Incremental updates in structured documents

Greger Lindén

Licentiate Thesis
February 5, 1994
Department of Computer Science
University of Helsinki
Acknowledgements

I am deeply grateful to my advisor Professor Heikki Mannila for his expert guidance and unfailing support during the course of this work. I also wish to thank Associate Professor Jukka Paakki for his many valuable comments on my work. I thank Professor Martti Tienari, head of the Department of Computer Science, and my colleagues at the department, for providing excellent working conditions. The financial support of the Academy of Finland and of the Technology Development Centre (TEKES) is greatly appreciated.

Finally, I wish to thank all who have enriched my life during studies and research.
Contents

1 Introduction .................................................. 1
  1.1 Related work ........................................... 5
  1.2 Outline of the thesis .................................. 7

2 Lazy and incremental methods ......................... 9
  2.1 Incrementality ......................................... 9
  2.2 Laziness ................................................ 12
  2.3 Discussion ............................................. 13

3 Structured text and the HST system .................. 15
  3.1 Grammars for structure description .................. 15
  3.2 Views of documents .................................. 17
  3.3 The document preparation model of the HST system .... 19
  3.4 Lazy transformation ................................... 21
  3.5 Incremental updates .................................. 22
  3.6 Combining incrementality with laziness .............. 24

4 Grammars, parsing and tree distances ............... 26
  4.1 Context-free grammars ................................ 26
  4.2 Parse trees ............................................ 28
  4.3 Some further definitions and notations ................ 29
  4.4 Recursive descent parsing .............................. 31
  4.5 The LL(1) conditions .................................. 32
  4.6 Grammars with variable strings ....................... 33
  4.7 The LL(1) condition for textable grammars .......... 35
  4.8 Syntax-directed translations ........................... 36
  4.9 Edit distances between trees ........................ 41
5 Incremental parsing

5.1 Related work .................................................. 54
5.2 Incremental Parsing ............................................ 55
5.3 Data structure support ........................................ 56
5.4 Start position .................................................. 59
5.5 Termination conditions ....................................... 60
5.6 Incremental parsing: the method ......................... 65
5.7 Iterations ...................................................... 75
5.8 Error recovery ................................................ 77
5.9 Complexity .................................................... 77
5.10 Improving the incremental parser ....................... 79
5.11 An example of incremental parsing ...................... 81

6 Incremental syntax-directed translation ................. 86

6.1 Incremental translations ................................... 86
6.2 Data structure support ..................................... 88
6.3 Termination conditions .................................... 89
6.4 Iterations .................................................... 92
6.5 A naive algorithm .......................................... 93
6.6 An enhanced algorithm ................................... 95
6.7 An example of incremental tree translation .......... 98

7 Lazy syntax-directed translation .......................... 103

7.1 A lazy view grammar ....................................... 103
7.2 Iterations .................................................... 105
7.3 Continuous transformation ................................ 106
7.4 Combining lazy translation and incremental updates . 108
7.5 Examples of lazy transformation ......................... 110

8 The HST approach and implementation .................. 116

8.1 The HST system .............................................. 116
8.2 The incremental parser ................................... 118
8.3 Test results .................................................. 118
8.4 The lazy translator ......................................... 120
8.5 Incremental translation ................................... 120
8.6 Discussion .................................................. 120
8.7 Future work ................................................ 121
9  Conclusion  122
References  125
Index  139
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Document preparation model of HST.</td>
<td>4</td>
</tr>
<tr>
<td>3.1</td>
<td>A document tree.</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>An annotated parse tree.</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>Different representations of a document in the HST.</td>
<td>20</td>
</tr>
<tr>
<td>3.4</td>
<td>A document with only a part of the logical representation transformed.</td>
<td>21</td>
</tr>
<tr>
<td>3.5</td>
<td>An incremental update in a document with several views.</td>
<td>23</td>
</tr>
<tr>
<td>3.6</td>
<td>A lazily transformed document and its update.</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>A tree and its translation.</td>
<td>40</td>
</tr>
<tr>
<td>4.2</td>
<td>Two labeled trees with minimum edit distance 1.</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>Two labeled trees with ordered nodes.</td>
<td>46</td>
</tr>
<tr>
<td>5.1</td>
<td>A scenario for incremental parsing.</td>
<td>56</td>
</tr>
<tr>
<td>5.2</td>
<td>A parse tree and its corresponding token list.</td>
<td>61</td>
</tr>
<tr>
<td>5.3</td>
<td>A parse tree with with one token not synchronized with the input token list.</td>
<td>62</td>
</tr>
<tr>
<td>5.4</td>
<td>A parse tree and two modifications.</td>
<td>63</td>
</tr>
<tr>
<td>5.5</td>
<td>A parse tree and its modification.</td>
<td>65</td>
</tr>
<tr>
<td>5.6</td>
<td>The incremental parser: main parse functions.</td>
<td>67</td>
</tr>
<tr>
<td>5.7</td>
<td>An instance of a parse tree and the corresponding editor tokens.</td>
<td>82</td>
</tr>
<tr>
<td>5.8</td>
<td>A parse tree and modified editor tokens.</td>
<td>83</td>
</tr>
<tr>
<td>5.9</td>
<td>A partly updated parse tree and corresponding editor tokens.</td>
<td>84</td>
</tr>
<tr>
<td>5.10</td>
<td>An updated parse tree and corresponding editor tokens.</td>
<td>84</td>
</tr>
<tr>
<td>6.1</td>
<td>A scenario for incremental syntax-directed translation.</td>
<td>87</td>
</tr>
<tr>
<td>6.2</td>
<td>Two corresponding trees of a translation form.</td>
<td>90</td>
</tr>
</tbody>
</table>
## List of Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-down parsing of a nonterminal</td>
<td>31</td>
</tr>
<tr>
<td>Syntax-directed translation of a tree</td>
<td>38</td>
</tr>
<tr>
<td>Preorder transformation</td>
<td>49</td>
</tr>
<tr>
<td>Parse transformation</td>
<td>53</td>
</tr>
<tr>
<td>Incremental update traverse</td>
<td>69</td>
</tr>
<tr>
<td>Incremental top-down parsing of nonterminal</td>
<td>70</td>
</tr>
<tr>
<td>Incremental top-down parsing of constant string</td>
<td>72</td>
</tr>
<tr>
<td>Incremental top-down parsing of variable string</td>
<td>73</td>
</tr>
<tr>
<td>Incremental top-down parsing of empty string</td>
<td>74</td>
</tr>
<tr>
<td>Naive incremental syntax-directed translation</td>
<td>94</td>
</tr>
<tr>
<td>Enhanced incremental syntax-directed translation</td>
<td>96</td>
</tr>
<tr>
<td>Lazy translation</td>
<td>107</td>
</tr>
<tr>
<td>Lazy incremental document preparation</td>
<td>109</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Document preparation has been an increasingly important application of computers for over thirty years. The first systems were only extensions to program editing tools, low level formatters used to construct mainly lines of equal length, and to produce justified pages. Today we see a large number of different systems, processing, among other things, multimedia, structured documents, and hypertext.

What is actually a document? Most dictionaries define it as an official record, e.g., the Collins Cobuild English Language Dictionary defines a document as “one or more pieces of paper with writing on them which provide an official record or official evidence about something” [Sin92]. We shall use the term in a somewhat broader sense. We see a document as a paper, memo, note, computer program, or any written material (electronic or on paper) that can be read or modified. A document may also be circularly defined as the product of a document preparation system [AFQ89a].

Document preparation systems can be classified into batch-oriented systems and interactive systems. Batch-oriented systems such as T\textsc{e}X [Knu87], \textsc{bte}X [Lam86] and \textsc{Scribe} [Rei80] transform an input file of text and markup commands into a formatted output file. This process takes place without the interaction of the user, and the effect of a local change can be seen only by reformating the entire document. Interactive systems, on the other hand, like Grif [QV86] and Microsoft’s \textsc{Word} [Mic88] permit the author to view and edit the formatted text directly. The effect of changes in the document are immediately shown. These systems are often called WYSIWYG (What you see is what you get). Interactive systems combine the editor and the for-
matter into a single system, so that users can interact with both components without changing environments [CKS+81].

Document preparation systems can also be classified according to their procédurality. This classification is based on the level of the commands by which the user communicates with the system. In a procedural system, the user controls formatting by low level commands, such as “skip two lines”, or “use Roman style”. The formatter follows the instructions by the user without any understanding of the reason behind the commands. On the other hand, a declarative system operates on a higher level. The user marks his document with tags that identify parts of the document such as footnotes, sections, section titles etc. Each tag is then interpreted by the system which produces the correct format for the tagged object. Examples of declarative systems are Scribe [Rei80] and BT\TeX [Lam86]. For surveys on document preparation systems, see [FSS82, And86, Fur92].

Recent document preparation systems provide several views of a document [Bro91, QV86, CH88]. Special research effort has been put into two-view editors that show a textual (declarative) view and a formatted view of a document simultaneously. The idea is that the user can edit any view; changes are propagated to the other view immediately. Propagation demands that processing is fast and efficient. One approach of achieving this is to use incremental updates. An incremental update performs minimal update work after a modification.

Maintaining several views of a document usually relies on the structure of the document. Some systems give the user the opportunity to define a general structure of the document. The simplest document model represents a document as a sequence of characters. A more advanced model decomposes the document into logical parts. In a structured document the logical parts of the document are organized hierarchically. For example, a structured document can consist of a title and several sections. The sections can consist of subsections or paragraphs. Examples of structured documents are reports, dictionaries, and manuals.

In this thesis, we present incremental techniques for updating structured documents. The Helsinki Structured Text Database System (HST) [KLMN90] is an environment for creating, modifying and querying structured documents. In a structured document, there are usually logical parts that we like to distinguish from formatting details. In HST, the user can define the hierarchical structure of a generic document. We call this defini-
tion the logical definition of a (generic) document. Documents or instances over this generic definition are called logical documents. The logical definition defines the logical parts of a document such as title, list of sections, and bibliography, but it does not deal with formatting details such as newlines, blank spaces, and fonts.

In HST, a document over the generic definition can be presented in different formatted versions called views. For every different formatted version of a document there is a separate view definition. A view definition corresponds closely to a logical definition. Most logical parts of a document given in a logical definition are usually enclosed also in the view definition. But a view definition can remove some logical parts, and reorder other parts. E.g., a view definition can state that a view of a document should contain section titles only.

In order to present a logical document in some formatted version, we define a transformation from the logical document to a view. For every different view of the document, we need a separate transformation. We also define the inverse transformation. With an inverse transformation we can extract the logical document from a view.

When the user creates a document, he follows a certain view definition. On demand, the HST system extracts the logical structure by an inverse transformation. If the user wants to see different formatted versions of the document, he defines additional views, i.e., transformations. The system automatically transforms the logical document and presents these views. This we call the document preparation model of HST (Figure 1.1).

Updates in one view are propagated first to the corresponding logical document and then to other views. The propagation of a modification in a view of a document can take considerable time. First the modified logical structure must be extracted from the view. This computation reconstructs the entire logical document. The next step is to update other views by transforming the logical document and reconstruct the views.

In this thesis, we study modifications on the document and the problem of how to efficiently update the different representations due to these modifications. The suggested solution is based on incremental algorithms. An incremental algorithm updates an old representation instead of reconstructing the entire representation. Updating due to a modification in a view starts with incrementally extracting the logical structure, i.e., we update the old logical document. If other views of the document exist, they must also be
updated. Therefore, the logical document is *incrementally transformed* to other views. The principal idea is to minimize the amount of update work and thereby make the system more efficient and interactive.

We also study lazy techniques for producing a part of a view or a *lazy view* of a document. A lazy view shows only a part of a formatted document. There are several reasons for using a lazy view. First, the user can only work on a part of a document view at a time. Therefore it suffices to transform a part of a logical document. Second, transforming a large document entirely is time consuming. *Lazy transformation* produces a part of a view only on demand. For example, when the user wants to see a document, only the first part of it is transformed into a view. The amount that is transformed can depend on the size of the editor where the view is presented, or on the size of the window in which the view is shown. The principal idea here is to postpone as much work as possible until the user actually demands a transformation.

It should be noted that the HST system is *not* a formatter producing high quality paper output. The system is mainly used for producing different views of a document on the screen. It can well be used as a preprocessor to *\LaTeX* [Lam86] or some other formatting system. The user can define *\LaTeX* views of a document and prepare documents to be printed by *\LaTeX*.

It should also be noted that the logical definition of a document is given

![Diagram](image)

**Figure 1.1:** Document preparation model of HST.
as a context-free grammar. Preparing a document will therefore include both
parsing and syntax-directed translation. Consequently, incremental updates
will include incremental parsing and incremental syntax-directed translation.
We present these concepts in Sections 3, 5 and 6.

1.1 Related work

As mentioned, there exist several different sorts of document preparation
systems. We shall here give a short description of systems related to the
HST system [KLMN90]. These systems usually present to the user two or
more views of a document.

The Sam system [Tri81] was one of the first two-view editors for graphi-
cal pictures. It combines graphics and a layout language; the user can edit
a picture in two views. The program view displays a textual description of
graphical objects in a declarative layout language. The layout view is a pic-
ture of the graphical objects themselves. The two views are connected by
an internal representation, a parse tree, over the layout language. A syntax-
directed editor in the program view prevents the user from introducing in-
consistencies. Only modifications consistent with the syntax of the language
are allowed. All modifications are reflected in the internal structure incre-
mentally, but the incremental updates are greatly simplified thanks to the
restricted modifications in the program view.

Janus [CKS+81, CBG+82] was the first two-view text processing system.
It provides the user with two views of a document on two different screens.
The markup view shows the document marked with SGML tags [Bar89]. The
formatted view shows a page of the formatted document. Only the markup
view is editable, except for some simple layout editing in the formatted view.
Modifications in the markup view are updated by reformatting the corre-
spounding output view. Modifications can have far-reaching effects due to
forward and backward references and therefore the reformatting of the entire
document is inefficient. Because reformatting is not done incrementally, the
user can instead edit a document in different modes. In fast mode, the data in
both views is locally consistent, but there might be errors in, e.g., pagination.
Safe mode ensures that everything is correct except for forward references.
This can be viewed as the best single-pass formatting. Perfect mode ensures
that both views are consistent. In general, this can be achieved only by
multipass formatting. When the internal structure changes both views are updated.

**Juno** [Nel85] integrates a language describing graphical pictures with a WYSIWYG image editor. In the textual view, pictures are described with procedures using a declarative language based on points and constraints. In the graphical view the user can create, select, and move points in addition to expressing various constraints on the points. The textual view is not updated on every operation, but new procedures are added when the user so requests.

**VORTEX** [CCH86, Che88, CH88, CHM88] is a TeX-based [Knu87] document preparation system. Both source and target representations of TeX documents are maintained and shown. Changes to the source representation are automatically propagated to the target representation. The internal structure of the document is represented as a tree, combining both the source representation and the target representation and some auxiliary information. The leaves of the subtrees in the source representation form the actual contents of the source file. The target representation has TeX boxes as leaves. Corresponding nodes in the source and target representations are linked. When the source is modified, the corresponding changes are made in the target representation incrementally. The two views are redisplayed after each modification.

The Grif system [QV86, QVB86a, QVB86b, FQA88] is an interactive system for editing structured documents. It is a structure-oriented editor which guides the user in accordance with the structure of the document. The user can define new document structures. The presentation of a document is defined as a presentation schema. A document can have several presentation schemas, defining different views of the document. The default view shows the entire document. Other views are, e.g., a table of contents and the list of all formulas defined in the document. Grif is not a formatter even if the presented views are almost WYSIWYG. Instead, Grif can be used as a preprocessor to TeX and \LaTeX{} [QVB86a]. A presentation schema defines some conversion rules that transform a document into TeX or \LaTeX. Especially the structure is here taken into account. The transformation cannot only remove certain parts from a document, it can also reorder elements in the document. The internal structure of Grif is a tree where the logical document is saved. A modification in a view propagates through the tree. Grif knows what constraints exist between the modified part and its neighbors, and updates can be done incrementally [QV87].
Tweedle [Ase87] lets the user edit graphical objects using a procedural language called Dum. The internal data is represented as a parse tree. When the user makes changes to the textual representation, Tweedle must decide how to change the graphical picture. Reparsing is done from scratch, and the old and the new parse tree are compared to determine where the changes occur in the graphical picture. This technique is claimed to work acceptably well in practice. The textual representation is reexecuted to form the new picture. Often reexecution means execution from scratch, but in some cases it can be incremental.

Lilac [Bro88, Bro91] offers both WYSIWYG editing and language-based description side by side. First, the user defines the logical structure of the document. Then he can edit a document in either the language description view or the WYSIWYG view. Modifications in one view are immediately reflected in the other view. An incremental interpreter keeps the two views up to date.

Quill [CHL+88, CHP88, Cha88, Lun88, Cha90] is a WYSIWYG system designed to support full integration of various sorts of graphics editing together with text editing. The logical document is described with SGML tags [Bar89]. The document is organized as a hierarchy of elements of mixed types. The system is organized as a collection of cooperating editors, one for each type of material in the document. The underlying data structure is a document tree. The logical document is incrementally updated when a modification is made in the (WYSIWYG) document. The rest of the WYSIWYG document is reformatted in the background while the user continues to modify the current view.

A related area is incremental program generation consisting of incremental parsing and incremental compilation [Rei84, Fri88]. We review some related work later in Section 2 on incrementality and laziness in general and in Section 5 on incremental parsing.

1.2 Outline of the thesis

This thesis is organized as follows. In Section 2 we present lazy and incremental techniques and give their connection to document preparation. In Section 3 we give an introduction to structured texts and the HST system [KLMN90], which is an example of a document preparation system that han-
dles structured texts. We also informally study the goals of this thesis. In Section 4 we present some concepts that we need for explaining the incremental techniques. We give an overview of grammars and parsing, and follow up with defining syntax-directed translations and tree-to-tree corrections. In Sections 5 and 6 we study two cases of incremental tree updates in particular: incremental parsing and incremental syntax-directed translation. In Section 7 we define a model of lazy transformation. Section 8 discusses some implementation details of the HST system and how incrementality and laziness can be merged into the existing system. Finally, we give a summary of the thesis in Section 9.
Chapter 2

Lazy and incremental methods

Traditionally, computable problems have been solved through what could be called a one-shot paradigm: An algorithm computes a complete solution to a problem instance each time it is activated. However, an incremental paradigm is a more natural characteristic for a number of problems. An incremental algorithm updates an old solution instead of recomputing the solution entirely. Moreover, some problems are instances of a lazy paradigm where a computation is not made until it is absolutely needed. In the following we give a brief overview of incrementality and laziness and refer to some research in the area.

2.1 Incrementality

In the one-shot paradigm, the solution of a modified problem instance is completely recomputed. Many problems, however, are incremental in their nature. An incremental algorithm updates an existing solution in response to a change in the problem instance. Typical applications include program development (parsing, compiling), text processing (formatting), window management, and maintenance of consistent information (spreadsheets). There is a distinction between incremental analysis and on-line analysis. The former studies real time as a function of the size of a change in the underlying data structures, whereas the latter studies real time as a function of the size of the data structures themselves and the number of requests made \cite{AHR88}.

Assume that an incremental algorithm has solved a problem instance
and denote by \( S \) the solution to \( P \). Assume \( P \) is modified to \( P' \) by modification \( \Delta P \). The task of the incremental algorithm is to produce \( \Delta S \), an update to the old solution \( S \).

Batch-mode computations usually study the complexity of algorithms as a function of the size of the entire input. The sum of the sizes of the changes in the problem instance and in the solution is a better parameter when evaluating the time complexity of an incremental algorithm. Denote by \( \text{CHANGED} \) the sum \( |\Delta P| + |\Delta S| \). Note that \( \text{CHANGED} \) only characterizes the amount of work absolutely necessary to perform a given incremental problem. It does not take into account updating costs of various internal data structures used by the particular incremental algorithm. \( \text{CHANGED} \) is also not known a priori; when the update process begins, only \( |\Delta P| \) is known [RR91].

An incremental algorithm is \textit{bounded} if the computation of the solution update takes time dependent only on \( \text{CHANGED} \), and not on the size of the entire input. Otherwise, an incremental algorithm is \textit{unbounded}. A problem is said to be incremental if it has a bounded algorithm. Boundedness is not the only relevant criterion in the study of incremental computation, and in fact relatively few bounded incremental algorithms are known [RR91].

Some examples of research on incrementality are the following. Ramalingam and Reps [RR91] have studied incrementality in general and they order known incremental algorithms into a complexity hierarchy. They emphasize that the sum of the sizes of the changes in the input and in the output is a better parameter than the size of the entire input when evaluating complexities. Berman et al. [BPR90] present a general method for proving lower bounds of incremental algorithms. As mentioned, there are several application areas for incremental algorithms. Incremental text processing, as in incremental syntax-directed editing [Rep82, RT89] and incremental formatting [CHM88], has received some attention. A closely related area is incremental program generation (parsing and compiling) [Fri82, Fri88], incremental semantic analysis [Hed92], and incremental program execution [KW87, Bha87, HM85]. Incremental attribute evaluation has been studied both \textit{per se} [DRT81, Rep84, Kap87, Alb91] and as part of applications [Hud86, HK88]. Finally, incremental algorithms have been applied on user interfaces [Hol87, Hol88] and data flow analysis [Ryd83, Zad89].
**Example 1** We shall give an example of incremental update in a constraint system. Assume that we have the following equations

\[
\begin{align*}
    a &= 1 \\
    b &= 2 \\
    c &= a + b \\
    d &= c + 1 \\
    e &= d + 1 \\
    f &= b + 4
\end{align*}
\]

The system consists of six equations. Dependence between equations is important. In this system we say that an equation \( eq \) depends on an equation \( eq_0 \) if the left hand side of equation \( eq_0 \) is a part of the right hand side of \( eq \). The dependencies can be shown in the following way:

\[
\begin{array}{c}
    a = 1 \\
    \downarrow \\
    \quad \quad c = a + b \\
    \uparrow \\
    b = 2 \\
    \downarrow \\
    e = c + 2 \\
    \downarrow \\
    f = b + 4
\end{array}
\]

In the figure, a dependency is denoted by an expression \( eq_0 \rightarrow eq \) with the meaning that equation \( eq \) depends on equation \( eq_0 \). The system is consistent when every variable on the left hand side of an equation has the value computed by the right hand side of the same equation. This system is consistent when we have the following values for the variables: \( a = 1, b = 2, c = 3, d = 4, e = 5, \) and \( f = 6 \).

Assume that we change the value of variable \( a \) to 2. A reevaluation from scratch would reevaluate all variables, including \( b \) and \( f \) even if they do not depend on \( a \). An incremental evaluation would only reevaluate variables that are changed, i.e., the variables \( c, d, \) and \( e \). The values that would make the system consistent are \( a = 2, b = 2, c = 4, d = 5, e = 6, \) and \( f = 6 \).

Now, assume that we swap the values of \( a \) and \( b \) in the original system, i.e., we set \( a = 2 \) and \( b = 1 \). A reevaluation from scratch would again reevaluate all variables. An incremental evaluator would reevaluate \( f \) and \( c \). After reevaluating \( c \), the evaluator would notice that the value of \( c \) does not change and therefore reevaluation of variables depending only on the equation \( c = a + b \) would be unnecessary. The values that would make the system consistent are \( a = 2, b = 1, c = 3, d = 4, e = 5, \) and \( f = 5 \). □
2.2 Laziness

*Laziness* is a concept related to incrementality. Traditionally, the one-shot paradigm computes the entire solution of a problem instance. Again, many problems benefit from a lazy paradigm. A lazy algorithm postpones the computation as much as possible. There can be several reasons not to perform a computation completely. Some part of the computation may be totally irrelevant or not needed immediately, or some results are not visible to the user and therefore redundant. Applications include user interface management, database management, and maintenance of consistent information as in a constraint restriction system.

