MASTER’S THESIS

An Integrated Scanner Framework for the Borland Delphi IDE

by
P.J.M. van Zutven

Supervisors:  dr. ir. C. Hemerik
dr. ir. G. Zwaan

Eindhoven, October 24, 2005
Abstract

In this thesis the design and development of a framework of lexical scanners and scanner-generators for the Borland Delphi environment is detailed. This framework provides easy and intuitive interfaces to define language definitions, as well as a set of scanners and scanner-generators that can use these definitions. The framework is integrated in the Delphi IDE, enabling it to place generated scanners directly into active projects. Additionally, the framework provides interfaces that can allow specially designed components to use any scanner from the framework.

Keywords: lexical analyzer, scanner, scanner generator, language, language definition
Preface

When I first started work on my master’s thesis for Software Engineering, I was asked to design and develop a toolkit for abstract and concrete syntax-trees. For some time I worked on this before me and my supervisors came to the conclusion that it proved itself a bit too abstract for me. By shifting the focus to the tools that would eventually be needed for the toolkit, the idea for a scanner framework for the delphi IDE was born. This scanner framework would include pre-programmed and generated scanners into a single system that scanner-using components could use. This is the project I ended up doing, though ironically enough, when the time came to generate code for the T-Recipe scanner, I realised that the initial toolkit would have come in quite handy.

The road travelled while doing the project has been as bumpy and bendy as the road from Eindhoven station to university. Fortunately for me, my supervisor didn’t give up on me and, with the help of my friends, my family and, most of all, my girlfriend, I eventually managed to arrive at my destination. For this I would like to thank them all.
Chapter 1

Introduction

1.1 The situation

During the development of applications programmers can encounter situations where a string has to be subdivided into separate lexical elements. This can range from cutting up a comma-separated list of words to something as complex as reading in and scanning a scripting language. Lexical scanners enable said programmer to do this.

The subject of lexical scanning is well-documented and many methods are known. Many of these methods can be automated. To save the programmer the effort and time of handwriting a lexical scanner, a large range of scanner generators has been created, most notably LEX [Lesk75] and its many variants like Bison [Dono85] and flex, but also other like Coco/R [Moss85]. These scanner generators take a definition of the various lexical elements and generate the source-code for the scanner. These generated scanners can then be used by the programmer in his project. This is often easier, faster and less error-prone, enabling the programmer to focus on other important parts of his application.

Three aspects are of particular importance for these scanner generators: ease of use, generality and speed. The ease of use determines how quickly and easily the programmer can write a language definition and generate the corresponding scanner. The generality is a measure of the amount of freedom the programmer has in his language definition, i.e. its ability to handle complex definitions or the ability to generate for a number of languages. Finally the speed determines the relative execution speed of the generated scanner, most commonly compared to handwritten scanners for the same language.

1.2 The shortcomings

At the time of this writing there are a multitude of scanner-generators available. Some scanner-generators are adaptations or rewrites of another existing scanner-generator, only for a different platform or language. LEX [Lesk75] and its many variants are a good example of this. Other scanner generators try to use different ways of defining the language or use different algorithms for scanner-generation to reach a more general or faster variant.
Many of these scanner generators seem to have a few things in common. Firstly, in many cases, the generator is an external program, generating the source for the scanner which then needs to be included into the project. Secondly many of these generators are practically command-line driven, taking a definition file that contains the language definition and any associated settings as input and outputting one or more files containing the generated scanner. Finally, most scanner-generators only offer one method to generate the scanner and, if this method is not suitable for the particular project, then the only option is to find a different scanner-generator. This other scanner-generator likely generates a slightly different scanner, requiring the programmer to spend more time to make it suitable for his application.

1.3 The solution

This project tries to address these three problems. The intention is to create a toolkit that contains several different scanners, some of which are fixed, some of which are generated. All of these scanners are based on the same basic interface. Developed for the Delphi environment, all generated scanners will be a direct part of the project in which the scanner is needed. All will have an intuitive and easy-to-use interface that enforces correctness of the input. The scanners generated by these scanner-generators will be available to other components, designed to use a scanner, like syntax highlighting tools and parsers.

Since the project encompasses several scanners built on the same basis, the ease-of-use, speed and generality become a per-scanner concern. Several scanners can be made, each focusing on one or more of the three aspects at the cost of the other ones. Depending on the requirements during the project, it is then possible to, for example, switch from an easy, but slow scanner to a more complex, but faster scanner. Methods to use language definitions for multiple types of scanners and scanner-generators would make this transition even easier.

1.4 Structure of Thesis

This thesis details the design and construction of a framework of scanners and scanner-generators for use with the Delphi IDE. Chapter 2 covers the basic definitions and concepts needed in the rest of the document. Chapter 8 delves into the Delphi IDE, its capabilities and how those capabilities can be applied to the scanner framework. In chapter 3 the actual framework is developed based on an analysis of its requirements. Chapter 4 covers two methods to represent language definitions for the scanner used in the framework, as well as the interfaces that allow the user to modify them. Chapters 5, 6 and 7 detail the development and construction of three different scanners, respectively the paradigm scanner, the T-recipe scanner and the automaton-scanner. In chapter 10 the three scanner-types are put to the test, making sure that each returns the correct results and, where the scanners are compatible, whether the scanners return the same results. Chapter 9 discusses the design and development of a syntax highlighter designed specifically to use the scanners from the scanner framework. Finally, chapter 11 contains an evaluation of how well the stated goals have been reached, followed by conclusions drawn from the development of the framework and information on possible future avenues of research.
Chapter 2

Preliminaries

2.1 Regular Expressions

Regular expressions are well-documented throughout computer science literature. To avoid reinventing the wheel, the definition from “Parsing Theory - Vol. I: Languages and Parsing” [Sipp88] is given here for the base set of regular expressions.

A regular expression over an alphabet $V$ is a well-formed expression whose arguments are elements in $V \cup \{\epsilon\}$ and whose operators are chosen from among $*$ (repetition), $\cdot$ (concatenation), and $|$ (selection). Here $\epsilon$ is a special symbol denoting the empty string.

Let $e$ be a string over $V \cup \{\epsilon, *, \cdot, |, \}\}$. 

- $e$ is a regular primary over $V$ if it is a symbol in $V \cup \{\epsilon\}$ or is of the form $(e_1)$ where $e_1$ is a regular expression over $V$.
- $e$ is a regular factor over $V$ if it is a regular primary over $V$ or is of the form $e_1^*$, where $e_1$ is a regular factor over $V$.
- $e$ is a regular term over $V$ if it is a regular factor over $V$ or is of the form $e_1 \cdot e_2$, where $e_1$ is a regular term and $e_2$ is a regular factor over $V$.
- $e$ is a regular expression over $V$ if it is a regular expression over $V$ or is of the form $e_1 \mid e_2$, where $e_1$ is a regular expression and $e_2$ is a regular term over $V$.

The language $L(e)$ denoted (or described) by a regular expression $e$ over $V$ is defined inductively on the length of $e$ as follows:

- $L(\epsilon) = \epsilon$
- $L(\alpha) = \alpha$ for $\alpha \in V$.
- $L((e)) = L(e)$ for all regular expressions $E$ over $V$.
- $L(e^*) = L(e)^*$ for all regular factors $E$ over $V$.
- $L(e_1 \cdot e_2) = L(e_1) \cdot L(e_2)$ for all regular terms $e_1$ and regular factors $e_2$ over $V$. 
$L(e_1 \mid e_2) = L(e_1) \cup L(e_2)$ for all regular expressions $e_1$ and regular terms $e_2$ over $V$.

$\text{RE}(X)$ is the set of all possible regular expressions over the alphabet $X$.

Two functions $\text{null}$ and $\text{first}$ can be defined on the regular expressions. Function $\text{null}(e)$ returns whether the expression $e$ is capable of producing the empty language. Function $\text{first}(e)$ returns the set of characters that can appear first in the expression.

$\text{null}(\epsilon) = \text{true}$
$\text{null}(\alpha) = \text{false}$
$\text{null}(e^*) = \text{true}$
$\text{null}(e_1 \cdot e_2) = \text{null}(e_1) \land \text{null}(e_2)$
$\text{null}(e_1 \mid e_2) = \text{null}(e_1) \lor \text{null}(e_2)$

$\text{first}(\epsilon) = \emptyset$
$\text{first}(\alpha) = [\alpha]$
$\text{first}(e^*) = \text{first}(e)$
$\text{first}(e_1 \cdot e_2) = \begin{cases} \text{first}(e_1) & \text{if } \neg \text{null}(e_1) \\ \text{first}(e_1) \cup \text{first}(e_2) & \text{if } \text{null}(e_1) \end{cases}$
$\text{first}(e_1 \mid e_2) = \text{first}(e_1) \cup \text{first}(e_2)$

### 2.2 Extended regular expressions

Although the set of regular expressions described above is complete, it can be difficult to express certain simple expressions that are common when using regular expressions. Given a set of characters, one may often wish to accept any character from this set, i.e. perform a selection on all elements in the set. Even for a small set like the ten digits, this already leads to an expression like $'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'$. It would be easier to just specify the range of characters. A range will be denoted by square brackets with a pair of characters inside. The digits for example, can then be written as $['0' \ldots '9']$. The statement is equivalent to a selection on all characters, so $L(['\alpha' \ldots '\beta']) = L(\alpha \mid \ldots \mid \beta)$.

It might be desirable to perform a repetition one or more times instead of zero or more times. This is commonly denoted by the expression $e^+$. This statement is equivalent to $e \cdot e^*$, i.e. $L(e^+) = L(e \cdot e^*)$.

Likewise it might be needed at times to make part of a regular expression optional, i.e. handle part of the expression zero or one times. This is often denoted by the expression $e?$. This expression is the same as a selection on epsilon and the expression $e$, so $L(e?) = L(\epsilon \mid e)$.

A sequence of terminals like $'b' \cdot 'e' \cdot 'g' \cdot 'i' \cdot 'n'$ can be rewritten by combining all terminals into a single string $'begin'$. For any given string $\sigma$, the language it generates is the string itself, i.e. $L(\sigma) = \sigma$ for $\sigma \in V^*$.

### 2.3 Lexical Scanners

A lexical scanner or lexical analyzer is an algorithm that, given a token-definition consisting of a set of regular expressions $T$ and a string $w$ subdivides $w$ into a
list of token-string pairs \((t_1, w_1) \ldots (t_n, w_n)\) such that \(w_i \in T(t_i)\) for \(0 \leq i \leq n\). This subdivision is often called an interpretation of \(w\).

Ideally the language definition is unambiguous, so that there is always at most one possible interpretation of \(w\). Unfortunately in more cases than not the language definition is not, in which case a choice must be made on which interpretation is the desired one.

One common technique used in scanners to alleviate this problem is to repeatedly take the longest match in the string and return the resulting interpretation. This method is called the greedy algorithm. As long as care is taken when defining the language the scanner needs to recognize, the greedy algorithm is fast and adequate, but it is not perfect. This can be demonstrated by looking at the token-definitions \(T = \{A \rightarrow 'a', B \rightarrow 'aa', C \rightarrow 'ab'\}\) and the input string \(w = 'aab'\). If the greedy algorithm is used here, the scanner will recognize token (B, 'aa') before failing instead of recognizing the correct and terminating interpretation (A, 'a')(C, 'ab').

2.4 Scanner Generators

Due to vast differences in the ways in which scanner generators can be implemented, it is difficult to specify a single formal specification that encompasses all different types. Instead, I will briefly discuss the most common implementations here and how they work.

One approach to scanner-generation is to create code that is tailored to the language definition that needs to be recognized. These generally work by creating a direct mapping from the definition to code, possibly through an intermediary format that allows for optimizations. A simple version of this technique is the T-Recipe [tEik00], which performs a direct mapping from regular expressions to code.

A second approach is to use a fixed algorithm for the scanner that uses a data-set to determine the tokens. This data-set can then be generated based on the language definition that needs to be recognized. A common example of this technique is a DFA-based scanner-generator like LEX [Lesk75].

2.5 Example language

All scanners developed in this document will have their own advantages and disadvantages. To make it possible to compare the scanners on these points, a fixed example language can be used with each of them. For this purpose, I have designed the language definition for a simple calculator program. It will be able to perform basic actions on integer or real numbers, i.e. addition, subtraction, division and multiplication. Additionally it will have a number of predefined constants like \(\pi\) and \(E\), as well as the ability to assign intermediate results to identifiers and then use these again in later calculations. Identifiers are strings of characters and the predefined constants are a subset of these. This leads to the language definition shown below.
2.6 Borland Delphi Environment

Borland Delphi is a so-called rapid application development (RAD) environment. RAD environments provide the developer with tools to quickly construct complex graphical interfaces without him having to manage every little detail. The developer can concentrate on the functionality of his program while the RAD environment takes care of the standard functionality of the interface.

Initially Delphi gives the developer a blank form that forms the basis of the project. The developer can create the graphical user-interface of his program by placing components onto this form. A subset of these components, named controls, are visual elements of the applications interface, such as buttons, checkboxes and editboxes. The rest of the components are non-visual elements that give the developer access to commonly used actions. Examples for non-visual components are timers and database-access components.

All components have large numbers of settings, called properties, that can be configured from within the Delphi IDE. For controls, these properties can be things such as the components color, its contents or the method in which those contents should be displayed. For non-visual components, the properties can be the name of the database to connect to or the interval at which the timer works.

Most components can trigger a number of events. These events point to portions of the developers code that are executed whenever the event is triggered. Examples of possible event causes are mouse-clicks on buttons, text-changes in editboxes or failure to connect to a database. Events assist the developer in rapid application development because he only needs to write code for those events that he is interested in.

It is also possible for developers to extend the Delphi IDE with new controls and components that can interact with the system in the same way as the default sets of components. Additionally, the developer can interact directly with the Delphi IDE through a system called the OpenTools API. Creation of custom components and use of the OpenTools API are discussed in greater detail in chapter 8.
Chapter 3

Scanner Framework

In this chapter the basic structure of the scanner-framework is defined. First a representative set of use-cases is specified, which can then be analyzed to determine a logical division of the framework into separate parts with a specific functionality. By considering each of these parts and specifying the exact functionality they must provide, their public interface can be defined. Once this is done, it is possible to re-visit the use-cases to give a more in-detail explanation of the actions needed.

3.1 Use-cases

Before the use-cases can be discussed, it is essential to define some of the terms used in them. The followings words are used throughout this chapter to specify certain scanner types.

pre-compiled scanner A scanner that has been created during component design-time and as such is already compiled and available during application design-time.

generated scanner A scanner that is placed in a unit in the active project at application design-time and will become available at run-time.

scanner-enabled component A component that uses a scanner from the framework to perform part of its functionality.

Most of the use-cases follow directly from a pair of factors. The first factor is the way the user wishes to use the scanner. He can do this either directly or through a component designed to work with these scanners. The second factor is the source of the scanner. This scanner can be either pre-compiled, generated or supplied by the user. This gives us six possible use-cases, one of which is trivial. The case where the user wants to use his own scanner directly will not be discussed any further.

A second source of use-cases is when the user wants to perform run-time generation of scanners. A possible use for this would be to create a stand-alone version of the scanner-generators (instead of the project-bound method). This situation only creates one additional use-case though, as neither the use of the scanner or where it came from have any meaning for this case.
The six use-cases are then:

- The user wants to use a pre-compiled scanner directly.
- The user wants to use a generated scanner directly.
- The user wants to use a pre-compiled scanner with a scanner-enabled component.
- The user wants to use a generated scanner with a scanner-enabled component.
- The user wants to use a handwritten scanner with a scanner-enabled component.
- The user wants to generate the source-code for a scanner during application runtime.

3.2 Analysis

Based on the use-cases from 3.1, three types of scanners can exist within the framework: pre-compiled scanners, generated scanners and handwritten scanners. Of these three, the first two use a language-definition to determine which tokens the scanner should recognize.

In the case of the pre-compiled scanner it uses the definition to change its internal state to recognize the correct tokens. In the case of the generated scanner, the code is created in such a way that the tokens are recognized according to the language definition. There are cases imaginable where both a pre-compiled and a generated scanner may use the same language definition.

Also, it might be desirable to use the same language definition for multiple scanners. This facilitates the interchangeability, allowing users to swap from one scanner to another without having to redefine the language.

These two points give reason to a separation of the scanner and the language definition into different classes. This enables multiple scanner-types to use the same definition while at the same time making it so that the type of scanner can be changed easily with little effort, provided that the language-definition is compatible with the scanners in question.

Scanner-enabled components are components that one can place on a form in Delphi that use a scanner to perform some of their functionality. For this they need a link to the scanner that they should use. The pre-compiled scanner is available at application design-time and, if made as a Delphi component itself, could be linked directly to the scanner-enabled component through use of a property. However, both the generated and the handwritten scanner are not available at application design-time and as such cannot be linked directly to the scanner-enabled component.

To solve this problem it is necessary to abstract from the source of the scanner. This can be done by creating an intermediary component that is always available during design-time by using the factory design pattern [Gamm95]. This intermediary component can then return an instantiation of a scanner whenever it is needed. How it produces this instantiation becomes the responsibility of the framework and not of the, possibly user-created, scanner-enabled component.
Scanner-enabled components have only limited functionality if they have no information about the language that the scanner recognizes. One can think about the number of token-types it may return along with the name of each of those token-types. This information can help create better and easier-to-use scanner-enabled components.

All this information can be extracted from the language definition and therefore it seems only logical that the functionality to request such information is part of the language definition class as well.

This creates a minor problem in the case of the handwritten scanner. Since it has no language definition within the framework, the information about the scanner has to be provided by the author of the handwritten scanner instead.

Though it is possible to give scanner-enabled components a direct link to a language definition so that it can extract language data, this will cause a duplication of data as the scanner or scanner-generator already have a link to this definition as well. To avoid this duplication, the factory can take over the duty of handling the scanner definition, providing it to scanners it creates, providing it to scanner-generators it uses to generate scanners and providing it to any scanner-enabled components using that factory.

To perform run-time generation of scanners, all scanner-generators must be available at run-time. Additionally the definitions for all generated classes must be available and editable during run-time. During design-time validation of the input can be performed by specialised dialogs, while during run-time the definition would have to be changed through the use of properties. These properties are inevitably more complex and ensuring the validation of these properties is a non-trivial task as well. Therefore the run-time generation capabilities of scanner will be handled on a scanner-by-scanner basis instead. If it makes sense for the scanner-type then it can be added, but the scanner-framework does not supply extra functionality to support it.

3.3 Structure

Based on the observations in the analysis from section 3.2 the basic class-structure for the scanner framework can then be given by figure 3.1.

The scanner base class is handled in section 3.4, the scanner definition base class in section 3.5 and the factory base class in section 3.6.

3.4 Scanner Base Class

The most important class in this structure is the scanner base class. It needs to define the basic interface that all scanners in the framework must implement. Thus it is important to keep this interface general, but still with sufficient functionality to facilitate easy scanner development in derivative classes.

3.4.1 Actions

The basic functionality of the scanner is to divide a string into a series of tokens. It does this through a single abstract method that can be called repeatedly to move on to the next token. The method is abstract, because the exact functionality contained in the method is different for each type of scanner.
procedure NextSym;

3.4.2 Input: Text

An essential part of a scanner is the text it needs to operate on. Needed is a set of functions that allows the scanner to work with this text. The three most important actions are a method to retrieve characters from the text, a method to determine whether the end is reached, and a method to restart at the beginning. Additional methods can allow moving backward and forward in the text by a specified number of characters and can allow marking positions to which the scanner can return at a later time.

Since not much is known about the source and the size of the text to be scanned, care must be taken when choosing the method of implementing this property in the scanner base class. The \texttt{string} type seems appropriate, since the user can place the text in there from any source and it can be up to 2Gb in size. This upper limit in size is deceptive however, as strings are more realistically limited by the memory of the machine on which the scanner is running.

An alternative to the \texttt{string} type is to use Delphi’s \texttt{TStream} type. It supports data from a variety of sources, among which strings, memory, files and even win-sockets. If necessary, other data-sources can also be supported by creating a \texttt{TStream} interface for them. Since the data can now be stored on other mediums until needed, the memory-size limit on the data is no longer applicable. A counter-point against streams is that some of the functionality mentioned above is not directly supported by some streams, in particular the functions to move back and forward in the input. Since scanners generally perform these actions only across small distances, these limitations can however be removed by buffering the input.

