

test

edser

July 2, 2018

Chapter 1

Project Description

1.1 Introduction

Developers all over the world use their favourite editors in order to software. One of the things all these editors tend to have in common is support for syntax highlighting. Syntax highlighting is the process of colouring certain words or sections of text in a document in order to give visual meaning to them. This is useful since humans can easily identify things by colour and the colours give an overall more structured look to an otherwise plain looking text document. Creating programs that do this highlighting for us is a development task in its own. The syntax highlighters tend to use (some form of) the underlying mathematical formalism of a programming language called the Context-Free-Grammar of this language. This is necessary in order to generate correct context-aware highlights. There are different forms of syntax-highlighters, some work with states and a simple stack much like finite state machines. Others, like the Rascal highlighter in Eclipse, use the Grammar and its parse-trees directly. Just as regular languages can not express everything context-free languages can, state-based highlighters cannot express everything that the context-free languages can. Hence, a translation step is needed.

1.2 Problem Definition

Creating state-based syntax highlighters for different languages is a repetitive and seemingly similar process. Many editors (e.g., VS Code, Textmate, SublimeText, Atom, ACE, CodeMirror etc.) use some implementation of these state-based highlighters. This thesis is focused on generating such highlighters from the context-free definition of the language. As suggested by my supervisor I took an algorithm for transforming a context-free grammar to a regular approximation as a starting point (Mohri and Nederhof, 2000). From here on this thesis will describe how to solve the following question.

How can we derive state-based highlighters from context-free grammars?

The algorithm was written for Rascal and this thesis will use Rascal as the main tool to fit the algorithm to, however it is described in a general way such that it is extendable to different languages. The ultimate goal would be to be able to generate a proper highlighter for Rascal's own grammar. Which contains close to all special things you can do with Rascal's grammar formalism. A second important portion of the goal is to be able to deal with tougher highlighting tasks like nested comments and string interpolation.

1.2.1 Steps

In order to tackle this problem the main question is divided in a number of subquestions. Each being fairly detachable from the problem as a whole, but contributing to the final result when joined with the rest.

1. How to embed highlighting information in a context-free grammar?
2. Can a Rascal grammar be written as the standard 4-element tuple?
3. How to convert a context-free grammar to a regular grammar approximating the same language?
4. How to create useful state-machines from these approximations?
5. How to map these machines to an actual state-based syntax highlighter?

1.2.2 Performance

The designed algorithm is build logically and works for relatively simple cases. Once larger grammars and more difficult cases are tested a certain number of errors become apparent. The origin of these errors are identified and methods to avoid them are presented where possible.

Chapter 2

Background Information

2.1 Graph

2.1.1 Definition

A graph is defined as a structure with a set of nodes called vertices (V) and a set of links between these nodes called edges (E). By this definition a graph (G) becomes a two-element tuple of the following form.

$$G = (V, E) \quad (2.1)$$

Traversing a graph from some vertex $v1$ to some $v2$ along the edge $(v1, v2)$ is written as:

$$v1 \xrightarrow{(v1, v2)} v2 \quad (2.2)$$

A path with an arbitrary number of edge-traversals leading from $v1$ to $v2$ is written as:

$$v1 \rightarrow^* v2 \quad (2.3)$$

2.1.2 Strongly Connected Components

A directed graph is said to be *strongly connected* if every vertex $v \in V$ is reachable from any other vertex $v2 \in V$. Here $v1$ and $v2$ can be the same vertex.

$$v1, v2 \in V : v1 \rightarrow^* v2 \quad (2.4)$$

The set of *strongly connected components* of a directed graph G is defined as the partitioning of G into smaller graphs, such that all of these subgraphs are strongly connected.

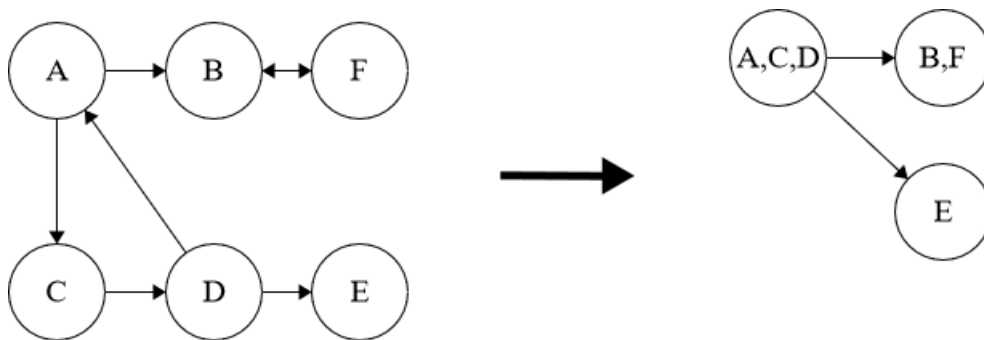


FIGURE 2.1: The strongly connected components of a graph

Kosaraju's algorithm

Numerous algorithms for computing the strongly connected components of a directed graph exist. Most of these also have nice time-complexities. Kosaraju's algorithm works in $\mathcal{O}(V + E)$. Officially published by (Sharir, 1981), it uses the property of the inverse of a directed graph having the same strongly connected components as the original graph. Below follows a brief description of the algorithm. A few things beforehand:

1. A component is represented by assigning every vertex a root vertex of its component.
2. The algorithm requires an ordered list L of vertices. It will grow to contain each vertex once.

The algorithm then becomes:

1. For each vertex $u \in V$, mark u as unvisited. let L be empty.
2. For every vertex $u \in V$ *Visit*(u), where *Visit*(u) is:
 - if u is unvisited:
 - Mark u as visited
 - For each out-neighbour v of u , *Visit*(v)
 - Prepend u to L
3. For each element u in L (in order), *Assign*(u, u), where *Assign*($u, root$) is:
 - If u has not been assigned to a component:
 - Assign u as belonging to the component whose root is $root$
 - For each in-neighbour v of u , do *Assign*($v, root$)

2.2 Context-Free Grammar

2.2.1 Definition

A context-free grammar (CFG) is a 4-element tuple. It has two disjoint alphabets V and Σ , a set of production rules P and a startsymbol $S \in V$. A production rule p is a pair (A, w) with $A \in V$ and $w \in (V \cup \Sigma)^*$. The set Σ is the set of terminals and V the set of non-terminals. Empty sentences are defined by ϵ

$$CFG = (V, \Sigma, P, S) \quad (2.5)$$

$$p \in P, A \in V, A \rightarrow w \in (V \cup \Sigma)^* \quad (2.6)$$

2.2.2 Graph of a CFG

The graph of a context-free grammar is defined as follows. It is a directed graph with vertices for each non-terminal in CFG and an edge from a non-terminal $A \in V$ to some other $B \in V$ iff B is in w for some production rule $A \rightarrow w$.

$$\begin{aligned} CFG &= (V_{cfg}, \Sigma, P, S) \\ E &= \{(A, B) | (A \rightarrow w) \in P, B \in w, B \in V_{cfg}\} \\ G &= (V_{cfg}, E) \end{aligned} \quad (2.7)$$

2.2.3 Strongly Regular Grammars

Normal context-free grammars can generate languages that are not regular. In other words, these can not be mapped to equivalent finite state machines. The subclass of grammar called *strongly regular grammars* are the set of grammars guaranteed to generate regular languages. This set is also the set of grammar without self-embedding (Chomsky, 1959).

Strongly regular grammars are grammars in which the rules of each set M of mutually recursive nonterminals are either all right-linear or all left-linear. All non-terminals $T \notin M$ are considered terminals here. For the sake of this thesis I chose to use the case of right-linear grammars and rewrite all conflicting rules to right-linear ones. Mohri and Nederhof, 2000 describe a simple transformation to transform any grammar into a strongly regular one. The new grammar accepts a superset of the language of the original grammar.

1. Identify all sets M of *mutually recursive non-terminals*¹ such that not all $p : (A \rightarrow w), A \in M$ are right- or left-linear with respect to other elements of M .
2. For all non-terminals with conflicting productions $A \in M, p : (A \rightarrow w)$ introduce a new non-terminal $A' \notin V$.
3. If A is directly reachable from some other non-terminal $X \notin M$ introduce a rule $A' \rightarrow \epsilon$.
4. For all rules with left hand side $A \in M$:

$$A \rightarrow \alpha_0 B_1 \alpha_1 B_2 \dots \alpha_m B_m$$

with $m \geq 0, B_1, \dots, B_m \in M, \alpha_0, \dots, \alpha_m \in (\Sigma \cup (N - M))^*$. Replace with:

$$\begin{aligned} A &\rightarrow \alpha_0 B_1 \\ B'_1 &\rightarrow \alpha_1 B_2 \\ B'_2 &\rightarrow \alpha_2 B_3 \\ &\dots \\ B'_{m-1} &\rightarrow \alpha_{m-1} B_m \\ B'_m &\rightarrow \alpha_m A' \end{aligned}$$

5. if $m = 0$, we end up with just $A \rightarrow \alpha_0 A'$.

This transformation ensures that all rules of the grammar become right- or left-linear with respect to their set of recursively reachable symbols. This ensures a regular grammar and therefore that it is possible to convert this new grammar into state machines

¹This coincides with the strongly connected components of the grammar

2.3 State Machines

2.3.1 Definition

A deterministic finite state machine (DFSM or DFA) S is a 5-element tuple with a set of states (Q), an alphabet (Σ), an initial state ($q_0 \in Q$), a set of final states ($F \subseteq Q$) and a transition function ($\delta : (Q \times \Sigma \rightarrow Q)$) describing how to move from one state to the next. Alongside this we define a transition as a 3-element tuple.

$$S = (Q, \Sigma, q_0, F, \delta) \quad (2.8)$$

$$Tr = (q_1, a \in \Sigma, q_2) \quad (2.9)$$

For a non-deterministic variant (NFSM or NFA) we make 2 small changes.

1. ϵ is allowed as an element of Σ . This was not allowed for a DFA.
2. δ becomes a function to the power set of Q : $(Q \times \Sigma) \rightarrow \mathbb{P}(Q)$

2.3.2 Machines for Strongly Regular Grammars

You can construct finite automata for strongly regular grammars. You can either generate one (potentially enormous) machine, or go with a more compact representation. The general steps of the algorithm for this compact version can be sketched as follows:

1. Determine the sets of mutually recursive non-terminals.
2. For each set M create one machine $\mathcal{K}(M)$ using the classical construction of an automaton from a grammar. $\mathcal{K}(M)$ is now an NFA for which we have left the starting state unspecified. Here are all non-terminals that are not part of M seen as terminals and put on transition-arcs.
3. To obtain a machine that accepts string generated by some non-terminal A we just pick the machine $\mathcal{K}(M)$ for which $A \in M$. Now we choose the state corresponding to A as the starting state (q_A). This is the machine accepting the language of A .
4. To accept the entire language find the following machine:

$$\mathcal{N}(S) = \mathcal{K}(M), S \in M, (q_0 = q_S \in Q)$$

Now while processing input strings w we lazily substitute new machines into $\mathcal{N}(S)$ once we encounter non-terminals outside $M : S \in M$.

2.4 State-Based Syntax Highlighters

2.4.1 Definition

Different editors tend to use different formats for state-based highlighters. Although much of the structure and elements are similar. This thesis is focused on working with Sublime Text as a first try. This was chosen for convenience of availability and clarity of a so-called `.sublime-syntax` file.

A state-based syntax highlighter is basically a simple automaton. It has states, sentences determining where to go next and a starting state (*main*). There is the addition of a stack to the machine. This stack is for keeping track of which state(s) the highlighter is in. Transitioning from one state to another, means replacing the current state with the next state on the stack. This operation is called *set*. Besides this action we have three additional options. 1. *push* the next state on the stack, 2. *pop* the current state off of the stack, 3. *push* or *set* a list of states. This last one means simply pushing a list of contexts, or for the *set*-action it means replacing and pushing. Sadly there is no *pop n* states from the stack.

The states of this 'state machine' are called *contexts*. These are state like structures which *match* a number of regular expressions. These cause action (like changing context or colouring the match). The type we assign to our matches are called *scopes*. This is a highlighting scope. These are predefined and help all editors and different colouring themes to still produce correct results with the same highlighter. The combination of a regular expression and assigning contexts or scopes are appropriately called *matches*. *Variables* are predefined regular expressions that can be reused throughout different contexts.

In formal terms this becomes, where yet to be defined terms will be explained in the rest of this section:

$$\begin{aligned}
 \text{Highlighter} &= H \\
 \text{Contexts} &= C \\
 \text{Matches} &= M \\
 \text{Scopes} &= S \\
 \text{Variables} &= V \\
 \text{Actions} &= A \\
 \text{main} &\in C \\
 H &= (\text{name}, \text{file_extensions}, V, C, \text{main}) \\
 (c \in C) &\Rightarrow c = (M' \subseteq M, s \in S) \\
 (s \in S) &\Rightarrow s \in (\text{meta_scope}(\text{scope_name}), \text{scope}(\text{scope_name}), \text{null}()) \\
 (m \in M) &\Rightarrow s = (\text{regex}, s \in S, a \in A) \\
 (a \in A) &\Rightarrow a \in \{\text{set}(S' \subseteq S), \text{push}(S' \subseteq S), \text{pop}(), \text{noact}()\}
 \end{aligned}$$

Below are the namings and actions of two editors and their syntax-definitions shown. They both show an example of a highlighter producing equivalent results. The just defined names are closely related to how Sublime text works, because that has been the main editor used in this project. Therefore the highlighter information on Text-Mate is also more bare.

2.4.2 The TextMate JSON-like Syntax Definition

In TextMate² you make use of a *JSON*-like definition as seen below. This highlighter highlights two things. The words 'for', 'while', 'return', 'if' are highlighted as keywords. And everything from an opening double-quote up till the corresponding closing one is highlighted as if it were a string. Except for characters that are escaped by a backslash. These get a special colouring for constants.

As can be observed below contexts are named patterns for TextMate. They have either simple matches, with a scope assigned to *name* or a 'begin' and 'end' match. This corresponds to: match 'begin' → push nested 'patterns' (contexts) on the stack. Then on matching the 'end' you pop the nested context(s) off of the stack. The actual presence of the stack is more concealed for TextMate. For these second type of matches, if the match has a scope specified, all characters that are passed whilst being in this context will be highlighted with that scope (including the begin and end matches, however this is adjustable).

A TextMate highlighter

```
1 {  scopeName = 'source.untitled';
2    fileTypes = ( );
3    patterns = (
4      {  name = 'keyword.control.untitled';
5         match = '\b(if|while|for|return)\b';
6      },
7      {  name = 'string.quoted.double.untitled';
8         begin = '"';
9         end = '"';
10        patterns = (
11          {  name = 'constant.character.escape.untitled';
12             match = '\\.';
13          }
14        );
15      },
16    );
17 }
```

LISTING 2.1: A TextMate example highlighter

²source: *Language Grammars*

2.4.3 The Sublime YAML-like Syntax Definition

As seen in the files below there are some differences between Sublime³ and TextMate. However they also have a lot in common. The biggest difference may be that Sublime does not have the nice 'begin' and 'end' matches which make multiline matches much easier. In Sublime you have to do this yourself. This can cause problem when working with multiline portions like comments. Beside that, the starting context is always named "main"

A few more things can be done with the Sublime highlighters (many of these are also available in some form in TextMate). First of all the "- include:" statements. These statements allow to be in multiple contexts at a time. It can be seen as a union of this context with the included one. The formal definitions of the highlighter H and contexts C could be extended by adding a set of *includes* to the tuple. Secondly, just like with TextMate you can nest contexts and make them nameless. Finally there is the concept of the *prototype* scope. This is a scope that is always included in every other scope (unless specified otherwise). This is useful in for example highlighting comments, these can occur nearly everywhere. This could also be added to the definition of H .

Just like TextMate you can have everything that is encountered whilst being in a context highlighted with the same colouring. This is done through assigning a so-called meta-scope to the context.