A *lazy algorithm* solves only a part of a problem. Let the problem instance $P$ consist of some possibly overlapping instance parts. A lazy algorithm solves a sub-problem by giving a *part of the solution*. There is a fundamental distinction between incrementality and laziness. The incremental algorithm updates the complete solution after a modification in the problem instance; the lazy algorithm solves different parts of the problem instance. The lazy algorithm must also be told which part of the instance to solve.

Lazy methods are used, e.g., in program generation [HKR87, HKR91]. Traditional program generators usually require that a program must be generated in its entirety before it can be used. If generation time is scarce, it may be better to generate only those parts that are indispensable for processing some particular data. A related application is lazy generation of scanners and parsers [HKr89, HKR92, Kos90]. A lazy scanner postpones the construction of parts of an automaton for recognizing certain textual patterns until the parts are needed. Similarly, a lazy parser generates a parse table during parsing, but only those parts that are needed to parse the input sentence at hand.

User interfaces are another application area [Hud86, HK88, Hud91]. Various components of the user interface are produced only when they are directly or indirectly observed by the user. The UIMS user interface management system [HK88] supports direct manipulation. It provides semantic feedback with a graphical representation and automatic screen update. The underlying data model is an attributed graph which is incrementally updated when a change occurs in the user interface. The update consists of two phases, a *mark phase* that traverses dependencies in the graph and marks nodes out of date, and a *reevaluation phase* that only evaluates nodes that are out of
date and important. A node is *important* if it has an immediate effect on the system. Only nodes that need to be reevaluated are evaluated. Hudson also suggests a way of having different views of the data in an attributed graph [Hud89, Hud91]. For each node there is an attribute $V$ which computes the view node. If a subview is missing, i.e., it is not visible, it is replaced by dots. Views are constructed first top down (what is visible) and then bottom up (subviews). The user interface is updated lazily, i.e., only attributes that affect what the user currently sees are updated.

**Example 2** Let us study a small example of lazy update in the previous constraint system of Example 1. We have the following dependencies.

\[
\begin{align*}
  a &= 1 \\
  b &= 2 \\
  c &= a + b \\
  d &= c + 1 \\
  \quad e &= c + 2 \\
  f &= b + 4
\end{align*}
\]

Assume that the user is only interested in the variable $e$. There could be several reasons for this. For example, the variable $e$ could be the only one whose value is currently shown to the user, or it could be the only one that affects some system that the user is working with.

Assume that we change the value of the variable $a$ to 2. A reevaluation from scratch would reevaluate all variables. An incremental evaluation would only reevaluate the variables $c$, $d$, and $e$. A lazy evaluation would reevaluate only variables $c$ and $e$. The value of $d$ would be inconsistent with its function; $d$ would be updated later when the user wants the value of $d$. The values of the variables after a lazy reevaluation would be: $a = 2$, $b = 2$, $c = 4$, $d = 4$, $e = 6$, $f = 6$.

\[\square\]

### 2.3 Discussion

We saw in Section 1 that there exists a lot of document preparation systems that use incremental updates. Incrementality in document preparation has been considered both important and necessary. Especially in interactive
systems the response time is of greatest importance. Shneiderman [Shn92]
argues that a response time that is too fast can produce stress and higher
error rates. He thinks that the response time should be kept within a two sec-
ond range. This is of course highly dependent on the application. We do not
expect to reach this range even with incremental updates in our document
preparation system HST; there should be no danger of too fast response. On
the contrary, any speed up in the response time is needed.

Laziness, on the other hand, can further improve the response time\(^1\). We
have not seen laziness used with document preparation but expect it to be
well suited for the area.

Incrementality and laziness are orthogonal concepts; we can have one
without the other. The obvious approach, however, is to combine them.
If only some parts of a problem instance are computed, an update algo-
rithm needs to concentrate only on these parts. In Section 3 we see how
these concepts are combined. In order to enhance the efficiency of the doc-
ument preparation system HIST we apply lazy and incremental methods on
the transformations between different representations of a document. The
methods are divided into two parts: lazy transformation or transformation
of a part of the logical document to a view, and incremental transformations
between the document representations.

\(^1\)It would seem that laziness would not make anything faster. However, we urge the
reader to remember that we use the term laziness in a slightly different manner than in
daily life.
Chapter 3
Structured text and the HST system

Text with structure is quite common: dictionaries, reference manuals and annual reports are typical examples. In recent years, research on systems for writing structured documents has been very intensive. Recent surveys of the field are [AFQ89a, AFQ89b, Qui89, Sal92]. The interest in the area has led to the creation of several document standards, e.g., ODA and SGML [Hor85, HKN89, Jol89, Bar89, Bro89, App91]. In this section we give an example of structured document preparation and also outline the goals of the thesis.

3.1 Grammars for structure description

One way to describe the structure of a document is to use context-free grammars [BR84, CIV86, QV86, GT87, FQA88, KP91]. Thus, in database terminology, grammars correspond to schemas, and parse trees to instances.

The Helsinki Structured Text Database System (HST) is an environment for writing, editing and querying structured documents [KLMN90, KLM+91a]. The document model of HST is based on the work of Gonnet and Tompa [GT87], extended with some further enhancements [KLM+91b].

In HST we describe the document structure with a context-free grammar, a document grammar, and we define views of the document by annotating this grammar. An annotated grammar, a view grammar, defines a view
of the document. It thereby also defines a transformation from the logical representation to a document view. We can define several view grammars for a document.

A document tree is a parse tree over a document grammar. A document tree describes the structure of a document. Additionally it contains the text of the document in its leaves. A view tree is a parse tree over a view grammar. The view tree describes the structure of a view. The textual view of a document instance consists of the terminal leaves, i.e., the frontier, of a view tree.

When the user writes and modifies a document, the HST system maintains three representations of it: the document tree, at least one view tree, and the plain text in an editor.

Example 3 Assume that the user wants to establish a collection of bibliographic references. He describes the structure of the document by Grammar 1. For simplicity, we use a very short grammar. (See Section 4 for definitions of grammars and parse trees.)

Grammar 1

1. \( \text{List} \rightarrow \text{Publications} \)
2. \( \text{Publications} \rightarrow \text{Publication}' \)
3. \( \text{Publication} \rightarrow \text{Author Title Journal Year} \)
4. \( \text{Author} \rightarrow \text{Text} \)
5. \( \text{Title} \rightarrow \text{Text} \)
6. \( \text{Journal} \rightarrow \text{Text} \)
7. \( \text{Year} \rightarrow \epsilon \)
8. \( \text{Year} \rightarrow \text{Text} \)

Note that the grammar defines only the logical structure; no formatting details, e.g., newlines or blank spaces, are present here. A parse tree over this grammar is shown in Figure 3.1.

In the following we denote nonterminals with an emphasized string beginning with a capital letter, e.g., \( \text{Journal}, \text{A} \), or \( \text{Listed_Publications} \). Terminals are denoted by boldface strings, e.g., \textbf{end}, \textbf{;}, or \textbf{Murder_in_Police}. For clarity, we can enclose terminals in single quotation marks, e.g., \text{'}, \text{'}, or

---

16
Figure 3.1: A document tree.

'Murder on the rocks'. Note that any symbols can be part of a terminal, especially also blank spaces and newlines. The empty string is denoted by $\epsilon$ and a blank space by $\perp$.

3.2 Views of documents

In the HST system, the user defines views by annotating the document grammar. An annotated grammar or view grammar contains a modified production for each production of the document grammar. The modified production can omit some of the nonterminals of the original production, reorder the remaining nonterminals and insert terminal strings.

Example 4 Assume that we have the logical document in Example 3 and that the user wants to produce some views of it. Assume that he wants a simple view of his references. Then he can produce a representation of the following form.

Here \n is the newline character. When adding new references, a repre-
sentation such as

**Author:** Fletcher
**Title:** Murder
**Journal:** Police
**Year:** 1990

could be more useful.

Suppose the user has edited the list of references and wants to produce
input for the \LaTeX{} document preparation system [Lam86]. Then he would
like to have a representation such as

\item Fletcher: Murder. {\em Police}, 1990.

Grammar 2 is an annotation of Grammar 1. A parse tree over this gram-
mar is a view of the document instance. The productions have been num-
bered to show the correspondence with the productions in Grammar 1.

Grammar 2

1. $List \rightarrow Publications$
2. $Publications \rightarrow Publication^*$
3. $Publication \rightarrow Author '^\backslash u' Title '^\backslash u' Journal Year '^\backslash u' \backslash n$
4. $Author \rightarrow Text$
5. $Title \rightarrow Text$
6. $Journal \rightarrow Text$
7. $Year \rightarrow \epsilon$
8. $Year \rightarrow '^\backslash u' Text$

Terminal symbols are surrounded by simple quotation marks, and the empty
string is denoted by $\epsilon$. The symbol '\n' stands for the newline character. A
parse tree according to this annotated grammar is shown in Figure 3.2.
3.3 The document preparation model of the HST system

The document representations are processed in different ways (Figure 3.3). The textual view is parsed to form a view tree, which is transformed to form the document tree. A document tree can be transformed into several view trees. The frontier of a view tree always forms a textual view of the document, so the view tree is flattened and the catenation of the strings in the leaves is presented as a textual view. The transformations are further described in [Nik90]. For efficiency reasons, a transformation is always done only by demand (when a view is saved). The user is given the possibility to complete a modification before other representations are updated. Automatic transformations, even if sometimes desirable, would easily produce a great amount of intermediate errors.

The textual view of a document is presented in an editor. It must be noted that the HST system is not a syntax-directed editor. Instead, the user has the freedom to write what he wants. On demand, the text is saved and parsed. Like in program generation, the text must be syntactically correct or the parser will notify errors. We will take a closer look at the parsing process in Section 4.

HST document preparation can be compared with program generation.
Figure 3.3: Different representations of a document in the HST.

When we produce a program, we develop different views or representations of it, e.g., the source code, the object code and the executable code. We start by writing the source code with an editor. Thereafter this source code is parsed and compiled to produce the object code which is linked to produce the executable program.

Many document preparation systems follow a similar scenario. E.g., a \LaTeX{} document [Lam86] is first produced as \LaTeX{} code with an editor. Thereafter we have two phases: the \LaTeX{} code is formatted into a device independent form (dvi), which furthermore is transformed into Postscript code [Ado90a, Ado90b]. If we reverse the document flow in the \LaTeX{} system, we could, e.g., update the Postscript version of a \LaTeX{} document in order to get modifications in the \LaTeX{} source.

An incremental document flow in the HST system comprises incremental parsing and incremental transformation. An additional phase is incremental update of the editor, which we do not consider here. Closely connected is lazy transformation which only transforms a part of the document. The incrementalization of the document flow can be compared with incremental compilation [Rei84, Fri88] where the program transformations are made incrementally. Incremental compilation consists of incremental analysis, incremental parsing, incremental code generation and incremental linking. Furthermore, we can have incremental execution, where execution is continued
Figure 3.4: A document with only a part of the logical representation transformed.

after a program has been recompiled.

3.4 Lazy transformation

When working with large documents, the transformation of the entire logical document to a view can take considerable time. On the other hand, we are mainly interested in what we can see in the editor. This suggests postponing a part of the transformation. According to lazy transformation only a part of the logical document is transformed into a view. The part is determined by a predicate telling where to start the transformation in the document tree and how much to transform. For example, we could try to minimize the transformation and transform only the amount of the document that fits into one editor window.

Assume that the editor contains the first part of the document (Figure 3.4). If the user wants to move forward in the document, the next part of the logical document must be transformed and loaded into the editor. At the same time some parts of the document might have to be removed from the editor window.

The swapping of transformed document parts must be taken into account whenever the user performs an editing operation that moves through the text,
e.g., an operation that moves to the end or beginning of the text, moves one page backwards or forwards or searches the entire text for a certain string.

We assume that the document tree contains the entire document, but that the view tree can be partly empty depending on what parts of the logical document have been transformed. Before a new part is loaded into the editor, the old part must be saved so that possible modifications to the text are saved. If the text is read-only, the part can easily be erased as the underlying logical document remains unmodified in the database.

### 3.5 Incremental updates

Assume that a user is working on a document. He has opened two or more views of it and he is making modifications in one view. The HST system updates the document and thereby the other views in order to make them consistent with the modifications. When working with large documents or many views, these updates considerably slow down the work. What we need here is an incremental update, where as little work as possible is done to make the system consistent. Instead of scanning the entire document, only the modified portion is processed and the modifications are mapped to other views.

The changes are propagated through the different representations. When the user makes a modification in one textual view, the text is parsed and the view tree is transformed to form a new document tree. The document tree is transformed for every other (open) view to form new view trees, and the textual views are updated.

**Example 5** Assume that the user has opened three views of a document and that he is currently making a modification to the first view (Figure 3.5).

When he saves the first view, the modification must be propagated to the different representations of the document. The incremental update starts by parsing the modification and updating the first view tree (Figure 3.5, parse a). The modified part of the view tree is inverted back to the document tree, updating the old tree (Figure 3.5, transformation b). After that the modifications in the document tree are transformed to update the other view trees (Figure 3.5, transformations c and d). The update ends with substituting the corresponding text in the editor windows (Figure 3.5, substitutions e and f). All modifications are made incrementally.
A simple modification modifies only one leaf in the view tree. It can be a change, insertion, or deletion. A modification to the textual view consists of a simple modification or a series of simple modifications. Usually, simple modifications are easier to update, but we place no restrictions on the modifications made. Depending on the view (document) grammar, a simple modification can affect the entire view (document) tree.

Several modifications can be considered as a series of simple modifications and processed as a series of incremental updates, i.e., the incremental update algorithm is called for every simple modification. Several modifications can
also be considered as one large modification and the incremental update algorithm is called only once. Sometimes two modifications to a text might cancel the effects on the view tree, sometimes two modifications only affect the view tree in the close vicinity of the corresponding leaves.

3.6 Combining incrementality with laziness

Laziness and incrementality combine very well. We give an example of how to benefit from the advantages of both facilities.

Assume that the user has opened a lazily transformed document. Assume also that the space of the editor is limited. Before transforming and loading more parts of the document, the current part must be parsed and transformed back to the document tree.

![Diagram showing lazily transformed document and its update.](image)

Figure 3.6: A lazily transformed document and its update.

If the user has made some modifications to the text, the relevant part of the document is incrementally parsed and the view tree is updated. The modifications are incrementally transformed to the document tree and other possible views are incrementally updated (Figure 3.6). Subsequent parts of the document can be updated in the same way. Lazy transformation is applied in one direction from the logical representation to the textual representation, whereas the incremental update is applied first towards the logical document and then towards other views.
Some questions remain open. For example, we must decide what the granularity of the lazy transformation is. A natural size of the portion to be processed is, of course, a subtree. But subtrees can be of various sizes. Implementation plans and technical problems are discussed in Section 8.
Chapter 4
Grammars, parsing and tree distances

Many types of structural information can be defined by context-free grammars. We use context-free grammars to define the structure of documents. The text of a document is parsed to form a parse tree over a certain grammar. Here we use the recursive descent parsing method.Parsed documents can be transformed into other parse trees for producing different views. These transformations are achieved by syntax-directed translations. In this section we give some general definitions on context-free grammars, parsing, and syntax-directed translations. We also present the tree-to-tree correction problem and define some different ways to update a tree. These concepts will be used later in the development of our incremental parser and our incremental syntax-directed translator.

4.1 Context-free grammars

We start with some general definitions. A rewriting system is a pair \( R = (V, P) \), where \( V \) is an alphabet and \( P \) is a finite set of ordered pairs of strings over \( V \). The elements of \( P \) are referred to as rewriting rules or productions and denoted \( \alpha \rightarrow \beta \), where \( \alpha, \beta \in V^* \).

Given a rewriting system the yield relation \( \Rightarrow \) on the set \( V^* \) is defined as follows. For any strings \( \alpha \) and \( \beta \), the relation \( \alpha \Rightarrow \beta \) holds if and only if there are strings \( \alpha_1, \alpha_2, \gamma, \delta \), such that \( \alpha = \alpha_1 \gamma \alpha_2 \) and \( \beta = \alpha_1 \delta \alpha_2 \) and \( \gamma \rightarrow \delta \)
is a production in the system. We also say that $\alpha$ derives $\beta$. The reflexive transitive closure of the relation $\Rightarrow$ is denoted by $\Rightarrow^*$. Thus $\alpha \Rightarrow^* \beta$ holds if and only if there are $n \geq 1$ strings $\gamma_1, \ldots, \gamma_n$ such that $\alpha = \gamma_1$, $\beta = \gamma_n$ and $\gamma_i \Rightarrow \gamma_{i+1}$ holds for every $i = 1, \ldots, n - 1$. Correspondingly, the transitive closure of the relation $\Rightarrow$ is denoted by $\Rightarrow^+$, and $\alpha \Rightarrow^+ \beta$ holds if and only if there are $n \geq 2$ strings $\gamma_1, \ldots, \gamma_n$ such that $\alpha = \gamma_1$, $\beta = \gamma_n$ and $\gamma_i \Rightarrow \gamma_{i+1}$ holds for every $i = 1, \ldots, n - 1$. We say that \textit{“$\alpha$ derives $\beta$ in zero or more steps”} if $\alpha \Rightarrow^* \beta$ and that \textit{“$\alpha$ derives $\beta$ in one or more steps”} if $\alpha \Rightarrow^+ \beta$.

A grammar is a quadruple $G = (N, T, P, S)$, where $(N \cup T, P)$ is a rewriting system, and $S \in N$. We call $N$ the nonterminal alphabet or the set of nonterminals. The set of tokens $T$ is called the terminal alphabet or the set of terminals. The symbol $S$ is the start symbol of $G$.

Grammars are classified according to the Chomsky hierarchy into four groups [Cho59, Sal90]. It can be shown that the hierarchy is a strictly decreasing hierarchy of language families. In the first group, we set no restrictions on the productions.

In the second group, the productions are of type $A \beta \rightarrow \alpha \delta \beta$, where $\alpha$ and $\beta$ are arbitrary strings over $V$, $\delta$ is a nonempty string over $V$ and $A$ is a nonterminal symbol. Also productions of the type $S \rightarrow \epsilon$ are possible ($\epsilon$ is the empty string), but only if $S$ does not occur on the right hand side of any production. These grammars are called context-sensitive.

In the third group, the productions are of type $A \rightarrow \alpha$, where $A \in N$ and $\alpha$ is an arbitrary string over $V$. These grammars are called context-free.

In the fourth group, the productions are of type $A \rightarrow aB$, where $A, B \in N$ and $a \in T$. We can also have productions of the type $A \rightarrow \epsilon$ where $A \in N$. These grammars are called right-linear or regular.

The language $L(G)$ generated by a given grammar $G$ is the set $\{w | S \Rightarrow^+ w\}$ of strings{
4.2 Parse trees

A parse tree shows pictorially how the start symbol of a grammar derives a string in the language. Before we define parse trees formally, we take a general look at trees.

A tree is a collection of elements called nodes, one of which is distinguished as a root, along with a relation (parenthood) that places a hierarchical structure on the nodes. Formally, a tree can be defined in the following manner. 1. A single node is itself a tree. The node is also the root of the tree. 2. Suppose $n$ is a node and $t_1, t_2, \ldots, t_k$ are trees with roots $n_1, n_2, \ldots, n_k$, respectively. We construct a new tree by making $n$ the parent of nodes $n_1, n_2, \ldots, n_k$. In this tree, $n$ is the root and $t_1, t_2, \ldots, t_k$ are the subtrees of the root. Nodes $n_1, n_2, \ldots, n_k$ are called the children of node $n$ [AHU87].

Often it is useful to associate a value or label with a node of a tree. The label is not the name of the node, even if we often identify a node by its label.

Trees are usually traversed in preorder, postorder, or inorder. In preorder, the root of a subtree is first processed, thereafter the children from left to right. In postorder, the children are first processed from left to right and thereafter the root. In inorder, we first the process the first child, then the root, and thereafter the rest of the children from left to right.

Given a context-free grammar $G$, a parse tree is a tree with the following properties [ASU66]:

1. The root is labeled by the start symbol.
2. Each leaf is labeled by a terminal or by $\epsilon$.
3. Each interior node is labeled by a nonterminal.
4. If $A$ is the nonterminal labeling some interior node $m$ and $X_1, X_2, \ldots, X_n$ are the labels of the children of that node from left to right, then $A \rightarrow X_1X_2 \cdots X_n$ is a production of the grammar $G$. Here $X_1, X_2, \ldots, X_n$ stand for symbols that are either terminals or nonterminals. As a special case, if $A \rightarrow \epsilon$ is a production in $P$ then a node labeled $A$ may have a single child labeled $\epsilon$.

We say that the production $A \rightarrow X_1X_2 \cdots X_n$ has been expanded or applied at node $m$. The leaves of the parse tree read from left to right form the
yield or the frontier of the tree, which is the string generated or derived from the nonterminal at the root of the parse tree. The grammar $G$ is unambiguous if for each terminal string there is at most one possible parse tree.

**Example 6** We already saw some examples of parse trees, which were document instances and views. See Examples 3 and 4 on pages 16 and 17. □

When we speak about a symbol in a production rule of a grammar, we talk about its occurrence. A symbol can occur several times in the right hand side of a production. When we expand a production at a node in a parse tree, the label of the node is an instance of a symbol. Let $n$ be a node in a parse tree and let $p$ be the production expanded at $n$. Then the node $n$ has as many children as there are symbol occurrences on the right hand of the production $p$. The labels of the children, the symbol instances, correspond to the occurrences on the right hand side of the production.

### 4.3 Some further definitions and notations

Let $G = (N, T, P, S)$ be a context-free grammar. Let $V = N \cup T$. We define the following functions on $V$.

- **First** $(X) = \{ a \mid a \in T \text{ and } X \Rightarrow a\alpha \text{ for some } \alpha \in V^\ast \}$
- **Follow** $(X) = \{ a \mid a \in T \text{ and } S \Rightarrow \alpha Xa\beta \text{ for some } \alpha \in V^\ast \text{ and } \beta \in V^\ast \}$
- **Nullable** $(X) = \text{ if } X \Rightarrow \epsilon \text{ then TRUE else FALSE}$

The function **First** determines the first set of the symbol $X$. This set contains all the terminals that $X$ can derive as its first symbol. Correspondingly, the **Follow** function determines the follow set of the symbol $X$. The follow set contains all the terminal symbols that immediately can follow the symbol $X$ in a derivation. The Boolean function **Nullable** tells if $X$ can derive the empty string. We extend the function **Nullable** for a sequence of symbols in $V$. A sequence $X_1X_2\cdots X_n$ is nullable, if every symbol $X_i$ in the sequence is nullable, i.e., **Nullable** $(X_1X_2\cdots X_n)$ if and only if **Nullable** $(X_i)$ for every $i$, where $1 \leq i \leq n$.

We define on $V^*$ the following function.
The \textit{Last} function determines the \textit{last symbol} of \( \alpha \).

In a parsing process we usually want to be able to choose the appropriate production by only looking at the next input symbol. Therefore we define the function \textit{Dirsym} on productions of \( G \) [MPS90].

\[
\text{Dirsym}(X_0 \rightarrow X_1 X_2 \cdots X_n) = \{ a \mid a \in \text{First}(X_i), \text{ where } 1 \leq i \leq n \text{ and } \text{Nullable}(X_1 X_2 \cdots X_{i-1}) \} \cup \begin{cases} \text{if Nullable}(X_1 X_2 \cdots X_n) \text{ then } \text{Follow}(X_0) \text{ else } \emptyset \end{cases}
\]

The \textit{Dirsym} function determines the \textit{dirsym} of a production. The set contains all terminals that can be the first symbols derived using this production. If all nonterminals on the right hand side of the production are nullable, we include the follow set of the left hand side nonterminal \( X_0 \).

\textbf{Example 7} Consider the context-free grammar Grammar 3. Boldface characters stand for terminal symbols and capital letters for nonterminals.