Based on these considerations streams appear the best solution for the input of the scanner. This because they offer direct support for the use of files as input.
Additionally, strings can still be used easily through the **TStringStream** class, which provides a TStream-wrapper around the original string.

This makes the input related methods of the scanner base class the following:

```pascal
property Source: TStream;
procedure Reset;
function IsFinished;
procedure NextChar;
procedure PrevChar;
procedure SkipChars(count: integer);
procedure RevertChars(count: integer);
procedure Mark;
procedure Recall;
```

### 3.4.3 Output: Tokens

When **NextSym** recognizes the next token, information about this token needs to be returned to the caller. Since the caller cannot access the stream directly, the first thing it should return is the exact text of the token. Additionally the position of the token in the stream can be given as well, along with a value of type **TTokenType** corresponding to the type of the token is returned.

Although the best type for **TTokenType** would be an enumeration, this is not possible since the pre-compiled scanners would have to have access to the type-declaration at component design-time. Since the user specifies the language at component run-time, another type is needed. Therefore the type chosen for **TTokenType** is the integer.

```pascal
property SymType: TTokenType
property SymText: string
property SymPosition: integer
```

In the case of the number 3.14 at position 500 in the text, **SymType** would contain the token type corresponding to the number token, **SymText** would contain '3.14', **SymPosition** would contain 500.

### 3.4.4 Output: Errors

When an error occurs during scanning, the user needs to be notified of this, preferably in combination with information on what kind of error and where in the input text it occurred. For this purpose, an Exception class is made called **EScanError**. This exception will have the fields **Position** to indicate the position in the input-stream at which the error occurred. Any scanners derived from the generic scanner interface can sub-class the **EScanError** and thus provide more detailed error-messages.

### 3.5 Definition Base Class

Though the language definition is usually a very specialized class, the ability to return meta-information about the language defined is possible in all definitions. The meta-information it can provide is the number of tokens, their names and their types.
function GetNumberOfTokens: integer
function GetTokenName(idx: integer): string
function GetTokenType(idx: integer): TTokenType

3.6 Factory Base Class

Although the factory base class is the connecting point of most components in the scanner framework, its interface is surprisingly simple. All it contains are a method to create an instantiation of a scanner and a property for the scanner definition.

function CreateScanner: TSFWScanner
property Definition: TSFWDefinition

3.7 Use-cases revisited

Now that the structure of the scanner framework is defined, it is possible to take a look back at the use-cases in section 3.1 and fill out the actual steps needed to perform the actions. The generation of a scanner into a string during run-time is no longer included in this section as the framework provides no special functionality for it, based on the decision in section 3.2.

Direct use of a pre-compiled scanner.

Application Design-time
- the user places a language-definition component
- the user places a factory for the wanted scanner-type
- the user links the factory with the language-definition

Run-time
- the user requests a scanner from the factory
- the factory creates a scanner and links it with the scanner-definition

Direct use of a generated scanner.

Application Design-time
- the user places a language-definition component
- the user places a factory of the wanted scanner-type
- the user links the factory with the language-definition
- the factory creates a scanner-generator
- the factory generates the scanner into a unit
- the factory generates an event to instantiate the scanner

Run-time
- the user requests a scanner from the factory
- the factory calls the generated event
- the event instantiates the generated scanner and returns it
  (alternative: the user creates an instance of the generated class directly)

Use of a pre-compiled scanner with a scanner-enabled component.

Application Design-time
the user places a language-definition component
the user places a factory of the wanted scanner-type
the user links the factory with the language-definition
the user places the scanner-enabled component
the user links the scanner-enabled component with the factory

Run-time
the scanner-enabled component requests a scanner from the factory
the factory creates a scanner and links the definition with it

Use of a generated scanner with a scanner-enabled component.
Application Design-time
the user places a language-definition component
the user places a factory of the wanted scanner-type
the user links the factory with the language-definition
the factory creates a scanner-generator
the factory generates the scanner into a unit
the factory generates an event to instantiate the scanner
the user places the scanner-enabled component
the user links the scanner-enabled component with the factory

Run-time
the scanner-enabled component requests a scanner from the factory
the scanner-factory calls the generated event
the event instantiates the generated scanner and returns it

Use of the user’s scanner with a scanner-enabled component.
Application Design-time
the user implements his own scanner based on the basic scanner interface
the user places the external factory
the user places the external language-definition
the user adds an event to the factory that instantiates scanner
the user places the scanner-enabled component
the user links the scanner-enabled component with the factory

Run-time
the scanner-enabled component requests a scanner from the factory
the factory calls the user-supplied event
the event instantiates the generated scanner and returns it

3.8 Framework Structure
Based on the scanner-framework structure from 3.3 a set of scanners can be chosen that best encompass the full design of the framework.

3.8.1 Scanner Types
With the overall structure of the scanner framework now known, the time has come to decide on the initial set of scanners for the framework. To put the design of the scanner framework to full use, one scanner from each category is
needed, i.e. a suitable pre-compiled scanner and a suitable generated scanner. Additionally a third kind of scanner could be of interest, namely one that can be implemented both as a pre-compiled and as a generated scanner.

For a purely pre-compiled scanner it is easiest to have one that performs a fixed set of actions that can be influenced through the use of parameters. The scanner chosen for this is the paradigm scanner. The paradigm scanner uses some simple rules to recognize the tokens for most common programming languages out there and is discussed in greater detail in chapters 4 and 5.

The T-Recipe, as discussed in [Eik00], is a prime example of an algorithm that can be used only in purely generated scanners. This is because the T-Recipe directly maps a regular expressions to code through the use of a set of rules. These rules for regular expressions are explained and expanded upon in chapter 6.

A very well-known class of scanners are the DFA-based scanner, which translate a set of regular expressions into an equivalent DFA. This DFA can then be used to by a scanner to recognize tokens by traversing the DFA through a simple algorithm, called a driver. The DFA is commonly stored in a table, which can be used by pre-compiled drivers and generated drivers alike. This scanner-type is further explored in chapter 7.

The final scanner-types the framework will handle are the ones supplied by the user. The scanner framework has to provide the functionality needed to allow the user to integrate his scanners into the framework, making it possible for them to interact with scanner-enabled components.

### 3.8.2 Language Definitions

Since the types of scanners described in the previous section are vastly different, each of them will require their own specific language definition as well. The different language definitions can be made to work together to a certain degree though.

Both the T-Recipe scanners and the automaton-scanners use regular expressions as the input. As such it stands to reason to give these two language definitions a single base-class. This will facilitate interchangeability, as the definition for one might be useable by the other as well.

Although the paradigm scanner does not use regular expressions as input, it is possible to choose the settings in such a way that a set of regular expressions can be made based on them. As such, it is possible to make the language definition for the paradigm scanner a derivative of the regular language definition base-class, though a highly restricted one.

The handwritten scanner doesn’t require a language definition in itself, but scanner-enabled components will need a list of all the tokens it may return. For this purpose a dummy definition component gives the user the ability to specify which tokens his scanner returns and what types are associated with those tokens.

### 3.8.3 Factories

There are essentially two types of factories. Those for pre-compiled scanners and those for generated scanners. The basic functionality of either type of factory is the same, i.e. they take a language definition and produce a scanner.
that complies to this definition, but the internal mechanics of both are vastly different.

The factory for the pre-compiled scanners only needs to produce an instance of the scanner, which is easy since the scanner is already available at application design-time. It then just needs to link this scanner with the language-definition and its work is done.

In the case of the generated scanners, things are more complex. This factory has to create a unit in the current project and generate a scanner into it based on its scanner definition. It then has to add the code to create an instantiation of this scanner into an event-handler of itself. The actions required for this are described in section 8.3.

The factory for the handwritten component can be kept simple. All it needs to do is provide the user with a way to specify an external definition a way to instantiate his scanner, which can be done in much the same way as the generated scanner.

3.8.4 Generators

In the case of the T-Recipe scanner, a generator is needed to construct the actual scanner class that will be added to the project. This generator is very specific and as such does not use a base-class like the scanners and factories.

A similar generator can be created for the automaton-scanner where it generates a scanner class based on a language definition. However, since the DFA scanner is added to the framework as a precompiled scanner and a generated scanner, a separation of the table-generation and the scanner-generation is needed. Once the table is generated, it can then be used by the pre-compiled scanner or the scanner-generator to create the code.

3.8.5 Class Diagram

Combining the observations in the previous sections with the structure of the scanner framework leads to the following list of classes to be implemented.
Figure 3.2: Class-structure of the Scanner Framework
Chapter 4

Language Definitions

In section 3.8.2 the need to have two different methods to edit language definitions was shown, as well as a meta-language definition for the external scanners. Each of the three definition types is dealt with in this chapter.

Starting with a more detailed description of the ways to represent regular expressions in section 4.1, the public interface, textual representation and the graphical user-interface of the regular language definition is then explained in section 4.2.

In section 4.3 an analysis is made of existing programming languages and their common elements. Based on this analysis, a set of parameters for the paradigm scanner is chosen and a public interface and graphical user interface are designed based on these parameters.

Section 4.4 details the requirements of the external definition and documents the graphical user interface required to complete these requirements.

4.1 Regular Expressions

In section 2.1 and section 2.2 the regular expressions were introduced. In this section, the internal representation of these regular expressions is discussed, as well as a textual representation that is more conducive to storage and editing. Both representations are subsequently used for the regular language definition in section 4.2.

4.1.1 Internal representation

Regular expressions can be represented internally through trees of objects corresponding with the elements of the regular expression. The nodes in this tree correspond to the operators and terminals in the regular expression. The set of available node-classes is depicted in figure 4.1.

There are numerous algorithms that can perform actions on object-trees. Most of these algorithms take advantage of the tree-structure, defining a specific path that the computations will follow. Examples of tree-traversal algorithms are the treewalker and the visitor design-pattern [Gamm95].
**TNode**

The **TNode** class provides some generic functionality for nodes of object-trees. It gives access to the parent of the tree-node through the `Parent` property and access to its children through `ChildrenCount` and `Children`.

```plaintext
property Parent : TNode;
property ChildrenCount : integer;
property Children[idx : integer] : TNode;
```

**TRENode**

The base-node for all regular expression node **TRENode** is a direct descendant of **TNode** without any additional functionality. It exists so that type-checking can be performed on nodes to ensure that they are really regular expression nodes.

**Terminals**

The terminal nodes have no additional properties in common, but each separate node needs to store information unique to that terminal, with the exception of the epsilon-node. The character node gets a property `Character` of type `char` to hold which character it represents. The string gets a property `Text` of type `string` to hold the string it represents. The range gets two properties `StartChar` and `EndChar` of type `char` to hold the first character, respectively the last character of the range. Finally the identifier-node gets a property `Identifier` of type `string` to hold its name.
Unary Nodes

All unary nodes have a property named `Child` of type `TRENode` which provides easy access to the argument of the operator. This value is the same as the single child accessible through `Children` but does not need any additional typecasting.

```plaintext
property Child : TRENode;
```

None of the unary nodes require any additional information stored that is unique to that specific type of node. The set of unary nodes are `TREStar`, `TREPlus`, and `TREOption` for the repetition, once-or-more-repetition and the option respectively.

Binary Nodes

All binary nodes get an additional two properties named `Left` and `Right` of type `TRENode`. These properties hold, respectively, the left and the right argument of the operator in question. Though these arguments are accessible through the `Children` property as well, they provide easier access without a need to typecast.

```plaintext
property Left : TRENode;
property Right : TRENode;
```

The two binary operators are the selection and the concatenation, their nodes dubbed `TREStick` and `TREDot` respectively. Neither of them requires any additional information unique to their nodes.

4.1.2 Textual representation

A textual representation of regular expressions is needed for storage purposes, as well as plaintext editing-methods for users.

Single characters are placed between single or double quotes, so ‘a’ and ’a’ both denote the character a. Should the character be a special character like line-feed or carriage-return, it can be denoted as `#num` as well, where `num` is the decimal ASCII-value of the character in question. Strings can be placed between quotes as well, the string `"text"` denoting the regular expression `t · e · x · t`. A range of characters is denoted as `[a-z]` with a and z the first, respective the last character of the range. Identifier-labels are represented as quote-less strings of characters and numbers.

The operators for regular expressions are expressed in much the same way as they are represented on paper, with . being concatenation, | being selection, * being repetition, + being once-or-more repetition and ? being the option. Finally ε is denoted as &. Braces can be used throughout the regular expression to enclose sub-expressions and to enforce proper evaluation of the sub-expressions.

Summarized, this results in the following representations of the various parts of the regular expressions:
\[\epsilon \quad \rightarrow \quad k\]
\[\alpha \quad \rightarrow \quad \alpha' \quad (\alpha \in V)\]
\[[\alpha - \beta] \quad \rightarrow \quad [\alpha' - \beta'] \quad (\alpha, \beta \in V)\]
\[\sigma \quad \rightarrow \quad \sigma' \quad (\sigma \in V^*)\]
\[e_1|e_2 \quad \rightarrow \quad (e_1|e_2)\]
\[e_1 \cdot e_2 \quad \rightarrow \quad e_1 \cdot e_2\]
\[e^* \quad \rightarrow \quad (e)^*\]
\[e^+ \quad \rightarrow \quad (e)^+\]
\[e? \quad \rightarrow \quad (e)?\]

4.2 Regular Language Definition

As both the T-Recipe scanner and the automaton scanner use regular expressions as input, it stands to reason to create a common base-class for the language definitions of both. In this chapter a regular language definition base class is designed, along with methods to enforce the correctness of the definition and those of any derivative classes. Finally a user interface is developed that can be used to easily edit the language definition.

4.2.1 Formal Definition

A regular language definition D is a 4-tuple \((V, N, R, T)\) with:

- \(V\) a set of terminal characters
- \(N\) a list of non-terminals \(\{A_1, \ldots, A_k\}\)
- \(T\) a list of tokens \((T \subseteq N)\)
- \(R\) a mapping from \(N \rightarrow RE(V \cup N)\)

The set of tokens \(T\) specifies those non-terminals that need to be recognized by lexical scanners using the language definition. This set is taken as a subset of all non-terminals to allow for auxiliary expressions that allow for easier language design.

Several restrictions apply to the expressions in \(R\). Firstly, each expression can only refer to non-terminals that have already been defined, i.e.

\[
(\forall i : 1 \leq i \leq k : R(A_i) \in RE(V \cup \{A_1 \ldots A_i - 1\}))
\]

Secondly, no expressions are allowed to refer to tokens, i.e.

\[
(\forall i : 1 \leq i \leq k : R(A_i) \in RE(V \cup N - T))
\]

Thirdly, tokens are not allowed to be able to produce the empty string, i.e.

\[
(\forall t : t \in T : \neg null(R(t)))
\]

The language defined by a language definition \(D\) is

\[L(D) = L((t : t \in T : R(t)))\]

Given the formal definition it is then possible to rewrite the example language from section 2.5 to an equivalent one for the regular definition:
4.2.2 Textual representation

The textual representation of a regular language definition is straightforward. The only parts that need to be stored are the lists of name/expression pairs for the auxiliary expressions and the tokens. It is imaginable however that in derivative classes more information needs to be stored and restored, so the format should be suitable for expansion.

The format I have chosen is the same format used in common .ini files. This format consists of a list of name/value pairs of the form name=value. These name/value pairs can be separated into sections by placing a tag in front of them, for example like [tag].

The grammar describing .ini files is as follows:

\[
\text{DOCUMENT} \rightarrow \text{GROUP}\n\]
\[
\text{GROUP} \rightarrow \text{GROUPLABEL} \cdot \text{PAIRS}\n\]
\[
\text{GROUPLABEL} \rightarrow ['] \cdot \text{NAME} \cdot [']
\]
\[
\text{PAIRS} \rightarrow \text{NAME} '=' \text{VALUE}
\]

In the case of the regular language definition, only two groups are needed, namely the auxiliary expressions and the tokens. Each name/value in these groups consists of the name of the expression or token, followed by the regular expressions that corresponds with it. Thus, the following two rules can be added to the above grammar, where the regular expression corresponds to the textual representation from section 4.1.2:

\[
\text{NAME} \rightarrow (['a'-'z'] | ['A'-'Z'] | ['0'-'9'])^*
\]
\[
\text{VALUE} \rightarrow \text{textual regular expression}
\]

Representing the example language from 2.5 in this manner leads to the following textual representation:

\[
\text{[exprdefinitions]}
\]
\[
\text{digit} = ['0'..'9']
\]
\[
\text{[tokendefinitions]}
\]
\[
\text{identifier} = ([a-z] | [A-Z])^*
\]
\[
\text{predefined} = 'pi' | 'E'
\]
\[
\text{operator} = '=' | '+' | '-' | '/' | '*' | '(' | ')'
\]
\[
\text{number} = \text{digit}^+ \cdot (\epsilon | '\'.\'. \text{digit}^+)
\]
An additional advantage of this format is that it allows derivative classes to add more information under different group tags without affecting the stored regular language definition itself. This allows the storage of multiple different representations of the same data, where one representation might be used by one type of scanner or scanner-generator, while another representation is used by a different type.

4.2.3 Class Interface

The definition of the regular language definition class follows almost directly out of the formal definition. The regular language definition class should contain a list of expressions on which information can be requested. Adding and removing expressions should be possible and, since the order of the expressions is important, it must be possible to move expressions up and down in the list. The public interface for the expressions is then:

- function GetNumberOfExpressions: integer;
- function GetExpressionNode(idx: integer): TRENode;
- procedure SetExpressionNode(idx: integer; aNode: TRENode);
- function GetExpressionName(idx: integer): string;
- procedure SetExpressionName(idx: integer; aName: string);
- function GetExpressionIndex(aName: string): integer;
- procedure ExchangeExpressions(src, dst: integer);
- procedure AddExpression(aName: string; aExpr: TRENode);
- procedure DeleteExpression(idx: integer);
- procedure ClearExpressions;

The number of tokens and their names and types are accessible through the functions inherited from the language definition base class. This functionality can be completed by adding methods that allow the user to add and remove tokens, edit their names, types and expressions as well as re-order them.

- function GetNumberOfTokens: integer;
- function GetTokenNode(idx: integer): TRENode;
- procedure SetTokenNode(idx: integer; aNode: TRENode);
- function GetTokenName(idx: integer): string;
- procedure SetTokenName(idx: integer; aName: string);
- function GetTokenType(idx: integer): TTokenType;
- procedure SetTokenType(idx: integer; aType: TTokenType);
- function GetTokenIndex(aName: string): integer;
- procedure ExchangeTokens(src, dst: integer);
- procedure AddToken(aName: string; aExpr: TRENode);
- procedure DeleteToken(idx: integer);
- procedure ClearTokens;

Instead of enforcing the correctness of the language definition at all times, the correctness check is placed in a method that can be called at the times that correctness is required. This alleviates problems during the construction of the language, at which point momentary inconsistencies in the language might be required for the ease of use. It also makes it easier to change the restrictions in derived classes by overriding the validation method.
If the language definition turns out to be incorrect, then a list of errors is returned detailing at which points it failed the validation. Each error is accompanied by the name of the expression and the exact node of the regular expression in which it occurred. This leads to the addition of the following methods:

```plaintext
procedure Validate;
function GetNumberOfErrors: integer;
function GetError(idx: integer): string;
function GetErrorExpression(idx: integer): string;
function GetErrorNode(idx: integer): TRENode;
procedure AddError(msg, expr: string; node: TRENode);
procedure RemoveError(idx: integer);
procedure ClearErrors;
```

The exact validation requirements are discussed in more detail in section 4.2.4.

### 4.2.4 Validation

For the basic regular language definition, the following restrictions should hold:

1. All identifiers should be declared earlier in the definition
2. No token-labels may be used as identifiers
3. Token-definitions may not produce the empty string

Checking the identifiers is a matter of traversing the expressions and tokens in the language definition. If an identifier is encountered, a check is made whether there is an expression of the same name defined earlier in the list of expressions. If not, then validation will fail and the user will be informed. Should it not exist as an expression, then a check against the list of tokens can be made in the effort to provide a more accurate error message to the user.

To make sure the scanner always makes progress, tokens may not be able to produce the empty string. If there were a token able to produce the empty string, then the scanner could continuously recognize this token without moving ahead in the source, which is obviously not a desirable situation.