Line endings

In Sublime the regular expressions are always matched against a single line. This can cause trouble in a number of situations. Imagine having a context that processes whitespace without doing anything with it. It should pop itself off of the stack once there is no more whitespace:

```

1  Whitespace:
2    - match: '[\n\t\ \f\r]'
3    - match: '?!([\n\t\ \f\r])'
4    pop: true
5
```

This seems to be fine. It keeps matching whitespace until it no longer matches whitespace (using lookahead so it does not consume the matched character). Then pops itself of. However in the code below this will produce unexpected results. after the opening '{' it will find and parse the space and newline, match these and do nothing. Now the following characters will be a tab, however the highlighter is now right before the line-ending (\$). This will fail the first match and pass the second match, since it matches not whitespace following the caret, and pop off the context. This type of behaviour should be taken into account.

```

1  if (condition) {
2    Statement;
3  }
4
```

³source: [Documentation: Syntax Definitions](#)

A Sublime highlighter

```
1 %YAML 1.2
2 ---
3
4 name: Untitled
5 file_extensions: []
6 scope: source.untitled
7
8 contexts:
9
10   main:
11     - include: Keywords
12     - include: String_begin
13
14   String_begin:
15     - match: '"'
16       push: String
17
18   String:
19     - meta_scope: string.quoted.double.untitled
20     - match: '\\\.'
21       scope: constant.character.escape.untitled
22     - match: '"'
23       pop: true
24
25   Keywords:
26     - match: '\b(if|while|for|return)\b'
27     scope: keyword.control.untitled
```

LISTING 2.2: A Sublime example highlighter

Chapter 3

The Algorithm

3.1 Approach

The algorithm can be divided in a number of parts that will be discussed during the coming sections. These steps need to be taken in order to complete the process of converting a *CFG* to a syntax highlighter. This algorithm was build with Sublime Text in mind, however it is easy to make extensions or modifications to it in order to make it work for other editors.

The idea is to convert the grammar to its regular approximation using the algorithm presented in Mohri and Nederhof, 2000. Then turn this approximation into the corresponding state machines. With these machines, the states can then be mapped to contexts and from those a highlighter is created. A few things to consider here are:

1. How to preserve highlighting information found in the original *CFG*?
2. Do the transformations cause information loss that is essential for correct highlighting?
3. Does the final result do a good (enough) job at highlighting?

3.2 The Algorithm's Pipeline

For embedding highlighting information this algorithm needs information per token in the right-hand side of a production rule. This is a specification of the scope to use for highlighting these tokens. This allows for the same non-terminals being coloured differently depending on the context (Think of identifiers as function names or as simple variable names in *Java* or *C*, the first is blue and one is white). Below is a simple example of what a rule could look like. Indicating that all elements on the right-hand side should be coloured with the named scope. The specification could either be a list of scopes, a single scope, or no scope(s) at all.

$$String \rightarrow @Context = "string.quoted.double" \quad " \quad StringChar \quad "$$

Any form that indicates information about the tokens on the right-hand side of a rule will be sufficient. For this project the above notation will be used with the following rules:

1. The "@Context" tag is followed by a space-separated list of scopes enclosed by a pair of double quotes. There is one scope for each token on the right hand side.
2. If no context is specified, no tokens will be coloured.
3. The token *null* means no colouring.
4. If there are less than N scopes specified, where N is the number of tokens on the right-hand side, then the last scope in the list will be used for the remaining items.

3.2.1 Example grammar

The upcoming sections describe the different steps to take in the algorithm. For illustrative purposes the following grammar will be used. This is a plain context-free grammar and these steps will for now be untailored to specific programming languages.

```

1 start Statement -> Id ":" Literal
2           -> @@Context="keyword.control.flow null">@
3             "if" "(" Condition ")" "{" Statement "}"
4
5 Condition -> Id ">" Id
6           -> Id "<" Id
7           -> Id "==" Id
8
9 Literal   -> String
10          -> Integer
11
12 Integer  -> @@Context="constant.numeric.integer">@ [0-9]+
13 Id       -> [a-z]+
14 String   -> @@Context="string.quoted.double">@ "\"" !["\"]* "\""

```

3.2.2 Simplifying the grammar

The first step is to reduce any grammar formalism used to the standard 4-element tuple as defined in 2.2.1. Most programming languages do not have such a clean implementation of context-free grammars. In order to make any implementation work it is essential to convert the input grammar to the default form. Do make sure to keep the information about the scopes. What to do here is different per programming language. How this works for Rascal will be discussed in the chapter on implementation (4).

3.2.3 Grammar to Graph

After the grammar is clean, compute the graph of the grammar implementing the method described in 2.2.2. Since the grammar is reduced to (V, Σ, P, S) it is a simple loop over all elements of P to compute this.

Example

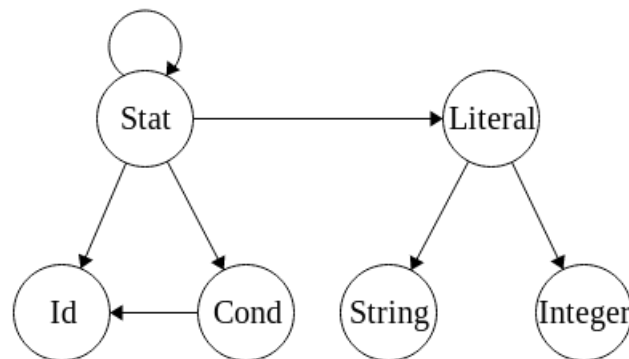


FIGURE 3.1: The constructed graph of the grammar

3.2.4 CFG to Strongly Regular Grammar

To make this conversion use the algorithm described in 2.2.3. First take the graph obtained from the previous step and compute its strongly connected components using Kosaraju's algorithm (2.1.2). These components correspond to the mutually recursive sets of symbols that Mohri's algorithm uses. Since the new symbols that are introduced by the transformation correspond to the ending of the original symbols, A' can be viewed as something like A_{end} .

Example

The strongly connected components of the graph is the same as the graph itself since it contains no cycles. Only the self loop of *Statement* would be gone. Therefore it is not included again. The strongly regular version of this grammar is shown below. There is only one component in violation of the either all left- or right-linear production rules. These are the rules of *Statement*. An extra non-terminal is introduced and the transformation is performed.

```

1 start Statement -> Id ":" Literal Statement_end
2               -> "if" "(" Condition ")" "{" Statement
3
4 Statement_end ->
5               -> "}" Statement_end
6
7 Condition     -> Id ">" Id
8               -> Id "<" Id
9               -> Id "==" Id
10

```

```

11 Literal      -> String
12              -> Integer
13
14 Integer      -> [0-9]+
15 Id           -> [a-z]+
16 String       -> "\"" ![""]* "\""

```

3.2.5 Strongly Regular Grammar to Automaton

Now that the grammar is converted into a strongly regular one accepting a superset of the original language, the algorithm defined in 2.3.2 can be applied. For this first recompute the set of mutually recursive non-terminals, since the grammar has changed since the earlier step, so the components have as well. Now the described algorithm can be used to produce a set of NFA's, for each recursive set S one machine. These machines can have the following tokens on their arcs: $Token \in (\Sigma \cup (V - S))$.

Following this, the created NFA's should be converted to DFA's, since highlighters cannot work properly with ϵ -transitions and other non-determinism. In fact the Sublime highlighter simply chooses the first match that fits and since an epsilon match always matches, this can produce very wrong results. In cases of non-determinism it will never reach the second possible match. This is why converting the machines to DFA's is an important step. This step ensures however that we loose the property of being able to identify multiple non-terminals with one machine. This is because generating a DFA from an NFA is dependent on the start state (at least in case of the powerset construction). After this, every non-terminal has its own specific DFA. The default powerset construction works here even though it generates a possible $\mathcal{O}(2^n)$ states. The machines tend to have lots of ϵ -transitions, so it is a good idea to cache the results of computing ϵ -closures. This ensures that you never compute the ϵ -closure for a given state more than once.

Example

The new components become the following graph. It just adds the *Statement_end* to the graph.

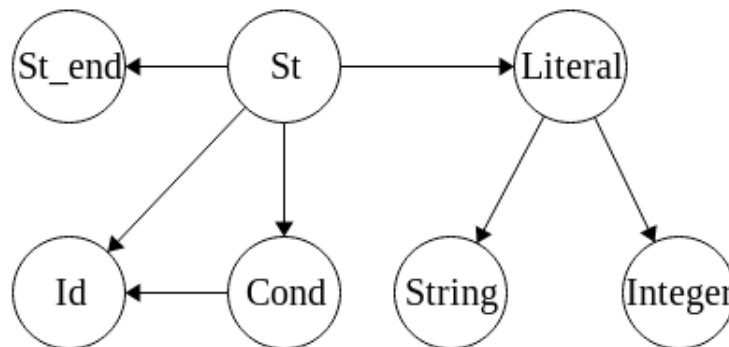


FIGURE 3.2: The graph of the strongly connected components of the example grammar

The machine showed below is the machine generated from the *Statement* non-terminal. As seen below all non-terminals that are not part of the same strongly connected component are viewed as terminals and put on arcs here. This is because there is certainty that this machine will not be reached again from within that machine such that infinite recursion could occur. This *DFA* is a slightly more optimized version than what would have been obtained from following the described algorithm strictly. The other machine would have been larger, because naive powerset construction would embed the entire machine a second time due to the choice of the start state.

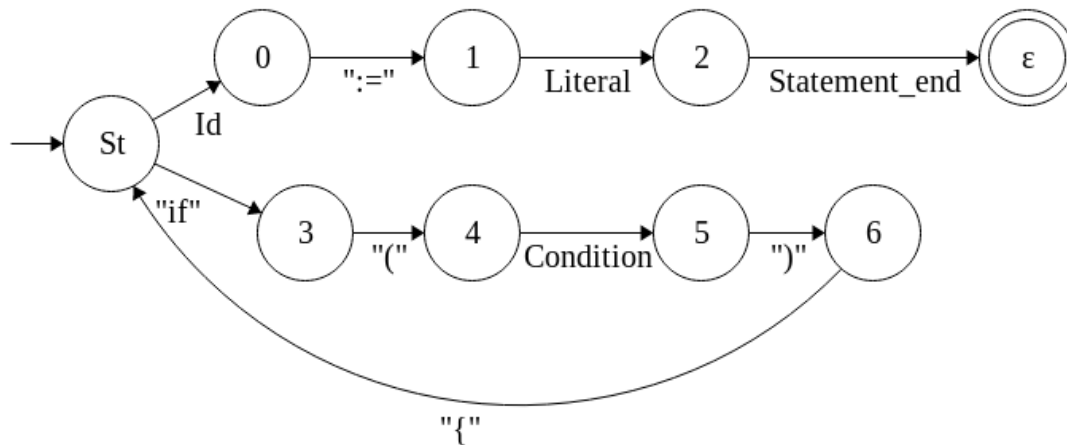


FIGURE 3.3: The DFA for the non-terminal Statement of the strongly regular grammar

Conflict identification and resolution

Now creating all these *DFA*'s helps with resolving much of the non-determinism for the syntax highlighter. However, there are still conflicts possible. As figure 3.4 below shows, conflicts can still arise if a machine has an outgoing arc with a terminal and one with a non-terminal. If the corresponding machine of this non-terminal has the same terminal on an arc from its initial state then a choice conflict arises.

The simplest fix for this is substituting the entire conflicting machine into the parent-machine. After this, rerun the *NFA2DFA* algorithm. If conflicts remain, repeat. There are a number of functions needed in order to identify and fix the conflicts. Of which the most important ones are illustrated below. These functions ensure that conflicts are solved, however can take very long to complete. When working with smaller grammars, like DSL type of languages, these functions works fine. When working with larger grammars, where many conflicts arise, this process can take quite a while. The generated machines can also grow very large, in the order of tens of thousands states.

To make this process more efficient, construct a tree-like structure of the connected components of the strongly regular version of the grammar. This is guaranteed to be a tree, because the connected components automatically join all cycles into one node, there can be multiple edges going into the same node though. Now use depth-first search from the start symbol's machine and resolve all conflicts in the leaf nodes first and work upwards. This prevents having to solve the same conflict in the same machine multiple times.



FIGURE 3.4: Machine B has a conflict on encountering a terminal 'a'

Example

As the original example grammar did not contain any conflicts, the figure above will be used as an example. It shows two machines, one for the non-terminal A , and one for the non-terminal B . Now, for the output of the algorithms and functions below:

$$\text{alphabetOfState}(A, M(A)) = \{a, c\}$$

$$\text{alphabetOfState}(\epsilon, M(A)) = \{\}$$

$$\text{alphabetOfState}(B, M(B)) = \{a, A\}$$

$$\text{alphabetOfState}(\epsilon, M(B)) = \{\}$$

$$\text{firstTerminalOf}(M(A), \{A, B\}, \{M(A), M(B)\}) = \{a, c\}$$

$$\text{firstTerminalOf}(M(B), \{A, B\}, \{M(A), M(B)\}) = \{a, c \text{ (from } M(A)) \ a \text{ (from } M(A))\}$$

$$\text{findConflicts}(M(B), \{A, B\}, \{M(A), M(B)\}) = \{(B, A, \epsilon)\}$$

1 conflict in $M(B)$ being : from B , taking A , to ϵ

$$\text{findConflicts}(M(A), \{A, B\}, \{M(A), M(B)\}) = \{\}$$

0 conflicts in $M(A)$

Below are the resulting machines shown that come from the algorithm. Left is the NFA obtained from only replacing the transition with the entire machine. Right is the DFA that is obtained from this NFA. With that, the DFA is the output of $\text{solveConflicts}(M(B), \{M(B), M(A)\}, \{A, B\}, \{(B, A, \epsilon)\})$.

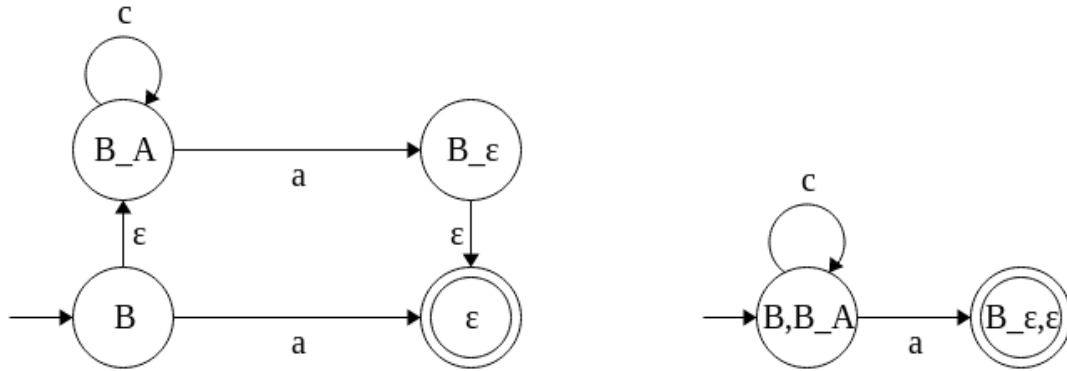


FIGURE 3.5: The NFA obtained from replacing the conflicting transition and the DFA obtained from the NFA.

Algorithm 1 Resolve Conflicts

```

1: function SOLVECONFLICTS(dfa, allDfas, nonTerminals, conflicts)
2:   for all {tr | tr ∈ conflicts} do
3:     nfa ← replaceTransitionWithMachine(tr, dfa)
4:     dfa ← NFA2DFA(nfa)
5:     newConflicts ← findConflicts(dfa, nonTerminals, allDfas)
6:     if newConflicts ≠ {} then
7:       return solveConflicts(dfa, allDfas, nonTerminals, newConflicts)
8:     else
9:       return dfa
10: end function

```

Algorithm 2 Find Conflicts

```

1: function FINDCONFLICTS(dfa, nonTerminals, allDfas)
2:   conflicts ← {}
3:   for all {state | state ∈ dfa.Q} do
4:     tokens ← alphabetOfState(state, dfa)
5:     ntsOfState ← {nt | nt ∈ tokens ∧ nt ∈ nonTerminals}
6:     terminalsOfState ← tokens − ntsOfState
7:     for all {nt | nt ∈ ntsOfState} do
8:       nextTerminals ← firstTerminalOf(allDfas[nt], nonTerminals, allDfas)
9:       if (nextTerminals − terminalsOfState ≠ {}) then
10:        conflicts ← conflicts + (state, nt, dfa.δ(state, nt))
11:   return conflicts
12: end function

```

Algorithm 3 Get the terminals reachable from the first state of a machine

```

1: function FIRSTTERMINALOF(dfa, nonTerminals, allDfas)
2:   tokens ← alphabetOfState(dfa.q0, dfa)
3:   terms ← tokens − nonTerminals
4:   result ← terms
5:   for all {nt | nt ∈ (tokens − terms)} do
6:     result ← result + firstTerminalOf(allDfas[nt], nonTerminals, allDfas)
7:   return result
8: end function

```

3.2.6 Mapping machines to contexts

Once all non-terminals have their own conflictless machine it is possible to make a deterministic mapping from the machines to contexts and respective scopes. The general idea is to define a datatype corresponding to a state-based syntax highlighter. An algebraic datatype works very well because of the nested structures of a highlighter. Once the machines are translated into this datatype, define different methods on this datatype in order to write out highlighters for different editors. One general *SyntaxHighlighter*-datatype like this greatly improves on extensibility to new editors that use a similar structure. An alternative approach could be to define one *SyntaxHighlighter*-datatype per editor and write a specific machines-to-highlighter function per editor.

