CS444 Assignment 1 Design Document

An, Qin Kuan, Wei Heng McIntosh, Daniel David

February 10, 2020

Overview

There are three major components in this phase of the compiler: scanner, parser and weeder. The scanner extracts list of lexical tokens from raw character input stream. The parser performs bottom up LALR(1) parsing using the DFA generated from context free grammar description, creating a parse tree in the process. The weeder traverses the parse tree to check for various violations among the additional syntactical constraints placed on the Joos language.

Scanning

Scanning is the process of breaking a sequence of input characters into a list of lexical tokens. Since it serves as the starting point of the compiler, and it is where raw character streams are first interpreted, the robustness of this phase is extremely important.

The scanner is ultimately parameterised by a DFA which describes the the lexical grammar. This DFA describes everything that is needed to successfully tokenize any Joos program. The DFA is obtained in a few steps:

1. The lexical rules are written in a specific format and stored in the file `joos.txt`
2. At the start of the program, this file is read and an NFA is constructed directly
3. The NFA is converted to the desired DFA

After the DFA is generated, the scanner executes the maximal munch algorithm on the DFA to tokenize the input character stream.

NFA Specification

Here is a brief excerpt from `joos.txt` to illustrate how the NFA is specified:

```
...
IntegerLiteral:
  emit 10
  DecimalIntegerLiteral

DecimalIntegerLiteral:
  DecimalNumeral
```

```
DecimalNumeral:
  any 0
  any 123456789
  NonZeroDigit Digits

Digits:
  Digit
  Digits Digit

Digit:
  any 0123456789

NonZeroDigit:
  any 123456789
...
```

The file formats used is similar to the one found in Chapter 3 of the Java Language Specification (JLS). This means that a significant portion of `joos.txt` could be directly reused from the JLS verbatim, reducing the possibility of error from having to manually implement them otherwise. However, the JLS does not use the format exclusively to describe all of the lexical grammar and we had to fill in the missing parts. Nonetheless, the amount of work saved was considerable.

We also added some extensions to the format to simplify parts of `joos.txt`. For example, "any" allows multiple characters to make up permitted transitions from one NFA state to the other; (Taking "Digit", for instance, the use of "any" means that the set "0123456789" contains all the digits that will make the starting state of "Digit" to transit to its accepting state.) "emit *priority*" tells the scanner that a particular token should be emitted with *priority* when the corresponding accepting state is reached. Conflicts are properly resolved by the scanner using priority values when a generated DFA state contains multiple accepting NFA states.

Next, the scanner turns this NFA into a DFA using the ϵ-closure algorithm described in the textbook.

Problems and Resolutions

Sharing NFA States

We encountered an interesting problem in how the NFA is encoded in the text file. Consider the following example describing a certain token where `x`, `y` and `z` are atomic characters. `A` and `B` are abstract constructs to organize the text file to minimize duplication.

```
Goal:
  A

A:
  B x B

B:
  y z
```

Taking this as an example. Originally, the algorithm of text file to NFA translation works by first creating small NFA's that treat the internals of A and B as black boxes and link them using ϵ-transitions. In this case, the starting state of A has an ϵ-transition to B, and the accepting state of B is linked with an intermediate state that accepts character 'x'. That state has an ϵ-transition to B again, and finally the accepting state of B is linked to accepting state of A using ϵ-transition.

Although it looks obvious at this point that the above construction is wrong, as it incorrectly accepts "y z" because after seeing the first occurence of B, there is an ϵ-transition directly to the accepting state of A, bypassing required sequences "x y z" that follows, it tooks us a while before realizing it is not possible to reuse existing smaller NFA as black boxes as part of a larger NFA.

One way to solve this problem is to actually duplicate each abstract construct each time it is used. This requires cloning a graph that can potentially have cycles somewhere, and we abandoned the idea relatively quickly because the lexical grammar specification is filled with self referencing rules and the problem is too complex to solve 100% correctly given time and effort we have. In the end, we manually duplicated some of the abstract construct that are used at many places (for example, LineTerminator is used at many places, and EscapeSequence is used both for character and string literals), and avoided having to code up some fancy algorithm that can potentially be hard to debug and prove correct.

Optimising the NFA to DFA translation

Our initial NFA to DFA implementation was quite naive: the ϵ-closure of a set of NFA states is recalculated every time it is needed. We quickly realized that it is possible to precompute the ϵ-closure of each NFA state only once at the beginning and cache them. Then, the ϵ-closure of a set of NFA states can be computed by taking the union of the ϵ-closure of the members. Since there are only ~500 states in our NFA, we were able to store each state set using a lightweight 64-byte bitfield. The union operation can be performed by simply taking the bitwise-or of 8 pairs of 64-bit integers. We are satisfied with memory locality and cache friendliness of this approach.

The above optimization was a significant improvement over the naive algorithm and speeded up the NFA to DFA construction greatly. It is a worthwhile investment in the long term since this piece of code is always executed whenever the compiler is run, and every bit of efficiency gained in the write-run-debug cycle helps us get work done faster.