### 4.2.5 Graphical User Interface

In order to modify the regular language definition, a suitable GUI is needed. This GUI must contain the following elements and abilities:

- A display for the definition
  - the ability to add tokens/expressions
  - the ability to remove tokens/expressions
  - the ability to re-order tokens/expressions
  - the ability to edit tokens/expressions
  - the ability to load/save the definition
• An editor for expressions/tokens
  – the ability to enter a regular expression
  – visual feedback of the entered regular expression
  – the ability to change an expression’s or token’s name
  – the ability to change a token’s type

• A display for feedback in case of errors
  – the ability to see where the error occurred

The GUI displayed in picture 4.2 is constructed based on these requirements. Tokens and expressions are displayed on the left, with options to add, remove, and re-order tokens and expressions. Additionally the option to save or load a definition is displayed there. Selecting a token or expression in this list displays it in the expression/token editor, allowing it to be edited.

While the expression is being edited in the editbox, one of two things can happen. Either the expression can not be parsed successfully, in which case an error-message detailing why it has failed is displayed in the status-window below, or the expression can be parsed successfully, in which case a graph-representation of the regular expression appears in the display above. More about this graph-representation in a moment.

If an expression successfully parses, validation is performed on the full language definition. Any validation-errors found during this time are displayed in the status-window. Selecting any of these messages displays the token or expression where validation failed in the editor above.

A popular way of visualizing regular expressions is by displaying a graph of the NFA that can be constructed out of this regular expression. The algorithm for transforming a regular expression to an NFA is straightforward, mapping elements of regular expressions to structures in the NFA, as such displaying
an accurate representation of the regular expression. However, this technique, used in [Kahr99] and [Krus01], places a strong emphasis on how the program will proceed to use the regular expression.

The regular definition in the scanner framework is used by both the T-recipe and the automaton scanners. Although the latter transforms the regular expressions to finite state machines, the former does not. As such, the visualization-technique described above is far from ideal, suggesting that the framework always uses finite state machines even though it doesn’t.

Instead, the graph-representation in this implementation has been chosen to accentuate the structure of the regular expression. It does this by focusing on the terminals of a regular expression. These include the epsilon, the characters, the character-sets, the strings and, more or less the identifiers. Instead of displaying these elements next to transitions between states, they are made the main elements by displaying them as boxes containing the specified values.

The operators in the regular expression are now displayed as edges that define the flow through the terminals. The selection is shown by a vertical branching in the flow, indicating that either the upper or the lower flow may be taken at that point. The concatenation is represented by horizontal placement of two expressions, connected by a single edge. The once-or-more repetition is displayed as an expression with an edge looping from the end to the beginning, while the option is displayed as an expression with an edge running from beginning to end. The repetition is a combination of these two, allowing the user to bypass an expression or loop back to the beginning.

4.3 Paradigm Definition

A paradigm scanner is built on the assumption that all programming languages are constructed out of a similar set of lexical elements and that these individual elements are similar in structure between the languages. Based on this observation, the paradigm scanner offers an easy way to create a scanner by giving the user a fixed set of token-types and easily configurable settings to define the structure of these types.

This section will cover the development of the definition for a paradigm scanner, starting with an analysis of how common languages are constructed in order to find out what elements should be included and how these are structured. These elements are then captured in a more formal declaration, where the exact parameters for the token-types are given as well. Based on these parameters an appropriate graphical interface is then defined.

4.3.1 Analysis of lexical elements

Through the use of some existing programming languages, it is possible to determine the elements that the paradigm scanner must be able to handle and how these elements are most commonly constructed. Three of these existing languages are Pascal [ISO7158], C [ISO9899] and Haskell [Peyt98]. Examples of these languages can be found in listings 4.1, 4.2 and 4.3 respectively.
Listing 4.1: Pascal Sample

function countC ( str : string ) : integer ;
{ ret: number of C's in string str }
var i , cnt : integer ;
begin
  cnt := 0 ;
  for i := 1 to length ( str ) do
    if ( str [ i ] = 'C' ) then cnt := cnt + 1 ;
  result := cnt ;
end

Listing 4.2: C Sample

/* returns: number of C’s in string str */
int countC ( string str ) {
  int i , cnt ; // cnt = number of C’s so far
  cnt = 0 ;
  for ( i = 0 ; i < strlen ( str ) ; i ++ ) {
    if ( str [ i ] == 'C' ) then cnt += 1 ;
  } return cnt ;
}

Listing 4.3: Haskell Sample

countC :: String -> Int
countC [] = 0
countC x : xs = if ( x == "C" ) then 1 + countC xs else countC xs

By studying these examples, the required lexical elements for the paradigm scanner can be determined easily. The examples above are all constructed completely out of the following seven elements:

**Identifiers** Variable- and function-names.
Ex: countC, i, cnt, xs

**Reserved words** Words that have a special meaning in the language.
Ex: if, then, else, for, return

**Symbols** Short sets of characters with a special meaning.
Ex: :=, =, ++

**Numbers** Lists of digits that represent a numerical value.
Ex: 0, 1

**Strings** Lists of characters that represent a literal text.
Ex: ‘C’, “C”
Comments  Extra information embedded into the program-text.
   Ex: { ... }, /* ... */ , // ...

Whitespace  The characters that divide the elements listed above.
   Ex: spaces, tabs, carriage-return and line-feed

   The exact structure of each of these elements is discussed in greater detail in the following section.

4.3.2 Structure of lexical elements

Now that it is clear what elements are needed for the paradigm scanner, a closer study of each of them is needed to determine the shapes in which they come. For each of the seven elements the most common forms are discussed and a list of possible properties is given that will encompass most or all of these forms.

Identifiers

Identifiers exist in practically every programming language, among other things serving as variable-, procedure- and function-names. However, in almost all of these languages, identifiers have a similar structure that can used for the paradigm scanner.

Almost all identifiers are constructed by one value from a set of allowed first characters, followed by zero or more values from a second set of follow-up characters. These sets are often constructed out of a combination of letters, digits and some special symbols like the underscore. Many languages also place a limit on the maximum length of the identifier.

One might think that case-sensitivity should also be part of the identifier definition, but whether foo and Foo denote the same variable is really for later stages to decide where the meaning of the identifiers comes into play. The only case-sensitivity that is of importance to the scanner is whether identifiers should be lowercase, uppercase or whether they can be a mix of both.

Based on these observations, the properties needed to define the structure of identifiers in the paradigm scanner definition are:

- the existence of identifiers
- the set of initial characters
  chosen as a combination of a–z, A–Z, _
- the set of follow-up characters
  chosen as a combination of a–z, A–Z, 0–9, _
- the maximum length (if any)

Reserved Words

Many languages have a small set of reserved words that should not be interpreted as identifiers. These reserved words are commonly a subset of the identifiers that have a special meaning attached to them, often because they are statements in the programming-language. Examples of reserved words for the Pascal language are: if, case and then.
For the paradigm scanner, the reserved words are defined as a small subset of all possible identifiers. Unlike with the identifiers, case-sensitivity plays a role for the reserved words, meaning that, if the reserved words are case-sensitive and `foo` is a reserved word, but `Foo` is not, then `Foo` would be seen as an identifier.

The properties for the reserved words are:

- the list of reserved words
  chosen as a subset of all possible identifiers
- the case-sensitivity

Symbols

Symbols serve a similar function as the reserved words in that they are strings of characters that have a special meaning in the language. One can think here of statement-separators, like `;`, brackets, like `[ ]`, and operators, like `+`, `-`, and `*`.

In the paradigm scanner, the only property needed for symbols is the following one:

- the list of symbols

Numbers

Numbers come in four different shapes: integers, real values, exponential values and numbers in different bases. Any language supporting numbers supports the integral value, though many will support the real value and exponential values as well. Numbers in different bases are less common and are usually limited to binary, octal and hexadecimal values.

Though the exact value ranges and types may vary from language to language, the notation of these numbers is the same in almost all of them. Integers are strings of digits, like `0`, `19`, and `481`. Real values are two strings of digits separated by a decimal point, like `0.10`, `3.14`, and `12.312`. Exponentials are either integers or reals followed by `E` or `e`, an optional sign `+` or `-` and an integer. Examples of exponentials are `1E+10`, `0.33E1`, and `2.73e-4`.

When different bases are involved, things become a bit more complex. Every language uses the same sets of characters, i.e. `0`–`1` for binary, `0`–`7` for octal and `0`–`9A`–`F` for hexadecimal. However, not all languages use the same prefixes. One very uniform way of handling the prefixes is to use `2r`, `8r`, and `16r`, i.e. the radix followed by an `r`. Another variant, used in C++, is a `0b` prefix for binary, a `0` prefix for octal and a `0x` prefix for hexadecimal numbers.

Based on the observations above, the properties needed for the numbers in the paradigm scanner definition are then as follows:

- the existence of integer, real, and exponential values
- the existence of binary numbers
- the prefix(es) of binary numbers
  chosen as `2r` or `0b`
- the existence of octal numbers
Strings

Strings are used to represent literal strings of characters in programming languages. A pair of opening- and closing-delimiters is used to indicate the beginning and the end of the string. Since these delimiters might need to appear in the string itself, most languages have a special construct to allow this. Two common methods are an escape symbol to mark that the next character is not a delimiter or conversion of a repeated delimiter to a single character.

The most common pairs of opening- and closing-delimiters are " and ", used in most common languages except for a few more exotic cases. Most languages use either one or the other, though there are cases where both are used as long as the opening- and closing-delimiter of each string are the same. In almost all languages where escape sequences exist, the backslash symbol is used as the escape character.

The properties that best define strings for the paradigm scanner are then as follows:

- the existence of strings
- the opening- and closing-delimiters
  chosen as " and/or ",
- the existence of escape-sequences
- the escape character
- the existence of the double-delimiter escape method

Comments

Comments can be used to place additional information in the program-text without it affecting the actual program-flow. Two types of comments exist in languages, namely the end-of-line comment, which runs to the end of the line and the delimited comment that runs from an opening-delimiter to a closing-delimiter.

Most comments can be classified as either C-style comments or Pascal-style comments. C can handle both end-of-line comments and delimited comments, the former with the // delimiter, the latter with the /* and */ delimiters. Pascal has two more varieties for delimited comments, namely the { ... } and ( ... ) delimiters. In cases like ini files, only end-of-line comments are used with ; and % as the opening-delimiters.

The properties for comments are as follows:

- the existence of comments
• the single-line comment-delimiters
  chosen from //, ;, %, --

• the multi-line opening- and closing-delimiters
  chosen from /\*...*/, {...}, (\*...*)

Whitespace

Whitespace ensures the separation of the tokens and allows for a more orderly structure to any documents written in the programming language. In most languages whitespace consists of spaces, tabs, carriage return and linefeed characters.

The only property needed for whitespace in the paradigm scanner is the following:

• the whitespace characters
  chosen from cr, lf, tab, space

4.3.3 Formalization of lexical elements

With the knowledge of what elements there are and how they are constructed, it is possible to write down a more formal representation of them. This will help during the implementation phase as it is relatively easy to transform regular expressions to scanner-code.

Before the formalization of the elements can start, there is one thing that needs to be defined first and that is the following:

\[ V : \text{the alphabet on which the scanner will function.} \]

Identifiers/Reserved Words

For the formalization it is easiest to handle the identifiers and reserved words at the same time as the two depend on each other. The reserved words need the structure of the identifiers and the identifiers have the reserved words as exceptions.

First a set is needed that contains all the potential identifiers and reserved words. This set can be constructed by all combinations of one character from the initial character set \( S \) and zero or more characters from the follow-up character set \( T \).

\[ S \subseteq V \]: the set of initial characters
\[ T \subseteq V \]: the set of follow-up characters
\[ PreId = S \cdot T^* \]: the set of pre-identifiers

If a maximum length is applicable on the identifiers, then an extra value \( M \) is needed to denote the length along with a restriction on the \( PreId \) set that must now hold.

\[ M \]: the maximum length of identifiers
\[ (\forall s : s \in PreId : |s| \leq M) \]

Since the final set of identifiers depends on the reserved words, the reserved words have to be dealt with first. The reserved words consist of two parts,
namely the actual list of reserved words and an equivalence function that is used to compare strings. This function is used to deal with the case-sensitivity of the reserved words list.

\[
RW \subseteq PreId \quad : \text{the list of reserved words}
\]

\[
eq_V : V \times V \rightarrow 2^B \quad : \text{the equivalence function on characters}
\]

\[
eq^* : V^* \times V^* \rightarrow 2^B \quad : \text{the induced equivalence function on strings}
\]

Given those definitions it is then possible to give the lists of keywords and identifiers. The keywords are all possible words that are equivalent to an entry in the reserved words list using the equivalence function. The identifiers are all possible identifiers, given in \(PreId\), minus any identifiers that are reserved words.

\[
\text{Keywords} = \{ w | s \in RW \land \eq(w, s) \}
\]

\[
\text{Identifiers} = PreId - \text{Keywords}
\]

**Numbers**

To be able to give a formal definition of numbers for the paradigm scanner, a number of auxiliary sets are needed first for the digits of the bases that should be supported by the scanner.

\[
\text{Dig} = [0 \ldots 9] \quad \text{the decimal digits}
\]

\[
\text{BinDig} = [0, 1] \quad \text{The binary digits}
\]

\[
\text{OctDig} = [0 \ldots 7] \quad \text{The octal digits}
\]

\[
\text{HexDig} = [0 \ldots 9, A \ldots F, a \ldots f] \quad \text{the hexadecimal digits}
\]

Given those auxiliary sets, it is then easy to give the definition for the three special bases, as the number is simply a prefix followed by one or more of the digits for that base. The formalization of the decimal numbers consists of three parts, namely the integer part, the decimal part and the exponential part. Depending on the exact settings for the scanner, one or more of these parts may not exist.

\[
\begin{align*}
\text{Int} & = \text{Dig}^+ \\
\text{Dec} & = (', ') \cdot \text{Dig}^+ | \varepsilon \\
\text{Exp} & = (('e' | 'E') \cdot (', +', ') - ' | \varepsilon) \cdot \text{Dig}^+ | \varepsilon \\
\text{Number} & = (0^* | 2r') \cdot \text{BinDig}^+ \\
& = (0^* | 8r') \cdot \text{OctDig}^+ \\
& = (0^* | 16r') \cdot \text{HexDig}^+ \\
& = \text{Int} \cdot \text{Dec} \cdot \text{Exp}
\end{align*}
\]

**Strings**

Thanks to the choices made earlier the definition of a string is straightforward, i.e. a delimiter followed by a string-character, an escape-sequence or a double-delimiter, eventually followed by the same delimiter again. As with the numbers, some parts may not exist, specifically the double-delimiter portion and the escape-sequence.
String = \' * ((V - \[\]) | e \cdot V | "\" )\^* \cdot \'
= " * ((V - "\]) | e \cdot V | \"\" )\^* \cdot "
e = escape character

Comments

Delimited comments are formally much the same as the strings, with the exception that they do not have escape-sequences or double-delimiters. End-of-line comments are much like delimited comments except that they have cr or lf as closing delimiters.

Comment = ( // | ; | % ) \cdot (V - [cr,lf] )\^* \cdot ( cr | If )
= /\* ( V\^* - ( V\^* \cdot /\* ) ) \cdot */
= (\* ( V\^* - ( V\^* \cdot \* ) ) \cdot \*)
= { ( V\^* - ( V\^* \cdot \* ) ) \cdot \} }

Symbols

The symbols are straightforward to formalize though restrictions are required on them to remove the potential for ambiguity from the scanner. One restriction is that symbols cannot be of the form \( S \cdot T\^* \) as this would make it possible for a string to be both a symbol and an identifier. Another is that symbols are not allowed to start with a digit as this digit might be considered a number by the scanner. The formalization of the symbols then becomes.

\( SYM \subseteq (V - Dig - S)\cdot V^* \) : The list of symbols

Symbols = SYM

Whitespace

Formalizing the whitespace is easy. Given the set of whitespace-characters, the definition of whitespace is any string consisting of these characters.

\( WS \subseteq [cr,lf,tab,space] \)

Whitespace = WS\^* 

4.3.4 Class Interface

The public interface of the paradigm definition component expands on that of the definition base class in section 3.5. The number of tokens defined is always 6, namely: identifier, reserved word, symbol, number, string, and comment. The types for each of these tokens are the ones specified by the user. More on this in section 4.3.5.

The rest of the public interface of the paradigm definition component follows almost directly from the findings on the structure of lexical elements presented in section 4.3.2. For all lexical elements a boolean enabled value denotes whether or not that particular lexical elements should be scanned or not.

Identifiers:

Enabled:boolean
FirstChar:set of (fcLowercase, fcUppercase, fcUnderscore)
NextChar:set of (ncLowercase, ncUppercase, ncDigits, ncUnderscore)
HasMaximumLength:boolean
MaximumLength:integer

The FirstChar and NextChar sets correspond to the three, respectively four sets of characters that are used for the identifiers. These sets of enumerated values are then used to construct the sets $S$ and $T$. The HasMaximumLength property determines whether a check for maximum length is made and, if so, the MaximumLength property will hold the maximum length, corresponding to value $M$.

Reserved words:
   Enabled:boolean
   List:TStringList
   CaseSensitive:boolean

   The List property contains the list of strings corresponding to $KW$ that should be interpreted as reserved words. This list is restricted by the settings in the identifier part of this interface as only reserved words valid under the restrictions of the identifiers are permitted. The CaseSensitive property determines whether the eq function is a case-sensitive or case-insensitive comparison function.

Symbols:
   Enabled:boolean
   List:TStringList

   The List property for the symbols contains the list in $SYM$. A number of restrictions are applicable on this list. No reserved words are permitted on this list, no symbols starting with digits and no symbols that correspond with active string- or comment-delimiters.

Numbers:
   Enabled:boolean
   HasDecimal:boolean
   Decimal:set of (dnDecimalPart, dnExponentialPart)
   HasBinary:boolean
   HasOctal:boolean
   HasHexadecimal:boolean
   NumberPrefix:(npZeroBased, npRadix)

   The four boolean properties HasDecimal, HasBinary, HasOctal and HasHexadecimal determine whether the scanner should respectively parse the decimal, binary, octal or hexadecimal numbers. The enumerated set Decimal is used to determine whether the decimal and exponential parts of number should be recognized. The NumberPrefix property determines which of the two prefix sets should be used.

Strings:
   Enabled:boolean
   Delimiters:set of (sdSingleQuote, sdDoubleQuote)
   HasEscapeSequence:boolean
   EscapeCharacter:char
   HasDoubleDelimiters:boolean

33
The enumerated set **Delimiters** determines the types of delimiters the strings can use. **HasEscapeSequence** denotes whether escape-sequences are allowed and, if so, **EscapeCharacter** will contain the character used for these escape-sequences. **HasDoubleDelimiters** specifies whether a pair of delimiters directly following each other is converted to a single delimiter character instead. Two restrictions apply to these variables. The single and double quote can not be listed as delimiters if they appear in the symbols list. The escape character cannot be one of the active string-delimiters either.

**Comments**
- Enabled:boolean
- HasEndOfLineComments:boolean
- HasDelimitedComments:boolean
- EolDelimiters:set of (edSlashSlash, edSemicolon, edPercentage, edDashDash)
- DelDelimiters:set of (ddSlashStar, ddBraces, ddParenthesisStar)

**HasEndOfLineComments** and **HasDelimitedComments** specify that the scanner should recognize any of end-of-line comments and delimited comments respectively. Which delimiters should be recognized are stored in the two enumerated sets **EolDelimiters** and **DelDelimiters**. The restriction applies that, if the opening-delimiters of any of the comments appears in the list of symbol, these cannot be active as comment delimiters here.

**Whitespace**
- Enabled:boolean
- Characters:set of (wcSpace, wcTab, wcCarriageReturn, wcLinefeed)

The enumerated set **Characters** specifies which of the four possible whitespace characters should actually be used as whitespace.

### 4.3.5 Graphical User Interface

The graphical user-interface for the paradigm language definition component follows almost directly out of its public interface. Enumerations are visualized as radio-buttons, while enumerated sets are visualized as groups of checkboxes. Lists of strings can be represented as a grouping of a string-list, an edit-box, ‘add’ and ‘remove’ buttons to respectively add or remove strings and ‘up’ and ‘down’ buttons to re-arrange the strings in the string-list. Boolean values correspond to a single checkbox and integers and characters can be entered through edit-boxes. This results in the interfaces for each of the lexical elements as listed in figure 4.3.