The first of the aforementioned approaches is chosen and this translation step from machine to highlighter can be described as follows:

1. For each machine M create a context c_q per state $q \in M.Q$
2. For each context c_{q_1} add matches for each $tr = (q_1, a, q_2) \Rightarrow q_1, q_2 \in Q \wedge a \in \Sigma$:

$$c_{q_1}.matches += (a, null(), set(c_{q_2}))$$
3. For context c_{q_1} add matches for each $tr = (q_1, A, q_2) \Rightarrow q_1, q_2 \in Q \wedge A \in V$:

$$tokens = firstTerminalsOf(M(A))$$

$$c_{q_1}.matches += (lookahead(any(tokens)), null(), set([c_{q_2}, c_{A.q_0}]))$$

The regular expression is supposed to be a lookahead match on any of the terminals of the machine $M(A)$. This is needed because lookahead does not consume the actual pattern that was matched. This will be done by the first context of $M(A)$. The action replaces the old context c_{q_1} with the c_{q_2} and pushes the first context of machine $M(A)$ called $c_{A.q_0}$ on top of that. So the highlighter is now in machine $M(A)$ and once this pops itself off, it ends up in the next state of the original machine.
4. For each context $c_{A.q} \Rightarrow q \in M(A).F$ take one of the action described in the next section (**Final State Cases**).
5. For the start symbol S create a context $c_{main} = include(c_{S.q_0})$
 Creates an empty *main* context that includes the first actual context.

$$H.main = c_{main}$$
6. Identify all matches that need to have a scope assigned and do so. This can be done based on the information gathered in 3.2. For giving entire machines a scope you need to assign the scope to all matches of all contexts corresponding with this machine.
7. Remove all contexts that are unreachable

Final State Cases

There are two cases for a final state of a machine. The state is either final with no outgoing arcs (I.E. a *sink-state*). Or it still has other outgoing arcs (*non-sink state*). These cases could be treated differently when mapping them to contexts.

- **Sink-states:**

For each sink-state there will be no match created (yet), because it has no outgoing arcs. The only thing to be added is a match that pops this context and machine off of the stack.

Let a final sink-state be q_{fs} then:

$$c_{q_{fs}}.matches += (\epsilon, null(), pop())$$

This just matches the empty regex, this always succeeds, and then pops. A different approach would be to choose, in the last match that would lead to $c_{q_{fs}}$, to $pop()$ here and to not $push()$ nor $set()$ this context in the first place. In terms of the above sketch:

For each $c_{q_1} \in C$, for each $tr : (q_1, a, q_{fs}) \Rightarrow q_1, q_{fs} \in Q \wedge a \in \Sigma$

$$c_{q_1}.matches += (a, null(), pop())$$

For each $c_{q_1} \in C$, for each $tr : (q_1, A, q_2) \Rightarrow q_1, q_{fs} \in Q \wedge A \in V$

$$tokens = firstTerminalsOf(M(A))$$

$$c_{q_1}.matches += (lookahead(any(tokens)), null(), set(c_{A.q_0}))$$

- **Non-Sink states:**

These states are harder to reason about. There are choices that need to be made, which is hard to do in a deterministic way. A state like this means that you can either *pop* the current machine, or continue consuming with this machine. There could be conflicts here again as well, for example:

The current machine is B .

A final non-sink state $A.q_f \in F$ has 1 transition: $tr = (q_f, a, q_f) : a \in A.\Sigma$

The state $B.q_x$ has a transition: $tr = (q_x, a, q_y) : a \in B.\Sigma$

The context stack looks like this: $[c_{B.q_x}, c_{A.q_f}]$

This is a conflict, do you *pop* and consume a , or continue and consume a ?

There is a choice to be made:

1. Resolve these end-of-machine conflicts
2. Always pick the *pop*-action
3. Always pick the *continue*-action

Highlighters do not need to be perfect, so for now pick option 3.

To accomplish this, add the same match as if this were a *sink-state*.

This has to be the final match, or the other matches become unreachable.

Example

Below is a stepwise example of the just described mapping of the machine shown in figure 3.3. For ease of reading it is written in Sublime Syntax format.

1. For each machine M create a context c_q per state $q \in M.Q$

```

1 Statement_St:
2 Statement_0:
3 Statement_1:
4 Statement_2:
5 Statement_3:
6 Statement_4:
7 Statement_5:
8 Statement_6:
9 Statement_epsilon:

```

2. For each context c_{q_1} add matches for each $tr = (q_1, a, q_2) \Rightarrow q_1, q_2 \in Q \wedge a \in \Sigma$:

```

1 Statement_St:
2   - match: 'if'
3     set: [Statement_3]
4 Statement_0:
5   - match: ':= '
6     set: [Statement_1]
7 Statement_1:
8 Statement_2:
9 Statement_3:
10  - match: '('
11    set: [Statement_4]
12 Statement_4:
13 Statement_5:
14  - match: '))'
15    set: [Statement_6]
16 Statement_6:
17  - match: '{'
18    set: [Statement_St]
19 Statement_epsilon:

```

3. For context c_{q_1} add matches for each $tr = (q_1, A, q_2) \Rightarrow q_1, q_2 \in Q \wedge A \in V$:

```

1 Statement_Start:
2   - match: 'if'
3     set: [Statement_3]
4   - match: '?=([a-z]+)'
5     set: [Statement_0, Id_Start]
6 Statement_0:
7   - match: ':= '
8     set: [Statement_1]
9 Statement_1:
10  - match: '?=(\"| [0-9]+)'
11    set: [Statement_2, Literal_Start]
12

```

```

13 Statement_2:
14   - match: '?=({| })'
15     set: [Statement_epsilon, Statement_end_Start]
16 Statement_3:
17   - match: '('
18     set: [Statement_4]
19 Statement_4:
20   - match: '?=[a-z]+'
21     set: [Statement_5, Condition_Start]
22 Statement_5:
23   - match: ')'
24     set: [Statement_6]
25 Statement_6:
26   - match: '{'
27     set: [Statement_Start]
28 Statement_epsilon:

```

4. For each context $c_{A,q} \Rightarrow q \in M(A).F$ take one of the action described in **Final State Cases**.

```

1 Statement_epsilon:
2   - match: ''
3     pop: true

```

5. For the start symbol S create a context $c_{main} = include(c_{S,q_0})$

```

1 main:
2   - include: Statement_Start

```

6. Identify all matches that need to have a scope assigned and do so. This can be done based on the information gathered in 3.2. For giving entire machines a scope you need to assign the scope to all matches of all contexts corresponding with this machine.

```

1 Statement_Start:
2   - match: 'if'
3     scope: keyword.control.flow
4     set: [Statement_3]
5   - match: '?=[a-z]+'
6     set: [Statement_0, Id_Start]

```

7. Remove all contexts that are unreachable

-

Chapter 4

Rascal Implementation

4.1 Motivation for Rascal

Rascal is a language workbench. It is also a language to be used for many high-level tasks involving the development of new languages and analysis on those and other languages. Since it is very high-level it makes it easier to tackle a rather difficult problem as this project is faced with. Doing something like this in a language like C would be near-impossible. Rascal provides a grammar formalism, algebraic datatypes and high-level operations on structures like graphs and sets.

The official website¹ states:

"Rascal integrates source code analysis, transformation, and generation of primitives on the language level. It can be used for any kind of metaprogramming task: to construct parsers for programming languages, to analyze and transform source code, or to define new DSLs with full IDE support. . . Rascal primitives include immutable data, context-free grammars and algebraic data-types, relations, relational calculus operators, advanced pattern matching, generic type-safe traversal, comprehensions, concrete syntax for objects, lexically scoped backtracking, and string templates for code generation."

The feature that is being developed by this project is not yet implemented in Rascal, as Rascal does syntax highlighting based on the CFG inside the Eclipse IDE. Rascal itself is also not exported to the various editors with any proper highlighting. This together with all of the above are reasons why Rascal was chosen as the language to work with for this project.

¹source: CWI, 2014

4.2 Grammar formalism

Rascal² has a build in grammar formalism that can be used to develop grammars. Rascal will then generate a parser for this language and generate parse trees and abstract syntax trees for the grammar. One of the special things about Rascal is that it does not care about the complexity of the grammar. It does not need a grammar that is $LR(1)$, theoretically it can generate parsers for any CFG. However, there is often ambiguity found in a grammar, this can cause the parser-generator to fail. For this reason Rascal has something called disambiguation constructs. These are handles for the generator such that it can generate parsers without ambiguity. Since this algorithm operates on the grammar-level there is no need to dive into the parsing and evaluation that Rascal does. It is only important that all features that the grammar formalism uses are covered by the syntax-highlighter generator.

4.2.1 Simple grammar definition

Below is a simple grammar shown in Rascal. The keyword `start` denotes the start-symbol S of a grammar. There are four keywords that denote a non-terminal token. These are `syntax`, `lexical`, `layout` and `keyword`. These all have different meanings to the parser-generator. `syntax`-tokens and their rules are internally interleaved with the `layout`-token that is defined for the module the grammar is defined in (or if unspecified the `$default$` layout which is empty):

```
syntax S = A B
internally becomes
syntax S = A Layout B
```

`lexical`-tokens are the same as `syntax`, except for the fact that these do not get interleaved with `layout`-tokens. The `keyword`-tokens are a special kind of tokens that are used for disambiguation. It is used for making sure that keywords are not seen as identifiers for example. These tokens can only have single character classes or string literals as their rules. So no non-terminals nor regular expressions.

```
1 start syntax S
2   = A B
3   ;
4
5 lexical A
6   = "a" "a"
7   ;
8
9 lexical B
10  = "b" "b"
11  ;
12
13 layout Layout = [ \ ]*;
```

LISTING 4.1: A Simple Example Grammar in Rascal

²source: *Rascal tutor documentation*

4.2.2 More advanced constructs

Regular tokens

Rascal has a number of things which help the grammar-writer to develop new grammars quicker without the need to introduce new intermediate tokens to achieve the same result. These are explained and shown below.

- *Character classes*

The simple regular expression concept that defines a character class between square brackets. Negative classes are an option and Rascal supports everything including Unicode characters. Characters are internally represented as integer numbers.

```
1 lexical charclass
2   = [a-z]
3   | ![%]
4   ;
```

- *Star, Opt and Plus-operator*

The familiar Kleene star and plus operation for zero or more and one or more repetitions respectively. Together with the question mark for zero or one occurrence.

```
1 lexical B_STAR = B*;
2 lexical B_PLUS = B+;
3 lexical B_OPT = B?;
```

- *Begin- and End-of-Line*

Also simple and familiar, the beginning and end-of-line requirements for a regular expression. Denoted with ^ and \$.

- *Iterseps*

A more special kind of token, also defined on Kleene star and plus with 0 or 1 or more. This is a structure that takes a main token and separator. This means that any list of the Main tokens, interleaved with the list of separators, is accepted. This is very handy in for example parsing lists of parameters that are split by a comma.

```
1 lexical ParamList = {IdType ","}*;
2 /*parsing for example: int a, float b, string c*/
```

Labels

Labels are simple structures that allow naming of tokens inside a production rule, or to an entire production rule. Below are lhs, op, rhs all labels given to tokens and is num the name given to the entire rule with "42".

```
1 lexical Exp
2   = Exp lhs "+" op Exp rhs
3   | num: "42"
4   ;
```

Tags

Tags are a special kind of annotation that can be added to production rules. They can give certain messages to the parser-generator. This can be messages like: this type of rule is a comment. Another message not intended for the generator could be: the tokens of this rule should receive a certain colouring.

```

1 lexical Comment
2   = @@Comment>@ "/" ComChar "*" /
3   ;
4
5 lexical String
6   = @@Context="string.quoted.double null null string.quoted.double">@
7     StringBody "\" Interpolated "\" StringBody
8   ;

```

Disambiguation

Rascal has a number of handles to remove ambiguity from a grammar. This is useful for parser-generating, however not always for syntax-highlighting. Nevertheless the implementation should take all these factors into account.

- *Keywords*

The first is the keywords construct that was mentioned in the previous section. It can be used to match a certain expression minus the tokens specified in a **keyword**. The expression below matches all tokens consisting of the regex, except for those that match something in *Keywords*

```

keyword Keywords = "if" | "else" | "for" | "while";
lexical Id = [a-zA-Z][a-zA-Z0-9]* \ Keywords;

```

- *Precede and Follow restrictions*

Because Rascal accepts all general CFG's it does not implement longest or shortest match. It matches all possible variants. The only way to force rascal to take either longest and/or shortest match is through follow and precede-restrictions. In simple terms: `[a-z]*` and input `abc`, matches: `[ϵ , a, b, c, ab, bc, abc]`. One can either enforce that something follows or precedes, but also the 'not' of those two.

<code>input = 'abc'</code>	
<code>lexical Id = [a-z]*;</code>	matches: <code>{ϵ, a, b, c, ab, bc, abc}</code>
<code>lexical Id = [a-z]* !>> [a-z];</code>	matches: <code>{abc, bc, c}</code>
<code>lexical Id = [a-z]* !<< [a-z]*;</code>	matches: <code>{a, ab, abc}</code>
<code>lexical Id = [a-z]* !<< [a-z]* !>> [a-z];</code>	matches: <code>{abc}</code>

- *Associativity*

Rules can receive associativity which reduces ambiguity in parsing. Think of a grammar like expression. Normally you have to write the grammar in a `Exp = Expr "+" Term` way. Otherwise your parse-tree ends up being wrong on evaluation. However Rascal allows `Exp = Exp "+" Exp`, together with the associativity ruling to influence a parse tree to nest correctly and prevent ambiguity. The website states:

Using Associativity declarations we may disambiguate binary recursive operators. The semantics are that an associativity modifier will instruct the parser to disallow certain productions to nest at particular argument positions:

- Left and assoc will disallow productions to directly nest in their right-most position.
- Right will disallow productions to directly nest in their left-most position.
- Non-assoc will disallow productions to directly nest in either their left-most or their right-most position.

- *Priority*

Rules can receive priority over other rule(s) which reduces ambiguity in parsing. Think of a grammar like expression. Normally you have to write the grammar in terms of Exp, Term, Factor way. Otherwise there is no way to express operator precedence. However in Rascal you can do the following: Exp = Exp "+" Exp, Exp = Exp "*" Exp, together with the precedence rules you can parse and find correct parse trees. In the example below, the rule "42" has precedence over the "times" operator and that has precedence over the plus.

```
1 lexical Exp
2   = "42"
3   > left Exp "*" Exp
4   > left Exp "+" Exp
5   ;
```

- *Except*

Except rules are used in order to prevent certain labeled tokens or productions to be matched in a rule. This is done through putting an exclamation mark together with the to-be-excepted labelname at the end of a production rule.

```
1 syntax S = String!illegal;
2
3 lexical String
4   = legal: "whatstringscanbe"
5   | illegal: "whatstringscannotbe"
6   ;
```

An advanced example grammar

In appendix A a grammar is shown that makes use of most of the more advanced constructs that were discussed in these past sections.

4.2.3 Internal representation

Internally the grammars are represented a little differently. In short, all tokens (terminals and non-terminals), are of a datatype called `Symbol`. This is an algebraic datatype that can have symbols nested. A production rule also has a datatype called `Production`. This is also an algebraic datatype that shows a lot of the structure of the formal definition of a rule $p \in P \Rightarrow A \rightarrow w$.

Symbols

Symbols are ADT's and divided into four sections seen below. The first one is the one to denote that some symbol is a start symbol. Line four through seven represent the tokens `syntax`, `lexical`, `layout`, `keyword`. The parameterized versions of `syntax` and `lexical` are special versions of these definitions. They can be defined with a special datatype in order to generalize certain symbols. These are virtually nowhere to be found in any grammar used for this project and therefore not much attention was paid to these two symbols.

Following these lines there are three options defining terminal tokens. These are string-literals, case-insensitive literals and the character-classes.

Lines 15 through 22 represent the regular options for a token. Examples of these in order are: `"`, `A?`, `A+`, `A*`, `{A " , " }+`, `{A " , " }*`, `(A|B)`, `(A B)`. These last two are used for alternatives and sequences that are inside a single production rule in parentheses.

