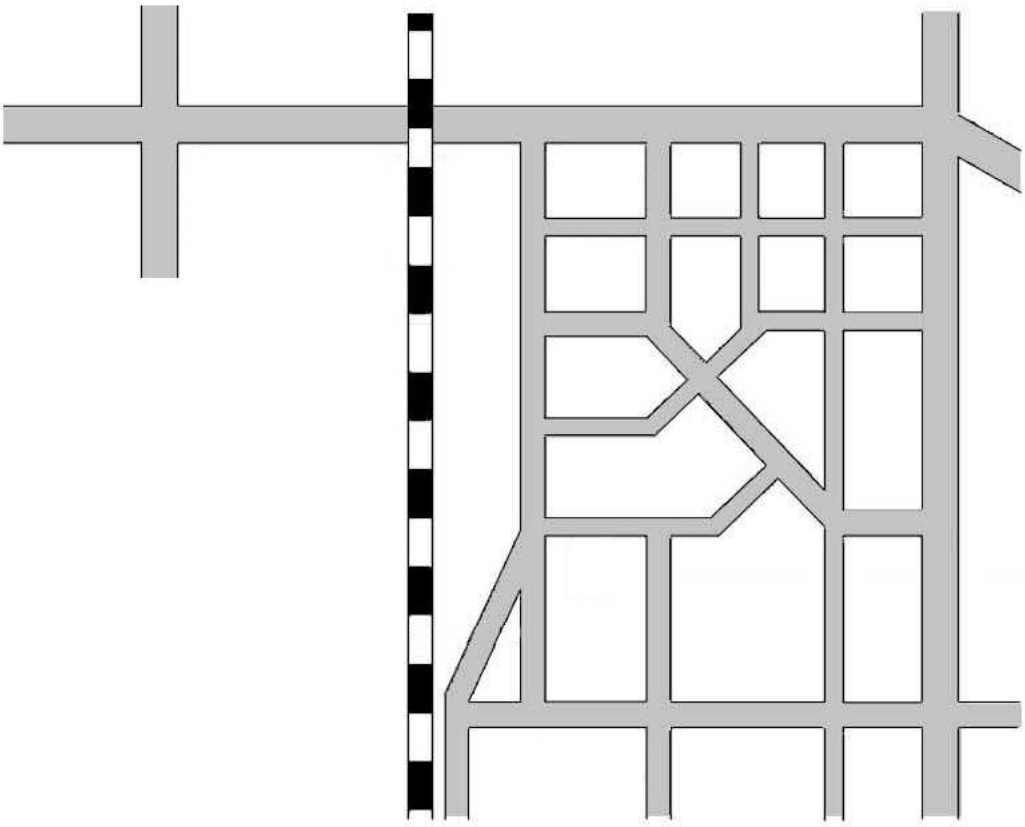
SVG Beautifier

From GEO+ to beautySVG



Abstract

We have several goals for this sixth semester project. We need to design a programming language, that will aid in creating road-maps. These road-maps should then be translated into viewable scalable vector graphics (SVG), which then should be transmitted to a mobile device, where it will be translated into a beautiful presentation for the mobile user.

Upon reading this report the reader will have been introduced to the theory of and surrounding language processors. The reader should have a good overview of the fundamental components that need to be implemented in order to develop a language processor. The reader will also gain an insight into the SVG standard that will influence much content on the Internet for many years to come.

Aalborg University the 28th of May 2004:

Frederik Dannemare

Rene Peder Vestergaard Madsen

Kasper Ørum Nielsen

Peter Sønder

Kenneth Vittrup

Thomas Winterberg

Morten Zinck

Contents

Chapter 1

Introduction

This project focuses on designing and implementing a translator. The idea behind this project is to use a mobile device as the display device for an interactive vector graphics map with roads and buildings etc. The fact that mobile devices have limited resources combined with the high price for transmitting data [14][10], makes it desirable to minimize the usage of bandwidth. To create a functioning system for showing off this functionality, we will also need to design an appropriate client/server architecture.

This report is split into four parts in addition to this introductory. The first part focuses on the theory that is needed in order to understand how the subsequent parts and the final solution is to be structured. This part is split into two minor parts; one that focuses on the different technologies employed in our solution system and one that focuses on the theory of languages and language processors. The second part focuses on describing the domain of the solution in order to better understand in what domain the solution is to operate. The third part presents the designed solution and how the theory of languages and language processors is used together with the presented technologies in order to make the solution workable in the domain. In the fourth and final part we will make a summary of the solution and the entire project.

History of cellular/mobile phones

Cellular phones and mobile communication has its roots back in the 1940's when commercial mobile telephony began. These early mobile telephone systems used push-to-talk operation. Due to the size and cost of the electrical componants, mobile communication did not really take off until after low cost microprocessors and digital switching became available in the 1980's.

Although it was AT&T Bell Labs who introduced the idea of mobile communications in 1947 with the police car technology, Motorola was the first to actually incorporate the technology into a portable device that was designed for use outside of a automobile. Dr Martin Cooper, a former general manager for the systems division at Motorola, is considered the inventor of the first modern portable handset, as he made the first call on a portable cellular phone in April 1973. He made the call to his rival, Joel Engel, Bell Labs head of research[1].

Public trials of a new prototype cellular system made by AT&T Bell Labs was launched in 1978 for 2000 trial customers, and in 1981, Motorola and American

Radio telephone started a second cellular phone system test. In 1983 Motorola officially introduced the world's first commercial portable cellular phone, the Motorola DynaTAC 8000X (see Figure 1.1). Also, it was in 1983 the first American commercial for analog cellular service based on the AMPS (Advanced Mobile Phone Service) operating standard was offered by Ameritech in Chicago[6].

Figure 1.1: Motorola DynaTAC 8000X

Cellular phones as we know them today with respect to size and look, however, was not reality until the mid 1990's. At least in Denmark the late 1990's was the booming years for cellular phones and soon every man and woman on the street had a cellular phone. In contrary, the market for some of the very first analog cellular phones began in Denmark in the late 1980's. In other words: it took a whole decade for the technology to mature sufficiently. Also, the price has dropped tremendeously since the late 1980's which, of course, always plays a big role in the adoption of technology.

Cellular phones are named so, because the phone system is divided into many base stations (cells). This allows for cellular calls to be transferred from base station to base station as a user travels from cell to cell. This helps ensuring that the user gets the best possible signal.

The digital technology primary in use today is called Personal Communi-

cation Service (PCS) and is a 1900 MHz digital service for cellular phones. In the PCS category we find the GSM[1] and TDMA/CDMA[2] operating standards. PCS is better suited than analog or 800/900 MHz digital for sending data such as Internet access or live video streaming.

Digital requires less power and batteries last longer as with analog phones, meaning talk and standby times are higher. But then again, the newest cellular phones of today have colour screens, built-in cameras, FM radios, and Java applications (including Internet applications such as browsers), so maybe battery time is not that much higher in the end, afterall.

Cellular phones have come a long way since its birth. As an example of what to expect from the industry in the nearest future, Motorola recently introduced their state-of-the-art cellular phone named A768 (See Figure 1.2 which features a Linux Operating system running on a 206 MHz processor with 96 MB shared memory. It has a 240x320 pixel TFT touchscreen with 65K colours. There is even handwriting and speech recognition built into the phone, along with many other features such as support for Java applications, built-in digital photo/video camera, bluetooth, MP3 player, PIM functionality, and WAP [5].

Figure 1.2: Motorola A768

[1]Global System For Mobile Communications (GSM) operates at 1900 MHz in the US and at 900/1800 MHz in other countries.

[2]Time Division Multiple Access (TDMA) and CDMA (Code Division Multiple Access).

Chapter 2

Representation and management of spatial data

2.1 introduction to the Data model

There are many ways of representing geographical data as models. Many of the methods described in [3] and [4] are used for helping algorithms to find the shortest path between two locations. We do not focus on this, but on representing the map, for the user of the mobile device, as an easily readable and usable map. Below is a list of valid geographical objects we wish to model:

- road
- intersections of roads
- buildings
- markers
- nature

The underlying data model of these objects must be able to request parts of maps, and give an easy way of limiting the different geographical objects on the map. The use of spatial data are a good tool for solving this problem and are described in the following.

2.2 Spatial division

This section gives a brief overview of the term spatial division and is inspired by [9]. Spatial division is typically used in Global Information Systems *(GIS)* applications, where it is used to give rapid replies to user requests for objects within a certain area. Besides this it is used in collision detection in game programming. Spatial division is a technique to divide a space such that one can search for an object in the space efficiently. This is done by encapsulating each object in the scene in separate boxes[1] and structure them such that one

[1] These are called minimum bounding boxes.

can get rapid reply to inquiries. In Figure 2.1 an example of a space containing objects is illustrated.

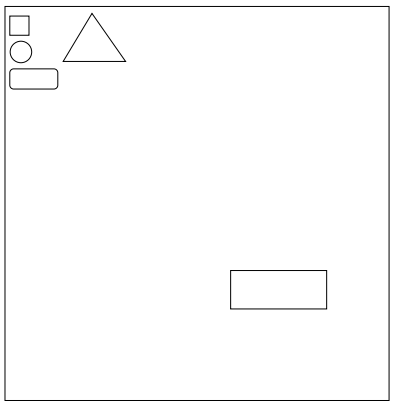


Figure 2.1: A space containing objects

There are two general methods for dividing space: by partioning and by indexing.

2.2.1 Spatial division by partitioning

Spatial division by partitioning *(SDP)* first encapsulates the entire space in one box. Then it divides the space into four equal size subspaces. Subspaces containing more than one spatial object are once again divided into four equal size subspaces and so forth. This process is repeated until each of the spatial objects is contained within its own box. Once this is done the entire space can be represented as a tree with the initial box as the root node. Trees for two dimensional spaces are called an quad-trees and for three dimensional spaces they are called oct-trees. The spatial division of the space shown in Figure 2.1 using SDP is illustrated in Figure 2.2.

The main advantage of using SDP is that the algorithm simply divides the space in equal sizes, which keeps the complexity of the algorithm simple. The disadvantage of using this division is that the tree representation can be highly unbalanced as shown in Figure 2.2. [9]

2.2.2 Spatial division by indexing

Spatial division by indexing *(SDI)* uses minimum bounding boxes to divide the space. The purpose of this indexing mechanism is to create a more balanced tree. Nevertheless one can structure the minimum bounding boxes in an inefficient manner, so that each element in the scene is encapsulated in its own box. This would give a brute force search time where each object in the scene would be tested for membership. This effect is illustrated in Figure 2.3.

This is, however, avoided by building the tree starting from the bottom and ending at the top. First each object in the space is bound to a box. Next each of the boxes is grouped with other boxes by inserting them into a new box. This process continues until one box encapsulates the entire space. Once this is done

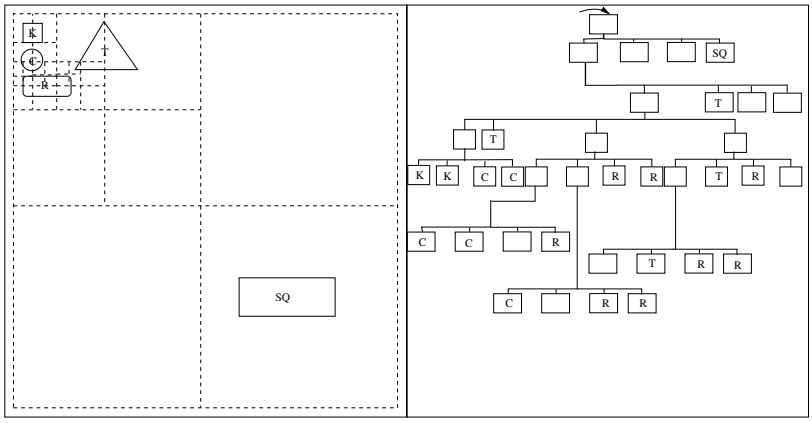


Figure 2.2: Spatial division by partitioning of Figure 2.1

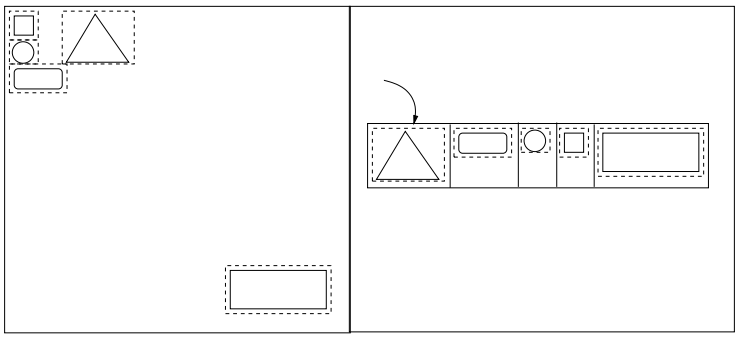


Figure 2.3: Spatial division of Figure 2.1 using brute force.

the space can be represented as a tree with the final box as the root node. The tree is called a balanced tree *(B-tree)*. The process of dividing the space shown in Figure 2.1 is illustrated in Figure 2.4.

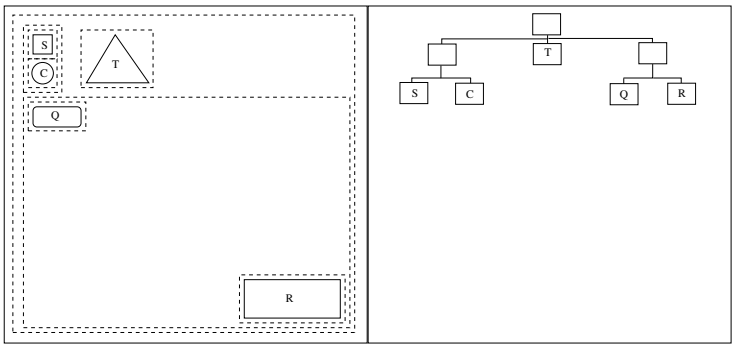


Figure 2.4: Spatial division of Figure 2.1 by indexing.

The advantage of SDI is that the representation of the divided space can be structured as a balanced tree, which improves the worst case search time. The disadvantage is that the algorithm has to evaluate object coherence, which increases the complexity of the algorithm. [9]

2.3 Representations

There are several possible methods to graphically represent a map on a computer display. In computer science a modern way of doing this is using SVG, which is based on Extensible Markup Language *(XML)* described in Appendix A. In the following we first present subsets which describes the technologies relationship. Then a presentation of the SVG and the technology XML Schema used for verification of valid XML code follows.

In order to verify that any code is valid XML code we present a technology for checking this named XML Schema (see Section 2.4.3 for more details). This section elaborates on SVG and XML Schema .

2.4 Subsets

Before we have described the different technologies this section describes their relationships. Here miniSVG and beautySVG are our own subsets of XML. The relationships between the languages relevant for this project are:

$$\text{miniSVG} \subset \text{beautySVG} \subset \text{SVGT} \subset \text{SVGB} \subset \text{SVG} \subset \text{XML}$$

As described, XML contains everything that SVG does and more, so it is the largest set of these languages. The other mentioned languages are therefore all subsets of XML. Not that we use XML directly, but we use a subset of XML. The formula illustrates that SVG is a subset of XML and that SVG has SVGB and SVGT as subsets. These are the standard languages given by World Wide Web Consortium *(W3C)* [22]. To extend this we have defined two new languages, namely miniSVG and beautySVG . As described beautySVG is a subset of SVGT and the miniSVG language is a subset of beautySVG . Both miniSVG and beautySVG will be described later on.

2.4.1 SVG

SVG is based on technologies like SGML, HTML and Cascading Style Sheets *(CSS)*. More recent technologies like Document Object Model *(DOM)* and XML are also major contributers to SVG. All of these technologies are futher described in Appendix A. SVG is a platform for two-dimensional graphics. It is used in web graphics, animation, user interfaces, mobile applications and high--quality design of software. SVG is a strong tool for development of graphics and therefore gets industrial support from firms like *Adobe Systems inc.*, *Canon*, *HP*, *Microsoft* and *Sun Microsystems*. SVG builds upon many other successful standards such as XML, DOM and CSS.

Traditionally images come in two varieties: *raster* and *vector*. A raster image is represented by defining the color of the single pixels that make up the image. A vector image is made of a series of mathematical definitions of simple

shapes like lines and curves. This has the advantage of a vector image not becoming blurry when zooming in on the image. Vector images are therefore ideal for artificial images like drawings, maps etc. On the other hand, a raster image is better when representing real things like an photo and can show the same shapes as a vector image [7].

In 1998 the W3C established a workgroup charged with drawing up proposals for the standard for SVG, the standard can be found at [22]. At this time XML had been approved as the notation for new web languages, so SVG would be an XML application. The group used a wide range of expertise from computer companies involved with graphics software. Especially Precision Graphics Markup Languages *(PGML)* from Adobe Systems inc. and Vector Markup Language *(VML)* from Microsoft was used as the base for SVG [7]. SVG was also based on PostScript *PS* and Portable Document Format *(PDF)*. Adobe's offer to write the first SVG plug-ins for *Netscape* and *Internet Explorer* ensured a good compatibility with the underlying graphic models [7].

Mobile devices have different characteristics in regard to CPU, memory, and color support. To accommodate this, two profiles are defined. A low-level profile called SVG Tiny *(SVGT)*, suitable for highly limited mobile devices, and SVG Basic *(SVGB)* , which is targeted on higher-level mobile devices.

SVGT

SVGT is specified to be a proper subset of SVGB to ensure interoperability between the two, hereby ensuring as much scalability as possible. SVGB is then again a proper subset of *SVG 1.1*, which is the current version of SVG.

SVGB

SVGB allows profiles to describe the different modules, hereby using a smaller number of restrictions or extensions of elements provided by those modules. On the other hand, the "full" profile of SVG 1.1 is the collection of all modules listed in the specification [20].

The advantages of SVG is illustrated in Figure 2.5. On the left, the original image is shown. The middle image shows a zoom of the image when it is vector based and the image on the right shows a zoom of the image when it is raster based. Vector graphics approach clearly has great potential, especially given the fact that it does not take up anywhere near as much memory as the raster graphics approach.

Figure 2.5: Example of raster and vector graphics.[12]

2.4.2 miniSVG / beautySVG

We have defined our own two SVG subsets, miniSVG and beautySVG . These are extremely narrow specifications with regards to what is considered valid miniSVG and beautySVG code. The difference between the two is the styling part. While beautySVG allows styling, miniSVG does not. Below is the description of the two:
- miniSVG allow nothing but the SVG path tag (without styling).
- beautySVG allow nothing but the SVG path tag (with styling).

Refer to Appendix G and H for details on what our XML Schemas will see as valid code.

2.4.3 XML Schema

XML Schema is a language used to describe the contents and structures of documents written in XML format. XML Schema is, in other words, an XML-based alternative to DTD (described in Appendix A). Although XML Schema serves the same purpose as the DTD language, it provides a much more powerful and flexible way of specifying constraints for an XML document.

One of the main weaknesses of DTD is the lack of support of other data types than character strings. XML Schema provides a wider range of primitive data types, including string, boolean, real, decimal, and integer. Furthermore, these built-in data types can be extended in many ways thereby creating user defined data types for complex constraints on XML documents. Figure 2.6 shows how to extend the simple string type by adding restrictions. These restrictions are expressed as regular expressions.

```
<xsd:simpleType name="path_style">
 <xsd:restriction base="xsd:string">
  <xsd:pattern value="(font-size:\d{2};)?fill:(blue|black);?"/>
 </xsd:restriction>
</xsd:simpleType>
```

Figure 2.6: Example of XML Schema

Whether to use DTD or XML Schema depends on the task at hand. In this project we are working with SVG, which is a subset of XML. Therefore, we need to validate our XML document with some form of validating rules. This can be done by DTD. DTD and XML Schema are very alike, but there are some noticeable differences. One is that XML Schema can deal with namespaces, which DTD cannot. Another is that DTD does not have XML syntax and offers only limited support for types.

One of the advantages of using XML Schema is that we need to check the boundaries of the map. Another is that we need to define our own types. A type could be an extended house, which extends a normal house, because it has six corners instead of the usual four. Another reason is that DTD uses a different syntax than XML Schema , and since XML Schema ' syntax is great for XML, it is a good idea to use XML Schema [23].

Chapter 3

Language theory

The primary concern of this project is translation and interpretation. When reading this chapter keep in mind that our focus is on the the creation of a translator, but also modification and use of an abstract interpreter. The abstract interpreter can interpret beautySVG files, because it is made up of a SVG viewer and the interpreter it runs on. This abstract interpreter is made up of a SVGT viewer and a Java Virtual Machine.