The restrictions on the reserved words and symbols are handled by disallowing the user to enter any reserved words or symbols that conflict with the settings on other parts of the interface. In all cases, an explanation is given on why the text is not a legal entry and what might be done to change this. The restrictions on the string- and comment-delimiters are handled by disallowing the user to change any options that would conflict with the symbols-list. Once again, an explanation will be given to the user as to the reason.
Figure 4.3: Interfaces
4.4 External Definition

The external language definition does not so much provide a language definition than that it allows the user to specify meta-information about an external scanner. This meta-information allows the framework to handle tokens from an unknown scanner without needing to know exactly how these tokens are recognized by the scanner.

The information required by the scanner framework consist of a list of token-names and their respective token-types. The public interface of the external definition reflects these requirements, implementing the, from the base definition inherited, properties `GetNumberOfTokens`, `GetTokenName`, and `GetTokenType`. In addition to these it gets a pair of methods `SetTokenName` and `SetTokenType` to allow the user to set these properties. `AddToken` and `RemoveToken` allow the user to add or remove a token respectively, while `ClearTokens` clears all token-definitions in the external definition. The public interface of the external definition is then as follows:

```
function GetNumberOfTokens : integer;
function GetTokenName(idx : integer) : string;
procedure SetTokenName(idx : integer; aName : string);
function GetTokenType(idx : integer) : TTokenType;
procedure SetTokenType(idx : integer; aType : TTokenType);
function AddToken(aName : string; aType : TTokenType) : integer;
procedure RemoveToken(idx : integer);
procedure ClearTokens;
```

4.4.1 Graphical User Interface

The user-interface of the external definition editor reflects the simple requirements of the external definition, as shown in figure [4.4]. A list of token/type values allows the editing of already existing tokens while maintaining integrity of the definition by disallowing duplicate token names. Two buttons to the right enable the user to add and remove tokens, while another pair allow him to move tokens up and down. The latter is mostly for user comfort as the order of the token-definitions have no meaning in the external definition class.

![External Definition Editor GUI](image)

Figure 4.4: External Definition Editor GUI
Chapter 5
Paradigm Scanner

A pre-compiled parameterized scanner can be developed based on the paradigm definition from section 4.3. This scanner is described in 5.1. The associated scanner factory is developed in section 5.2.

5.1 The Scanner Class

The implementation of the paradigm scanner uses a pre-compiled scanner that has been constructed in such a way that it performs parameterized scanning. This section covers the various parts of the paradigm scanner definition and how the scanner can be implemented to handle the settings in each of these parts. A brief report on what methods remain to be implemented in section 5.1.1 is followed by a more in-depth discussion of the steps needed to implement the paradigm scanner in sections 5.1.2 to 5.1.13.

5.1.1 Public Interface

The interface of the paradigm scanners follows almost directly out of the interface for the scanner base class from section 3.4.

```delphi
property Source: TStream;
property SymText: string;
property SymStart: integer;
property SymType: TSymType;
procedure Reset;
function IsFinished: boolean;
procedure NextSym;
```

The only method that still needs implementing is the `NextSym` method. The `Reset` and `IsFinished` methods are already handled in the scanner base class.

Additionally three values need to be set on each iteration of the scanner. These are the values for the text of the symbol, its starting position and its type, `SymText`, `SymStart`, and `SymType` respectively.
5.1.2 “Nextsym” Structure

When attempting a one-character look-ahead, the global structure of the scanner is practically fixed. What must be done is to decide which element comes next based on the first character of the input-text. Based on this observation, the global structure becomes as shown in listing 5.1

Listing 5.1: NextSym Implementation

```pascal
procedure NextSym;
begin
  "remove whitespace";
  SymStart:=Text.Pos;
  if Character in "first(identifier)" then
    "scan identifier/reserved word"
  else if Character in "first(number)" then
    "scan number"
  else if Character in "first(symbol)" then
    "scan symbol"
  else if Character in "first(string)" then
    "scan string"
  else if Character in "first(comment)" then
    "scan comment"
  else
    "return error"
  SymLength:=Text.Pos - SymStart;
end
```

5.1.3 Guards

For that global structure to work, the guards of the if-statement should be mutually exclusive, preferably with a simple one-character look-ahead. For the identifiers and numbers, this holds, but there is potential for overlap in the initial characters for the symbols, strings and comments. Before solving that problem though, it is easiest to get the guards for the identifiers and numbers out of the way.

The guard for the identifiers should check whether the next character is in the set of first characters for identifiers. Fortunately, this set already exists, so first(identifier) can be changed to a set of characters IdCharFirst constructed out of Identifier.FirstChar.

Due to the choices made in section 4.3.2 for the prefixes for the binary, octal and hexadecimal numbers, all numbers start with a digit, regardless of the type. Therefore, checking whether the next lexical element is a number is done by checking whether the first character is a digit.

The other three guards are not that easy. The sets for the string- and comment-guards are mutually exclusive, but the symbols guard-set can contain characters that also appear in one of the other two sets. Placing limitations on the list of symbols does not work here, as this would limit the paradigm scanner to the point of uselessness. A good example of this would be the symbol ` and the comment-opening-delimiter (*, as symbol ` is present in most languages.
Normally this problem can be solved by factorization. This way, the choice on which way to go is made at a later stage when it is possible to make a distinction based on the next character. For example, instead of writing the regular expression $a|ab$ it is possible to write $a \cdot (c|b)$. However, this method requires knowledge of the exact list of symbols and since it can be set by the user, this list is not known at the time of implementation.

The solution to get past this problem is to create a combined set of strings that holds the strings- and opening-delimiters for the strings and comments. This way the three initial guards are combined and a one-character look-ahead set can be given. Only after the string has been scanned the three possibilities are separated again, marking the recognized string as a symbol only if it does not correspond with a string- or a comment-delimiter. If it does correspond with a string- or comment-delimiter, then the scanner can scan the remainder of the string or comment.

With all these considerations in mind, the body of NextSym then becomes the one in listing 5.2:

Listing 5.2: NextSym Implementation

```pascal
procedure NextSym;
begin
  "remove whitespace";
  SymStart := Text.Pos;
  if Character in IDCharFirst then
    "scan identifier/reserved word"
  else if Character in ['0'..'9'] then
    "scan number"
  else begin
    "scan symbol/delimiter";
    if "string delimiter scanned" then
      "scan string"
    else if "end-of-line comment delimiter scanned" then
      "scan end-of-line comment"
    else if "delimited comment delimiter scanned" then
      "scan delimited comment"
    else if "symbol scanned" then
      SymType := <Symbol>
    else
      "return error"
  end;
end
```

5.1.4 Trie

What is needed is a method to quickly recognize one string out of a list of possible strings. The structure used for this is a tree-shaped finite state-machine called a trie. A trie can be used to recognize strings by moving from the start-state of the trie to an end-state over a series of transitions matching the characters of the string. If no end-state is reached, then the string is not accepted.
by the trie.

Since the trie is a finite state-machine, its definition can be given by a five-
tuple $T = (Q, V, \delta, q_0, F)$. If $K$ is the set of strings to be stored in the trie, with
$K \subseteq V^*$ and finite, then the components of the trie can be given by:

$$
\begin{align*}
V & \quad \text{all characters in } K \\
Q & \quad \{s | s \cdot t \in K \land t \in V^*\} \\
\delta & \quad \in Q \cup \{\bot\} \times V \rightarrow Q \cup \{\bot\} \\
\delta(q, a) & \quad \text{if } qa \in Q \\
& \quad \bot \text{ if } qa \not\in Q \\
F & \quad K
\end{align*}
$$

A public interface can then be defined for the trie implementation. A num-
ber of methods allow the user to set and check the state that the trie is in. 
Reset returns the trie to its initial state, isFinalState return whether a state
is a final state, Mark and Recall allow the user to respectively store the cur-
rent state and set the current state to the stored value. HasTransition($\alpha$)
returns whether the current state has a valid transition with label $\alpha$, while
MakeTransition($\alpha$) sets the current state to the state reached by making a
transition $\alpha$. Text returns the string that has been recognized up to the current
state, and Length returns its length.

The actions performed by these methods can then be formalized in terms of
the trie definition below. With $s$ equal to the state in which the trie is now, the
actions performed by the methods are:

- **Reset** $s := \varepsilon$
- **hasTransition($\alpha$):boolean** $\delta(s, \alpha) \neq \bot$
- **makeTransition($\alpha$)** $s := \delta(s, \alpha)$
- **isFinalState** $s \in K$
- **Mark** $t := s$
- **Recall**
  - $s := t$ if $t \neq \bot$
  - $s := \varepsilon$ if $t = \bot$
- **Text:string**
  - $s$ if $s \neq \bot$
  - "$" if $s = \bot$
- **Length:integer**
  - $|s|$ if $s \neq \bot$
  - 0 if $s = \bot$

### 5.1.5 “Remove Whitespace”

Removing the whitespace in front of any potential tokens can be done easily by
removing all characters from the whitespace set until one encounters a character
that is not part of the set. This character must then be the start of a new token.
Listing 5.3: "remove Whitespace" Implementation

```plaintext
{ wsChars = set of whitespace-characters }
while (Character in wsChars) do
    NextChar;
```

5.1.6 “scan identifier/reserved word”

In the implementation for the identifiers/reserved words, the variables IdCharFirst and IdCharFollow are sets of characters constructed based on the settings in Identifier.FirstChar and Identifier.NextChar respectively. The trie RW Trie has only lowercase-entries if the reserved words are not case-sensitive, otherwise it will contain all entries in a case-preserving manner.

Listing 5.4: "scan identifier/reserved word" Implementation

```plaintext
{ Character in IdCharFirst
    RW Trie = trie of reserved words
    setcase(ch) = ch if RW case-sensitive
                 = lowercase(ch) otherwise }

RW Trie.makeTransition(setcase(Character));
NextChar;

while (Character in IDFollow) do begin
    RW Trie.makeTransition(setcase(Character));
    NextChar
end;

SymLength := Text.Pos - SymStart;
if (Identifiers.hasMaximumLength) then
    if (SymLength > Identifiers.MaximumLength) then
        error('identifier/reserved word is too long');

if (RW Trie.isFinalState) then
    SymType := ReservedWord.SymType
else
    SymType := Identifier.SymType
```

5.1.7 “scan number”

The first step of scanning numbers is to determine whether the number is preceded by the prefix of one of the non-decimal bases. For this purpose a trie is constructed holding only the active number prefixes. This is then used to scan the text until it recognizes a prefix or until the trie fails to find one. In the former case, a comparison is done on the prefixes to determine which base the number was in. In the latter case a decimal number is scanned.
Listing 5.5: "scan number" Implementation

{ Character in [ '0' .. '9']
  NumPref = trie of active number-prefixes }
Mark;
NumPref.Reset;
while NumPref.hasTransition(Character) do begin
  NumPref.makeTransition(Character);
  NextChar;
  if NumPref.isFinalState then begin
    { Text[SymStart..TextPos) is longest match so far }
    Mark;
    NumPref.Mark;
  end;
end;
Recall;
NumPref.Recall;
if (NumPref.Text = '') then
  "scan decimal number"
if (NumPref.Text = '0b') or (NumPref.Text = '2r') then
  "scan binary number"
if (NumPref.Text = '0') or (NumPref.Text = '8r') then
  "scan octal number"
if (NumPref.Text = '0x') or (NumPref.Text = '16r') then
  "scan hexadecimal number"

5.1.8 Scanning non-decimal numbers

Once it has been determined that a non-decimal number will follow (binary, octal or hexadecimal), all that remains to be done is to recognize the string of digits that should follow the number-prefix. If no digits are found at all, then an error will be returned.

Since the implementation is exactly the same for all three non-decimal number-types, only the version for octal numbers is given here. All that differs for the other two versions is the set of characters in Digits.

Listing 5.6: Non-decimal numbers Implementation

{ digits = [ '0' .. '7'] }
if (Character in digits) then begin
  while (Character in Digits) do begin
    NextChar;
  end;
end else begin
  error('character from [ '0' .. '7'] expected');
end;
SymType := Number.SymType
5.1.9 Scanning decimal numbers

Scanning the decimal numbers is more involved than the non-decimal numbers. Though there is no prefix to deal with, the decimal numbers consist of one necessary and two optional parts. Scanning these is done by checking for a string of numbers first. If this string is followed by a decimal point it will recognize it and check for the string of number corresponding with the decimal part. Using the same method it then checks for the E for exponential values, the optional sign for them and the string of numbers corresponding to the exponential part.

Listing 5.7: decimal numbers Implementation

```pascal
{ Character in ['0'..'9'] }
if (Character in ['0'..'9']) then
  while (Character in ['0'..'9']) do begin
    NextChar;
  end;

if (Numbers.hasDecimal) then begin
  if (Character = '.') then begin
    NextChar;
    if (Character in ['0'..'9']) then begin
      while (Character in ['0'..'9']) do begin
        NextChar;
      end;
    end else begin
      error('digit expected');
    end;
  end;
end;

if (Numbers.hasExponential) then begin
  if (Character in ['e', 'E']) then begin
    NextChar;
    if (Character in ['+', '-']) then begin
      NextChar;
      end;
    if (Character in ['0'..'9']) then
      while (Character in ['0'..'9']) do begin
        NextChar;
      end;
    else begin
      error('digit expected');
    end;
  end;
end;
SymType := Number SymType
```

43
5.1.10 "scan symbol/delimiter"

With a trie, scanning for a symbol/delimiter is a matter of traversing the trie with the relevant symbols and delimiters until the final state with the longest match has been found. The implementation for this is shown in listing 5.8.

Listing 5.8: "scan symbol/delimiter" Implementation

```pascal
Mark;
SymTrie . Mark;
while SymTrie . hasTransition ( Character ) do begin
  SymTrie . makeTransition ( Character ) ;
  if SymTrie . isFinalState then begin
    { Text[SymStart..TextPos) is longest match so far }
    Mark;
    SymTrie . Mark;
  end;
end;
Recall;
SymTrie . Recall;
```

5.1.11 "scan string"

The implementation for "scan string" would have been straightforward if it wasn’t for the potential presence of double delimiters in strings. This makes it so that one cannot tell whether the string has ended without looking two characters ahead. Through a rewrite of the definition for the strings, this problem can be dealt with elegantly though.

The definition \( \cdot ((V — [‘’])\cdot V'\cdot ‘’\cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot + \cdot \) can be rewritten as \( \cdot ((V — [‘’])\cdot V'\cdot ‘’\cdot \cdot \cdot \cdot \cdot \cdot \cdot + \cdot \) This changes the two-character look-ahead to a double loop, which is more elegant and less error-prone to implement. Using this alternate definition the implementation for "scan string" then becomes as in listing /refStringImpl.

Listing 5.9: String Implementation

```pascal
del := SymTrie . Text;
repeat
  while ( Character <> del ) do begin
    if ( Character = EscapeCharacter )
      and ( hasEscapeSequence ) then
      NextChar;
    NextChar;
  end;
  NextChar;
until not (( Character = del ) and ( hasDoubleDelimiters ));
SymType := String . SymType
```
5.1.12 “scan end-of-line comment”
An end-of-line comment can be scanned by scanning all characters until a carriage return or linefeed symbol is encountered. This process is straightforward and does not require much explanation.

Listing 5.10: "scan end-of-line comment” Implementation

```pascal
{ SymTrie.Text = end–of–line delimiter }
while not (Character in [cr, lf]) do begin
  NextChar;
end;
NextChar;
SymType := Comments.SymType
```

5.1.13 “scan delimited comment”
Delimited comments are among the more difficult elements to scan. Once an opening-delimiter has been recognized, the matching closing delimiter is chosen. Since the closing-delimiter can only be up to two characters long, a simple algorithm can be used to check whether the closing delimiter appears in the text.

Listing 5.11: "scan delimited comment” Implementation

```pascal
if (SymTrie.Text = '(*') then cdel := ')*';
if (SymTrie.Text = '{') then cdel := '}';
if (SymTrie.Text = '/*/') then cdel := '*/';

state := 1;
while (state <= length(cdel)) do begin
  if (Character <> cdel[state]) then
    state := 1;
  if (Character = cdel[state]) then
    inc(state);
    NextChar;
end;

SymType := Comments.SymType
```

5.2 The Paradigm Factory
The design of the paradigm scanner factory component is straightforward. A single published property `ParadigmDefinition` enables the user to select a suitable paradigm scanner definition within the Delphi IDE. A single public function `CreateParadigmScanner` returning a `TParadigmScanner` produces a suitable scanner that will use the specified definition.
The inherited property **Definition** is read-only and returns the value of **ParadigmDefinition** and the function **CreateScanner** redirects the call to **CreateParadigmScanner** to create a suitable scanner.

This makes the public interface of the Paradigm Scanner Factory component as follows:

**Public**

property **Definition**: TSFWDefinition;

function **CreateScanner**: TSFWSScanner;

function **CreateParadigmScanner**: TParadigmScanner;

**Published**

property **ParadigmDefinition**: TPSDefinition;

5.3 Example

The example language from section 2.5 can be recognized by the paradigm scanner. This requires that the properties in the paradigm definition are set to the following values. Note that, for brevity, all properties that are not mentioned are either disabled or empty, depending on the property-type.

**Identifiers**

Enabled := true
FirstChar := [fcLowercase, fcUppercase]
NextChar := [fcLowercase, fcUppercase]
HasMaximumLength := false

**ReservedWords**

Enabled := true
List := ['e', 'pi']
CaseSensitive := false;

**Symbols**

Enabled := true
List := ['=', '+', '-', '/', '*', '(', ')']

**Numbers**

Enabled := true
HasDecimal := true
Decimal := [dnDecimalPart]

**Whitespace**

Enabled := true
Characters := [wcSpace, wcTab, wcCarriageReturn, wcLineFeed]
Chapter 6

T-Recipe Scanner

The T-Recipe \cite{Eik00} was originally designed to construct acceptors for simple regular expressions. By expanding upon this basis, it is possible to create a scanner that can recognize a specific subset of the regular languages. This chapter describes this algorithm.

Starting with an explanation of the original T-Recipe in section 6.1.1, the algorithm is then further expanded in sections 6.1.3 to 6.1.6. Section 6.2 details the problems one encounters when trying to construct a T-Recipe scanner and how these can be solved. Section 6.3 explains how the T-Recipe can be used to create a scanner-framework compatible scanner, while section 6.4 discusses the implementation of the accompanying T-Recipe Scanner Factory. Section 6.5 expresses the example language in it.

6.1 The T-Recipe

6.1.1 The T-Recipe for regular expressions

The T-Recipe, as described in \cite{Eik00} and \cite{Eik91}, is an algorithm to convert a regular expression into an equivalent acceptor. This is done by defining a recursive function on the regular expressions that describes how each part of the expression can be converted into code.

The “naive” version of this algorithm is a function $T : RE_V \rightarrow GCL$ with the following specification:

\[
\begin{align*}
T(\epsilon) &= \text{skip} \\
T(\alpha) &= \text{skip} \quad (\alpha \in V) \\
T(e_1 \cdot e_2) &= T(e_1); T(e_2) \\
T(e_1 | e_2) &= \\
&\quad \text{if} (\text{character} \in \text{First}(e_1)) \rightarrow T(e_1) \\
&\quad \quad \text{if} (\text{character} \in \text{First}(e_2)) \rightarrow T(e_2) \\
T(e^*) &= \\
&\quad \text{do} (\text{character} \in \text{First}(e)) \rightarrow T(e) \text{ od}
\end{align*}
\]
A few issues arise with this version. Consider the regular expression $e = (a \mid b^*) \cdot c$. Applying the T-Recipe gives this (intermediate) result:

$$T(e) =
\begin{cases}
\text{if (character } \in \text{ ['a']) } \rightarrow \text{T(a)} \\
\text{[] (character } \in \text{ ['b']) } \rightarrow \text{T(b*)} \\
\text{fi;}
\end{cases}
\text{T(c)}$$

Although $c \in L(e)$, the string $c$ is not accepted because none of the guards are true, causing the statement to abort instead. What went wrong here is that, since the expression $b^*$ can be empty, it should have had $c$ in the guard as well, because the way $c$ is recognized is by doing zero times $b$ followed by $c$.