The final line shows how Conditions are represented. These include all disambiguation constructs on `Symbol`-level and the begin- and end-of-line operators. The datatype of the `Condition` can be found in appendix B.

```

1 data Symbol
2   = \start(Symbol symbol)
3
4   | \sort(str name)
5   | \lex(str name)
6   | \layouts(str name)
7   | \keywords(str name)
8   | \parameterized-sort(str name, list[Symbol] parameters)
9   | \parameterized-lex(str name, list[Symbol] parameters)
10
11  | \lit(str string)
12  | \cilit(str string)
13  | \char-class(list[CharRange] ranges)
14
15  | \empty()
16  | \opt(Symbol symbol)
17  | \iter(Symbol symbol)
18  | \iter-star(Symbol symbol)
19  | \iter-seps(Symbol symbol, list[Symbol] separators)
20  | \iter-star-seps(Symbol symbol, list[Symbol] separators)
21  | \alt(set[Symbol] alternatives)
22  | \seq(list[Symbol] symbols)
23
24  | \conditional(Symbol symbol, set[Condition] conditions)
25 ;

```


Productions

The production datatype is the type that represents elements of P in a grammar. Below and in the appendices the datatype is shown. However there a number of unused rules when it comes to the algorithm. The *Rascal Tutor* states:

- A prod is a rule of a grammar, with a defined non-terminal, a list of terminal and/or non-terminal symbols and a possibly empty set of attributes.
- A regular is a regular expression, i.e. a repeated construct.
- An error represents a parse error.
- A skipped represents skipped input during error recovery.
- priority means ordered choice, where alternatives are tried from left to right;
- assoc means all alternatives are acceptable, but nested on the declared side;
- reference means a reference to another production rule which should be substituted there, for extending priority chains and such.

From this it is clear to see that `error` and `skipped` are useless. Less obvious is the `regular`, one might think that this refers to the regular symbols that are possible. This however refers to the built-in Regex-engine that Rascal has and is therefore not part of this. Reference could be used in a grammar, however this was never encountered during any of the grammars that were tried. Recursive priorities are flattened to only a single priority. Every nested group of rules are rebuild to a `\choice()` and put as a single item in the list of choices in the priority object.

```

1 data Production
2   = prod(Symbol def, list[Symbol] symbols, set[Attr] attributes)
3     | regular(Symbol def)
4     | error(Production prod, int dot)
5     | skipped()
6     | \priority(Symbol def, list[Production] choices)
7     | \associativity(Symbol def, Associativity \assoc, set[Production]
8       alternatives)
9     | \reference(Symbol def, str cons)
10    | \choice(Symbol s, set[Production] choices)
11    ;

```

4.3 Implementing the Algorithm in Rascal

The upcoming section discusses the implementation in Rascal. This includes datatypes, functions and algorithms, however no results or examples are shown. All examples generated from the implementation are results and will be discussed in the corresponding chapter.

4.3.1 ToPlainGrammar

The first step is to reduce the input to a plain grammar that was defined in 2.2.1. Using all that was discussed above there are number of features that Rascal supports that are not of importance to the algorithm and need to be removed or rewritten. These occur at production- as well as symbol-level. A list of features to be removed or redone at the symbol-level include:

- labels, they will not be used in the highlighter and can be removed
- tags that do not describe contexts can be removed
- conditions, some of these are not important to a highlighter.

`\at-column(_)` state-based highlighters do not support a check like this.

`\except(_)`, labels were removed so this is not useful anymore either.

- All regular-symbols can be rewritten to terminals, non-terminals and production rules.

```

1  A?      -> A_OPT      = A | ;
2  A+      -> A_PLUS     = A | A A_PLUS;
3  A*      -> A_STAR     = | A A_STAR;
4  (A | B) -> A_B_ALT    = A | B;
5  (A B)   -> A_B_SEQ    = A B;
6  {A " , " }+ -> A_,_ITERSEPS = A A_,_ITERSEPSTAIL
7              -> A_,_ITERSEPSTAIL = " , " A A_,_ITERSEPSTAIL | ;
8  {A " , " }* -> A_,_ITERSTARSEPS = A A_,_ITERSTARSEPSTAIL | ;
9              -> A_,_ITERSTARSEPSTAIL = " , " A A_,_ITERSEPSTAIL | ;

```

The Sublime syntax-highlighters support character classes as well and therefore it is unnecessary to remove these. The other conditionals can be represented with a special regular expression using lookahead and lookbehind. Also all tokens that have nested symbols being a terminal are not rewritten to new Symbols but kept as regular expressions. So `[a-z]+` is kept as is and does not become.

`[a-z]PLUS = [a-z] [a-z]PLUS | [a-z];`

The best approximation of a formal production rule $p \in P$ is the

`\prod(Symbol lhs, list[Symbol] rhs, set[Attr] attributes).`

Of the productions that can be encountered there are 3 in conflict: priority, associativity and choice. Choice only holds a set of Productions together with the left-hand symbol that these have in common. If all elements of choice are of type `prod`, then choice can remain for ease of use. A highlighter does not produce parse-trees so the associativity rules are not important and can therefore be removed. This is done through extracting the set of alternatives from the Production and removing the associativity attributes from the rules found in the set.

For priority there are two options. Either rewrite the rules to keep the structure, or just remove them as done with associativity. The first option is like rewriting

```
Exp = "42" > Exp "*" Exp > Exp "+" Exp;  
to
```

```
Exp = Exp "+" Exp | Exp1; Exp1 = Exp1 "*" Exp1 | Exp2; Exp2 = "42";
```

The second option is just like with associativity and flatten priority to a choice Production. I have implemented both versions which will be discussed in the [Results & Evaluation](#) chapter. The nice thing about rewriting it in this way is that you can keep using the grammar formalism of Rascal and all its features, like writing an internal representation back to the human representation.

Layouts

I chose to remove all layouts from all of the production rules. This is due to the fact that generally the `layout`-tokens are used for 2 things: whitespace and comments. Although comments are important to the highlighter, whitespace is not. I chose to extract them from the productions but keep them behind such that these can go into a prototype context³. This way the rest of the machines become way less cluttered with possible whitespace parsing.

Scope information extraction

The preserving of the scopes to be assigned are handled in the beginning of the process. Every symbol of the simplified grammar is mapped to the corresponding scope that is saved for later use. However there is a limitation in the current implementation which causes loss of power in the potential highlighter. This is the fact that any token can only have one scope assigned to it over the stretch of the entire grammar. This is true for non-terminals as well as terminals (so no different colouring between function call, definition or variable declaration identifiers). If nested conflicts arise then the deepest scope is assigned to the matched token. (E.G. an entire machine of some non-terminal wants scope A and a Token inside this non-terminal's rules has scope B). Due to a lack of time and performance of the algorithm this was implemented in this way and not extended to the better variant described in earlier chapters.

³prototype contexts were contexts that are always included (2.4.3)

The implementation

The runtime of this part of code is linear with the size of the input grammar, where the size of a grammar is defined by the number of production-rules together with the total number of Symbols present in the grammar (including nested ones). This can be seen in the function below. First it rewrites all rules of a grammar to a set of plain elements of the type `\prod(Symbol lh, list[Symbol] rhs, set[Attr] attributes)`. After this, all symbols in the right-hand side of a rule are rewritten in the way that was just described. Then, as a final action, based on the passed parameter, remove all the interleaved `layout`-tokens. However, rules corresponding to layouts themselves are kept. This results in a new grammar in which by normal means the `layout`-symbols are unreachable, however this is fixed through adding them as a prototype context later.

```

1 Grammar toPlainGrammar(Grammar g, bool keepLayout=false) {
2   set[Production] rules = getRewrittenRules(g);
3   set[Production] new_rules = {};
4   for (rule <- rules) new_rules += rewriteSymbolsInRule(rule);
5   if(!keepLayout) new_rules = removeLayouts(new_rules);
6   return grammar(g.starts, new_rules);
7 }

```

4.3.2 Grammar2Graph

There was already something present in Rascal that created a graph with dependencies of symbols. This however did not search recursively enough, so I wrote my own variant that dives recursively into a symbol and finds any general non-terminal token that is not inside a `Condition`. The build-in Graph datatype is nice and generically written, however the function to compute its strongly connected components was not yet implemented. So I implemented Kosaraju's algorithm in order to compute the components.

Both the *grammar2graph* and *Kosaraju's Algorithm* are shown below. The functioning of *grammar2graph* can be written as: For each `Symbol elem` in the right hand side of any rule in the grammar, check whether the left-hand side *s* is a label with a plain symbol or a plain Symbol itself. Assign the name *from* to the plain symbol. Now if *from* is some instance of a non-terminal, enter the second for-loop. For each symbol that is a plain non-terminal token inside *elem* (including *elem* itself), add an edge from the symbol *from* to the plain non-terminal that was found.

One extra feature was added to this function. Every vertex gets a self-loop. This is because of Rascal's graph implementation being just the set of edges. This disallows disconnected vertices to be part of a graph. To circumvent this, I added this little line that is not entirely correct, but helps Kosaraju's algorithm to find all components, including singular components with no edges.

```

1 Graph[Symbol] grammar2graph(Grammar g) {
2   out = {};
3   for(/prod(Symbol s,[_*,Symbol elem,_],_) := g,
4     (label(_,Symbol from) := s || Symbol from := s),
5     from is sort || from is lex || from is \parameterized-sort
6     || from is \parameterized-lex || from is layouts) {
7
8     for (to <- getDeepestSymbols(elem), to is sort || to is lex
9       || to is \parameterized-lex || to is \parameterized-sort
10      || to is layouts)
11       out += <from, to>;
12     out += <from, from>;
13   }
14   return out;
15 }

```

Kosaraju

Below is the algorithm of Kosaraju shown. *components* is a map that maps every edge to its corresponding component. Returning the range of this map is the set of strongly connected components. It is written generically so it can be used on any Rascal-defined Graph.

```

1 public set[set[&T]] stronglyConnectedComponents(Graph[&T] G) {
2   set[&T] visited = {};
3   list[&T] L = [];
4   for (u <- domain(G)) {
5     <sub_L, sub_visited> = visitNode(G, visited, u);
6     L = sub_L + L;
7     visited = sub_visited;
8   }
9   map[&T, set[&T]] components = ();
10  set[&T] passed = {};
11  Graph[&T] inverted = invert(G);
12  for (u <- L) {
13    <sub_components, sub_passed> = assign(G, inverted, u, u, components,
14    passed);
15    components += sub_components;
16    passed += sub_passed;
17  }
18  return range(components);
19 }

```

4.3.3 ToStronglyRegularGrammar

The transformation described in Mohri and Nederhof, 2000 is implemented in rascal. It handles all symbols except parameterized lexicals and parameterized syntax-tokens. These throw exceptions everywhere they cause a problem, so this is easy to extend. These were never encountered so not really in scope for this project. The conversion is preceded by performing the *ToPlainGrammar*-function, since it expects a grammar consisting of only `\prod(_,_)` and `\choice(_)`. As an intended side-effect this transformation removes all attributes from the grammar. This makes the new grammar loose all information on contexts and scopes since this should be extracted from the plain version of the grammar and not the strongly regular one. Below is the *strongly regular* version of the expression-grammar. The *A*'s are replaced with *A_end*, since the primed versions of the tokens correspond to parsing the ending of a token.

4.3.4 Converting the grammar to StateMachines

After the grammar is rewritten to a strongly regular one it needs to be rewritten to a set of statemachines. Rascal has a datastructure called `map[&T from, &S to]`. This can be used to map a single Symbol or set of Symbols (I.E. a component) to a StateMachine. Using the algorithm from 2.3.2 the machines can first be generated per component and then mapped to the Symbols which are then mapped to *DFA*'s. This is done through the standard powerset construction. So no *NFA*'s per component because the next step is to convert them to *DFA*'s. The implementation is slightly different from the one described, because it introduces a few extra states. Instead of introducing one transition (q_S, a, q_B) for the rule $S = "a" B$, there are two rules introduced: (q_S, a, q_{term_a}) and $(q_{term_a}, \epsilon, q_B)$. These intermediate states are all uniquely named. Therefore, not every instance of a terminal *a* goes to the, same q_{term_a} .

StateMachine datatype

In the section below⁴ the datatype of a StateMachine is shown. States are represented by strings, a Token is something that can be on an arc, which in this case is a Symbol. A Transition is a 3-element tuple that holds two states, from and where, and a Token that triggers the transition. A DeltaFunction is a set of Transitions that can be indexed as an array. For example: Transition $tr[s_0]$ returns a set of 2-element tuples of type `set[tuple[Token, State]]`, which are all possible combinations of tokens and resulting state that s_0 has in the machine. Now a statemachine is either an *NFA* or a *DFA*. This is decided with the constructor that is chosen on instantiation. There is no actual prevention of adding non-determinism to a *DFA*. This is left to the programmer. For ϵ -transitions I created an extension to Rascal's Symbol called `\eps()`. As can be observed the states are not saved as separate items in the machine since they can be obtained from δ .

```

1 alias State = str;
2 alias Token = Symbol;
3 alias DeltaFunction = rel[State from, Token token, State to];
4 alias Transition = tuple[State from, Token token, State to];
5
6 data StateMachine
7   = \sm-dfa(DeltaFunction delta, State q0, set[State] F)
8   | \sm-nfa(DeltaFunction delta, State q0, set[State] F)
9   ;

```

Conflict resolution

Resolving the conflicts is done as described in 3.2.5. This is done in almost the exact way the algorithms describe.

4.3.5 Mapping to Contexts

The mapping to contexts is largely done through the description in chapter 3. Rascal has a few extra features that are left to consider, being the Conditional Symbols. These are harder to process since the conditions themselves can result in a prefix or a postfix of a regular expression. For example: `[a-z] !<< [a-z]+ !>> [a-z]` can be expressed with lookahead and -behind as `(?![a-z])([a-z]+)(?![a-z])`. Luckily they are all expressible in the regex engine that Sublime and other editors like TextMate use. Precede and follow restrictions translate into lookbehind and lookahead. The `\delete(_)` becomes negative lookahead before the actual match (E.G. `id "for"` result in match: `'(?!for)(id)'`). The begin and end-of-line are also implemented in the regex engines as the same tokens. So except for doing a few tricks with generating correct regular expressions this step is done in the same manner as was described in the previous chapter.

⁴In the actual code *delta*, *q0* and *F* are replaced with *transitions*, *startState* and *finalStates*.

Highlighter datatype

Below is the datatype that represents a state-based syntax highlighter in Rascal. It represents much of the structure of a Sublime Highlighter, but with minimal extensions or modifications other editors could be added. There is also a function that writes an instance of this type to a *.sublime-syntax* file. An *SHRegex* is regular expression to be set as a match. The *SyntaxHighlighter* is the top level object. It contains a set of *SHVar* which are named regular expressions that can be defined for reuse. The rest of the constructors and types speak for themselves.

```

1 data SHRegex
2   = \str-regex(str regex)
3   ;
4
5 data SyntaxHighlighter
6   = \highlighter(str name, set[str] extensions,
7       set[SHvar] vars, map[str, Context] contexts)
8   ;
9
10 data SHvar
11   = \shvar(str name, SHRegex regex);
12
13 data Context
14   = \context(str name, Scope scope, set[Match] matches,
15       set[Context] includes, bool includePrototype=true)
16   | \main(Scope scope, set[Match] matches, set[Context] includes)
17   | \prototype(Scope scope, set[Match] matches, set[Context] includes)
18   ;
19
20 data Scope
21   = \scope(str name)
22   | \meta-scope(str name)
23   | \null()
24   ;
25
26 data Action
27   = \push(list[str] contexts)
28   | \setact(list[str] contexts)
29   | \pop()
30   | \noact()
31   ;
32
33 data Match
34   = \match(SHRegex regex, Scope scope, Action action)
35   ;

```


Chapter 5

Results & Evaluation

In this section I will discuss a number of cases and snippets to showcase the results and behaviour of the implemented algorithm. This includes trade-offs, design choices and flaws. In this section the different reasons that cause problems in the final highlighters will become apparent as well as which parts are very useful for the process.