\textbf{Grammar 3}
\[
\begin{align*}
S & \rightarrow B A \quad C \rightarrow g \\
A & \rightarrow a b c \quad C \rightarrow \epsilon \\
A & \rightarrow d e f \quad D \rightarrow h \\
B & \rightarrow C D \quad D \rightarrow \epsilon
\end{align*}
\]

Here we have the following dirsym sets.

\begin{center}
\begin{tabular}{ccc}
Production & Dirsym set & Production & Dirsym set \\
\hline
\( S \rightarrow BA \) & \{ g, h, a, d \} & \( C \rightarrow g \) & \{ g \} \\
\( A \rightarrow abc \) & \{ a \} & \( C \rightarrow \epsilon \) & \{ a, d, h \} \\
\( A \rightarrow def \) & \{ d \} & \( D \rightarrow h \) & \{ h \} \\
\( B \rightarrow CD \) & \{ g, h, a, d \} & \( D \rightarrow \epsilon \) & \{ a, d \}
\end{tabular}
\end{center}

It is important that the grammar is unambiguous. Here we see that if two productions have the same left hand symbol, their dirsym sets are always disjoint. The parser will know which production to expand by only looking at one symbol at a time. The requirement that the dirsym sets are disjoint is actually equivalent to the LL(1) condition to be given in Section 4.5. \( \square \)
Algorithm 1 (Top-down parsing of a nonterminal)

```
procedure parseX
Input:    Text to be parsed.
Output:   Parsed subtree according to nonterminal X.
Task:     Parse input and construct parse tree (top-down).
begin
1   Let a be the next input symbol ;
2   if X is a terminal then
3       if X = a then
4           Read a;
5         else
6           error;
7       else
8         Let a be in the dirsym set
9         Dirsym(X → X₁ X₂ ⋯ Xₖ);
10        /* Expand the production X → X₁ X₂ ⋯ Xₖ */;
11        for all Xᵢ do
12           parseXᵢ;
end;
```

4.4 Recursive descent parsing

Parsing is the process of determining if a string of tokens can be generated by a grammar. Most parsing methods fall into one of two classes, called top-down and bottom-up methods. These terms refer to the order in which nodes in a parse tree are constructed. In the former, construction starts at the root and proceeds towards the leaves, whereas in the latter, construction starts at the leaves and proceeds towards the root.

Recursive-descent parsing is a top-down method of syntax analysis in which we execute a set of recursive procedures to process the input. A procedure is associated with each nonterminal of the grammar. Here we consider a special form of recursive-descent parsing called predictive parsing, in which the lookahead symbol, i.e., the next symbol in the string being parsed, unambiguously determines the procedure selected for each nonterminal. The
sequence of procedures called in processing the input implicitly defines a
parse tree for the input [ASU86].

Each procedure in a predictive parser does two things. First, it decides
which production to use by looking at the lookahead symbol. The production
$A \rightarrow \alpha$ is used if the lookahead symbol is in $\text{Dirsym}(A \rightarrow \alpha)$. If there is a
conflict between two right sides for any lookahead symbol, we cannot use this
 parsing method.

Second, it uses a production mimicking the right side. A nonterminal
results in a call to the procedure for the nonterminal, and a token matching
the lookahead symbol results in the next input symbol being read. If at some
point the token in the production does not match the lookahead symbol, an
error is emitted. The parse function for symbol $X$ is given in Algorithm 1.

The parse procedures can be automatically constructed from a given
grammar. The parsing can also be done in an interpreting way. See [Nik90]
for further details on parsing structured documents.

## 4.5 The LL(1) conditions

Before the parser expands a production with a certain right hand side, it
checks if the next input symbol belongs to the dirsym set of the production.
Therefore we demand that the dirsym sets of two productions with the same
left hand side must be disjoint, that is, for each pair of productions $A \rightarrow \alpha$
and $A \rightarrow \beta$ we have that

$$\text{Dirsym}(A \rightarrow \alpha) \cap \text{Dirsym}(A \rightarrow \beta) = \emptyset$$

This equation is equivalent with the so called $LL(1)$ conditions for a grammar.
We say that the grammar is $LL(1)$ if the following conditions hold for each
pair of productions $A \rightarrow \alpha$, $A \rightarrow \beta$.

1. $\text{First}(\alpha) \cap \text{First}(\beta) = \emptyset$.
2. Only one of $\alpha$ and $\beta$ can derive the empty string.
3. If $\beta \Rightarrow^{*} \epsilon$, then $\alpha$ does not derive a string beginning with a terminal in
   $\text{Follow}(A)$.

These conditions guarantee that the grammar is unambiguous.
4.6 Grammars with variable strings

In HST we use context-free grammars not only to describe the logical structure of the documents, but also to describe formatting details. We extend the grammars to include two kinds of terminals: constant strings and variable strings. The constant strings can consist of any characters in the character set. Examples of constant strings are 'if', '.' and '#$. In the following we denote constant strings with $c$ and assume that we can substitute this string for an any character sequence. The constant strings are given in the grammar. The variable strings can contain any character sequence, but we do not know before parsing what the string will be. A variable string is derived from the nonterminal Text and its instance depends on the succeeding constant strings. The Text nonterminal is nullable. We call grammars with variable strings textable.

Example 8 Consider the grammar

$$S \rightarrow AA#$$
$$A \rightarrow Text$$

where & and # are constant strings. The nonterminal A derives variable strings: the first A in the first production derives only strings that do not contain & as a substring, the second A derives strings that do not contain # as a substring. Therefore an occurrence of the Text nonterminal depends on what constant strings succeed it in a derivation and we can say that the grammar is context-sensitive if we write it in the form

$$S \rightarrow AA#$$
$$A \rightarrow Text$$
$$Text&w_2 \rightarrow c_1&w_2$$
$$Text# \rightarrow c_2#$$

where $c_1$ denotes any string that does not contain the constant string & and $c_2$ denotes any string that does not contain the constant string #. In the following, however, we consider a context-free grammar extended with textable productions still to be context-free. □
Parsing is always done according to a view grammar. Therefore it is not wise to have productions of type \( A \rightarrow \text{Text Text } c \); because the second occurrence of the Text nonterminal would always derive an empty string. If we have a delimiter (a constant string) between the two occurrences of Text nonterminals, the parser knows when to stop parsing the first occurrence of Text and when to continue with the second. In a document grammar this sort of production is possible; the document tree is translated from a view tree, where the textual instances of the two Text occurrences are clearly separated.

We define a new Boolean function \( \text{Textable} \) on \( V \), which tells when a symbol can derive a variable string as its immediate starter.

\[
\text{Textable}(X) = \begin{cases} 
\text{true} & \text{if } X \Rightarrow \text{Text } a \text{ for some } a \in V^* \\
\text{false} & \text{else}
\end{cases}
\]

We extend the definition of the function \( \text{Textable} \) for sequences of symbols in \( V \). A sequence \( X_1 X_2 \cdots X_n \) is textable if for some \( i \) we have \( \text{Textable}(X_i) \) and \( \text{Nullable}(X_1 X_2 \cdots X_{i-1}) \). We say that a production is textable, if its right hand side is textable.

All terminal symbols are of type string. For textable grammars, the token classes are determined by the grammars. When we talk about tokens in the sequel we mean constant or variable strings.

Parsing is usually preceded by a lexical analysis process. An analyzer reads the input and returns input tokens to a parser. When the tokens are not context-dependent, the analyzer is implemented as a separate process. This is the case in most programming languages. A program consists of some reserved words (constant strings) and some identifiers (variable strings). The identifiers can be determined without looking at the grammar, e.g., an identifier might be an alphanumeric string that starts with a letter. When we deal with context-dependent tokens we cannot separate the analyzer from the parser. Instead the tokens are determined by a grammar. The parser reads the input; a grammar determines where to stop and start reading the next token. This is implemented by giving a set of “stop strings” as a parameter to the parser. Similar techniques can be found in [Kos90, JKP91] discussing modular implementation techniques of programming languages.

In the sequel, we consider our grammars to be context-free. For this reason we must slightly modify the LL(1) condition for our textable grammars.
4.7 The LL(1) condition for textable grammars

Here we study what the LL(1) conditions mean for textable grammars. Assume that the grammar $G$ contains the productions $A \rightarrow \alpha$ and $A \rightarrow \beta$.

1. First we demand that $\text{First}(\alpha) \cap \text{First}(\beta) = \emptyset$.

Then $\alpha$ and $\beta$ cannot start with the same terminal, but we have an important exception. We allow production pairs of type

$$
\begin{align*}
A &\rightarrow \text{Text} \\
A &\rightarrow c
\end{align*}
$$

where $\alpha = \text{Text}$ and $\beta = c$ which is a constant string. Here the variable string is determined by the constant string. It can contain any substrings but it cannot start with the constant string $c$. Two textable productions of the same grammar must, however, have different left hand sides, i.e., we do not allow production pairs of type $A \rightarrow \text{Text}_1$, and $A \rightarrow \text{Text}_2$.

2. Only one of $\alpha$ and $\beta$ can derive the empty string.

The $\text{Text}$ nonterminal is nullable. If $\alpha \Rightarrow^* \text{Text}$ ($A$ is textable), then we may not have $\beta \Rightarrow^* \text{Text}$ or $\beta \Rightarrow^* \epsilon$. But, e.g., $\beta \Rightarrow^* c \text{Text}$ is acceptable.

3. If $\beta \Rightarrow^* \epsilon$, then $\alpha$ does not derive a string beginning with a terminal in $\text{Follow}(A)$.

Here again we have to take into account that the $\text{Text}$ nonterminal is nullable. If we have

$$
\begin{align*}
S &\rightarrow AB \\
A &\rightarrow \text{Text} \\
A &\rightarrow &c \\
B &\rightarrow &c^*
\end{align*}
$$
we have a problem parsing the sentence `&&` (because the grammar is ambiguous). The nonterminal `A` is nullable and can start with a terminal that belongs to its follow set.

When checking the LL(1) conditions we have to remember that the `Text` nonterminal is nullable. The only exception that we make is to allow production pairs of type `A → Text` and `A → c`. This is not actually an exception, because when we introduce the production `A → c`, we give the `Text` nonterminal a new meaning. Without such a production in the grammar, `Text` could derive all possible strings over the terminal alphabet, and now it can derive all strings that do not contain the constant string `c`.

### 4.8 Syntax-directed translations

In this section we define the meaning of an annotated grammar formally. A *syntax-directed translation scheme* (SDTS) [AU72] is a 5-tuple `F = (N, Σ, Δ, R, S)`, where `N` is a finite set of nonterminal symbols, `Σ` is a finite input alphabet, and `Δ` is a finite output alphabet. `R` is a finite set of rules of the form `W → α; β`, where `W ∈ N`, `α ∈ (N ∪ Σ)⁺`, and `β ∈ (N ∪ Δ)⁺`, and the nonterminals of `β` are a permutation of the nonterminals of `α`. Finally `S` is a distinguished nonterminal of `N`, the start symbol.

Now, suppose the user has described the structure of a document by a context-free grammar `G = (N, T, P, S)` and that he has annotated the productions of `P`. Then an annotated grammar for `G` is a translation scheme `F_G = (N, T, T’, P’, S)`, where `N` is the (common) set of nonterminals of `G`, and `T` and `T’` are the sets of terminals of the grammar `G` and of the annotated version, respectively, and `P’` is the set of productions `W → α; β`. Here `W → α` is the original production of `G` and `W → β` is the annotated version of it [Nik90].

Let `W → α; β` be a rule. To each nonterminal of `α` there is *associated* an identical nonterminal of `β`. If a nonterminal `B` appears only once in `α` and `β`, then the association is obvious. If there are more than one occurrence of `B` in `α`, we associate the `B’s` in `α` with `B’s` in `β` in the same order1. The first

---

1 We could also use integer superscripts to indicate the association. This association would be an intimate part of the rule. For example, in the rule `W → B¹ CB²; B² B¹ C`, the three positions in `B¹ CB²` are associated with positions 2, 3, and 1, respectively, in `B² B¹ C` [AU72].
B in \( \alpha \) is associated with the first \( B \) in \( \beta \) and so on. This concerns especially iterations. In the rule \( W \rightarrow B^* ; B^* \), the occurrences of \( B \) in \( \alpha \) are associated with the occurrences of \( B \) in \( \beta \) in the same order.

A translation form of a syntax directed translation schema \( F \) is defined as follows [AU72]:

1. \((S, S)\) is a translation form and the two \( S \)'s are said to be associated.

2. If \((\alpha A \beta, \alpha' A' \beta')\) is a translation form, in which the two explicit occurrences of \( A \) are associated, and if \( A \rightarrow \delta \); \( \delta' \) is a rule in \( R \), then \((\alpha \delta \beta, \alpha' \delta' \beta')\) is a translation form. The nonterminals of \( \delta \) and \( \delta' \) are associated as they are associated in the rule \( A \rightarrow \delta \); \( \delta' \) and the nonterminals of \( \alpha \) and \( \beta \) are associated with those of \( \alpha' \) and \( \beta' \) in the new translation form \((\alpha \delta \beta, \alpha' \delta' \beta')\) in the same way as they are associated in the translation form \((\alpha A \beta, \alpha' A' \beta')\).

The relation between the translation forms \((\alpha A \beta, \alpha' A' \beta')\) and \((\alpha \delta \beta, \alpha' \delta' \beta')\) is denoted by \(\Rightarrow F\) and we write \((\alpha A \beta, \alpha' A' \beta') \Rightarrow F (\alpha \delta \beta, \alpha' \delta' \beta')\). We often drop the subscript \( F \). The relation is transitive and reflexive and we use \(\Rightarrow\) to stand for the reflexive transitive closure.

The syntax-directed translation (SDT) defined by \( F \), denoted \( \tau(F) \), is a set of pairs, \( \tau = \{(x, y) \mid (S, S) \Rightarrow (x, y) \text{ where } x \in \Sigma^* \text{ and } y \in \Delta^* \} \). Finally, the input grammar \( F_i = (N, \Sigma, P, S) \) of an SDTS \( F = (N, \Sigma, \Delta, R, S) \) has the production set \( P = \{W \rightarrow \alpha \mid W \rightarrow \beta; \beta \in R\} \) and the output grammar \( F_o = (N, \Delta, P', S) \) of \( F \) has the production set \( P' = \{W \rightarrow \beta \mid W \rightarrow \alpha; \beta \in R\} \).

Let \( F \) be an SDTS and let \( x \) be a string of the language \( L(F_i) \). The view of \( x \) is a string \( y \) in \( L(F_o) \) such that \((x, y) \) is in \( \tau(F) \). Both grammars \( F_i \) and \( F_o \) are considered to be unambiguous. The translation can therefore be considered as a function mapping strings to strings. Here we consider the translation as a function mapping trees to trees; every string in \( L(F_i) \) has a corresponding parse tree which is mapped to the parse tree of the string \( y \in L(F_o) \), if \( y \) is the view of \( x \).

**Example 9** Consider a syntax-directed translation schema \( F \) with productions
Algorithm 2 *(Syntax-directed translation of a tree)*

```plaintext
procedure translate(t: parse tree);
Input: Tree t.
Output: Tree t translated into t'.
Task: Translate tree t recursively.
begin
    n := t.root;
    for i = 1 to n.size do
        if (n.child[i] is a leaf) then
            delete n.child[i];
        let A → α ∈ G be the production expanded at n
        and let A → β be the corresponding annotated
        production in G;
        permute the children of n in accordance with the
        association between nonterminals of α and β;
        insert leaves into n so that the labels of its
        children form β;
        for i = 1 to n.size do
            if (n.child[i] is NOT a leaf) then
                translate(n.child[i].root.label(n.child[i]));
end;
```

We have the following derivation

```
Author → First_Name Middle_initial* Last_name ;
          Last_name ', First_Name Middle_initial*
First_name → Text ; Text
Middle_initial → Text '.' ; Text '.'
Last_name → Text ; Text
```
(Author, Author)

\[ \Rightarrow (\text{First\_name Middle\_initial Last\_name }, \]
\[ \text{Last\_name }', ' \text{First\_name Middle\_initial}) \]
\[ \Rightarrow (\text{Text Middle\_initial Last\_name }, \]
\[ \text{Last\_name }', ' \text{Text Middle\_initial}) \]
\[ \Rightarrow (\text{Jessica Middle\_initial Last\_name }, \]
\[ \text{Last\_name }', ' \text{Jessica Middle\_initial}) \]
\[ \Rightarrow (\text{Jessica Text }', ' \text{Last\_name }, \]
\[ \text{Last\_name }', ' \text{Jessica Text }') \]
\[ \Rightarrow (\text{Jessica B . Last\_name , Last\_name }', ' \text{Jessica B }'.') \]
\[ \Rightarrow (\text{Jessica B . Text , Text }', ' \text{Jessica B }'.') \]
\[ \Rightarrow (\text{Jessica B ' . Fletcher , Fletcher }', ' \text{Jessica B }'.') \]

A view or translation of the string Jessica B. Fletcher is the string Fletcher, Jessica B. Note that even if the translation forms above contain several instances of the nonterminal Text, it is always clear how they are associated due to the definition of associated nonterminals. If we present the strings as parse trees \( t \) and \( t' \) we say that a translation of tree \( t \) is tree \( t' \) (Figure 4.1).

\[ \square \]

A translation can be done top-down starting at the root. Assume that we have a grammar \( G_i \) as an input grammar of an SDTS \( F \) and let \( G_o \) be the output grammar of \( F \). Let \( t \) be a parse tree over \( G_i \) representing a document. We transform the tree into a parse tree \( t' \) over \( G_o \). The translation method is given in Algorithm 2. The algorithm is given in [Nik90, pages 28 – 29].

We use the following notations in the algorithm. Let \( t \) be a parse subtree. Then \( t.root \) is the root of the tree. Let \( n \) be a node in the tree. Then \( n.size \) indicates the number of children of the node \( n \), \( n.child \) indicates the subtree rooted at the \( i \)th child of \( n \), and \( n.label \) indicates the label of the node \( n \).

**Example 10** We continue Example 9. Consider a syntax-directed translation schema \( F \) with the following productions.
Figure 4.1: A tree and its translation.

\[
\begin{align*}
\text{Author} & \rightarrow \text{First\_name Middle\_initial}^* \text{ Last\_name} ; \\
\text{First\_name} & \rightarrow \text{Text} ; \text{Text} \\
\text{Middle\_initial} & \rightarrow \text{Text} \,' ; \text{Text} \,' \\
\text{Last\_name} & \rightarrow \text{Text} ; \text{Text}
\end{align*}
\]

and the parse tree

\[
\begin{align*}
\text{Author} & \rightarrow \text{First\_name Middle\_initial}^* \text{ Last\_name} ; \\
\text{First\_name} & \rightarrow \text{Text} ; \text{Text} \\
\text{Middle\_initial} & \rightarrow \text{Text} \,' ; \text{Text} \,' \\
\text{Last\_name} & \rightarrow \text{Text} ; \text{Text}
\end{align*}
\]
We now use the translation algorithm to transform the tree. First the subtrees of the node *Author* are permuted (a) and the terminal children are inserted to the root (b). Then the terminals of the input grammar are replaced by the terminals of the output grammar, but here this phase is trivial, because they do not change. Actually, the constant string child '.' of the node *Middle_initial* is first removed and then reinserted.

\[
\begin{align*}
\text{Author} & \quad \Rightarrow \quad \text{Author} \\
\text{First\_name} & \quad \text{Middle\_initial} & \quad \text{Last\_name} \\
\text{Text} & \quad \text{Text} & \quad \text{Text} \\
\text{Jessica B. Fletcher} & \quad \text{Fletcher, Jessica B.}
\end{align*}
\]

\[\square\]

### 4.9 Edit distances between trees

In this section we give some *measures* of difference between two trees, taking into account how transformations between the trees are done. These measures help us to express the efficiency of different incremental update algorithms (see Sections 5.9, 6.5.1 and 6.6.1).

The *tree-to-tree correction problem* is to determine for two labeled trees $t$ and $t'$ the distance from $t$ to $t'$ as measured by the minimum cost sequence of edit operations needed to transform $t$ into $t'$. We consider three kinds of operations: *changes*, *deletes* and *inserts*. Since a string can be considered a tree of depth two with a virtual root added, the *string-to-string correction problem* [WF74] is just a special case of the tree-to-tree correction problem.

The tree-to-tree correction problem has been studied first by Selkow [Sel77]. Tai [Tai79] presented an algorithm with time complexity $O(|t| \times |t'| \times d(t) \times d(t'))$, where $|t|$ and $|t'|$ are the number of nodes in each tree, respectively, and $d(t)$ and $d(t')$ stand for the depths of the trees. Zhang and Shasha have presented an algorithm with time complexity $O(|t| \times |t'| \times$
\[ \min\{d(t), l(t)\} \ast \min\{d(t'), l(t')\}, \] where \( l(t) \) and \( l(t') \) are the number of leaves in the trees [ZS89]. They have given further enhancements in [SZ90].

We define below three different edit distances between trees. The \textit{minimum edit distance} tells the minimal cost of all edit sequences that transform one tree into another (no restrictions on the trees). The \textit{preorder distance} tells the minimal cost of all edit sequences that are done in strict preorder and transform one tree into another and, finally, the \textit{parse distance} tells the minimal cost of all edit sequences that transform one tree into another where the trees follow a certain grammar (and the parsing is done top-down).

\subsection{The minimum edit distance}

We consider here three kinds of operations. A \textit{change} in a node means changing the label of the node. \textit{Deleting} a node \( n \) means removing the node and attaching its children to the parent of \( n \). \textit{Inserting} is the complement of delete. When we insert a node \( n \) as a child of a node \( n' \), we make it the parent of a consecutive sequence of children of the node \( n' \).

Let \( \Lambda \) denote the null node. An edit operation is written \( b \leftarrowrightarrow c \), where \( b \) and \( c \) are either nodes or \( \Lambda \). We let the label of the node represent the node when no misunderstanding can occur. The operation \( b \leftarrowrightarrow c \) is a change operation if \( b \neq \Lambda \) and \( c \neq \Lambda \). It is a delete operation if \( b \neq \Lambda \) and \( c = \Lambda \), and it is an insert operation if \( b = \Lambda \) and \( c \neq \Lambda \). The null operation \( \Lambda \leftarrowrightarrow \Lambda \) is not considered. The null node is more a technicality for defining the delete and insert operations. When we define a tree, we never explicitly say that it can contain null nodes.

Let \( S \) be a sequence \( s_1, \ldots, s_k \) of edit operations. An \textit{S-derivation} from tree \( t \) to tree \( t' \) is a sequence of trees \( t_0, \ldots, t_k \), such that \( t = t_0, t' = t_k \), and we get tree \( t_i \) from tree \( t_{i-1} \) by applying the edit operation \( s_i \), where \( 1 \leq i \leq k \).

Let \( \gamma \) be a cost function that assigns to each edit operation \( b \leftarrowrightarrow c \) a non-negative real number \( \gamma(b \leftarrowrightarrow c) \). This cost can vary for separate nodes. We constrain \( \gamma \) to be metric, so the following conditions must hold:

1. \( \gamma(b \leftarrowrightarrow c) \geq 0 \) and \( \gamma(b \leftarrowrightarrow b) = 0 \)
2. \( \gamma(b \leftarrowrightarrow c) = \gamma(c \leftarrowrightarrow b) \)
3. \( \gamma(a \leftarrowrightarrow c) \leq \gamma(a \leftarrowrightarrow b) + \gamma(b \leftarrowrightarrow c) \)
Figure 4.2: Two labeled trees with minimum edit distance 1.

We extend the cost function $\gamma$ to the edit sequence $S = s_1, s_2, \ldots, s_k$ by letting $\gamma(S) = \sum_{i=1}^{k} \gamma(s_i)$. Now, we can define the minimum edit distance between two trees [Tai79].

**Definition 1** Let $t$ and $t'$ be two labeled trees. The *minimum edit distance* for $t$ and $t'$ is

$$mindist(t, t') = \min \{ \gamma(S) \mid S \text{ is an edit sequence transforming } t \text{ to } t' \}$$

The distance function $mindist$ is metric due to its definition.