The solution to this problem is to keep track of the characters that may follow a certain expression, then use this set in cases where a guard may be followed by an empty statement. This is done by extending the T-Recipe with an additional parameter. Before the revised T-Recipe can be given, an auxiliary function has to be defined first. For $e \in \text{RE}(V)$ and $F \in \mathcal{P}(V)$:

$$h(e, F) = \begin{cases}
\text{First}(e) \\
\text{First}(e) \cup F
\end{cases} \text{ if } \left( \begin{array}{c}
\text{Null}(e) \\
\text{Null}(e)
\end{array} \right)$$

Given this function it is then possible to rewrite the specification of the T-Recipe by introducing a follow-set $F$. This follow-set contains all characters that might follow the expression. The algorithm $T : \text{RE}(V) \times \mathcal{P}(V) \rightarrow \text{GCL}$ then becomes:

$$T(\epsilon, F) = \text{skip}$$
$$T(\alpha, F) = \begin{cases}
\text{skip} & (\alpha \in V) \\
\text{if (character } = \alpha) \rightarrow \text{NextChar fi}
\end{cases}$$
$$T(e_1 \cdot e_2, F) = T(e_1, h(e_2, F)); T(e_2, F)$$
$$T(e_1 | e_2, F) =
\begin{cases}
\text{if (character } \in h(e_1, F)) \rightarrow T(e_1, F) \\
\text{[] (character } \in h(e_2, F)) \rightarrow T(e_2, F) \\
\text{fi}
\end{cases}$$
$$T(e^*, F) = \text{do (character } \in \text{First}(e)) \rightarrow T(e, \text{First}(e) \cup F) \text{ od}$$

### 6.1.2 det-conditions

One problem remaining is that of determinism. If an ambiguous expression such as $e = (ab \mid ac)$ is used in the T-Recipe, it will cause the guards in the case-statement to overlap. For $e$ this leads to this ambiguous statement:
T(e, F) =
  if (character ∈ ['a']) → T(a, [b]); T(b, F)
  [] (character ∈ ['a']) → T(a, [c]); T(c, F)
fi

When recognizing strings from L(e), the algorithm cannot make a correct decision when an a is encountered. For example, when recognizing the string ac, it may take the first branch and end up in a deadlock, even though there is a branch in the program that would have led to successful acceptance.

Therefore, it is important to place a few conditions on the regular expressions that serve as an input so that the acceptor will finish successfully whenever possible. The first requirement is that the guards for the selection are mutually exclusive, i.e. for e1|e2 with a follow-set F, h(e1, F) ∩ h(e2, F) = ∅.

A second requirement is that the repetition must terminate. To ensure this, the body of the repetition should remove at least one character from the input. This means that, for a repetition e*, e ∉ L(e), i.e. ¬null(e).

In the case of a repetition e* a deterministic decision to either continue the repetition or to end it must be possible. This can only be done if the characters following the repetition are different from the set of initial characters of the repetition itself, so the requirement first(e) ∩ F = ∅ must hold.

Since these conditions must hold for subexpressions as well, a mapping \( det : REV \times P(V) \rightarrow boolean \) is defined that determines whether a given regular expression and its sub-expressions comply with the conditions given above. These conditions will from here on be called the det-conditions.

\[
\begin{align*}
\text{det}(\epsilon, F) &= \text{true} \\
\text{det}(\alpha, F) &= \text{true (for all } \alpha \in V) \\
\text{det}(e_1 \cdot e_2, F) &= \text{det}(e_1, h(e_2, F)) \land \text{det}(e_2, F) \\
\text{det}(e_1 | e_2, F) &= h(e_1, F) \cap h(e_2, F) = \emptyset \\
&\quad \land \text{det}(e_1, F) \land \text{det}(e_2, F) \\
\text{det}(e^*, F) &= \neg \text{null}(e) \land \text{first}(e) \cap F = \emptyset \\
&\quad \land \text{det}(e, \text{first}(e) \cup F)
\end{align*}
\]

### 6.1.3 n-ary operators

It stands to reason that this method of generating a scanner doesn’t always produce the most readable and efficient code. There are a number of things that can be done to improve on this. The first optimization is in the way that the selection is handled. Expressions of the form \( a \mid b \mid c \mid d \), when read as \(((a \mid b) \mid c) \mid d\) can lead to code that looks like this:

\[
\begin{align*}
\text{if (character ∈ ['a', 'b', 'c'])} \rightarrow \\
&\quad \text{if (character ∈ ['a', 'b'])} \rightarrow \\
&\quad \quad \text{if (character ∈ ['a'])} \rightarrow \\
&\quad \quad \quad \text{if (character = 'a') → NextChar fi} \\
&\quad \quad \text{[] (character ∈ ['c'])} \rightarrow \\
&\quad \quad \text{if (character = 'b') → NextChar fi} \\
&\quad \text{fi} \\
&\quad \text{[] (character ∈ ['c'])} \rightarrow
\end{align*}
\]
Although in this case the result is still somewhat overseeable, the addition of other elements to the regular expression will quickly complicate it to the point of unreadability. A better way to interpret an expression like $a \mid b \mid c \mid d$ would be to treat the whole as a selection statement of arbitrary arity. In terms of the T-Recipe this then leads to the replacement of the rule for $T(e_1 \mid e_2)$ with $T(e_1 \mid ... \mid e_n)$ for $n > 1$. This enables us to use a single if-statement instead of a series of nested if-statements.

Like the selection, the concatenation can be made n-ary as well. Though this does not affect the readability, it provides a more consistent definition for the regular expressions. Through recursion, the n-ary definition of the concatenation for $n > 1$ can be given as follows:

$$
T(e_1 \mid ... \mid e_n, F) = \begin{cases} 
T(e_1, F) & \text{if } (\text{character } \in h(e_1, F)) \\
T(e_2, F) & \text{if } (\text{character } \in h(e_2, F)) \\
... & \text{if } (\text{character } \in h(e_n, F)) \\
\end{cases}
$$

6.1.4 Pascal representation

Although GCL is a good language for theoretical discussions, it has some drawbacks when it comes to more practical application of the code described with it.

One problem is that GCL uses an implicit error-state in its if-statement. Should it occur that none of the guards in the statement are true, then execution of the program will simply halt. This is neither desirable nor possible in Delphi. The solution is to add an extra error-case to all selections which, should all guards in the selection turn out to be false, informs the user and gives him information on what characters were expected instead.

The translation of the selection from GCL to Pascal is best done with a case-statement due to the large number of statements possible within the selection. In GCL it does not matter whether the guards in the selection are mutually exclusive, but for Pascal’s case-statement it is actually required that they are. This due to the way case-statements are commonly implemented. Fortunately, the det-conditions discussed in section 6.1.2 already take care of this requirement, so no special arrangements need to be made for this.

One special case for the translation is that of the $\epsilon$. There is no equivalent instruction in Pascal for the GCL skip-statement. Although it is possible to generate the Pascal empty statement \[ISO7158\], this might lead to confusing code after generation. Therefore a method called skip is added to the generated code, which does nothing. This allows us to translate the $\epsilon$ as a skip.
T(ε, F) = skip

T(α, F) = if (character = α) then NextChar else Error (α ∈ V)

T(e₁, e₂, F) = T(e₁, h(e₂, F)); T(e₂, F)

T(e₁ · ... · eₙ, F) = T(e₁ · ... · eₙ−1, h(eₙ, F)); T(eₙ, F) (n > 2)

T(e₁ | ... | eₙ, F) =
  case character of
    h(e₁, F)) : T(e₁, F);
    ... : ...
    h(eₙ, F)) : T(eₙ, F);
  else
    Error
end

T(ε*, F) =
  while (character in First(ε)) do begin
    T(ε, First(ε) + F)
  end

6.1.5 Optimizations

Although the T-Recipe produces correct code, it is occasionally far from efficient. This is particularly noticeable in the way certain checks and actions are duplicated. If one were to apply the T-recipe to the expression \( a \cdot b \mid c \), one would get the following code:

```plaintext
case character of
  'a' : begin
    if (character = 'a') then NextChar else Error;
    if (character = 'b') then NextChar else Error
  end;
  'c' : if (character = 'c') then NextChar else Error;
else
  Error;
end
```

Suppose the first character in the input is an 'a', then the program will execute the corresponding block of the case-statement. This would be directly followed by the if-statement performing another check to see whether the character is an 'a', even though this was already established in the case-statement. A similar situation occurs if the first character is a 'c'.

To avoid situations like these, it is possible to extend the T-Recipe with an extra parameter \( P \) which contains the set of characters of which it is known
that the next character is part of it. This parameter can then be used to check whether it is possible to choose a more efficient translation for that particular statement. The possibility of optimization occurs in any locations where a choice is made, so for characters, repetition and selection.

For characters the optimization is easy. If the set of previously recognized characters is equal to the character being recognized, then all that needs to be done is to move on to the next character. If the character being recognized is not in the set P, this means that it can never occur at that point in the acceptor, thus an error is thrown. If neither of these two conditions holds, the normal translation is used.

Since all guards of the selection are disjoint, optimization focuses on whether the set P is contained fully in one of these guards. If so, then only that option can be true and the selection can be reduced to the statement following that guard. If this is not the case, then some optimization can still be obtained by removing any guards that are completely disjoint with the set P, as these can not be true.

The only optimization possible for a repetition $e^*$ is to check whether the guard and the set P are disjoint. If they are, then the repetition will never enter its body and as such can be replaced by a simple skip statement. Care must be taken on which value for P to pass to the T-Recipe call in the body, as P is only a valid value for that on the first pass. Since the guard established that the next character is in First($e$), this is the smallest set for P that is safe at all times.

\[
T(\epsilon, F, P) =
\]

\[
\text{skip}
\]

\[
T(\alpha, F, P) =
\]

1. NextChar
2. Error
3. if (character = $\alpha$) then NextChar else Error

\[
T(e_1 \cdot e_2, F, P) =
T(e_1, h(e_2, F), P); T(e_2, F, V)
\]

\[
T(e_1 \cdot \cdots \cdot e_n, F, P) =
T(e_1 \cdot \cdots \cdot e_{n-1}, h(e_n, F), P); T(e_n, F, V)
\]

\[
T(e_1 \mid \cdots \mid e_n, F, P) =
\]

1. $T(e_1, F, P)$
2. ...
3. $T(e_n, F, P)$
4. case character of
   - h($e_1, F$) : $T(e_1, F, P \cap h(e_1, F))$
   - \vdots
   - h($e_n, F$) : $T(e_n, F, P \cap h(e_n, F))$
   else
     Error
end

\[
T(e^*, F, P) =
\]

1. skip
2. while (character in First($e$)) do begin
   $T(e, \text{First}(e) \cup F, \text{First}(e))$
end
6.1.6 Extended expressions

Although the extended regular expressions can be handled by transforming them through the rules in section [6.1.5] before applying the T-recipe, this can lead to inefficient and unnecessarily complicated code. Instead it is possible to define specific rules for the transformation of these expressions.

A character set \([\alpha - \beta]\) can be handled in much the same way as a single character. If the next character in the input is part of the set, then the scanner can move on to the next character, otherwise it must throw an error. If the set \(P\) is a subset of the range, then checking the guard is unnecessary and the acceptor can immediately move on to the next character. If set \(P\) and the range are disjoint, an error is thrown instead. This leads to the following translation for character-sets.

\[
T([\alpha - \beta], F, P) =
\]
1. NextChar \[P \subseteq [\alpha - \beta]\]
2. Error \[[\alpha - \beta] \cap P = \emptyset\]
3. if (character in \([\alpha..\beta]\)) then NextChar else Error otherwise

A more efficient translation for the one-or-more repetition can be found by actually using its definition and applying the T-recipe to it.

\[
T(e^+, F, P) =
T(e; e^*, F, P) =
T(e, h(e^*, F, P)); T(e^*, F, V) =
T(e, First(e) \cup F, P);
\]
while (character in First(e)) do begin
\[
T(e, First(e) \cup F, First(e))
\]
end;

By taking the first T-Recipe call into the loop, it can be rewritten to a nearly equivalent repeat-statement. The only difference is that there may be a loss of optimization in the body now because the set First(e) has to be expanded to \(P \cup First(e)\).

\[
T(e^+, F, P) =
\]
repeat
\[
T(e, First(e) \cup F, P \cup First(e))
\]
until not (character in First(e))

A statement \(e?\) indicates that the expression \(e\) occurs once or not at all. This is equivalent to \(e \mid \epsilon\). Putting this into the T-Recipe gives the following translation.
This leaves the translation of the string. The string can be written as a repeated application of the T-Recipe on characters, where the preceding character set for the first call is equal to P, while equal to V for all following calls.

\[
T(\sigma, F, P) = T(\sigma_0, F, P) \cdot T(\sigma_1, F, V) \cdot \ldots \cdot T(\sigma_{|\sigma|}, F, V)
\]

The det-conditions for the extended expressions can be derived from the det-conditions of their equivalent regular expressions.

\[
\begin{align*}
det([\alpha - \beta], F) &= \text{true (for all } \alpha, \beta \in V) \\
det(e^+, F) &= \neg \text{Null}(e) \wedge \text{First}(e) \cap F = \emptyset \wedge \det(e, \text{First}(e) \cup F) \\
det(e?, F) &= \neg \text{Null}(e) \wedge \text{First}(e) \cap F = \emptyset \wedge \det(e, F) \\
det(\sigma, F) &= \text{true (for all } \sigma \in V^*)
\end{align*}
\]

### 6.2 Scanner Construction

#### 6.2.1 T-Recipe Language Validation

In addition to the already existing restrictions for the regular language definitions, the T-Recipe requires extra restrictions to ensure the correctness of the language definition and to ensure that a correctly functioning scanner is generated. The required restrictions have already been defined, namely the det-conditions.

The det-conditions can be applied almost directly to the regular expressions listed in the language definition. For each token in the definition, the det-conditions check should be called with a set of follow-characters \(F_t\) discussed in section 6.2.3. The expressions do not need to be checked separately, because identifiers that may occur in the token definitions should be checked in the context in which they appear.

#### 6.2.2 Forming expressions

Now that the T-Recipe for regular expressions is known, it is time to describe the steps required to build a T-Recipe scanner out of a language definition. Before an exact specification can be given though, it is necessary to take a look at the parts from which a language definition is constructed.

A language definition is a mapping from labels to regular expressions. Whenever a label is encountered in one of these expressions, it should, for all practical
purposes, be replaced by the regular expression corresponding with that label. It is imagineable to place each expression in a separate method of the to-be-generated scanner-class so that the scanner can call these methods each time a label is encountered. This leads to a very clear mapping of the elements of the language definition to the logical elements within the generated scanner.

Had the translation from T-Recipe to Delphi code relied on the expression only, then this method could have worked. Unfortunately the set of follow-up characters needs to be taken into account as well. Only when this set is fixed for all calls to the method would the generated code within the method be correct. It is easy to prove that this is not the case. Take the example regular expressions \( A = a \cdot C \cdot a \mid b \cdot C \cdot b \) and \( C = c \mid d^* \). The label \( C \) is encountered twice in expression \( A \), once with a follow of \( a \) and once with a follow of \( b \). As shown earlier in section 6.1 the code generated from expression \( c \mid d^* \) requires the addition of the follow in one of its guards to result in a correct acceptor. This means that there would have to be two versions of \( C \), one generated with a follow of \( a \) and one with a follow of \( b \).

One obvious solution to this problem that springs to mind is to pass the follow-set along to the method through the use of a parameter. This would allow it to dynamically alter its behaviour based on the follow-set. Since the follow-set is still known in each occasion where the method is called, the det-conditions can still be checked to ensure that each method call accepts all the strings that it should.

This solution has a couple of problems that make it less than worthwhile though. Attempting to introduce the parameter into the generated code of the T-Recipe will lead to additional parameters in the T-Recipe itself. These parameters will again lead to case-distinctions, obfuscating the algorithm to the point where it becomes difficult to follow, if not plain unusable.

A second, more practical, problem occurs when it comes to generating the correct code. Since the parameter is for the follow-set and the follow-set is used for guards, the parameter would end up in the guards of the case-statements. Due to the way case-statements are implemented in Delphi, guards in case-statements need to be resolveable during compile-time, so this is not permitted. Therefore it would be necessary to use nested if statements instead, which results in less readable code too.

A second solution that springs to mind is to just forget about the methods and instead insert the translations of any expressions and tokens in any place where a label is encountered. This method, often called inlining, would result in one big unreadable clump of code. Since this is exactly what needed to be avoided in the first place, this option is not pursued any further.

A reasonable compromise between the first and latter solution is possible. References to tokens cannot exist in any expression in the language definition. Instead, tokens are implicitly used in the top level of the T-Recipe scanner. Since they are used only once there, they will always have the same set of follow-characters, which for now will be called \( F_t \) with \( t \in T \). This allows the scanner generator to create a separate method for each of the tokens, while inlining any sub-expressions that may appear in the tokens. This still leads to a clear, logical separation of the tokens in the scanner-class, and avoids making the T-Recipe unnecessarily complex.

Keeping the above considerations in mind, it is then possible to give a translation algorithm for the separate tokens. Given a language definition
\[ D = (V, N, R, T), \text{ the body of a method for a token } t \in T \text{ is the expression } T(R(t), F) \].

Since only labels to expressions can occur, the T-Recipe for labels and its associated det-conditions can be defined easily by applying the T-Recipe and the det-conditions to the expression that corresponds with the label. Thus the T-Recipe for the label is as follows:

\[
T(l, F, P) = T(R(l), F, P)
\]

\[
\det(l, F) = \det(R(l), F)
\]

The top-level of the scanner is a method that decides which token to scan next. This can be handled by constructing an expression \( e = (| t : t \in T : R(t) |) \) on which the T-Recipe is then applied with the follow-sets \( F_t \). Repeated calls to this method will result in the tokenisation of the input text.

### 6.2.3 Determining \( F_t \)

There are three things that can follow a token: the end-of-file marker, whitespace or another token. Given these things, it stands to reason to use a single follow-set for all tokens, namely \( F_0 = (\cup t : t \in T : \text{First}(R(t))) \cup \{\text{space, tab, cr, If}\} \cup \{\bot\} \).

However, this set leads to a severe limitation of the languages recognizable by the T-Recipe scanner. Given \( id = (a - z). (a - z)^* \), calculating \( \det(id, F_0) \) would require the condition \( \text{First}(a - z) \cap F_0 = \emptyset \) to hold in the sub-expression \( (a - z)^* \). However, the definition of \( F_0 \) tells us that \( \text{First}(a - z) \in F \). Thus the det-condition would fail for this token. Since \( id \) is a simplified version of the most common definitions of identifiers the set \( F_0 \) as defined is too restricting.

Expanding the set \( F_0 \) is not an option as this would not solve the issue, instead it should be restricted. This should be done within limits though, as too small a set of characters will in turn lead to other limitations in the scanner. Taking set \( F_0 \) to be only the whitespace and the end-of-file marker would lead to a scanner that requires whitespace between all tokens.

What would be best is a set \( F_t \) for each token \( t \) that is the maximum subset of \( F_0 \) that does not violate the det-conditions for that token. By analyzing what det-conditions are affected by the follow-set, it is possible to create a function \( V \) that returns all characters that would violate the det-conditions if they were to appear in the follow-set.

\[
\begin{align*}
V(\epsilon) &= \emptyset \\
V(\alpha) &= \emptyset \\
V(e_1 \cdot e_2) &= V(e_2) & \text{if } \neg \text{Null}(e_2) \\
&= V(e_1) \cup V(e_2) & \text{if } \text{Null}(e_2) \\
V(e_1 | e_2) &= V(e_1) \cup V(e_2) & \text{if } \neg \text{Null}(e_1) \land \neg \text{Null}(e_2) \\
&= V(e_1) \cup V(e_2) \cup \text{First}(e_2) & \text{if } \text{Null}(e_1) \\
&= V(e_1) \cup V(e_2) \cup \text{First}(e_1) & \text{if } \text{Null}(e_2) \\
V(e^*) &= \text{First}(e)
\end{align*}
\]
The three cases for the selection follow from the \( h(e_1, F) \cap h(e_2, F) = \emptyset \) requirement in its det-conditions. If \( \text{Null}(e_1) \) holds, this resolves to \( (\text{First}(e_1) \cup F) \cap \text{First}(e_2) = \emptyset \). This is only true if \( F \cap \text{First}(e_2) = \emptyset \), so \( F \) cannot contain any character from \( \text{First}(e_2) \) without violating the det-conditions. Similar reasoning leads to the case when \( \text{Null}(e_2) \) holds. The det-conditions already preclude the occurrence of \( \text{Null}(e_1) \land \text{Null}(e_2) \), so this does not need to be checked.