We begin this chapter of theory by giving an introduction to tombstone diagrams. After that we proceed to syntax theory, which is the subsection 3.2. This is followed by contextual analysis, and we finish this chapter by taking a look at semantics.

3.1 Tombstone diagrams

Tombstone diagrams consist of a set of "puzzle pieces" we can use to reason about language processors and programs. The four pieces that make up these pieces are displayed on Figure 3.1. Piece *(a)* illustrates a program named α written in language δ. Piece *(b)* illustrates how a piece is capable of translating program α to program β. This piece runs on δ. Piece *(c)* shows an interpretor between language α and β. Finally the piece *(d)* shows a system capable of executing programs in language δ.

These pieces can be combined as long as the languages match. Upon starting a project a limited set of pieces is available. More pieces can be made using the translator piece *(b)*.

3.2 Syntactic analysis

Syntax is the term that is used when dealing with the form of programs. The syntax of a language defines which symbols are considered legal symbols in that language. Phrases written in a given language must consist of the allowed symbols and sub phrases of that language. Examples of such phrases are commands, expressions, declarations and even complete programs.

Since the syntactic analysis phase brakes down into the sub phase of scanning, parsing and creation of abstract syntax trees, these sub phases are dis-

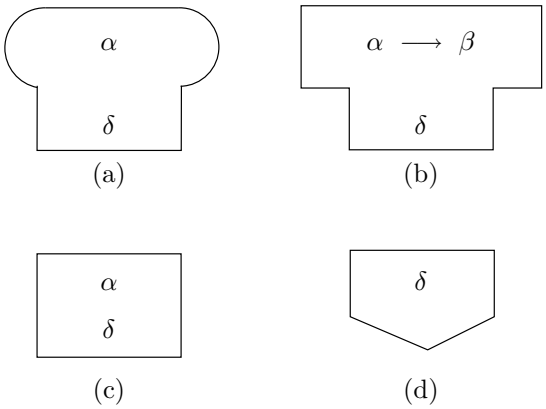


Figure 3.1: Different components for tombstone diagrams

cussed in the following subsections. Before discussing those topics we shall take a look at grammars which are relevant to syntactic analysis.

3.2.1 Tokens and lexemes

Before we proceed to discussions on grammars, scanning and parsing, we must explain the concepts of lexemes and tokens.

Lexemes are made up of terminal symbols from the programming language and belong to some category of token in the language. Lexemes are not abstract. They are instances of tokens, which means that they belong to a token category. If a token has the kind "identifier", it could for example have the instances (or lexemes) "sum", "total", "person" or "address".

When lexemes are being classified according to the kind of each token the only rule for two lexemes being of the same kind is that they can be interchanged in a program without affecting the program's phrase structure. Given the description just given of lexemes, a program can be considered a string of lexemes.

The descriptions of lexemes are often not included in the formal definitions of the syntax of the programming language. Since lexemes are the lowest--level units in a programming language, they are often described in a lexical specification, which is separate from the language's syntactic description. This is done to simplify the formal description of the language.

Tokens make up the lowest level of abstraction and are therefore the level of abstraction closest to lexemes. Tokens are categories of lexemes, and in some cases a token can have only a single lexeme, which is the case for the plus operator "+". It could have a token category (often refered to as the token's kind) named "plus_op". They are identifiers, literals, operators, keywords and the punctuation symbol ".". Compilers and interpreters use tokens to determine a program's phrase structure and compilers in particular use the tokens to build an abstract syntax tree (described below).

A token is completely described when its kind and spelling have been defined. Spaces between tokens and comments only exist in a source program to make

it easier for humans to read and understand the source program, so they are discarded.

To compare and visualize this relationship between lexemes and tokens, examples of corresponding lexemes and tokens are listed Table 3.1.

Lexemes	Tokens
2	int_literal
17	int_literal
+	plus_op
*	mult_op
index	identifier
count	identifier

Table 3.1: Example of corresponding lexemes and tokens.

The term phrase structure has been mentioned a couple of times here, so before we move on, we give an example of the phrase structure of the short sentence "the cat sees a rat." written in an extremely simple language to help the reader visualize what a phrase structure is. The language of the grammar is shown on top of Table 3.2 while the visualization is shown on bottom of Table 3.2.

Sentence	::=	*Subject Verb Object*
Subject	::=	*I* | *a Noun* | *the Noun*
Object	::=	*me* | *a Noun* | *the Noun*
Noun	::=	*cat* | *mat* | *rat*
Verb	::=	*like* | *is* | *see* | *sees*

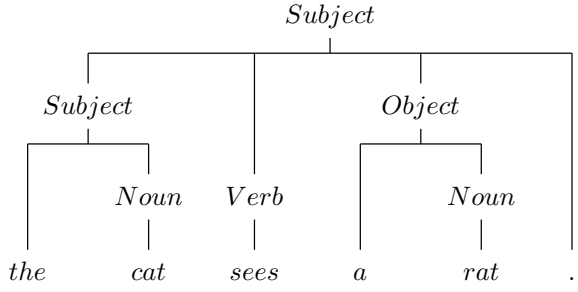


Table 3.2: A simple grammar and a break-down of the sentence "The cat sees a rat."

3.2.2 Grammars

When the grammar of a programming language needs to be specified it can be done formally or informally. If done informally the specification will, in general, be easier to understand. This is because it can be written in plain English or in another natural language, thus it can very easily turn out to be ambiguous and inconsistent. This means that it can easily be misinterpreted. By writing a formal grammar misinterpretation can be avoided, but the specification will

be harder to read and understand, especially for people not familiar with the notation in which the specification is written; ex. regular expressions, Backus-Naur-Form and Extended Backus-Naur-Form.

A context-free grammar can be used to specify a programming language in a formal manner and consists of the following elements:

- A finite set of terminal symbols (called terminals) – These are the atomic symbols we enter at the keyboard, when we compose a program in a programming language; ex. "*while*", "*:*" and "==".
- A finite set of nonterminal symbols (called nonterminals) – The nonterminals of a language each represent their own class of phrases in the language; ex. "*Program*", "*Command*" and "*Expression*".
- A start symbol – This represents the principal class of phrases in the languages. The start symbol is often "*Program*".
- A finite set of production rules – These are the rules that define how phrases may be composed from terminals and sub phrases.

A grammar is a collection of production rules and these rules describe what the different abstractions in the grammar are composed of.

Regular expressions

A *regular expression* generates a set of strings of terminals, and all those strings make up a language, appropriately called a regular language, but regular expressions can not generate complex languages, since they do not allow self-embedding. Identifiers and literals of programming languages can be described using regular expressions.

The regular expression form of notation uses four very important and practical operators. These are:

- "|" which separates alternatives.
- "*" which is an iteration symbol and indicates that the previous item may be repeated zero or more times.
- "(" and ")" which show groupings.

In Table 3.3 are some examples of these operators in use.

foo | bar	generates {foo, bar}
foo*	generates {foo, foofoo, foofoofoo, ...}
(foo | bar)*	generates {foo, bar, barfoo, foofoobarfoo, . . . }

Table 3.3: Example of regular expressions.

Backus-Naur-Form

Backus-Naur-Form allows more complex languages to be generated than regular expressions does. With Backus-Naur-Form and also Extended-Backus-Naur-Form, which will be described shortly, we are allowed to use self-embedding, which enables us to write recursive production rules in our grammars.

A Backus-Naur-Form description is a collection of grammatical rules. Such rules are made up of abstractions and lexemes as is explained below. Abstractions in Backus-Naur-Form are called nonterminals, and lexemes and tokens are called terminals.

A meta symbol is a symbol that is only used for notational purposes. They are not terminal symbols in the syntactic definitions they appear in. Should one or more meta symbols also happen to be terminals in the language they describe, they can be underlined to separate them from the terminals.

The meta symbols in Backus-Naur-Form are:

- ::= meaning "may consist of"
- | meaning "or alternatively"
- < and > angle brackets are used to show category names

With these meta symbols we can formulate anything that we can formulate with Extended-Backus-Naur-Form, which will be described shortly, but grammars created with Backus-Naur-Form are a little bit more confusing for humans to read, so we prefer to use Extended-Backus-Naur-Form.

All Backus-Naur-Form production rule is formulated as $N ::= X$, where N is a nonterminal and X is an extended regular expression, which is an regular expression constructed from both terminals and nonterminals.

Examples of Backus-Naur-Form rules written using the meta symbols just described follow. Notice that both examples below also show that the angle brackets < and > are used to show the names of nonterminals.

Below is an example of the use of the ::= operator. Here the abstraction $< assignment >$ is defined as a variable that is assigned the resulting value of an expression:

$$< assignment > ::= < var > = < expression >;$$

Note that in production rules the left-hand side of the rule, which in this example is $< assignment >$, is refered to as LHS. The right-hand side of the rule, which is $< var > = < expression >;$, is refered to as RHS.

The rule below shows how the | operator can be used to define the different possible definitions that the abstraction ¡command¿ can evaluate to. Here the abstraction $< command >$ can evaluate to more than one possibility:

$$< command > ::= begin < command > end; \mid if < expression > then < command >;$$

The rule above tells us that $< command >$ may consist of either $begin < command > end;$ or $if < expression > then < command >;$.

Extended Backus-Naur-Form

Extended Backus-Naur-Form is fundamentally just Backus-Naur-Form with the power of regular expressions added to it. Backus-Naur-Form can express the exact same context free grammars as Extended Backus-Naur-Form, but it takes a bit more work with regular Backus-Naur-Form and the meaning of the grammar written in Backus-Naur-Form is not as transparent as it would be if it had been written in Extended Backus-Naur-Form. Several production rules written in Backus-Naur-Form can often be compacted into a single rule. Because of this we naturally prefer to use Extended Backus-Naur-Form.

There are several different variations of Extended Backus-Naur-Form, but we will only mention features that most of them have in common. By adding the abilities of regular expression in our expressive capabilities, we automatically add the three new meta symbols "*", "(" and ")".

In Extended Backus-Naur-Form parenthesis can be used for grouping, but note that in the rest of the text about language theory, bold parentheses, **(** and **)**, are terminal symbols in a programming language. The meta symbol for alternatives, which is "|", can now also be used inside parenthesis. Lastly, the meta symbol "*" can be used for iteration.

The uses of the meta symbols are illustrated below with an Extended Backus-Naur-Form grammar.

$$
\begin{aligned}
&\textit{Expression} ::= \textit{primary-Expression}\ (\textit{Operator}\ \textit{primary-Expression})^* \\
&\textit{primary-Expression} ::= \mathit{Identifier} \mid \textbf{(}\ \textit{Expression}\ \textbf{)} \\
&\mathit{Identifier} ::= a \mid b \mid c \\
&\textit{Operator} ::= + \mid - \mid * \mid /
\end{aligned}
$$

The grammar can generate self-embedded expressions. This is possible because *Expression* and *primary-Expression* are mutually recursive. The grammar generates such strings as the following and many more:

$$
\begin{aligned}
&a \\
&b \\
&a - c + b/a \\
&c * (a + b) \\
&a - (a/(b + c))
\end{aligned}
$$

The Extended Backus-Naur-Form notation mentioned above is known as the Watt and Brown notation (named after David A. Watt and Deryck F. Brown), but there is also an ISO Extended Backus-Naur-Form notation, and it is important to be familiar with it. In the ISO notation braces ("{" and "}") are used to denote iteration and square brackets ("[" and "]") are used to denote optional parts.

The additions to Backus-Naur-Form are relatively simple but do, to a fairly large degree, increase both the readability and writability of grammars.

Grammar transformations

Transformations must sometimes be performed on grammars to make them unambiguous. Most parsers are recursive-descent parsers so it is a good idea to design a grammar with such a parser in mind. To ensure that the grammar,

that has been created, conforms to the needs of such a parser one should design a so called *LL(1)* grammar, so lets just give a brief overview of what is required of a grammar to be LL(1):

For a grammar to be LL(1) it must adhere the following conditions, where X and Y are regular expressions:

- Where the grammar contains X | Y as a RHS, starters[[X]] and start starters[[X]] and starters[[Y]] must be disjoint.
- Where the grammar contains X*, starters[[X]] must be disjoint from the set of tokens that can follow X* in this particular context.

The way to transform a CFG into an LL(1) grammar is to:

1. Remove all (indirect) left recursion.
2. Left factorize all the necessary rules.

The existence of left recursion in a grammar makes top-down parsing impossible because left recursion causes the parser to make recursive calls endlessly. Consider the rule $N ::= N + M$. If a recursive-descent parser was to parse N, then it would endlessly exchange N by $N + M$ since it goes from left to right and therefore always meets a new N!

The first transformation that will be mentioned is therefore elimination of left recursion. This is an example of the elimination of left recursion:

$$N ::= X \mid NY$$
$$N ::= X(Y)^*$$

The example above is of removal of direct left recursion as opposed to indirect left recursion, but the indirect variant is just as troublesome:

$$A ::= BaA$$
$$B ::= Ab$$

Indirect left recursion such as illustrated above is, of course, less obvious than direct left recursion, but equally serious.

The next transformation is left factorization. Left factorization is all about compacting alternatives. Given that X, Y and Z are regular expressions, the two expressions XY and XZ can be rewritten to $X(Y|Z)$.

Right factorization on a grammar can also be performed, but it is not very useful in practice, so it is not covered here.

The last grammar transformation, that will be mentioned, is substitution of nonterminal symbols. The idea is that by removing one or more of the nonterminals in the grammar, it becomes simpler and easier to read.

If we have a production rule $N ::= X$ then we might try to remove all occurrences of N in our grammar by exchanging them with X. Note thought that if there are only a few occurrences of N and if X is not complicated then leaving N in place may offer a grammar that is easier to understand.

Starter sets

The starter set of a regular expression is the set of terminals that can start one of the strings that can be generated by the regular expression. Given an regular expression named X its starter set is written $starter[[X]]$.

These simple example illustrates starter sets:

$starters[[foo|bar]] = f, b$
$starters[[(foo)^*bar]] = f, b$

The following is a precise and complete definition of starter sets, where X and Y are arbitrary regular expressions:

$starters[[\varepsilon]] = \{\}$
$starters[[t]] = t$ where t is a terminal.
$starters[[XY]] = starters[[X]] \cup starters[[Y]]$ if X generates the empty string ε.
$starters[[XY]] = starters[[X]]$ if X does not generate the empty string ε.
$starters[[X|Y]] = starters[[X]] \cup starters[[Y]]$
$starters[[X^*]] = starters[[X]]$

Starter sets can be generalized by allowing the starter set of a nonterminal N to be defined as $starters[[N]] = starters[[X]]$.

Below is shown a simple grammar, which is followed by the corresponding example of illustrating the use of starter sets:

$Expression ::= primary\text{-}Expression\ (Operator\ primary\text{-}Expression)^*$
$primary\text{-}Expression ::= Identifier \mid (Expression)$
$Identifier ::= a \mid b \mid c$
$Operator ::= + \mid - \mid * \mid /$

$starters[[Expression]] = starters[[primary\text{-}Expression\ (Operator\ primary\text{-}Expression)^*]]$
$= starters[[primary\text{-}Expression]]$
$= starters[[Identifier]] \cup starters[[(\ Expression\ )]]$
$= starters[[a \mid b \mid c]] \cup \{(\}$
$= \{a \mid b \mid c \mid (\}$

3.2.3 Scanning

Scanning is also known as lexical analysis and is the process of transforming the source program into a stream of tokens, which is then parsed by the parser. Since the scanner must create a stream of tokens from the source program it must be able to recognize the language that the source program is written in. Scanning is therefore analogous to parsing but it works at a lower and finer level.

When a lexical grammar has been constructed, a lexicon of the source language can be created using it. The terminals of the lexical grammar are the individual characters in the source program and the nonterminals are tokens, separators, identifiers, integer-literals, operators and comments.

It is very important to note that the lexical grammar must not use self-embedding.

A scanner can be developed in a similar fashion to a parser. The way to develop a scanner is to:

1. Create the lexical grammar in Extended Backus-Naur-Form and apply the necessary grammar transformations to it.
2. Use each production rule N ::= X to create a new method in the scanner, whose body is determined by X.
3. Make the scanner consist of:
   - A private variable *currentChar*;
   - The private methods *take* and *takeIt*;
   - The private scanning methods developed in step two, enhanced to record each tokens kind and spelling;
   - A public method in the scanner that scans '$Separator^*\ Token$', discarding any separators that may exist before it finds the next token, which it returns.

When the scanner calls one of its methods *scanN* the variable *currentChar* should hold the first character in an N-phrase. When *scanN* is exited *currentChar* should hold the character right after the last character in the N-phrase.

The two methods *take* and *takeIt* work as follows. *take* starts by fetching the next source program character. If this character corresponds to the argument character, that *take* takes, then it is stored in *currentChar*. *takeIt* just fetches and stores the next source program character in *currentChar*.

When the method *scan* is called it gets the next token from the source program. If any separators precede that token they are discarded. In most programming languages separators can freely be used between tokens in the source program.

Finally, the scanner's performance can be optimized by eliminating one or more of the nonterminals, if appropriate. This optimizes because there will be less methods of the form *scanN*, so less calls are made when scanning. Typical nonterminals to remove are *Identifier*, *Integer-Literal*, *Operator* and *Comment*. Removing these four nonterminals will for example mean that the methods *scanIdentifier*, *scanIntegerLiteral*, *scanOperator* and *scanComment* are removed.

3.2.4 Parsing

Parsing a source program to discover its phrase structure is the main part of syntactic analysis. When parsing, tokens are treated as terminal symbols. The scanner will not examine the spelling of the tokens, because their spelling does not help define the phrase structure of the source program. Only the tokens' kinds matter here. The spelling of the tokens is not examined until contextual analysis and/or code generation.

When the parser receives an input string, the parser tries to recognize the string to determine if it is a sentence of the grammar. If successful it determines

the sentence's phrase structure. The phrase structure can be represented by a syntax tree if needed or by some other data structure, but an interpreter does not need a syntax tree since it interprets and executes one instruction of the source program at a time. Two approaches to parsing will be explained in this section as well as a real top-down parsing algorithm known as recursive-descent parsing. The names top-down parsing and bottom-up parsing come from the order in which these methods build a syntax tree.

Bottom-up parsing

The idea with bottom-up parsing is to build the syntax tree starting with the terminal nodes of the tree and ending with the creation of the root.

To explain the method we assume a set of production rules in our grammar, which are of the form $N ::= X_1 \ldots X_n$, where N is a non-terminal symbol in our grammar and each X_i is a terminal or non-terminal symbol in our grammar. We also assume that the start symbol of our grammar is S.

This parsing method examines the input string's terminal symbols from left to right.

When it has read a sequence of symbols and sub-trees of the syntax tree that match the right-hand side of a production rule in our grammar, it creates a new N-tree, which is a sub-tree of the final syntax tree. This new tree is then available to our parser as it continues to read input symbols from the input string.

If the parser eventually has reduced the entire input string to an S-tree then the parsing was successful and the S-tree is the finished syntax tree.

At each step a bottom-up parser must take prior steps into consideration along with the next input terminal symbol before it decides on what to do next.

Top-down parsing

Top-down parsing takes the completely opposite approach of bottom-up parsing and starts by building the root of the tree and then working its way down the tree, finishing off by creating the terminal nodes. We assume the same set of production rules in our grammar as in the description of bottom-up parsing.

Just like with bottom-up parsing this parsing method examines the input string's terminal symbols from left to right.

Since this method starts at the top, the first part of the syntax tree that is created is the root node, which is labeled by the start symbol S.

The first input symbol is then examined and depending on that symbol a production rule is chosen and based on that rule some stubs are created under the root node with branches connecting the new stubs to the root node.

The stub in the tree that is currently considered at any point during parsing is the leftmost stub that is not connected to anything below it.

If at any point the terminal symbol, that is currently under consideration, can be legally connected, according to the grammar, to the leftmost stub by creating a new branch between them, then this is done.

If the parser at any point needs to create a new stub under the stub that is currently under consideration, because of a production rule it has just chosen, then this is also done.

When a terminal symbol from the input string has been connected to the syntax tree, the parser moves on to consider the next terminal symbol in the string. It may be, though, that the parser needs to consider both the terminal symbol it is currently considering and the terminal symbol after that in the input string to pick the right production rule, so the parser must be able to do this.

Using these rules the parser will eventually have reduced the whole input string to a syntax tree. If at any point during parsing the parser does not find a production rule that matches the terminal symbol under consideration it must create a syntactic error.

