CS 444 - Compiler Construction

Assignment One Design Doc

Preamble

Tech Stack

Our compiler is built and tested in scala 2.10, and runtime jre 6. We have the following dependencies:

- **scalatest:** a testing framework for scala
- **guava:** a java-based collections and io library

Our build system is Simple Build Tool (sbt). Sbt will automatically pull all dependencies and allow you to run the project.

Project Layout

We have organized top-level directory structure as to handle a different phase of the compiler. We have the following top-level directories (sub-projects):

project: Contains the sbt build file and all building logic required to compile, test and run joosc

preprocessor: Generates assets which are required by the various stages of the compiler.

1. Generates the DFA to be used by the Scanner (joos-1w-dfa.dfa)
2. Translates the context-free-grammar specified in joos-1w-grammar.txt into the appropriate .cfg format (joos-1w-grammar.cfg)
3. Translates joos-1w-grammar.cfg into an lr1 file to be used by the parser (joos-1w-grammar.lr1)

scanner: Contains code responsible for handling the tokenizing stage.

1. Reads in joos-1w-dfa.dfa generated by the preprocessor
2. Uses the DFA to tokenize inputs of the source program.
3. Hands off the generated sequence of tokens to the parser

parser: Contains code responsible for handling the parsing and weeding stage. It will:

1. Read in joos-1w-grammar.lr1 generated by the preprocessor
2. Use the grammar to build the parse tree of the input
3. Weeds the parse tree of invalid programs

Group 12
Asmar Khalid, ShengMin Zhang, Freddie Feng

common: Contains code which is re-used across multiple sub-projects. This includes:

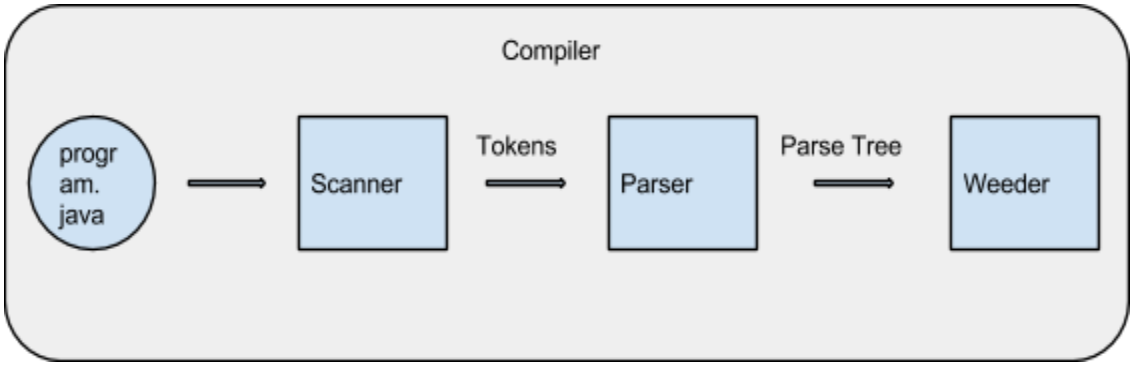
1. Regular expression to NFA to DFA library
2. Token classes
3. Parse tree classes
4. Miscellaneous utility classes

compiler: Contains driver code to bootstrap the compilation of an input file

Compiler Design

Our compiler follows three of the four recommended stages. We opted to defer AST building as it was not necessary for us to build for the purposes of assignment one. OK.

Architecture



Driver

The driver is implemented in *Compiler.scala*. It delegates the compilation process to each of the following stages, in order.

Scanning

This stage is responsible for tokenizing an input file into a sequence of tokens. It is implemented over a DFA, where accepting states corresponding to different token kinds.

Within *TokenKind.scala* is an enumeration of all possible token kinds. Each has an associated regular expression defined in *TokenKindRegexp.scala*. These regular expressions are backed by an NFA with epsilon transitions. We have created such a library in *RegularExpression.scala*. These regular expressions support concatenation operations (as well as others), and hence to generate the NFA of the Joos language, we concatenate all TokenKind regular expressions.
I think you mean "alternation" here rather than "concatentation".

Finally, within *Dfa.scala* we support the conversion from an NFA to a DFA. This process can be lengthy, and so we try to avoid re-computing this on every compilation request.
I'm glad that you implemented it this way.
The preprocessor executes the above steps if the DFA has changed, using a versioning scheme. It will then serialize the Dfa for the driver to load on a compilation request. This logic is present in the preprocessor task *DfaGeneratorTask.scala*.

The scanner itself is implemented in *Scanner.scala*. It implements a maximal-munch over the given DFA, and outputs the ordered set of tokens.

