A Generator of Divide-and-Conquer Lexers

A Tool to Generate an Incremental Lexer from a Lexical Specification

Master of Science Thesis [in the Programme MPALG]

JONAS HUGO

KRISTOFER HANSSON

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, July 2013

An not to long headline describing the content of the report
A Subtitle that can be Very Much Longer if Necessary
JONAS HUGO,
KRISTOFER HANSSON,

Examiner: NAME A. FAMILYNAME

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover:
an explanatory caption for the (possible) cover picture
with page reference to detailed information in this essay.

Department of Computer Science and Engineering
Göteborg, Sweden July 2013

Abstract

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

The Authors, Location 11/9/11

Contents

1

Introduction

This master-thesis is carried out at Chalmers, on the department of computer science.

1.1 Background

Editors normally have regular-expression based parsers, which are efficient and robust, but lack in precision: they are unable to recognize complex structures. Parsers used in compilers are precise, but typically not robust: they fail to recover after an error. They are also not efficient for editing purposes, because they have to parse files from the beginning, even if the user makes incremental changes to the input. More modern IDEs use compilerstrength parsers, but they give delayed feedback to the user. Building a parser with good characteristics is challenging: no system offers such a combination of properties

1.2 Scope of work

*Usage of BNFC *With help of regexp build a finit state machine that will lex a code string. *Give finite states with corresponding Monoid data type. *Flag for errors from the Lexer, give meningfull info to the user, and stop the worklow after lexer, until new updated text. *If no errors, handel layout *Parse the Monoid data type tree, AKA integrate the result with an existing parser. *Smile and be happy!

2

Lexer

A Lexer, lexical analyser, is a pattern matcher. It's job is to find sequence of characters in a larger string. The Lexer is a front end of a syntax analyser. [1] This can be done by using regular expressions, regular sets and finite automata. Which are central concepts in formal language theory. [2] All on which will be described in this chapter.

2.1 Lexing vs Parsing

There are several reasons why a compiler should be separated in to lexical analyser and a parser (syntax analyser) phases. Simplicity of design is the most important reason. When dividing the task in to these to sub task, it allows the system to simplify one of these sub-tasks. For example, a parser that has to deal with white-spaces and comments as syntactical units would be more complex then one that can assume white-spaces and comments have already been removed by an lexer. Also when the two tasks are divided into sub-tasks it can lead to cleaner overall design when designing a new language [3] Lexers work as a subprogram to the parser, giving it's result to the syntax analyser. So the only thing the syntax analyser will see is the output from the lexer, tokens and lexemes which will be described later in this chapter. [1] The lexer also skips comments and white-spaces, since these are not relevant for the syntax analyser. [1] Also overall efficiency of the compiler can be improved. When separating the lexical analyser it allows for appliance of specialised techniques that serve only the lexical task. [3] Last compiler portability can be enhanced. That is Input-device-specific peculiarities can be restricted to the lexical analysis. [3] So the lexer can detect syntactical errors in tokens, such as ill-formed floating-points literals, and report these errors to the user. [1] Breaking the compilation before running the syntax analyser, saving computing time.

2.2 Token Specification

This section will describe how to write rules for the tokens patterns.

2.2.1 Languages

An alphabet is an finite set of symbols, such an alphabet is for example the unicode, which includes approximately 100,000 characters. A language is any countable set of strings over some fixed alphabet. [3] The term formal languages refers to languages which can be described by a body of systematic rules. Like any formal language, a regular language is a set of strings. In other words a sequence of symbols, from a finite set of symbols. Only some formal languages are regular; in fact, regular languages are exactly those that can be defined by regular expressions. [4]

2.2.2 Regular Expressions

Say that we want to express the set of valid C identifiers. Use of regular expressions make it very easy, as shown in example 2.2.1.

Example 2.2.1 (Valid C Idents)**.** Say we have a element $letter \in \{\text{a}\ldots\text{z}\} \cup \{\text{A}\ldots\text{Z}\} \cup \{\_\}$ and another element $digit \in \{0\ldots9\}$ Then with help of regular expressions the definition of all valid C identifiers would look like this: $letter(letter|digit)*$. [3]

In definition 2.2.2 is the formal definition for regular expressions.

Definition 2.2.2 (Regular Expressions)**.** [2]

1. The following characters are meta characters $\{'|', '(', ')', '*'\}$.
2. A none meta character a is a regular expression that matches the string a.
3. If r_1 and r_2 are regular expressions then $(r_1|r_2)$ is a regular expression that matches any string that matches r_1 or r_2.
4. If r_1 and r_2 are regular expressions. $(r_1)(r_2)$ is a regular expression of the form that matches the string xy iff x matches r_1 and y matches r_2.
5. If r is a regular expression the $r*$ is a regular expression that matches any string of the form $x_1, x_2, \ldots, x_n, n \geq 0$. Where r matches x_i for $1 \leq i \leq n$, in particular $(r)*$ matches the empty string, ε.
6. If r is a regular expression, then (r) is a regular expression that matches the same string as r.