Note that if the grammar used is ambiguous then the parser would have to guess which way is the correct way to build the syntax tree. If it makes the wrong choice it could end up in a situation where it can not possibly parse the whole input string and so it fails. It is therefore a good idea to make ones grammar unambiguous, meaning that any sentence in the grammar has exactly one syntax tree. This makes it a lot easier to write a parser since it then does not have to make intelligent guesses about which way to build the syntax tree.

Recursive-descent parsing

The recursive-descent parser is an actual algorithm, which uses the top--down parsing approach. It is an effective algorithm and is relatively easy to understand.

This parsing algorithm has a method for each non-terminal symbol N in our grammar. We will for practical reasons use the naming convention *parseN* for these methods. Any *parseN* method can parse a corresponding N-phrase, and all the *parseN* methods can together parse complete sentences. This algorithm also has a method called *accept* which accepts the terminal symbol currently in the *currentTerminal* variable without checking it. This method provides the benefit of being able to accept a terminal symbol when it is known in advance that the terminal symbol will be approved. This enhances the performance of the parser. When the parser exits the *accept* method the variable *currentTerminal* must contain the terminal symbol following the terminal symbol in *currentTerminal* upon entering the method *accept*.

The algorithm works as follows: It has a variable, which we here call *currentTerminal*, that holds the terminal symbol the parser has currently gotten to and all the *parseN* methods mentioned has access to that variable. The variable *currentTerminal* must contain the first terminal symbol in an N-phrase when the parser enters a *parseN* method and when the parser exits the method, *currentTerminal* must contain the first terminal symbol after the last terminal symbol of the N-phrase. If these conditions are not met the parser must create a syntactic error.

The name of the algorithm comes from the facts that it is a top-down algorithm and if the production rules of the grammar are mutually recursive then the parsing methods are too.

Abstract syntax trees

Abstract Syntax Trees *(AST)* are not always necessary to construct. A one pass compiler with a recursive-descent parser, for example, determines a pro-

gram's phrase structure implicitly during recursion and constructing an AST therefore becomes unnecessary. Iterative interpreters do not use ASTs at all. The iterative interpreter just begins to execute a program immediately, instruction by instruction, and has no knowledge of a program's structure during execution.

The way to construct an AST is conceptually fairly easy. Given that a recursive-descent parser is used, and given that the parser has a set of *parseN* methods that parse the N-phrases of the language, the additions that need to be made are to make each *parseN* method construct an AST for that N-phrase by combining the subphrases' ASTs (or creating terminal nodes where appropriate) and finally returning their own N-phrases. This will lead to the final AST being built.

3.3 Contextual analysis

When a program has been parsed it should be checked whether the program conforms to the contextual constraints of the language in which it is written. The contextual constraints are used to determine if a given phrase is correct in the context it exists in. Contextual constraints consist of a set of scope rules and type rules that apply in the language (assuming the language is statically typed and has static bindings, which is very typical).

Scope rules are used to determine the validity of declarations in different parts of a program, and to determine which occurrences of identifiers are linked to which declarations.

Type rules are used to determine the types of expressions in a program in order to check their validity.

3.3.1 Tasks in contextual analysis

Scope rules and type rules lead us to to tasks in contextual analysis: identification and type checking.

Identification is the process of figuring out which occurrences of identifiers relate to which declarations in a source program. To relate identifiers to their declarations in a source program a table can be created to hold the identifiers along with an attribute for each identifier that points to that identifier's declaration or holds all the necessary information for the contextual analyzer. This attribute can be used to find the identifiers' scopes. These relations need to be found and used often when a source program is processed by a language processor, so the way in which a language processor finds the declarations for the identifiers is important for performance reasons. It gets more serious the bigger the source program is because the identifiers are used in more places and even more identifiers are created. Searching a tree each time an identification needs to be made would be inefficient. Using a table is a faster and more efficient method.

Scope rules separate a source program into different blocks where different declarations of identifiers are valid. The phrases in a program implicitly create these block structures. The organization of the table mentioned previously, which had identifier-attribute pairs, will be influenced by the source language's

block structure. There are three types of block structures, which will be described here before we describe the task of type checking.

The first block structure type is the monolithic block structure. With this type of block structure all the declarations have global scope, meaning they are valid in all parts of the program. This also makes the table of identifiers and attributes simpler, since managing different scopes for identifiers is not necessary.

If, on the other hand, a programming language uses a flat block structure, then a program can be partitioned into several blocks of local scope that are disjoint. They may not, however, be nested. Beyond the local scopes of these blocks, there is also the global scope, which is known from the monolithic block structure. Note that the local scope blocks are nested in the global scope "block" which covers the entire program. A declaration made in a local block is only valid in that block, but a declaration made with global scope is valid in the entire program as in the monolithic block structure.

The table of identifiers naturally has to be extended to handle different scopes for identifiers, and this is done by associating a new attribute with each identifier in the table to indicate whether it has global scope or local scope. Locally declared identifiers should only exist in the table during the time when the block, in which they are declared, is examined by the contextual analyzer. This allows identifiers with the same name to be declared in different local scope blocks.

The last block structure type we will mention is the nested block structure. This type of block structure is actually much like the previous one but it is now also allowed to have local blocks inside other local blocks. This means that there can be many levels of scope. The global level is considered to be level 1; the local scope blocks immediately on top op the global scope is level two; the local scope blocks immediately on top of those level two scope blocks are level three and so on. This gives us the nested structure.

The table of identifiers has to be modified again to accommodate all of these scope levels. Whereas the flat block structure uses an attribute showing that an identifier is global or local, the table for the nested block structure instead has to have an attribute that shows the number of the scope level it is on.

When identification has been performed, type checking must be performed to determine whether the expressions and other relevant phrases, that have types, in the source program evaluate to the expected types. This can be done without running the program if the programming language is statically typed.

For statically typed languages, type checking is easily done with a bottom-up approach by starting with determining the types of the literals and identifiers and then moving up through progressively larger subexpressions. If there are types that are made up of several other types, in which case they are called component types, then these types have to be represented by trees.

Type equivalence can be handled in several ways and can be either structural equivalence or name equivalence. If handled as structural equivalence then two types are equivalent if their structures are the same, but if name equivalence is used then two types are only equivalent if the pointers to the objects are equal, which means that they point to the same object in memory. Every occurrence of a type constructor therefore creates a new and distinct type, because a new object is created in memory.

3.4 Semantics

The term semantics refers to the meanings of programs. The way in which the semantics of a language should be defined is not clear cut. Several methods exist for doing this. Before naming the methods, know that the subject of semantics is quite complex and lengthy and entire books could be written about each of the methods we will mention. For that reason we will only give short overviews of the methods used for handling semantics.

In the example sentence below the syntax is exactly what you see. It is a definition of how a while loop should be built up so that it is formulated correctly. The semantics, however, define the meaning of the sentence, so the semantics would tell us: "In a while loop, the boolean expression is evaluated, and if it evaluates to true the statement is executed. Then control is implicitly returned to the boolean expression again and it is evaluated once more. This continues until the boolean expression evaluates to false causing the while loop to be exited.".

$$\textbf{while}(< boolean_expression >)< statement >$$

There are two categories of methods for describing semantic: static and dynamic semantics. The methods in these two categories are described below in the appropriate subsections.

3.4.1 Static semantics

Static semantics are those semantic rules that can be checked at compile time, as a result of their static nature. Some semantic rules are very hard to express in BNF notation and some are impossible. Type compability rules, for example, are difficult to express. That is why attribute grammars where invented.

Attribute grammars is a well known mechanism used to both describe and checking the static semantics of programs is known as attribute grammars. The method was designed to check both the syntax and semantics of programs.

Attribute grammars work by associating attributes with the symbols of a grammar. Values can then be assigned to the attributes. Attribute computation functions, often refered to as semantic functions, are also associated with the rules of the grammar and these functions specify how the values of the attributes are computed. Lastly, predicate functions are associated with grammar rules to state some of the syntax and static semantic rules of the language. So, attribute grammars, attribute computation functions and predicate functions are the three tools that make attribute grammars work.

Additional features must be mentioned in order to more precisely define attribute grammars. For each symbol X in a grammar there is attached a set of attributes $A(X)$. $A(X)$ is itself devided into two disjoint sets:

$S(X)$ is the set of synthesized attributes, which are attributes used to pass semantic information up a parse tree.

$I(X)$ is the set of inherited attributes, which are attributes used to pass semantic information down a parse tree.

With each grammar rule is associated a set of attribute computation functions functions and a set of possibly empty predicate functions over the attributes of the symbols in the grammar. A synthesized attribute's value on a parse tree node depends on the attribute values of that node's children nodes. An inherited attribute's value on a parse tree node, on the other hand, depends on the attribute values of that node's parent node and sibling nodes.

The predicate functions are boolean expressions on attribute sets. For a derivation to be allowed and valid with an attribute grammar, all predicates associated with every nonterminal must be true. If one or more predicates are false, then a syntactic or static semantic rule has likely been broken.

The parse tree that is built using attribute grammars is based on the underlying BNF grammar as before. Each node in the tree has a set of attribute values attached to it, but any given node's set might be empty. A parse tree is called fully attributed if all the attribute values in a parse tree have been computed. A parse tree is not necessarilly completely built before attribute values are computed.

Attribute grammars can easily get out of hand if a very formal approach is taken on a programming language. The high number of attributes and semantic rules that will be created will make the grammar hard to read and write.

This completes the brief overview of attribute grammars and static dynamics. Next up is a look at methods used to handle dynamic semantics.

3.4.2 Dynamic semantics

Three methods, that can be used to describe dynamic semantics, are mentioned here. The first method of handling semantics, that we will mention, is called operational semantics. With this method a program's meaning is defined by its behavior when it executes on a machine. This is done by describing the effects of the implementation language's constructs on a machine's state (which is made up of registers and memory locations, including condition codes and status registers). A thourough examination of the operational semantics of a statement executed on a machine therefore means examining the state of the machine before and after executing the statement. The differences between those two states define the operational semantics. Operational semantics is not based on mathematics but on lower-level programming languages and if one is not careful circularities in the definitions of concepts can occur, meaning concepts are indirectly defined by themselves.

Another way to define semantics is to use axiomatic semantics, which is based on formal logic. This method was invented to aid in the task of proving the correctness of programs. It can be quite valuable to use such correctness proofs to prove that a program does what it is supposed to. When prooving that a program does what it should, each statement in it must be preceeded and followed by a logical expression that specifies the constraints on the program's variables. Predicate calculus is used as the notation to describe the mentioned constraints and the language itself. Constraints can often be express by boolean expressions. The problem with axiomatic semantics is that it requires an axiom or an interference rule to be defined for each statement type in the programming language. Some of these axioms and interference rules can be very difficult to create, and if one chooses to design the programming language to be easy to

work with when using axiomatic semantics, then the language will end up being too simplistic and lacking in expressive power.

The last method we will mention is denotational semantics, which is the most widely known method for describing the meaning of programs. Entities in the language are converted into mathematical objects, which then represent the meanings of the language's entities. A function is also defined for each language entity, which must map instances of that entity onto instances of the corresponding mathematical object. Denotational semantics is based on recursive functions and the conversions are made with the appropriate recursive functions. The reason for using mathematic objects to represent the entities is that mathematical objects have very precise definitions and have very rigorous rules for the manipulations allowed but language entities often don't in comparison at least. The trouble with this method is that the mathematical objects and mapping functions are very difficult to create. Because the mathematical objects denote the meaning of their corresponding syntactic entities the method has been given the name "denotational". When a complete system has been defined for a language, it can be used to determine the meanings of programs in that language.

Chapter 4

Interpreters and translators

Interpreters and compilers fundamentally approach interpretation and execution of source programs in completely opposite ways. A compiler translates the entire source program into machine code (called the object program) before the compiled program is executed. The way an interpreter goes about it is to interpret and execute the statements in the source program one by one. This means that the interpreter will not be able to execute a program at full machine speed. Whereas a compiler prepares the entire source program for running on a machine before it is executed, an interpreter begins running the source program immediately.

An interpreter is a program that runs another program in three steps by fetching, analyzing and executing the source program instructions into the interpreters own language. For comparison know that a program running on a machine uses a *fetch-decode-execute* cycle, whereas a program running on an interpreter uses a *fetch-analyze-execute* cycle, but the output is the same given the same input. The source program is said to run on top of the interpreter. It is the interpreter that directs the analysis/decoding of the source program code into machine code, but it is the CPU that directs the fetching and execution of code. Since the interpreter itself is a program, it can only run on a machine if the interpreter is itself expressed in the machine's machine code.

The interpretation of a high-level language can be up to 100 times slower than running an equivalent program, that has been compiled into machine code, so it is not always feasible to use an interpreter.

4.0.3 Real and abstract machines

An interpreter is called an abstract machine whereas a computer is a real machine. An intuitive way of looking at the relationship between interpreters and machines is that an interpreter is a machine implemented by software, and a machine is an interpreter implemented in hardware. There is no functional difference between running a program on either of these types of machines. The only difference is the speed of execution. A hardware machine is naturally faster than a software "machine" at executing a program. Also an interpreter's fetch-analyze-execute cycle is very similar to a hardware machine's fetch-decode-execute cycle.

4.0.4 Pure interpretation

With pure interpretation a high-level language is interpreted one line of code at a time by the interpreter, so no translation is performed. There are more than one way to handle interpretation since it may involve at least some translation before the actual interpreter is run. This is explained in the part about hybrid implementation systems.

Pure interpretation, however, is the most extreme form of interpretation. No translation is performed at all. This also means that it is the slowest form of interpretation to execute since the high-level source program has not been prepared for faster interpretation. A positive thing though is that one does not need to run the source code through a compiler first before the interpreter can be used, so in some cases it is more practical to use a pure interpreter if execution speed is not a priority but the ability to execute the program immediately is.

An example of this could be students learning to program. Most of their first programs are so simple that compilation can seem like an irritating waste of time, since they just want to see if their program can run and what the output looks like. An example of a situation where pure interpretation would be too slow to use would be for enterprise applications where extremely fast execution speed is paramount.

4.0.5 Hybrid implementation systems

A hybrid implementation system is a compromise between a traditional compiler and a pure interpreter. Whereas a compiler takes a source program as input and creates a program in machine code for a certain hardware platform, a hybrid interpreter translates the source program into an intermediate language which is more suitable for interpretation and therefore executes faster when it is running on the interpreter.

The source language statements are decoded only once so this method of interpretation is much faster than pure interpretation. When the intermediate code is generated, lexical analysis is performed on the original source program, which creates lexical units, and then syntactic analysis is performed, which creates the parse trees.

4.1 Different kinds of interpreters

Interpreters can be built using different strategies. We will mention two strategies here: iterative and recursive interpretation. Whether one chooses to use one or the other depends on the complexity of the programming languages that the source programs use.

If a program is written in a high-level language, then it will almost certainly make use of some of the more advanced features of the language, such as functions that take functions as parameters. In that case the interpreter must be able to handle the instruction-tree that the program makes up. In other situations the source program may only be made up of primitive instructions, in which case a simpler interpreter can be used. These considerations will be touched upon in the following subsections.

4.1.1 Iterative interpretation

An iterative design of an interpreter is a relatively simple when compared to the design of an interpreter with recursive capabilities. The iterative nature means that the interpreter is not capable of interpreting complex programs written in high-level languages, since it can not traverse the source programs instruction-tree. Recursive capabilities are required for this. If the interpreter only needs to interpret source programs that use primitive instructions, then the simple nature of the iterative interpreter is preferred to cut down on the interpreters implementation details. The simpler nature of iterative interpreters also has performance benefits. This is because an interpreter is itself a program and the simpler the program, the faster it executes.

An iterative interpreter has its uses in the interpretation of machine code, command languages and simple programming languages. In the case of machine code interpreter, it is usually called an emulator because an interpreter running a machine's machine code can do everything the machine can do, but the interpreter naturally does it slower. There are a lot of companies that use an emulator to test hardware designs before sending the design to the fabrication factory.

If an interpreter is designed to interpret a command language, it will wait for a command to be entered at a command console and then execute that command exactly once. In such an environment it is expected that the computer system returns a response immediately, when a command has been entered. Command languages are in fact meant for interpretation by design.

The final example we give of the use of iterative interpretation is the case of simple programming languages. Here, a command from the language serves as an instruction. The trick is that the language must not contain composite commands, because of the lack of recursive capabilities in this type of interpreter. Programs running on such an interpreter must be a stream of primitive commands. Interpreting each command is a process of syntactic analysis and possibly also contextual analysis, and it is therefore more efficient to interpret machine instructions.

4.1.2 Recursive interpretation

As mentioned, it is possible to interpret programs written in high-level programming languages, but it is very slow and more difficult than interpreting programs written in low-level programming languages. Because the interactive interpretation scheme is not capable of interpreting the complex program structures found in higher-level programming languages, the recursive interpretation scheme must be used in such situations. The mentioned complex structures are, for example, composite commands, where a command has subcommands, which can also have subcommands and so on.

The recursive scheme first fetches and analyzes the entire program and only then begins executing the program's instructions when it knows the programs phrase-structure. The analysis and execution steps are recursive. The analysis step performs syntactic and contextual analysis and produces a decorated AST.

Because recursive interpreters must go through and analyze an entire program before they begin execution, they forego the advantage that interpreters are known to have, which is the ability to begin executing instructions im-

mediately. It is therefore normally preferred to use a hybrid implementation system, which compiles to a lower-level intermediate language and then runs that intermediate-level code on the interpreter.

4.2 Translators in general

In computer science a translator is a program that takes a source language as input for the translation process. From this source language a semantically equivalent target language is generated. An example of a such translators is e.g. a PHP-to-ASP translator.

Translators such as a PHP-to-ASP or Java-to-C++ translator are high-level translators (i.e. they translate between two high-level programming languages). At the other end of the scale we find the low-level translators, namely the *assembler* and the *compiler* that translate from assembly language to machine code and from a high-level language to a low-level language[1], respectively.

Code can, however, also be translated the other way. A translator that translates from machine code to assembler code is called a disassembler, and one that translates from a low-level language into a high-level language is called a decompiler.

Translators do not just translate a source program into an object program. Before that whole process takes place, the translator will parse the source program to make sure that it is well-formed. If it is not, then an error report will be generated for the user. However, if the source program is well-formed, then an object program will be created.

Translator and interpreters use much of the same theory, so there is no need to explain that theory again, but we will briefly just mention that translators also must check syntax and semantics. They handle code generation differently that interpreters, because they have to generate the whole object program in one go before it can be executed.

There are some design issues to consider when creating a translator. A translator can be implemented as a either a one-pass or a multi-pass translator. A one-pass translator is faster than a multi-pass translator, but it will usually use more memory since it must handle all phases of the translation all at once. If modularity in the structure of the translator code is a priority then the multi-pass design is the only choice. The syntactic analyzer part of a one-pass design must continuously call the contextual analyzer and code generator, so it will be filled wth code that has nothing to do with syntactic analysis and this prevents modularity. By using multiple passes, the code for those three phases can easily be separated into modules, making the translator's code more readable and maintainable. The multi-pass design also allows for greater flexibility during the whole translation phase. Because the code generator does not have to generate code during the first pass, it can traverse the AST as it wishes and create much more optimized code in the resulting object program. Multi-pass translators that go to the extreme concerning optimization perform so called semantics-preserving transformations. Such transformations require careful analysis of the source program before code generation, and if the designer choses to have such transformations, then it requires a multi-pass design. The design of a source language can also affect whether a one-pass or multi-pass translator is

[1] A low-level language is either an assembly language or machine code.

required. If variables are not required by the language to be declared before they may be used then multiple passes will be needed to translate correctly. All these design issues and considerations illustrate the choices a language designer faces and some of the choices do conflict.

This concludes the introduction to the chapter about language processors. The next chapter is about our language design.

Chapter 5

Language design issues

This chapter deals with the concept of designing a grammar for our language. The language is named GEO . When one creates a grammar for a language it is a good idea to identify and examine different design issues. These could e.g. be the choice of paradigm, how the syntax should look and what the design criteria are for a good language/our language [8]. These criteria will be discussed later in this chapter.

In this chapter we will examine and evaluate different design solutions for our language. We will compare different design alternatives and try to find a solution which we find to be suitable for our project. We will then summarize what we have found out. At the end of this chapter we will show how the GEO+ language is translated into a SVG file.

5.1 Design criteria

There are many different reasons why programmers prefer one language to another. A way of differentiate programing languages is to look at the criteria for the design of a language. We are aware of that there are many different criteria in a good language design. We have chosen only to examine the criteria we think have relevance in our project. The criterias are readability and writability.