**Example 1** In Figure 4.2 we have two labeled trees $t$ and $t'$. Changing tree $t$ to tree $t'$ can be done in several ways. Let the labels of the nodes stand for the nodes themselves. Then the edit sequence $A \mapsto B, B \mapsto C, C \mapsto A$ changes $t$ to $t'$. Under the unit cost function (i.e., $\gamma(b \mapsto c) = 1$, when $b \neq c$), the cost of this sequence is 3 (three edit operations). Another sequence, $A \mapsto A$, transforms $t$ to $t'$ with cost 1, which is obviously the minimum cost. Therefore $mindist(t, t') = 1$ under the unit cost function.

4.9.2 Mappings

The number of different sequences of edit operations which transform $t$ into $t'$ is infinite. Therefore it is hard to enumerate all valid sequences and find
the minimum cost. We define mappings \cite{Tai79} that help us compute the distance in polynomial time. Intuitively, a mapping is a description of how a sequence of edit operations transforms \(t\) into \(t'\), ignoring the order in which the edit operations are applied. Before defining a mapping we need to define some auxiliary concepts. The ancestor of a node in a tree is its parent or a parent of an ancestor. The root of a tree is an ancestor of all nodes in the tree (except of the root itself). Consequently, a node \(m\) is a descendant of a node \(n\) if \(n\) is an ancestor of \(m\). A node is a sibling of another node if both nodes have the same parent. We say that a node \(m\) is to the left of another node \(n\) if \(m\) occurs before \(n\) in postorder and \(m\) is not a descendant of \(n\).

Suppose we have an ordering for each tree. Any complete ordering will do. We denote by \(t[i]\) the \(i\)th node of tree \(t\) in the given ordering. A mapping is a triple \((M, t, t')\), where \(M\) is a set of pairs of integers \((i, j)\). The following conditions must hold.

1. for any \((i, j) \in M\) we have \(1 \leq i \leq |t|\) and \(1 \leq j \leq |t'|\)

2. for any pairs \((i_1, j_1)\) and \((i_2, j_2)\) in \(M\) we have
   (a) \(i_1 = i_2\) if and only if \(j_1 = j_2\) (one to one relation)
   (b) \(t[i_1]\) is to the left of \(t[i_2]\) if and only if \(t'[j_1]\) is to the left of \(t'[j_2]\)
   (c) \(t[i_1]\) is an ancestor of \(t[i_2]\) if and only if \(t'[j_1]\) is an ancestor of \(t'[j_2]\).

Condition 2b) states that the sibling order is preserved, and condition 2c) that the ancestor order is preserved.

When we transform \(t\) into \(t'\) we perform the following operations. The nodes in \(t\) that are mapped on nodes in \(t'\) are relabeled if their labels differ from those of their corresponding nodes in \(t'\). The nodes in tree \(t\) that are not mapped are removed and the nodes in \(t'\) which do not have a corresponding node in \(t\) are inserted into \(t\). When there is no confusion we use \(M\) instead of \((M, t, t')\). Let \(M\) be a mapping from \(t\) to \(t'\) and let \(U_t\) and \(U_{t'}\) be the set of indices of nodes in \(t\) and \(t'\), respectively, which are not members of any pair in \(M\). Then we can define the cost of \(M\)

\[
\gamma(M) = \sum_{(i, j) \in M} t[i] \leftrightarrow t'[j] + \sum_{i \in U_t} t[i] \leftrightarrow A + \sum_{j \in U_{t'}} A \leftrightarrow t'[j]
\]

The cost is the sum of the costs of all changes, deletes, and inserts.

44
The relation between a mapping and a sequence of edit operations is as follows.

**Lemma 1** Given a sequence \( S = s_1, \dots, s_k \) of edit operations from \( t \) to \( t' \), there exists a mapping \( M \) from \( t \) to \( t' \) such that \( \gamma(M) \leq \gamma(S) \). Conversely, for any mapping \( M \), there exists a sequence of editing operations such that \( \gamma(S) = \gamma(M) \).

Proof. See [ZS89].

Hence, we have the following corollary.

**Corollary 1** We can define the *minimum edit distance* between two trees \( t \) and \( t' \) as

\[
\text{mindist}(t, t') = \min\{\gamma(M) \mid M \text{ is a mapping from } t \text{ to } t'\}
\]

Proof. Follows from Lemma 1.

**Example 12** In Example 11 we saw two edit sequences that changed a tree \( t \) to a tree \( t' \). Assume that the nodes of the trees are ordered in preorder, i.e., the node order of tree \( t \) is \( S, A, B, C \) and the order of tree \( t' \) is \( S, B, C \) (Figure 4.3, nodes are indexed with order number). We have a mapping \( M_1 = \{ (1,1), (2,2), (3,3) \} \). The cost of this mapping is (assume a unit cost function)

\[
\gamma(M_1) = \gamma(t[1] \mapsto t'[1]) + \gamma(t[2] \mapsto t'[2]) + \gamma(t[3] \mapsto t'[3]) + \\
\gamma(t[4] \mapsto \Lambda) = \gamma(S \mapsto S) + \gamma(A \mapsto B) + \gamma(B \mapsto C) + \gamma(C \mapsto \Lambda) = 0 + 1 + 1 + 1 = 3
\]

Another mapping is \( M_2 = \{ (1,1), (3,2), (4,3) \} \). The cost of this mapping is

\[
\gamma(M_2) = \gamma(t[1] \mapsto t'[1]) + \gamma(t[3] \mapsto t'[2]) + \gamma(t[4] \mapsto t'[3]) + \\
\gamma(t[2] \mapsto \Lambda) = \gamma(S \mapsto S) + \gamma(B \mapsto B) + \gamma(C \mapsto C) + \gamma(A \mapsto \Lambda) = 0 + 0 + 0 + 1 = 1
\]
This is less or equal then the cost of any mapping and consequently we have \( \text{mindist}(t, t') = 1 \). Both mappings \( M_1 \) and \( M_2 \) follow the conditions given above. But, e.g., the triple \( \{(1, 2), (2, 1)\}, t, t' \) is not a mapping as there is no conservation of the ancestor order.

The solutions to the tree-to-tree correction problem are based on minimal mappings and several algorithms have been developed for computing the mappings [Tai79, ZS89, SZ90].

### 4.9.3 The preorder distance

The minimum edit distance corresponds to the minimum number of edit operations needed to transform one tree into another. Let \( t \) be a parse tree constructed according to a grammar \( G \) and some input text. When we modify the tree \( t \) because of a modification in the input we get an updated tree \( t' \). The structure of \( t' \) is unknown at the start. We know, however, that the tree \( t' \) follows the productions of the grammar. The work that must be done to construct the updated tree \( t' \) does not relate directly to the minimum edit distance between the trees. Here we define two other distance functions that more accurately measure the reparsing.

We assume the recursive descent parsing technique to be applied for a text that follows a context-free textable grammar. Therefore, it is natural to measure the distance of the trees by counting the mismatches when the trees are traversed in preorder. We consider the same three operations: changes,
inserts and deletes. Informally we define the preorder distance as the minimum cost sequence of edit operations needed to transform tree $t$ into tree $t'$ when we traverse and transform the trees simultaneously in preorder. When a mismatch occurs, the proper edit operation (change, delete or insert) must be applied to correct the mismatch.

In order to define this distance formally, we first define the preorder mapping between two trees. Suppose we have an ordering for each tree (any complete ordering will do). We denote by $t[i]$ the $i$th node of tree $t$ in the given ordering. A complete preorder mapping $(M_{Pr}, t, t')$ is similar to an ordinary mapping, but with the following conditions.

1. for any pair $(i, j) \in M_{Pr}$ we have $1 \leq i \leq |t|$ and $1 \leq j \leq |t'|$

2. $(i, j) \in M_{Pr}$ if and only if
   
   a) $|t| > 0$ and $|t'| > 0$ and $i = j = 1$

   (that is, the root of $t$ is mapped onto the root of $t'$)

   or

   b) there is a pair $(i', j') \in M_{Pr}$ such that

   (a) $t[i']$ is the father of $t[i]$ and

   (b) $t'[j']$ is the father of $t'[j]$ and

   (c) $t[i]$ and $t'[j]$ are both the $n$th child of $t[i']$ and $t'[j']$, respectively.

A complete preorder mapping induces conditions that are much stricter than the conditions of an ordinary mapping. This means that usually more editing actions must be taken when the transformation is done. We define the cost of the preorder mapping $M_{Pr}$ in the same way as the cost of $M$:

$$
\gamma(M_{Pr}) = \sum_{(i,j) \in M_{Pr}} t[i] \leftrightarrow t'[j] + \sum_{i \in U_t} t[i] \leftrightarrow A + \sum_{j \in U_{t'}} A \leftrightarrow t'[j]
$$

Here $U_t$ and $U_{t'}$ are the sets of indices of nodes in $t$ and $t'$, respectively, that are not members of any pair in $M_{Pr}$. Before we define the preorder distance, we give the following lemma.
Lemma 2 The complete preorder mapping is unique.

Proof. The preorder is unique, therefore there can be only one preorder mapping. \hfill \Box

Now we can define the preorder distance formally:

Definition 2 Let \( t \) and \( t' \) be labeled trees. The preorder distance of the trees is

\[
\text{predist}(t, t') = \gamma(M_{\text{Pre}})
\]

where \( M_{\text{Pre}} \) is the complete preorder mapping of \( t \) and \( t' \). \hfill \Box

The function \( \text{predist} \) is metric due to its definition.

The algorithm of the preorder transformation is given as Algorithm 3.

The algorithm uses the following notations. Let \( t \) be a tree. Then \( t.\text{root} \) is the root node of tree \( t \). Let \( n \) be a node in the tree. Then \( n.\text{label} \) is the label of node \( n \), \( n.\text{size} \) the number of the children of node \( n \), and \( n.\text{child}_{i} \) the \( i \)th child of node \( n \). The time complexity of the algorithm is \( O(|t| + |t'|) \). The trees are traversed exactly once each, possibly (nearly all of) the nodes of the tree \( t \) must be removed and new nodes according to the tree \( t' \) inserted which requires a total time of about \( 2(|t| + |t'|) \), under unit cost.

The preorder distance can be determined by the algorithm by counting the number of change, delete and insert operations that the algorithm performs. It can easily be shown that the preorder distance is always greater or equal to the minimum edit distance.

Example 13 In Example 12, we have that \( M_{\text{Pre}} = M_{1} \). \hfill \Box

4.9.4 The parse distance

The preorder distance does not yet take into account the grammar which \( t \) and \( t' \) follow. We define the parse distance as the total size of subtrees that are changed, inserted or deleted when transforming \( t \) into \( t' \). A subtree is considered to have changed if the production expanded at the root of the subtree has been substituted for another production. If the production expanded at the root of tree \( t \) is changed, the parse distance is the sum of
Algorithm 3 *(Preorder transformation)*

<table>
<thead>
<tr>
<th>procedure preorder_transform(t, t')</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: Two subtrees t and t'.</td>
</tr>
<tr>
<td>Output: The subtree t transformed into t'.</td>
</tr>
<tr>
<td>Task: Check and change nodes in preorder.</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>1 n = t.root;</td>
</tr>
<tr>
<td>2 n' = t'.root;</td>
</tr>
<tr>
<td>3 for i = 1 to max(n.size, n'.size) do</td>
</tr>
<tr>
<td>4     if (n'.child_i = nil) then</td>
</tr>
<tr>
<td>5       for all nodes m in n.child_i do</td>
</tr>
<tr>
<td>6           DELETE m;</td>
</tr>
<tr>
<td>7         DELETE n;</td>
</tr>
<tr>
<td>8     else</td>
</tr>
<tr>
<td>9       if (n.child_i = nil) then</td>
</tr>
<tr>
<td>10         INSERT node with label n'.child_i.label</td>
</tr>
<tr>
<td>11             (and no children) as i'th child of n;</td>
</tr>
<tr>
<td>12       else if (n.child_i.root.label ≠ n'.child_i.root.label) then</td>
</tr>
<tr>
<td>13           CHANGE n.child_i.root.label to n'.child_i.root.label;</td>
</tr>
<tr>
<td>14        preorder_transform(n.child_i, n'.child_i);</td>
</tr>
<tr>
<td>end;</td>
</tr>
</tbody>
</table>

the size of t and t'. If the production of the root stays the same we traverse the children from left to right and check the productions expanded at the children. If an expanded production has been changed, we increment the parse distance with the sum of the size of the subtrees at these children.

We define the parse distance by means of a mapping. Suppose we have an ordering for each tree (any complete ordering will do). We denote by t[i]

\[49\]
the $i$th node of tree $t$ in the given ordering. Suppose that the trees follow a certain grammar $G$ and that at every node in a tree a certain production has been expanded. A complete parse mapping $(M_{\text{parse}}, t, t')$ is similar to an ordinary mapping, but with the following conditions.

1. for any pair $(i, j) \in M_{\text{parse}}$ we have $1 \leq i \leq |t|$ and $1 \leq j \leq |t'|$

2. $(i, j) \in M_{\text{parse}}$ if and only if
   
   a) $|t| > 0$ and $|t'| > 0$ and $i = j = 1$ and the productions applied at the roots of the trees are the same
   
   (that is, the root of $t$ is mapped onto the root of $t'$)
   or
   
   b) there is a pair $(i', j') \in M_{\text{parse}}$ such that

   (a) $t[i']$ is the father of $t[i]$ and
   
   (b) $t'[j']$ is the father of $t'[j]$ and
   
   (c) $t[i]$ and $t'[j]$ are both the $n$th child of $t[i']$ and $t'[j']$, respectively
   
   (d) the productions applied at $t[i]$ and $t'[j]$ are the same.

A complete parse mapping induces conditions that are even stricter than the conditions of a complete preorder mapping.

We define the cost of a complete parse mapping $M_{\text{parse}}$ in the same way as the cost of $M$ and $M_{\text{pre}}$:

$$\gamma(M_{\text{parse}}) = \sum_{(i, j) \in M_{\text{parse}}} t[i] \mapsto t'[j] + \sum_{i \in U_t} t[i] \mapsto A + \sum_{j \in U_{t'}} A \mapsto t'[j]$$

Here $U_t$ and $U_{t'}$ are the sets of indices of nodes in $t$ and $t'$, respectively, that are not members of any pair in $M_{\text{parse}}$. There is a mapping from a node in tree $t$ to a node in tree $t'$ only if the same production has been expanded which implies that the labels of the nodes must be the same. Therefore the sum $\sum_{(i, j) \in M_{\text{parse}}} \gamma(t[i] \rightarrow t'[j])$ is zero and the cost of the complete parse mapping is reduced to

$$\gamma(M_{\text{parse}}) = \sum_{i \in U_t} t[i] \mapsto A + \sum_{j \in U_{t'}} A \mapsto t'[j]$$
that is, the sum of the costs of all inserts and deletes. We have

**Lemma 3** The complete parse mapping is unique, if the grammar is unambiguous.

*Proof.* Skipped.

The parse distance is defined accordingly.

**Definition 3** Let $t$ and $t'$ be labeled parse trees over some grammar $G$. The *parse distance* is

$$\text{parsedist}(t, t') = \gamma(M_{\text{Parse}})$$

where $M_{\text{Parse}}$ is the complete parse mapping of $t$ and $t'$.

The function $\text{parsedist}$ is metric due to its definition. The parse transformation algorithm is given as Algorithm 4. The notation is the same as used in Algorithm 3. Additionally, we assume that each node has information about what production has been expanded at the node; this is indicated by $n_{\text{prod}}$.

The *copy* command copies a tree preserving the structure of the tree. This corresponds fairly well to the amount of reparsing done when updating the tree $t$ to get $t'$. When reparsing, assume that we have to change the production at a certain node. This means that we have to continue the recursive descent parsing according to this production. We start by calling the first child of the production, etc. If we do not have to change the production, we still have to traverse the children and check if some of the productions at these nodes change.

If two corresponding nodes in $t$ and $t'$ have a production that contains an iteration on its right side, and the iteration is used to produce a different number of children for the nodes, we assume that the production is the same in both nodes. Consider, e.g., the production $S \rightarrow A^*$ used in corresponding nodes $t[i]$ and $t'[j]$, such that $t[i]$ has two children (two $A$s) and $t'[j]$ has three children (three $A$s). Then the productions used at both nodes are the same, and so are (probably) the productions used at the two first children of the nodes. Only the third child differs, which is then the only difference affecting the cost function.

**Example 14** Assume that we have the same trees as in Example 12. The $\text{mindist}$ and $\text{predist}$ functions do not take any grammar into account. For the $\text{parsedist}$ function, we need the grammar of the parse trees (the same
grammar for both trees). Assume that the grammar contains the productions $S \rightarrow ABC$ and $S \rightarrow BC$. The trees in Figure 4.3 must be constructed with two separate productions, because the grammar is supposed to be LL(1). The complete parse mapping is empty, because the productions expanded at the roots of trees $t$ and $t'$ are different. The cost of the complete parse mapping $M_{parse}$ is (assume again unit cost)

$$\gamma(M_{parse}) = \gamma(t[1] \rightarrow \Lambda) + \gamma(t[2] \rightarrow \Lambda) + \gamma(t[3] \rightarrow \Lambda) + \gamma(t[4] \rightarrow \Lambda) + \gamma(A \rightarrow t'[1]) + \gamma(B \rightarrow t'[2]) + \gamma(C \rightarrow t'[3])$$

$$= \gamma(S \rightarrow \Lambda) + \gamma(A \rightarrow \Lambda) + \gamma(B \rightarrow \Lambda) + \gamma(C \rightarrow \Lambda) + \gamma(A \rightarrow B) + \gamma(A \rightarrow C)$$

$$= 1 + 1 + 1 + 1 + 1 + 1$$

$$= 7$$

Now, $parsedist(t, t') > predist(t, t')$ because the production expanded at the root is different in $t$ and $t'$. For this pair of trees we have $mindist(t, t') < predist(t, t') < parsedist(t, t')$.

We can show that

**Lemma 4** For any pair of labeled trees $t$ and $t'$ we have

$$mindist(t, t') \leq predist(t, t')$$

and if the trees are constructed over a certain unambiguous grammar (grammars) we have

$$mindist(t, t') \leq predist(t, t') \leq parsedist(t, t')$$

**Proof.** From discussion above.

Generally we do not expect the distances to be equal.
Algorithm 4 (Parse transformation)

procedure parsing_transform(t, t')
Input: Two subtrees t and t'.
Output: The subtree t transformed into t'.
Task: Parse transform t into t'.
begin
1   n = t.root ;
2   n' = t'.root ;
3   for i = 1 to max(n.size, n’.size) do
4       if (n’.child_i = nil) then
5           for all nodes m in n.child_i do
6               DELETE m;
7           DELETE n;
8       else if (n.child_i = nil) then
9             /* INSERT nodes of t’.child_i into t.child_i */
10            Copy t’.child_i as i’th child of t
11       else if n.child_i.root.prodn ≠
12          n’child_i.root.prodn then
13             for all nodes m in t do
14               DELETE m;
15           DELETE n;
16             /* INSERT nodes of t’.child_i into t.child_i */
17            Copy t’.child_i as i’th child of t
18       else
19           parsing_transform(n.child_i, n’.child_i);
end;
Chapter 5

Incremental parsing

A parser structures a text forming a parse tree over a grammar. If we modify the text, the parser updates the tree to correspond to the modifications. The parse-from-scratch technique updates the tree by reparsing the entire text. If the text is large, this technique can be very time-consuming. **Incremental parsing**, however, reuses the old parse tree modifying it only where it should be updated. We use parsing for capturing the structure of text documents. Incremental parsing brings efficiency to the structuring.

5.1 Related work

Themes connected to incremental parsing are present already in [Loc65, Lin70]. Research on incremental parsing handles both LR parsing [Cel78, GM79, GM80, Weg80, JG82, AD83, DMM88, Lar90], and LL or recursive descent parsing [Kah79, Str82, SDB84, MPS90]. We distinguish incremental parsing from incremental generation of parsers [Kos90, GHKT88], where a modification in the grammar leads to an incremental update of the parser.

In the HST system [KLMN90] we use a recursive descent parsing technique for capturing the structure of documents. We shall, therefore, take a closer look at some incremental LL parsers. All techniques start by parsing the input text. They differ, however, in the way they update the parse tree after a modification in the input.

Strömb erg [Str82] presents a way to incrementally parse (Pascal) programs. Two attributes are associated with every node in the parse tree. One
tells the length of the frontier of the subtree for the node and the other
tells the length of the reserved word that may be associated with the node.
When an insertion occurs in the input, the incremental parser obtains the
insert position and the length of the inserted text as parameters. The parser
then traverses the parse tree top-down looking for a modified node $n$. The
node $n$ is found by comparing the insert position with the cumulated sum of
the lengths of subtree frontiers and reserved words depending on how far the
traversal has proceeded in the tree. When the right node has been found, the
inserted text is parsed and inserted into the tree. The parser then updates
the length attributes of the ancestor nodes of $n$. If parsing fails, the parser
returns to the parent of the node $n$ and tries again. This is a weak point of
the incremental parser; if the parser backtracks far enough, it must reparse
the entire text.

Schwartz et al. [SDB84, DMS84, MS81] present another incremental re-
cursive descent parser for Pascal. When the program is modified, the parser
splits the parse tree into a sequence of parse subtrees. The subtree corre-
sponding to the modification is removed, possible insertions are parsed, and
the sequence of parse subtrees is joined to form the complete parse tree. The
fragmentation technique used depends on language constructs and there-
fore this sort of incremental parser cannot be constructed automatically (as
pointed out by [MPS90]).

Murching et al. [MPS90] present an incremental parser that can be auto-
matically generated for any LL(1) grammar. The parser forms a parse tree
and connects the input tokens with the leaves of the tree. The input tokens
are also connected, forming a list. When the input is modified, the parser up-
dates the list by marking some of the tokens deleted, inserted, or unchanged.
The parser traverses the token list and updates the parse tree. It finds the
modified nodes in the parse tree by following links from the input tokens to
the leaves of the tree. We extend this technique and present an incremental
parser for structured documents. The extended method is presented in the
following sections.

5.2 Incremental Parsing

Incremental parsing aims at minimizing the work performed in a parse tree
update. Let $G = (N, \Sigma, P, S)$ be a context-free unambiguous grammar and
let \( x \) be a string of the language \( L(\Sigma) \). Let \( t \) be the parse tree over \( G \) with frontier \( x \). Assume that we modify the string \( x \) obtaining the new input string \( x' \). The parse-from-scratch technique reads the entire string \( x' \) and reconstructs the parse tree giving \( t' \). The incremental technique, however, reuses the old parse tree \( t \), updates it, and produces a modified parse tree \( t' \) (Figure 5.1).

An optimal incremental parser traverses and updates the parse tree only where it should be modified. A lower bound for an incremental parser is \( \text{mindist}(t, t') \), the cost of the edit sequence required to transform tree \( t \) into tree \( t' \). We do not expect to reach this bound. A more practical bound will lie somewhere between \( \text{predist}(t, t') \) and \( \text{parsedist}(t, t') \).

### 5.3 Data structure support

The parse-from-scratch technique always rebuilds the complete parse tree; there is no need to express the associations between the tree and the input string. Incremental parsers, however, have to connect the input with the
parse tree. When the input is modified, the parser efficiently finds the affected nodes in the parse tree.

While the input string is being parsed, the parser connects pieces of it with corresponding leaf nodes in the parse tree. The parser divides the input string into *input tokens*. An input token is determined by the textable grammar, and is either a constant string or a variable string. Every input token is *tagged*, determined by a *start tag* and an *end tag*. The tags are not visible to the user. The tags are needed by the incremental parser for recognizing the structure of the text.

Every input token is connected to a *parse tree token*, the corresponding leaf node in the parse tree. We say that the input token and the parse tree token are *associated*. We connect the input tokens to the parse tree tokens only at the leaf level. The input tokens corresponding to a subtree are determined by the associated parse tree tokens of the subtree. Associated tokens are connected with *links* that are bidirectional.