5.1 Embedding scope information

The system of embedding highlighting information established in 3.2 is quite literally used in the Rascal grammars. An example of this is included here as well as in the appendix (C.0.4). A number of things become clear about this way of embedding information:

Pros:

1. Assigning scopes in this manner is easy and intuitive.

Cons:

1. Having to write the entire scope for every rule quickly becomes tedious. Especially for simple cases (E.G. `MyKeywords`).
2. Unable to assign scopes to nested symbols that rascal views as one. For example: `"+"` and `"-"` inside the last priority group of `Expression`
3. Knowledge of what Rascal sees as one symbol and what not is required. `Identifier` has only 1 symbol on its right-hand side.

```

1 layout MyLayout = [\t\n\ \r\f]*;
2
3 lexical Identifier = [a-z] !<< [a-z]+ !>> [a-z] \ MyKeywords;
4
5 keyword MyKeywords
6   = @@Context="keyword.flow.conditional">@ "if"
7   | @@Context="keyword.flow.conditional">@ "then"
8   | @@Context="keyword.flow.conditional">@ "else"
9   | @@Context="keyword.flow.conditional">@ "fi"
10  ;
11
12 syntax Expression
13   = id: Identifier id
14   | @@Context="constant.language">@ null: "null"
15   | bracket "(" Expression ")"
16   > @@Context="null keyword.operator.arithmetic null">@
17     left multi: Expression l "*" Expression r
18   > left ( add: Expression l "+" Expression r
19           | sub: Expression l "-" Expression r
20           )
21 ;

```

LISTING 5.1: The advanced example grammar with scope information

To make the process of scope assigning less tedious, one could add scopes on Symbol level as well as production level. In this way `MyKeywords` could get one scope assigned instead of four times the same one. To circumvent the second issue, one has to rewrite parts of the grammar that are in specific need of colouring. This is not the end of the world, but not preferable. A better possible approach would be to allow the assigning of scopes to labels. Since labels can be found and assigned inside the nested symbols of a production. The third issue is not as much an issue in the sense that it obstructs the usefulness of the algorithm, however it represents the fact that the implementation is not very user-friendly towards beginners of Rascal.

5.2 Simplifying the grammar

Inside the project there is a module called *ToPlainGrammar*. This module does the rewriting of a standard Rascal grammar to the plain form. Results of this can be viewed below. This is done in smaller snippets from larger grammars. 2 snippets are shown here, the Expression non-terminal of the grammar in the previous section. The second snippet shows a formulation for a list of declarations.

For a complete look at the pipeline one can look at appendix C through F. There are two forms of the *toPlainGrammar*-function, one that preserves the priority and one that does not. The first can be viewed in the appendices, the second is used here.

```

1 //After
2 syntax Expression
3   = @@Context="constant.language">@ "null"
4   | Identifier
5   | bracket "(" Expression ")"
6   | @@Context="null keyword.operator.arithmetic null">@
7     Expression "*" Expression
8   | Expression "+" Expression
9   | Expression "-" Expression
10  ;
11 /* ----- */
12 //Before
13 syntax Declarations
14   = @@Context="keyword.declaration null">@ "declare" {IdType ","}* ";";
15   ;
16
17 //After
18 syntax Declarations
19   = @@Context="keyword.declaration null">@ "declare" IdType__,"__STARSEPS
20     " ";
21   ;
22
23 syntax IdType__,"__STARSEPS
24   = IdType IdType__,"__STARSEPS_TAIL
25   |
26   ;
27
28 syntax IdType__,"__STARSEPS_TAIL
29   = "," IdType__,"__STARSEPS
30   |
31   ;
32
33 syntax IdType = Id ":" Type;

```

LISTING 5.2: Snippets showcasing the behaviour of *ToPlainGrammar*

As seen above the removal of unwanted features functions as it should. It properly rewrites regular structures into a set of non-terminals, terminals and productions. It also properly removes parse-tree altering structures on the rules as well as labels and some of the tags (last of these are removed in the *ToStronglyRegular*-section). The original version did not account for *layout*-symbols nor for parameterized tokens. These are all taken care of in this new implementation as well.

5.2.1 Scope information

The scope-information that is present in the grammar is saved in a proper manner. The contexts are saved as tags which are of type *Attr*. These end up being in the `\prod(_,_,set[Attr] attributes)`. Therefore rewriting the rules to `\prod(_,_,_)` or `\choice(_,_)` does not cause any loss of information. The symbol rewriting does not cause loss of information, because everything that is seen by the parser as one token is rewritten to still be one token (E.G. `syntax A = A (B C) {D ","}* ;` has only 3 tokens). These are kept during this transformation and will be extracted from the "plain grammar" once requested by the algorithm.

5.3 ToStronglyRegularGrammar

5.3.1 Grammar2Graph

The method present that converted a grammar into a graph did not perform well enough and has been rewritten to recurse (deeper) into tokens in order to find the deepest symbols present. The function visits all right-hand side elements once. Therefore this conversion runs in linear time with the size of the total amount of symbols of the grammar (nested included).

5.3.2 Kosaraju

Below the set of components of the advanced example grammar (Rascal expression) as well as the strongly regular version are shown. As can be observed, `keyword` MyKeywords is nowhere to be found. This is due to the fact that keywords are always excluded since they only serve as disambiguation constructs and never as real non-terminals. They could never be rewritten to other non-terminals because of the implementation of the Rascal-language.

```
1 //Original version
2 set[set[Symbol]]: {
3     {sort("Expression")},
4     {layouts("MyLayout")},
5     {lex("Identifier")},
6 }
7 //Strongly regular version
8 set[set[Symbol]]: {
9     {
10         sort("Expression"),
11         sort("Expression_end")
12     },
13     {layouts("MyLayout")},
14     {lex("Identifier")}
15 }
```

5.3.3 Mohri and Nederhof's transformation

In appendix A.1.2 the example from the original paper is included together with the output this module produces. It has some meaningless productions, this is part of the algorithm and the rules that it produces. Self-loops of the type $A \rightarrow A$ could be removed.

Below the advanced example grammar is shown again, it shows how Expression is rewritten into two non-terminals being Expression and Expression_end.

```

1 syntax Expression_end
2   = "+" Expression
3   | ")" Expression_end
4   | Expression_end
5   | "*" Expression
6   | "-" Expression
7   |
8   ;
9
10 keyword MyKeywords = "else" | "if" | "fi" | "then";
11
12 layout MyLayout = [\t-\n \a0C-\a0D \ ]*;
13
14 lexical Identifier = [a-z] !<< [a-z]+ !>> [a-z] \ MyKeywords;
15
16 syntax Expression
17   = Expression
18   | "null" Expression_end
19   | Identifier Expression_end
20   | Expression
21   | bracket "(" Expression
22   ;

```

LISTING 5.3: The strongly regular approximation of the advanced expression grammar (C.0.7)

The implementation of Mohri and Nederhof's algorithm works for all the cases that were tested and shown in this thesis, however there is an edge-case that is not covered in the implementation. This throws a warning, so once someone meets this edge-case it will be noticed. The case is matched when a mutually recursive set of non-terminals is completely left-linear, but not right-linear. If all rules are left-linear then this set of rules should not be rewritten. However on converting this set of rules to a NFA the algorithm crashes because it uses the simple NFA-construction algorithm from a right-linear grammar. In order to circumvent this one could rewrite the left-linear sets to right-linear ones with the standard rewriting-algorithm.

5.4 The Strongly Regular Grammar to StateMachines

In the appendix a number of NFAs as well as DFAs are included for other grammars and tokens. Here is a DFA and corresponding grammar that show the results of the algorithm and some of its flaws.

```

1 start syntax C = @@Context="comment.block">@ Comment;
2
3 lexical Comment = "/*" (ComChar | Comment)* "*/";
4
5 lexical ComChar
6 = ![*/]
7 | [*] !>> [/]
8 | [/] !>> [*]
9 ;

```

LISTING 5.4: A grammar showing nested comments

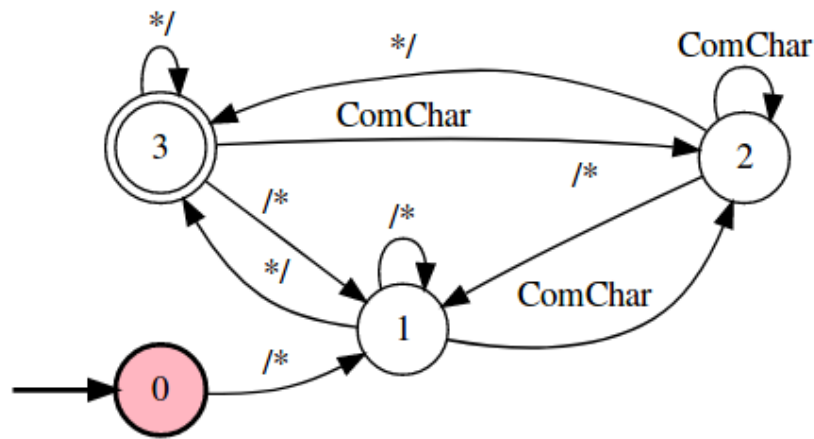


FIGURE 5.1: The DFA for Comment, produced from the strongly regular approximation

At first the machine seems proper, if one were to feed this a nested comment as input it is clear that it could consume this. This is good because it guarantees that there was no loss in information up till here. In terms of the algorithm and the final results however, this is a point which leads to problems in the final highlighter. When writing highlighters by hand, highlighting nested comments is easy because of the context-stack. The flow is simple: on opening comment push a comment context onto the stack, on closing comment pop a comment context off of the stack. The algorithm is designed in a different way, it creates a machine for each component and only pushes a machine if it finds a non-terminal on an arc such that when the pushed machine pops itself off, we end up in the next state. This ensures that within the machine above only the ComChar machine can be pushed, but never a nested comment machine. The choice made in 3.2.6 ensures that we remain in the same machine if we can in a non-sink state (state 3). Since a ComChar generally matches anything, so also code outside a comment, it chooses to remain in the machine forever. On top of that it is able to match as many opening and closing comments as it can without the need for balance. This means that the algorithm loses the possibility to properly highlight nested structures within the same component that require some sort of balancing (like bracket balancing and nested comments).

5.5 The conflict resolution

After implementing this feature it became apparent that there were a few things wrong with this step. The thing it does do well is serving its purpose. It completely removes all conflicts in a machine except for the ones mentioned in 3.2.6. These are similar to shift-reduce and reduce-reduce conflicts found in parsers.

The issues with this algorithm are:

1. This recursive process has a very bad time-complexity. It traverses all symbols in all components and all the transitions in the machines corresponding to these symbols. If a conflict is found, it runs the powerset-construction *NFA2DFA*-algorithm which has a bad time-complexity, and then reruns the search for conflicts.
2. Large grammars generate larger machines. Programming languages build on the recursive structure in a grammar and therefore a number of components tend to be large of size. This leads to enormous NFA's and DFA's. If conflicts arise, which happens at higher level tokens more often (like Statement in C), the fixing of these conflicts cause large waiting times. This is due to the bad complexity.

Small example: `x` can be a variable or a function-call. Both have identifiers and are at the start of the statement, this will cause a conflict.

```
1 void x() { printf("function x\n"); }
2 int main() {
3     int x;
4     x = 42;
5     x(y);
6 }
```

3. Another thing adding to the bad runtime of this algorithm is the fact that the machines of different symbols of the same component usually contain (many of) the same conflicts. These are all repetitively found and solved, including nested cases.

Except for the bad runtime however, the algorithm does work. So it is useful in the process, only the grammar writer would really benefit from thinking thoroughly about the conflicts he or she might introduce.

5.6 ToContexts

In this implementation, replacing an arc in a machine means that this non-terminal on this arc is gone afterwards. If a scope was assigned to this arc that information is lost from that point on, since it becomes part of this machine it will receive custom contexts in the next step of the algorithm. Therefore the highlighting might perform worse than expected.

Implementation of this part proved harder since there was a lot of rewriting symbols into regular expression involved, where information for this process was often not available at this point. Especially handling symbols that were of type conditional (`_`, `_`) proved difficult. There are no real intermediate results from this step as they tie together everything and basically produce the final highlighters to be discussed in the next section.

5.7 Resulting highlighters

5.7.1 Final results

Together with the final highlighters the appendices now show the complete path that a grammar takes through the algorithm together with intermediate results. The results being: original grammar → plain grammar → strongly regular grammar → an example NFA constructed from this regular grammar → the corresponding DFA → The final highlighter as a *.sublime-syntax*-file. Finally there is a small example program included in the form of a picture, showing the job the highlighter does. After this I have included a hand-written highlighter, together with a picture of how that one does.

This process is shown for the following grammars: The advanced example grammar, a simple grammar showing nested comments, a simple grammar showing basic string-interpolation and the Pico grammar (an easy example DSL-like language).

5.7.2 Overall results

As can be seen in especially the final results, the highlighters do not do a particularly good job. First of all, looking at the nested comment grammar. It becomes clear that it does not do a proper job. This is mainly due to the following things:

1. In converting the strongly regular grammar into machines all elements of a component are put in one machine. In order to properly highlight nested comments a structure like this is needed: On start of new comment → push new machine that highlights a comment. On ending of a comment → pop the current machine. This is what the stack is useful for, to keep track of how many times the highlighter is nested. However because of the merging of all comments into one machine and the way the *ToContexts* part is constructed a new machine is only pushed if the highlighter moves from one machine to one that is outside its current component (I.E. a non-terminal arc). This never occurs for the nested comment case and this is why it accepts endless amounts of opening and closing comments without ever pushing nor popping machines.

2. There is also the issue discussed in 2.4.3 that brings problems in the final algorithm. Since the pushing of new machines is triggered on lookahead of one of the starting tokens of that machine problems arise when end of lines (\$) are the next token to be parsed. The Sublime highlighter matches per-line, so never past an end of line. If we have a context of the type:

```
1 context1_A:  
2   - match: '?(a)'  
3     set: [context1_B, context2_A]  
4   - match: ''  
5     pop: true
```

If we have input of the type `bla bla| $a b c` and the highlighter is at `|`. The highlighter will now check and try the matches of the context in order. The first one fails, because it can't look past a new-line, since it matches one line at a time. Therefore it will arrive at the final match and pop the context off, even though the real next token is `a` and therefore the highlighter "should" push the next machine on top. This can be seen very well in the *Pico* example program.

3. A general problem with the approach as a whole is that the algorithm tries to retain as much of the original structure of the grammar as possible. However when looking at hand-written highlighters, those programs do not really care about the grammar, it just searches for words to match and colour them. In other words, the generated highlighters act as parsers. This is bad for a multitude of reasons:

1. Highlighting will not continue properly once past a syntax error.
(E.G. keyword in a "wrong" place will get no highlighting)

2. Normal highlighters just skip tokens that do not need highlighting. It simply continues along them. On matching something it colours correspondingly. The parser-like behaviour wants to match every token it encounters otherwise it does not continue to the next context. This is bad for a simple reason: more meaningless matching and trying leads to a larger chance of errors.

3. It introduces the shift-reduce and reduce-reduce like conflicts found when the highlighter is in the final state of a machine. Even though highlighters should easily be able to highlight grammars outside $LR(1)$

4. The last problem is more of a beauty remark. The created highlighters are huge and get out of hand fairly soon. A highlighter that was generated for an 80-line grammar describing JSON generated a highlighter of over 750 lines of code. One can only imagine if this were run on for example the C-language (One time this finished after about 20 minutes of conflict-solving and generated about 40000 states in its DFA's).

Statistics on the final results

Below are a few tables that show and discuss the performance of the highlighters included in the appendices. They are compared with highlighters that were written by hand. In the appendices there are also examples included of the actual highlights the different versions produce.

TABLE 5.1: Statistics on generated highlighters vs hand-written ones

Name	Generated	SLOC	NumContexts	Performance (++)/+/+/-/-)
Nested comments	true	100	11	-
	false	27	3	++
String Interpolation	true	77	11	++
	false	33	4	++
Pico	true	566	57	-
	false	61	8	++

In here it is clear to see that the parser-like behaviour breaks the highlighter. It introduces many lines and contexts that do not contribute to the performance of the highlighters.

Chapter 6

Conclusion

6.1 Answers to subquestions

In 1.2.1 a number of subquestions were posed. These questions are useful in answering the main question of this thesis. The posed questions are:

1. How to embed highlighting information in a context-free grammar?

The Algorithm:

The idea is to assign scopes (or some other information) to the tokens of production rules. It is intuitive and easy to do, however it might be a good idea to allow certain generalizations, like assigning information to a Symbol as a whole, meaning to all Symbols of all rules at the right-hand side. The embedded information is easy to extract and is relevant for the rest of the algorithm. Changes to the algorithm might enforce the type of information to be changed. More on this in the Discussion.

The Implementation:

The grammar formalism Rascal uses nicely allows for this kind of embedding. Many types of information, not just scopes, could be embedded with this method. Implementation-wise there were also a number of things that could have been done better. For one, assigning scopes to nested Symbols was impossible. In the Results & Evaluation chapter a number of issues and possible solutions were posed. For example assigning to the information to different labels in Rascal would help to circumvent the nested Symbol problem.