Readability tells how easy the language is to read and to understand. A syntax that is particularly dense and cryptic often makes a program easy to write for the experienced programmer, but can be quite hard to learn and use for the unexperienced programmer. Also, if the program has to be modified later it can be difficult to figure out the purpose of specific parts of the program.

Writability is a measure of how easy a language can be used to express a computation correctly, concisely, and quickly. It should be easy to write a program quickly, when the programmer has some practice in the language.

Readability and writability must of course be considered in the context of the problem domain.

5.2 Language paradigms

We have chosen to examine the term paradigm, because we think that when one knows about different paradigms one can be inspired by the idea that is behind the paradigm.

There are four major basic paradigms that describe most programming languages that are in use today.

Before one can decide which paradigm one wants to use or be inspired by, it is important to know the basic rules of the paradigms. Therefore we will shortly describe each of the paradigms.

Imperative programming Imperative languages are command driven or statement-oriented languages. A program consists of a sequence of statements. The execution at each of the statements causes the computer to change the value of one or more locations in memory. In this paradigm the goal is to find an algorithm that solves a central problem piece by piece. The syntax of a program in such a language has the form:

Statement_1;

Statement_2;

...

An example of an imperative language is the programming language *C*.

Object-oriented programming In object-oriented programming complex data objects are built. A limited set of functions are designed to operate on these data objects. Objects are created instead of sequences of instructions.

Examples of object-oriented programming language are *C++* and *Java*.

Logic/declarative programming Logic programming is based on mathematical logic. Logic based languages are executed by checking for the presence of a certain enabling condition. If the condition is meet, an appropriate action is executed. It is called logic programming because the basic enabling conditions are certain classes of predicate logic. The syntax of a logic based language is similar to the following:

enabling condition_1 $\rightarrow \text{action}_1$

enabling condition_2 $\rightarrow \text{action}_2$

enabling condition_n $\rightarrow \text{action}_n$

Programming logic based languages often consist of building a table of possible conditions. If a condition occurs an appropriate action is taken. *YACC* (Yet Another Compiler Compiler) is an example of a parse program that uses a logic-based technique.

An example of a logic programming language is *Prolog*.

Functional/applicative programming Functional programming focuses on the task that the program represents rather than on the state changes upon

execution. Computations are made by calling functions, which evaluate to valid data just like numbers, characters and strings are.

The syntax of a applicative language is similar to:

$$function_n(...function_2(function_1(data))...)$$

Examples of a functional languages are *Lisp* and *ML*.

We know that it is not the case that a programmer only programs imperatively in an imperative language or object-oriented programming in an object-oriented language, but it comes naturally to use the respective programming paradigms in the languages that use those paradigms. We have chosen to let us be inspired by the imperative paradigm. Our reason for chosing this paradigm is that it results in accurate execution of computing steps and that the paradigm somewhat resembles manual routines from our everyday lives such as following a recipe while cooking.

5.3 GEO

GEO is a language that makes it possible for a user to create a map. It is used as the front-end enabling data to be stored. The language is inspired by some *COBOL* lookalike language and focuses on making it readable using short sentences that specify what is to be drawn on the map. The language is inspired by the imperative paradigm as mentioned above.

Every map created in GEO starts by using the keyword *map* followed by the height and width of the map. Following this initial statement is the map specifications enclosed between **(*** and ***)**.

The elements on the map are then created using the keyword *draw* followed by a type definition for the kind of feature currently being processed. In order to incorporate features to be used by the other features a unique name might be required. This follows the type and is closed in quotation marks, which is done in order to allow spaces to be used in the unique identifiers. Semicolons are delimiters between feature declarations.

The syntax for each of the possible features in GEO is inspired by the English language. In order to describe a road on the map one might use the phrase: "I would like a highway named E21 to start in the upper-left corner and end in the lower-right corner". This is simulated in GEO by using the following phrase:

draw highway "E21" from 0,0 to 800,600;

It is required to exchange the abstract concepts "upper-left" and "lower-right" with usable coordinates. One might argue that the phrases could easily be translated into the corresponding coordinates, however as the map becomes more complicated more precise coordinates are needed. All strings written in GEO must obey the rules given in Appendix C. In order to emulate different hight levels on our maps all coordinates are defined using three components: one for the horizontal value, one for the vertical value, and finally a component emulating height. This makes it possible to put a map feature on top of another.

This grammar has some semantics, which will be described in the following.

Roads The different roads are *highway*, *interstate*, *local*, *path*, *cycling* and *railroad*. They all share some common properties: all have a start point and an end point. Furthermore they can all curve, which is represented by any number of intermediate coordinates that the road must pass. The following sentence in GEO will draw a road of type "interstate" named "I21", starting at coordinate (50, 50) and ending at coordinate (100, 100), that must pass coordinate (80, 70)[1].

> *draw interstate "I21" from 50, 50, 0 to 100, 100, 0 using 80, 70, 0;*

The third component of the coordinate is optional, and has a default value of 0.

Intersections At first it was considered a possibility to draw the intersection between roads using two coordinates. The idea was abandoned as these coordinates would be dependent on the intersecting roads, and when moving a road crossing another road the intersection would have to be moved manually. Instead the intersection is dependent on the identifiers of the crossing roads. The following should clarify:

> *draw roundabout where "I21" intersects "E12";*

Buildings Following we will define features not related to traffic. The first non-traffic feature needed is the ability to place buildings on the map. In order to simplify the map all buildings are defined by two points, effectively defining a rectangular box. The buildings are either *residential*, *commercial* or *factory*. Placing a building on the map is done by issuing the following command:

> *draw commercial "Anderton Mall" from 100, 100, 0 to 200, 200, 3;*

Nature Just like we had to be able to put buildings on the map, we also need to display areas of nature on it. Just like above we simplify this part and simply define nature to be a circle. With that in mind the following is a valid GEO syntax.

> *draw nature "Central park" with center in 100, 250 and radius of 50;*

Attraction An attraction is defined by marking one of the features above. A valid syntax for defining the above as a tourist-attraction is:

> *mark "Central park" as tourist-attraction;*

In order to summarize the features and the syntax of GEO a small syntax example is presented in Table 5.1.

[1] That is a small section of a curved interstate road. (this will not be the case in the real world as all interstate are connected to several other roads, and this serves only to demonstrate the syntax for drawing roads in GEO).

```
map 800, 600 (*
  draw highway “E21” from 0,0 to 800,600 using 100,200 and 200,100;
  draw interstate “I12” from 800,0 to 600,0;
  draw cross where “E21”crosses “I12”;
  draw commercial “Georges gas” from 850, 800 to 800, 850;
  mark “Georges gas” as gas-station;
  mark “E21” with recommended speed limit of 80 km/h;
*)
```

Table 5.1: Small example of the syntax of GEO .

The language GEO was written in a COBOL look-alike language and is inspired by the English language. Because of the COBOL look-alike syntax it makes it easy to read and modify a program even for an experienced and an unexperienced programmer. The language GEO uses readability as the design criteria. When the user writes a map he must write the words *from* and *to* a lot of times, which in the long term could be annoying and time consuming for the user. We believe that the user of our language is a person who wants to write a map quickly. Therefore we decided to examined if we could improve the language GEO so that it would be easy and quick to write create map in. This is where GEO+ enters the project.

5.4 GEO+

GEO+ does not use English words like *from* and *to* to increase readability, and in contrast to GEO , GEO+ uses a *Prolog*-like syntax. These two things combined make the language more writable. Just like Prolog, GEO+ uses a sequence of facts that acts as a knowledge database. We are, however, still inspired by the imperative paradigm because we still use controlled execution steps in our language. The syntax for GEO+ can be found in Appendix C.

We will just use the example from before with the phrase: “I would like a highway named E21 to start in the upper-left corner and end in the lower-right corner”. This would in GEO+ be simulated by the following phrase:

road(highway,E21,(0,0)(800,600));

5.4.1 Dividing the GEO+ map into sub-maps

The language GEO does not, in its current design, support more than one map. This restriction is not satisfiable, since maps normally tend to be rather huge. Since GEO+ is the successor to GEO , this feature has been implemented. The idea is to use a fixed map size for all maps and use one map as a center map. All other maps will be placed according to this center map, which is illustrated in Figure 5.1. The sign $\times$ marks $(0, 0)$ on the global map and $(0, 0)$ on the center map. Each map will have its own local $(0, 0)$ coordinate. The center map will have the map ID $(0, 0)$. The map to the left of the center map will have the ID $(-1, 0)$. This relative coordinate is used to place the local map relative to the

center map. In the same manner, the map to the right of the center map will have the ID $(1, 0)$ and so forth.

Another reason for not using one huge map and instead using many small maps is to optimize the bandwidth usage as described in 5.4.2 and 7.2.

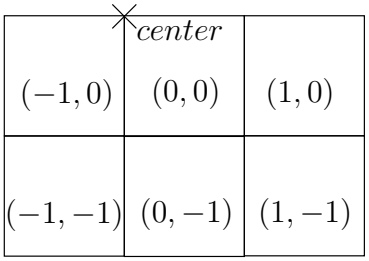


Figure 5.1: Dividing the GEO+ map into sub-maps

When creating a map, one must start by defining which map is to be created. Taking the previous example with the highway, the map should be created as follows (if it is the center map):

map (0,0){
road(highway,E21,(0,0)(800,600));
}

5.4.2 The semantics of GEO+ .

In order to start creating a map, the reserved word *map* must be used as this is the start tag of each map. The map tag must then be followed by the word *COORD*, which represent the position in the map grid. *COORD* should be within the range of -1024 and 1024. Following this position, a number of statements are listed. These statements are in the context free grammar defined as *FUNCTIONS*. These functions could for example be *road*s. The entire syntax is as follows:

map (COORD,COORD) { $FUNCTION^{+}$ }

Example 5.2 shows a map, which position is $(0, 0)$, and therefore is the first map. If you want your next map to be placed left of this one, the coordinate would be $(-1, 0)$, and to the right would be $(1, 0)$ and so forth.

When creating a map, one must take into account that 200 units in our map coordinate system is equal to 1 kilometer.

Functions Upon creating a map, *FUNCTION*s must be used, if one wants anything on the map. The *map* mode simple creates a map, whereas a *FUNCTION* draws roads, intersections, buildings and signs on the map.

Roads The things that need to be drawn the most is likely to be roads. A *road* contains a *ROADTYPE*, which must be either *local*, *highway*, *interstate*, *path*, *cycling* or *railroad*. It must also contain a *VARIABLE*, as described in 5.4.2. Then the last part is an *EXPRESSION*, which is described in 5.4.2.

```
map (0,0){
  road(highway,E21,(0,0)(100,200)(200,100)(800,600));
  road(highway,I12,(800,0)E21(2)(600,0));

  intersection(ramp,E21,I12,(200,100));
  intersection(bridge,E21,I12,E21(2));

  building(commercial,(750,800)(800,750));

  sign(gasstation,I12(2));
  sign(speed 80,(700,0));
}
```

Table 5.2: Small example of the syntax of GEO+ .

Each road is bound to a variable. These variables can be referred to as sub-variables. Coordinates on a road can be a sub-variable of another road, as in example 5.2. Here the second coordinate in the definition of road I21 is defined to be the same point as the second coordinate in E12.

road(ROADTYPE, VARIABLE, EXPRESSION$^+$);

Upon translating a map from GEO+ to miniSVG , each road will get a specific level of display (as described in 7.3). Highways, local roads, interstates and railroads will all be display on level 1, meaning all levels, whereas path and cycling first will be displayed on level 3.

Intersections *INTERSECTION* is a way to define the representation of two roads intersecting each other. An intersection can be a *cross*, a*roundabout*, a *t-cross*, a *ramp* or a *bridge*.

intersection(INTERTYPE, VARIABLE, VARIABLE, EXPRESSION);

There are two ways of doing this, as shown in Figure 5.2. Both are examples of I12 going onto E21, but are in fact two different intersections. The first one refers to a coordinate and the other one refers to a sub-variable.

The first variable in the line is the one which will be drawn as the topmost road. In the example, I12 will go under E21 as E21 is written first and therefore becomes the topmost road.

Buildings When a building needs to be drawn, it is done using the reserved word *building*. A *building* consists of a *BULDINGTYPE*, which can be *residential*, *commercial*, *factory* or *farm*. The different types will be drawn on the map with different colors.

building(BUILDINGTYPE, EXPRESSION$^+$);

As with the intersection, one can choose sub-variables instead of coordinates as the points that define the building.

When a building is translated to miniSVG , a *Z* tag will be added to the line. This means that the last coordinate will be joined by a line from the first coordinate. For instance, if you want to create a building consisting of four corners, you will only need the four corners, because the line from the fourth corner to the first will be drawn by the system. If the first and last corner are equal, a line will still be drawn, but will only consist of one point, and therefore it will not be visible, as it will be covered by two lines going from and to the point.

Markers For landmarks and road signs on the map the user simply uses *marker*. This should contain a text, which can be defined by the user. For example, he could write "Al's gasstation", and the marker on the map would say "Al's gasstation". It could also be speed limits. Each marker also contains a coordinate, which again could be a sub-variable.

This should contain a type, just like intersections and buildings do, which can be *gasstation*, *church*, *zoo*, *tourist-attraction* or *speed* $NUMBER^+$, used for speed limits, where $NUMBER^+$ is a list of numbers.

marker(MARKER, EXPRESSION);

Waypoint A *waypoint* is like a *marker*, but it differs in the way it is to be used. Waypoints are used as a sort of name tags for cities, and should be used on center of the city. It only contains one coordinate, and the name of the waypoint should be unique. This is also a logic idea, because each city only contains one center.

waypoint(MARKER,(COORDX,COORDY));

Nature When designing nature the user simply uses the *nature* tag. This should be followed by *lake*, *river*, *coastline* or *forest*. On Figure 5.2 is displayed how a coastline should be drawn. Notice, on the "correct" map the coordinate $(0, 0)$ has been added, and thereby "closing" the path whereas the "wrong" map does not contain this coordinate.

Without the "closing" coordinate the *coastline* for instance would not look as the user exspected.

nature(NATURETYPE, (COORDX,COORDY)$^+$);

As with buildings, a Z will be added to the translated miniSVG , and hereby creating a line from the last coordinate to the first.

Variables Each variable is an *IDENTIFIER*, which simply is a unique name. This could for example be a road named *E21*. If another map named *E21* is created, this will result in an error when trying to interpret the GEO+ -file.

Some characters are illegal to use. These are "(", ")", ";" and ",". If the characters are used, one will get a translation error from the translater. It is possible to used "[", "]", "{" and "}" instead.

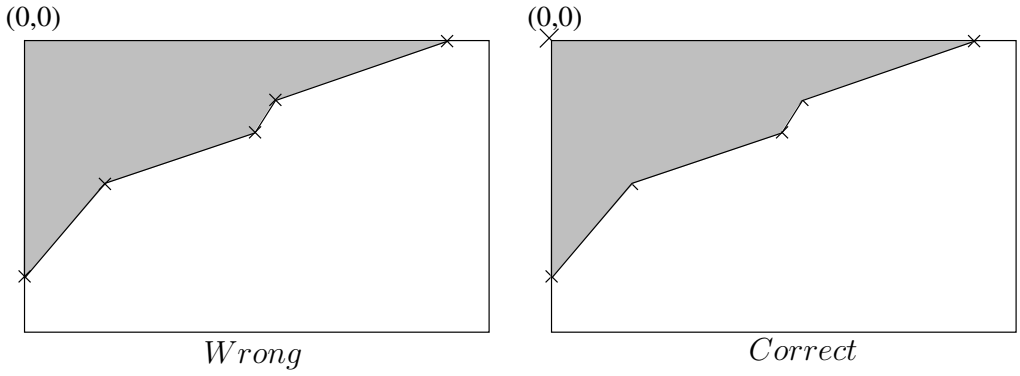


Figure 5.2: Wrong and correct way to create a coastline.

Coordinates Coordinates, *COORDX* and *COORDY*, are used to describe positions on the map. This could be positions of roads, intersections and so forth. Everything placed on the map needs coordinates. The range for *COORDX* is 0 to 800 and the range for *COORDY* is 0 to 600. A range larger than this will result in an error upon interpretation.

COORDX and *COORDY* are used to define the size of each sub-map. When taking into account that 200 units in our coordinate system, as mentioned, is equal to 1 kilometer, each map will have a size of 4 kilometers in width and 3 kilometers in height. This rather small size is chosen to optimize the bandwidth utilization, as coordinates can grow quite big.

Sub-variables Sub-variables are defined by a *VARIABLE* followed by a number of *NUMBER*s. This could for example be a variable of type *road*, named *E21(2)*. The tag *E21* is the unique road name, and *(2)* refers to the second coordinate of the road. Say, for example, that the interstate *I12* and highway *E21* are defined by the coordinates in Table 5.3.

I12	*E21*
I12(1)$= (0, 300)$	*E21(1)*$= (100, 100)$
I12(2)$= (200, 200)$	*E21(2)*=*I12(2)*
I12(3)$= (200, 0)$	*E21(3)*$= (300, 300)$

Table 5.3: Example of sub-variables.

In this example in Table 5.3, *E21* crosses *I12* in $(200, 200)$. Therefor *E21(2)* is pointing to *I12(2)* using a sub-variable. If, by mistake, the user is making a sub-variable to *E21(4)*, then when trying to interpret the GEO+ -file, he will get an error, because this coordinate is not a part of the road.

VARIABLE(NUMBER$^+$*)*

Expressions An *EXPRESSION* is either a coordinate, *COORD*, or a subvariable,*SUBVARIABLE*.

The language GEO+ was written in a Prolog look-alike language. Because of the Prolog-like syntax it makes it easy to write a map quickly, which was our goal in the GEO+ conclusion. The language GEO uses the design criteria: writability. When the user now writes a map he does not use all those English words used in GEO anymore, which is less timeconsuming. As mentioned earlier in this chapter our language is still inspired by the imperative paradigm even though we use a Prolog-like syntax.

5.4.3 Creating SVG from GEO+

In this section we will describe how a map file, written in the language GEO+ , is tranlated into an SVG file. We will also show how we design the program that makes this translation. We need to design and implement a translator that translates a map from our language GEO+ to an object program. This object program is expressed in the SVG format.

Because GEO+ is a very simple language, our design and implementation of the translator will be very simple. We will therefore only use translator theory that deals with the front-end. The front-end is the syntax analysis where the translator is scanning, parsing and if necessary constructing an abstract syntax tree. We have been inspired by the front-end, but we have chosen to omit the scanning and parsing elements. The reason is that our translator is very simple and it would be overkill to use scanning and parsing in our translator.

Our translator is reminiscent of a command line interpreter. This is because our translator reads one line at a time and processes it. Every time the translator reads a line it analyses it and checks the syntax of the line. The major difference between a command-line interpreter and our translator is that our translator does not execute the object program. It instead translates the source program to an object program.

Before the design of the translator is begun, it would be a good idea to know what steps the design process of the translator will have. This is illustrated in Figure 5.3.

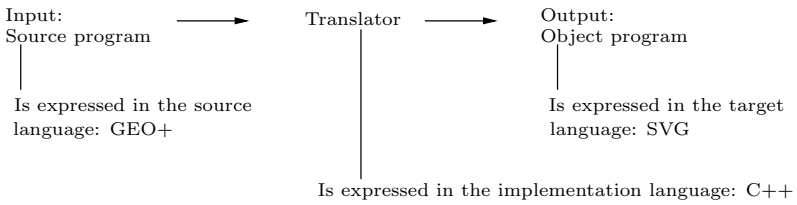


Figure 5.3: Illustration of the phases in the design

The figures shows that the translator takes an input source program. As mentioned earlier this input is a map file. This file is translated into an SVG file. The source code for this translation is expressed in the programming language C++. This figure, however, does not give us enough details to design our program. We therefore need to expand the figure with more details. This is illustrated in Figure 5.4.

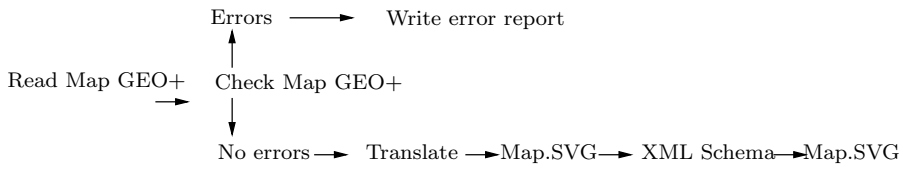


Figure 5.4: Illustration of the phases in the design

The figure shows that there are four major steps in the design of the program:

- Read the map.geo+ file
- Check syntax of the map.geo+ file
- Translate the map.geo+ file
- Check the map.geo+ file using XML Schema

The first step in our program is to read the map file. An example of a simple map is illustrated in Figure 5.4. The program will read the source program one line at a time. When one line has been read all empty spaces will be removed and the line will be saved.