For the repetition, the set of violating characters is decided by the \( \text{First}(e) \cap F = \emptyset \) portion of its det-conditions. This can only be true if \( F \) does not contain any characters from \( \text{First}(e) \), so the set of violating characters equals \( \text{First}(e) \).

Given the function \( V \), it is then possible to define \( F_t \) as an equation over \( F_0 \) and \( V \). The follow-set of token \( t \) is the follow-set \( F_0 \) minus any characters that might violate the det-conditions, so \( F_t = F_0 - V(R(t)) \).

### 6.3 T-Recipe Generator

In section 6.2.2 it has already been established that each token can be generated into a separate method. Though it is possible to apply the T-Recipe directly to the token-definitions, this approach leads to the combination of two separate actions, namely the generation of the appropriate statements in Delphi and the representation and layout of these statements.

By adding a level of indirection between the regular expressions and the actual code, these two actions can be separated into distinct steps. First, the generator builds a syntax-tree defining the structure of the Pascal-code. Once constructed, this tree is then flattened into a textual representation of the actual scanner-class. The details of this approach are handled in this section.

#### 6.3.1 Syntax tree Definition

Syntax trees hold information about the structure of code without specifying a representation for it yet. Just as the regular expressions in section 4.1 have types of nodes for each of the operators, so do syntax trees have types of nodes for each of the statements.

The simplest statement in the syntax-tree is that of the method-call. It requires the name of a method to call and an optional list of parameters that need to be passed along. For the T-Recipe, this node is used to call NextChar, Skip and Error. Of these, only the call to error requires a parameter. This to explain the exact cause of the error.

There are two possible statements for selection, namely the if-statement and the case-statement. The if-statement takes a set of characters for the guard and a statement for when the guard is true. For the case where the guard is false it automatically adds a call to the error-method, stating that a character from the guard was expected. The case-statements takes an arbitrary number of charset/statement pairs. Like the if-statements the case-statement automatically adds a call to the error-method stating that a character from one of its guards was expected.

There are two possible statements for repetition as well, namely the repeat and the while. Both have the same signature, taking a character-set as guard and a statement as the body for their parameters. The main difference between
the two is that in the case of the repeat, the guard is evaluated afterwards, while in the case of the while, the guard is evaluated first.

Concatenation is handled through a statement-list that takes any number of statements as parameters.

The signatures for the statement-nodes for the T-Recipe are as follows:

ASSIGNMENT: string \times expr
METHOD : string \times string*
IF : charset \times stat
CASE : (charset \times stat)*
REPEAT : charset \times stat
WHILE : charset \times stat
STATLIST : stat*

6.3.2 Regular Expression to Syntax Tree

Given the syntax nodes from section 6.3.1, it is then possible to rewrite the T-Recipe to the mapping \(T: \text{RE} \rightarrow \text{STAT}\). To keep this mapping readable, a pair of commonly used statements are defined first.

\[
\begin{align*}
T_{\text{nextchar}} & \rightarrow \text{METHOD('NextChar',[])} \\
T_{\text{skip}} & \rightarrow \text{METHOD('Skip',[])} \\
T_{\text{error}}(\text{msg}) & \rightarrow \text{METHOD('Error', [msg])}
\end{align*}
\]

Using these, the mapping from regular expressions to statements then becomes:

\[
\begin{align*}
T(\epsilon, F, P) = \\
T_{\text{skip}}
\end{align*}
\]

\[
\begin{align*}
T(\alpha, F, P) = \\
1. T_{\text{nextchar}} & \quad [\alpha] = P \\
2. T_{\text{error}}(\text{'character }\alpha\text{ expected'}) & \quad \alpha \notin P \\
3. \text{IF}(\alpha, T_{\text{nextchar}}) & \quad \text{otherwise}
\end{align*}
\]

\[
\begin{align*}
T(\alpha \cdot \beta, F, P) = \\
1. T_{\text{nextchar}} & \quad \text{P} \subseteq [\alpha - \beta] \\
2. T_{\text{error}}(\text{'character }\alpha\text{ expected'}) & \quad \text{[\alpha - \beta] \cap P = \emptyset} \\
3. \text{IF}([\alpha - \beta], T_{\text{nextchar}}) & \quad \text{otherwise}
\end{align*}
\]

\[
\begin{align*}
T(e_1 \mid e_2 \mid \ldots \mid e_n, F, P) = \\
1. T(e_1, F, P) & \quad \text{P} \subseteq h(e_1, F) \\
2. \ldots & \quad \text{...} \\
3. T(e_n, F, P) & \quad \text{P} \subseteq h(e_n, F) \\
4. \text{CASE}( & \quad \text{otherwise} \\
\quad (h(e_1, F), T(e_1, F, P \cap h(e_1, F))), & \quad \text{...} \\
\quad (h(e_n, F), T(e_n, F, P \cap h(e_n, F))))
\end{align*}
\]
T(e₁ · e₂, F, P) = STATLIST([T(e₁, h(e₂, F), P), T(e₂, F, V)])

T(e₁ · e₂ · ... · eₙ₋₁ · eₙ, F, P) = STATLIST([T(e₁ · ... · eₙ₋₁, h(eₙ, F), P), T(eₙ, F, V)])

T(e∗, F, P) =
1. T_SKIP First(e) ∩ P \neq \emptyset
2. WHILE(First(e), T(e, First(e) ∪ F, First(e))) otherwise

T(e⁺, F, P) = REPEAT(First(e), T(e, First(e) ∪ F, P ∪ First(e)))

T(e¿, F, P) = CASE( otherwise
                   (h(e₁, F), T(e₁, F, P ∩ h(e₁, F)));
                   (F, T_SKIP)
                     )

T(σ, F, P) = STATLIST([T(σ₀, F, P), T(σ₁, F, V), ..., T(σ|σ|, F, V)])

Due to the recursive nature of the n-airy version of the T-Recipe for the concatenation, this definition constructs trees of statement-lists containing only two arguments. These statement-lists can be safely collapsed though. Whenever one statement-list is added to another, the two can be combined into one statement-list, provided that the order of the elements in them is maintained.

### 6.3.3 Flattening the syntax-tree

Though it is possible to give a formal definition of the flattening algorithm for the syntax tree, this leads to a very complex representation of the algorithm which does little to further ones understanding of it. Instead, the flattening algorithm will be demonstrated through a representative commented example. The process of flattening any nodes not discussed in this example is similar enough that this should not lead to problems.

The example presented here is for the regular expression \( e = ['a'-'z']^+ | ['.' ['a'-'z']^* .'] \). By applying the T-Recipe from section 6.3.2 to it, the syntax-tree displayed below is constructed. A diagram of the syntax tree is also depicted in figure 6.1.

```plaintext
CASE(
   ([ 'a'-'z' ], REPEAT([ 'a'-'z' ], METHOD('NextChar', []))
   )
   )
```

```plaintext
STATLIST(
   METHOD('NextChar', []),
   WHILE([ 'a'-'z' ], METHOD('NextChar', [])),
   IF([']'), METHOD('NextChar', []))
)
```
Each of the syntax-tree nodes can be mapped to an equivalent Pascal-statement. For the CASE, IF, REPEAT and WHILE, a suitable call to the error-method is made if none of the guards are true, specifying which characters were expected. Applying this to the example results in the code shown in listing 6.1.

Listing 6.1: Generated Example Pascal code

```pascal
case character of
  'a'..'z': begin
    repeat
      NextChar
    until not (character in ['a'..'z'])
  end;
  '{': begin
    NextChar;
    while (character in ['a'..'z']) do
      NextChar
    if (character in ['']) then
      NextChar
    else
      Error('character from set ['']' expected')
  end;
else
  Error('character from set ['']''{'} expected')
end:
```

Figure 6.1: Example Syntax Tree
6.3.4 Scanner Template

Generating the T-Recipe scanner class itself is done by filling out a T-Recipe scanner class template that contains all those elements of the scanner that do not change between implementations. Before dealing with the exact implementation details of the scanner template, it is useful to take a look at what exactly is required.

The T-Recipe scanner is based on the scanner base class from section 3.4 As such, the main method that needs to be implemented is NextSym. Additionally the scanner should return the text, position and type of tokens it recognizes in SymText, SymStart and SymType respectively. The SymText property is generally updated around calls to NextChar. Although the T-Recipe could be expanded to do this too, it is easier and clearer to override the NextChar, Mark and Recall methods in the scanner base class instead so that they keep track of the recognized text instead. This avoids unnecessary bloating of the by the T-Recipe generated code. Finally two methods are added that were needed for the T-Recipe itself, namely the Skip and Error methods, which do nothing and throw a, by the scanner specified, error respectively. This leads to the T-Recipe scanner template in listing 6.2.

Listing 6.2: T-Recipe scanner template

```pascal
unit <<modulename>>;

interface

uses SFWScanner;

type
T<<Scannername>> = class(TSFWScanner)
  private
    SymMarkedText: string;
  protected
    procedure Skip; virtual;
    procedure Error(aMsg: string); virtual;
    procedure NextChar; override;
    procedure Mark; override;
    function Recall: char; override;
<< Token method interfaces >>
  public
    procedure NextSym; override;
end;

implementation

procedure T<<Scannername>>.Skip;
begin
  {do nothing}
end;

procedure T<<Scannername>>.Error(aMsg: string);
```

61
begin
  raise ESFWException.Create(aMsg)
end;

procedure T<<Scannename>>.NextChar;
begin
  SymText := SymText + Character;
  inherited;
end;

procedure T<<Scannename>>.Mark;
begin
  inherited;
  SymMarkedText := SymText
end;

function T<<Scannename>>.Recall: char;
begin
  result := inherited Recall;
  SymText := SymMarkedText
end;

<< Token Method Implementations >>
<< Nextsym Method Body >>
end.

A number of things are needed to enter into the template. The module-
name and scanner-name can be retrieved directly from the generated factory.
The NextSym body is a simple template itself which initializes the scanner’s
symbol properties, removes whitespace, then performs the token-level selection
as described in section 6.2. The body of this is shown in listing 6.3. The body of this is shown in listing 6.3.

Listing 6.3: T-Recipe NextSym Template

{ Whitespace = set of whitespace characters }
procedure T Scanner>>.NextSym;
begin
  SymType := -1;
  while (character in Whitespace) do
    NextChar
    SymStart := Position;
    SymText := ' ';
    << token-level selection >>
  end;

end;
Through the use of the algorithms described in sections 6.3.1, 6.3.2, and 6.3.3 the token-definitions in the T-Recipe definition can be transformed into acceptors. For each token a reference to a method is added to the scanner interface and an implementation containing the acceptor in the body. The template for a token-method is simple, consisting of the generated acceptor followed by the \texttt{SymType} being set to the recognized token-type. The body for this method is shown in listing 6.4.

Listing 6.4: Token-method Template

\begin{verbatim}
procedure <<Scanner>>.Scan<<token>>;
begin
  << Flattened syntax tree of token >>;
  SymType := <<token-type>>;
end;
\end{verbatim}

6.4 T-Recipe Factory

The T-Recipe scanner factory needs to generate a scanner and add it to the current project. Though it is possible to automatically generate into a new unit or into the unit into which the scanner has been previously generated, this approach has the potential of overwriting hand-written code that the user has made. For example, if he decided to save units under different names, causing one of his own units to get the name of the unit in which the scanner used to be generated. Also, the user may wish to generate multiple implementations of the scanner or slight variations thereof.

To avoid the potential overwriting and give the user the option to specify a location, a dialog will pop up each time the T-Recipe scanner factory is activated. This dialog, depicted in Figure 6.2 gives the user the option to either generate into a new, unnamed unit, or to specify a unit from the project into which the scanner should be generated. The default options in this dialog can still be set by the T-Recipe factory to the ones it deems most likely.

After the user has made a selection, either a new unit can be generated or an older unit updated through the process described in section 8.3.

![Figure 6.2: T-Recipe Scanner Generator Dialog](image)

The T-Recipe factory provides a single published property \texttt{TRecipeDefinition} that enables the user to select a T-Recipe language definition. The value of this property is also returned by the factory in the inherited \texttt{Definition} property. Finally a call to \texttt{CreateScanner} makes a call to its \texttt{OnCreateScanner}
property to instantiate a scanner. Note that the source of this event-handler is generated by the factory and should not be altered by the user.

The public interface of the T-Recipe Scanner Factory component is then as follows:

**Public**

- `property Definition: TSFWDefinition;`
- `function CreateScanner: TSFWScanner;`

**Published**

- `property TRecipeDefinition: TTRecipeDefinition;`

### 6.5 Example

Most of the example language from section 2.5 can be recognized. An issue occurs when trying to define the predefined strings set, since the firsts of this set overlap with the first of the identifier set. As a result the det-conditions fail, thus the scanner cannot be generated. To resolve this problem, the set of predefined values must be removed before the language can be recognized. Thus the language definition becomes:

```
V = { 'a'..'z', 'A'..'Z', '0'..'9', '+', '-', '/', '*', '(', ')', '=' }
N = { digit, identifier, operator, number }
T = { identifier, operator, number }
R =
  digit → '0'..'9'
  identifier → ('a'..'z', 'A'..'Z')+
  operator → '=' | '+' | '-' | '/' | '*' | '(' | ')' |
  number → digit* · (digit | '.' · digit*)
```

To avoid filling up a few pages with the resulting scanner, only the method for the identifier acceptor is shown here. It can found in listing 6.5. Note that the currently generated code is still not entirely minimal, since the two guards of the case-statement could be combined into one.

Listing 6.5: Example Language - ScanIdentifier method

```
procedure TScanner.ScanIdentifier;
begin
  repeat
    case character of
      'a'..'z': NextChar;
      'A'..'Z': NextChar;
    else
      Error('Character from A..Z, a..z expected')
    end
  until not (character in ['A'..'Z', 'a'..'z']);
  SymType := 0
end;
```

64
Chapter 7

Automaton Scanner

Of all scanner-types, the automaton scanner is probably the best known, thanks to the popular scanner-generator LEX [Lesk75] and its descendants. Most automaton scanners construct a finite automaton out of a set of regular expressions through a variety of algorithms. More information on these algorithms can be found in [Wats94]. These automata are then used to perform the tokenization of the lexical scanner.

The automaton scanner is a scanner capable of being used as both a pre-compiled and a generated scanner. This is possible because it uses a fixed driver algorithm to interpret data stored in specially constructed tables. These tables can be hardcoded into an automaton-scanner template for the generated scanner or passed as parameters to a pre-compiled automaton-scanner.

In this chapter both the pre-compiled and generated versions of the automaton-scanner are documented, in section 7.2 and 7.3 respectively. Before the scanners are designed though, the algorithm to generate the required tables is worked out in section 7.1.

7.1 Table-generation

The process to build DFA tables from a regular language definition consists of a number of separate steps. The first step is to construct a non-deterministic finite automaton (NFA) directly from the definition. This NFA is then converted to a deterministic finite automaton (DFA), which is then minimized. From this minimized DFA the DFA tables can then be constructed. Each of these steps is detailed separately below.

7.1.1 Regular Expression to NFA

An NFA is a finite state machine consisting of states with transitions between them. These transitions are labeled with characters from the input-alphabet but may be empty as well. Given a start-state and a set of end-states, an NFA accepts a string $w$ if there exists a path from the start state $q_0$ to one of the accepting states $F$ labeled by $w$. 
Figure 7.1: Mapping of Regular Expressions to NFA
The NFA used for the automaton-scanner is a 6-tuple \((Q, \Sigma, \gamma, q_0, F, T)\):

- \(Q\): the states
- \(\Sigma\): an input alphabet
- \(\gamma: Q \times \Sigma \rightarrow \mathcal{P}(Q)\): a transition relation
- \(q_0\): a start state \((q_0 \in Q)\)
- \(F\): the accepting states \((F \subseteq Q)\)
- \(T: F \rightarrow \text{TokenType}\): a mapping from final states to token-types

An NFA acceptor can be constructed out of a regular language definition through use of an algorithm called Thompson’s Construction [Wats94]. The algorithm is depicted in figure 7.1 and consists of a recursive transformation on the elements of the regular expressions to states and transitions in the NFA. In this graph the states of the NFA are represented by circles, with double circles being the endstates. Transition are represented by labeled arrows.

An initial start-state \(q_0\) is created. Each token in the definition is then transformed using the algorithm above, an epsilon-transition is added from \(q_0\) to it’s start-state and its end-state is added to the mapping \(T\) along with its token-type.

This version of the NFA construction algorithm results in more states than absolutely necessary. Not all epsilon transitions are strictly necessary and can be removed by combining start and end-states of sections of the NFA. A more efficient algorithm that produces fewer states can be found in [Kahr99].

### 7.1.2 NFA to DFA

The main difference between an NFA and a DFA is that every state has exactly one transition out of it under every character of the input alphabet. This ensures that, for any state of the DFA and any character from the alphabet, a single new state can be determined. This property is desirable as it is computationally less complex than maintaining a set of states and calculating destination-states for transitions in an NFA.

The DFA for the automaton-scanner is a 7-tuple \((Q, \Sigma, \delta, q_0, q_s, F, T)\):

- \(Q\): the states
- \(\Sigma\): an input alphabet
- \(\delta: Q \times \Sigma \rightarrow Q\): a transition relation
- \(q_0\): a start state \((q_0 \in Q)\)
- \(q_s\): a sink state \((q_s \in Q)\)
- \(F\): the accepting states \((F \subseteq Q)\)
- \(T: Q \rightarrow \mathcal{P}(\text{TokenType})\): a mapping from states to sets of token-types

The NFA generated in section 7.1.1 can be converted to an equivalent DFA through an algorithm called the subset-construction [Hopc79]. Before the subset-construction can be discussed though, a supporting function must be defined first. The epsilon-closure of a set of states is the set of all states reachable through 0 or more \(\epsilon\)-transition from the initial set of states, i.e.

\[
\text{EpsilonClosure}(S) = \{ t | t \text{ is reachable through } \epsilon\text{-transitions from a state } s \in S \}
\]

The subset construction algorithm performs a step-wise construction of the DFA. Initially it starts with a sink-state and a DFA-state corresponding to all
start-states in the original NFA, i.e. $\text{EpsilonClosure}(q_0)$. It then checks for all characters in $V$ which transitions are possible from that initial set of states and what NFA-states these lead to, creating new DFA-states that correspond with the sets of NFA-states encountered. If a DFA-state already exists that corresponds to a certain set of NFA-states, then this DFA-state is used again instead of creating a new one. Once this is done it moves on to the next DFA-state and repeats the process until no more DFA-states are added. The resulting DFA then recognizes the same language as the original NFA.

Given an initial NFA $N = (Q, \Sigma, \gamma, q_0, F_N, T_N)$ the equivalent DFA $D = (S, \Sigma, \delta, s_0, s_s, F_D, T_D)$ can be constructed through the algorithm appearing in [Zwaa05] and shown in listing 7.1.

---

**Listing 7.1: DFA to NFA algorithm**

```plaintext
\{ construct the sink-state \}
A[0] := \emptyset

for all $a \in V$ do $\delta(0, a) := 0$ od;

\{ construct the new start-state \}
A[1] := $\text{EpsilonClosure}(\{q_0\})$;

$p, q := 1, 2$;

do $p \neq q$ →
    for all $a \in V$ do
        \{ construct the set of all NFA-states reachable through a transition $a$ \}
        $A[q] := \emptyset$;
        for all $s \in A[p]$ do
            for all $t \in \gamma(s, a)$ do
                $A[q] := A[q] \cup \text{EpsilonClosure}(t)$
            od
        od;
    \{ determine whether the new state already exists \}
    $r := 0$;
    do $A[r] \neq A[q] \rightarrow r := r + 1$ od;
    if $r = q \rightarrow q := q + 1$
    [ ] $r \neq q \rightarrow \text{skip}$
    fi;

    $\delta(p, a) := r$

od;
$p := p + 1$

od
```

Due to the nature of the algorithm, the sink-state and start-state are fixed, respectively $s_s = 0$ and $s_0 = 1$. The final states are all states that encompass at least one NFA end-state, i.e.

$$F_D = \{ t : A[t] \cap F_N \neq \emptyset \}$$

68
Since one DFA end-state might encompass multiple NFA end-states, the mapping \( T_D \) is now a mapping from states to sets of tokentypes. States that contain no final states can now be included in this mapping, as those are mapped to the empty set. The mapping \( T_D \) is

\[
T_D(s) = (\cup q : q \in F_N \land q \in A[s] : T_N(q))
\]

### 7.1.3 Minimizing the DFA

Although the DFA generated in section 7.1.2 correctly recognizes the language specified by the initial definition, it may not be the most minimal DFA to do so. To lower the storage-requirement of the DFA in memory, the DFA can be minimized to an equivalent DFA with fewer states. This is done by calculating an equivalence relation between the states of the DFA. Any states that are equivalent can be combined to reduce the size of the DFA. The algorithm described here is one that uses unordered approximation as described in [Wats95].