Parser

Context Free Grammar Creation

For the parser we used the LALR(1) grammar from the first edition of the JLS as a starting point. Specifically, we used the HTML version of the first edition specification since it was easy to copy the text from, and had convenient `<i>` and `</i>` tags to differentiate Non-terminal symbols from terminal symbols. With a little bit of automated cleanup using regular expression find-and-replace, we quickly removed the tags we didn't want to keep and tidied up the position of those we did so they didn't include whitespace. A few more regular expression substitutions and we were able to expand rules which contained optional elements into two separate rules - one with the element, and one without. This left us with a file which contained rules in the following format:

```
NonTerminal1:
```

```
NonTerminal2
</i>Terminal1<i> NonTerminal3 </i>Terminal2<i>
```

A repeated find-and-replace-all on three copies of the file allowed us to extract a list of terminals (with duplicates) and the rules in the form needed for the .cfg file format. To extract the non-terminals, we used a find-all to select the `NonTerminal:\n` pattern, inverted the selection, and deleted. All that was left then was to augment the grammar, paste those three together, along with the start node and the number of lines in each, and we had our .cfg file.

However, as we worked to design our AST structure, we modified the grammar several times. Since the .cfg file format is rather difficult to read as a human, we kept a copy of the file before extracting the necessary parts in `JavaSpecLALR.txt`. Then, as we made changes, we were able to quickly recreate the .cfg file from the modified grammar any time we needed.

Bottom-up AST generation

One of the other, unexpected benefits of keeping the grammar laid out in a readable fashion was that it was easy to notice patterns in the grammar. Furthermore, with small tweaks to the grammar, we noticed that it would be possible to build the AST on the fly during parsing. This is accomplished by associating each parse tree node/non-terminal symbol with either an AST class or a Pseudo-AST (pAST) class. The AST can then be built bottom up during parsing by building an AST/pAST node together with each parse tree node. For a given parse tree node, the associated AST/pAST node is built using the AST/pAST nodes which are associated with the children of the parse tree node. After construction, we don't want any pAST nodes to be left in our tree. To accomplish this, anytime a pAST node is seen during construction, instead of keeping a pointer to it, its contents are read and used instead.

For example, derivation for the non-terminal `Modifiers` contains a `Modifier` and potentially another `Modifiers`, but that's not a convenient representation for our AST. As such, `Modifiers` maps to a pAST instead of an AST. To keep our lives simple, the pAST for `Modifiers` will contain a `vector<Modifier>`. Later, when we are building something which has this `Modifiers` in its derivation, such as another `Modifiers`, we move the vector into the new parent, and append the other `Modifier` in the parent's derivation. When we are building an AST node with the `Modifiers` in its derivation such as a `FieldDeclaration`, all we need to do is move the vector into the AST node for the `FieldDeclaration`. Note that we don't necessarily have to keep the format of the contents the same between the pAST node and the parent AST node. If for example, we decided we wanted a bit-field in `FieldDeclaration`s, in the constructor for a `FieldDeclaration` AST node, we could convert the vector into a bitfield. This allows a great amount of flexibility while keeping area-of-concern for an AST node contained to the constructor for that AST node.

Ultimately, we didn't end up using bottom-up AST generation, but we did stick with a grammar which would allow it.

Parse Tree Construction

Type safety is an especially desirable property for the core data structures used in this project such as the parse tree and AST. The programs represented by these data structures possess certain invariants that must be respected by any code that uses them and each usage is a new opportunity to break these invariants. This is especially true for the AST which is the sole subject of interest to the compiler in almost every stage of the compilation process.

Thus, it is desirable to design types that encode these invariants to minimize the number of invalid Joos programs that are representable at runtime. We will discuss how this is achieved for the parse tree. The natural way is to define a each new type for each non-terminal in the grammar corresponding to a node type in the parse tree. Each type is a struct whose members are terminals or pointers to other non-terminal nodes. Having unique node types allows the compiler to reject ill-formed parse trees that should never be generated with the grammar, saving us from an entire class of implementation errors.

In the interest of implementation simplicity, it is sometimes useful to be able to treat the nodes as untyped, such as during the construction of the parse tree. To do so, each struct also inherits from a generic `Tree` struct which stores the children in a list.

Writing all the struct definitions and associated code by hand would have been a tedious and error-prone process. Since it's a mechanical process, our solution is to use automatic code generation. Any code that is completely determined by the grammar definition, is emitted by the generator. This includes the code for manipulating the tree stack and populating the corresponding fields of a node for all rules. Having a code generator spared us from the tedium and possiblility of bugs in writing the generated code manually and also allowed us to iterate on the design of the grammar easily without the burden of having to rewrite the affected code.

Of course, if there is a bug in the generator, the generated code will also be incorrect. Luckily, such errors are easy to detect using `dynamic_cast` and C++ runtime type information. The generator also emits assertions liberally throughout the generated code to catch errors early.

Weeder

The weeder verifies that the parse tree holds the invariant of representing a syntactically valid Joos program. This invariant is expected to simplify the next phase of compilation where the parse tree is converted into an AST.

Since the weeder is meant to catch errors that would have been too complex to express in the grammar, the implementations of the weeder checks are rather ad-hoc. Most weeder checks boil down to the checking for the presence or absence of certain types of nodes in particular subtrees.

Testing

Since we are given the files used by Marmoset for testing, we decided to test locally before submitting to Marmoset. Testing is automated and implemented alogside the application logic of our code, which makes it more robust than using simple shell scripts. To run the tests, the user sets an environment variable. This is necessary since command line arguments are reserved for input files. When the program starts and detects the environment variable, it scans all test files in the relevant directory and tries to compile them. Testing is further integrated in our development workflow via a Gitlab pipeline that builds the application for testing on each commit and before merging to master. It provides direct information on whether a certain commit is good to go or needs rework.

While unit testing is a good idea in general, we do not think it is suitable for this project due to limited time and the amount of code that we expect to write. So we rely mainly on integration tests.