The program will now start to check the syntax and semantics of each of the lines. If there is an error the program writes an error report and halts. The program will use the grammar for the language GEO+ to check the syntax and semantics of the source program. If there are no errors the source program will be processed to completion.

Every time the translator sees a token the it will translate the token into a valid SVG tag. After a line has been translated it will be saved. The translated SVG file is shown in Figure 5.5. After the translation, XML Schema will check if there are any syntactic or semantic errors in the source program. XML Schema will also use the defined syntax and semantics in GEO+ to check if there are any errors. Before XML Schema is used, it is necessary to define how XML Schema should perform a check. This is defined in an XML Schema file. This is shown in Appendix G. After this the SVG file can be read by the software on the mobile device. We make two checks in our translator, since we have chosen to use the parser XML Schema to check that we have generated a valid SVG file.

5.5 Discussion

In this chapter we have tried to create a grammar which is suitable for our project. During the construction of our syntax a trade-off has been made between two important design criteria: *Readability* and *Writability*. As mentioned, when we first created a grammar, GEO , we chose to design a language that was very easy to read, and the grammar has a syntax that looks somewhat like COBOL's syntax. The language enables us to read programs written in it much like we read basic English sentences. It is therefore easy for a new programmer to understand and read a program. It does not require much practice to learn the language GEO .

```
map (10,-1){
  road(highway,E21,(0,0)(100,200)(200,100)(800,600));
  road(interstate,I12,(800,0)(100,200)(600,0));
  road(path,Aalborg-stien,(500,0)(400,5)(100,453));

  intersection(ramp,E21,I12,(100,200));

  building(commercial,(400,400)(400,500)(500,500)(500,400));
  building(residential,(0,0)(50,0)(50,75)(0,75));
  building(factory,(100,600)(200,600)(200,500)(100,600));
  building(farm,(200,200)(250,200)(250,250)(200,250));

  marker(gasstation,(600,0));
  marker(speed 80,(700,51));

  nature(coastline,(0,400)(70,376)(200,165)(450,11)(602,0)(0,0));

  waypoint(Aalborg,(200,351));
}
```

Table 5.4: Example of a map in GEO+ , which position is (10,−1).

Then, later on, we evaluated our language and came to the conclusion that we should redefine and expand our grammar. We found out that it was necessary to add new features to the existing language GEO . We therefore added sub-maps, variables, subvariables, naturetypes, expressions, signs and coordx/coordy. It is worth mentioning sub-maps because sub-maps is a very different way of handling maps in our language GEO+ compared to GEO .

It is, as mentioned earlier, now possible in GEO+ to divide big maps into smaller sub-maps. We believe that this is a good idea because a map tends to be rather huge. But with sub-maps a huge map will be divided into a number of smaller maps. We also decided that it was more important that the new language would be writable rather than readable. The reason for this is that the user of the language should be able to write maps efficiently and quickly. The source code should not be time consuming to write. We therefore extended and changed the existing language GEO e.g. we removed the English words that made GEO very readable. We know that when a user has to learn GEO+ he has to learn to write the language, learn a new vocabulary, a new syntax and new semantics (new words, sentence structure and meaning), but because our language is very small and very simple we do not consider this to be a problem. The language now has a syntax much like Prolog's.

During the design of our language we have chosen to be inspired by the imperative paradigm. The reason for it is that we would like to have a statement-oriented language. Such a language reminds one of manual routines from our everyday life, which in addition can help the user to understand the syntax and semantics of our language.

```
<?xml version="1.0"?>
<svgObjects xmlns="http://www.sentinel.dk"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.sentinel.dk
  miniSVG_XML-Schema.xsd">

<svg id="(10,-1)" width="800" height="600" x="0" y="0">

<path id="road_highway_E21_0" d="M0 0 L100 200 L200 100 L800 600"/>
<path id="road_interstate_I12_0" d="M800 0 L100 200 L600 0"/>
<path id="road_path_Aalborg-stien_2" d="M500 0 L400 5 L100 453"/>

<path id="intersection_ramp_E21-I12_0" d="M100 200"/>

<path id="building_commercial__1" d="M400 400 L400 500 L500 500 L500 400Z"/>
<path id="building_residential__1" d="M0 0 L50 0 L50 75 L0 75Z"/>
<path id="building_factory__1" d="M100 600 L200 600 L200 500 L100 600Z"/>
<path id="building_farm__1" d="M200 200 L250 200 L250 250 L200 250Z"/>

<path id="marker_gasstation__2" d="M600 0"/>
<path id="marker_speed80__2" d="M700 51"/>

<path id="nature_coastline__1" d="M0 400 L70 376 L200 165 L450 11 L602 0 L0 0Z"/>

<path id="waypoint_Aalborg__1" d="M200 351"/>
</svg>
</svgObjects>
```

Table 5.5: Example of an SVG file that has been translated from a GEO+ map

Chapter 6

Server

To store the map on the server, there are initially two choices to chose from. The first is to store a big SVG file on the server and extract the data using XML queries. The second option is to structure the SVG file into pieces and put them into a database. The latter option was chosen as this would allow us to query a database structured using spatial division (for more on this see Section 2.2), and we can then retrieve only minor pieces of the map, if we wish. This chapter gives an overview of the role the server plays in this project.

On the server-side we have decided on using the PostgreSQL database for storing all data related to SVG maps. Also, enabling an extension to PostgreSQL called PostGIS allows for storing geometric objects in the database and making spatial search requests on these data. Please refer to section 6.1 and 6.2 for further information about PostgreSQL and PostGIS.

Alongside our database backend runs the Apache[1] web server which serves the purpose of receiving HTTP requests from clients and then replying to them. Also, a PHP module is linked (using mod_php[2]) to Apache allowing for server-side interpretation of PHP-scripts written by us. Refer to Appendix E to see how this was accomplished.

Enabling the server to create dynamic content to be returned to the client can be achieved using several different technologies and techniques. Common Gateway Interface *(CGI)* is an interface that allows programmers to create pages for the web with dynamic contents using their favorite language. Some of the most often used languages for this purpose include *Perl*, *Python* and *C*.

PHP: Hypertext Preprocessor *(PHP)*[3] is a scripting language made by Rasmus Lerdorf designed for the same purpose as CGI. PHP was our language of choice on the server-side (manipulating data in a database) for the simple reason that we already had experience with this language. Had we decided not to use PHP, we could easily have accomplished the same thing using Perl, Python, or any or suitable language.

Yet another technology is JSP. According to Sun, "JavaServer Pages (JSP) technology provides a simplified, fast way to create dynamic web content. JSP technology enables rapid development of web-based applications that are server- and platform-independent."[11]

[1] http://www.apache.org
[2] http://www.php.net/install.apache
[3] http://www.php.net

The server responds to a request from a client by examining the information passed along the HTTP. The information passed consists of the coordinates of the top-left and bottom-right corners of a rectangular geometric shape. A number specifying the level of detail, that one wants, is also passed.

Using this information a query to the database is issued. The PHP then makes a spatial search through the database. Thus, it is checked if each shape in the database is positioned within the rectangular shape specified by the client. If a shape intersects with the rectangular box, the shape is modified to specify exactly the parts of the shape that are parts of the rectangular box.

The shapes in the rectangular box are made into an SVG file using PHP. This file is transmitted to the client for further processing (covered in the next chapter).

6.1 PostgreSQL

PostgreSQL is an object-relational database management system (ORDBMS) that supports the SQL92 and SQL99 standards. PostgreSQL was chosen as database for our project, because we needed at database that would give us support for geographic objects. Actually, PostgreSQL itself doesn't provide this feature, but an extension called PostGIS.

MySQL has some support for geographic objects as well, but compared to PostgreSQL it still needs some maturing. Thus, opting for PostgreSQL seemed like the right choice for us (not considering any of the commercial databases such as Oracle), and since PostgreSQL is released under the BSD-license (and PostGIS is GPL), it gives us the power to freely download, compile[4], and install it on our own server for development and testing purposes throughout the whole project.

6.2 PostGIS

PostGIS is an extension to PostgreSQL which allows GIS (Geographic Information Systems) objects to be stored in the database. PostGIS includes support for GiST-based R-Tree spatial indexes. Also, functions for basic analysis of GIS objects are included.

PostGIS implements the "Simple Features"[5] of the OpenGIS Abstract Specification as defined by the OpenGIS Consortium, Inc. (OGC)[6]. This allows for storing geometric objects of the following types: Point, Line, Polygon, MultiPoint, MultiLine, MultiPolygon, and GeometryCollection.

GeometryCollection

A GeometryCollection is a geometry which is a collection of 1 or more geometries.

[4]Please see Appendix E for details.
[5]http://www.opengis.org/docs/99-049.pdf
[6]http://www.opengis.org

Point and MultiPoint

A Point is a 0-dimensional geometric object and represents a single location with a x- and y-coordinate, while a MultiPoint is a 0-dimensional geometric object and represents a collection of locations with each their own x- and y-coordinate.

Line and MultiLine

A Line is a 1-dimensional geometric object has exactly two points, while a MultiLine is a 1-dimensional geometric object that has more than two points making it a geometric collection of Lines all connected to form a MultiLine. Each line segment in a MultiLine runs between a consecutive pair of points.

Polygon and MultiPolygon

A Polygon is a 2-dimensional geometric object. It is classified as a simple surface type with associating boundaries as the set of closed curves corresponding to its 'exterior' and 'interior' boundaries.

The simple planar surface of a Polygon consists of a 'patch' which is associated with one exterior boundary and zero or more interior boundaries. Each interior boundary defines a hole in the Polygon.

A MultiPolygon is a geometric collection of Polygons.

Dealing with spatial data in PostgreSQL

A spatial referencing system identifier (SRID) is required when creating spatial objects for insertion into the database. Figure 6.1 shows an example of a valid insert statement to create and insert a spatial object.

```
INSERT INTO TABLE ( GEOMETRY, NAME ) VALUES (
GeometryFromText('POINT(265.21 745.32)', 312), '742 Evergreen Terrace' );
```

Figure 6.1: Example of insert statement.

For this database we have chosen spatial division by indexing (see 2.2.2). We estimated that the time needed to implement spatial division by partitioning (see 2.2.1) was too great. We limited us to using brute force, because of its simple approach. If the focus was different the method chosen would have been a combination of spatial division by partitioning and spatial division by indexing, because the insertion, searching, updating and deletion operations on the database are less time-consuming on this structure.

But after investigating the structures, only the point structure was used. This was because of the richness of the information of the SVG file. And the database needed to abstract from all the "unneeded" information and only look at points. So each line of SVG was saved in the database separately and coordinates of connection points of each line was added into an index table.

The database has tree tables: DATA, MAP and NAMELIST. These are shown in Figure 6.2. The MAP table is the index where the spatial queries are performed. Each ID in the MAP database is related to the DATA table. A line in the SVG file or an object on the map, such as a road, are represented as

DATA	MAP	NAMELIST
id : varchar(500)	Point : Point	name : varchar(500)
object : varchar(500)	id : varchar(500) refered DATA	x : numeric
		y : numeric

Table 6.1: Tables in database

rows in the DATA table and a unique ID is placed on each line. An extra table (NAMELIST) had to be implemented to accomidate the waypoints 5.4.2. When the client starts up the program will request a list of name form this table. The X and Y coordinates from NAMELIST will be used as center coordiantes of the map.

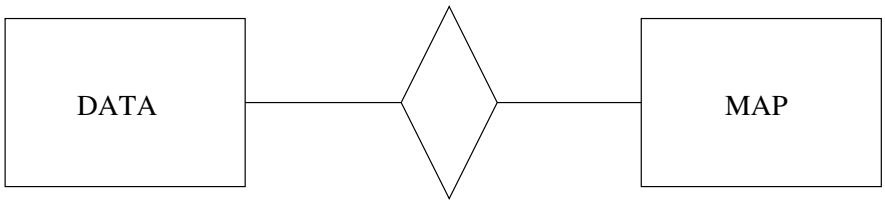


Figure 6.2: E/R-Diagram of database

Each table has only columns as shown in Table 6.1.

The underlined rows are primary keys of the table. The object entry will contain each line of SVG, which will have a unique ID connected to it.

6.3 Storing SVG in the database

When an SVG map has been created it will be stored in the database. A UNIX shell script carefully splits all data elements apart. This includes all the geometric objects and other information about the map such as map size, name of map etc. Each data element is then inserted into the database using SQL insert statements.

Please refer to Appendix F for the complete UNIX shell script.

6.4 Server queries through PHP

PHP is used as a tool for requesting data from the database whenever the server recieves requests from clients. These requests contains specifik map informations. PHP's task is then to run thru every entry in the database, and get the map from the database, and remove unnecessary coordinates from the map. These unnecessary coordinates are all geometric objects, which is on the current map but outside the view of the client.

The most of the PHP's task is relative easy and uninteresting. The function *check_coord* on the other hand is interesting. It takes two coordinates, (x_1,y_1) and (x_2,y_2), as argument. It then creates a virtual box using (x_1,y_1),(x_2,y_1),(x_2,y_2)

and (x_1,y_2). It then selects all id's related to the coordinates from the database, which are inside the box. All coordinates of the object id are then taken out and sorted. So only the coordinates in the box, and connecting points, are left. The result is then a valid SVG map, which the client then recieves.

Chapter 7

Client

This chapter describes the solution for the client program. First we descripe the platform on which the client is developed and run. This is followed by a section which describe the prices for mobile network traffic, where we consider the clients need for all map details, or consider if the client should go through some beutification process. Section 7.5 then describes when network communication is needed. This is followed by a description of the merging and beautifying process. At the end af the chapter we will give a describe of the design of the client.

7.1 J2ME

When we started to collect informations about software that could be implemented on a mobile devices we found three main programming environment. A detailed description and comparision of these enviroment can be found in Appendix D. We have chosen to use J2ME as the framework for developing the software on the mobile device.

J2ME is part of Sun's Java 2 family of platforms. J2ME is a set of core Java API's (Application Programming Interface) and execution environment particulary tailored for resourceconstrainted devices such as mobile phones, two way pagers and PDA's.

J2ME has a structure which is divided into layers. This structure is shown in Figure 7.1.

On top of the Operating system there is a virtual maschine and a configuration. The configuration is composed of a virtuel maschine and a minimal set a class libraries, classes and API's. J2ME is a subset of the Java Standard Edition and therefore it doesn't contain all the packages and classes that are in the Java Standard Edition library. There is e.g. no support for StringTokenizer in J2ME. A configuration specify the central libraries and the properties for the virtual machine on the device.

On top of this is a profile. The profile depends on which device one is developing to. The core in the J2ME platform consists of two different configurations. CDC (Connected Device Configuration) and CLDC (Connected Limited Device Configuration).

CDC: is designed for devices that has more memory, faster processors and more

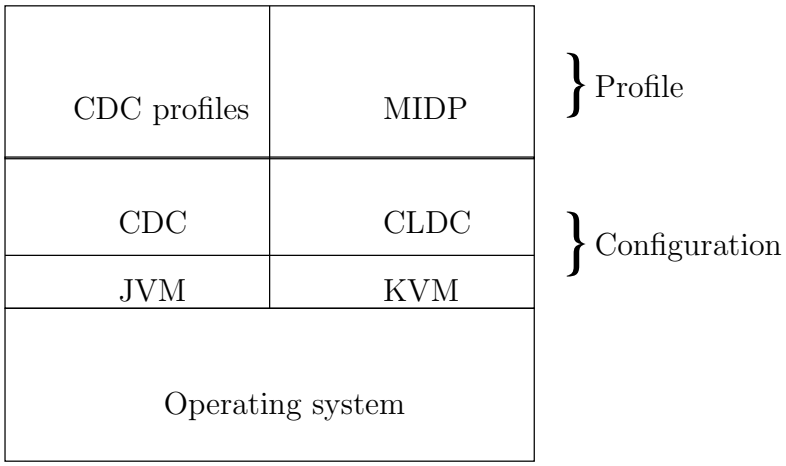


Figure 7.1: J2ME structure

network bandtwidth such as TV boxes, residence gateways and high-end PDA's. CDC include a full virtual Java machine.

CLDC: is the smallest of the two configurations. CLDC is designed for devices that has periodic network connection, slow processor, and limited memory. These devices e.g. are mobile phones, PDA's and two way pagers. These devices have typically 16 or 32 bit CPU and minimum 128 kb or 512 memory available for the Java J2ME platform. CLDC does not include a full virtual maschine. CDC uses a virtual maschine, named Kilo Virtual Maschine *(KVM)*.

As mentioned on top of configurations one will find the profiles, which defines the functionality for a specific category of devices. MIDP is a profile for CLDC based devices such as mobile phones. MIDP defines among other things the utilization of the user interface, consistent storage and network functions. MIDP is shorten for Mobile Information Device Profile. With a profile it is possible for a manufacturer to create his own profile which is designed for a specific device.

J2ME uses midlets. Midlets are small programs that are designed to run on a mobile device. The word midlets comes from the word MIDP and can be opfattes analog with the word "applet" Like applets must extend form the class Applet, midlets are also forced to extend from a superclasse. Midlets must extend from the class MIDlet.

The Sun Microsystems Wireless Toolkit, known as WTK creates J2ME midlets. The WTK contains the libraries required for creating midlets. WTK is a tool where it is possible to compile and run its midlets. One of the ideas of J2ME is to give platform independence. Programs written to the MIDP profile on the CLDC configuration can also be executed unmodified on many different mobile devices.

7.2 Bandwidth usage and time

An example on limiting the bandwidth is a road that consists of three parallel lines. Either three lines are transmitted (worse) or the single middle line is transmitted and the client decides how to display the map (better). An additional initiative for minimizing the bandwidth is that only the data required for displaying a more detailed zoom-level will be transmitted. Thus, the mobile device needs to be aware of the current level of detail and be aware of how to merge several maps with different levels of detail into one.

When working with mobile devices money is always an issue. According to the Danish tele company, Sonofon [10], 1 MB costs 18.75 øre. This means that if the client receives a map, which has a size of 100 KBytes, then the client will pay 1.875 kr.

$$100 \times 1.875 \text{ øre/KByte} = 187.5 \text{ øre/KByte} = 1.875 \text{ kr/KByte}$$

Therefore, if we transmit the entire map of 100 KBytes, the client will have to pay 1.875 kr for the map. If the client then does not view the entire map or zoom (see zooming in Section 7.3) he will end up paying for information he does not use. Therefore the solution is not optimal, and it becomes worse if he views many different maps.

Therefore another aspect comes into consideration: time. Since the entire map is not transfered all at once, more connections to the server will have. A mobile device can transmit between 9 Kbit/s and 54 Kbit/s [16]. The transfer time of a 100 KByte file will then be:

At max:
100 KBytes / 9 Kbits/s = 100 KBytes / (9/8) KBytes/s = 88.89 s

At min:
100 KBytes / 54 Kbits/s = 100 KBytes / (54/8) KBytes/s = 14.81 s

A typical size of a map is less than xxx KBytes and since a HTTP request is less than 1 KByte, transferring the map in pieces will be faster and cheaper than transferring the entire map all at once.

7.3 Structure

The map on the client is split into four levels. A level is a two-dimensional representation of features on a map like roads, buildings etc. Three of these levels are closely connected while the fourth is loosely connected with the others.

The first three levels are actually different levels of zoom on the map. The level with the lowest zoom level is made up of highways, interstates, railroads and local roads. The middle zoom level is made up of buildings and nature and the level with the most zoom is made up of cycling paths, paths and signs. The zoom levels are denoted as “zoom level 1” (least zoom) through “zoom level 3” (most zoom).

The final level is actually the above three levels merged into a single map. The features on this map is made up of all the features from zoom level 1, combined with the features from zoom level 2 and zoom level 3 that overlap.

7.4 Considerations

When the mobile device at first establishes contact with the server an SVG file, defining the entire map, is retrieved. This is stored for further reference. By applying a static set of rules to the SVG file a beautified SVG file is created, and this file is shown to the user of the mobile device.

Upon doing the actual zooming, we as system designers have two considerations to make. Should we send all map information at once or should we only send one zoom level at a time ?

Send the entire map at once If we send the entire map at once, we must take some considerations into account. One is that by doing this we may use more bandwidth than necessary. For instance we could send map information which is not needed by the client – this could be houses outside the view upon zooming (which only will be displayed when zooming). As mentioned in Chapter 1, the client pays for each packet of data sent, and therefore he might have to pay for something he will not see. But on the other hand, on maps with few details on level 2 and 3 this might be preferable, because all necessary details will then be sent at once.