**Example 15** Assume that we have two parse trees corresponding to the productions *Author* → *Text* and *Year* → *e*, respectively. The first tree consists of a nonterminal node, *Author*, a parse tree token, *Fletcher*, and an input token, *Fletcher*, which is the piece of string that is associated with the tree. Note that we distinguish input tokens from parse tree tokens by framing them. The second tree is similar; it consists of a nonterminal node, *Year*, a parse tree token, *e*, and an input token, *e*, which in this case is an epsilon token (empty string). Epsilon tokens are tagged but not visible to the user.

```
    Author
      |   
    Fletcher
      ↑
  Fletcher

    Year
      |   
      e
      ↓
    e
```

For further examples of tagged text see Figure 5.2 of Example 16 (page 61).

We see that every input token is connected to its associated parse tree token. This is true only when the input string has been parsed. An unparsed string has no associations.
5.3.1 Editor support

Tagged text can be supported by the editor in different ways. We have used the Sun Microsystems Textedit Tool in OpenWindows [Sun90] which automatically supports invisible tags in the text. The XView developing environment [Hel90] offers some procedures for maintaining, creating, destroying, and listing tags in the text. In the example above assume that the epsilon node $\epsilon$ is the following leaf node after the node Fletcher in left to right order. In the editor we would have the following text.

\begin{verbatim}
Fletche r
↑↑
t_1  t_2  t_3 t_4
\end{verbatim}

There are four tags inserted, two ($t_1$ and $t_2$) delimiting the terminal string Fletcher, and two ($t_3$ and $t_4$) delimiting the (invisible) epsilon token. The HST system [KLMN90] maintains a table linking a pair of tags to the corresponding parse tree token. In the example we would have the following table.

\begin{verbatim}
Start Tag  End Tag  Link to Node
  t_1  t_2  123
  t_3  t_4  456
\end{verbatim}

The numbers 123 and 456 are links to the Fletcher and $\epsilon$ nodes, respectively.

We are of course not limited to editors that support invisible tags in the text. The document preparation system in hand can explicitly maintain the tags itself. We could also use a representation that explicitly shows the tags in the text. An example is the following.

\begin{verbatim}
<Author/#123>Fletche r</Author> <Year/#456></Year>
\end{verbatim}

The input tokens are delimited by SGML-like tags [Bar89] augmented with links to the corresponding parse tree tokens. This representation is not very user friendly as links are interspersed with ordinary text. Additionally, there can be problems with the consistency between links and the parse tree if the user is allowed to modify the links.
5.3.2 Parse tree notation

We associate some information with each tree node in a parse tree. Let \( t \) be a parse tree and \( n \) a node in it.

- \( t.root \): the root node of tree \( t \)
- \( n.size \): number of children of node \( n \)
- \( n.child_i \): \( i \)'th child (subtree) of node \( n \)
- \( n.label \): label of node \( n \) (left side of production expanded at \( n \))
- \( n.prod \): production expanded at node \( n \)
- \( n.string \): constant or variable string connected with leaf node \( n \)
- \( prod.symbol_i.label \): label of \( i \)'th symbol on the right hand side of production

The attribute \( n.size \) denotes the number of the children of node \( n \). A leaf node has always size 0. The attribute \( n.child_i \) denotes the \( i \)'th child of node \( n \). The label of node \( n \) is denoted by \( n.label \). The expanded production at \( n \) is denoted by \( n.prod \). All the leaf nodes \( n \) (parse tree tokens) contain a string denoted by \( n.string \). The notation \( prod.child_i.label \) refers to the labels of non-terminals or to the terminals on the right hand side of a production \( prod \). If \( n \) is a node in a parse tree, we have \( n.child_i.root.label = n.prod.symbol_i.label \).

5.4 Start position

Incremental parsing divides into three subproblems. The parser must first decide where to start reparsing. It must determine which node in the tree is the first to be modified. The second subproblem is the method: how will the parser proceed updating the parse tree efficiently. For the third, a parser must know when it can terminate. In this section, we motivate the choice of the start position. In the following section we define the termination conditions, before going into the method.

We locate the start position as in [MPS90]. Let \( x = vwz \) be the old input string, where \( v, w, z \in T^* \) and let \( x' = vw'z \), where \( w' \in T^* \) be the modified string. Let \( t \) be the parse tree corresponding to the string \( x \). The old input string \( x \) does not exist any more; it has been replaced by the new input...
string \( x' \). However, the old parse tree \( t \) still corresponds to the string \( x \). The substring \( v \) has not been modified; the part of the parse tree corresponding to this string should, in consequence, not change. The parser, however, needs to look ahead one symbol before choosing which production to expand at a certain node. The string \( w' \) can be preceded by epsilon tokens that are not part of the unmodified string \( v \). In order to capture such possible epsilons, the incremental parser starts reparsing at \( \text{Last}(v) \), i.e., the last input token of the string \( v \).

From the last input token of \( v \) we follow the link to the associated parse tree token in the parse tree. The comparison between the parse tree and the input starts at these two tokens (see Section 5.6.1 for details).

## 5.5 Termination conditions

Determining the termination position of a reparse is not straightforward. A simple modification in the input text can sometimes affect the entire parse tree. We give here termination conditions that indicate when the parser can stop updating the parse tree. If these conditions hold we claim that the parse tree is consistent with the modified input, even if we have not traversed the input string totally.

Termination conditions for incremental LR parsing have been presented by [Weg80, JG82, Yeh83, YK88]. We present some conditions that concern incremental LL parsing. Some further enhancements can be made by applying the skipping heuristic presented in [YK88] (but introduced by [Weg80]), where parts of parsing are skipped even if the parser has not passed the last edit change in the text.

### 5.5.1 Local synchronizations

The termination conditions are divided into two concepts, local synchronizations and possible termination nodes. We show that reparsing can terminate if a local synchronization holds at a possible termination node.

Assume that we have an input string, the corresponding parse tree and links between associated input and parse tree tokens. Assume also that the input has been modified and that the parse tree is not up to date.
**Definition 4** We say that all unmodified input tokens are *locally synchronized* with their associated parse tree tokens. All modified input tokens, or inserted text that does not belong to an input token, are not synchronized.

We use the term locally synchronized because other parts of the parse tree and the input string can still be inconsistent.

**Example 16** In Figure 5.2 we have an example of a parse tree with associated input tokens. The parse tree is constructed over Grammar 2 (given in Example 4 on page 17).

The parse tree represents the input string **Fletcher**:Murder,. Police, 1990. All input tokens are locally synchronized with their associated parse tree tokens. Now, assume that we modify the input string, e.g., substituting the string **Police** by **Crook**. Then the new input token **Crook** maintains its link to the parse tree token **Police**. The tokens are, of course, not synchronized. All the other input tokens are locally synchronized with their associated parse tree tokens (Figure 5.3).
A local synchronization says that if a tagged input token has been parsed before, the parser does not have to update the parse tree at the associated parse tree token because it is already correct. A local synchronization between two tokens is not a sufficient termination condition. There are two reasons for this. First, if the parser has not passed the last modified input token, there will most certainly be other modifications in the parse tree. Second, the structure of the tree may have changed due to the update; this means that the parse tree may be restructured even after the last input modification.

**Example 17** Consider a grammar $G$ with productions $P = \{ S \to ABCD \mid \text{EBFG}, A \to a, B \to b, C \to c, D \to d, E \to e, F \to cd \mid f, G \to \epsilon \}$. The grammar is LL(1) and unambiguous. We can derive three different terminal strings, $abcd, ebcd, \text{and ebf}$.

We shall show why a local synchronization is not a sufficient termination condition. First, we show that the parser must proceed until it has passed the last modified input token. Assume that we have a parse tree for the string $abcd$ (Figure 5.4, tree $t$). Assume also that we change the terminal

![Figure 5.3: A parse tree with with one token not synchronized with the input token list.](image)

```plaintext
List
  Publications
    ...
        Publication ...

    Author
    Title
    Journal Year

Fletcher Murder Police, 1990

Fletcher Murder Crook 1990
```
string $a$ to $e$ and the terminal string $c d$ to $f$. The corresponding tree is given as tree $t'$ in Figure 5.4. The only locally synchronized node, $B$, and the corresponding terminal string, $b$, have been surrounded by a dashed box. We start updating the tree $t$ by parsing the string $e$. If we stop when we reach the local synchronization, the rest of the tree will never be updated (nodes $F$ and $G$).

Second, we show that a local synchronization is not a sufficient termination condition because the structure of the tree might change because of an earlier modification. Assume that we start from the tree $t$ in Figure 5.4 and change the terminal string $a$ to $e$. This time, the last modified input token is the token corresponding to the parse tree token $e$ in tree $t''$. But the parser cannot stop when it reaches the local synchronization at node $b$. Even if there has been no modifications among the input tokens after the string $b$, the tree is still changed.

We can extend the definition of local synchronization for a sequence of input tokens. A sequence of input tokens is locally synchronized with a parse subtree, if each input token is synchronized with a parse tree token in the subtree and the input tokens occur in the same order as the parse tree tokens in the frontier of the parse subtree.
5.5.2 Termination nodes

We now define a property of a node in the parse that says that it is possible to stop the reparsing there. This property depends on the history of the node, that is, on the development of its ancestors, its right brothers and its right ancestors (right brothers of its ancestors).

**Definition 5** A node $n$ is a *possible termination node* for reparsing if during the reparsing

a) no production expanded at an ancestor of $n$ is changed
or
b) no label of a right ancestor or a right brother is changed
and no right brother or right ancestor of $n$ is deleted or inserted

We should make two important comments on this definition. First, condition a) does not imply condition b) or vice versa. Second, because reparsing is done top-down and the production to be expanded at a node is chosen before the parser reaches the children of the node, we always know (1) if an ancestor has changed its production, and (2) if an ancestor or a right ancestor has changed its label.

**Example 18** In Figure 5.5 we see a parse tree $t$ and its modification $t'$ after a reparse. The grammar contains the two productions $S \rightarrow ABC$ and $S \rightarrow BC$ and satisfies the LL(1) conditions. We study the node $B$ in tree $t'$. The production expanded at the root of $t$ has changed, so condition a) in the definition above does not hold at $B$. But no ancestor, right ancestor or right brother of $B$ has changed its label and therefore it is a possible termination node. Also nodes $S$ and $C$ are possible termination nodes.

The following lemma gives sufficient conditions for stopping a reparse and still producing a correct new parse tree for the modified text.

**Lemma 5** Let $G$ be a context-free, unambiguous LL(1) grammar. Assume that we have an input string $x = v w z$ where $v, w, z \in T^*$, its parse tree $t$ over
the grammar $G$ and links between the associated tokens in the input and the parse tree. Assume that the input is modified to $x' = vw'z$ where $w' \in T^*$ and that we reparse it. The parser starts at the last input token of string $v$. The parser can stop the reparsing if the following termination conditions hold

a) The last modified input token has been parsed.
b) Parsing is locally synchronized.
c) The parser has reached a possible termination node.

Proof. The proof should be obvious and we will only informally sketch it. Assume that the three conditions given above hold. If the last modified input token has been parsed, the parser must be processing the string $z$. This string has been parsed before. Additionally we know that the structure of the tree concerning the string $z$ has not changed because the parser has reached a possible termination node. We also know that the parser has found a local synchronization. Because the grammar is LL(1) and unambiguous, there can be no other changes in the tree after this local synchronization. $
$\end{proof}

### 5.6 Incremental parsing: the method

In this section we concentrate on the parsing method. We extend the technique presented by Murching et al. [MPS90]. They give a top-down incre-
mental parsing algorithm which can be automatically constructed from a context-free grammar. We give four extensions. First, we let the parser handle also variable strings. Second, we use the termination conditions of Section 5.5 to stop the incremental parsing. Third, we also give an enhancement of the algorithm that uses termination conditions for locally skipping unmodified text strings during an incremental parse. Fourth, we discuss iterations and their update problems.

5.6.1 Scenario of the reparse

Before we present the incremental algorithms, we take an informal look on the method. We start by some notations. Let \( x = vwz \) be the old input string, where \( v, w, z \in T^* \) and let \( x' = vw'z \) be the modified string, where \( w' \in T^* \). The old string \( x \) does not exist any more; it has been replaced in the editor by the new string \( x' \). However, before the old parse tree is updated, we can reconstruct the old string \( x \) by concatenating the strings in the parse tree tokens (the leaves). Let \( t \) be the old parse tree corresponding to the string \( x \). The incremental parsing algorithm updates the tree \( t \) to a parse tree \( t^\prime \).

The incremental parsing algorithm traverses the old parse tree and the new input string simultaneously, updating the tree \( t \) when necessary. Traversing is done with the help of two pointers, oldptr that points to parse tree tokens in tree \( t \) and newptr that points to input tokens in the string \( x' \). Because newly inserted text is not divided into input tokens before the text has been parsed, newptr can point to a certain inserted character. During parsing, oldptr points to the tree node that is currently being updated and newptr points to the next character being read from the input.

Incremental parsing starts with the initialization of oldptr and newptr. Then the parser calls the function Traverse for traversal of the old parse tree simultaneously with reading the new input string. The Traverse function first determines the start position (ClimbUpTree) and then calls the appropriate parse functions (Parse, Parsee, ParseT, or ParseE, see Figure 5.6).

The parser sets the pointers to point to \( \text{Last}(v) \) of \( x \) and \( x' \) respectively, i.e., oldptr is set to point to the parse tree token that comes directly before the first one to be modified, and newptr is set to the last unmodified input token before the first modified token. The pointers are set this way because of possible epsilon tokens in the parse tree and the editor. Because the structure
of the parse tree is inherent in the editor (through the input tokens), invisible epsilon tokens must be inserted into the editor. This is done solely by the parser when needed for consistency with the parse tree. If there are epsilon tokens following $Last(v)$ among the editor tokens, the number of these must be made equal with the number of epsilon tokens at the corresponding place in the parse tree.

When the number of epsilon tokens has been made equal in the editor and in the parse tree (Algorithm 5, lines 3–11), the actual reparsing can start. Both pointers, $oldptr$ and $newptr$ are advanced to the following parse tree token and input token, respectively. If the strings of the associated tokens differ, the text has been modified and must be reparsed. The question of which parse function to use is solved in the following way. The parse tree is traversed upwards from the $oldptr$ token until a node is reached that is not the leftmost child of its parent. This is the first node (in a top-down traversal) where the expanded production can change. The subtree at this node is reparsed. After the reparse, $oldptr$ is advanced beyond the yield of the old subtree. The pointer $newptr$ is advanced automatically when reading input.

The traversal of the parse tree continues, gradually changing it to the new tree $t'$. The pointer $oldptr$ is advanced, a start position is found by traversing the tree upwards from the $oldptr$ token to a node that is not the leftmost child of its parent, and the correct parsing function is called. The update can stop when a termination node has been reached.

The parsing functions need some further explaining. A parsing function

![Diagram](image-url)
recursively parses the new input text. However, it tries to avoid reparsing unmodified parts of the parse tree. Nonterminals, constant strings, variable strings, and the empty string all have slightly different functions. Assume that a parse function of a certain nonterminal is called when the traversal of the parse tree has reached (should reach) node \( n \). If the node \( n \) is missing, it is inserted, in addition to the children according to the production expanded at the node. If the production expanded at \( n \) is changed, new children are inserted. All these children must be reparsed.

If the production expanded at \( n \) does not change, parsing can continue in two ways. If the next input token is different from the next parse tree token, then either some node in the leftmost branch of the subtree at \( n \) changes its production, or the production at \( n \) is textable. (This is different from the corresponding algorithm in [MPS90], where there are no variable strings.) In these both cases the leftmost branch of the subtree is traversed and reparsed. The rest of the tree is searched for further modifications. If the input token and the parse tree token are locally synchronized, the leftmost branch of the tree does not change, but the rest of the subtree must be traversed because it might have to be be modified.

A traversal of a subtree is equivalent to the top level traversal explained above. This traversal is referred to as \( \text{traverse.yield.of.tree}(\text{tree}) \).

### 5.6.2 Algorithms for incremental parsing

The incremental parsing functions are given in Algorithms 6–9. The top level traversal algorithm is given in Algorithm 5. Some auxiliary functions are briefly explained.

**Traverse(oldptr, newptr)** reads the input from the first modification point until a termination node is reached. The function expects that \( \text{oldptr} \) and \( \text{newptr} \) have been set to accurate parse tree and input tokens. The Traverse function first balances the number of epsilon tokens in the parse tree and the input. Thereafter, it updates the parse tree at the \( \text{oldptr} \) node according to the input at \( \text{newptr} \). The function is given in Algorithm 5.

**Traverse.yield.of.tree(tree)** is similar to the Traverse function. This function traverses a subtree, updating it due to modifications in the input.
Algorithm 5 *(Incremental update traverse)*

```plaintext
procedure Traverse (oldptr, newptr: pointer);
Input: oldptr and newptr set to Last(v) in x and x'.
Task: Traverse parse tree and update it.
begin
  while not termination node do
    if Next(Lookahead(oldptr)) then /* equate eps*/
      while Symbol(oldptr) = ε do
        if Symbol(newptr) = ε then
          Link Symbol(newptr) to parse tree;
          Advance_newptr;
        else
          Insert epsilon token and link;
          Advance_oldptr;
    while Symbol(newptr) = ε do
      Delete epsilon token and Advance_newptr;
      Link Scan(Lookahead(oldptr)) to tree ;
      Advance_oldptr;
    else /* Lookahead(oldptr) <> Next(newptr) */
      n := ClimbUpTree(oldptr);
      Parse_n.label(n, Follow(n.label));
  end;
end;
```

Parse\(X\)(tree, followset). For each nonterminal \(X\) in the grammar there is a parse function \(Parse_X\)(tree, followset). The function expands a production with left hand side \(X\) at the root \(n\) of the tree.

First, if no tree node corresponding to the nonterminal exists, a new tree node with label \(X\) is inserted. This may be the case if the production expanded at a node has been changed. If the new production has more children than the old one, additional children must be inserted. Depending on the productions the node may or may not have the same number of children after the change. Among the productions with \(X\) on the left hand side, the parser finds the production whose Dirsym set
Algorithm 6 \textit{(Incremental top-down parsing of nonterminal)}

\begin{algorithm}
\begin{algorithmic}
\Procedure{Parse}_X \text{(t: tree\_node; folset: strings)} \EndProcedure
\begin{itemize}
\item \textbf{Input:} Tree node \text{t} and follow set \text{folset}.
\item \textbf{Output:} Parse tree with root label X.
\item \textbf{Task:} Recursively parse input.
\end{itemize}
\begin{algorithmic}
\Begin
\If {\text{t} = \text{nil}} \Then \text{insert node with label } X; \EndIf
\State \text{prod} := \text{find production with lhs } X, \text{ whose DirSym set}
\State \text{contains next input token (possibly textable)};
\If {\text{t} \text{ is new or production of } \text{t} \text{ changed}} \Then
\ForAll {children \text{t}.child\text{\textsubscript{i}}}
\State \text{folset\text{\textsubscript{i}}} := \text{folset} \cup \text{Follow(\text{prod}.child\text{\textsubscript{i}}.label)};
\EndFor
\Else /* \text{t}.prod \neq \text{prod} */
\If {\text{Next(Lookahead(oldptr))}} \Then
\State \text{Traverse yield of } \text{t};
\EndIf
\Else
\State \text{folset\text{\textsubscript{i}}} := \text{folset} \cup \text{Follow(\text{prod}.child\text{\textsubscript{i}}.label)};
\State \text{Parse}_{\text{prod}.child\text{\textsubscript{i}}.label}(\text{t}.child\text{\textsubscript{i}}, \text{folset}\text{\textsubscript{i}});
\State \text{Traverse yield of } \text{t};
\EndIf
\State \text{Remove excess subtrees};
\End
\end{algorithmic}
\end{algorithm}
\end{algorithm}

the next input token belongs to. If one of the productions is textable, it is chosen as the last possible one. If no appropriate production is found, there is an error in the input.

The chosen production is expanded in one of the two following ways.

1. If the tree node \text{\text{\text{\text{\text{\text{\text{n}}}}}}}} \text{is newly created or if the old production expanded at } \text{n} \text{ is different from the new one, node } \text{n} \text{ is changed to represent the new production. The subtrees of the node are validated to represent the new production by calling the functions corresponding to the right hand side symbols of the production.
The pointer oldptr is advanced beyond the yield of the tree $t$. If the old production had more symbols on the right hand side than the new production, the parser deletes all the excess subtrees of the tree node.

2. If there is no change in the production expanded at tree node $n$, there are two possible cases.

   (a) The lookahead symbol of $oldptr$ is the same as the next input token. Then the leftmost branch of the tree will not change, but it is possible that some node within the subtree needs modification. Therefore, the parser traverses the yield of the entire subtree ($\text{Traverse} \cdot \text{yield} \cdot \text{of} \cdot \text{tree}(tree)$).

   (b) The lookahead symbol of $oldptr$ is different from the next editor token. This indicates that there is a change in some production in the leftmost branch of the subtree. Another possibility is that the nonterminal $X$ is textable. The parser calls the function corresponding to the first symbol on the right hand side of the production. Any other modifications in the other subtrees of the tree are taken care of by traversing the yield of the rest of the tree ($\text{Traverse} \cdot \text{yield} \cdot \text{of} \cdot \text{tree}(tree)$).

The parse functions are given a follow set as a parameter. There are two reasons for this. First, because the Text nonterminal is context-sensitive the parser must know when to stop parsing a certain Text instance. The text is scanned until one of the terminals in the follow set is encountered (see also the Parse$_{T}$ function below). Second, even if there is no error recovery included at the moment, there are plans to include it later (see Section 5.8). The Parse$_X$ function is given as Algorithm 6.

Parse$_c$(tree, followset). There is a function Parse$_c$(tree, followset) for every constant string $c$ ($\neq \epsilon$). The function inserts a node, if none exists. If the next input token is not $c$, we have an error in the input. Otherwise newptr is advanced to the character (the next input token) following the scanned token and the constant string is added as an attribute to the node. All excess children of the tree are deleted. The pointer $oldptr$ is advanced beyond the yield of the old subtree $t$. The Parse$_c$ function is given as Algorithm 7.
Algorithm 7 *(Incremental top-down parsing of constant string)*

<table>
<thead>
<tr>
<th>Procedure: Parse (_c) ((t): tree node; followset: strings);</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Tree node (t).</td>
</tr>
<tr>
<td><strong>Output:</strong> Parsed tree with label String and constant string as frontier.</td>
</tr>
<tr>
<td><strong>Task:</strong> Read input and form string until a symbol in followset is detected. If next symbol differs from constant string then error.</td>
</tr>
<tr>
<td><strong>begin</strong></td>
</tr>
<tr>
<td>1. if (t = \text{nil}) then</td>
</tr>
<tr>
<td>2. (t := \text{insert tree node with label String};)</td>
</tr>
<tr>
<td>3. else</td>
</tr>
<tr>
<td>4. Delete children of (t);</td>
</tr>
<tr>
<td>5. if Next(c) then</td>
</tr>
<tr>
<td>6. (t.\text{string} := \text{Scan until(followset)};)</td>
</tr>
<tr>
<td>7. else</td>
</tr>
<tr>
<td>8. Error;</td>
</tr>
<tr>
<td>9. Advance oldptr beyond yield of old tree(t);</td>
</tr>
<tr>
<td><strong>end</strong>;</td>
</tr>
</tbody>
</table>

**Parse** \(_\text{text}\) (tree, followset). There is a function **Parse** \(_\text{text}\_\epsilon\) (tree, followset) for parsing variable strings. This is an extension to the technique of Murching et al. [MPS90]. If there is no leaf node corresponding to the variable string, the parser inserts one. Then the input text is scanned until one of the strings in followset is discovered. The scanned text string is added as a child to the Text node (cf. **Parse** \(_\epsilon\)). The **Parse** \(_\text{text}\) function is given as Algorithm 8.

**Parse** \(_\epsilon\) (tree). The function **Parse** \(_\epsilon\) (tree) differs from **Parse** \(_\epsilon\) in the following respect. If \(newptr\) is pointing to an epsilon input token, only the links are updated. Otherwise a new epsilon token is inserted to the left of \(newptr\). The **Parse** \(_\epsilon\) function is given as Algorithm 9.