Many parentheses can be reduced by adopting the convention that the Kleene closure operator $*$ has the highest precedence, then concat and then or operator $|$. The two binary operators, cancat and $|$ are left left-associative. [2]

2.2.3 Regular Definitions

In a definition of a language it is useful to give regular expressions names, so they can for example be used in other regular expressions, as these names where themself symbols. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

$$\begin{array}{ccc} d_1 & \rightarrow & r_1 \\ d_2 & \rightarrow & r_2 \\ \vdots & \rightarrow & \vdots \\ d_n & \rightarrow & r_n \end{array}$$

where:

1. Each d_i is a new symbol, not in Σ and not the same as any other of the d's.
2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2 \dots d_{i-1}\}$

By restricting r_i to Σ and previously defined d's the regular definitions avoid recursive definitions. [3]

2.3 Tokens, Patterns and Lexemes

A lexical analyser uses three different terms. All which is described here below.

Token is a pair consisting of a token name and an optional attribute value. The token name is a abstract symbol corresponding to a lexical unit [3]. For example, a particular keyword, datatype or identifier. The token names is what is given to the parser.

Pattern is a description of what form a lexemes of a token may take. [3] For example, a keyword is just the sequence of characters that forms the keyword, an int is just a sequence consisting of just numbers.

Lexemes is a sequence of characters in the code that is being analysed which matches the pattern for a token and is identified by the lexical analyser as an instance of a token. [3]

As mention before a token consist of token name and a optional attribute value. This attribute is used when one lexeme can match more then one pattern. [3] For example the pattern for a digit token matches both 0 and 1, but it is important for the code generator to know which lexeme was found. Therefor the lexer often return not just the token but also an attribute value that describes the lexeme found in the source program corresponding to this token. [3] A lexer collects chars into logical groups and assign internal codes to these groups. according to there structure.[1] Where the groups of chars are lexemes and the internal codes are tokens. Here follows a example how a small piece of code would be divided.

Example 2.3.1 (Logical grouping). [1]
This is the code being lexed:

```
result = oldsum - value /100;
```

This is how it will be divided:

Token	Lexeme
IDENT	result
ASSING_OP	=
IDENT	oldsum
SUB_OP	−
IDENT	value
DIV_OP	/
INT_LIT	100
SEMICOLON	;

2.4 Recognition of Tokens

In previous section the topic have been, how to represent a pattern using regular expressions and how these expressions relates to tokens. This section will highlight how to transform a sequence of characters into a sequence of abstract tokens. First some basic understanding with transition diagrams.

2.4.1 Transition Diagrams

A state transition diagram, or just transition diagram is a directed graph. Where the nodes are labelled with the state name. Each node represent a state which could occur during the process of scanning the input looking for lexeme that matches one of several patterns.[3] The edges are labelled with the input characters that causes the transition among the states. An edge may also contain actions the lexer must perform when transition is token.[1] Here follows some properties for a transition diagram, One state is said to be initial state. The transition diagram always begins at this state, before any input symbols have been read. Some states are said to be accepting (final). They indicate that a lexeme has been found. The found token should then be returned with any additional optional values, mentioned in previous section.[3] Transition diagrams of the formed used in lexers are representations of a class of mathematical machines called finite automata. Finite automata can be designed to recognise members of a class of languages called regular languages, mentioned above #write about this shit in previus sections!!!! #. [1]

2.4.2 Finite Automata

A finite automata are essentially graphs, like transitions diagrams, with some differences:

- Finite automata are recognizers; they simply say "YES" or "NO" about each possible input string.
- Finite automata comes in to different forms:

  Nondeterministic Finite Automata (NFA) which have no restriction of the edges, several edges can be labelled by the same symbol out from the same state. ϵ, the empty string, is a possible label.

  Deterministic Finite Automata (DFA) for each state and for each symbol of its input alphabet exactly one edge with that symbol leaving that state

Both these forms of finite automate are capable of recognising the same language, called regular languages. These are languages that regular expressions can describe. [3] The formal definition of a finite automata follows:

Definition 2.4.1 (Finite Automata)**.** [5]
A finite automata is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the states,
2. Σ is a finite set called alphabet,
3. $\delta : Q \times \Sigma \to Q$ is a transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.

Nondeterministic Finite Automata

An NFA accept input x if and only if there is a path in the transition diagram from the start state to one of the accepting states. Such that the symbols along the way spells out x. [3]

There are different ways of representing an NFA, one is by transition diagrams another is by transitions table. This example shows how a regular expression can be converted in to a transition diagram.

