CS444 Assignment 1

Kevin Suwala, Kyle Verhoog

Feb 11, 2019

Contents

Scanning

File: `src/main/scala/compiler/scanner/Scanner.scala`

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 a regular expression library 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+`
- any character: `.`
- 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 is pretty good. With the maximal munch approach and DFA we acheive linear scanning performance.

Parsing

Grammar

File `src/main/resources/grammar.json`. File `src/main/resources/grammar.cfg`.

The grammar was based upon the grammar specified in the *Java Language Specification* version 1 text. We simply removed Java features not required for Joos1W from our grammar. A JSON representation of the grammar was constructed along with a Ruby helper script to convert the representation to

a format needed by the `Jlalr.java` utilities. This grammar turned out to be LR(1).

Algorithm

File `src/main/scala/compiler/parser/Parser.scala`.

The parsing algorithm we use is LR(1). We used the provided helpers in `Jlalr.java` to generate the parse table and then wrote the standard LR(1) parsing algorithm.

Weeding

Some basic weeding is done in the parser. This includes weeding on identifiers for Java keywords that are not apart of the Joos1W language. Examples of these are `try`, `catch`, `do` and `float`.

Abstract Syntax Tree

File: `src/main/scala/compiler/ast/AST.scala`

After parsing is completed, an Abstract Syntax Tree is generated from the parse tree.

Construction

The AST is constructed in one top-to-bottom traversal of the parse tree. The nodes of the Abstract Syntax Tree inherit from the following base class:

```
class AST(var parent: Option[AST] = None,
          var leftChild: Option[AST] = None,
          var rightSibling: Option[AST] = None) { ... }
```

Nodes have access to their parent, their left-most child and the sibling to their right. This was influenced by the design of Abstract Syntax Trees as described in *Crafting a Compiler*.

Representation

The nodes of our Abstract Syntax Tree are classes that represent elements from the grammar. We define classes with useful helper methods to represent language constructs. For example the AST node for `MethodDeclaration` contains helper functions like `modifiers()` for the modifiers of the method, `identifier()` for the name of the method and `returnType()` for the method return type. Instead of pulling this data up into the node when generating the AST, the helper functions are AST lookup methods that provide syntactic sugar for accessing child nodes. For example, with the aforementioned `identifier()` method looks like this:

```
def identifier: String = {
  this.getDescendant(2, Some(1)) match {
    case Some(n: MethodDeclarator) => n.identifier
    case e =>
      throw MalformedASTException(
        s"Method does not have MethodDeclarator child (got $e.")
  }
}
```

Weeding

Some weeding is performed during AST construction as the AST nodes have some insight to their own structure. For instance, the `MethodHeader` class will validate the modifiers that it has ensuring rules like "an abstract method cannot be `static` or `final`" are followed.

Weeding

File `src/main/scala/compiler/ast/Weeder.scala`.

After the AST has been generated, a pass is made through the AST from top-to-bottom performing any outstanding weeding. Here we check for things like

- filename matches class or interface name
- every class must contain at least one explicit constructor

and more.

Bringing it all Together

All the pieces are combined in `src/main/scala/Compiler.scala` and `src/main/scala/Joos1WCompiler.scala`. Here is where the entry point `main` is defined. Essentially all it does is call:

```
Weeder.weed(AST.convertParseTree(Joos1WParser.parse(Joos1WScanner.scan(argv(1)))))
```