Some other functions needed in the process are briefly explained below.
Algorithm 8 *(Incremental top-down parsing of variable string)*

```
procedure ParseText(t: tree_node; followset: strings);
Input: Tree node t.
Output: Parse tree with root label Text and variable string as frontier.
Task: Read input and form string until a symbol in followset is detected.
begin
  if t /= nil then
    Insert tree node with label Text;
  else
    Delete the children of t;
  str := Scan_until(followset);
  Add str as an attribute to t with label String;
  Advance_oldptr_beyond_yield_of_old_tree(t);
end;
```

`Advance_oldptr` moves `oldptr` forward to the next parse tree token in the parse tree. If there are no tokens, `oldptr` is set to nil.

`Advance_oldptr_beyond_yield_of_old_tree(t)` moves `oldptr` to the parse token following subtree `t`.

`Advance_newptr` moves `newptr` forwards in the input text. If the following input token is an old epsilon token, the parser sets `newptr` to point to this token; otherwise it is set to point to the following token (the next visible character that follows the token that was scanned when `newptr` pointed to the previous token).

`ClimbUpTree(oldptr)` finds a suitable start node for the reparsing. The function goes along the link from the parse tree token `oldptr` is pointing to and follows the ancestors upwards of this node until it finds a node that is not the leftmost child of its parent or until it reaches the root node.
Algorithm 9 *(Incremental top-down parsing of empty string)*

```plaintext
procedure Parseε(t: tree_node);  
Input: Tree node t.  
Output: Parse tree with root label Eps and  
        one epsilon child.  
Task: Insert epsilon child at t.  
begin  
1    if t = nil then  
2      t := create tree node with label Epsilon;  
3      Attach t to parent node;  
4    else  
5      Delete children of t;  
6      if Symbol(newptr) = ε then  
7        Update links from token to tree;  
8        Advance_newptr;  
9      else  
10     Insert new epsilon token to the left  
11     of newptr and update links;  
end;
```

Lexical analysis within the parser is important when dealing with variable grammars. Variable strings can be checked and read by scanning the input until a certain string is encountered (with the functions `Next_until` and `Scan_until`). We need the following lexical functions.

**Lookahead(oldptr)** returns the contents of the terminal node `oldptr` is pointing to. If it is an epsilon node, the contents of the following non empty leaf node is returned.

**Symbol(oldptr)** returns the token (string) that `oldptr` is pointing to. If `oldptr` is pointing to an epsilon token, this token is returned.

**Symbol(newptr)** returns the token (string) that `newptr` is pointing to. If `newptr` is pointing to an epsilon token, this token is returned.
Next$(Str)$ checks if the string starting at $newptr$ starts with the string $Str$. Does not move $newptr$.

Scan$(Str)$ scans the string $Str$ in the input and moves $newptr$ past the string $Str$.

Next\_until$(Strset)$ reads the input until a string in the set $Strset$ is encountered. The pointer $newptr$ is not moved.

Scan\_until$(Strset)$ reads the input until a string in the set $Strset$ is encountered. The pointer $newptr$ is moved past the scanned string.

For an example of incremental parsing, see Section 5.11. A part of this incremental parser has been implemented. See Section 8 for further details.

### 5.7 Iterations

In this section we study how to update iterations efficiently. An iteration production $S \rightarrow \alpha \beta^* \gamma$ contains an iteration part $\beta$ that can be repeated when the production is expanded. Consequently, an iteration production is expanded at an iteration node, which as its children gets iteration subtrees.

The algorithms in Section 5.6.2 do not handle iterations efficiently. When an iteration subtree is inserted or deleted at an iteration node, the incremental parser reparses all subsequent iteration subtrees. No subsequent iteration subtree is locally synchronized because $oldptr$ and $newptr$ are out of phase.

In practice, we mostly handle iteration productions of type $S \rightarrow A^*$, where the iteration part consists of one nonterminal. Every other iteration production can be rewritten in this form. We give three solutions to the problem of updating an iteration node.

First, consider the insertion or deletion of an iteration part as a change of production. The incremental parser reparses all subtrees with changed productions, thereby reparsing all iteration subtrees. The extra work makes this approach less efficient than the method used by the incremental parser in the previous section.

Second, let the parser reparse all iteration subtrees after the first deleted or inserted one. The parser produces new subtrees or deletes old subtrees at the end of the iteration node. The parser also knows the iteration part and how to build new iteration subtrees.
Third, let the parser reparse only the changed, inserted or deleted iteration subtrees. We take a closer look on this approach, which is more optimal than the other two.

5.7.1 A change in an iteration subtree

The incremental parser presented in Section 5.6.2 handles all changes in the parse tree efficiently. A modified iteration subtree is updated according to the changes in the input. The parser continues until it reaches a termination node. A change in an iteration subtree can, however, affect the rest of the tree; and the rest of the input might have to be completely reparsed.

5.7.2 An insertion/deletion of an iteration part

Consider first the deletion of an iteration part. The old input tokens that constituted the deleted iteration part have been tagged. When the input tokens are removed, the links must remain. The parser finds the nodes of the deleted iteration parts and removes them from the parse tree. When reaching an unchanged iteration part (input token), the parser notices a local synchronization. If there are no more modified input tokens, parsing can terminate.

When an instance of an iteration part has been inserted into the text, there is no tagged text marking the iteration subtree in the parse tree. Therefore the lack of links tells the parser that there is an insertion. When the insertion is parsed, the parser has to know what the iteration part is. For example, it could call the function of the parse function of the iteration production with a parameter telling from which child to start parsing. The parser continues until a termination node is reached. The termination node is not reached as long as inserted text is parsed.

5.7.3 Insertions, deletions and changes combined

The update of a change in an iteration subtree differs from the update of an insertion or a deletion. We now combine these two modifications.

The following is a suggestion for a solution. The parser starts reparsing at the first modified input token. By following links from the input to the parse tree tokens, the parser can deduce what subtree to construct. As long
as there are tagged input tokens, the parser changes and updates the parse

tree. If new text has been inserted, the parser constructs a new subtree. If the
construct is to be an iteration subtree, it is inserted. Otherwise the subtree is
inserted and a corresponding structure is removed from the tree. If the parser
encounters deleted input tokens (only links remaining) the associated tokens
(subtrees) are removed. The parser continues until it reaches a termination


node.

5.8 Error recovery

Error recovery is not considered in this thesis. We have, however, included
a follow set as a parameter for each parse function. The follow set of a parse
function for a nonterminal contains the tokens that can possibly follow the
nonterminal in any derivation. This makes it easy to include some form of
error handling. E.g., if a correct first symbol is not encountered in the input
text, the parser skips all tokens until a follow set symbol is reached. For this
kind of error handling see, e.g., [WM80, pages 99-103].

5.9 Complexity

Determining the complexity of the incremental parser is hard. A simple
 modification can have far-reaching effects on the parse tree, and many nodes
that do not change might have to be traversed. We will now use the tree edit
distances defined in Section 4 to measure the complexity of the incremental
parser at least in some cases.

First, consider the time complexity of the parse-from-scratch algorithm
given as Algorithm 1. Let \( t \) be the old parse tree corresponding to the in-
put \( x \) and \( t' \) the new parse tree corresponding to the modified input \( x' \). We
base the complexity of the algorithm not solely on the changes in input (the
modification) but also on the changes in the output (the parse tree). The
parse-from-scratch algorithm disregards the old parse tree \( t \). This tree must,
however, in any real application be destroyed to deallocate space. Destroying
the tree demands work clearly related to the size \( |t| \) of the tree. The parse-
from-scratch algorithm reconstructs the entire parse tree. The algorithm is
recursive, traversing and building one node at a time. The work done corre-
sp onds to the size \(|t'|\) of the resulting tree \(t'\). Additionally, the entire input is reread. The input is inherent in the new tree \(t'\) so we can assume that reconstructing the parse tree has a time complexity of \(O(|t'|)\). Clearly, it cannot be constructed in less time. Combining this result with the destruction of the old parse tree gives us a time complexity of \(O(|t| + |t'|)\).

When using the parse-from-scratch algorithm, the parse tree does not have to carry information about productions expanded at nodes. If we assume a simple implementation of a parse tree \(t\) where every node has information about (1) its label, (2) its first child, (3) its right sibling, and (4) its parent, we would need space of size \(4 \times |t|\). We conclude that the space complexity of the parse-from-scratch algorithm is \(O(|t| + |x|)\), where \(|t|\) is the size of the parse tree \(t\) and \(|x|\) is the size of the input. We assume that the grammar, first, follow, and dirsym sets, etc., must be stored in any case and we do not take them into account here.

Second, let us study the complexity of the incremental parser. Assume first that the modification in the input is a simple one that only changes one input token. Let \(t\) be the old parse tree corresponding to the input \(x = vwz\) and \(t'\) the updated parse tree corresponding to the modified input \(x' = vw'z\). The incremental parser reparses the modified input token and subsequent tokens until it reaches a termination node. Even a simple modification can affect all succeeding input tokens and the structure of the tree. In the best case, however, the tree is not affected except in the modified parse token. The update work corresponds to reading the modified input token and changing one parse tree token, resulting in a time cost of \(O(|w'|)\). Assuming the modified input token shorter than a constant, the work can be done in constant time.

In the worst case, however, all input tokens and parse tree tokens succeeding the modified one in addition to the structure of the tree might be affected by the simple modification. In this case, the rest of the input must be reread, and the “rest” of the tree, i.e., all nodes that are to the right of the path from the (first) modified parse token to the root, might have to be traversed. We get a time complexity of \(O(|w'z|)\) for reading the rest of the input and \(O(\text{parsedist}(t, t'))\) for traversing the rest of the tree. We therefore conclude that the time complexity of the incremental parser in the worst case is \(O(|w'z| + \text{parsedist}(t, t'))\).

Now let us consider arbitrary modifications. These consist of a sequence of simple modifications. Each simple modification is updated corresponding to
the time complexity given above. Our incremental parser, however, traverses
the entire tree between the first and the last modification. Therefore the
time complexity of an incremental update can be greater than the sum of
the complexities of the individual simple modifications. This makes it hard to
estimate the time complexity, but the lower bound will be $O(\text{parsedist}(t, t') + |w'|)$. In the next section we give some enhancements on the incremental
algorithm that result in a better time complexity.

We see that in either case, processing simple modifications or arbitrary
modifications, the update process is not dependent only on the size of the
changes in the input (input tokens) or output (parse tree). We therefore
conclude that the incremental parser is unbounded.

The incremental parser requires some additional information to be main-
tained in the parse tree. For example, it has to know which production has
been expanded at a certain node. Furthermore, input tokens are linked to
parse tokens and vice versa. These additions require an extra space (com-
pared with the original tree) of $3 \star |t|$. We conclude that the extra space
needed is $O(\max(|t|, |t'|))$.

Our conclusions are summarized in Observation 1.

**Observation 1** Let $G$ be a context-free textable grammar, $x = vwz$ a string
of the language $L(G)$, and $t$ the corresponding parse tree. Let $x' = vw'z$ be
a string of the language $L(G)$. The incremental parser (Algorithms 5–9)
executes correctly producing the updated parse tree $t'$ over $G$ corresponding
to the string $x'$. If the modification $w'$ is simple, the time complexity is in
the best case $O(|w'|)$ and in the worst case $O(\text{parsedist}(t, t') + |w'z|)$. An
arbitrary modification gives a time complexity of $O(|w'|)$ in the best case. In
the worst case the time complexity is $O(|t| + |t'| + |w'z|)$, the same as in the
parse-from-scratch technique.

### 5.10 Improving the incremental parser

The algorithms given by Murching et al. [MPS90] and extended in Section 5.6.2 are not optimal. The incremental parser starts at the first modified
input token and continues until it has passed the last modified input token
and reached a termination node. Input tokens in between are not reparsed
if they are unmodified; still the algorithm traverses all these tokens together
with associated parse tree tokens checking if they have been modified or not.
Therefore, a natural enhancement is to give as input to the algorithm a list of simple modifications \textit{list\_of\_changes}, i.e., a list of modified input tokens. When one simple modification has been reparsed and the parse tree updated, the algorithm continues with the next modification without unnecessary checking of unmodified editor tokens in between.

As explained earlier, a modification can have a far-reaching influence. We therefore demand that the reparsing process reaches a local termination node before skipping to the next simple modification. A \textit{local termination node} is a node where parsing is locally synchronized and where no ancestor, brother, or right ancestor of the node has changed (its production or contents). Local termination nodes are similar to termination nodes with the exception that we do not demand that the parser has passed the last modified input token.

Hence, the parser reparses only modified parts of the input string and skips the unmodified parts. For every modified part, if the production expanded at a node changes, the subtree at the node is reparsed. Therefore, we can expect a time complexity close to $O(\text{parsedist}(t, t') + |\text{list\_of\_changes}|)$ in the worst, where $t$ and $t'$ are the old and the updated parse tree, respectively. In the best case, where the modified input tokens only affect the associated parse tree tokens we have a time complexity of $O(|\text{list\_of\_changes}| + \text{length\_of\_modified\_tokens})$, where \text{length\_of\_modified\_tokens} indicates the sum of the length of modified input tokens.

A second enhancement is to take into account unchanged subtrees of a node with a changed production. This corresponds to a time complexity of $O(\text{predist}(t, t'))$. We leave this as an open problem.

We conclude our observations about the enhancement of the incremental parser in Observation 2.

\textbf{Observation 2} Let $G$ be a context-free textable grammar, $x = vwz$ a string of the language $L(G)$, and $t$ the corresponding parse tree. Let $x' = vv'z$ be a string of the language $L(G)$. Let $\text{list\_of\_changes}$ be a list of modified input tokens, where the modifications transform $w$ into $w'$. Then, the incremental parser (Algorithms 5–9) executes correctly producing the updated parse tree $t'$ over $G$ corresponding to the string $x'$. An arbitrary modification requires a time complexity of $O(\text{parsedist}(t, t') + |\text{list\_of\_changes}|)$ (worst case). In the best case, the time complexity is $O(|\text{list\_of\_changes}| + \text{length\_of\_modified\_tokens})$. \hfill $\Box$
### 5.11 An example of incremental parsing

**Example 19** Assume that $G$ is a grammar $G = (N, T, P, List)$, where $N = \{List, Publ, Author, Title, Journal, Year\}$, $T = \text{all possible (ascii) strings}$, and let the productions and the $First$, $Follow$ and $Dirsym$ sets be as follows.

<table>
<thead>
<tr>
<th>Prod</th>
<th>First</th>
<th>Follow</th>
<th>Dirsym</th>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>List → Publ*</td>
<td>$T$</td>
<td>${\text{eot}}$</td>
<td>$T$</td>
<td>yes</td>
</tr>
<tr>
<td>Publ → Author * Title</td>
<td>$T$</td>
<td>$T \cup {\text{eot}}$</td>
<td>$T$</td>
<td>yes</td>
</tr>
<tr>
<td>Author → Text</td>
<td>$T \setminus {?}$</td>
<td>${?}$</td>
<td>$T$</td>
<td>yes</td>
</tr>
<tr>
<td>Title → Text</td>
<td>$T \setminus {?}$</td>
<td>${?}$</td>
<td>$T$</td>
<td>yes</td>
</tr>
<tr>
<td>Journal → Text</td>
<td>$T \setminus {';','!'}$</td>
<td>${;,'!'}$</td>
<td>$T$</td>
<td>yes</td>
</tr>
<tr>
<td>Year → $\epsilon$</td>
<td>${,'}$</td>
<td>${,'}$</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Year → * Text</td>
<td>${,'}$</td>
<td>${,'}$</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

The terminal $\text{eot}$ stands for end of text. The column $Table$ indicates if the production is textable or not.

Assume that we have the text

$x = \text{Fletcher; Murder; Police; 1990.}$

which has been parsed and we are changing it to

$x' = \text{Fletcher; Murder; Crook.}$

Both texts are consistent with the grammar $G$. We denote $v = \text{Fletcher; Murder.}$, $w = \text{Police; 1990,}$, $z = .$, and $w' = \text{Crook}$. For the string $x$ we have the parse tree in Figure 5.7. This parse tree has been produced from scratch. The bottom row represents the input token list. These tokens are linked with the parse tree tokens. The parse tree is up to date, and all the associated tokens are locally synchronized; the associated tokens represent the same constant or variable string.

Now we modify the input tokens. We substitute the journal name Police by Crook and delete the comma and the year. Before the parsing is started we have the tree and the token list as in Figure 5.8. The inserted string
Crook is not yet an input token and has no associated parse tree token. The links corresponding to the tokens Police, ', and 1990 are still there but the input tokens are empty.

The incremental parser sets oldptr and newptr to point to Last(v) among the input tokens and parse tokens, respectively. The pointer oldptr is set to the parse tree token ')' and the pointer newptr is set to the input token 'C' (Figure 5.8).

After setting the pointers, the parser calls the function Traverse(oldptr, newptr) to reparse the modified part. Because newptr and oldptr are pointing to the same symbol and there are no epsilons involved, the pointers are advanced; oldptr is set to point to the token Police in the parse tree and newptr is set to point to the first character of the string Crook (i.e., 'C').

The parser goes to line 14 in the Traverse function. The function Climb-Up Tree(oldptr) returns the node with the label Journal. The parser sets t to point to this node and calls Parse Journal(t, Follow(Journal)). (Substitute Journal for X in Algorithm 6.)

The function Parse Journal(t, Follow(Journal)) does not create a new node, because we already have a node with the label Journal. Instead we look for
the proper production with the left hand side \( \text{Journal} \) to expand. There is only one production for \( \text{Journal} \) and its \( \text{Dirsym} \) set approves the character C. This is, of course, the same production that was expanded when the input token \( \text{Police} \) was parsed, so the parser goes to line 8 of the \( \text{Parse}_{\text{Journal}} \) function. Because the text has changed, it calls the function \( \text{Parse}_{\text{Text}}(t, \text{child}_1, \text{followset}) \). The parser scans the new string (\( \text{Crook} \)) until one of the strings in the follow set is encountered and attaches this string to the node. This moves \( \text{newptr} \) to the final input token \( . \). Now the parser moves \( \text{oldptr} \) past the yield of the old subtree which was (only) the parse tree token \( \text{Police} \). The pointer \( \text{oldptr} \) is therefore set to point to the comma token \( , \). The new scanned text is made into an input token, which replaces the old token (Figure 5.9).

The loop in the \( \text{Traverse} \) function has now been executed once. The pointer \( \text{oldptr} \) points to the comma token \( , \) and the lookahead symbol of \( \text{oldptr} \) is \( , \). But the input text starts with a period. The parser calls the function \( \text{ClimbUpTree}(\text{oldptr}) \) and sets \( t \) to point to the node with the label \( \text{Year} \). Thereafter the parser calls the function \( \text{Parse}_{\text{Year}}(t, \text{Follow}(\text{Year})) \).
Figure 5.9: A partly updated parse tree and corresponding editor tokens.

Figure 5.10: An updated parse tree and corresponding editor tokens.

84
The node exists, so no new one is created. We see that the production
that is expanded this time is a new one. The parser chooses the production
that derives the empty string because the following input characters
match the Dirsym set of this production. The parser calls the production
Parse$_t$(child$_1$).

Here the parser inserts an epsilon input token into the text and links
it with the tree. The parser returns to the Traverse function (and skips
Traverse$_{yield~of~tree}$) and removes the excess child (the second) of the
node with the label Year. The associated input tokens are also removed. The
pointer oldptr is then advanced beyond the yield of the tree (i.e., to point
to the period token "."). Now both pointers have been moved beyond the
change, the parsing is locally synchronized and oldptr points to a possible
termination node. Therefore, parsing terminates (Figure 5.10).

We can compare the amount of work done by the incremental parser with
the work done by a parser that reparses the entire parse tree. In this exam-
ple, the incremental parser calls 4 parse functions (Parse$_{Journal}$, Parse$_{Text}$,
Parse$_{Year}$, Parse$_e$). There is one call to the Traverse function and two calls
to the ClimbUpTree function. The parser (parse-from-scratch) would call 13
parse functions only for the Publication subtree (Parse$_{Publication}$, Parse$_{Author}$,
etc.). Of course, the parser would not be limited to this subtree but it would
traverse the entire parse tree. In this case, the advantage of the incremental
parser is obvious.

□

85
Chapter 6

Incremental syntax-directed translation

We have studied syntax-directed translations in Section 4. A string of the input language is translated to a string of the output language. Where parsing uses only one grammar for constructing the parse tree of a string, syntax-directed translation uses two grammars, an input and an output grammar. The translate-from-scratch technique retranslates the input string after it has been modified. Incremental syntax-directed translating reuses the old translation, updating the output string where needed. Because the input and output strings can be viewed as parse trees, we also use the term incremental tree translation.

6.1 Incremental translations

We start by defining what we mean with an incremental translation. Let $F = (N, \Sigma, \Delta, R, S)$ be a syntax-directed translation schema and let $x$ be a string of the input language $I(\Sigma)$ and $y$ a string of the output language $I(\Delta)$ such that $(x, y)$ is a translation form of $F$. Let $t_x$ and $t_y$ be the corresponding parse trees.

Assume that the string $x$ and hence its parse tree $t_x$ are modified giving a new string $x'$ and a parse tree $t'_x$. We can for simplicity assume that the tree is updated through incremental parsing, but this is not essential for the following discussion.
Figure 6.1: A scenario for incremental syntax-directed translation.

An *incremental translation* updates the tree \( t_y \) by retranslating modified subtrees in the tree \( t'_x \) (modified with respect to the tree \( t_x \)) and substitutes the corresponding subtrees in the tree \( t_y \) with the retranslated trees (see Figure 6.1). The goal is to do minimal work with respect to the update. As we shall see, this is not always possible.

### 6.1.1 Alternative approaches of retranslation

We give three alternatives for incrementally retranslating a tree. In the first alternative we start retranslating the tree \( t'_x \) at its first changed node in preorder (first edit change). The node can be found (top-down) by finding the first changed subtree or (bottom-up) by going upwards in the tree from the first edit change. The translation proceeds until it has passed the last edit change (the last changed subtree) and reaches a termination node (see below).

In the second alternative, the roots of the changed subtrees in tree \( t_x \) are given as input to the incremental translation. The list is traversed in left to right order retranslating changed subtrees. This is a more efficient method
than the first alternative, but demands additional parameters.

The third alternative is to combine incremental tree translating with incremental parsing or (another) incremental tree translation. As soon as a subtree in the input tree changes, it is retranslated and the corresponding subtree in the output tree is substituted.

6.1.2  Relation to incremental parsing

Incremental translation is clearly related to incremental top-down parsing. The relation to incremental parsing is explained through the following correspondences.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Incremental parsing</th>
<th>Incremental tree translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modification</td>
<td>Change in string →</td>
<td>Change in input tree →</td>
</tr>
<tr>
<td></td>
<td>Change in tree</td>
<td>Change in output tree</td>
</tr>
<tr>
<td>Method</td>
<td>Top-down parsing</td>
<td>Top-down translation</td>
</tr>
<tr>
<td>Input</td>
<td>String with changes</td>
<td>Input tree with changes</td>
</tr>
<tr>
<td></td>
<td>First and last edit</td>
<td>First and last edit</td>
</tr>
<tr>
<td></td>
<td>or list of changes</td>
<td>or list of changes</td>
</tr>
<tr>
<td></td>
<td>Parse grammar</td>
<td>Input and output grammar</td>
</tr>
<tr>
<td></td>
<td>Old parse tree</td>
<td>Old output tree</td>
</tr>
<tr>
<td>Output</td>
<td>Updated parse tree</td>
<td>Updated output tree</td>
</tr>
</tbody>
</table>

For a presentation of related work of incremental parsing see Section 5.

6.2  Data structure support

In order to do the retranslation efficiently, we need some information of the associated nodes in the input tree $t_x$ and its corresponding output tree $t_y$. Two nodes $n$ and $m$ (in $t_x$ and $t_y$, respectively) are associated, if the result of translating the subtree rooted at $n$ is the subtree rooted at $m$.

We assume that we can always find the associated node $m$ of $n$. This can be implemented in two ways. We can connect $n$ with $m$ with a physical link (reference). This is very time efficient, but demands an extra attribute of every node in the input tree.