Send one level at a time We could also choose to send one level at a time, as described in Section 7.4. This way the client will only receive data which are visible. The cost of this is that the client will have to use more time waiting for the request to the search and the server replying with the new details upon zooming.

By using levels of detail, we must now take care of another issue. This is to merge levels. When the user zooms, he will receive a new map, which needs to be merged with the existing map to create a new map. This task should be quite easy, and considering the growth within the size of computability provided by modern mobile devices, this should not take much time.

We have chosen to send one level at a time, because our system is mainly for use in cities. Therefore we could very well be sending information about houses and so which will never be displayed.

7.5 Communication

When a user requests specification on the map, the user informs the mobile device where to zoom. Using HTTP the mobile device requests specifications from the server. Using the information passed along the HTTP by the mobile device, the server is able to make an SVG file containing only the features from the next zoom that is within the coordinates. This is transfered to the mobile device, which then merges the new map with the old map stored on the client. Using the rules stored on the mobile device a beautified version of the map can be shown to the user of the mobile device.

If the user ever requests to zoom out, the client simply uses the latest map file and the map can be drawn without reconnecting to the server.

It can be useful to think of this as a stack that is pushed (when zooming in) and popped (when zooming out). This is illustrated in Figure 7.2.

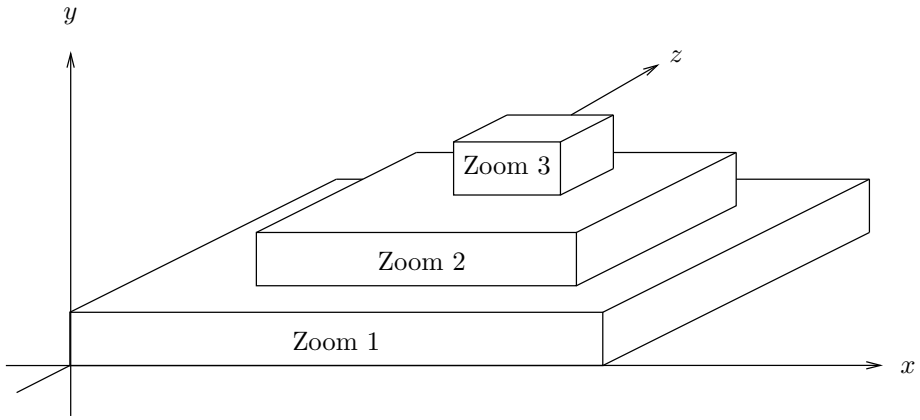


Figure 7.2: 3D illustration of layers on the map.

In Figure 7.2 one can see, that the size of the map decreases as the level of zoom increases.

Besides requesting more details on the map the user can also pan. When panning the user moves the view frustum and if the view frustum collides with the border values the client request a new map from the server. The border values is defined by the numbers of pan step multiplied with a pan step size. To handle the zoom and pan at the same time it is neccesary to decrease and increase the pan step and redefine the borers each time that a zoom has been made. Hereby the user can pan in different zoom levels and each time not request more data than needed. In Figure 7.3 the edges of the zoom levels are the borders and the distance from the view frustum to the edge is the numbers of pan step multiplied by the size of each pan.

7.6 Merging

When the user requests a zoom on the map a point is chosen. This point is used as the center of a virtual rectangular box. The box is then used to obtain new information from the server and when this information is received by the client it is stored as an object. As illustrated in Figure 7.2 the new informations needs to be scaled in size to match the size of the display. Furthermore, merging between two zoom levels also needs to be handled. To do this the application is split into two threads in order to avoid any idle state of the mobile device. The threads are described below:

1. To handle new information the coordinates defining the virtual box (the top-left and lower-right corners) are used to extract data from the server. When the data is retrieved it is stored and scaled to the size of the display.

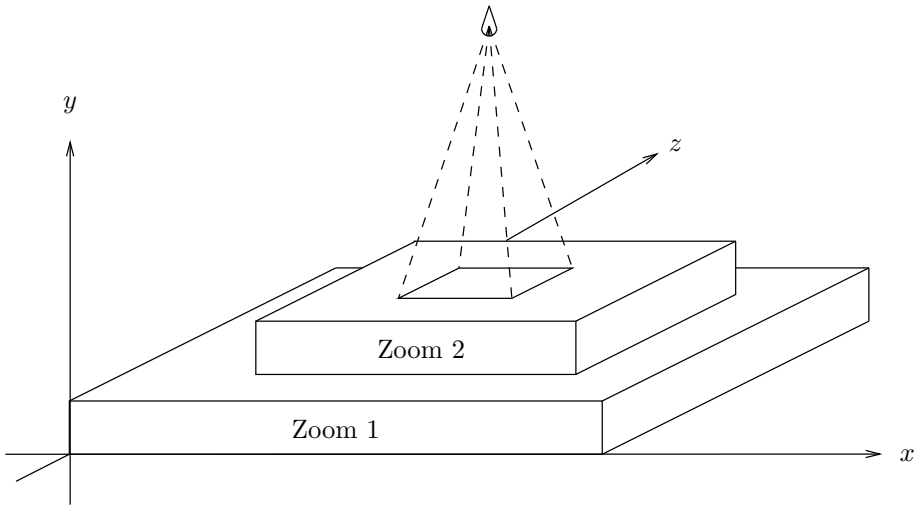


Figure 7.3: 3D illustration of the view on the map.

2. To handle the process of merging we first need to retrieve any elements located within the box at the old level of zoom. Each element on the map is checked for any intersection with the box. In case of an intersection the element is added to a temporary map. When all elements have been checked and possibly added to the temporary map, it is then scaled to fit the size of the display.

On completion of both threads the two maps need to be merged. This is done by combining all attributes from the two maps in a single new map. The complexity of the merger is quit simple compared to the beautifying process described in the next section. The reason for this is that it is irrelevant in which order the data is represented. An example of a merging is illustrated 7.4 where two simple maps written in beautySVG is merged into one map.

7.7 Beauty

In order to display the map in a more eye-pleasing manner we need to process the rough map into a more detailed version. This is done using a set of rules define on the client. To handle styling on the mobile device, this is done by applying the techniques presented in Section 4 on the mobile device.

The two figures in Figure 7.5 display the idea about having a simple rough map with few details on it (left) and the same map with more details than needs to be displayed on the client (right). Notice that the road is from zoom level 1 and the building (with the sign) comes from zoom level 2. Also notice that the road on the right will look different on the mobile device - the figure is only used to visualize how the beautifier works.

The compleksity of the beautifying is quit time consuming. The reason why it is nessesary to scan the entire data and when a token is found it is translated with relevant data. An example of beautifying is illustrated below where the

<HEADER....>
<SVG width=... height=...>
<path fill="none" stroke="red"
d="M 105 108 L ..... C ... />
<SVG>

<HEADER....>
<SVG width=... height=...>
<path fill="none" stroke="black"
d="M 82 76 L ..... C ... />
<SVG>

Merging

<HEADER....>
<SVG width=... height=...>
<path fill="none" stroke="black"
d="M 82 76 L ..... C ... />
<path fill="none" stroke="red"
d="M 105 108 L ..... C ... />
<SVG>

Figure 7.4: Merging two maps

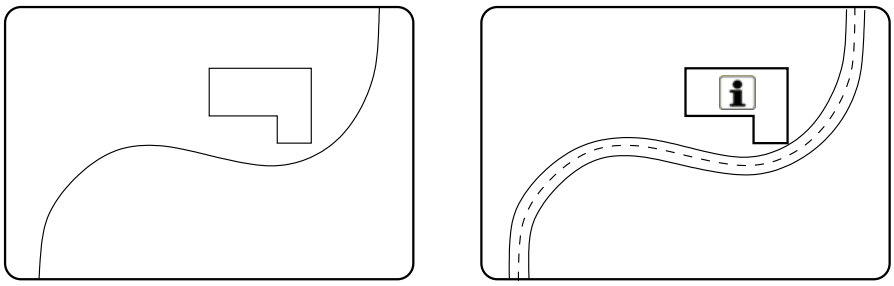


Figure 7.5: Example of the rough map (left) and the same map when beautified (right).

map written in miniSVG (the upper map) is translated into beautySVG (the lower map).

```
<svg id="Map" width="300mm" hight="300mm">
<path id="building_commercial__1" d="M400 400 L400 500" />
<path id="building_farm__1" d="M200 200 L250 200 L250 250 L200
    250" />
<path id="nature_coastline__1" d="M200 165 L450 11" />
</svg>

<svg id="Map" width="300mm" hight="300mm">
<path fill="blue" d="M400 400 L400 500" />
<path fill="brown" d="M200 200 L250 200 L250 250 L200 250" />
<path fill="darkblue" d="M200 165 L450 11" />
</svg>
```

7.8 Viewer

In order to display an SVGT file on a mobile device, a viewer capable of using the KVM on the mobile device is utilized. This project uses "TinyLine for J2ME MIDP 2.0" [15] for this purpose. However, as TinyLine is only capable of displaying SVGT it needs to be expanded with functionality for communication, merging and beautifying. Furthermore zooming and panning has to be elaborated so that they can handle request for new maps.

7.9 Design

This section describes the design of the svg-viewer which includes panning, zooming, translation and merging. The design is strongly influenced by TinyLine for J2ME MIDP 2.0 [15]. To handle the desired functionality we use several classes and these are illustrated in Figure 7.6 and described below.

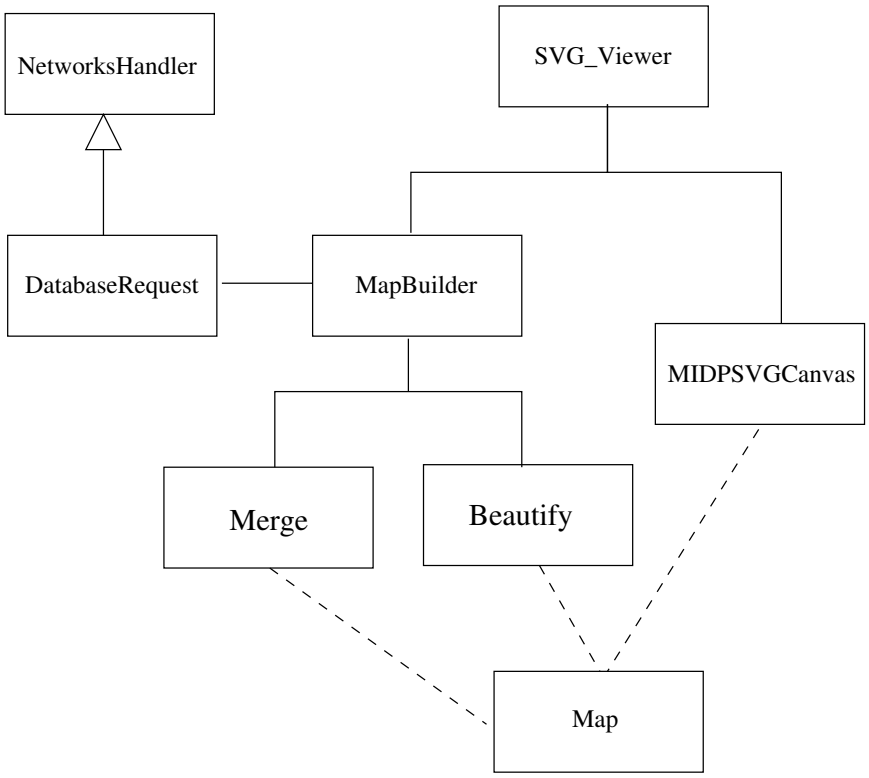


Figure 7.6: Client side class diagram

SVG_Viewer: This class is responsible for maintaining the Graphical User Interface (GUI) and handle all user inputs. When the application starts this class initiates Mapbuilder and MIDPSVGCanvas and call the Mapbuilder class which prepares a list of available maps. When the user clicks open map the name of the selected map is given to Mapbuilder which prepares

the map. Hereafter it is given to the MIDPSVGCanvas who draws the map. This process is described in figure **??**. Furthermore this class grabs user input such as zooming and panning and asks the MIDPSVGCanvas to handle it properly.

MIDPSVGCanvas: This class uses the two tinyline classes MIDPSVGRender and ImageConsumer. MIDPSVGCanvas handles the commands given from the SVG_Viewer which are zooming and panning. Furthermore it is responsible for creating MIDPSVGRender and ImageConsumer which is used for drawing.

MapBuilder: This class is responsible for storing all maps. The maps are stored in a Vector.

Map: This class encapsulate the different zoom level in an object making it possible to zoom out without requesting already requested map data. The method getBeauty() returns an input stream which the MIDPSVGCanvas uses to draw the map.

Merge: To separate the merging from the rest of the functionality this class handles all merging.

Beautify: To separate the beautifying from the rest of the functionality this class handles all merging.

NetworksHandler: The NetworksHandler is responsible for opening and closing connection for all network requests.

DatbaseRequest: Is a sub class of NetworksHandler and handles all input from the server.

Among all the functionality there are three essential flows which deserves further description. These are openMap(), zoomMap(), and panMap() and are described as follows.

7.9.1 openMap()

The user selects a map using the getSelectedBookmark() method. The request is then given to the MIDPSVGCanvas using openMap(String mapName) which set the borders, zoom level, view frustum position and size of the panning step. Hereafter the openMap(String mapName) calls openMap(String mapName, int fromX, int fromY, int toX, int toY) on the MapBuilder class which makes a Map object and returns it. This map object is then the argument for the drawMap(Map amap) function on the MIDPSVGCanvas which draws the map by getting the beautySVG code on the map object.

7.9.2 zoomMap()

The user zooms by setting the phone in zoom mode using selectMode(MIDPSVGCanvas.MODE_ZOO
on the MIDPSVGCanvas. After the canvas is in zoom mode the user can zoom in and out. This is handled by the keyPressed(int keyCode) method on the MIDPSVGCanvas. When the user zoom in the zoomMap(int pZOOMLEVEL) method is called and this method moves the borders, sets the panning step, sets the view frustum, request a new map from the Mapbuilder and draws it.

7.9.3 panMap()

Panning is just as zooming handled by the MIDPSVGCanvas class. Here the user sets the mobile phone to panning using selectMode(MIDPSVGCanvas.MODE_ZOOM). When the user the pans using keyPressed(int keyCode) the view frustum is moved the length of panning step. When the view frustum hits one of the borders the panMap(int pPanDirection) is called. This method requests a new map from the MapBuilder and this map is drawn using drawMap(Map aMap, int pViewFromX, int pViewFromY).

Chapter 8

Solution overview

As mentioned before this project is based on a client-server solution. What makes this ideal for this project is the fact that the mobile device is capable of exchanging data, with a server using HTTP. A service, that is run on the server, is managing the contact with the mobile devices. This service contacts the database and then structures the map to be sent to the client and returns this to the mobile device. This is illustrated in 8.1.

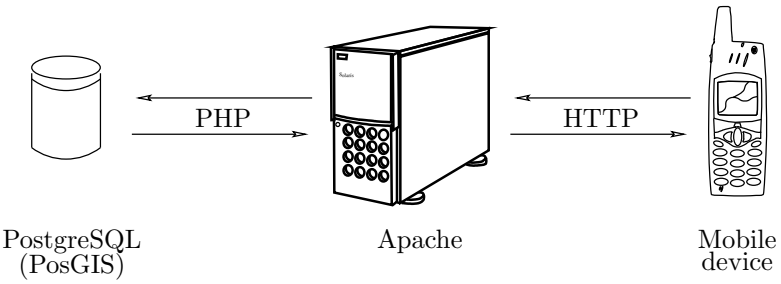


Figure 8.1: Model of the system.

Now that each of the elements have been described individually it is possible to give a full description of the entire solution. This section summarizes all elements of the solution from map creation to use on a mobile phone. Figure 8.3 and 8.2 described what is needed on the server side and Figure 8.5, 8.7, 8.6, and 8.4 illustrates the elements needed on the client side of the solution.

Maps are initially created by a user utilizing the GEO+ language. To ensure that the user-created maps uphold the syntax for GEO+ , all maps are parsed with a parser created specifically for this purpose. This parser takes a map and the CFG for GEO+ in EBNF form and validates the map. It is checked that the map is syntactically and semantically correct. The parser is a part of the translator, which is illustrated in Figure 8.3 in the upper-left corner.

The user can create several maps and each is translated into miniSVG . The miniSVG map on the server side is an "ugly" map that later becomes beautified. MiniSVG is defined by an XML Schema , which is used to validate that the translations from GEO+ -to-miniSVG are correct. This is done by parsing the translated maps using xmllint (see Appendix A), which takes an

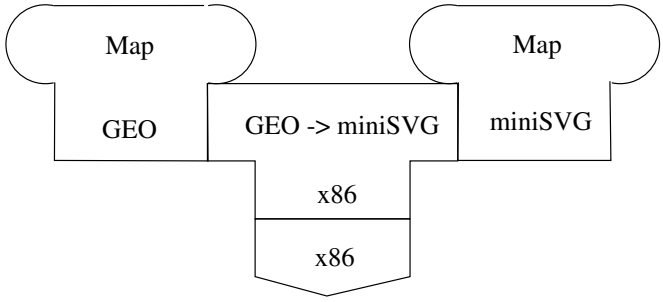


Figure 8.2: The translation of maps from GEO+ -to-miniSVG

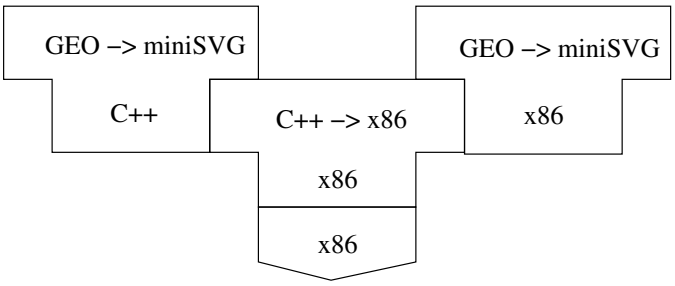


Figure 8.3: The compilation of the GEO+ -to-miniSVG translator

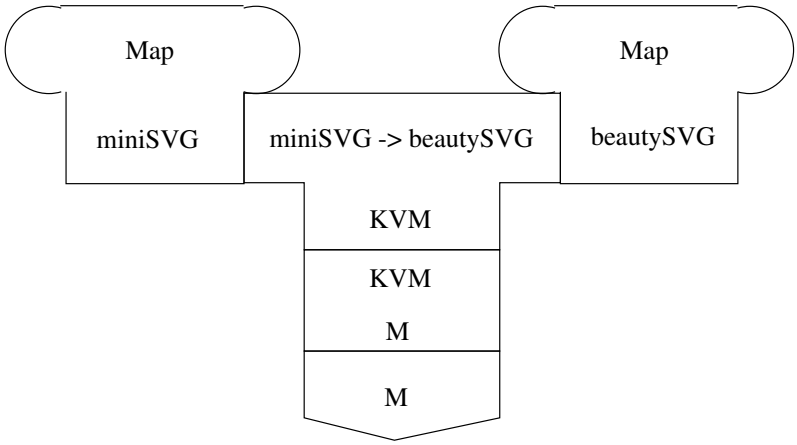


Figure 8.4: The translation of maps from miniSVG -to-beautySVG

XML Schema and a file as input. The parser then validates the XML Schema and the translated document. Now that it has been ensured that the translation is correct the maps are ready to be stored in the database.