Parsing

This stage is responsible for parsing the sequence of tokens output by the scanning phase into a parse tree. It is implemented using the lr(1) parsing algorithm with tree building on the course webpage.

The preprocessor will read in the grammar specified in joos-1w-grammar.txt. This is a human-readable representation of the Joos 1W grammar. This grammar was based off of the Java grammar, and pruned accordingly to fit the specifications of Joos 1W. The preprocessor will output the corresponding .cfg representation of using the *ContextFreeGrammar.scala* class. This logic is present in *CfgGeneratorTask.scala*

The preprocessor will then use the *ActionTableGenerator.java* class (borrowed and mofidied from the course website) to produce the .lr1 format required to parse input. This logic is seen in *ActionTableGeneratorTask.scala*.

These tasks are only executed when the correct version of joos-1w-grammar.cfg and joos-1w-grammar.lr1 are not present in the generated_resources directory.

On a compilation request, *LrOneReader.scala* will read in joos-1w-grammar.lr1 and produce an instance of *LrOneActionTable.scala*. This action table contains the shift and reduce rules of the grammar.

Finally, *ParseTreeBuilder.scala* will use the *LrOneActionTable* along with the tree-building lr(1) parsing algorithm to produce an instance of *ParseTree.*

The *ParseTree* is simply a pointer to a root *ParseTreeNode*, which may have multiple children. Each node represents a production rule, or is a terminal token.

A significant portion of syntactic rules of the Joos 1W language are enforced inside the parser while the other ones are left to the next step, weeding. Although it may be possible to omit

weeding and have the grammar handle all syntactic rules, the added complexity to the grammar may be greater than a post-processing weeder.

In general, we have tried to have the grammar enforce most syntactic checks, while deferring more complex checks to the weeder.

Weeding

After the *ParseTree* is successfully built, it is then passed to the Weeder to further process the parse tree.

In total there are 8 weeders applied to each node in the *ParseTree*, each representing a category of rules that are not handled in the parsing stage. Because of the compromises made in designing the LALR(1) grammar, we inevitably require further processing to ensure that the compiler will reject any invalid program. An example of this would be duplicated modifiers.

It would be good to list the specific compromises that you had to make to design the LALR(1) grammar in the document.

From a high level a weeder is a node visitor. Every weeder is applied to each node in a top-down breadth-first sequence. There is no dependency or information sharing between each weeder, but there is a special caveat discussed in the next section.

If a particular weeder determines a syntactic rule is violated, it will flag the tree as invalid. If no weeder has flagged the parse tree as invalid, we are finally confident that the input is valid Joos 1W.

Caveats & Challenges

Weeding without an Abstract Syntax Tree

Ideally, converting the *ParseTree* into an AST would immensely assist the weeding step, as weeders generally need to traverse several levels down the subtrees in the given *ParseTreeNode*.

For the majority of the restrictions, an AST would make the desired information immediately available on the next level. Unfortunately, due to time constraints and the additional testing efforts, we decided to defer AST building until after this assignment. OK.

Weeding out-of-range integers

Since we do not tokenize integers as negative, the *DecimalIntegerRangeWeeder* keeps state of visited *IntegerLiteral* that are preceded by a minus sign, since positive and negative integers have different bounds. If we do not maintain this state, we would incorrectly mark the parse tree as invalid when examining a *IntegerLiteral* that fits within the negated bounds.

Testing

We used the scalatest framework to test our project. Our tests are divided into two categories; integration tests and unit tests. We ensured that our code was well tested before submitting to marmoset using these two tests.

Unit Tests

We write isolated unit tests for classes which we deem complex enough. These tests are only concerned with the correctness of the specific class they are testing. We have attempted to test a wide variety of cases so that we may refactor or rework existing code in the future, and not cause any regressions in correctness.

For example, our regular expression to DFA library is exhaustively tested to ensure that it is working as desired. This reduces the scope when searching for bugs, as we have confidence that a certain part of the compilation pipeline is working as intended.

Unit tests for *<ClassName>.scala* can be found in *<ClassName>Spec.scala*.

Integration Tests

Integration tests run through the entire pipeline in the compiler. The idea is to make sure the end-to-end interactions are all working and an execution of the compiler performs as expected.

The marmoset tests were imported from the cs444 directory into our project.

MarmosetSpec.scala will run each sample marmoset file against our driver and check to see if it will output the expected result. Using this strategy, we were able to check what mark we would get on marmoset without actually submitting.

Good testing. Did you add any of your own Joos test programs in addition to the ones from Marmoset?

The document is clearly written and well organized. It is mostly sufficiently detailed, though more specific discussion of the LALR(1) grammar would have been appropriate. You made reasonable design decisions in this part of the compiler.

Presentation: 20/20
Content: 10/10
Total: 30/30