a *field*(*name*), ...
  structure should be a *compound*(..., *name* : *type*′, ...). Merge the grammar for *type*′ into the current grammar and return a list containing *type*′.

  To avoid a looping recursion, first complete the current production, before adding the production for *type*′.

- If *presentation* is an *alt*(), ... *structure* should be a *choice*($type_1, \ldots, type_n$). Create a new, unique type $type'$. For each $type_i$, merge the grammar for $type_i$ into the current grammar, and add the production "$type' \rightarrow type_i$". Finally, return a list containing $type'$.

- If *presentation* is a *loop*(*sub*), ... *structure* should be a *sequence*(*subtype*). Merge the grammar for *subtype* into the current grammar. Create new, types $type_1$ and $type_2$. Add the productions "$type_1 \rightarrow type_2$" and "$type_1 \rightarrow type_1 type_2$" to define a left-recursive sequence, and "$type_1 \rightarrow$ *build-rhs*(*sub, sub-structure*)", where *sub-structure* is the structure of *subtype*. Finally, return a list containing $type_1$.

- If *presentation* is a *seploop*(*sub, sep*), ... This is essentially the same as a *loop*(*sub*), except that the elements are separated by *sep*, which is typically a *literal*(", "). The second production differs from *loop*(*sub*). It is: "$type_1 \rightarrow type_1$ *build-rhs*(*sep*)$type_2$".

- If *presentation* is a *self*(), ... it is probably enclosed in a *loop*() or *seploop*(). Return structure.

The discussion of the generation of productions for graphical presentations is postponed to section 4.5.1.

4.1.3 Translating a parse tree to an object value

If a parse tree is constructed, it should be translated into the corresponding object value. Actually, the parse tree is not constructed. The edges from a stacknode to the previous nodes (see 4.4.1) are labeled with objects instead of parse trees. While the presentation of the type corresponds to the concrete syntax of the grammar, the type structure corresponds to the abstract syntax. To create the object value, which is the abstract syntax tree, the abstract syntax trees of all the children in the concrete syntax tree are merged. The structure and presentation of the type are compared to create the translation. For example, a *field*(f_i) in the presentation of a *compound*($f_1 : t_1, f_2 : t_2, \ldots, f_n : t_n$) means that the corresponding child in the concrete syntax tree is child number i in the abstract syntax tree. Presentations such as literals that do not correspond to elements in the abstract syntax are simply omitted, elements in the abstract syntax that do not occur in the concrete syntax will either receive the "unknown" value, or their old value, if that is available.

4.1.4 Example

For the type *stmt* and its subtypes on the left hand side, the grammar on the right hand side is generated by the rules above.

Type for *stmt* and subtypes	**Grammar**
Type stmt = compound(lhs: id, rhs: exp) presented row[lhs, ":=", rhs]	stmt → id ":=" exp
Type exp = choice(opexp, brexp, num) presented alt()	id → std(s) exp → opexp exp → brexp exp → num
Type opexp = compound(left: exp, oplus: op, right: exp) presented row[left, oplus, right]	opexp → exp op exp
Type brexp = like exp presented row["(", alt(), ")"]	op → op' op' → add op' → sub op' → mul op' → div
Type op = choice(add, sub, mul, div) presented alt()	add → "+" sub → "-" mul → "*" div → "/"
Type add = nil presented "+" Type sub = nil presented "-" Type mul = nil presented "*" Type div = nil presented "/"	brexp → "(" exp ")"
Type num = like(integer) presented std(i) Type id = like(text) presented std(s)	num → std(i)

4.2 Incremental lexical analysis

The task of the lexical analyzer is to group characters into recognized 'words'. For example if an expression is presented as '15+3', it consists of the words *num*(15), *op*(+) and *num*(3). If the + is deleted, the presentation is '153', and the words *num*(15) and *num*(3) should therefore be joined to form *num*(153). In the same spirit, the editor should —when working on a textual document that is divided in chapters, sections, paragraphs, sentences and words— join words when the spaces between them are deleted, split a paragraph in two when a delimiter (eg. blank line) is inserted, etcetera.

Regular expression matching A *regular expression* over an alphabet Σ specifies a *language* over Σ, a set of recognized strings of symbols taken from Σ. Regular expressions are built using the following rules:

1. ϵ is a regular expression that denotes the empty string.

2. For each symbol $a \in \Sigma$, the regular expression a denotes the language $\{a\}$.

3. If r and s are regular expressions denoting $L(r)$ and $L(s)$, then r and s can be combined by the following operators, introduced in ascending order of precedence:

   Alternation $r|s$ is a regular expression denoting $L(r) \cup L(s)$, the set of words w, where $w \in L(r)$ or $w \in L(s)$.

   Concatenation rs is a regular expression denoting $L(r) \cdot L(s)$, the set of words uv, where $u \in L(r)$, $v \in L(s)$.