2. Can a Rascal grammar be written as the standard 4-element tuple? The implementation of *ToPlainGrammar* shows that this is possible and also generalizes to larger grammars. The algorithm works even on large scale programming languages. It just fails on Rascal itself because of the use of parameterized lexicals and syntax-tokens.
3. How to convert a context-free grammar to a regular grammar approximating the same language? Mohri and Nederhofs transformation stood for this process. The approximation is correct and still understandable. This is important for the used algorithm, because it converts the grammar to statemachines which is not generally possible for context-free grammars.
4. How to create useful state-machines from these approximations? The statemachines generated from the transformed grammars are good for accepting the same language. Again however, this contributes to the parser-like behaviour of the algorithm. This will be more detailed in 7.

5. How to map these machines to an actual state-based syntax highlighter? The final step is the described mapping algorithm. This works for general cases, but encounters lots of problems for end-of-line characters in Sublime. These cause unintentional popping of contexts, breaking the "parser" that the highlighters try to be. Resulting in bad highlights.

6.2 How can we derive state-based highlighters from context-free grammars?

The question posed at the beginning could be rewritten after this thesis. This thesis has presented a possible approach to this problem and concluded that this is not the correct one. This thesis does not answer the question as a whole, but crosses off the described approach.

The generated highlighters have trouble navigating past end-of-line characters and act more as parsers than as highlighters. The highlighters perform badly and are for described reasons incapable of highlighting certain recursive structures like nested comments. I think it is still possible to generate highlighters from their definitions, however there is a need for a different algorithm and possibly other embedded information to support this new algorithm.

I have presented a possible approach to generating state based highlighters from Rascal context-free grammars using a regular approximation algorithm as a starting point. I conclude that, for various reasons, this approach is not capable of creating proper highlighters. Reasons include parser-like behaviour, unintentional popping of contexts on end-of-line characters and incapability of properly highlighting recursive structures like nested comments.

Chapter 7

Discussion

7.1 Strong points

7.1.1 Algorithm

The algorithm fails to serve its purpose and therefore it has little strong points. However a few things that are strong about the algorithm is that the way of embedding information is at the right position. It asks the grammar-writer not to write a separate part additionally to the grammar, but it asks extra information about the grammar. If the corresponding algorithm using the information (even if the information would be of a different kind), then generating a highlighter really comes from the grammar and part of the process of creating grammars.

The complexity of the components of the algorithm are also quite good. Usually linear with respect to the size of the input-grammar. Except for the solving of conflicts and the standard *NFA2DFA* algorithm. This gives hope for other approaches being able to tackle this problem, even for large grammars.

7.1.2 Implementation

The individual parts of the algorithm find use in multiple subjects (Graph theory, language engineering, etc). Therefore implementing all these separate parts or not of waste. Rascal did not have functions for computing strongly connected components, nor a highlighter datatype. These and others are examples of things that were created in the process of this algorithm, but reusable for other purposes.

Another strong point is the fact that the parts of the highlighters that do work benefit from the parser like behaviour. In handwritten highlighters you often have to delimit matches with word endings (`\b`). These make sure that words like "for" inside therefore do not get highlighted. The generated highlighters do not have this problem since it knows thanks to the context that its parsing something that is not a keyword.

7.2 Weak points

7.2.1 Algorithm

Due to the nature of state-based syntax highlighters (in at least Sublime) the approach taken will often crash on encountering end-of-lines. This leads to premature popping of contexts and bad highlighting.

Secondly, the above error occurs from using lookahead in the highlighter to determine what to do next. This is generally okay, since other highlighters do this as

well, however one should be careful in using this. Especially lookbehind and negative lookahead or behind are dangerous to use without an actual match. Since not (something) is true for lots of cases (E.G. `'(?!for)'` returns true for empty string). Thirdly, the algorithm to resolve conflicts in the generated statemachines is very inefficient. So bad that for larger grammars the algorithm will take too long to complete. Making it useless on larger grammars.

Finally, all of the above (and several other) issues with the algorithm would be solved if the algorithm would not induce the parser-like behaviour of the highlighters. Stopping the literal parsing removes lots of the matching and lookahead regular expressions always resulting in less possible errors. The generating of machines with all the conflicts is not needed if the highlighter does not care about most of the grammar. So these machines will not be generated and therefore no infinite waiting for conflict resolution.

7.2.2 Implementation

There are also a number of things to be said about the implementation. The realization that the approach would not work came too late and therefore there was no time to retry a different approach. Thanks to this the code for *ToContexts* was not written in a very good way. Long functions lots of long lines to just get it to work. Time became an issue here and therefore the module *ToContexts* suffered.

Another weak point about the implementation is the way that scopes are saved and then reapplied to the final contexts. As mentioned before the current implementation only allows one single context for a single token, wherever it occurs in the grammar. This is not desirable.

On top of that is the conflict resolution algorithm substituting the non-terminal tokens on arcs of the machines. However if this non-terminal had a scope assigned to it, this information is lost in the process. There should be a better way of keeping track where and to what which information belongs.

7.3 Overall

I believe that this project set out all wrong. This is partially to blame on me as researcher, but largely on the nature of the problem. I started off with no knowledge of state-based highlighters and how they work, nor any prior experience in this domain. This makes anticipating future problems very hard. After this project it becomes clear that this approach does not work and why it does not work. Due to limitations in time there was no time to revert the changes and take a different approach.

This thesis and research shows that this approach is not the way to go as it cannot produce proper highlighters for even simple grammars. This is largely due to the parser-like nature of the generated highlighters. I believe that fixing this, together with a slightly different way of embedding information could proof future researchers a great deal in tackling the same problem once again, but from a different angle. In this new approach much of the created algorithm can be reused in one way or the other, paving the path for future solutions to the problem at hand.

Chapter 8

Suggestions for Future Work

In this section I will describe a number of ideas for future research into this question. As the problem is not solved I will focus on ideas to solve this problem and less on follow-up research for after this problem.

The first and foremost problem to be solved is to prevent the parser-like behaviour. There needs to be a more general approach such that the highlighters keep highlighting when syntax errors occur. The simplest approach would be to make a number of contexts that highlight specific cases and to include all of these in the main context. This translates to something like a context for keywords, a (number of) context(s) to highlight strings, same for comments, etc. Then from the main context include all starting-contexts for these concepts. Possible problems for this approach could be that it becomes hard to determine when to nest contexts and how to do this nesting. Think of the example of function-definition identifiers getting a different colour from variable declarations.

A second idea could be to embed slightly different information from before. Instead of assigning scopes to the tokens, it might help to let the grammar-writer denote the starting and ending delimiters of a scope. Much like TextMate does in its highlighters. Taking this approach removes the passive popping of contexts because nothing matches, into a more active popping of contexts. Being: If match this, then pop, else stay inside context.

There is just a general different approach needed to tackle the problem. In future work there might not be a lot of use for the strongly regular grammars either. The transformation to these grammars are intended to keep as much of the original structure as possible, however there is no real need for that, because highlighters should not parse. When writing special highlighters that incorporate jump-to-definition and method-folding and such there might be more reason to want this, however this project is focused on just creating highlights and not those other features.

Appendix A

Grammars

A.1 Grammar specifications

A.1.1 Simple example grammar

```
1 start syntax S
2   = A B
3   ;
4
5 lexical A
6   = "a" "a"
7   ;
8
9 lexical B
10  = "b" "b"
11  ;
12
13 layout Layout = [ \ ]*;
```

LISTING A.1: A simple example grammar

A.1.2 Mohri and Nederhof's example

```

1 start syntax E
2   = E "+" T
3   | T
4   ;
5
6 syntax T
7   = T "*" F
8   | F
9   ;
10
11 syntax F
12   = "(" E ")"
13   | "a"
14   ;
15
16 /* ----- Into ----- */
17
18 syntax T_end =
19   "*" F
20   | E_end
21   ;
22
23 syntax T =
24   F
25   | T
26   ;
27
28 syntax E_end =
29   "+" T
30   | ")" F_end
31   |
32   ;
33
34 syntax F_end =
35   T_end
36   ;
37
38 start syntax E =
39   T
40   | E
41   ;
42
43 syntax F =
44   "a" F_end
45   | "(" E
46   ;

```

LISTING A.2: The example grammar that is used in Mohri and Nederhof, 2000 with corresponding outputted grammar from the *ToStronglyRegularGrammar* module

Appendix B

Datatypes

```

1 data Symbol
2   = \start(Symbol symbol)
3
4   | \sort(str name)
5   | \lex(str name)
6   | \layouts(str name)
7   | \keywords(str name)
8   | \parameterized-sort(str name, list[Symbol] parameters)
9   | \parameterized-lex(str name, list[Symbol] parameters)
10
11  | \lit(str string)
12  | \cilit(str string)
13  | \char-class(list[CharRange] ranges)
14
15  | \empty()
16  | \opt(Symbol symbol)
17  | \iter(Symbol symbol)
18  | \iter-star(Symbol symbol)
19  | \iter-seps(Symbol symbol, list[Symbol] separators)
20  | \iter-star-seps(Symbol symbol, list[Symbol] separators)
21  | \alt(set[Symbol] alternatives)
22  | \seq(list[Symbol] symbols)
23
24  | \conditional(Symbol symbol, set[Condition] conditions)
25 ;

```

LISTING B.1: The Symbol datatype of Rascal

```

1 data Condition
2   = \follow(Symbol symbol)
3   | \not-follow(Symbol symbol)
4   | \precede(Symbol symbol)
5   | \not-precede(Symbol symbol)
6   | \delete(Symbol symbol)
7   | \at-column(int column)
8   | \begin-of-line()
9   | \end-of-line()
10  | \except(str label)
11 ;

```

LISTING B.2: The Condition datatype of Rascal

```

1 data Production
2   = prod(Symbol def, list[Symbol] symbols, set[Attr] attributes)
3   | regular(Symbol def)
4   | error(Production prod, int dot)

```

```

5 | skipped()
6 | \priority(Symbol def, list[Production] choices)
7 | \associativity(Symbol def, Associativity \assoc, set[Production]
  | alternatives)
8 | \reference(Symbol def, str cons)
9 | \choice(Symbol s, set[Production] choices)
10 ;

```

LISTING B.3: The Production datatype of Rascal

```

1 alias State = str;
2 alias Token = Symbol;
3 alias DeltaFunction = rel[State from, Token token, State to];
4 alias Transition = tuple[State from, Token token, State to];
5
6 data StateMachine
7   = \sm-dfa(DeltaFunction delta, State q0, set[State] F)
8   | \sm-nfa(DeltaFunction delta, State q0, set[State] F)
9   ;

```

LISTING B.4: The StateMachine datatype in Rascal

```

1 data SHRegex
2   = \str-regex(str regex)
3   ;
4
5 data SyntaxHighlighter
6   = \highlighter(str name, set[str] extensions,
7     set[SHvar] vars, map[str, Context] contexts)
8   ;
9
10 data SHvar
11   = \shvar(str name, SHRegex regex);
12
13 data Context
14   = \context(str name, Scope scope, set[Match] matches,
15     set[Context] includes, bool includePrototype=true)
16   | \main(Scope scope, set[Match] matches, set[Context] includes)
17   | \prototype(Scope scope, set[Match] matches, set[Context] includes)
18   ;
19
20 data Scope
21   = \scope(str name)
22   | \meta-scope(str name)
23   | \null()
24   ;
25
26 data Action
27   = \push(list[str] contexts)
28   | \setact(list[str] contexts)
29   | \pop()
30   | \noact()
31   ;
32
33 data Match
34   = \match(SHRegex regex, Scope scope, Action action)
35   ;

```

LISTING B.5: The SyntaxHighlighter datatype in Rascal

Appendix C

Advanced Example Grammar

C.0.3 Original definition

```

1 // layout is lists of whitespace characters
2 layout MyLayout = [\t\n\ \r\f]*;
3
4 // identifiers are characters of lowercase alphabet letters,
5 // not immediately preceded or followed by those (longest match)
6 // and not any of the reserved keywords
7 lexical Identifier = [a-z] !<< [a-z]+ !>> [a-z] \ MyKeywords;
8
9 // this defines the reserved keywords used in the definition of Identifier
10 keyword MyKeywords = "if" | "then" | "else" | "fi";
11
12 // here is a recursive definition of expressions
13 // using priority and associativity groups.
14 syntax Expression
15   = id: Identifier id
16   | null: "null"
17   | bracket "(" Expression ")"
18   > left multi: Expression l "*" Expression r
19   > left ( add: Expression l "+" Expression r
20           | sub: Expression l "-" Expression r
21           )
22   ;

```

LISTING C.1: A grammar that uses almost all constructs. This is a slightly modified version of the one found here: *Rascal Syntax definition*

C.0.4 With context information

```

1 layout MyLayout = [\t\n\ \r\f]*;
2
3 lexical Identifier = [a-z] !<< [a-z]+ !>> [a-z] \ MyKeywords;
4
5 keyword MyKeywords
6   = @@Context="keyword.flow.conditional">@ "if"
7   | @@Context="keyword.flow.conditional">@ "then"
8   | @@Context="keyword.flow.conditional">@ "else"
9   | @@Context="keyword.flow.conditional">@ "fi"
10  ;
11
12 syntax Expression
13   = id: Identifier id
14   | @@Context="constant.language">@ null: "null"
15   | bracket "(" Expression ")"

```

```

16 > @@Context="null keyword.operator.arithmetic null">@
17   left multi: Expression l "*" Expression r
18 > left ( add: Expression l "+" Expression r
19         | sub: Expression l "-" Expression r
20         )
21 ;

```

LISTING C.2: The original grammar with contexts

C.0.5 Simplified without priority

```

1 keyword MyKeywords =
2   = @@Context="keyword.flow.conditional">@ "else"
3   | @@Context="keyword.flow.conditional">@ "if"
4   | @@Context="keyword.flow.conditional">@ "fi"
5   | @@Context="keyword.flow.conditional">@ "then"
6   ;
7
8 layout MyLayout = [\t-\n \a0C-\a0D \ ]*;
9
10 lexical Identifier = [a-z] !<< [a-z]+ !>> [a-z] \ MyKeywords;
11
12 syntax Expression
13   = @@Context="constant.language">@ "null"
14   | Identifier
15   | bracket "(" Expression ")"
16   | @@Context="null keyword.operator.arithmetic null">@
17     Expression "*" Expression
18   | Expression "+" Expression
19   | Expression "-" Expression
20   ;

```

LISTING C.3: The same grammar but simplified with priority removal

C.0.6 Simplified with priority

```

1 keyword MyKeywords =
2   = @@Context="keyword.flow.conditional">@ "else"
3   | @@Context="keyword.flow.conditional">@ "if"
4   | @@Context="keyword.flow.conditional">@ "fi"
5   | @@Context="keyword.flow.conditional">@ "then"
6   ;
7
8 layout MyLayout = [\t-\n \a0C-\a0D \ ]*;
9
10 lexical Identifier = [a-z] !<< [a-z]+ !>> [a-z] \ MyKeywords;
11
12 syntax Expression_2
13   = Identifier
14   | @@Context="constant.language">@ "null"
15   | bracket "(" Expression ")"
16   ;
17
18 syntax Expression_1
19   = Expression_2
20   | @@Context="null keyword.operator.arithmetic null">@
21     Expression_1 "*" Expression_1
22   ;

```

```

23
24 syntax Expression
25   = Expression_1
26   | Expression "+" Expression
27   | Expression "-" Expression
28   ;

```

LISTING C.4: The same simplified grammar with priority preservation

C.0.7 Strongly Regular

```

1 syntax Expression_end
2   = "+" Expression
3   | ")" Expression_end
4   | Expression_end
5   | "*" Expression
6   | "_" Expression
7   |
8   ;
9
10 keyword MyKeywords = "else" | "if" | "fi" | "then";
11
12 layout MyLayout = [\t-\n \a0C-\a0D \]*;
13
14 lexical Identifier = [a-z] !<< [a-z]+ !>> [a-z] \ MyKeywords;
15
16 syntax Expression
17   = Expression
18   | "null" Expression_end
19   | Identifier Expression_end
20   | Expression
21   | bracket "(" Expression
22   ;

```

LISTING C.5: The strongly regular approximation of the original grammar

C.0.8 NFA for strongly regular Expression



FIGURE C.1: The NFA for Expression, produced from the strongly regular approximation