Example 2.4.2 (RegExp to Transition Diagram)**.** [3]
Given this regular expression: $(a|b) * abb$
the transition diagram representing this regular expression is shown the figure below.

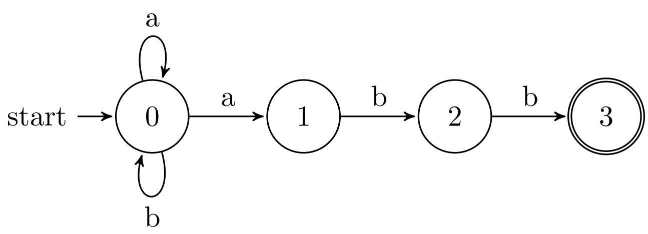

Figure 2.1: Transition Diagram, accepting the pattern $(a|b) * abb$

The following example shows how the transition table would look like for the same regular expression.

Example 2.4.3 (RegExp to Transition Table)**.** [3]
Given the regular expression from example 2.4.2 it can be converted into the following transition table:

State	a	b	ϵ
0	$\{0,1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

Figure 2.2: Transition Table representation of regular expression in example 2.4.2

Transition tables has the advantage that they have an quick lookup time. But instead it will take allot of data space, when the alphabet is large. Most states do not have any moves on most of the input symbols. [3]

Deterministic Finite Automata

DFA is a special case of an NFA where,

1. there are no moves on input ϵ and
2. for each state s and input symbol a, there is exactly one edge out of s labelled with a.

While NFA is an abstract representation of an algorithm to recognise the string of a language, the DFA is a simple concrete algorithm for recognising strings. Every regular expression can be converted in to a NFA and every NFA can be converted in to a DFA. [3] It is the DFA that is implemented and used when building lexical analysers.

Example 2.4.4 (DFA representation of RegExp)**.** [3]
A DFA representation of a regular expression shown in the following figure.

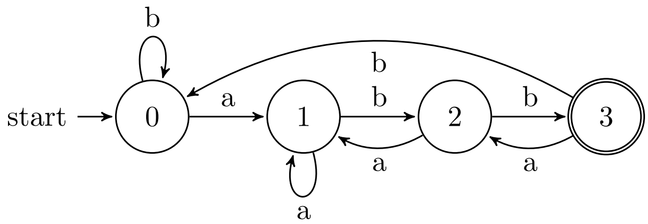

Figure 2.3: DFA, accepting regular expression in example 2.4.2

3

Divide-and-Conquer Lexer

An incremental lexer works by dividing the sequence, to be lexicaly analysed, into it's smallest part and analyse them and then combining them. In the base case the lexical analysis is done on a single character. The conquer step is then to combine the smaller tokens into as large tokens as possible. The end result should be a sequence of token that represent the code. How this is done will be described below. #Some ref to divide and conquer?

3.1 Lexing in the middle attack

When the code is divided the lexer doesn't know if the string (or character) it lexes is the first, last or is somewhere in the middle of a token. Instead of checking what type of token the string will be (if it were to begin from the starting state) it saves all the possible state transitions for that string.

In the examples that follow below state 0 is considered the starting state and state $1-6$ are considered accepting.

Example 3.1.1 (Transition map for a token)**.** A hypothetical transition map for the char 'i'.

$$
\begin{array}{cc}
\multicolumn{2}{c}{'i'} \\
in & out \\
0 & 1 \\
1 & 1 \\
8 & 7
\end{array}
$$

In the base case the lexer owill map all the transitions for all individual characters in the code and construct partial tokens of them. The conquer step will then combine two of these at a time by checking which possible outgoing states from the first token can be matched with incoming states from the second token. If there are such pairs of outgoing states with incomming states, then a new partial token is created.

Example 3.1.2 (Combining two tokens)**.** 'if' can be an ident (state 1) or part of 'else if' (state 5).

$$\begin{array}{cc} \multicolumn{2}{c}{'i'} \\ in & out \\ 0 & 1 \\ 1 & 1 \\ 8 & 7 \end{array} \text{ `}combineToken\text{` } \begin{array}{cc} \multicolumn{2}{c}{'f'} \\ in & out \\ 0 & 1 \\ 1 & 1 \\ 7 & 5 \end{array} = \begin{array}{cc} \multicolumn{2}{c}{'if'} \\ in & out \\ 0 & 1 \\ 8 & 5 \end{array}$$

If there are no pairs of outgoing states which match the incomming states the lexer will try to combine the first token with as much of the second token as possible. In this case there will be a remainder of the second token, The lexer can now be sure that the begining of the remainder is the begining of a token and that the merged part is the end of the token before. Since the lexer knows the remainder is the begining of a token it strips all transitions but the one that has incomming state as starting state. Since the start token is the end of a Token it strips all but the transitions ending in an accepting state.