To store maps in the database in such a way that enquiries from the mobile phone can be handled efficiently, spatial division has been considered. Nonethe-

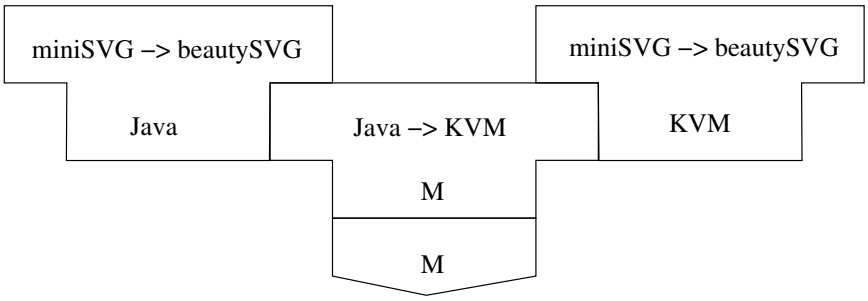


Figure 8.5: The compilation of the miniSVG -to-beautySVG translator

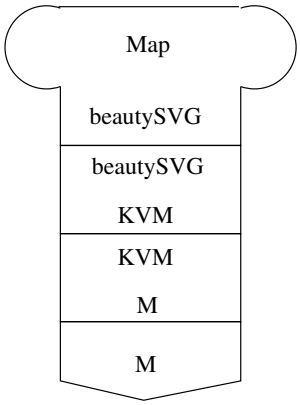


Figure 8.6: The beautySVG to KVM interpreter (The SVG-viewer)

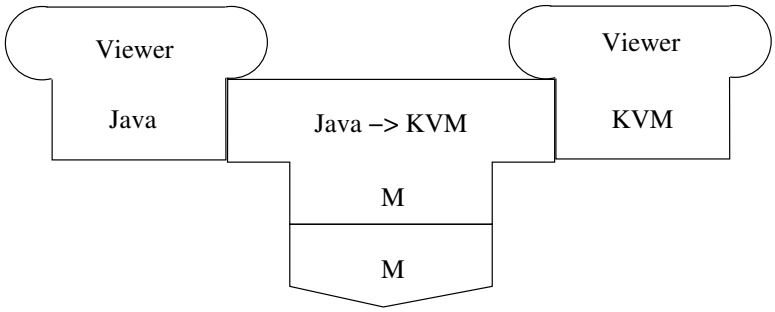


Figure 8.7: Compiling the SVG-viewer

less it was chosen not to implement the best solution for storing the maps. The reason being that the project focus is on languages and compilers. For more details see Section 2.2. Furthermore, the choice of database fell on the database with the best utilization for spatial enquiries being postGreSQL. After maps have been translated, they are stored in this database. This is done by a shell script that scans the maps and creates the tables and references. For further

details see Appendix F.

The use of our system on a mobile phone consists of several elements. First the user has to have a viewer on the mobile phone to view the maps. The viewer is an interpreter and is illustated in Figure 8.5. The viewer retrieves a map when activated but before it is drawn it is beautified on the client. The beautifier is a translator, which translates from miniSVG -to-beautySVG . This translator is illustrated in Figure 8.5. To verify this translation, xmllint is used to parse the output and the XML Schema that defines beautySVG . Zooming on a map has to be handled in a reasonable fashion where old and new data needs to be merged. The main goal for the client is to save bandwidth. To do this it is necessary to reuse as much data already in the client's memory as possible so that the amount of retrieved data is kept to a minimum. This is accomplished by merging and beautifying on the client side. For further details about the viewer, beautifier and merger see sections 7.

To summarize about the different parts of the solution, it consists of the following:

- A translator translating from GEO+ -to-miniSVG .
- A parser parsing the translated map (XML Schema)
- A translator translating from miniSVG -to-beautySVG .
- A parser which verifies the translation (XML Schema).
- An interpreter drawing the map.

Chapter 9

Conclusion

During the development of our system we have not focused much on implementing a high-performance and scalable solution. The primary reason for this is simply due to lack of time. Having a tight project schedule does not allow for dwelling too deeply into the realm of performance tuning and choosing best-suited components for a project. In our project this includes choosing the best-performing programming language for a particular purpose (e.g. database interaction). Whether PHP is actually the best fit for us is not easy to say. Using Perl or Python (and mod_perl / mod_python for Apache integration) instead may in fact have been better for performance and scalability. The best way to determine this is to implement the code in both PHP, Perl, and Python and then do some benchmarking.

When it comes to components such as web server and database, the reason for our choosing, again, related to the fact that we have had a tight project schedule. The fact that our project group had prior experience with Apache and PostgreSQL allowed for a seamless installation and configuration of these server services. MySQL was also considered as a database back-end, but due to a more experimental implementation of the spatial features of MySQL we decided to go with PostgreSQL, although MySQL is likely to have provided a faster database back-end for our system.

In general, we have not focused on optimizing our code with respect to performance and scalability. This should certainly be addressed, if we were to further develop the system.

As for the spatial search we chose to use spatial division by indexing. Never the less we did not use trees to represent the space, but simply queried every element in the scene. This is maybe a slow approach, but we only query the database the number of times which the user moves the map, compared to using a GPS system where we could query the system a lot more. Furthermore the application we developed only uses small maps containing few elements and therefore the consequence of not representing the space as a tree is limited. If the solution should be able to support bigger maps with more elements the time for requesting from the database would then increase and it would be favorable to structure the space using trees.

The technique of using the tombstone diagrams has proven very rewarding for

us. The general idea behind first puzzling all the pieces together was an excellent way to get overview of the project, what phases was completed and what still need work. The puzzle had a flaw in the fact that translating twice might be unnecessary as the two resembles one another. Meaning that the intermediate step might not have been necessary, if we chose to transmit the map in GEO to the mobile device and had the mobile device make a beautiful presentation of this map. The reasons for not doing this was to ease the translation on the client. By first translating the map into miniSVG and transmitting this map, the workload of the client is reduced. This reduction is mainly due to the fact that we do not need to translate the information on the locations of the features. The drawback of this approach is that we increase the amount of data transmitted to the mobile device. This is in contradiction with the one of the major goals in this project namely reducing the use of bandwidth.

After we finished writing the GEO+ CFG, we turned our attention upon how to translate the language into miniSVG . The translator has been implemented, and translates a GEO+ -map to miniSVG . The translator handles the translation, but does not handle errors very well. In its current form, the translator only parses through the map file once, and translates it into miniSVG . This makes it possible for the user to create a map, where two roads intersect even though they never cross each other. If we have had more time, the translator would have been implemented to take care of this. A solution to this could be to store the intersection coordinate, and parse through the map file once more to check whether the coordinates are valid.

We decided to use Tinyline as our basis for our client program. We then extended Tinyline with a translator that can receive a SVG file from the server and translate it into beautified file. We thought that we could use TinyLine with small modifications but we ended up with many modifications of the classes in TinyLine. If we in future projects need to implement a SVG viewer we would probably not use the Tinyline implementation but instead implement our own viewer using the library classes from TinyLine. We have use a lot of time investigating how the TinyLine was implemented and believe that it would be faster to implement our own viewer next time.

The basic features are implemented, but a lot of work need to be done for this to be a sellable system. Work like: time optimization and threading connections. A feature we have not implemented, but a feature we would like to do is making the beautification user defined. This would take an extension of both the server and the translator. User defined styles would be loaded from the server and then saved in the mobile phones record stores. This could be done e.g. every fiftieth time the mobile connect to the server. When the phone needs to translate a SVG file it will then load the style from the record store and then use the defined style to translate the SVG file into a beautified file. This will give a dynamic way of defining the styles.

We have chosen to beautify the SVG file on the client side. This, of course, will cost process time. We could have chosen to send the beautified SVG file from the server and then save time, but we focus on saving money and the most expensive resource we use are the number of characters sent to the client. That

is why we minimize the size of the SVG sent and then add extra characters on the client. This beautification process does cost time, but it is not a heavy calculation job, its only adding styles and in some cases extra lines. Even slow mobile phones should have an easy time processing this beautification and as mobile phones gets faster and faster it should be easy.

During our development process our XML Schema has been helpful in locating any possible non-valid SVG code generated by our GEO+ -to-miniSVG translator. On the client-side, on the other hand, we haven't yet implemented a feature that allows for validating the correctness of the SVG code generated by our miniSVG -to-beautySVG translator. For any future work, a solution could be to let the client push the beautified map to the server (only during a test phase) for validation using the xmllint parser. Another solution would be to copy the XML Schema for beautySVG to the client for client-side validation (still only for a test phase, until the miniSVG -to-beautySVG translator has proven stable and always produces valid output). This solution, however, would an XML parser (e.g. Java-based) on the cell phone.

According to Section 7.2 the price for transmitting data to a mobile device is expensive. One of the more interesting choices during the project was how much data to transmit and exactly what data to request from the server to the mobile device. We showed that we were able to minimize this usage as long as we only transmit small files in little quantity, this is of little interest. It becomes interesting when we transmit large files and/or in greater quantity. In this respect it was very interesting to realize that SVG-files is a very efficient way of drawing maps using a minimum of space.

In all, this project deals with language design and translations. We feel that this project describes the different considerations behind designing languages and the problems concerning these. The main focus have been to use translation and language design in a real world application. Therefore we have implemented a beautifier on the client showing maps that are translated from the original GEO+ language.

Appendix A

Glossary

SGML *Standard Generalized Markup Language* is an international standardized meta-language that has existed since the 1980s. It is a way of formally describing a markup language. A markup language is a way of marking a language with a sequence of characters not intended to be printed but to describe how other characters are to be printed (e.g. underline and italic). [13]

HTML *Hypertext Markup Language* is derived from SGML and is used for publishing hypertext on the World Wide Web using a static set of tags. It is not possible to define any tags yourself. These tags describe publication issues like hyper-linking, images etc. all in the syntax defined by SGML. [19]

XML *Extensible Markup Language* is derived from SGML. But while HTML is designed with a focus on how to display data and how data looks, XML is designed to describe data and focuses on what data is. Unlike HTML there exist no static tags, only dynamic user-defined elements. An XML element is made up of an opening tag between a less-than sign (<) and a greater-than sign (>), and a identical closing tag except for a forward slash (/) that appears before the name of the element. An XML element can contain attributes that are specified after the name and before the greater-than sign. XML is a meta-markup language, a set of rules for creating semantic tags used to describe data. Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. [18]

CSS *Cascading Style Sheets* is a technology to describe how documents are presented on screens. By attaching style sheets to a structured HTML document, authors can influence the presentation of documents without sacrificing device-independence or adding new HTML tags. W3C has actively promoted the use of style sheets on the Web since the Consortium was founded in 1994. [21]

DOM the *Document Object Model* is a platform and language-independent interface that will allow programs and scripts to dynamically access and

update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page. [17]

DTD *Document Type Definition* have been used since the 1970's. The purpose of a DTD is to define legal building blocks, such as tags and attributes, of any SGML-, XML-, and HTML-based document. DTD ensures that each tag is allowed and which tags can appear within other tags. The actual body of the DTD itself contains definitions in terms of elements and their attributes. For example, one could say that a $LIST$ tag can contain an $ITEM$ tag, but $ITEM$ cannot contain $LIST$ tags. This ensures that all the documentation is formatted in the same way. Applications will use a document's DTD to properly read and display the document's contents. Thereby, if one makes changes to the format of a document, these changes can easily be made by modifying the DTD. In DTD one can only define an element as text, and therefore it is not possible to force an element to be numeric. Although attributes can be forced to a specific range of defined values, they cannot be forced to be numbers.

xmllint program parses one or more XML files, specified on the command line as *xmlfile*. It prints various types of output, depending upon the options selected. It is useful for detecting errors both in XML code and in the XML parser itself.

Appendix B

Regarding real time systems...

According to [2] the quality of a real time system is dependent on both the time required for the system to compute a result and the logical correctness of that result.

Different kinds of real time systems include *hard* and *soft* real time systems. A hard real time system requires the system to respond with a result within a predefined deadline. If this deadline is not meet then the result is useless. A soft real time system also has a deadline, however, if this kind of system misses its deadline the result is not immediately useless but the quality of the response decreases over time, eventually also becoming useless. If a soft real time system misses a deadline now and then it will still function correctly, which is not the case for a hard real time system. If a deadline can be misses once in a while, but the information looses its value when delivered late, the deadline is said to be *firm*.

As the project at hand does not involve timeliness, this section elaborates only on creating a logically correct result. In order to create a logically correct result, two different terms will be examined in the following sections.

Fault prevention

In real time systems it can be very dangerous when errors occur. Our SVG map viewer application is, however, not an application that can possibly present any danger if errors and failures occur, but it will be annoying to a user if a map does not show up on the mobile device when requested. Fault prevention attempts to eliminate any faults that may exist in the system before it goes into operation.

Fault avoidance

Avoidance is about avoiding situations where the system will respond with an unusable result. In order to anticipate these situations several actions can be taken according to [2]. First it is important to make the specifications for the system cover a wide array of possible faulty situations. The more rigorous

the formal specifications of the requirements is, the more faulty situations can be avoided. Another step taken in order to avoid faults is using proved design methodologies. In order to avoid making programs with faults it is suggested using a programming language that has facilities for data abstraction and modularity – e.g. using a object-oriented language. Finally, tools for developing software can help the developer manage the complexity of large systems.

Fault removal

Even tough the above will remove most of the faults from the system, a test is still required. The test is made to verify the correctness of the system. This test is about showing the faults present in the system, not about showing that a part of the system is working correctly but produces the wrong result. Sometimes when conducting tests it is difficult to produce realistic conditions in order to verify the correctness of the entire system. It is therefore vital that the simulated test is made in as close to realistic working conditions as possible. Some of the initial assumptions made in the beginning of the development process might not hold and those faulty assumptions might only become visible when the final system is taken into use under the correct working conditions.

In regards to this project we have chosen not to use neither of the above. Instead we use XML Schema described in section 2.4.3. As this section describes, we can verify that any file is a valid XML document. With this in mind we can verify the correctness of the system by checking if the output is a valid XML document.

In general fault prevention will be unsuccessful when either the frequency of maintenance becomes unacceptable or the system can no longer be accessed for maintenance and repair.

Fault tolerance

The term *fault tolerance* is used to verify that, in case of a system fault, it will fail in a controlled manner. Three different kinds of fault tolerance exists: *full*, *fail soft* and *fail safe*. Full fault tolerance allows the system to keep operating even in presence of faults. Fail soft allows the system to keep operating with a lower performance while being recovered and/or repaired. Lastly, fail safe allows the system to maintain its integrity before finally halting for repair.

Dependability

According to [2] the *dependability* of a system is "the dependability of a system is that property of the system which allows reliance to be justifiably placed on the service it delivers". The following sections describe different notions of dependability for our system.

Availability

The availability of a system is the ability of the system to offer its services as time passes. It is in other words the system's ratio of up-time to down-time and therefore its "readiness for usage"[2] and it is important to understand exactly

in what situations the system is to operate. As the service provided by the system is dependant on the number of users, it is possible to have some delay when using its services. In case the service provided is used by emergency units like the fire department and the police it would be a good idea to provide the service on a dedicated server.

Reliability

This term covers "continuity of service delivery"[2]. A denial of service attack could lower the availability of the services provided by the server, this is not a problem as mobile devices are typically limited in ressources, but the HTTP could be accessed by computer(s) in order to achieve this effect.

Safety

The word safety covers a lot of ground in itself. Here it is a measure of the "non-occurrence of catastrophic consequences"[2] and as such it tells us something about the risk of a catastrofy occuring as a result of the system failing.

Confidentiality

In order to ensure "non-occurrence of unauthorized disclosure of information"[2] we have chosen stable and tested software to run on the server. As the only access to the server is made using the mobile device and all information present is not confidential, this point has been neglegted.

Integrity

The ability to achieve "non-occurrence of improper alteration of information"[2] is made possible as an entirely new language to be used for creating maps has been created. This will force anybody interested in altering the data to have knowledge of GEO+ and access to various software on the server. The software that was chosen for the server and client is very safe as it has been tested for various applications.

A way of securing the data on the server is to allow the user of the mobile device limited access to the server - e.g. no persistant information on the clients is present on the server. A very limited interface to the server is another way of restricting access to data on the server.

Maintainability

The ability "aptitude to undergo repairs and evolutions"[2] has been chosen as the single most important aspect of the entire project. The reason for this is that evolution of the solution presented in this project is scalable. This applies to many factors in the project including GEO+ and the ability of the client to zoom and merge maps.

Testing

Testing is a very important phase of development. During the development of the system that takes care of displaying SVG maps on a mobile display we have used an emulator of a mobile phone, running on a regular PC, to simulate the behaviour of our client system during operation. This has helped us root out many bugs in the system before it reaches its final version.

Appendix C

GEO / GEO+

MAPPER	::=	`map` NUMBER$^+$, NUMBER$^+$ (* FUNCTION* *)
FUNCTION	::=	`draw` ROADTYPE [“IDENTIFIER”] ROAD; | `draw` INTERTYPE [“IDENTIFIER”] INTER; | `draw` BUILDTYPE [“IDENTIFIER”] BUILD; | `mark` “IDENTIFIER” (`with`|`as`) MARKER;
ROADTYPE	::=	(`local`|`highway`|`interstate`|`path`|`cycling`|`railroad`)
ROAD	::=	`from` COORD `to` COORD (`using` COORDS)$^?$ | `from` COORD (`using` COORDS)$^?$ `to` COORD
INTERTYPE	::=	(`cross`|`roundabout`|`t-cross`|`ramp`)
INTER	::=	`where` “IDENTIFIER” (`crosses`|`intersects`) IDENTIFIERS
BUILDTYPE	::=	(`residential`|`commercial`|`factory`)
BUILD	::=	`from` COORD `to` COORD
MARKER	::=	(`gasstation`|`church`|`zoo`|`tourist-attraction`| (`recommended`)$^?$ `speed limit of` NUMBER$^+$ `km/h`)
IDENTIFIERS	::=	“IDENTIFIER” (`and` “IDENTIFIER”)*
IDENTIFIER	::=	(NUMBER|LETTER) (NUMBER|LETTER|<SPACE>)*
COORDS	::=	COORD (`and` COORD)*
COORD	::=	NUMBER$^+$, NUMBER$^+$(,NUMBER$^+$)$^?$
NUMBER	::=	`0..9`
LETTER	::=	`A..Z` | `a..z`

Table C.1: EBNF for GEO .

MAPPER	::=	map (COORD,COORD) { FUNCTION* }
FUNCTION	::=	road(ROADTYPE, VARIABLE , EXPRESSION$^+$); | nature(NATURETYPE, (COORDX,COORDY)$^+$); | building(BUILDTYPE, EXPRESSION$^+$); | intersection(INTERTYPE, VARIABLE, VARIABLE ,EXPRESSION); | sign(MARKER, EXPRESSION); | waypoint(MARKER,(COORDX,COORDY));
EXPRESSION	::=	(COORDX,COORDY) | SUBVARIABLE
SUBVARIABLE	::=	VARIABLE(NUMBER$^+$)
VARIABLE	::=	IDENTIFIER
IDENTIFIER	::=	[NUMBER|LETTER]$^+$
ROADTYPE	::=	local|highway|interstate|path|cycling|railroad
INTERTYPE	::=	cross|roundabout|t-cross|ramp
BUILDTYPE	::=	residential|commercial|factory|farm
NATURETYPE	::=	lake|forest|coastline|river
MARKER	::=	[NUMBER$^+$|LETTER*]$^+$ | [NUMBER*|LETTER$^+$]$^+$
NUMBER	::=	0..9
COORD	::=	-1024..0..1024
COORDX	::=	0..800
COORDY	::=	0..600
LETTER	::=	A..Z|a..z

Table C.2: EBNF for GEO+ .

Appendix D

J2ME

Hvis der er tid må der gerne væres source kode eksempler på J2ME, .NET og NAtive applications, så man kan se syntaks forskelle på de forskellige sprog.

When we started to collect informations about software that could be implemented on a mobile devices we found three main programming environment.

Choises

- Native applications
- The Microsoft .NET compact framework
- The J2ME platform

Native applications

Native applications are developed for a certain operation system such as Windows CE or EPOC. These applications are traditionally written in C/C++ or in some basic language. These applications can work in an offline environment, but they are completly depended on the operating system that is running on the device.

Creating native applications requires a higher level of expertise than writing ordinary thin client applications. The reason is because of the complexity of the user interface and the wide variations in OS capabilities when writing native applications. In addition, native applications are not portable to other operating systems without changes of the written code or compiling the source code for the specific operating system.

.NET compact framework

Before we describe the .NET compact framework, it is necessary to give an overview of the .NET framework. The reason why is that the .NET compact framework is a subsection of the .NET framework thus providing the same benefits as the .NET Framework.

The .NET Framework consists of two main parts: the common language runtime (CLR) and a unified set of class libraries, including ASP.NET for Web

applications and Web services, Windows Forms for smart client applications, and ADO.NET for loosely coupled data access. The CLR is an environment in which it is possible to run its .NET applications. .NET framework uses XML services to connect applications/software together. The idea behind XML services is to exchange data in a standardized form between different systems and applications. A .NET compiler compiles source code into intermediate language which is called "MSIL" (much like Java Byte code). The CLR then take care of converting the intermediate code to the host machine code opposite to Java which is interpreted by a Virtual Maschine(JVM). Java programmers are familiar with JRE (Java Runtime Environment). CLR can be considered as an equivalent to JRE.