In short, the algorithm groups states in the DFA \( D = (Q, \Sigma, \gamma, q_0, q_s, F, T) \) in such a way that, if two states are in one group, transitions from both states under the same label lead to the same destination groups. Initially, it groups the states into one group for non-final states and one group each for equivalent end-states. Two end-states \( f_1 \) and \( f_2 \) are unique if holds that \( T(f_1) = T(f_2) \).

After the initial grouping, the algorithm then proceeds to check all transitions leading out from the states in these groups. If states in a group are found to not be equivalent with other states in the group, then these states are moved to a new group and the process is repeated. This is repeated until all groups are equivalent under all characters from \( \Sigma \).

The algorithm described here is of \( O(|Q|^2) \). Hopcroft devised an algorithm that can perform the minimization in \( O(|Q| \log |Q|) \) [Hopc71] by selecting subdivisions of the groups in such a way that only one of the resulting groups would need to be checked again. For the DFAs created for the automaton-scanners, the algorithm described here is sufficiently fast and is therefore the one used.

Given a source DFA \( D = (Q, \Sigma, \gamma, q_0, q_s, F_D, T_D) \) the algorithm to construct the minimized DFA \( M = (S, \Sigma, \delta, s_0, s_s, F_M, T_M) \) can be divided into two steps. The first step is to construct the initial grouping of the states \( Q \). This step is displayed in listing 7.2. The second step is to check the equivalence of the states in these groups and, if necessary, subdivide them further. The algorithm for this is given in listing 7.3.
Listing 7.2: DFA minimization algorithm - Initial Grouping

{ group all states that lead to the same token-types }

\( q := 0 ; \)
for all \( s \in Q \) do
\( x, y := 0, q ; \)
while \( (x \neq y) \) do
    if \( (T_D(s) = S[x]) \) \( \rightarrow y := x \)
    \[ (T_D(s) \neq S[x]) \) \( \rightarrow x := x + 1 \)
fi
od
\( \)
if \( (y = q) \) \( \rightarrow \)
    \{ make a new group \}  
    \( S[q] := T_D(s) \)
    \( P[q] := \{s\} \)
    \( q := q + 1 \)
\[ (y \neq q) \) \( \rightarrow \)
    \{ a group exists for these token-types \}  
    \( P[q] := P[q] + s \)
fi
od
\[ \]

Listing 7.3: DFA minimization algorithm

\( p := 0 ; \)
while \( (q \neq p) \) do
\( q := p ; \)
for all \( a \in V \) do
    \{ check all groups under transition \( a \) \}  
    \( r := 0 ; \)
while \( (r < p) \) do
    \( P[p] := \emptyset ; \)
    \{ select any state in the current group \}  
    \( s: s \in P[r] ; \)
    \{ and determine its destination under \( a \) \}  
    \( d := \gamma(s, a) ; \)
    \{ check all other states in the group \}  
    for all \( s \in P[r] \) do
        \{ move any states that lead elsewhere under \( a \) to a new group \}  
        if \( (\gamma(s, a)) \neq d \) \( \rightarrow \)
            \( P[p] , P[r] := P[p] + s , P[r] - s ; \)
        \[ (\gamma(s, a)) = d \) \( \rightarrow \)
            skip
    fi
od
if \( P[p] \neq \emptyset \) \( \rightarrow p := p + 1 \)
\( r := r + 1 \)
\[
\text{reconstruct all transitions ∈ the new DFA }
\]
\[
r := 0;
\]
\[
\text{while } (r \neq p) \text{ do }
\]
\[
\text{for all } a \in V \text{ do }
\]
\[
s : s \in P[r];
\]
\[
d := \delta(s, a);
\]
\[
t : d \in P[t];
\]
\[
\delta(P[r], a) = t;
\]
\[
\text{od }
\]
\[
\text{od }
\]

The start-state \(s_0\) and sink-state \(s_s\) of the minimized DFA can be obtained by finding their equivalent states in the regular DFA, i.e. \(q_0 \in P[s_0]\) and \(q_s \in P[s_s]\). The end-states \(F\) and mapping \(T_M\) can be obtained directly from the grouped states:

\[
F_M = \{t | P[t] \cap F_D \neq \emptyset\}
\]
\[
T_M(s) = T_D(q) \text{ for } s \in S, q \in P[s]
\]

### 7.1.4 DFA to table

The final step is to convert the DFA into tables that can be used by the automaton-scanners. Two tables need to be created, namely the transition-table and the endstates-table. The transition table is a mapping from states and characters to other states. The endstates table is a mapping from states to numeric values indicating which token has been recognized in that state. An additional two values must be stored for the automaton-scanners to function, namely the start-state and the error-state.

**Transitions**: array of array [0..255] of integer;

**EndStates**: array of integer;

**StartState**: integer;

**ErrorState**: integer;

Given a DFA \(D = (Q, \Sigma, \delta, q_0, q_s, F, T)\), the transition table can be constructed directly out of the transition mapping \(\delta\). For all \(q \in Q\) and \(a \in \Sigma\):

\[
\text{Transitions}[q][a] = \delta(q, a)
\]

The **StartState** and **ErrorState** follow directly out of the DFA as well, as they are \(q_0\) and \(q_s\), respectively:

\[
\text{StartState} = q_0
\]
\[
\text{ErrorState} = q_s
\]

For the end-states an ordering on the token-types is needed to decide which token is recognized if a state indicates that more than one token can be recognized. This ordering will be denoted as \(s \prec t \ (s, t: \text{Token-types})\). This ordering
indicates that token-type \( t \) takes precedence over token-type \( s \). Given this relationship it is then possible to calculate the Endstates table.

\[
\text{Endstates}[q] = \{ s | s \in T(q) \land (\forall t : t \in T(q) \land t \neq s : t \prec s) \}
\]

The exact nature of this precedence can still be chosen. The most common way is the “first come, first serve” method, where the first token in the definition to match is chosen as the return-type, for example used by LEX \cite{Lesk75}.

Note that this disambiguation step could be delayed. It is imaginable that a scanner/parser combination can obtain enough information about the context to select the token-type on a case-by-case basis, choosing which token is best suited for the situation. Since the automaton-scanner of the scanner framework is a stand-alone scanner however, this contextual information is not available and as such the decision is made at this point to avoid complicating the scanner itself.

### 7.2 Pre-compiled scanner

The first scanner-type to use the tables generated in section 7.1.4 is the pre-compiled Automaton Scanner. It uses an at component design-time implemented scanner with properties for the tables and start- and error-state. It then uses a driver-algorithm to interpret these tables and return tokens based on the DFA encoded in them.

#### 7.2.1 Automaton Scanner

The automaton-scanner is based on the scanner base class from section 3.4. The only method that still needs to be implemented is \texttt{NextSym}, which should scan the next symbol and return its text, position and type in \texttt{SymText}, \texttt{SymStart} and \texttt{SymType} respectively.

The automaton-scanner uses a simple driver-algorithm that uses the tables generated in section 7.1.4. On each iteration of the scanner, this driver starts at the start-state of the DFA encoded in the tables, then makes transitions based on the input of the scanner, remembering the last token it recognized. Once the sink-state is reached, it either returns to the position at which it last recognized a token or raises an error if no token has been recognized since starting to scan.

The algorithm for this is straightforward and is shown in listing 7.4.

### Listing 7.4: Automaton Scanner Driver

```
{ Whitespace = set of whitespace characters }
procedure NextSym;
begin
    SymType := −1;

    while ( Character in [ Whitespace ] ) do
        NextChar;

    SymStart := Position;
    text := ‘’;
    state := StartState;
```
Mark;
while (not isFinished) do begin
  newstate := Transitions[state][ord(Character)];
  text := text + Character;
  NextChar;
  if (newstate = ErrorState) then break;
  if (Endstates[newstate] <> -1) then begin
    SymText := text;
    Mark;
    SymType := Endstates[newstate];
  end;
  state := newstate;
end;
Recall;
end;

7.2.2 Factory

A published property REDefinition of type TREDefinition gives the user a way to specify which regular language definition should be used in the automaton-scanner. The method CreateTableScanner instantiates a automaton-scanner and supplies it with the tables generated from the definition.

The inherited property Definition is read-only and returns the value of REDefinition and the function CreateScanner redirects its call to CreateTableScanner to instantiate a suitable scanner.

The public interface of the Pre-compiled Automaton Scanner Factory component is then as follows:

Public
  property Definition: TSFWDefinition;
  function CreateScanner: TSFWScanner;
  function CreateTableScanner: TTableScanner;

Published
  property REDefinition: TREDefinition;

7.3 Generated scanner

The second scanner-type to use the tables from section 7.1.4 is the generated automaton scanner. It uses an almost fixed scanner-template that is added during application design-time. The tables are hardcoded into the generated scanner so that they are available during application run-time. Like the pre-compiled version it uses a driver-algorithm to interpret the tables and scan the input text.

7.3.1 Scanner Template

The scanner template for the generated automaton scanner is very similar to the scanner designed for the pre-compiled automaton scanner. The driver-algorithm in NextSym stays the same as the one for the automaton-scanner in section
7.2.1 Instead of properties for the tables that can be set from the outside through, the generated automaton-scanner initializes these from its constructor. The process to initialize these is sufficiently trivial that it does not need further explanation.

7.3.2 Factory

The factory for the generated automaton-scanner is much the same as the one for the T-Recipe scanner in section 6.4. Its actions and the GUI are exactly the same. The only place in which it differs is the public interface.

The automaton factory has a published property `REDefinition` through which the user can specify the regular language definition to use. This value is also returned by the factory in the inherited `Definition` property. A call to `CreateScanner` will instantiate the automaton-scanner through use of the `OnCreateScanner` event, which is set by the scanner-generation process.

The public interface of the Generated automaton-scanner Factory component is then as follows:

Public

property `Definition` : TSFWDefinition;

function `CreateScanner` : TSFWScanner;

Published

property `REDefinition` : TREDDefinition;

7.4 Example

The example language from section 2.5 can be handled completely by both the generated and the pre-compiled automaton-scanner. The DFA produced by the algorithms, the one encoded in the tables, is depicted in figure 7.2. Note that the sink-state has been removed from this DFA, as well as all transitions to it. The labels in the end-states mark which token has been recognized, where OP stands for operator, PRE for predefined, ID for identifier and NUM for number.

Figure 7.2: DFA of the example Language
Chapter 8

Delphi IDE Integration

8.1 The Delphi IDE

The rapid application development environment Delphi allows users to quickly construct applications by placing components on forms. These components can be selected from a large number of default palettes, ranging from labels to complex database-table editors. All components have lists of properties and events through which their behaviour in the application can be customized. Properties alter the component directly while the events supply the user with a method to react to changes of and user-interaction with the component.

Additionally, Delphi provides a number of wizards that allow the user to rapidly build complex pieces of code. Commonly, these wizards generate a project or unit that provide a framework for the user to extend, based on a set of questions that the user needs to answer.

The true strength of Delphi however, is that a developer can add additional packages containing components and wizards to the Delphi environment. These components can then be placed and configured in the same way as any of the standard components. The wizards can hook themselves into the Delphi IDE as menus and added functionality, giving users easy access to them or making life easier for them. Both components and wizards have full access to the Delphi environment, allowing them to affect almost every facet of its operations.

8.1.1 Packages

A developer can extend the Delphi IDE by registering packages with Delphi. These packages can contain components, wizards, and additional classes and procedures.

Since packages are developed in the Delphi IDE themselves, it is important to make a distinction between the various phases of development. Packages are created during component design-time. Once installed, the contents of the package are available during application design-time. Once the application has been designed and implemented, it is run in the stage called application run-time.

There are two kinds of packages: run-time and design-time packages. The former kind contains code that should be available during application design-
time and application run-time. The latter kind contains all code that should only be available during application design-time.

This division between run-time and design-time packages is for two reasons. Firstly the separation ensures that only the code needed during run-time is added to the application, thus reducing its filesize. Secondly design-time code may not always make sense during run-time, for example code that affects the Borland IDE. This code is safely separated from the application run-time environment by placing it in a design-time only package.

8.1.2 Components

Components are created by inheriting from the TComponent class or one of its descendants. Functionality can be added to it by defining methods, properties and events. By placing the component definition in a package and registering it with the Delphi IDE, the component can then be made available during application design-time.

Delphi components have a number of properties, best described as accessors to fields in the component’s class. By changing the properties in the Delphi IDE, the user indirectly manipulates these fields, making his settings available to the component. Properties can be as simple as a boolean value or as complex as other objects. For example, one can think of the boolean property ParentColor or the object property Font of type TFont, respectively.

Where a boolean value is easy to edit directly in the IDE, things like string-lists or fonts require more advanced editing interfaces. The Delphi IDE provides functionality for this by allowing the developer to create property editors and component editors, specialized interfaces to edit properties and components respectively. By inheriting from the TComponentEditor or the TPropertyEditor class, the developer can override the default edit-method of any property and supply a more suitable interface himself.

In addition to properties, a developer can also add events to new components. Events are essentially pointers to user-created methods, allowing the user to specify callback procedures that are invoked whenever specific actions occur within the component. Which actions of the component warrant an event and when they are called is decided by the component-developer. Through the use of parameters in the method-call, the component can provide additional information on the nature of or the reason for the call. Common examples of events are OnCreate and OnClick, the latter of which has parameters that specify at which coordinates the click occurred.

8.1.3 Wizards

Wizards can be added by creating a class that inherits the IOTAWizard interface before registering it with the IDE. Wizards can create projects, forms, and units, and can hook themselves into the IDE, adding options to the menu and functionality to the IDE.

All wizards access the IDE through a variable called BorlandIDEServices, which contains an object that implements the IBorlandIDEServices interface. This interface is discussed in further detail in section 8.1.4.
8.1.4 OpenTools API

The Borland Delphi IDE contains a special interface called the OpenTools API. This API allows developers to access the internal workings of Delphi, allowing him to read and modify elements of the Delphi environment as well as add custom functionality. In this section, only the subset pertaining to the scanner framework is discussed, i.e. module- and editor-related elements. For a full overview of the OpenTools API’s interfaces see [Pach01] or the Borland Delphi help-files.

All access to the OpenTools API goes through the object in the BorlandIDEServices variable. Though the IBorlandIDEServices interface it implements does not provide much functionality itself, it can be cast to a number of interfaces that provide access to the various parts of the IDE.

In this project the IOTAModuleServices interface is used most often, which can be obtained directly from IBorlandIDEServices. This interface gives access to a list of all the modules currently open. These include the project-group, any projects that may be open and any units and forms these projects may contain. All modules implement the IOTAModule interface.

There is always one project-group active in Delphi. To find this project group it is necessary to iterate over all open modules and find the one that implements the IOTAProjectGroup. The project-group keeps track of all projects open in Delphi at that time.

In general the project needed is the currently active project. This can be obtained by accessing ActiveProject of the project-group module. The returned IOTAProject in turn provides information about all the modules it contains, including ones that are not currently open. The information for each module is accessible through GetModule and is stored in an object implementing the IOTAModuleInfo interface. This interface holds basic information such as its filename and its type.

A call to OpenModule in a module-info object can open the actual module. The resulting object implements IOTAModule, which provides access to editors and allows the registration of notifiers that can keep track of the module. More information about notifiers can be found in section 8.3.2.

To edit the module, a call to its CurrentEditor property can be made to obtain an IOTAEditor. If the module is a source-module, this editor can be cast directly to a IOTASourceEditor. This interface gives access to IOTAEEditReader and IOTAEEditWriter objects that can be used to modify the content of the source-module.

Another interface that the BorlandIDEServices variable can be cast to is the IOTAEEditorServices interface. This interface allows iteration over the edit-buffers in the Delphi environment, as well as more direct access to the currently open editors. The currently active editor can be accessed through the TopView or TopBuffer properties, returning an IOTAEEditView or IOTAEEditBuffer respectively.

The IOTAEEditView interface gives access to a IOTAEEditPosition interface through its Position property. This interface allows direct manipulation of the module’s source in much the same way as the user can edit his code in the IDE. It allows editing, copy/pasting, searching and replacing of text.
8.2 Unit Creation

Adding a unit to the current project can be done by creating a class inheriting the interface `IOTAModuleCreator`. All methods in this interface are callback methods that specify what kind of module needs to be created. Most of these callbacks can return either nil or an empty string. The only ones of importance are `GetOwner` and `GetCreatorType`.

`GetOwner` should return the owner of the unit, which in our case is the project in which the scanner is being generated. This is the active project, which can be obtained through the method described in 8.1.4.

`GetCreatorType` specifies what kind of module needs to be created. Possible values for these include 'sApplication', 'sLibrary', 'sConsole', 'sPackage', 'sUnit', 'sForm' and 'sText'. Since all scanners only require a single unit, the return-value for `GetCreatorType` is 'sUnit'.

The unit can then be created through the use of the `IOTAModuleServices` interface. By passing the object describing the unit to be created to its `CreateModule` method, the Borland IDE will create a unit with a default name and a default body and add this to the current project, returning a reference to the newly created module.

8.3 Scanner Generation

In section 3.6 factory-components have been mentioned that must handle the generation of scanners. The functionality of these scanners can be broken down into the following tasks:

1. Add a unit to the project.
2. Keep track of a unit for future use.
3. Generate a scanner into a specified unit.
4. Provide an instantiation of the generated scanner.

Each of these tasks is discussed in greater detail in the following sections.
8.3.1 Adding the unit

Adding a unit to the project has already been discussed in section 8.2. Only one additional step is required since it is now for scanner-generation. Since module-names are unique throughout a project, the name of the newly created module is stored in the factory that made it. This enables it to find the unit again when the user wants to re-generate the scanner.

8.3.2 Tracking a unit

Once a unit has been created a number of things can happen. The user may save it somewhere under a different name than the one under which it was first generated. The user might close the unit without saving it, thus completely removing it from the project. He might also remove it from the project directly, even after saving it. The scanner framework must be able to handle any of these actions and still continue functioning properly.

Fortunately the Delphi IDE provides a method to keep track of modules. Through the use of an IOTAModuleNotifier call-backs can be defined that are called when some of the actions listed above are taken by the user. The actions of interest to the framework are Destroyed and ModuleRenamed.

Destroyed is called when the unit is closed in the Delphi IDE. This is not the same as it being removed from the project. The only action that needs to be taken is to remove the notifier from the module and then destroy the notifier itself. Since the notifier is reference-counted, this will happen automatically.

ModuleRenamed is called when the module is saved under a different name. To handle this, the scanner framework must change the name stored in the factory that generated the unit. This can be done by adding a reference to this factory in the notifier. The notifier can then simply change the stored value to the new module name.

Storing a direct reference to the factory is dangerous though, as the user may decide to destroy the factory component without closing the module it generated. If he were then to save the module, the notifier would try to access the factory and, since it no longer exists, will fail with disastrous consequences. This can be solved by having the factory inform the notifier when it is being deleted. Care must be taken in how this is done though.

As stated in section 8.1.1 the run-time package created should not reference the design-time package. But for the factory from the run-time package to notify the notifier from the design-time package, it would need a reference to it. Although this reference cannot be created during component design-time for the above reason, it is possible to establish this reference during component run-time. An event is added to the factory of type TNotifyEvent, a method-prototype available at all times as it is part of the standard Delphi packages. When the notifier is created, it registers a method of itself as the event-handler. Now, when the factory is destroyed, it can call the event and this in turn will inform the notifier that the factory was destroyed.