88
The associated node can also be found by traversing: First find the parent $n'$ of $n$ and the associated node $m'$ of $n'$. Then traverse the children of $m'$ until the associated node $m$ of $n$ is found. If the SDTS is simple, i.e., nonterminals in the input and output productions occur in the same order, the traversal is easy. Let the label of $n$ be $l$. Now there might be several nodes among the siblings of $n$ with label $l$. Assume that $n$ is the $i$th node among siblings with label $l$ (i.e., there are $i-1$ siblings of $n$ with label $l$ to the left of $n$). Then $m$ is the $i$th $l$-labeled node among the children of $m^n$.

In our approach, we demand that iteration productions are simple and of the type $I \rightarrow A^*$. Then the $i$th subtree of an iteration always corresponds to the $i$th subtree of the associated subtree in the output tree. The nonterminals of other productions can be permuted, except for nonterminals with the same label; these always occur in the same order between themselves. Therefore, the associated node can be found as above in the case of a simple SDTS.

The traversal approach trades space for time. Let $k$ be the maximum length of an input-output production pair. Then the time complexity of the traversal is clearly $O(k^2)$ for the production in question. (Iterations are traversed differently, see below.) The link approach demands an extra attribute for every node, but an associated node is accessed in one step.

### 6.3 Termination conditions

As in incremental parsing, we define here some termination conditions for incremental tree translation.

Assume that a node $n$ is a termination node in an incremental update of a tree and that the tree is used as an input tree in an incremental translation. Let the associated node of $n$ in the output tree be $n'$. It is not certain, however, that the associated node $n'$ is a termination node for the incremental tree translation (see Example 20). As a matter of fact, if $n$ is a constant string there is no associated node in the output tree. But if we translate subtrees of the input tree in left to right order until we reach a termination node, all

---

1If the nonterminals can occur in different order in the input and output productions, the grammars must be considered when looking for the associated node. We assume, however, that if there are several occurrences of a nonterminal in two associated productions in an SDTS, the occurrences are associated in the order they appear. Compare with the definition of a translation form on page 37.
modified subtrees in the input tree will be correctly translated, as long as
the production expanded at the parent node of the termination node has not
changed.

**Example 20** Let $F$ be a syntax-directed translation schema with the pro-
duction $S \rightarrow ABC; CBA$. Assume that we have instances of the input and
output strings where the corresponding trees are $t_x$ and $t_y$ (Figure 6.2).

```latex
\begin{figure}[h]
\centering
\begin{tikzpicture}
  \node (S) at (0,0) {$S$};
  \node (A) at (-1,-1) {$A$};
  \node (B) at (0,-1) {$B$};
  \node (C) at (1,-1) {$C$};
  \node (S_y) at (2,0) {$S$};
  \node (C_y) at (2,-1) {$C$};
  \node (B_y) at (1,-1) {$B$};
  \node (A_y) at (0,-1) {$A$};
  \draw (S) -- (A) -- (S_y);
  \draw (S) -- (B) -- (S_y);
  \draw (S) -- (C) -- (S_y);
  \draw (S) -- (translation) -- (S_y);
\end{tikzpicture}
\caption{Two corresponding trees of a translation form.}
\end{figure}
```

If node $B$ in the input tree $t_x$ is a termination node when updating the
input tree, the corresponding node cannot be considered as a termination
node in the tree $t_y$; then the node $A$ would not be translated and the node
$C$ would be translated unnecessarily. If we, however, translate the nodes in
tree $t_x$ in left to right order *until we reach a termination node* in the input
tree, all subtrees are correctly translated.

If the production at the parent of a termination node changes in the
input tree, the entire subtree with the parent as root must sometimes be
retranslated. We give the following clarifying example.

**Example 21** Let $F$ be a syntax-directed translation schema with produc-
tions $S \rightarrow ABC; ABC\ a$ and $S \rightarrow DBC; DBC\ b$, where $a$ and $b$ are
constant strings. Assume that we have instances of the input and output
strings where the corresponding trees are $t_x$ and $t_y$ (Figure 6.3).

Let node $B$ in the input tree $t_x$ be a termination node. This means that
the node $B$ remains the same after the modification. Even if retranslations

90
Figure 6.3: A change of production during retranslation.

of all subtrees preceding $B$ are done, the resulting output tree is incorrect if
the subtree rooted at $S$ is not retranslated. The production at node $S$ has
changed, thereby changing constant strings.

We state our observations as a lemma (compare with Lemma 5 on page 64).

**Lemma 6** Let $F = (N, \Sigma, \Delta, R, S)$ be a syntax-directed translation schema.
Let $(x, y)$ be a translation form of $F$ with $t_x$ and $t_y$ as the corresponding parse
trees. Assume a left to right top-down incremental tree translation order.
Assume also that the string $x$ is modified to $x'$. The incremental retranslator
can stop retranslating when the following termination conditions hold.

a) All nodes with new expanded productions have been
   retranslated.

b) The retranslation has passed a termination node in the
   input tree.

**Proof.** We briefly sketch the proof. It is obvious from the example above
that the retranslator must process all nodes with a new expanded produc-
tion. Otherwise, consistency of the resulting tree with the input may not be maintained. On the other hand, the contents of a leaf node might have changed without changing the expanded production at an ancestor node. Therefore condition b) corresponds to conditions a) and c) in Lemma 5 ensuring that the translator has retranslated all the modified nodes.

If we want to set a termination node \( m \) for the output tree it can be determined as follows. Let \( n \) be the termination node of the input tree. If no ancestor of \( n \) in the input tree has changed its production, \( m \) is the associated node of \( n \) or the associated node of the nearest nonterminal node following \( n \) in preorder (if \( n \) does not have an associated node).

If some ancestor of \( n \) has changed its production, we search for the oldest such ancestor. The termination node \( m \) is the associated node of the first nonterminal node following the ancestor in preorder.

### 6.4 Iterations

We have so far handled only changes (changed contents or changes of expanded productions) in the input tree, not insertions or deletions. These can occur when the input and output grammar contain iterations.

Iterations are more difficult to retranslate: we must know where to insert (or delete) an iteration part, and how to permute the nonterminals in it. If an iteration part is inserted to or deleted from a subtree, we can consider the production to have changed at the root and retranslate the entire subtree. This solution is still better than the transformation-from-scratch method, but we might do a lot of extra work retranslating iteration parts that have not changed.

We restrict the iteration productions to the type \( I \rightarrow A^* \), where the iteration part consists of a single nonterminal (or a constant string). Other iterations can easily be rewritten in this form by adding more productions.

### 6.4.1 Deletions

Deletions in the input tree are easily updated in the output tree. But as the deleted subtrees are no longer present in the input tree, the incremental translation needs a separate list of deleted nodes. All deleted subtrees have an associated node in the output tree which is deleted.
In the implementation, deleted nodes can be removed from the iteration, marked as deleted and maintained as separate trees. This guarantees that they can be given as parameters to the incremental translation algorithm.

### 6.4.2 Insertions

Insertions are more difficult to handle than deletions. When retranslating changed and \emph{new} subtrees, new nodes belonging to iterations have no associated nodes in the old output tree. Because iteration productions are of the simple type \( I \rightarrow A^* \), there is no problem with permutations. A \emph{new node} in the input tree is translated top-down and the resulting subtree is inserted \textit{to the right of the associated node of the left brother of the new node}. If the new node has no left brother, the result is inserted as the first child of the iteration node in the output tree.

### 6.5 A naive algorithm

We give an algorithm that follows the second alternative approach of translation (Algorithm 10, p. 94). The algorithm is given a list of modified (inserted, deleted, or changed) subtrees to retranslate; a subtree has been modified if the production applied at its root has been changed, or if its contents (in the case of a Text node) has changed. Only non-overlapping modified subtrees are included in the list; modified subtrees contained in another modified subtree are \textit{not} included. E.g., if the production applied at the root of the input tree has changed, the list contains only one element, the root.

The algorithm traverses the list, handling one modified subtree at a time. If the subtree \( t \) has been deleted (and it is part of an iteration) it is also deleted from the output tree. If it has been inserted, an associated node of the root of the subtree must be inserted in the output tree. The insertion point is determined as follows. First find the left brother of the root \( n \) of \( t \) and its associated node \( m \) in the output tree. Then insert a new node \( m' \) to the right of \( m \). If there is no left brother of \( n \), insert \( m' \) as the first child in the iteration. Node \( m' \) is the root of a subtree \( t' \) corresponding to \( t \). Nodes are only marked as deleted or inserted if they are roots of subtrees in an iteration. Inserted (new) nodes are translated top-down.

If \( t \) was neither deleted nor inserted, it must have been modified (new
Algorithm 10 (Naive incremental syntax-directed translation)

```
procedure inc_translate(list_of_changes, tx, ty);
    Input: A modified input tree tx, a list of nonoverlapping, nonredundant modified subtrees in tx, and the old output tree ty.
    Output: Updated output tree ty.
    Task: Propagate modifications through tree.
    begin
        while list_of_changes not empty do
            t := next modified subtree;
            if t deleted then
                Remove associated node t';
                Mark t' deleted
            elseif t inserted then
                Insert an associated node t';
                Mark t' inserted;
                Top-down translate t at t';
            else /* t is modified */
                Top-down translate t;
            Replace associated node t' in ty;
            Mark t' modified;
        end;
```

production or modified contents). The subtree t is translated top-down and the result replaces the old associated tree t'.

Inserted and modified nodes are marked accordingly. All these nodes must be collected in a list of all modified iteration parts; this makes it possible to use the output tree as an input tree in another incremental translation.

The modified subtrees of the input list are completely retranslated. This is due to the translation-from-scratch algorithm which is called to translate the modified tree top-down. A tree is considered to be modified when the

---

This is actually the case, when different document representations are updated in the HST system. A view tree is retranslated to update the logical tree, which is retranslated to update other view trees, etc. (see Section 3).
production expanded at the root is changed. Therefore, if the production expanded at the root has changed, the input tree is completely retranslated. But even if a new production is applied at the root, some of the subtrees may remain unchanged (see Example 21 on page 90.).

6.5.1 Complexity

All subtrees with a changed production are retranslated. Therefore the work done by his algorithm is proportional to \( \text{parsedist}(t_x, t'_x) + \text{parsedist}(t_y, t'_y) \).

We traverse the modified subtrees in tree \( t'_x \) and translate them to subtrees in the tree \( t'_y \). The method that translates the tree from scratch demands time proportional to the sum of the size of trees \( t'_x \) and \( t'_y \), which is the complexity in the worst case of Algorithm 10.

Compared with translation from scratch, some extra space is needed for both the input and output trees. The algorithm reads (constructs) a list of modified subtrees. For every modified subtree on the list we need to know if it has been deleted, inserted, or modified. The references to the subtrees can be pointers. The incremental translation therefore demands extra space of size \( 2 \times |\text{list of changes}| \). An upper bound on the list of changes is the size of the input (output) tree. The algorithm needs links (assuming this implementation) for associated nodes. This extra space is proportional to the size of the input tree.

Our conclusions are summarized in Observation 3.

Observation 3 Let tree \( t_y \) be a translation of the tree \( t_x \), let \( t'_x \) be an update of the tree \( t_x \) and \( \text{list of changes} \) a nonredundant covering list of modified subtrees in \( t_x \). Then Algorithm 10 executes correctly producing the updated version of the output tree in time \( O(\text{parsedist}(t_x, t'_x) + \text{parsedist}(t_y, t'_y)) \) and extra space \( O(|t_x| + |\text{list of changes}|) \).

6.6 An enhanced algorithm

Algorithm 10 is not optimal. When the production at a node changes, it is still possible that parts of the subtree rooted at the node remain intact. The incremental tree translation Algorithm 10 retranslates all subtrees with changed productions in order to maintain consistency.
Algorithm 11 (Enhanced incremental syntax-directed translation)

```
procedure inc_translate2(t_x, t_y);
Input: Modified input tree t_x where all new and modified
      nodes have been marked, and old output tree t_y.
Output: Updated output tree t_y.
Task: Propagate modifications through tree.
begin
  1 let n be the root of t_x.
  2 let W → α be the production expanded at n and
     let W → β be the annotated production;
     Assume β consists of symbols X_1, X_2, ..., X_k;
  3 let m be the root of t_y.
  4 Reorder, insert and delete children of m so that
     their labels form β = X_1X_2⋯X_k;
  5 let J = {j_1, j_2, ..., j_p} be the permutation order,
     where p is the number of permuted nonterminals,
  6 for all marked nonterminal children n_i of n do
     inc_translate2 (n_i, m_{j_i});
  7 mark m_{j_i};
end;
```

Consider Example 21 on page 90. Because the production at node S is
replaced in the input tree, Algorithm 10 retranslates the entire tree. To be
more efficient, the algorithm could delete old constant strings and insert new
ones in the output tree and then retranslate subtrees until a termination node
in the input tree is reached. In the example, the constant string a could be
replaced with the string b, and as B is a termination node, only the subtree
at D should be retranslated.

We give an enhanced incremental translation algorithm (Algorithm 11)
which uses a more efficient method of change propagation. This new al-
gorithm can be called instead of the translation-from-scratch procedure in
Algorithm 10. The algorithm can also be executed with only the input and
output trees as parameters. A precondition of the algorithm is that all modified subtrees (nodes) in the input tree are marked.

The algorithm starts by finding the production expanded at the root of tree $t_x$ and its associated production for the tree $t_y$. Thereafter it reorders, deletes, and inserts nodes to the root of the output tree so that the labels of the nodes form the right hand side of the output production. Old nodes are reused as much as possible (even constant strings). As most usable productions contain less than a constant number of children (iterations excluded), this reordering is not too time consuming.

If, however, the production is an iteration (of type $I \rightarrow A^*$), we may assume that references to deleted, inserted, and modified nodes are kept in a list associated with their parent. The time saved might very well compensate for the extra space used.

Modified nodes are marked correspondingly and incrementally retranslated; in Algorithm 10 they were translated from scratch. Deleted or inserted nodes do not have to be marked (except in iterations); the translator notices them to be missing or present and can take appropriate actions.

### 6.6.1 Complexity

Algorithm 11 retranslates only modified subtrees. In order to determine which subtrees to translate, all the children of the root of a modified subtree are checked. Considering that checking for changed children takes much less time than retranslating a subtree, the time complexity is $O(\log |t_x| \text{predist}(t_x, t'_x) + \log |t_y| \text{predist}(t_y, t'_y))$, where the logarithmic factor stands for the path from the root to the modified node. In any practical case, the number of children is limited (iterations excluded) and the checking can be done in constant time. Iterations are handled in a different way by associating a list with deleted, inserted and changed nodes with the root of the iteration (in the input tree). These nodes can then be processed under the same time complexity.

The logarithmic factor disappears if the algorithm is given a list of all modified, inserted, or deleted nodes as input.

Algorithm 11 demands more space than the naive algorithm. For both the input tree and the output tree we need an attribute for each node telling whether it has been inserted or changed or if it is unmodified. Deleted nodes have been removed or are being removed from the tree and do not need to be
marked. Furthermore, the list at iteration nodes containing inserted, deleted, and changed nodes demands a number of references to these nodes, limited by the size of the tree. The link implementation (between associated nodes) demands an extra space of the size of the input tree. Therefore we conclude that the extra space needed is at most \(3*|t_x| + 2*|t_y|\) or \(O(|t_x| + |t_y|)\), where \(t_x\) and \(t_y\) are the input and output tree, respectively.

Our conclusions are summarized in Observation 4.

**Observation 4** Let tree \(t_y\) be a translation of the tree \(t_x\), let \(t'_x\) be an update of the tree \(t_x\), and assume that changed or inserted nodes in \(t_x\) have been marked. Assume also that iterations have been “marked” with a list containing all modified nodes. Then Algorithm 11 executes correctly producing the updated version of the output tree in time \(O(\log \|t_x\| \text{predist}(t_x, t'_x) + \log \|t_y\| \text{predist}(t_y, t'_y))\) and extra space \(O(|t_x| + |t_y|)\). \(\square\)

### 6.7 An example of incremental tree translation

We give a small example of incremental tree translation as a successive step to an incremental update (incremental parse or incremental translation) of the input tree \(t_x\).

**Example 22** Let \(F\) be an SDTS with the following productions. Note the two different representations of the *Publication* production. A blank space is denoted by \(\_\_\_\_\_\_\_\_\_\_.\)

**Grammar 4**

\[
\begin{align*}
    \text{List} & \rightarrow \text{Publications} \text{; Publications} \\
    \text{Publications} & \rightarrow \text{Publication}\_\_\_ \text{; Publication}\_\_\_ \\
    \text{Publication} & \rightarrow \text{Author} \_\_\_ \_\_\_\_\_\_. \_\_\_ \text{Title} \_\_\_ \_\_\_. \_\_\_ \text{Journal} \_\_\_ \_\_. \_\_\_ \text{Year} \_\_\_ \_\_. \_\_\_. \_\_. \_\_. \text{; Title Author} \\
    \text{Author} & \rightarrow \text{Text} \text{; Text} \\
    \text{Title} & \rightarrow \text{Text} \text{; Text} \\
    \text{Journal} & \rightarrow \text{Text} \text{; Text} \\
    \text{Year} & \rightarrow \_\_\_ \_\_. \_\_\_. \text{Text} \text{; Text}
\end{align*}
\]
As a syntax-directed translation from scratch of the tree in Figure 6.4 produces the tree in Figure 6.5.

A modification in the view tree, e.g., changing the parse tree tokens

\[ \text{Fletcher; Murder; Police; 1990}. \]

to

\[ \text{Author; Fletcher; Title; Crook}. \]

invokes the incremental parser to produce the corresponding modifications in the view tree (Figure 6.6). The period is the termination node of the reparse. No node after it is changed in the update of the view tree.

Now we invoke the incremental translator. The naive Algorithm 10 is given a list of modified subtrees (not contained in each other) consisting of one subtree, the subtree at \( \text{Publications} \). Because the production expanded at the root of this subtree was changed, the entire subtree is retranslated (see Figure 6.7). This corresponds to removing all 13 old nodes in the \( \text{Publication} \) subtree of tree \( t_y \) (Figure 6.5) and inserting 7 new nodes in the same subtree (Figure 6.7), all together 20 “node modifications” \( \text{parsedist}(t_y, t'_y) = 20 \).
In retranslating the subtree $t'_x$ it must be traversed. This corresponds to traversing 6 nodes in the subtree at `Publication`.

If we want to set a termination node for the updated parse tree, it is set to the `Author` node in the following subtree (Figure 6.6).

Let us assume that we invoke Algorithm 11 with the subtree at the `Publication` node as parameter. We also assume that modified nodes have been marked, in this case the `Publication` and `Title` nodes have been marked as changed. Additionally, two nodes have been deleted (`Journal` and `Year`). The incremental translator starts by processing the changed node `Publication`. The translator checks the children and removes the associated nodes which were deleted in the input tree. Then no nodes are inserted, and the remaining nodes are reordered (as in Figure 6.7). The node `Author` is not retranslated because it has not been marked. The node `Title` has been modified, and is retranslated. After this the termination node (the period in the input tree) has been reached and the incremental tree translation is terminated. We see that the incremental translator removes 6 nodes, reorders 2 nodes and retranslates 3 nodes (the subtree beginning at `Title`). All in all there are 11 “node modifications”.

Figure 6.5: The output string of a translation as a tree.
Figure 6.6: A parse tree of a modified input string.

Figure 6.7: A parse tree of a retranslated output string.
The termination node of the output tree is set to the *Author* node, which is the node following the last changed subtree (the second *Author* node in Figure 6.7). The result of the retranslation is of course the same as the result of the translation-from-scratch (Figure 6.7).
Chapter 7

Lazy syntax-directed translation

Lazy syntax-directed translation transforms a part of a document instance to a view. The subtrees in the part are all transformed forming a lazy view of the document. A lazy translation differs from pruning a document. A pruned view of a document contains only certain logical parts, e.g., all section titles. Lazy views and pruned views are both special cases of partial views. A general partial view can contain any transformed parts of a document instance. In this section we present a formalism and technique for lazy translation of a document instance. Lazy translation is combined with incremental updates.

7.1 A lazy view grammar

Lazy translation transforms a part of the document instance into a view. Let $G$ be a grammar and $G_a$ its annotation. We denote by $T(G)$ and $T(G_a)$ the sets of all possible parse trees over $G$ and $G_a$. Let $t \in T(G)$ be a parse tree for a document. We say that a lazily transformed document is a tree $w \in T(W)$, where $W$ is a suitable grammar (we explain $W$ later). Transforming a part of a document instance is a function $(T(W), R) \rightarrow T(W)$, where $R$ is a set of ranges that indicate where the transforming should start and how much of the document should be transformed.

We demand the grammar $W$ to be such that the lazily transformed tree $w$ can contain subtrees according to the grammar $G$. We furthermore demand
that \( T(G) \subseteq T(W) \) and \( T(G_a) \subseteq T(W) \). Let

\[ p_G = A_G \rightarrow B_1 B_2 \ldots B_n \]

be a production of \( G \) and let

\[ p_{G_a} = A \rightarrow b_1 b_2 \ldots b_n \]

be the corresponding annotated production of \( G_a \). A lazy production of \( p_G \) and \( p_{G_a} \) for a lazy view grammar is a production

\[ p_W = A \rightarrow b_1 b_2 \ldots b_n | A_G \]

a combination of \( p_G \) and \( p_{G_a} \). Each \( b_i \) is a terminal string or empty, and the indices \((i_1, \ldots, i_n)\) are a permutation of the sequence \((1, \ldots, n)\).

Let \( t \in T(G) \) be a document instance. We say that the parse tree \( w \in T(W) \) is a lazily transformed representation (lazy view) of \( t \) if for each node \( n \in w \) we have that the production of \( G_a \) corresponding to \( n \) has been applied, or the transfer production referring to \( G \) has been applied and the subtree \( n[t] \) for \( n \) is the corresponding subtree of \( t \) (i.e., the two trees share the subtree). Technically, we implement a transfer production as a link from the lazy view tree to the logical tree.

**Example 23** Let grammar \( G \) contain productions \( P = \{ S_G \rightarrow A_G B_G C_G, A_G \rightarrow a, B_G \rightarrow b, C_G \rightarrow c \} \) and let its annotated grammar \( G_a \) contain the corresponding productions \( P_a = \{ S_{G_a} \rightarrow A_{G_a} | B_{G_a} | C_{G_a}, A_{G_a} \rightarrow a, B_{G_a} \rightarrow b, C_{G_a} \rightarrow c \} \). The lazy grammar has productions \( P_W = \{ S_{G_a} \rightarrow C_{G_a} | B_{G_a} | A_{G_a}, B_{G_a} \rightarrow b, C_{G_a} \rightarrow c \} \cup P \). In Figure 7.1 we have parse tree \( t \) over grammar \( G \) and a lazy view \( t' \) of \( t \). The subtrees with roots \( C_G \) and \( B_G \) have been transformed. The subtree with root \( A_G \) has not been transformed. Instead the transfer production has been expanded in the lazy view \( t' \). In practice, we implement this by a special link from the node \( A_{G_a} \) to the associated node \( A_{G} \) in tree \( t \) (see Figure 7.1).

\[ \square \]

The grammar \( W \) is similar to the view grammar \( G_a \), but every production in \( G_a \) is extended with a nonterminal that transfers the production into \( G \). A tree \( w \) over \( T(W) \) can be partially transformed, but those parts that are
not transformed are directly linked to the document tree. Every node in a view tree is then transformed or linked to the document tree. A transformed node can be the root of a subtree that has been totally transformed. A transformed node can also have children that have not been transformed.