The Microsoft .NET Compact Framework shortend .NET CF is the smart device development framework for Microsoft .NET. The .NET Compact Framework is designed specifically for resource-constrained devices, such as PDAs and smart mobile phones. The .NET CF consists of a base class libraries and has a few additional libraries that are specific to mobility and device development. The .NET Compact Framework runs on a JIT Compiler. The CLR in .NET CF is built so that it runs more efficiently on small targeted devices that are limited in memory and resources. The .NET Compact Framework only runs on Windows CE/Pocket PC-powered high-end PDAs.

J2ME

J2ME is part of Sun's Java 2 family of platforms. J2ME is a set of core Java API's and execution environment particulary tailored for resourceconstrainted devices such as mobile phones, two way pagers and PDA's. The core in the J2ME platform consists of two different configurations. CDC (Connected Device Configuration) and CLDC (Connected Limited Device Configuration). A configuration specify the central libraries and the properties for the virtual machine on the device.

CDC:

CDC is designed for devices that has more memory, faster processors and more network bandtwidth such as TV boxes, residence gateways and high-end PDA's. CDC include a full virtual Java machine.

CLDC:

CLDC is the smallest of the two configurations. CLDC is designed for devices that has periodic network connection, slow processor, and limited memory. These devices e.g. are mobile phones, PDA's and two way pagers. These devices have typically 16 or 32 bit CPU and minimum 128 kb or 512 memory available for the Java J2ME platform.

MIDP:

On top of the two above-mentioned configurations one will find the profiles, which defines the functionality for a specific category of devices. MIDP is a profile for CLDC based devices e.g. mobile phones. MIDP defines among other

things the the utilization of the user interface, consistent storage and network functions.

J2ME allows for the best of both (native and thin clients) worlds creating a new breed of mobile applications knows as smart clients. J2ME provides all the elements such as API's for local persistence, network connectivity and user interface. Because J2ME is Java, J2ME inherits all the benefits of the Java language, including lower development costs than developing native clients targeted at multiple platforms. The above-mentioned arguments are some of the argumens for why we have chosen to use the J2ME platform to develop software for a mobile device. Another reasson is that all the group members are well experienced in programming in the language Java. J2ME uses the same syntax as Java, with some minor limitations.

J2ME versus native applications and .NET Compact Framework

Before we decide to choose the platform, we have to use on the mobile device we have to compare the above mentioned technologies. Compared with mobile devices on native platforms (such as eMbedded Visual C++ or C++ SDKs for the Symbian OS), Java- or .Net environments greatly improve developer productivity, application reliability, and mobile code security. As mentioned earlier creating native applications requires a higher level of expertise compared to J2ME and the .NET Compact Framework. Therefore we have chosen not to use native applications. The choice is now between the two technologies: J2ME and the .NET Compact Framework.

J2ME versus .NET CF

Multiple platforms

Today there are many different operating systems available for mobile devices. There are also many vendor specific operating systems such as Nokia Series 40. .NET CF only supports one OS platform.(Windows CE/Pocket PC operating system). On the otherhand the J2ME framework can run on many different OS platforms. Most of the different OS platforms have support for J2ME. Many vendors have support for J2ME in their mobile devices. Therefore it is possible to have Java source code running on different platforms. Java allows one to develop applications for many different OS platforms.

The specification process

The .NET CF provides at set of tools and API's. It is only Microsoft that decides which tools and API that should be in the framework. So it is Microsoft, not the costumer, who decides the important features that should be in the .NET CF.

Sun has taken another approach to evolve their J2ME framework. Sun has created a committee which is named JCP (Java Community Process). This committee consists of mobile solution providers. Sun has veto power only on Java language specifications. After the API specification is developed, each

company can develop its own implementation. This ensures portability. Since all API specifications are reached by industry consensus, most likely they will be supported in the future. The JCP develops all current J2ME configurations, profiles, and optional packages.

Development tools

One of the most important reasons for choosing J2ME and .NET CF is to leverage excisting developer skills - to ease the development process. Microsoft's flagship Visual Studio .NET is an excellent development tool that provides similar design interfaces for desktop and mobile applications, e.g. to migrate a desktop UI design to .Net CF, you merely copy and paste visual components to a new designer window. Visual Studio .Net supports debugging on both high-fidelity emulators and real devices. However, Visual Studio .Net is not a cheap product. As of today, no free command-line tool exists for .Net CF development.

On the J2ME front there exist many different vendor specific toolkits. Sun's J2ME Wireless ToolKit is a widely used development tool. For most developers, IDEs are still essential. All major Java IDEs now have J2ME modules or plug-ins. Below is listed some of the most well-known IDEs which have modules or plugins for J2ME.

- Sun ONE Studio Community Edition with wireless modules is free and has support for enterprise features.
- Eclipse is free and one can get a plugin for J2ME to Eclipse
- JBuilder with MobileSet has a great visual UI designer and good UML design support. JBuilder is a commercial product.
- IBM WebSphere Studio Device Developer is based on the Eclipse IDE platform. It supports both on-device and emulator debugging.

  A big challenge for all J2ME IDEs is vendor SDK integration. Many device vendor provides their own SDKs for their device emulators and proprietary J2ME extensions.

Conclusion

The .Net CF and J2ME are both excellent platforms for developing software for mobile devices. An disadvantage of .Net CF is that is runs only on Windows-powered high-end PDAs. J2ME supports a modular design and is portable across a variety of devices.

J2ME APIs undergo rigorous standardization processes to ensure wide industry support as apposed to Microsoft which controls the whole specifications process of .NET CF.

Visual Studio .NET is an easy to use and a powerful tool for .NET CF development. Visual Studio .NET is, however, the only tool available for .NET CF development and it is neither a cheap product. To the J2ME platform there exists many different tools for J2ME development. Many of these tools are powerful open source products.

We have chosen to use J2ME as the framework for mobile device development. The reason why is that most of the members of our group have programming experince in Java, and that J2ME is portable across different platforms and mobile devices. We therefore do not have to choose a specific mobile device or operating system for our project. One can say Write Once, Run Anywhere. Another reason is that all the members of the group have different Linux distributions or Unix operating systems installed on their computers. Microsoft have chosen not to port the .NET CF to the Linux/Unix platform. Therefore it is impossible to run the .NET CF on the group member's computers.

Appendix E

PostgreSQL + PostGIS + PHP install / config

*** postgresql 7.4.1 + PostGIS 0.8.1 ***

```
# Requirements for Debian Woody: Recent version of Java, Ant, Bison.
# Get Sun's Java (v1.4.x) and install it in /usr/local/java.
# Grab Ant from unstable with 'apt-get -t unstable install ant'.
# Backport Bison using 'apt-get source bison', 'apt-get build-dep bison',
# 'cd bison-<version>', 'dpkg-buildpackage -b -uc -us -rfakeroot'

echo "export JAVA_HOME=/usr/local/java" >> ~/.bashrc && source ~/.bashrc
echo "export PATH=/usr/local/java/bin:$PATH" >> ~/.bashrc && source ~/.bashrc
cd src/postgresql-7.4.1  (it's assumed that postgresql was untar'ed to this dir)
./configure \
--prefix=/usr/local/postgresql \
--with-java \
--with-readline \
--enable-debug \
--enable-thread-safety \
--enable-depend \
--enable-syslog
make
make install  (as root)

# Post-configuration:
groupadd -g 600 dbserv
groupadd -g 601 dbdata
useradd -d /usr/local/postgresql -g 600 -u 600 -s /bin/true dbserv
useradd -m -d /home/dbdata -g 601 -u 601 -s /bin/bash dbdata
mkdir /usr/local/postgresql/data
chown -R dbserv:dbserv /usr/local/postgresql
chown -R dbdata:dbdata /usr/local/postgresql/data
touch /usr/local/postgresql/logfile
chown dbdata:dbdata /usr/local/postgresql/logfile
```

```
su - dbdata
/usr/local/postgresql/bin/initdb -D /usr/local/postgresql/data
/usr/local/postgresql/bin/pg_ctl -D /usr/local/postgresql/data \
-l /usr/local/postgresql/logfile start
/usr/local/postgresql/bin/createdb test
/usr/local/postgresql/bin/psql test

# Set a password for your user:
ALTER USER dbdata WITH PASSWORD 'some_passwd';

# Create database:
CREATE DATABASE svg_db;
CREATE USER svg_user PASSWORD 'some_passwd';

# Create SQL syntax in text file for the database structure (tables, etc).
# Then Switch to svg_db and issue:
\i /path/to/foobar.sql

GRANT ALL ON geometry_columns,spatial_ref_sys TO svg_user;


# POSTGIS:
cd src/postgresql-7.4.1/contrib
tar zxvf ../../postgis-0.8.1.tar.gz
cd postgis-0.8.1
# edit Makefile to suit your needs...
make
make install  (as root)

# Enable PL/pgSQL
createlang plpgsql svg_db (as dbdata)

# Load the PostGIS object and function definitions into your database
psql -d svg_db -f /usr/local/postgresql/share/contrib/postgis.sql (dbdata)

# Load the spatial_ref_sys.sql definitions file and populate the
# SPATIAL_REF_SYS table
psql -d svg_db -f /usr/local/postgresql/share/contrib/spatial_ref_sys.sql (dbdata)


# DATABASE DUMP SCRIPT:
#
##############################################################################
#!/bin/sh
# psqldump script

DATE=`date +%y%m%d-%H%M%S`

echo "Beginning to dump all databases..."
DUMP_DIR=/home/dbdata/
```

```
DUMP_FILE=psqldump-$DATE.sql
touch $DUMP_DIR$DUMP_FILE $DUMP_DIR$DUMP_FILE.gz
chmod 600 $DUMP_DIR$DUMP_FILE $DUMP_DIR$DUMP_FILE.gz
/usr/local/postgresql/bin/pg_dumpall -U dbdata > $DUMP_DIR$DUMP_FILE
gzip -f $DUMP_DIR$DUMP_FILE
echo "All databases have been dumped to $DUMP_DIR$DUMP_FILE.gz"

exit 0
##############################################################################


# START/STOP SCRIPT:
#
##############################################################################
#!/bin/sh
# Postgresql start/stop script

PIDFILE=/usr/local/postgresql/data/postmaster.pid

case "$1" in
 'start')
  /usr/local/postgresql/bin/pg_ctl -D /usr/local/postgresql/data \
  -l /usr/local/postgresql/logfile start
  ;;
 'stop')
  /usr/local/postgresql/bin/pg_ctl -D /usr/local/postgresql/data -m smart stop
  ;;
 'restart')
  /usr/local/postgresql/bin/pg_ctl -D /usr/local/postgresql/data -m smart restart
  ;;
 'reload')
  /usr/local/postgresql/bin/pg_ctl -D /usr/local/postgresql/data reload
  ;;
 'status')
  /usr/local/postgresql/bin/pg_ctl -D /usr/local/postgresql/data status
  ;;
 *)
  echo "Use: $0 { start|stop|restart|reload|status }."
  exit 1
  ;;
esac
exit 0
##############################################################################


# PHP:
# Compiling php-4.3.5
./configure --prefix=/usr/local/php4 \
--with-apxs=/usr/bin/apxs \
--with-pgsql=/usr/local/postgresql \
```

```
--enable-ftp \
--enable-gd-native-ttf \
--enable-memory-limit \
--enable-bcmath \
--enable-calendar \
--enable-trans-sid \
--enable-track-vars \
--enable-sockets
make
make install
cp php.ini-dist /usr/local/php4/lib/php.ini
```

Appendix F

BASH Script: minisvg-to-sql.sh

```
#!/bin/sh

#set -x

#MAPFILE=mini_test.svg
echo "Enter path to map."
read MAPFILE

data_table=data
map_table=map

testing="no"

echo -e "\n\033[31mgrep'ing for objects and putting them into their"
echo -e "respective {path,rect,polygon}-objects.txt file...\033[0m"
egrep "^<path" $MAPFILE > path-objects.txt
egrep "^<rect" $MAPFILE > rect-objects.txt
egrep "^<polygon" $MAPFILE > polygon-objects.txt

echo -e "\n\033[31mReading object files line by line and doing db inserts...\033[0m"
for x in {path,rect,polygon}-objects.txt; do
  { while read object; do
    echo -e "\n\n\033[34m########################################\033[0m"
    echo -e "\n\n\033[31mNext object is $object\033[0m"

    # Get id of the current object
    id=`echo $object | awk -F= '{ print $2 }' | sed -e 's/\ .*//g' -e 's/\"//g'`
    echo -e "\033[31mid is: $id\033[0m"

    # Get the set of points of the current object
    point_set=`echo $object | awk -F= '{ print $3$4$5$6 }' | \
        sed -e 's/[A-Z,a-z,\"\/,>]//g'`
    echo -e "\033[31mpoint_set is: $point_set\033[0m"

    # Get zoom level of the current object
    zoom_lvl=`echo $id | cut -f4 -d_`
    echo -e "\033[31mzoom_lvl is: $zoom_lvl\033[0m"

    echo -e "\n\033[33mSQL statement: INSERT INTO $data_table VALUES \
        ('$id','$object');\033[0m"
    if [ "$testing" == "no" ]; then
      echo "INSERT INTO $data_table VALUES ('$id','$object');" | \
        psql svg_db svg_user -f -
    fi

    # Put points pair-wise on each line in file points.txt
    # How is it done?: sed looks for a strings sequence consisting
    # of [number][whitespace][number]    (numbers may have a dot in them).
```

```
    # Detailed sed search description (don't mind the escaped "(" & ")" ):
    #   1) \(\.\?          (0 or 1 dot)
    #   2) [0-9|\.]\+\     (0 or more digits OR 0 or more dots, and a whitespace)
    #   3) .\?             (0 or 1 dot)
    #   4) [0-9|\.]\+\)\   (0 or more digits OR 0 or more dots, and a whitespace)
    #
    # Then the second whitespace is placed with a : (colon).
    # Finally, tr is used to translate : (colons) into newlines.
    echo $point_set | sed -e "s/\(\.\?[0-9|\.]\+\ \.\?[0-9|\.]\+\)\ /\1:/g" | \
        tr : \\n > points.txt
    for y in points.txt; do
      { while read point; do
        echo -e "\n\033[33mSQL statement: INSERT INTO $map_table VALUES \
            ('$id',GeometryFromText('POINT($point)')::text);\033[0m"
        if [ "$testing" == "no" ]; then
          echo "INSERT INTO $map_table VALUES \
            ('$id',GeometryFromText('POINT($point)')::text,$zoom_lvl);" | \
            psql svg_db svg_user -f -
        fi
      done } < $y
    done

  done } < $x
done

echo -e "\n\033[31mRemoving temporary txt files...\033[0m"
rm -f {path,rect,polygon}-objects.txt points.txt

exit 0
```

Appendix G

XML Schema: miniSVG_XML-Schema.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.sentinel.dk"
            xmlns="http://www.sentinel.dk"
            elementFormDefault="qualified">

<xsd:simpleType name="id_string">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[a-z]+_[a-z0-9]+_[a-zA-Z0-9\-]*_(0|1|2)"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="d_string">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="M[0-9]+\s[0-9]+(\s(L[0-9]+\s[0-9]+\s)*(L[0-9]+\s[0-9]+))?Z?"/>
  </xsd:restriction>
</xsd:simpleType>

  <xsd:element name="svgObjects">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="svg">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="path" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
                  <xsd:attribute name="id" type="id_string" use="required"/>
                  <xsd:attribute name="d" type="d_string" use="required"/>
                </xsd:complexType>
              </xsd:element>
              <xsd:element name="g" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="path" minOccurs="2" maxOccurs="unbounded">
                      <xsd:complexType>
                        <xsd:attribute name="id" type="id_string" use="required"/>
                        <xsd:attribute name="d" type="d_string" use="required"/>
                      </xsd:complexType>
                    </xsd:element>
                  </xsd:sequence>
                  <xsd:attribute name="id" type="xsd:string" use="required"/>
                  <xsd:attribute name="style" type="xsd:string" use="optional"/>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="id" type="xsd:string" use="required"/>
            <xsd:attribute name="width" type="xsd:string" use="required"/>
            <xsd:attribute name="height" type="xsd:string" use="required"/>
```

```
            <xsd:attribute name="x" type="xsd:decimal" use="optional"/>
            <xsd:attribute name="y" type="xsd:decimal" use="optional"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Appendix H

XML Schema: beautySVG_XML-Schema.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.sentinel.dk"
            xmlns="http://www.sentinel.dk"
            elementFormDefault="qualified">

<xsd:simpleType name="id_string">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[a-z]+_[a-z0-9]+_[a-zA-Z0-9\-]*_(0|1|2)"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="d_string">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="M[0-9]+\s[0-9]+(\s(L[0-9]+\s[0-9]+\s)*(L[0-9]+\s[0-9]+))?Z?"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="path_style">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="(fill:(red|yellow|green|blue|none);)?(stroke:(black|yellow|orange);)?(stroke
  </xsd:restriction>
</xsd:simpleType>

  <xsd:element name="svgObjects">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="svg">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="path" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
                  <xsd:attribute name="id" type="id_string" use="required"/>
                  <xsd:attribute name="d" type="d_string" use="required"/>
                  <xsd:attribute name="style" type="path_style" use="required"/>
                </xsd:complexType>
              </xsd:element>
              <xsd:element name="g" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="path" minOccurs="2" maxOccurs="unbounded">
                      <xsd:complexType>
                        <xsd:attribute name="id" type="id_string" use="required"/>
                        <xsd:attribute name="d" type="d_string" use="required"/>
```

```
                        <xsd:attribute name="style" type="path_style" use="required"/>
                      </xsd:complexType>
                    </xsd:element>
                  </xsd:sequence>
                  <xsd:attribute name="id" type="xsd:string" use="required"/>
                  <xsd:attribute name="style" type="xsd:string" use="optional"/>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="id" type="xsd:string" use="required"/>
            <xsd:attribute name="width" type="xsd:string" use="required"/>
            <xsd:attribute name="height" type="xsd:string" use="required"/>
            <xsd:attribute name="x" type="xsd:decimal" use="optional"/>
            <xsd:attribute name="y" type="xsd:decimal" use="optional"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Index

Bibliography

[1] Mary Bellis. Selling the cell phone. http://inventors.about.com/library/weekly/aa070899.htm.

[2] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 3rd edition edition, 2001. ISBN 0-201-72988-1.

[3] Laurynas Speičys. Christian S. Jensen. Augustas Kligys. Computational data modeling for network-constrained moving objects. Department of Computer Science, Aalborg University.

[4] C. Hage C. S. Jensen T. B. Pedersen L. Speičys I. Timko. Integrated data management for mobile service in the real world. Euman A/S and Department of Computer Science, Aalborg University.

[5] Mobile-Electronics.net. Motorola a768. http://www.mobile-electronics.net/motorola/motorola_a768/.

[6] Inc PageWise. The history of the cellular phone. http://allsands.com/History/Objects/cellphone_wzr_gn.htm.

[7] Julius Mong (PhD). Reconciling the structure and appearance of digital documents using xml and svg. http://www.cs.nott.ac.uk/~jxm/PhD/.

[8] Terrence W. Pratt and Marvin V. Zelkowitz. Programmig languages. design and implementation. ISBN 0-13-027678-2.

[9] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison Wesley, 1989. ISBN 0-201-50255-0.

[10] Sonofon. Priser og takster for gprs 2-go. http://tinyurl.com/375so.

[11] SUN. Javaserver pages technology. http://java.sun.com/products/jsp/.

[12] Unknown. Example of svg. http://wdvl.internet.com/Authoring/Languages/XML/SVG/DoingIt/Examples/t%iger.svg.txt.

[13] Unknown. A gentle introduction to sgml. www.isgmlug.org/sgmlhelp/g-sg.htm.

[14] Unknown. Tdc - erhverv - mobilitet - abonnementer - gprs - priser. http://erhverv.tdc.dk/artikel.php?dogtag=tdc_e_mobil_gprs_pris.

[15] Unknown. Tinyline - mobile svg software for j2me. http://www.tinyline.dk.

[16] W-Insights.com. All about wireless and web communication. http://www.w-insights.com/tutorial/oct2002/gprs.htm.

[17] W3C. Document object model. www.w3.org/DOM.

[18] W3C. Extensible markup language. www.w3.org/XML.

[19] W3C. Html home page. www.w3.org/MarkUp.

[20] W3C. Scalable vector graphics (svg) 1.1 specification. www.w3.org/TR/SVG11.

[21] W3C. Web style sheets home page. www.w3.org/Style.

[22] W3C. World wide web consortium. http://www.w3c.org.

[23] W3C. Xml-schema. www.w3.org/XML/Schema.