   Kleene closure r^* is a regular expression denoting $L(r)^*$, the set of words $w_1 w_2 \cdots w_n$, $n \geq 0$, where $w_i \in L(r)$.

   Grouping (r) is a regular expression denoting $L(r)$.

   Some shorthands commonly used are $r^+ = rr^*$, for one or more occurrences of r (r^* specifies zero or more), $r? = r|\epsilon$ for zero or one, and character ranges: $[abc] = a|b|c$, $[a\text{-}z] = [abc \cdots z]$, $[\hat{}\, xyz] = \Sigma - [xyz]$.

A *nondeterministic finite automaton* (NFA, for short) is a mathematical model that consists of an alphabet Σ, a set of states S, a start state $s_0 \in S$, a set of accepting (final) states $F \subset S$ and a transition function *move* that maps state-symbol pairs to sets of states. A recognizer for a language L is a program that determines whether an input string x is in the language L. A recognizer for a regular expression r may operate by creating an NFA for r, and try and find a sequence $s_0, s_1, \ldots, s_k$ such that $s_{i+1} \in move(s_i, a_i), s_k \in F$ and $a_1 a_2 \cdots a_k = x$. A lexical analyzer is a program that takes a string x of symbols from Σ, and a set of tokens T with associated regular expressions $R(T)$, and divides x into *lexemes*, each of which is recognized by some regular expression $r \in R(T)$. The output is the a sequence (t_i, l_i), where t_i is the token matching l_i, and $l_1 l_2 \cdots l_n = x$.

Current implementation The input is a sequence of words, the output is a sequence of matches, generated by a *map*(*match, input*), where *match*(*word*) is a function that finds the longest, first match of *word*, according to the set of regular expressions and literals, splitting the word if necessary. Between every two consecutive matches, a link exists[1] that joins them if that results in a longer match.
The matching function is built on top of the `regexp` library. The advantages are in simplicity; the use and maintainance of the DFA's is done by the library functions.

[1] This scheme is maintained with a structure watcher, the same way as in the bubblesort example (section 3.2.1).

Example The set of regular expressions is {"+", *num*: [0-9]$^+$}, the input is "[15+3]". The function *match*(15+3) will find *num*(15) as best match, and split the input into [15, +3]. The function *match*(+3) will then find + as best match, and split the input into [15, +, 3]. *match*(3) will the find a complete match *num*(3). The generated output is then [*num*(15), +, *num*(3)]. If the + is next removed from the input, the invariant *join*(15, 3) then finds a longer match, *num*(153), and joins the two words to form [153], and output [*num*(153)].

Problem If the set of regular expressions is [aaa, a], the input [aaa] will match aaa. If the last a is deleted, the input word will split, to form [a, a]. Now appending an a will result in [a, a, a], and the longer match [aaa] is not found. The matches found can therefore depend on the history of edit actions, which is incorrect. The problem originates from the fact that it is not possible (in the current implementation) to trace a match extending over more than two consecutive tokens.
Two possible solutions are:

- The boundaries of a word in the input depend on the matches of the preceding words. Changing a word might enable a longer match that starts earlier in the input. The matching should therefore start again, starting at the earliest position in the input that might form a match extending past the changed word. The rematching can stop when a word boundary is found, after the changed word, that existed before the rematching. Since the starting position is not known, the only safe approach is to rematch the entire input. The simplest solution is therefore to implement the invariant —the output is a sequence of matches of the input— as one monolithic function, instead of a fine-grain mechanism as above. For large inputs, this approach is unacceptable.

- The other option is to associate some information with each character, or word, in the input. This information could be a set of states in the DFA, or a pointer to the earliest position in the input that tried to form a match extending past this position, but failed. Both types of extra information imply that the use of the `regexp` library is not satisfactory.

For more on lexical analysis, see [1, ch. 3].

4.3 Parsing strategies

The parser generator should be capable of handling general context-free languages. Although the LALR(1) class is usually large enough to define programming languages, the larger class of context-free grammars is desired for the following reasons:

- Many parser generator systems do not allow certain kind of rules, like left-recursive or epsilon rules. This forces the grammar writer to avoid these constructs, and specify the grammar in a less obvious and elegant way.
- In VIEWS, the grammars are created from the type system. Narrowing the parser generator to a limited class like LALR(k) restricts the possibilities to derive grammars from types, which is undesirable.
- It is not possible to exclude just the ambiguous grammars, as it is undecidable whether a grammar is ambiguous [7]. In practice, one can only ensure that a grammar is non-ambiguous by restricting it to a smaller class of grammars, like LR(k) or LL(k). On the other hand, it is not desirable to exclude the ambiguous grammars, since using an ambiguous grammar can be more convenient to specify some grammatical construct. A well-known example is a grammar for expressions[2]. Here, the ambiguities can be resolved by using operator precedences.