7.2 Iterations

We modify the iteration productions of a lazy view grammar for better efficiency. According to the definition of a lazy production given above, the root of an iteration subtree must be totally transformed (all children present), or linked to the associated iteration node in the document tree. This violates the lazy principle. Obviously, we want to transform only those iteration subtrees that are needed. As a solution we give a slightly different lazy iteration production. Assume that we have the production

\[ p_G = A_G \rightarrow B_G^* \]

and its annotated version

\[ p_{G_a} = A_{G_a} \rightarrow (B_{G_a} ;'\;')^* \]

with a semicolon delimiting each \( B_{G_a} \). We have the lazy production
If we have an iteration in \( n \) and the transfer production has been applied, the subtree \( n[t] \) beginning at \( n \) is the combination of the corresponding subtree in \( t \) and a predicate \( \varphi \) that indicates those nodes of the iteration that have been transformed.

**Example 24** In Figure 7.2, we have a lazy view tree, where the first two subtrees \( B_G \) of an iteration have been transformed. The subtree for \( A_G \) is part of the document \( t \). The predicate \( \varphi \) tells how much of the iteration has been transformed.

### 7.3 Continuous transformation

Transforming more of a lazily transformed document is a function \((T(W), R) \rightarrow T(W))\), where \( R \) is the set of ranges. Assume that \( w \in T(W) \) is a lazy view of the document. When applying the lazy translator, we choose some node \( n \in w \), which is indicated by the range \( r \in R \) and where the transfer production has been applied, and call the appropriate translation procedures. The result is a lazily transformed tree. Algorithm 12 performs a lazy translation.
Algorithm 12 \textit{(Lazy translation)}

\begin{verbatim}
procedure Lazy_translate(t, r);
Input: Lazy view tree t and range r.
Output: Lazy view tree.
Task: Transform t in range r.
begin
1 while in range r do
2 n := next node in preorder;
3 if transfer production n\_G applied then
4 Translate n\_G; 
5 Replace n with result;
end;
\end{verbatim}

We denote the range \( r \in R \) by a triple \((n, ch\#, amount)\), where \( n \) is a node in the lazy parse tree and \( ch\# \) is the ordinal number of a character in the frontier of the subtree with the node \( n \) as a root. This character is the first character that is part of the result of the transformation. The variable \textit{amount} indicates how much (how many characters) the result should comprise. The amount can also be negative indicating that the direction of the transformation should be from right to left in the input tree\footnote{This can be done by changing line 12 in Algorithm 2 on page 38 to \textit{for i = n.size to 1 do}, i.e., by processing the children of a node in opposite order}. This way we can specify a \textit{forward} or \textit{backward} lazy transformation. Obviously, it is not sufficient to specify the starting point by simply giving a node; a \textit{Text} node can contain an arbitrarily long text.

We can compare the lazy translator with the incremental translator. Every untransformed subtree in the lazy view tree can be considered as a modified subtree. The lazy translator only retranslates such subtrees. The incremental translator finds associated nodes in the input and output trees through links. The lazy translator uses transfer productions for finding associated nodes.

The complexity of Algorithm 12 can be compared with the complexity of the incremental translator (Algorithm 10). The lazy translator transforms
only untransformed subtrees. Still, the time complexity depends much on the given range $r$. If the range also includes transformed parts of the view tree, the lazy translator traverses these in order to find untransformed subtrees. In this case the lazy translator is not bounded by, e.g., the measures $\text{parsedist}$ or $\text{predist}$, but only by the size of the range $r$. If the range $r$ is carefully chosen, the algorithm will traverse only untransformed parts of the view tree. When it finds an untransformed node, it traverses the associated subtree of the input tree and performs the transformation. Let $t_1, t_2, \ldots, t_m$ be the nonoverlapping subtrees in the range $r$. The subtrees can be transformed, untransformed, or partially transformed. Let the corresponding transformed subtrees be $t'_{1}, t'_{2}, \ldots, t'_{m}$. We have that $t_j = t'_{j}$ for some $1 \leq j \leq m$ (transformed subtrees) and $t_k \neq t'_k$ for some $k$ (untransformed, partially transformed). We conclude that the time complexity of the lazy translator lies between $O(|t_1| + \cdots + |t_m|)$ and $O((|t_1| + \cdots + |t_m|) + (|t'_1| + \cdots + |t'_m|))$, where $|t_i|$ denotes the size of tree $t_i$.

If we improve the time complexity by associating with every node in the view tree the status of the node (transformed, untransformed, or partially transformed), the lazy translator does not have to traverse transformed subtrees and the time complexity is $O(\text{predist}(t_1, t'_1) + \cdots + \text{predist}(t_m, t'_m))$, where $\text{predist}(t_i, t'_i) = 0$ if $t_i = t'_i$.

### 7.4 Combining lazy translation and incremental updates

We combine lazy translation with incremental updates. Assume that the user is modifying a large document. Assume also that editor space is limited. When the user opens the document, only the first part of the document is transformed and loaded into the editor. Subsequent parts are transformed and loaded when the user moves to other parts of the document. Consequently, some parts must then be removed from the editor to make space. If the user has made modifications in some parts, these must be parsed and the document tree must be updated. All updates are done incrementally.

The processing algorithm is sketched as Algorithm 13. A portion of text is either loaded or saved. If it is saved, the textual view is first parsed to update the view tree. The view tree is translated to the document tree. If
Algorithm 13 (Lazy incremental document preparation)

```plaintext
procedure Process(act, t, r);
Input: act: load or save, view tree t, range r.
Task: Transform and load more of a lazy view or
      save and update a part, propagate changes
      through representations.
begin
1   if action = save and view modified then
2       Incrementally parse view -> view tree t;
3       Incrementally translate
4           view tree t -> logical tree t_l;
5           if other views open then
6               Incrementally translate
7                 logical tree t_l -> view trees;
8       Update views;
9 elseif action = load then
10      if no more space then
11         Process(save, t, r_old);
12         Remove parts of view;
13         Lazy_translate(t, r);
14   Update view;
end;
```

other views are open, the document tree is translated to different view trees
and the corresponding textual views are updated. If a portion of a textual
view is loaded, actions are first taken possibly to save loaded portions. These
portions are saved, and their position in the view tree t is indicated by a range
parameter that has been updated for every load. When there is sufficient
room in the editor the lazy translator can proceed and transform some part
of the document. The textual view is updated with the result.
7.5 Examples of lazy transformation

We look at a larger example of lazy transformation. We concentrate on lazy transformation, not on incremental updates. Examples of incremental parsing and translation can be found in Sections 5 and 6.

Example 25 Assume we have the following context-free grammar (Grammar 5) and its annotated version (Grammar 6) for modeling the structure of the document.

<table>
<thead>
<tr>
<th>Grammar 5</th>
<th>Grammar 6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Document grammar</strong></td>
<td><strong>View grammar</strong></td>
</tr>
<tr>
<td>List → Publs</td>
<td>List → Publs</td>
</tr>
<tr>
<td>Publs → Publs*</td>
<td>Publs → Publs*</td>
</tr>
<tr>
<td>Publ → Authors</td>
<td>Publ → Authors</td>
</tr>
<tr>
<td></td>
<td>Title → Title</td>
</tr>
<tr>
<td></td>
<td>Journal → Journal</td>
</tr>
<tr>
<td></td>
<td>Volume → Volume</td>
</tr>
<tr>
<td></td>
<td>Year → Year</td>
</tr>
<tr>
<td></td>
<td>Pages → Pages</td>
</tr>
<tr>
<td>Authors → Author*</td>
<td>Authors → Author*</td>
</tr>
<tr>
<td>Author → Initials</td>
<td>Initials → Initials</td>
</tr>
<tr>
<td>Name → Text</td>
<td>Name → Text</td>
</tr>
<tr>
<td>Title → Text</td>
<td>Title → Text</td>
</tr>
<tr>
<td>Journal → Text</td>
<td>Journal → Text</td>
</tr>
<tr>
<td>Volume → Text</td>
<td>Volume → Text</td>
</tr>
<tr>
<td>Year → Text</td>
<td>Year → Text</td>
</tr>
<tr>
<td>Pages → Start End</td>
<td>Pages → Start End</td>
</tr>
<tr>
<td>Start → Text</td>
<td>Start → Text</td>
</tr>
<tr>
<td>End → Text</td>
<td>End → Text</td>
</tr>
</tbody>
</table>

A part of a document tree might look like shown in Figure 7.3. The tree only contains one instance of Publication

If the entire document tree in Figure 7.3 is translated we get an annotated view tree which is shown in Figure 7.4.
The view that is loaded into the editor is

\[ \text{J. Fay: Me Life: 3 (89) 6-9}. \]

Assume that this text does not fit totally in the editor (what an editor!). Only some characters can be loaded at a time. The lazy translator starts with the document tree in Figure 7.3 and transforms the first part of it. Assume that the editor can only load 12 characters at a time. Then this reference to an article really is too long.

Before calling the lazy translator, we must specify the range parameter \( r \). Obviously, because the document is not open we want to transform the first part of it, and the range is set to \( r = (\text{Publ#}, 1, 12) \), where \( \text{Publ#} \) is a link to the root of the subtree at \( \text{Publ} \), the number 1 indicates that the result should start with the leftmost character of the frontier of the subtree, and the number 12 indicates how many characters should be in the result at most.
Figure 7.4: A totally computed view subtree.

The lazy translator does not find a view tree so first it calls $\text{Translate}_{\text{List}}$ (see Algorithm 2 on page 38). $\text{Translate}_{\text{List}}$ calls $\text{Translate}_{\text{Publs}}$ and so on until the translator has a view tree that contains about 12 characters (i.e., at least 12 characters, but if the last node transformed is omitted, there should be less than 12 characters). We get a lazy view tree like the one in Figure 7.5. Only the first part of the document tree has been transformed. We see that the node $\text{Authors}$, the string 'Fay', the node $\text{Title}$ and the string 'Me Life' together contain 12 characters (they will fit into the editor). But if also the node $\text{Journal}$ is included we get too many characters. Consequently, the rest of the view tree is not yet computed. Instead some nodes have transfer productions (denoted by $\triangledown$) and they are linked to the associated nodes in the docu-
ment tree. The range parameter is updated to $r = (\text{Journal}#, 1, 12)$, where $\text{Journal}#$ stands for a link to the $\text{Journal}$ node. The string (12 characters) loaded into the editor is

\[ \text{J.} \text{Fay:} \text{Me.} \]

The lazy translator knows that the last node that was loaded was the node with the constant string `'.'`. The nodes $\text{Journal}$, $\text{Volume}$, $\text{Year}$ and $\text{Pages}$ remain untransformed in addition to the nodes $\text{List}$, $\text{Pubs}$, and $\text{Publ}$ (partly untransformed). The rest of the nodes are (totally) transformed.

We now try to load more of the view into the editor. The lazy translator starts with the node $\text{Journal}$. It is not yet transformed, so the lazy translator
transforms it and continues, taking into account the constant strings. It has to go as far as the terminal string following the node Year to get a new string that contains exactly 12 characters. The node Pages still remains untransformed. The range parameter is again updated to \( r = (Publ \#, 25, 12) \). The string that is the result of this translation is

\[ \text{Life} : \underline{3} \underline{9} \]

The corresponding view tree is shown in Figure 7.6. The next load brings the rest of the text into the editor. The rest of the text is

\[ \text{J., Fay, Me, Life} : \underline{3} \underline{9} \]

114
The totally transformed subtree is shown in Figure 7.4. The range parameter would now be $r = (Publ\#; 33, 12)$. At this moment the entire view is transformed. If the user does not modify the view, new parts of the view can easily be loaded. If the user chooses to move backwards in the document, the amount parameter in the range is set to a negative value and the transformed view is loaded from a certain position in the tree.

We have not considered here incremental updates as we saw several examples of these in Sections 5 and 6.
Chapter 8

The HST approach and implementation

The Helsinki Structured Text Database System (HST) is an environment for preparing structured documents. The system keeps different representations of a document: plain text in an editor, a view tree, and a document tree. In this section we briefly present the incremental parser that has been developed. We also give a brief discussion on problems related to the implementation of the incremental and lazy translators.

8.1 The HST system

The Helsinki Structured Text Database System [KLMN90] was developed during a three years period starting in 1988. The project was part of the FINSOFT software technology program under the Technology Development Center (TEKES) and the work was supported by TEKES and the Academy of Finland. We presented the functional principles in Section 3. Figure 8.1 shows the user interface of the system. In the figure, we see two views of a document containing references from a bibliography database.

The system was developed in C under Sun’s OpenWindows environment [Sun90]. For developing the user interface we used the XView [Hel90] macro package. We also used a relational database system as storage for the structured documents.

We divided the HST system into two parts, the user interface and the
PQLM process. The user interface (Figure 8.1) lets the user conveniently manage different views of a document. The PQLM\(^1\) process handles parsing, transformations and the connection to the database (see Figure 8.2).

\(^{1}\)PQLM stands for 'P-string Query Language Machine', an implementation for the p-string query language PQL we constructed [KLM+91b] based on the p-string language of Gonnet and Tompa [GT87].

---

Figure 8.1: An overview of the user interface of the HST system.

Figure 8.2: The two parts of HST.


8.2 The incremental parser

We have built a simple incremental parser following the specification given in Section 5. Some restrictions were made. For simplicity, we excluded iterations. We did not integrate the incremental parser with the entire HST system but only with the PQLM process. We started by writing an incremental parser for a certain view grammar and then continued by developing a parser generator. The generator consists of about 1800 lines of code in the PQL language, which is a procedural tree manipulation language [KLM+91b]. The generator produces incremental parse functions for context-free grammars with textable productions.

8.3 Test results

We have tested the incremental parser and compared its performance with parsing from scratch. We have also compared it with the original interpreting parser \(^2\) used in HST. The results show that the incremental parser is faster than parsing from scratch. It should be noted though, that we use the incremental parser to parse from scratch by forcing it to parse also unmodified input tokens. The incremental parser also seems to be slightly faster than the interpreting parser. The interpreting parser consists of 460 lines of PQL code whereas the generated parser for this example consists of 960 lines of PQL code. The tests were run on a Sun Sparc station SLC.

We first tested the parsers on the documents and grammars of Example 19, where we had a very small document. We have the text


and change it to

Fletcher: Murder. Crook.

We parsed the first text with the interpreting parser and the incremental parser (from scratch). Then we made the modification and parsed the modification with the interpretive parser and the incremental parser from scratch.

\(^2\)An interpreting parser is a generic parser that takes a grammar and a text as input. The text is parsed according to the grammar. No parse functions are generated. See [Nik90] for details.
as well as incrementally (only the modified part) with the incremental parser. The results are shown in Table 1.

Table 1

<table>
<thead>
<tr>
<th></th>
<th>Interpreting parser (from scratch)/ms</th>
<th>Incremental parser (from scratch)/ms</th>
<th>Incremental parser (incrementally)/ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>First text</td>
<td>216</td>
<td>66</td>
<td>−</td>
</tr>
<tr>
<td>Modification</td>
<td>133</td>
<td>66</td>
<td>33</td>
</tr>
</tbody>
</table>

The documents were too small (32 and 25 bytes) to give reliable results but the tendency shows that the incremental parser performs well. The fact that the interpreting parser performs differently on the two strings may be due to the fact that the modified string is shorter and therefore parsing it takes less time.

In the next test we used the same grammar, but enlarged the documents to about 30 kB. The author’s name was now about 30 kB and the rest of the documents were the same as in the case above. We also used the same modification in the first document to create the second text. The results are shown in Table 2.

Table 2

<table>
<thead>
<tr>
<th></th>
<th>Interpreting parser (from scratch)/ms</th>
<th>Incremental parser (from scratch)/ms</th>
<th>Incremental parser (incrementally)/ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>First text</td>
<td>2600</td>
<td>2200</td>
<td>−</td>
</tr>
<tr>
<td>Modification</td>
<td>2500</td>
<td>2100</td>
<td>66</td>
</tr>
</tbody>
</table>

Here, it is clear that the parser loses time when it is obliged to read the entire text (about 30 kB) instead of just the modification. The results also show that the parser with generated functions (the incremental parser) is faster than the interpreting parser. In our opinion this clearly shows that the incremental parser is a valuable extension to HST.
8.4 The lazy translator

There are several problems connected with the implementation of lazy translation. We want lazy translation to be invisible to the user. This means that the user prepares a document without knowing that only parts of it are loaded. Consequently, parts must be loaded and saved automatically when the user moves in the document. Edit operations like move, search, etc., affect this process. Another problem is the granularity of the document parts. How much should the translator transform at a time? This problem arises especially with textable productions. Take as an example the production $S \rightarrow Text$. A corresponding parse tree can contain a Text node with an arbitrarily long string. The lazy translator must be able to divide the string during transformation.

8.5 Incremental translation

There are also several problems connected with the implementation of an incremental translator. Incremental translation is heavily based on associated nodes and links between them. It is easy to link a view tree to the corresponding document tree. But a document tree can have several corresponding view trees. A solution might be to link associated nodes in view trees and the document tree circularly. A second problem is the update of the node status. A modification can mark a node inserted, deleted, changed, or unchanged. All nodes in a tree cannot be marked this way after each modification. If the modifications are propagated further to other trees, the status mark must be maintained during the propagation.

8.6 Discussion

As mentioned, we have only implemented a part of the incremental parser. The incremental translator and the lazy translator have not been implemented. The HST system is based on a procedural query language PQL, which has been implemented earlier [KLM+91b]. Our intention was to implement searches, parsing, translations, etc., of a document in this language. In fact some research has been done on structured searches [Kil92]. However,
programming with the PQL language proved to be difficult. Only a subset of the features of a conventional procedural language was implemented in the PQL language. This makes it hard to implement certain data structures. This is also a reason for the large size of the incremental parser generator.

8.7 Future work

We are at the moment thinking of rebuilding the entire system from scratch, possibly moving it to other environments than the Sun/OpenWindows environment. In this case incrementality and laziness could be part of the new system. No definite decisions have been made.
Chapter 9

Conclusion

We have studied some incremental and lazy techniques customized for the preparation of structured documents.

Incremental updates have an advantage over non-incremental updates; we have shown that they are faster. However, all problems are not incremental in nature. Incremental algorithms demand extra space and maintenance of auxiliary data structures. Closely related to the incremental paradigm is the lazy paradigm. We also expect lazy algorithms to be faster than algorithms that compute the entire solution.

Parsing and syntax-directed translation are two areas where we can use incremental and lazy algorithms. We have extended an incremental parsing algorithm by Murching et al. [MPS90] to handle structured texts.

We divide incremental parsing into three subproblems. First, the starting position of the parse tree update must be determined. Our algorithm finds the starting position by comparing the first modified input token with the old parse tree.

The second problem is the method used when reparsing. We have given two different solutions. The first algorithm starts reparsing at the first modified input position and continues until it has passed the last modified input position. It also reparses or traverses input tokens and parse subtrees that have not been modified. We have shown that time complexity does not depend on the size of the modifications. Still, it is bounded by the size of the input and the size of the parse tree. We have also shown that in the case of a simple modification, the time complexity is bounded by the size of the modification and its update. Our enhancement of the parsing algorithm
traverses only modified parse subtrees. The input and the old parse tree can be consistent with each other between modifications in the input. We have discussed how the algorithm could be implemented to skip such local synchronizations. Under the assumption that the algorithm is given a list of modified input tokens as a parameter this latter algorithm has a lower time complexity than the former.

The third subproblem is the termination of a reparse. We have shown that a modification can have a far-reaching influence on the parse tree. A simple modification can restructure the entire parse tree from the first modified token to the last token in the tree. We have given two termination conditions; when the conditions hold, and the last modified input token has been parsed, the incremental parser can stop reparsing. The first condition captures synchronizations between input and the parse tree. Input tokens that have already been parsed before are not reparsed, if they are in the right position. The second condition restricts the development of the parse tree structure. If the structure of parse tree is not modified beyond a certain node, this node can be considered as a possible termination point. The top-down incremental parser is aware of the modifications in the tree structure during traversal from the root towards the leaves, and any modification on a higher level in the tree is always noticed before reaching a lower level.

Incremental translation is closely related to incremental parsing. We have presented an incremental syntax-directed translator that efficiently updates an output parse tree due to modifications in an input parse tree. A naive translator transforms all modified input subtrees. The algorithm is given as input a list of non-overlapping modified subtrees, thereby corresponding to the enhanced version of the incremental parser. We have also presented an enhancement to the incremental translator. A modified input subtree can contain unmodified parts. Our enhanced algorithm is optimal in the respect that it retransforms only modified parts.

The start position of the incremental translator is the first modified subtree, i.e., the first subtree in the list given to the translator. The termination position can be determined from the termination position of the input tree. When the input tree is updated by incremental parsing or translation, a termination position is determined. This termination node has a corresponding position in the output tree (not necessarily its associated node).

We have presented a lazy syntax-directed translator that transforms only part of an input tree. Applied on structured document preparation, the
algorithm computes lazy views of document instances. Lazy and incremental techniques combine very well. A lazy view is updated incrementally. We have presented a scenario of lazy incremental document preparation. In the process, parts of a document instance are automatically transformed into lazy views, which are incrementally updated when needed.

Finally, we have discussed the implementation and test results of a simple incremental parser and some implementation problems of the incremental translator and the lazy translator in the structured text database system HST.
Bibliography


126


[DMM88] Pierpaolo Degano, Stefano Mannucci, and Bruno Mojana. Efficient incremental LR parsing for syntax-directed editors. ACM
Transactions on Programming Languages and Systems, 10:345 – 373, July 1988.


[HKr89] J. Heering, P. Klint, and J. rekers. Incremental generation of parsers. In proceedings of the SIGPLAN ’89 Conference on Programming Languages Design and Implementation, Portland, Ore-


[Qui89] Vincent Quint. Systems for manipulation of structured documents. In J. André, R. Furuta, and V. Quint, editors, Struc-


Index

Advance_newptr 73
Advance_oldptr 73
ancestor 44
annotated grammar 17
applied production 28
associated nodes 88
associated nonterminals 36
associated tokens 57
batch-oriented systems 1
bidirectional link 57
bottom-up parsing 31
bounded incremental algorithm 10
change 23
child of a node 28
Chomsky hierarchy 27
ClimbUpTree 73
complete parse mapping 50
complete preorder mapping 47
constant string 33
context-free grammar 15, 27
calendar-sensitive grammar 27
declarative document preparation
  2
deletion 23, 92
derivation 27
derived string 29
descendant 44

Dirsym 30
dirsym set 30
document flow 19
document grammar 15
document instance 3
document preparation model of HST
  3
document preparation system 1
document preparation 1, 20
document standard 15
document tree 16
document 1, 3
edit distance 41
end tag 57
expanded production 28
First 29
first set 29
Follow 29
follow set 29
formatted view 5
formatting detail 2
frontier of tree 16, 29
general document structure 2
generated string 29
grammar 27
graphical view 6
Helsinki Structured Text Database
System (HST) 15
HST document model 15
incremental algorithm 9
incremental analysis 9
incremental applications 9
incremental compilation 7, 20
incremental document flow 20
incremental methods 9
incremental parsing 5, 20
incremental syntax-directed translation 5, 86
incremental transformation 20
incremental translation 87
incremental tree translation 86
incremental update of an editor 20
incremental update 2, 22
inorder 28
input grammar 37
input token 57
insertion 23, 92
instance of a symbol 29
interactive systems 1
inverse transformation 3
iteration node 75
iteration part 75
iteration production 75
iteration subtree 75
label 28
language of a grammar 27
Last 30
last symbol 30
layout view 5
lazy applications 12
lazy iteration production 105
lazy methods 9
lazy paradigm 12
lazy production 104
lazy transformation 20, 21
lazy view grammar 104
lazy view 4, 103
link 57, 60, 88
LL(1) grammar 32
local synchronization 60
local termination node 80
locally synchronized 61
logical definition 3
logical part 2
Lookahead 74
mapping 44
markup view 5
mindist 43
minimum edit distance 42, 43, 45
newptr 66
Next 75
Next_until 75
node 28
node label 28
nonterminal 27
nonterminal alphabet 27
Nullable 29
occurrence of a symbol 29
ODA 15
oldptr 66
on-line analysis 9
one-shot paradigm 9
output grammar 37
ParseX 69
ParseX 72
variable string 33
view definition 3
view grammar 15, 17
view tree flattening 19
view tree 16
view 3, 17, 37

WYSIWYG systems 1

yield of tree 29
yield relation 26