CS 444 - Lexical and syntactic analysis phase

dburgoyn
udimitri
x2fang

Section 001
2015-01-13

This document describes the design phase of our compiler, discusses challenges that we encountered and how we tried to overcome them, and explains the testing that we did before submitting to Marmoset.

1 Scanning

We wrote a lexical scanner based on Brzozowski's ideas that were presented in class. Our `Regex` class contains functionality to compute Brzozowski derivatives using structural recursion, and to test whether a regular expression is equivalent to the empty set. We found a paper online (”Regular-expression derivatives reexamined” by Scott Owens et al.) which describes a method for converting a regular expression into an equivalent DFA using Brzozowski's ideas. The paper also explored the idea of character equivalence classes that was hinted at in lecture, as well as a list of regular expression equivalence rules. We adapted these ideas for use in our scanner.

We constructed a regular expression for each Joos 1W lexical token. We encountered difficulty constructing some of these, such as the regular expressions for C-style and Javadoc comments, because these were defined using mutually-recursive productions in the lexical grammar and while our construction supported referencing previously-defined regexes in a given regex, it did not support mutual references between regexes.

Our scanner works by iteratively computing the Brzozowski derivative of each of these lexical regexes with respect to the input program, character by character, until all derivatives become equivalent to the empty set. At this point, we determine which lexical regex matched the longest prefix of the input, and consume that longest prefix as a token of the corresponding type. Since we do not use a line-by-line scanning approach, we must perform careful arithmetic to record accurate positional information in each lexical token.

We tested the scanner against a few valid inputs before beginning work on the parser.

2 Parsing

To facilitate debugging, we built a human-readable description of a context-free approximation to Joos 1W's syntactic grammar by reading the relevant sections of the Java Language Specification 2.0. We then wrote a tool to convert this human-readable grammar into a CS 241-style .cfg file, which we fed into the provided parse table generator to build an LR(1) parse table.

Our parser reads the LR(1) parse table and stores its terminals, non-terminals, start symbol and productions in internal data structures. It then uses the standard LR(1) algorithm to produce a parse tree from the stream of lexical tokens produced by the scanner. If no transition exists given the current state and input token at any point in the parse, the parser reports the unexpected token along with its positional information for debugging purposes.

We tested the parser against a few valid inputs (produced by our scanner) before beginning work on the weeder.

3 Weeding

TODO: Describe the weeding process.

Once the weeder was largely complete, we began to test our compiler as a whole.

4 Testing

TODO: Describe the testing process.

TODO: Run Marmoset tests locally, and describe the process.