The parsing strategies considered are:

LR parsing LR(k) and LALR(k) parsing algorithms, as used for example by the parser generator YACC [8], are controlled by a parse table that is constructed beforehand by a table generator. The number k is the number of lookahead symbols available to the parser. The complexity of the parse table depends upon k, while the parser's efficiency does not; the parser works in linear time. With conventional LR or LALR table generation algorithms it

[2] Two expression grammars:

ambiguous	unambiguous
Exp → Exp + Exp	Exp → Term
Exp → Exp - Exp	Exp → Exp + Term
Exp → Exp * Exp	Exp → Exp - Term
Exp → Exp / Exp	Term → Factor
Exp → (Exp)	Term → Term * Factor
Exp → Num	Term → Term / Factor
	Factor → (Exp)
	Factor → Num

is difficult to update an already generated parse table incrementally if the grammar is modified. For a detailed description of LR parsing, see [1, ch. 4.7].

Generalized LR parsing This is an extended LR parsing algorithm, see [12], that requires a conventional (but possibly multi-valued) LR parse table. The parser starts as an LR parser, but when it encounters a multi-valued entry in the parse table (conventionally known as a conflict), it splits up into as many parsers working in parallel as there are conflicting possibilities. If two parsers have the same state on top of the stack, they are joined in a single parser with a joined stack. If a parser encounters an error entry in the parse table, it is killed by removing it from the set of active parsers. Using this method, the parser can use as much lookahead as needed to resolve an ambiguity, using simple LR(0) tables. For ambiguous input, all possible parsings are rendered.

IPG An incremental, lazy parser generator for parsing context-free grammars is described in [10]. It is an incremental version of the generalized LR parsing method. Parsing starts with an empty parse table, which is expanded by need during parsing. A change in the grammar is handled incrementally by removing those parts of the parse table that are affected by the change; these parts are recomputed for the modified grammar when the parser needs them again.

The parsing strategy chosen for the VIEWS system is analogous to the IPG.

4.4 The VIEWS parser

Basic to LR parsing, of which the VIEWS parsing method is a derivate, is the notion of an *item*; a production with a dot in its right-hand side, which denotes how much of the rule has been parsed. Items are grouped into sets, which give rise to states in the parser. *Kernel* items are the initial item $Start \rightarrow \bullet X$, where X is the start symbol of the grammar, and all items whose dots are not at the left end. The *closure* of a *kernel* item set is generated by adding, for each item $A \rightarrow \alpha \bullet B\beta$, and each production $B \rightarrow \gamma$, the item $B \rightarrow \bullet\gamma$. In other words, having seen α of A, the next thing expected is a B, of which nothing is seen yet.
The function $transition(I, X)$ for an item set I and a grammar symbol X is the set of all items $A \rightarrow \alpha X \bullet \beta$ such that $A \rightarrow \alpha \bullet X\beta$ is in $closure(I)$. The function $reductions(I)$ for an item set I is the set of

productions $A \rightarrow \alpha$ for which $A \rightarrow \alpha\bullet$ is in *closure*(I). Intuitively, if the parser is in some state with kernel I, it can, recognizing some part of the input as X, move to the state *transition*(I, X), or perform a reduction for any rule in *reductions*(I).

The *parse table*, the set of states for the parser, is constructed by the following algorithm:

- *States* $= \{ \{Start \rightarrow \bullet S\} \}$
- while there is a set I in *States*, and grammar symbol X, for which the state *transition*(I, X) is not empty and not in *States*, add *transition*(I, X) to *States*.

In the parsing process, the parse table is generated by need. It consists of unexpanded states, which have a kernel, and expanded states, which have a kernel, a transition table, and the set of reductions. Initially, it contains the unexpanded state with kernel "$Start \rightarrow \bullet X$", where X is the left-hand-side nonterminal in the first production of the grammar. If the parser needs to know a state's transitions or reductions, the state is expanded, possibly creating new (unexpanded) states to which transitions exist. This lazy generation mechanism is transparent to the parser.

The parser starts in the first state. If the parser is in state S_i and finds symbol X_i, and $S_{i+1} =$ *transition*(S_i, X_i) exists, it shifts X_i and S_{i+1} onto its stack and moves to the state S_{i+1}. If a reduction $A \rightarrow \alpha$ exists, and n is the number of symbols of α, it then pops n symbols and states of the stack, creates a parse tree labeled with A, and the n symbols as children. It then pushes the parse tree and then new state $S_{i+1} =$ *transition*(S_{i-n}, A) onto its stack and moves to S_{i+1}.

There is one *accept* state: If X is the start symbol, then the first state has kernel "$Start \rightarrow \bullet X$", and a transition for X to a new state which has kernel "$Start \rightarrow X\bullet$", and is therefore always the *accept* state.

In some states, more than one action is possible. This is known as a conflict. A LR parser chooses one action, but the VIEWS parser splits into as many parsers as there are conflicting possibilities.