Regrettably, there is no way to re-attach the notifier when the project is closed and re-loaded. This means that when the module is saved under a different name after a reload it will not be detected. Thus the factory will no longer be able to find the module. If this happens, the user may still have to specify the unit that should be used next time the unit needs to be generated.
8.3.3 Generating the scanner

As stated in 8.1.4, modules have access to a IOTAEditWriter object, which allows write-access to the source-code through a stream paradigm, allowing the user to add source into the source-code at places while just copying the rest. Since the old source is no longer of relevance, it is discarded and completely replaced by the newly generated code.

This newly generated code can be obtained from the generators for whichever scanner the factory is working with. Parameters that are needed by these generators include the name of the unit and the name of the scanner it should generate. The source-code generated can then be streamed into the IOTAEditWriter object to be placed into the unit.

8.3.4 Instantiating a scanner

Due to its nature, a generated scanner can only be instantiated during application run-time. This can be accomplished by adding a scanner creation event to its factory. An event-handler that can instantiate the previously generated scanner is then generated during design-time. When the factory is then asked to create a scanner during run-time it can call this event-handler and return the scanner it generated.

Two steps must be made in creating this event-handler. Firstly the method must be created and connected to the event in the factory class. Secondly the units uses-clauses must be updated so that they link to the units required by the event.

Method Creation

Creation of the required method can be done through the CreateMethod method of the TComponentEditor.Designer. The exact form of the event-handler can be specified in a TTypeData record, which is then passed to CreateMethod along with the preferred name of the event-handler. On successful creation, it will return a handle to the newly created method.

To register the newly created method as the event-handler of the factory component, a call to setmethodprop must be made with the factory component, the name of the event that should be set and the handle to the method it should call. Likewise, the handle to an event-handler can be reclaimed from the factory component by a call to getmethodprop with the same arguments, except for the method of course.

Placing the required scanner-creation code in the method is done easiest through a slightly indirect route. By calling the ShowMethod method of the Designer with the method-handle, the method is opened in the editor and the cursor is placed on the first line of its body. If this is done just after creation of the method, then the required code can be inserted right there at the position of the cursor through the IOTAEditPosition interface discussed in 8.1.4.

In case the method already exists, but needs to be updated because, for example, the scanner-name has changed, then the original content of the method needs to be removed first. By enclosing all generated code in two special comments, this can be accomplished by starting at the cursor-position and seeking just past the end of the closing-comment. This block of text can then be replaced.
by the new method-body. Both actions can be done through the IOTAEdit-
Position interface as well.

Adding to a uses clause

The event-method created through this technique refers to the scanner-class in
the generated unit as well as the scanner base-class. Since both classes are in
different units, a reference needs to be added to the uses-clauses of the unit
containing the factory object. There can be at most two uses-clauses in a
Delphi unit: one in its interface section, one in its implementation section. The
reference to the generated unit must be placed in the implementation section, as
it is only referenced from within the method. The reference to the scanner-base
class unit however, must be placed in the interface section as it appears in the
parameters of the method declaration.

To be able to add units to these clauses, the framework needs to know
whether either of these uses-clauses exists and, if so, where these uses-clauses
are located in the unit. Finding these locations can be done by using a simple
scanner/parser combination. The scanner is configured to recognize the Pascal
language. The parser then uses this scanner to find the locations of the inter-
face, the implementation, and the uses keywords. Once found there are four
possibilities:

1. There were no uses keywords
2. There was no uses keyword between the interface and implementation
   keywords
3. There was no uses keyword after the implementation keyword
4. There was a uses keyword after both the implementation keyword and
   the interface keyword

When adding the scanner base-class unit, a new uses-clause must be added
directly after the interface keyword in cases 1 and 2. In cases 3 and 4, the unit
must be appended to the list of units directly following the existing uses-clause,
provided it does not exist yet.

When placing the generated unit in the uses-clause of the implementation
section, a check should first be made to see whether it already appears in a
uses-clause in the interface section. If it does not, then either a new uses-clause
must be added directly after the implementation keyword in cases 1 and 3,
or the unit must be appended to the list of units directly following the existing
uses-clause in cases 2 and 4.

8.4 Terminology

Throughout this document references will be made in relation to the Delphi
environment. The following are a number of terms that will appear in this
document:

Component Design-time This is the stage of development where compo-
nents are designed.
**Application Design-time** This is the stage of development in which the components designed during component design-time are used in applications. Whenever the word "design-time" is used in this document, the application design-time is meant.

**Application Run-time** This is the stage during where the application is run.
Chapter 9

Proof of Concept

With the scanner-framework and four scanner-types designed, it is time to put it to the test. This is done by creating a simple syntax highlighter that can use any of the scanners in the scanner framework, as well as external scanners provided they use the scanner-framework interfaces and components.

9.1 Specification

The syntax highlighter is a descendant from the TRichEdit control. It uses a scanner to tokenize its contents, displaying each token-type in a way specified by the user. This scanner is obtained through a scanner factory. This leads to the following set of requirements:

- The capability to use any scanner-framework scanner
- The ability to edit the appearance of tokens

The link to the scanner-factory is handled through a single property ScannerFactory of type TSFWFactory. The editing of the appearance of the tokens is handled through a component-editor, which is discussed in greater detail in section 9.2.

9.2 Token Attribute Editor

The GUI through which users can specify the display-settings requires a list of token-names. This list can be obtained through the definition base class as described in 3.5. A reference to this definition can be obtained through the syntax highlighter’s scanner-factory property.

For each token-name in the definition, a number of properties should be configurable. The font-style used for any token can be a combination of bold, italic, and underlined. Additionally, the colour of the token can be specified.

These considerations lead to the interface in figure 9.1.

In this interface, the list to the left contains the tokens defined in the language definition. Selecting any of these tokens displays its properties in the area to the right, along with a preview of how it will look in the syntax highlighter.
9.3 Implementation

The implementation of the syntax highlighter is straightforward. It inherits most of its functionality from the TRichEdit component. The only point at which it differs is when the user changes the text in the component. When the text changes, the syntax highlighter runs it through the scanner. For each token it encounters it takes the attributes the user supplied and applies those attributes to the text of the token.

This is demonstrated in the driver-algorithm in listing 9.1.

Listing 9.1: Syntax Highlighter Driver

```pascal
Scanner.Source := TStringStream.Create(Text);
try
  while not (fScanner.IsFinished) do begin
    Scanner.NextSym;
    if (Scanner.SymType = -1) then break;

    // select the text of the current token
    SelStart := Scanner.SymPosition - 1;
    SelLength := length(Scanner.SymText);

    // determine the attributes needed and apply them
    Attributes := TokenAttributes[Scanner.SymType];
    with SelAttributes do begin
      Style := [];
      if (Attributes.Bold) then
        Style := Style + [fsBold];
      if (Attributes.Italic) then
        Style := Style + [fsItalic];
      if (Attributes.Underline) then
        Style := Style + [fsUnderline];
      Color := Attributes.Color;
    end;
  end;
except
```

Figure 9.1: Syntax Highlighter Token-Attribute Editor
on ESFWScanException do
end;

It should be noted that, since the syntax highlighter is a proof-of-concept component, it’s error-handling is less than perfect. If it encounters an error in the input, it will fail silently, stopping its syntax-highlighting from the location of the error onward.
Chapter 10

Unit Testing

In this chapter, the various elements of the scanner framework are tested, starting with tests for the framework as a whole in 10.1 before performing correctness tests on the scanners that are contained within the framework in section 10.2.

10.1 Framework Tests

The framework tests are designed to check the correctness and robustness of the scanner framework as a whole.

10.1.1 Use-cases re-re-visited

In section 3.7 the use-cases for the scanner framework were described through numbers of concrete steps within the framework. Now that the framework is completed, these steps can be tested with the various scanners to see whether they function as they should. For the respective use-cases, these steps are:

**Direct use of a pre-compiled scanner**
- Direct use of a Paradigm scanner
- Direct use of an automaton scanner

**Direct use of a generated scanner**
- Direct use of a T-recipe scanner
- Direct use of a generated automaton scanner

**Use of a pre-compiled scanner with a scanner-enabled component**
- Direct use of a Paradigm scanner with a syntax highlighter
- Direct use of an automaton scanner with a syntax highlighter

**Use of a generated scanner with a scanner-enabled component**
- Direct use of a T-recipe scanner with a syntax highlighter
- Direct use of a generated automaton scanner with a syntax highlighter

**Use of the user’s scanner with a scanner-enabled component**
- Direct use of a handwritten scanner with a syntax highlighter
10.1.2 Robustness tests

Users are unpredictable at best and as such the framework must be robust enough to correctly handle anything a user does. The framework tests are designed to see how the framework handles itself when components are not linked correctly or when links between components are severed, either because the link was deleted or because one of the components has been deleted. They also cover incorrect use of components and attempts at scanner generation without proper definitions.

The following situations and the frameworks reactions to them need to be tested:

A component linked with another component is deleted

A component that is linked to another is removed completely. This can occur between language definition and factories, and factories and scanner-enabled components. In both cases, the component losing the link should continue functioning as if it never had the link in the first place.

A component is used without being linked to a required component

An attempt is made to use a component without another component with which it needs to be linked. These cases are when an attempt is made to generate a scanner without a language definition, when an attempt is made to create a pre-compiled scanner during run-time without specifying a language definition or when an attempt is made to create a generated scanner without first generating the scanner. In the first case, the framework should display an appropriate error-message, while in the latter cases it should throw an exception specifying that the scanner could not be created.

An incorrect definition is loaded into the regular definition editor

When a definition that does not comply to the textual representation of the regular definitions is loaded into the regular definition editor, it should display an error-message indicating where it went wrong and clear its list of tokens and expressions.

A unit is generated with a non-validating definition

Before generation of the scanner all scanner-generators using the regular definitions should validate the input. As such, this test case should result in the scanner-generators refusing to generate a unit with a message that states why not.

10.2 Scanner Tests

The scanner validation tests are designed to test the separate scanners of the framework. These encompass tests to see whether the scanners return the correct results, as well as tests to ensure that the scanners handle themselves correctly in boundary-cases.
10.2.1 Tokenization tests

The tokenization tests are designed to ensure that all scanners in the framework produce their expected results. This includes correct tokenization on correct input and error-messages on faulty input.

Empty Language

The first language to be tested is almost a boundary test itself, namely the empty language. If given any input other than whitespace characters, scanners should throw an error about encountering unexpected characters. Additionally, the scanner generators should still create compileable code.

Operator Checks

The second set of tests is designed to check each of the operators for the scanners using regular language definitions. These are the T-recipe and the automaton scanners. For each operator and terminal, a separate test is constructed containing that specific operator. Note that, in cases where the operator to be tested might produce the empty string, an additional terminal has been added to the beginning of the token to ensure that the token-definition is valid.

Token: Character = 'a'

\[ \text{'aab'} \rightarrow (\text{Character}, 'a')(\text{Character}, 'a')(\text{ERROR}) \]

Token: String = 'test'

\[ \text{'testtext'} \rightarrow (\text{String}, 'test')(\text{ERROR}) \]

Token: Range = ['a'-'z']

\[ \text{'abc;'} \rightarrow (\text{Range}, 'a')(\text{Range}, 'b')(\text{Range}, 'c')(\text{ERROR}) \]

Expression: Label = 'a'

Token: Character = Label

\[ \text{'aab'} \rightarrow (\text{Character}, 'a')(\text{Character}, 'a')(\text{ERROR}) \]

Token: Dot = 'a'.'b'.'c'

\[ \text{'abcd'} \rightarrow (\text{Dot}, 'abc')(\text{ERROR}) \]

Token: Stick = 'a' | 'b' | 'c'

\[ \text{'abcd'} \rightarrow (\text{Stick}, 'a')(\text{Stick}, 'b')(\text{Stick}, 'c')(\text{ERROR}) \]

Token: Star = 'a'.'a'*

\[ \text{'a aab'} \rightarrow (\text{Star}, 'a')(\text{Star}, 'a')(\text{ERROR}) \]

Token: Plus = 'a'+

\[ \text{'a aab'} \rightarrow (\text{Plus}, 'a')(\text{Plus}, 'a')(\text{ERROR}) \]

Token: Option = 'a'.'a'?

\[ \text{'aaab'} \rightarrow (\text{Option}, 'aa')(\text{Option}, 'a')(\text{ERROR}) \]

Token: Epsilon = 'a'.('a' | $\epsilon$)
10.2.2 Boundary Tests

The boundary tests are designed to make sure that the scanners react correctly to certain exceptional cases. This includes constructs near the end of the input. Of particular interest are those parts of the language definition that can be optional. Examples of these are the exponential parts of the paradigm language definition or a repetition near the end of a regular expression.

For the paradigm scanner a number of boundary tests can be identified. The scanner should correctly recognize all of the following tokens at the end of the input:

- A single character identifier
- A number without a decimal or exponential part
- A number without an exponential part
- Any of the end-of-line comments

For the T-Recipe scanner and automaton-scanner all regular expressions that might or might not produce an empty string must be tested. These are the repetition, the option, and the selection. They can be tested by using the tokens \( 'a' . 'a' * \), \( 'a' . 'a' ? \), and \( 'a' . ( 'a' | \epsilon ) \) respectively. In each of the three cases, the scanner should accept the tokens \( 'a' \) and \( 'aa' \) even when placed at the end of the input.

10.3 Results

The results for the unit tests can be kept short. The steps for use-case verification listed in section 10.1.1 match the implementation of the framework. The framework maintained its integrity during the robustness tests from section 10.1.2. The expected results for the tests in section 10.2 matched the results obtained from the scanners of the framework.
Chapter 11

In Closing

11.1 Framework Evaluation

At the start of this project the aim was to overcome three problems with existing scanner generators.

The first problem was that all scanner-generators were external programs, requiring the developer to switch back and forth between the generator program and his programming environment. By creating the language definitions and scanners as components and integrating them into the delphi IDE, the scanner framework was successful at solving this problem.

The second problem was that scanner-generators in general were practically command-line driven, requiring their input from a file and placing the output in another file. The scanner framework does not have this problem, since it uses components with specially designed editors to provide the required functionality for language editing and scanner creation. This allows the developer to create scanners directly in the delphi IDE through a simple and intuitive system.

The third problem was that most scanner-generators can only generate one specific scanner-type. By providing a basic scanner-class that all scanners have to implement, the scanner framework ensures that all scanners it contains can provide the same basic functionality. This basic functionality allows developers to use different kinds of scanners without having to change any code. It even allows the development of components that can work with all scanners in the framework.

Thus, as far as the overall functionality of the framework is concerned, the project was a definite success.

As far as the ease-of-use goes, the paradigm scanner leads by far. Its graphical user interface makes specifying many language definitions a matter of minutes. Unfortunately this comes at the cost of generality, as the paradigm can only recognize languages that fall within its parameters.

Although the T-Recipe scanner and the automaton scanner use the same interface for editing, the T-Recipe is slightly more difficult to use due to the conditions it places on the input. Likewise, the generality of the T-Recipe scanner suffers from the det-conditions. Finally, some expressions that can be used in the automaton-scanner are not expressible for the T-Recipe scanner.

The automaton scanner can handle most language definitions and as such
scores highest of all three scanner types on the generality. Its ease of use is lower than that of the paradigm scanner but due to the fewer restrictions on it, it is easier to use than the T-Recipe Scanner.

Regrettably the speed of the three scanners currently implemented in the framework is lower than those generated by most scanner-generators like LEX \cite{Lesk75}. This, however, can be attributed mostly to the fact that these scanners weren’t designed with speed in mind.

11.2 Conclusions

Based on the evaluation and the experiences gathered while creating the framework, a number of conclusions can be reached.

- It is feasible to integrate language-definitions and scanner-generators into the Delphi IDE. The integration into the IDE, as demonstrated in this project, makes it easier and faster to modify definitions and to re-generate the accompanying scanner.

- Design of language definitions can be facilitated through properly constructed graphical user interfaces. The developer has a better overview of the definition in progress. The interfaces can enforce correctness whenever necessary and visual feedback of errors in the definition assist in quickly locating potential trouble spots.

- Using both pre-compiled and generated scanners in the framework is possible, but, if it is required for them to be interchangeable, then this leads to limitations in the scanners. Examples of these limitations are that, in generated scanners, the token-type returned is far better off as an enumerated type than an integer, but the pre-compiled scanner cannot handle these. Also, it may be desirable to specify actions that need to be performed upon recognition of a token, but pre-compiled scanner cannot handle this. Although it would be possible to still use actions in generated scanners, it would lead to a separation in the language definitions, which would essentially split the framework into a pre-compiled and a generated part.

11.3 Personal Evaluation

If one were to ask whether I feel that I’ve made a massive breakthrough with this project, then the answer would have to be no. However, I do feel that I’ve proven the viability of an integrated scanner- and parser-framework for the Delphi IDE. The process as described in this document works and, with further research, could be used to create very easy to use compiler-generators. As such, it is my hope that this project forms a first step towards a larger framework for language- and parser-design that is fully integrated into the Delphi IDE.

As with any project, there are plenty of “had been” and “should have” situations. Had I stuck to the toolkit for abstract and concrete syntax, would I have created something more revolutionary? Had I been more focused, it would’ve been done quicker. I should have spent more time on some things and less on others. Had I made certain decisions differently, would it have been
better? Most of these thoughts are useful though, as I’ll never know the answer unless I were to do it all again.

There will always be a few regrets. For one, the automaton scanner could have used a slightly more advanced language definition to handle certain fringe cases that make language-design in it more difficult now. As far as proving that the scanner framework can work, it proves its purpose though, so it’s a minor regret. Another regret is that I didn’t really learn until close to the end that having a list of things you’re going to do that day ensures a better motivation than vague goals for that day. This regret is only about the past though, because now that I know it works, I can use it for future projects as well.

Ironically enough it is only at the end of the project that I realise one possible reason why so few graphical interfaces exist for compiler-generators. Many of the current compiler-generators are created by initially developing a small, simple version of the generator, which is then used to create a better, more powerful version through a process called bootstrapping. Since the input comes from text-files, each successive version can have a more powerful language definition without requiring additional manual labour to make this work. If the compiler-generator has a graphical user-interface, this is no longer possible as the user-interface will have to be manually updated each time the bootstrap process is repeated.

11.4 Further Research

There are several areas in which further research can be made based on this project.

**An integrated scanner- and parser-framework:** a re-design of the scanner framework, firstly to separate it into a pre-compiled and a generated part and secondly to combine it with scanners and/or scanner-generators. The former to avoid one type of system hindering the other, the latter to complete the framework’s functionality.

**Visual language editing:** the Development of a set of graphical editors for language definitions. With this I don’t necessarily means ones for the framework suggested in the previous option, but for language definitions in general. Scanner-generators like LEX could benefit from proper user interfaces as well.

**Construction and alteration of source code:** a system to allow easy creation and editing of Pascal source-code. Among the functionality would be the ability to add and remove classes and methods, editing the contents of method-bodies and adding and removing units from the uses-clauses.
Appendix A

List of Terms

**Application Design-time** This is the stage of development in which the components designed during component design-time are used for the application design.

**Application Run-time** This is the stage during where the application is run.

**Component** Visual and non-visual controls and elements that can be placed and configured directly within the Delphi environment.

**Component Design-time** This is the stage of development where components are designed.

**Design-time Package** A set of wizards, component-editors and units that are available during application design-time, but not during application run-time.

**DFA** (deterministic finite automaton) A finite state machine where for each pair of state and input character there is a deterministic next state.

**Generated Scanner** A scanner that is placed in a unit in the active project at application design-time and will become available at run-time.

**NFA** (nondeterministic finite automaton) A finite state machine where for each pair of state and input character there may be several possible next states.

**OpenTools API** An interface that can be used by wizards and component-editors to access the underlying functionality of the Delphi IDE.

**Pre-compiled Scanner** A scanner that has been creation during component design-time and as such is already compiled during application design-time.

**Run-time Package** A set of components and units that are available during application design-time and application run-time.

**Scanner-enabled Component** A component that uses a scanner to perform part of its functionality.

**Wizard** A tool used in programs to quickly and easily create complex program elements or perform complex operations through the use a list of settings.
Bibliography


[Gamm95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software,” (Addison-Wesley Professional, 1995)