Example 3.1.3 (Combining a token a part of the second token)**.** 'ie' ends in the accepting state for ident (1) and '_' starts in the starting state.

$$\begin{array}{cc} \multicolumn{2}{c}{'e\_'} \\ in & out \\ 10 & 8 \end{array} = \begin{array}{cc} \multicolumn{2}{c}{'e'} \\ in & out \\ 0 & 11 \\ 1 & 1 \\ 6 & 1 \\ 10 & 9 \end{array} \text{ `}combineToken\text{` } \begin{array}{cc} \multicolumn{2}{c}{'\_'} \\ in & out \\ 0 & 2 \\ 2 & 2 \\ 9 & 8 \end{array}$$

$$\begin{array}{cc} \multicolumn{2}{c}{'i'} \\ in & out \\ 0 & 1 \\ 1 & 1 \\ 8 & 7 \end{array} \text{ `}combineToken\text{` } \begin{array}{cc} \multicolumn{2}{c}{'e\_'} \\ in & out \\ 10 & 8 \end{array} = \begin{array}{cc} \multicolumn{2}{c}{'ie'} \\ in & out \\ 0 & 1 \\ 1 & 1 \end{array} ++ \begin{array}{cc} \multicolumn{2}{c}{'\_'} \\ in & out \\ 0 & 2 \end{array}$$

However the remainder may not have the start state as a possible incomming state. In this case the lexer tries to find the largest possible token (that has the starting state as incomming state) and tries to construct a token of the rest of the remainder, repeating this procedure until the entire remainder has been split into acceptable tokens. All the tokens accept the one that is on the very end of the sequence will have all but their accepting states stripped. This case does occur quite frequently since most languages has comments and strings which can contain anything.

Example 3.1.4 (Handling the remainder)**.** '_' starts in the starting states and ends in an accepting state and 'e' starts in the starting state, it doesn't have to end in an accepting state.

$$\begin{array}{cc} \multicolumn{2}{c}{'\_i'} \\ in & out \\ 9 & 7 \end{array} = \begin{array}{cc} \multicolumn{2}{c}{'\_'} \\ in & out \\ 0 & 2 \\ 2 & 2 \\ 9 & 8 \end{array} \text{`}combineToken\text{`} \begin{array}{cc} \multicolumn{2}{c}{'i'} \\ in & out \\ 0 & 1 \\ 1 & 1 \\ 8 & 7 \end{array}$$

$$checkRemainder \left(\begin{array}{cc} \multicolumn{2}{c}{'\_i'} \\ in & out \\ 9 & 7 \end{array} \right) = \begin{array}{cc} \multicolumn{2}{c}{'\_'} \\ in & out \\ 0 & 2 \end{array} ++ \begin{array}{cc} \multicolumn{2}{c}{'i'} \\ in & out \\ 0 & 1 \end{array}$$

When all partial tokens has been combined in this way the resulting sequence of tokens represents the the code the lexer was run on.

3.2 Lexical Errors

Since the lexer has to be able to handle any kind of possible not c̈ompleteẗokens, error handling can be done in different ways. One approach is to simply return as many tokens as possible from the code and where there might be lexical errors the lexer returns the error in as small parts as possible.

Example 3.2.1 (A lexer that only lexes letters)**.** When the lexer encounters the string ẅhat @ dayït would return:

String	Type
What	*Word*
'_'	*Space*
'@'	*No_Token*
'_'	*Space*
day	*Word*

3.3 FingerTree

\# We should here also talk about some datastructers that is needed for a incremental lexer to work.

4

Parallelism

Our solution should be able to run on several cores. This chapter shoul be about why how and so on.

5

Used Structures

#Unsertain of the name of this chapter. But here we should talk about bnfc and alex. What we use from the differnet programs. How this is usefull is it becouse of lazynes or are the existing solutions good??

6

Testing

#How is the testing of this project preform!?

7

Result

#What have been the result?

8

Performance Analysis

#How fast is the lexer. How have we come to this conclusion?

9

Discussion

#Discuss discuss!!

10

Conclusion and Futher Work

#what will our Minions do????

Bibliography

[1] R. Sebesta, Concepts of Programming Languages [With Access Code], Always learning, Pearson Education, Limited, 2012.
URL `http://books.google.se/books?id=h9d0ygAACAAJ`

[2] A. V. Aho, Handbook of theoretical computer science (vol. a), MIT Press, Cambridge, MA, USA, 1990, Ch. Algorithms for finding patterns in strings, pp. 255–300.
URL `http://dl.acm.org/citation.cfm?id=114872.114877`

[3] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools (2nd Edition), Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[4] A. Ranta, M. Forsberg, Implementing Programming Languages, College Publications, London, 2012, pp. 38–47.

[5] M. Sipser, Introduction To The Theory Of Computation, Advanced Topics Series, Thomson Course Technology, 2006.
URL `http://books.google.se/books?id=SV2DQgAACAAJ`