After handling the last token, only the parser in the *accept* state is interesting. It contains the (possibly ambiguous) parse tree for the input text. If no parser is in the *accept* state, then no parse tree can be found; the input does not match the grammar. The operation of the parser is described now in a mathematical formulation, followed by some examples demonstrating its behavior.

4.4.1 The parser's operation

First, some types are defined. The presentations are omitted, since the parsing structures are internal, invisible, objects:

Type	Structure
StackNode	*compound* (*state* : *State*, *backlinks* : *BackLinks*)
BackLinks	*sequence* (*BackLink*)
BackLink	*compound* (*tree* : *ANY*, *node* : *NodeRef*)
NodeRef	*reference* (*StackNode*)
ActiveParsers	*sequence* (*StackNode*)

A stack node is the basic object in the parse stack. It contains a reference to the state the node is in, and links pointing back to the previous stack node. These links are labeled with the parse tree found while moving from the previous state to the current.
Parsing starts in state 0, and reductions (if appropriate) are applied: $NullParsers = ReduceStates[startstate(table(grammar))]$, where the function *ReduceStates* is defined later. The predicate *accept?* determines whether a stack node is in the *accept* state. At the end of the token stream, we find the active parsers:

$$LastParsers = ParseWord \not\rightarrow_{NullParsers} TokenStream$$

and the accepting parsers:

$$AcceptingParsers = accept? \triangleleft LastParsers$$

Since parsers in the same state are joined, *AcceptingParsers* contains at most one stacknode, so the parse trees are found :

$$ParseTrees = \pi_{tree} * (firstAcceptingParsers).backlinks$$

The function *ParseWord* finds all states $\{transition(p,t) | p \in P\}$, applies all available reductions, and joins all nodes which are in the same state:

$ParseWord : ActiveParsers \times Token \rightarrow ActiveParsers$
$ParseWord\, P\, t = JoinNodes(ReduceStates(transition(\square, t) * P))$

The function *JoinNodes* is $InsertNode \not\rightarrow_{[]}$;

$InsertNode : ActiveParsers \times StackNode \rightarrow ActiveParsers$
$InsertNode[x_1, \ldots, x_{n-1}]\, x_n =$

$[x_1, \ldots, x_{n-1}]$	if $x_n = \perp$,
$[x_1, \ldots, x_{i-1}, m, x_{i+1} \ldots, x_{n-1}]$	if $m = MergeNodes(x_i, x_n) \neq \perp$,
$[x_1, \ldots, x_n]$	otherwise.

Two nodes can be merged if they are in the same state, so:

$$\begin{array}{l} \mathit{MergeNodes} : \mathit{StackNode} \times \mathit{StackNode} \to \mathit{StackNode} \\ \mathit{MergeNodes}(x, y) = \\ \quad \begin{array}{ll} (x.\mathit{state}, x.\mathit{backlinks} +\!\!+ y.\mathit{backlinks}) & \text{if } x.\mathit{state} = y.\mathit{state} \\ \bot & \text{otherwise.} \end{array} \end{array}$$

The function *ReduceStates* extends a set of active parsers by adding all parsers that can be derived by applying some reduction of one of the parsers. This is specified by a function *Closure*, parameterized with a function *Derivations*:

$$\begin{array}{l} \mathit{ReduceStates} : \mathit{ActiveParsers} \to \mathit{ActiveParsers} \\ \mathit{ReduceStates} = \mathit{Closure}\ \mathit{Derivations} \end{array}$$

$$\begin{array}{l} \mathit{Closure} : (\alpha \to [\alpha]) \to [\alpha] \to [\alpha] \\ \mathit{Closure}(\mathit{derivations}, X) = \\ \quad \{x_k \mid \forall i \in [1..k] : x_i \in \mathit{derivations}(x_{i-1}), x_0 \in X\} \end{array}$$

$$\begin{array}{l} \mathit{Derivations} : \mathit{StackNode} \to [\mathit{StackNode}] \\ \mathit{Derivations}(s) = \\ \quad \left\{ s' \;\middle|\; \begin{array}{l} A \to \alpha \in \mathit{reductions}(s.\mathit{state}), \\ n = \sharp\alpha, \\ \forall i \in [1..n] : (t_i, \uparrow s_i) \in s_{i-1}.\mathit{backlinks}, \\ s_0 = s, \\ s'.\mathit{state} = \mathit{transition}(s_n.\mathit{state}, A), \\ t' = \mathit{Tree}(A \to \alpha, [t_n, t_{n-1}, \ldots, t_1]), \\ s'.\mathit{backlinks} = [(t', \uparrow s_n)] \end{array} \right\} \end{array}$$

Tree is the function that builds a parse tree from the production and the sequence of subtrees. See section 4.1.3.