C.0.9 DFA for strongly regular Expression

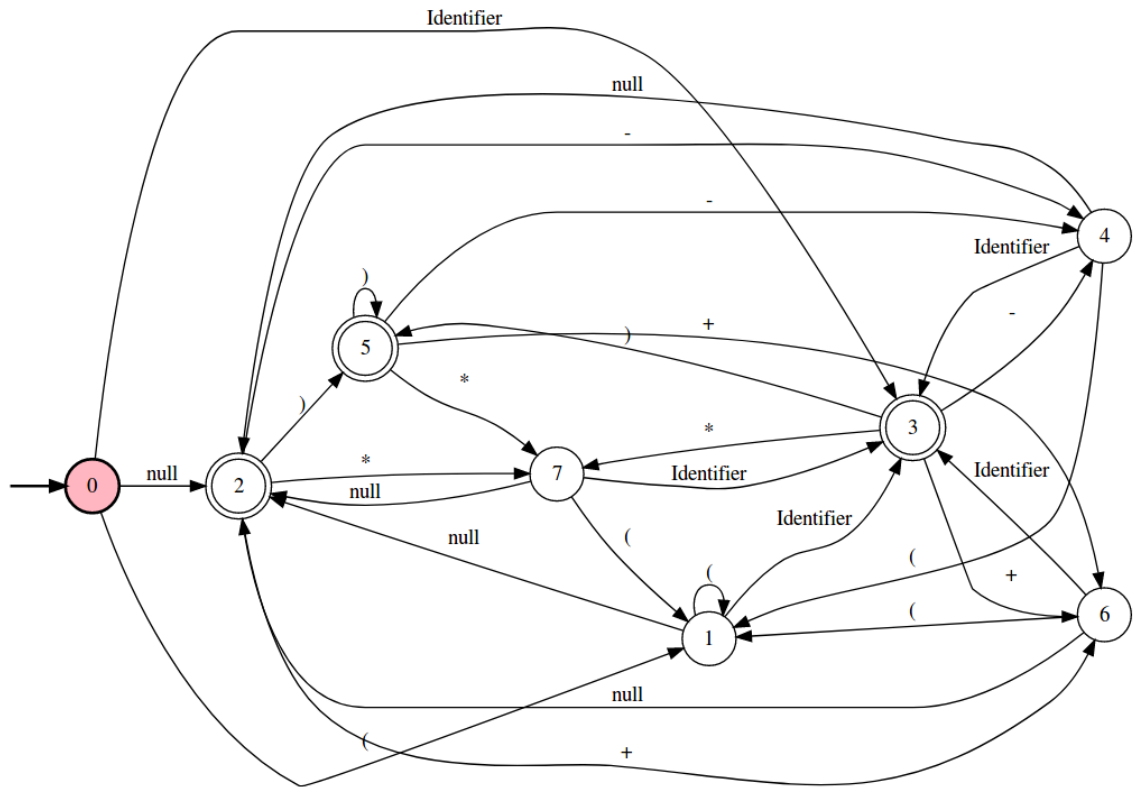


FIGURE C.2: The DFA for Expression, produced from the strongly regular approximation

Appendix D

Nested Comments Grammar

D.1 Original definition

```

1 start syntax C = @@Context="comment.block">@ Comment;
2
3 lexical Comment = "/*" (ComChar | Comment)* "*/";
4
5 lexical ComChar
6   = ![*/]
7   | [*] !>> [/]
8   | [/] !>> [*]
9   ;

```

LISTING D.1: A grammar showing nested comments

D.2 Simplified without priority

```

1 lexical ComChar =
2   ![* /]
3   | [*] !>> [/]
4   | [/] !>> [*]
5   ;
6
7 lexical Comment =
8   "/*" ComChar_Comment_ALT_STAR "*/"
9   ;
10
11 syntax ComChar_Comment_ALT =
12   ComChar
13   | Comment
14   ;
15
16 start syntax C
17   = @@Context="comment.block">@ Comment
18   ;
19
20 lexical ComChar_Comment_ALT_STAR =
21   ComChar_Comment_ALT ComChar_Comment_ALT_STAR
22   |
23   ;

```

LISTING D.2: The same grammar but simplified with priority removal

D.3 Strongly Regular

```

1 syntax ComChar_Comment_ALT_end =
2   ComChar_Comment_ALT_STAR
3   ;
4
5 lexical ComChar =
6   ![* /]
7   | [*] !>> [/]
8   | [/] !>> [*]
9   ;
10
11 lexical Comment_end
12   =
13   | ComChar_Comment_ALT_end
14   ;
15
16 lexical Comment =
17   "/*" ComChar_Comment_ALT_STAR
18   ;
19
20 syntax ComChar_Comment_ALT =
21   Comment
22   | ComChar ComChar_Comment_ALT_end
23   ;
24
25 start syntax C = Comment;
26
27 lexical ComChar_Comment_ALT_STAR_end =
28   "*/" Comment_end
29   | ComChar_Comment_ALT_STAR_end
30   ;
31
32 lexical ComChar_Comment_ALT_STAR =
33   ComChar_Comment_ALT
34   | ComChar_Comment_ALT_STAR_end
35   ;

```

LISTING D.3: The strongly regular approximation of the original grammar

D.3.1 NFA for strongly regular Comment

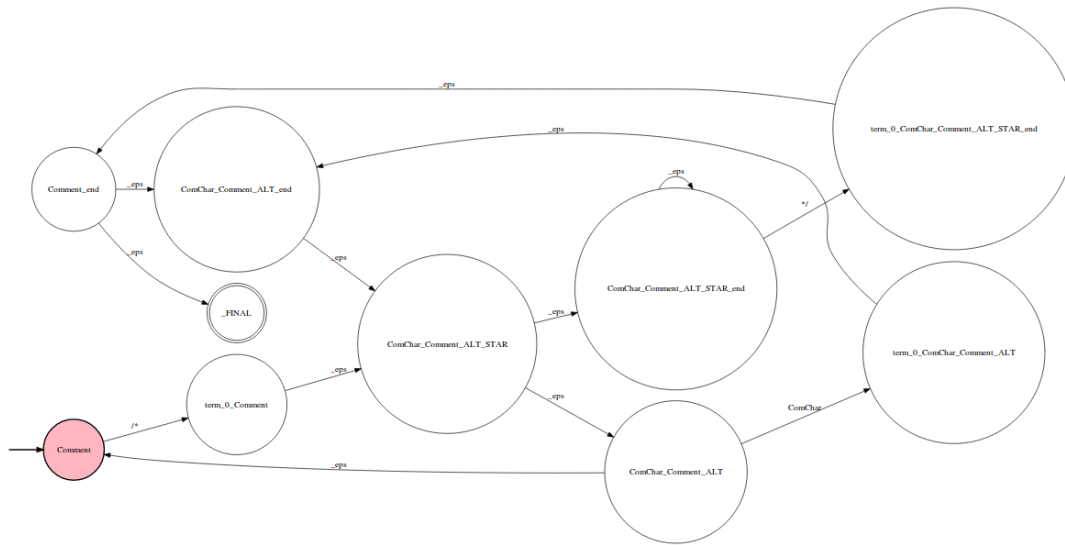


FIGURE D.1: The NFA for Comment, produced from the strongly regular approximation

D.3.2 DFA for strongly regular Comment

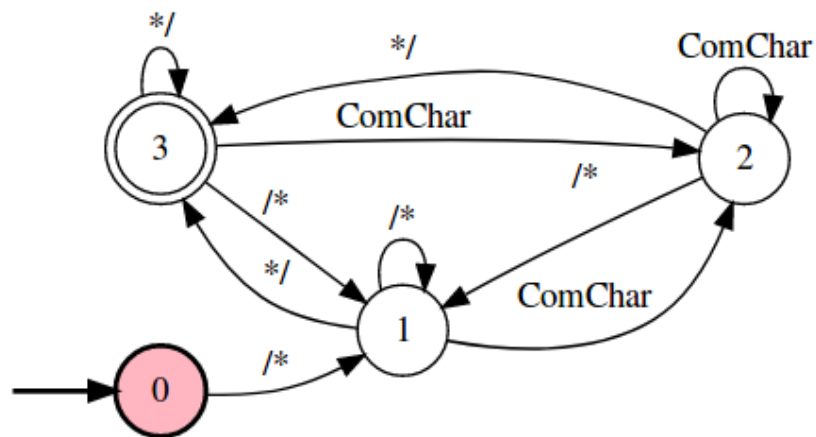


FIGURE D.2: The DFA for Comment, produced from the strongly regular approximation

Appendix E

String Interpolation Grammar

E.1 Original definition

```

1 lexical Interp = ![\>]+;
2
3 start syntax String = StringLiteral;
4
5 lexical StringLiteral
6   = @@Context="string.quoted.double">@ "\"" StringBody "\""
7   ;
8
9 lexical StringBody
10  = @@Context="string.quoted.double">@
11    ("\\<" | "\\\"" | "\\>" | ![" \< \>])
12    | @@Context="string.quoted.double null null null string.quoted.double">@
13      StringBody "<" Interp ">" StringBody
14    ;

```

LISTING E.1: A simple string-interpolation language as a Rascal grammar

E.1.1 Simplified without priority

```

1 lexical Interp = ![\>]+;
2
3 start syntax String = StringLiteral;
4
5 lexical StringLiteral
6   = @@Context="string.quoted.double">@ "\"" StringBody "\""
7   ;
8
9 lexical StringBody
10  = @@Context="string.quoted.double">@
11    ( "\\<" | "\\\"" | "\\>" | ![" \< \>])
12    | @@Context="string.quoted.double null null null string.quoted.double">@
13      StringBody "<" Interp ">" StringBody
14    ;

```

LISTING E.2: The plain grammar with priority removal

E.1.2 Strongly Regular

```

1 lexical Interp = ![\>]+;
2
3 start syntax String =
4   StringLiteral
5   ;
6
7 lexical StringLiteral = "\"" StringBody "\"";
8
9 lexical StringBody
10  = ("\\>" | "\\\"" | "\\<" | ![" \< \>])+ StringBody_end
11  | StringBody
12  ;
13
14 lexical StringBody_end
15  =
16  | "\" Interp "\" StringBody
17  | StringBody_end
18  ;

```

LISTING E.3: The strongly regular approximation of the original grammar

E.1.3 NFA for strongly regular StringLiteral

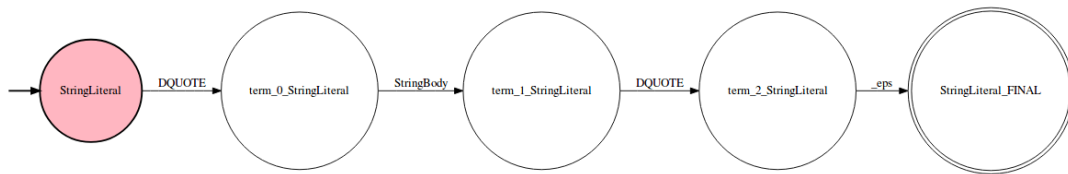


FIGURE E.1: The NFA for StringLiteral, produced from the strongly regular approximation

E.1.4 DFA for strongly regular StringLiteral

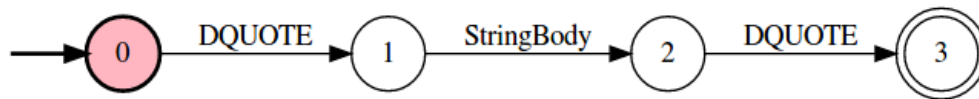


FIGURE E.2: The DFA for StringLiteral, produced from the strongly regular approximation

E.1.5 Results of generated highlighter

```
~/Desktop/CS/Year 3/Blok IIb/bachelor_project/coding/projects/tryout/highlighter/interp.... - + x
File Edit Selection Find View Goto Tools Project Preferences Help

interp.str  nestedcomments.comm x  picoprogram.pico x
1 An actual functional example
2 "now this is a string"
3 this is not a string
4 "this is an <interpolated> string and <it works> multiple times"
5
6 "we can escape chars \< \" \> "
7 "not on multiple
8 lines though since it ends the match at a $
9 Then continues to <expression> the next state"
10
11 " bla bla multiline interpolation < will also succeed
12 | some random interpolated stuff
13 > empty strings not supported"
14
15
Line 15, Column 1 Tab Size: 4 String
```

FIGURE E.3: Results of the highlighter generated for this grammar

E.1.6 Results of hand-written highlighter

```
~/Desktop/CS/Year 3/Blok IIb/bachelor_project/coding/projects/tryout/highlighter/interp.... - + x
File Edit Selection Find View Goto Tools Project Preferences Help

interp.str  nestedcomments.comm x  picoprogram.pico x
1 An actual functional example
2 "now this is a string"
3 this is not a string
4 "this is an <interpolated> string and <it works> multiple times"
5
6 "we can escape chars \< \" \> "
7 "not on multiple
8 lines though since it ends the match at a $
9 Then continues to <expression> the next state"
10
11 " bla bla multiline interpolation < will also succeed
12 | some random interpolated stuff
13 > empty strings not supported"
14
15
Line 9, Column 36 Tab Size: 4 StringHand
```

FIGURE E.4: Results of a handwritten highlighter for this grammar

Appendix F

Pico Grammar

F.1 Original definition

```

1 start syntax Program
2   = @@Context="keyword.control.flow null null keyword.control.flow">@
3     "begin" Declarations {Statement ";"* "end" ;
4
5 syntax Declarations
6   = @@Context="keyword.declaration null">@ "declare" {IdType ","}* ";"
7   ;
8
9 syntax IdType = Id ":" Type;
10
11 syntax Statement
12   = Id "!=" Expression
13   | @@Context="keyword.control.flow null keyword.control.flow null keyword.
14     control.flow null keyword.control.flow">@
15     "if" Expression "then" {Statement ";"}* "else" {Statement ";"* "fi"
16     | "if" Expression "then" {Statement ";"* "fi"
17     | @@Context="keyword.control.flow null keyword.control.flow null keyword.
18     control.flow">@
19     "while" Expression "do" {Statement ";"* "od"
20   ;
21
22 syntax Type
23   = @@Context="storage.type">@ "natural"
24   | @@Context="storage.type">@ "string"
25   | @@Context="storage.type">@ "nil-type"
26   ;
27
28 syntax Expression
29   = Id
30   | @@Context="string.quoted.double">@ String
31   | @@Context="constant.numeric.integer">@ Natural
32   | bracket "(" Expression ")"
33   > left Expression "|" Expression
34   > left ( Expression "+" Expression
35         | Expression "-" Expression
36         )
37   ;
38
39 layout Layout = WhitespaceAndComment* !>> [\ \t\n\r%];
40
41 lexical WhitespaceAndComment
42   = [\ \t\n\r]

```

```

41 | @@Context="comment.block">@ "%" ![%]+ "%"
42 | @@Context="comment.line">@ "%%" ![\n]* $
43 ;
44
45 lexical Id = ([a-z][a-z0-9]*) !>> [a-z0-9] \ Keywords;
46 lexical Natural = [0-9]+ ;
47 lexical String = "\"\" ![\"]* "\"";

```

LISTING F.1: The Pico language as a Rascal grammar

F.1.1 Simplified without priority

```

1 syntax Type
2   = @@Context="storage.type">@ "string"
3   | @@Context="storage.type">@ "nil-type"
4   | @@Context="storage.type">@ "natural"
5   ;
6
7 lexical Id = ([a-z] [0-9 a-z]*) !>> [0-9 a-z];
8
9 lexical WhitespaceAndComment =
10  [\t-\n \a0D \ ]
11  | @@Context="comment.block">@ "%" ![%]+ "%"
12  | @@Context="comment.line">@ "%%" ![\n]*$
13  ;
14
15 syntax Statement =
16  Id ":" Expression
17  | @@Context="keyword.control.flow null keyword.control.flow null keyword.
18    control.flow null keyword.control.flow">@
19    "if" Expression "then" Statement__;"__STARSEPS "else" Statement__;"
20    __STARSEPS "fi"
21  | @@Context="keyword.control.flow null keyword.control.flow null keyword.
22    control.flow">@
23    "while" Expression "do" Statement__;"__STARSEPS "od"
24  | "if" Expression "then" Statement__;"__STARSEPS "fi"
25  ;
26
27 syntax Statement__;"__STARSEPS_TAIL =
28  ";" Statement__;"__STARSEPS
29  ;
30
31 syntax IdType__;"__STARSEPS_TAIL
32  =
33  | ";" IdType__;"__STARSEPS
34  ;
35
36 syntax Declarations
37  = @@Context="keyword.declaration null">@ "declare" IdType__;"__STARSEPS
38    ";"
39  ;
40
41 syntax Expression =
42  bracket "(" Expression ")"
43  | Expression "||" Expression
44  | @@Context="string.quoted.double">@ String
45  | @@Context="constant.numeric.integer">@ Natural

