CS444 Assignment 1

Kevin Suwala, Kyle Verhoog

Feb 10, 2019

Contents

Scanning

Our design goal with the scanner was to make it as generic and configurable as possible. We opted to develop something similar to `lex`. So tokens can be specified with regular expressions provided in a configuration file. The scanner reads in this configuration and generates a DFA which can be used to scan source files into the specified tokens.

Implementation

The scanner works by generating or loading a DFA which specifies the tokens to scan. The scanner then iterates over a given string with maximal-munch and scans the longest matching strings using the DFA and produces a list of tokens to be used by the parser.

The tokens are specified by a configuration file of the form

```
<TOKEN1> "<REGEX1>"
<TOKEN2> "<REGEX2>"
...
<TOKENN> "<REGEXN>"
```

DFA Construction

In order to construct the DFA used for scanning, we decided to write our own regular expression library. The motivation for doing so was to gain familiarity with developing regular expression libraries as well as to make generating the DFA for the scanner generic and configurable.

Implementation

The DFA generation works by reading in the regular expressions, generating an NFA for each and then combining the NFAs. The resulting NFA is then converted to a DFA.

Regular Expression Library

Defined in `src/main/scala/regex`, the regex library was written from scratch. Inspiration was taken from the well-written articles from Russ Cox listed here: https://swtch.com/~rsc/regexp/. Our regex library implements the following features:

- concatenation: `ab`
- union/alternation: `a|b`
- repetition zero-or-more: `a*`; one-or-more `a+`
- zero-or-one: `a?`
- ranges: `[a-z]`

Limitations

The regular expression language takes advantage of the fact that Joos source files can only be expressed with the ASCII character set. We abuse unicode characters heavily in our implementation to provide support for ASCII characters like new-line, tab and others.

Problems

The primary problem we had with defining the Joos tokens with our regular expression implementation was specifying multi-line tokens. This could easily be done in most regular expression libraries that support *lazy* matching with the pattern `/\*.*?\*/`. We worked around this with a pretty dirty hack that involved replacing the comment characters with a special unicode character and then matching the special characters.

Performance

The NFA combining is not very efficient and no optimizations are made to remove unreachable states. It takes about a minute to generate the DFA for the Joos language. An easy optimization would be to cut out unreachable states for each regular expression NFA as they are added.

Persisting the DFA

Because DFA generation takes a significant amount of time, we persist the generated DFA and since the DFA does not change we simply read it in on program start-up.

Helpful Error Messages

Care was added so that if a scanning error were to occur, an exception is raised containing useful debugging information like the line, column number and the string scanned so far.

Performance

Scanning performance

Parsing

Algorithm

The

AST

Construction