4.4.2 Parsing example

According to a simple grammar for assignment statements, the string `x := 3+7` will be parsed. The set of regular expressions used for recognizing the tokens are: { *id*: `[a-z]`$^+$, *num*: `[0-9]`$^+$, `"+"`, `"*"`, `":="` } The output given by the lexical analyzer is "*id*(`x`) `:=` *num*(`3`) `+` *num*(`7`)". The grammar used is:

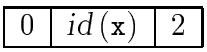
$$\begin{array}{l} \text{Stmt} \to \text{id } \texttt{":="}\ \text{Exp} \\ \text{Exp} \to \text{Exp } \texttt{"+"}\ \text{Exp} \\ \text{Exp} \to \text{Exp } \texttt{"*"}\ \text{Exp} \\ \text{Exp} \to \text{num} \end{array}$$

The fully expanded parse table is:

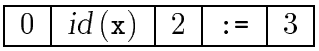
State	Kernel	Transitions		Reductions
0	*Start* → • Stmt	Stmt	1	
		id	2	
1	*Start* → Stmt •			*accept*
2	Stmt → id • := Exp	:=	3	
3	Stmt → id := • Exp	Exp	4	
		num	5	
4	Stmt → id := Exp •			Stmt→ id := Exp
	Exp → Exp • + Exp	+	6	
	Exp → Exp • * Exp	*	7	
5	Exp → num •			Exp → num
6	Exp → Exp + • Exp	Exp	8	
		num	5	
7	Exp → Exp * • Exp	Exp	9	
		num	5	
8	Exp → Exp + Exp •			Exp → Exp + Exp
	Exp → Exp • + Exp	+	6	
	Exp → Exp • * Exp	*	7	
9	Exp → Exp * Exp •			Exp → Exp * Exp
	Exp → Exp • + Exp	+	6	
	Exp → Exp • * Exp	*	7	

Steps in the parser

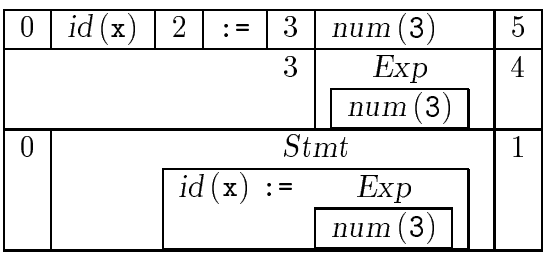
1. Parsing begins in state 0. After the token *id*(x), we reach state 2:



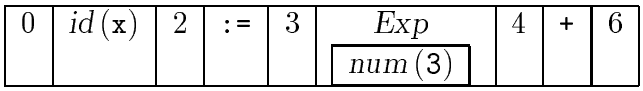
2. After the token :=, we reach state 3:



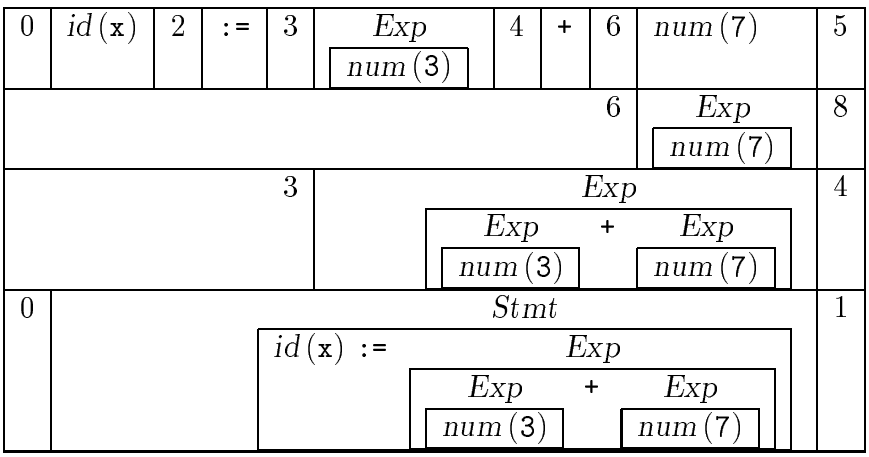
3. After the token *num*(3), we reach state 5. In this state, a reduction *Exp* → *num* is possible, and then *Stmt* → *id* := *Exp*, so we have:



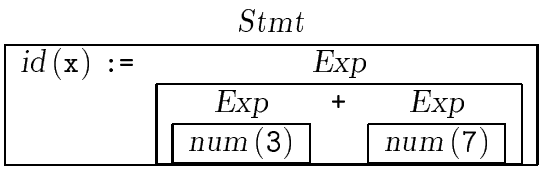
4. Only state 4 has an action for the token +, so the other possibilities are rejected, and we have:



5. After the token *num*(7), we reach state 5, and we can reduce again:



6. At the end of the token sequence, one parser is in the *accept* state, and its parse tree is:

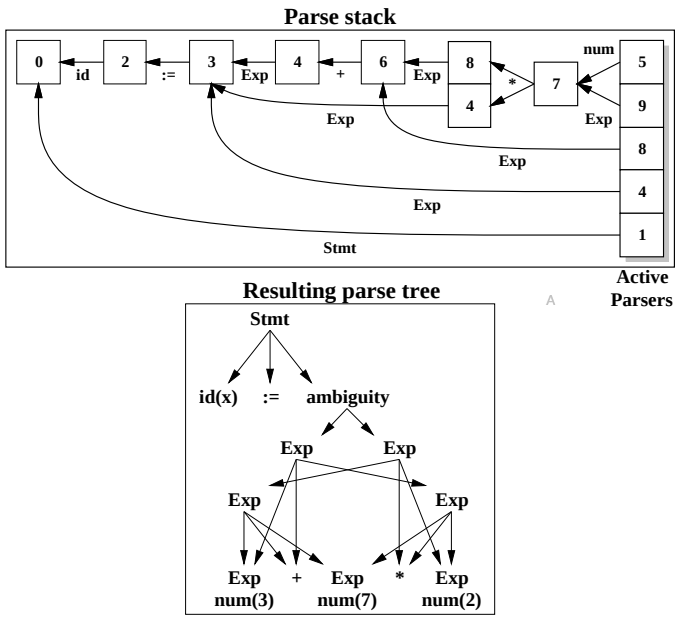


4.4.3 Local ambiguity packing

To demonstrate how the parser deals with ambiguity, the input is now extended to "*id*(x) := *num*(3) + *num*(7) * *num*(2)". The parsers in states 8 and 4 can shift * to state 7, which shifts *num*(2) to state 5. Then, the following reductions can be performed:

- State 5 performs an *Exp* → *num* reduction and moves to state 9.
- State 9 performs an *Exp* → *Exp* * *Exp* reduction, following two paths: one back to state 6, and one back to state 3, so the new parser states are 8 and 4.
- State 8 performs an *Exp* → *Exp* + *Exp* reduction and moves to state 4.
- State 4 performs a *Stmt* → *id* := *Exp* reduction, moving to the *accept* state.

State 4 was reached in two different ways, caused by the ambiguity. The parse stack and tree is:



4.4.4 Lookahead example

A conventional LR(k) parser can use up to k symbols of lookahead to determine what to do. Since the VIEWS parser uses LR(0) tables, this information is not available. The parser is nevertheless capable of making the right choices, simply by trying all options, expecting one of the possibilities to succeed eventually.
In this example, a reduce/reduce conflict is resolved by making both reductions, since no lookahead information is available.

Grammar

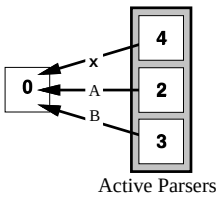
S → A `a`
S → B `b`
A → `x`
B → `x`

Parse table

State	Kernel	Transitions	Reductions
0	*Start* → • S	S 1	
	S → • A `a`	A 2	
	S → • B `b`	B 3	
	A → • `x`	`x` 4	
	B → • `x`		
1	*Start* → S •		*accept*
2	S → A • `a`	`a` 5	
3	S → B • `b`	`b` 6	
4	A → `x` •		A → `x`
	B → `x` •		B → `x`
5	S → A `a` •		S → A `a`
6	S → B `b` •		S → B `b`

If the input is `xa`, the parser takes the following steps:

- Parsing starts in state 0. After the first token, `x`, the parser shifts to state 4, where 2 reductions are available. The set of active parsers is then:



- At the next token, `a`, only the parser in state 2 can shift, and the other parsers are killed. In state 5, a reduce to the *accept* state is possible. The parser has thus been able to postpone the choice in the reduce/reduce conflict until it could be made without using lookahead.

4.4.5 Handling ϵ-productions

If a state's closure contains the item $X \rightarrow \alpha \bullet Y\beta$, then $Y \rightarrow \bullet\gamma$ is also in that closure. If $\gamma = \epsilon$, then the rule $Y \rightarrow \epsilon$ is present in the reductions of that state. If the parser encounters this state, it is possible to perform this reduction.

Example

In the grammar of this example, which describes the language `xb`*, a reduction A $\rightarrow \epsilon$ must be done for every `b` in the input string.

Grammar

S → A S `b`
S → `x`
A → ϵ

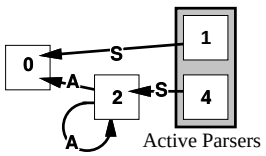
Parse table

State	Kernel	Transitions	Reductions
0	*Start* → • S	S 1	
	S → • A S `b`	A 2	
	S → • `x`	`x` 3	
	A → • ϵ		A→ ϵ
1	*Start* → S •		*accept*
2	S → A • S `b`	S 4	
	S → • A S `b`	A 2	
	S → • `x`	`x` 3	
	A → • ϵ		A→ ϵ
3	S → `x` •		S → `x`
4	S → A S • `b`	`b` 5	
5	S → A S `b` •		S → A S `b`

If the input is `xbbb`, the parser takes the following steps:

- Parsing starts in state 0, and immediately, the first $A \rightarrow \epsilon$ reduction can be made, going to state 2, where another $A \rightarrow \epsilon$ reduction can be done, going to state 2. To avoid a loop, the stack nodes are shared, having a number of links to previous nodes. In this case, the parse forest becomes cyclic.