```

```

43 | Id
44 | Expression "+" Expression
45 | Expression "-" Expression
46 ;
47
48 syntax IdType =
49   Id ":" Type
50 ;
51
52 layout Layout =
53   WhitespaceAndComment_STAR !>> [\t-\n \a0D \ %]
54 ;
55
56 syntax IdType__","__STARSEPS =
57   IdType IdType__","__STARSEPS_TAIL
58 |
59 ;
60
61 syntax Statement__";"__STARSEPS
62   = Statement Statement__";"__STARSEPS_TAIL
63 |
64 ;
65
66 lexical Natural =
67   [0-9]+
68 ;
69
70 start syntax Program
71   = @@Context="keyword.control.flow null null keyword.control.flow">@
72     "begin" Declarations Statement__";"__STARSEPS "end"
73 ;
74
75 lexical WhitespaceAndComment_STAR
76   =
77   | WhitespaceAndComment WhitespaceAndComment_STAR
78 ;
79
80 lexical String = "\"\" ![\"]* \"";

```

LISTING F.2: The plain grammar with priority removal

F.1.2 Strongly Regular

```

1 syntax Type =
2   "nil-type"
3   | "natural"
4   | "string"
5   ;
6
7 syntax Statement__";"__STARSEPS_end =
8   "od" Statement_end
9   | Statement__";"__STARSEPS_TAIL_end
10  |
11  | "fi" Statement_end
12  | "else" Statement__";"__STARSEPS
13  ;
14
15 lexical Id =

```

```

16 ( [a-z] [0-9 a-z]* ) !>> [0-9 a-z]
17 ;
18
19 syntax Expression_end =
20   "-" Expression
21   | ")" Expression_end
22   | "+" Expression
23   | "||" Expression
24   | Expression_end
25   |
26   ;
27
28 syntax Statement_end =
29   Statement__ ";"__STARSEPS_TAIL
30   ;
31
32 lexical WhitespaceAndComment =
33   "%%" ![\n]*$
34   | "%" ![%]+ "%"
35   | [\t-\n \aOD \ ]
36   ;
37
38 syntax Statement =
39   "while" Expression "do" Statement__ ";"__STARSEPS
40   | "if" Expression "then" Statement__ ";"__STARSEPS
41   | Id ":" Expression Statement_end
42   ;
43
44 syntax Statement__ ";"__STARSEPS_TAIL =
45   ";" Statement__ ";"__STARSEPS
46   | Statement__ ";"__STARSEPS_TAIL_end
47   ;
48
49 syntax IdType__ ","__STARSEPS_TAIL
50   =
51   | "," IdType__ ","__STARSEPS
52   ;
53
54 syntax Declarations =
55   "declare" IdType__ ","__STARSEPS ";"
56   ;
57
58 syntax Expression =
59   "(" Expression
60   | Natural Expression_end
61   | String Expression_end
62   | Id Expression_end
63   | Expression
64   ;
65
66 syntax IdType =
67   Id ":" Type
68   ;
69
70 layout Layout =
71   WhitespaceAndComment_STAR !>> [\t-\n \aOD \ %]
72   ;

```

```

73
74 syntax IdType__","__STARSEPS =
75   IdType IdType__","__STARSEPS_TAIL
76   |
77   ;
78
79 syntax Statement__";"__STARSEPS_TAIL_end =
80   Statement__";"__STARSEPS_end
81   ;
82
83 syntax Statement__";"__STARSEPS =
84   Statement
85   | Statement__";"__STARSEPS_end
86   ;
87
88 lexical Natural =
89   [0-9]+
90   ;
91
92 start syntax Program =
93   "begin" Declarations Statement__";"__STARSEPS "end"
94   ;
95
96 lexical WhitespaceAndComment_STAR =
97   WhitespaceAndComment WhitespaceAndComment_STAR
98   |
99   ;
100
101 lexical String = "\" ![\"]* \"";

```

LISTING F.3: The strongly regular approximation of the original grammar

F.1.3 NFA for strongly regular WhitespaceAndComment

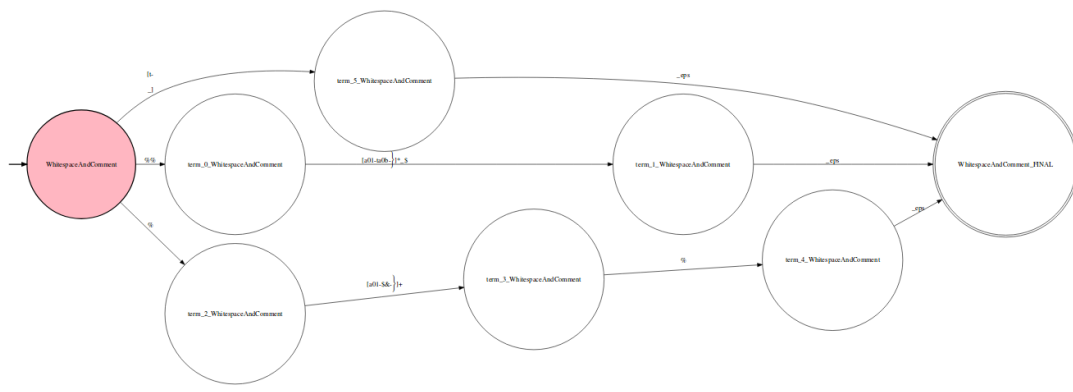


FIGURE F.1: The NFA for WhitespaceAndComment, produced from the strongly regular approximation. Used in the prototype context.
The weird characterclasses are equal to `![%]` and `![\ n]`

F.1.4 DFA for strongly regular WhitespaceAndComment

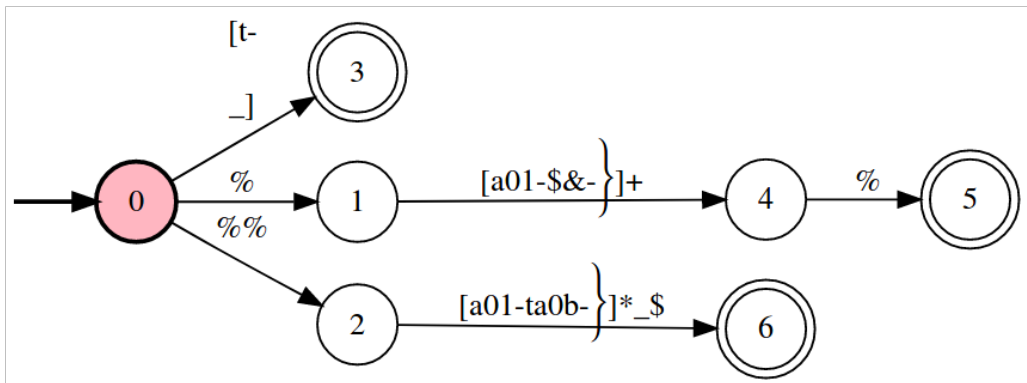
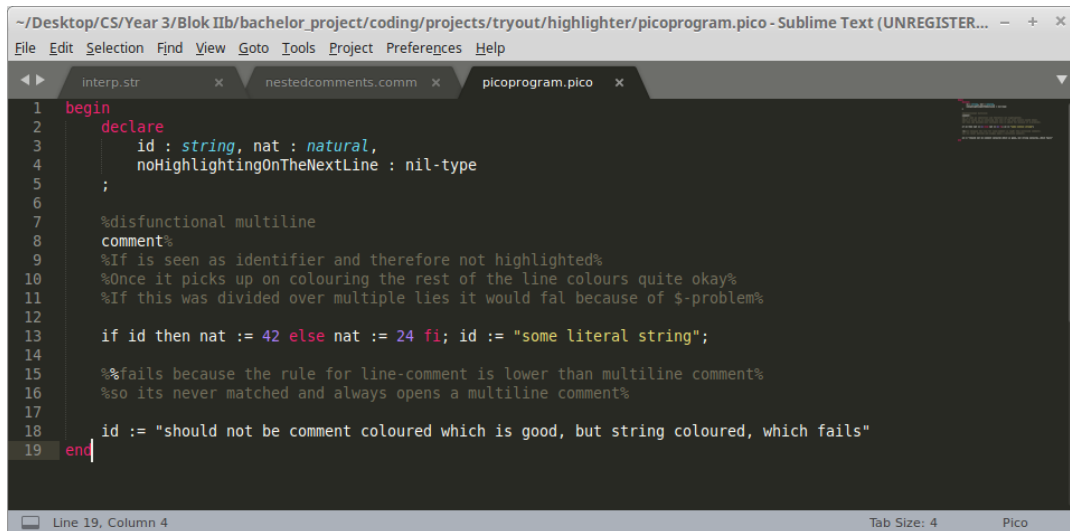


FIGURE F.2: The DFA for WhitespaceAndComment, produced from the strongly regular approximation. Used in the prototype context.
The weird characterclasses are equal to `![%]` and `![\ n]`

F.1.5 Results of generated highlighter



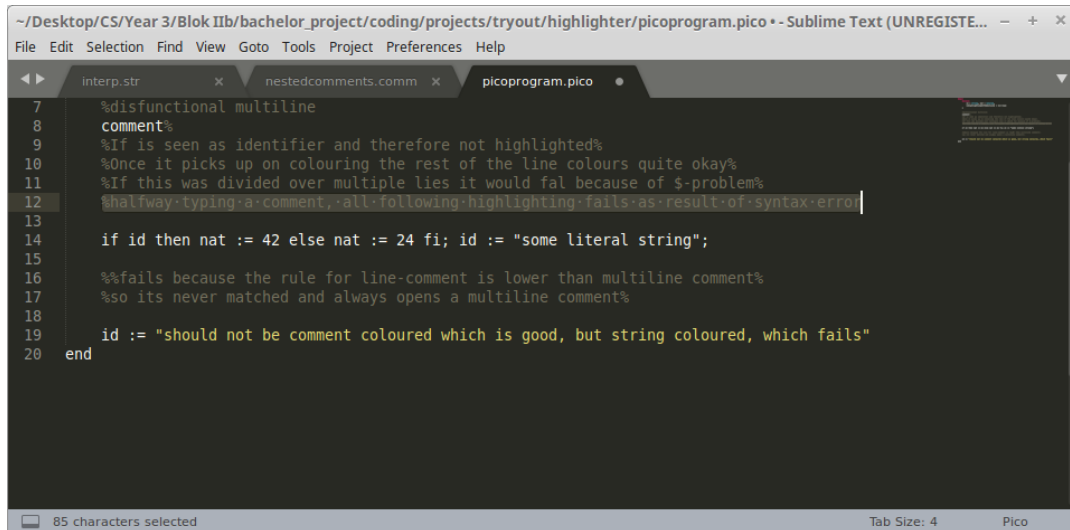
The screenshot shows a Sublime Text editor window with the file path `~/Desktop/CS/Year 3/Blok IIb/bachelor_project/coding/projects/tryout/highlighter/picoprogram.pico`. The editor displays a Pico program with syntax highlighting. The code is as follows:

```
1 begin
2   declare
3     id : string, nat : natural,
4     noHighlightingOnTheNextLine : nil-type
5   ;
6
7   %disfunctional multiline
8   comment%
9   %If is seen as identifier and therefore not highlighted%
10  %Once it picks up on colouring the rest of the line colours quite okay%
11  %If this was divided over multiple lines it would fail because of $-problem%
12
13  if id then nat := 42 else nat := 24 fi; id := "some literal string";
14
15  %%fails because the rule for line-comment is lower than multiline comment%
16  %so its never matched and always opens a multiline comment%
17
18  id := "should not be comment coloured which is good, but string coloured, which fails"
19 end
```

The status bar at the bottom indicates "Line 19, Column 4", "Tab Size: 4", and "Pico".

FIGURE F.3: Results of the highlighter generated for the pico grammar

F.1.6 Results of generated highlighter with syntax error



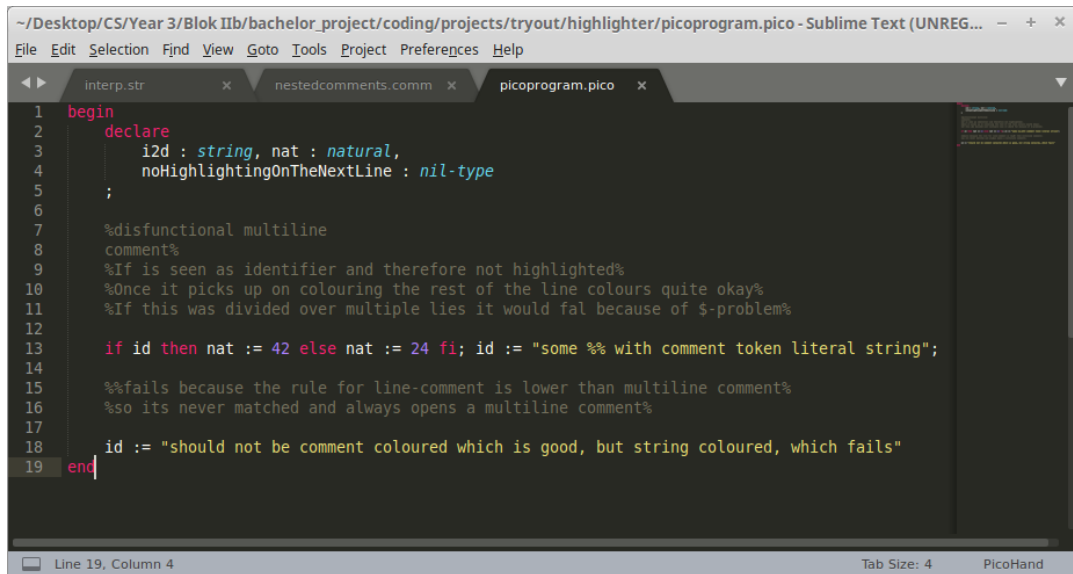
The screenshot shows the same Sublime Text editor window as Figure F.3, but with a syntax error. The code is the same as in Figure F.3, but with a comment on line 12 that is not properly closed. The error is highlighted in yellow:

```
7   %disfunctional multiline
8   comment%
9   %If is seen as identifier and therefore not highlighted%
10  %Once it picks up on colouring the rest of the line colours quite okay%
11  %If this was divided over multiple lines it would fail because of $-problem%
12  %halfway typing a comment, all following highlighting fails as result of syntax error
13
14  if id then nat := 42 else nat := 24 fi; id := "some literal string";
15
16  %%fails because the rule for line-comment is lower than multiline comment%
17  %so its never matched and always opens a multiline comment%
18
19  id := "should not be comment coloured which is good, but string coloured, which fails"
20 end
```

The status bar at the bottom indicates "85 characters selected", "Tab Size: 4", and "Pico".

FIGURE F.4: Results of the highlighter on a syntax-error

F.1.7 Results of hand-written highlighter



```
1 begin
2   declare
3     i2d : string, nat : natural,
4     noHighlightingOnTheNextLine : nil-type
5   ;
6
7   %disfunctional multiline
8   comment%
9   %If is seen as identifier and therefore not highlighted%
10  %Once it picks up on colouring the rest of the line colours quite okay%
11  %If this was divided over multiple lines it would fail because of $-problem%
12
13  if id then nat := 42 else nat := 24 fi; id := "some %% with comment token literal string";
14
15  %%fails because the rule for line-comment is lower than multiline comment%
16  %so its never matched and always opens a multiline comment%
17
18  id := "should not be comment coloured which is good, but string coloured, which fails"
19 end
```

FIGURE F.5: The DFA for StringLiteral, produced from the strongly regular approximation

Bibliography

- Chomsky, N. (1959). In: *Information and Control* 2, pp. 137–167.
- CWI. *Rascal Syntax definition*. URL: <http://tutor.rascal-mpl.org/Rascal/Rascal.html#/Rascal/Declarations/SyntaxDefinition/SyntaxDefinition.html>.
- *Rascal tutor documentation*. URL: <http://tutor.rascal-mpl.org/Rascal/Declarations/SyntaxDefinition/SyntaxDefinition.html>.
- (2014). *Rascal's website*. URL: <https://www.rascal-mpl.org/>.
- Ltd, Sublime HQ Pty. *Documentation: Syntax Definitions*. URL: <https://www.sublimetext.com/docs/3/syntax.html>.
- Mohri, M. and M.J. Nederhof (2000). “Regular Approximation of Context-Free Grammars through Transformation”. In: *Robustness in Language and Speech Technology*. Kluwer Academic Publishers. Chap. 9, pp. 251–261.
- Sharir, M. (1981). “A strong-connectivity algorithm and its applications in data flow analysis”. In: *Computers and Mathematics with Applications* 7(1), pp. 67–72.
- TextMate. *Language Grammars*. URL: https://manual.macromates.com/en/language_grammars.