CHALMERS

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 analyzer, is a program which jobb is to convert a string of a formal language into a sequence of tokens. #Hitta REF. This can be done by using regular expressions, regular sets and finite automata. Which are centerel consepts in formal language theory. [1]

2.1 Lexing vs Parsing

There are several resons why a compiler should be seperated in to lexical analyser and a parser (syntax analyser) phases. Simplicity of design is the most important reson. When deviding the task in to these to sub task, it allows the system to simplify one of these subtasks. For example, a parser that has to deal with whitespaces and comments as syntactical units would be more complex then one that can assume whitespaces and comments have alreday been removed by an lexer. Also when the two tasks are devided into subtasks it can lead to cleaner overall disgn when designing a new language [2] Also overall efficiency of the compiler can be improved. When seperating the lexical analyser it allows for applience of specielised techniques that serv only the lexical task. [2] Last complier portability can be enhanced. That is Input-device-specific peculiarities can be retricted to the lexical analysis. [2]

2.2 Languages

2.2.1 Formal Languages

2.2.2 Regular Languages

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 regulat expressions. [3]

2.3 Regular Expressions

Regular expressions are used to describe a patterns in a string. In a regular language, a programming language, this is usefull. Since these languages are build on very strict rules on how strings must follow a pattern. #Ref på detta!! #Some text here about how BNFC can be used to get a lexical file filled with regualr expressions.

Definition 2.3.1 (Regular Expressions [1]).

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 parantheses 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. [1]

2.4 Tokens, Patterns and Lexemes

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

Token is a pair consosting of a token name and an optional attribute value. The token name is a abstract symbol corresponding to a lexical unit [2]. 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. [2] 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 analysied which matches the pattern for a token and is identified by the lexical analyser as an instance of a token. [2]

2.4.1 BNFC

2.5 Finite State Machine

2.5.1 NFA

2.5.2 DFA

2.6 Known Solutions

2.6.1 Alex

3

Incremental Lexer

Here we should talk about how a incremental lexer work. What specifc techniques it uses, how this can be usefull in the real world and so on!

3.1 FingerTree

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

3.2 Devide and Conquer

#how dose ot work, how can we use it, what will it do for us?

3.3 Branch and Bound

#Maybe something about this???

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] 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`

[2] 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.

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