- After the token `x`, state 0 and state 2 both shift to state 3, where a reduction $S \rightarrow$ `x` is possible, bringing state 0 to the *accept* state and state 2 to state 4, where it is ready to shift another `b`.



- After every `b` that follows, the parser in state 1 dies, but the parser in state 4 shifts to state 5, and reduces $S \rightarrow A\,S$ `b`, splitting into two parsers: one in the *accept* state, and one in state 4. The parser can therefore perform just as many $A \rightarrow \epsilon$ reductions as needed, by using the cyclic parse stack.

4.4.6 Error recovery

The parser should not only respond correctly to syntactically correct inputs, but also try and interpret incorrect inputs, which are very likely to occur as intermediate states in the editing. Simply complaining that the parser's input is incorrect and disposing of the parse tree is unwise. A few general error recovery strategies used in compilers are:

Panic-mode This is the simplest method to implement. On discovering an error, the parser discards input tokens until one of a designated set of synchronizing tokens is found. The synchronizing tokens are usually delimiters whose role in the program is clear, such as a semicolon or the keyword `end` for a PASCAL parser. It is however hard, if not impossible, to automatically select these delimiters from the automatically generated grammars.

Phrase-level recovery On discovering an error, the parser may perform local correction on the input. Typical local corrections are replacing a comma by a semicolon, or inserting a token that enables the parser to continue. Its major drawback is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection.

Error productions If a good idea of the common errors that might be encountered can be obtained, the grammar can be augmented with productions that generate the erroneous construct. If the parser then needs to use an error production, an appropriate error message can be generated. Since it is very likely that the input is in an intermediate state, it could be sensible for a production $A \rightarrow XYZ$ to create the error productions $A \rightarrow YZ$, $A \rightarrow XZ$ and $A \rightarrow XY$. The size of the augmented grammar would then become extremely large.

The approach chosen is a combination of the first two strategies: If none of the active parsers can shift:

- If some parser has a transition for some symbol S to a new state, and *ParseWord*(*Parsers,S*) has a transition for the token, it may seem logical to presume that S is missing, and insert a placeholder for it in the input.
- If that fails, try a replacement: if some parser has a transition for some symbol S to a new state, and the new state can shift the *next* token, replace the token by a placeholder for S.
- If that fails, flag the token as erroneous and skip it; that means passing the active parsers to the next step.

4.4.7 Locality of changes

The implementation of the parser should be highly incremental; minor changes should require minimal updates. This is achieved by specifying the parser as a left-reduce that breaks the parser into the set of invariants $r_i = \mathit{ParseWord}\ r_{i-1}\ a_i$, where r_{i-1} is a set of parsers trying to shift token a_i, forming r_i. Suppose the *old* text is changed to *new*, where both can be derived from some nonterminal β:

$$\begin{array}{lll} \mathit{Start} & \Rightarrow^* & \alpha\beta\gamma \\ \beta & \Rightarrow^* & \mathit{old} \\ \beta & \Rightarrow^* & \mathit{new} \end{array}$$

The changes are then local to β, so only the part of the parse tree recognizing β needs to be recalculated. After the last token in the derivation of β, the parser is in the state $A \rightarrow \alpha\beta \bullet \gamma$, whether β derived *old* or *new*. For example: if a new token is inserted into the tokenstream, then the invariant $r_i = \mathit{ParseWord}\ r_{i-1}\ a_i$ is replaced by $r_i = \mathit{ParseWord}\ r_{new}\ a_i$ and $r_{new} = \mathit{ParseWord}\ r_{i-1}\ a_{new}$. r_i was in a state recognizing a sequence, and will be the same now, so the changes do not propagate along the left-reduce chain.

4.5 Parsing structures

Although the preceding sections have assumed a linear (textual) input, the approach does not exclude parsing structures. If an object (of type t_1) is copied, and then pasted into a placeholder for type t_2, it is not desirable to reject that action if t_1 is not identical to t_2. Rather, the pasted object should be cast to type t_2. This type casting can be done by parsing the object to the grammar of the required type. If the types are identical, there is a transformation for t_1 in the start state of the grammar for t_2 to the *accept* state. If $t_1 \neq t_2$ and the object is a composite object, its children can be parsed to form an object of type t_2.
The function $ParseWord(P, t)$ as defined in section 4.4.1 (page 35) is extended in the following way:

$ParseWord : ActiveParsers \times Token \rightarrow ActiveParsers$
$ParseWord\ P\ t =$
    $JoinNodes(ReduceStates(transition(\square, t) * P))$
        if t is a token
    $JoinNodes(ReduceStates(T))$
        if t is an object of type τ,
        $T = transition(\square, \tau) * P)) \neq \bot$
    $ParseWord \not\rightarrow_P t$
        otherwise

4.5.1 Parsing graphical objects

If the structure is maintained in the edit level, presentations that are non-textual, like *circled*(*something*), can then be parsed. A type with a graphical presentation, for example Type $T = S$ presented *circled*(p), could have the production $T \rightarrow$ *circle build-rhs*(p, S) (see section 4.1.2). If the contents of the circle is updated, the circle does not have to be parsed, since it is known to surround its contents. Yet the circle and contents together can be parsed if they are of the correct type. This introduces grammars for which (some) sentential forms cannot derive sentences. A sentence is a sequence of symbols from the grammar's alphabet that can be derived from the start symbol. This is not a problem, since the edit level contains the structure of the presentation, and can therefore offer the sentential form to the parser.

4.5.2 Parsing combined presentations

In the same manner, it is not a problem that objects are presented on top of each other, since in the (structured) presentation, the elements

are distinguishable. The only problem is that the order is missing. The problem is therefore to match a bag of items to a bag of patterns. A brute-force approach would be to try and match every permutation of the items with patterns, which costs $\mathcal{O}(n!)$.

Chapter 5

Conclusions

5.1 General observations

Because the parser is defined in a strictly functional manner, implementing the basic functions it is composed of in an incremental way should guarantee that the parser itself is also fully incremental without any special care.
The parser generator should be incremental, since the presentations may change on the fly, but the demands are not as high as in the case of the ASF+SDF programming environment development system, where a user is editing the grammar rather often. The incremental parser generator IPG [10] of which the main ideas for implementing the VIEWS parser originate, was developed for that system. A less efficient parser generator could therefore suffice, but the generated parser itself should be very efficient and highly incremental. The efficiency of the parser and the parser generated have not been tested.

5.2 Current Situation

Structure watchers It should be possible to apply structure watchers not just to sequences, but to any structured object. Note that insertions and deletes may not be applicable to compounds and choices, but replacing a child by another can be interpreted as a delete followed by an insert.
More important, it is possible to completely overwrite the structured object by some other object, for example in a paste operation. This too can be regarded as a series of deletes and insertions, but it is also possible to regard the paste as one single operation. This is the case in the current implementation of the paste operation, and therefore, the structure watcher, and all applications that rely on it, fail to handle a

paste operation on the entire object correctly. This is a bug, which can be solved by either changing the paste operation to perform a series of deletes and insertions, or by extending the update-functions object with a third callback-function for paste operations.
The update-functions object should contain entries, or methods, for elementary operations on the structure of objects. The decision which operations are considered elementary may depend on the type of the object a structure watcher is attached to.

Incremental list operators Apart from the paste-bug mentioned above, the implementation of the sorting program and the map, reduce and filter operations is fairly good. For easy programming using lists, more list operators should be implemented in the kernel, for example $x = \mathit{first}[x_1, \ldots]$, $L = x : [x_1, \ldots]$ which is just the inverse, $L = X +\!\!+ Y$, $L = \mathit{merge}(X, Y)$, and some predicates, for example $\mathit{in} = x \in L$.

Implementation of the parser The lexical analyzer works, but contains a bug as explained in section 4.2. The parser generator is implemented and seems to work reasonable well. An extremely fragile prototype of the parser was implemented. The mapping of types to grammars has not been implemented yet.

Bibliography

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

[2] R.S. Bird. An introduction to the theory of lists. *Logic of Programming and Calculi of Discrete Design.* Springer-Verlag, (1987) 3–42.

[3] R.S. Bird and P.L. Wadler. *Introduction to Functional Programming.* Prentice Hall International, 1988.

[4] E. Boeve. *PhD thesis,* draft, 1992.

[5] J. Early. An efficient context-free parsing algorithm. *Communications of the ACM,* 13(2): 94–102, 1970.

[6] A.J. Field and P.G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[7] M.A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, 1978.

[8] S.C. Johnson. *YACC: yet another compiler-compiler.* Bell Laboratories, 1986. UNIX Programmer's Supplementary Documents, Volume 1 (PS1).

[9] S. Pemberton. *The Views Application Environment.* CWI Report CS-R9257, CWI, Amsterdam, December 1992.

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

[11] T.W. Reps and T. Teitelbaum. *The Synthesizer Generator. A system for constructing language based editors.* Springer Verlag, New York, 1989.

[12] M. Tomita. *Efficient Parsing for Natural Languages.* Kluwer Academic Publishers, 1985.