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 DFA generated from context free grammar description, and creates a parse tree along the way. The weeder traverses the parse tree, and checks for various violations of additional constraints placed on the syntactical rule of Joos language.

Scanning

Scanning is the process of breaking 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 creates a DFA from this giant NFA using the ϵ-closure algorithm as described in the textbook.

Problems and Resolutions

In this section, we will discuss some interesting problems encountered during implementation.

Sharing NFA States

First is a problem during 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 then 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

(1)

Bottom-up LALR(1) parsing

(2)

Parse Tree

Type safety is an especially desireable property for the core data structures used in this project such as the parse tree and AST. The programs represented by these data structures possesses 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.

Parse Tree Construction

One of the most important job of a parser is to create a parse tree. A well-defined and easy-to-use parse tree not only makes weeding simple, but also makes abstract syntax tree creation less error-prone and straightforward. It might be tempting to do it in a simple way: having a generic Tree structure storing a list of children, and having a tree stack alongside with state stack and symbol stack which are manipulated in the same fasion. But we imagined at later stage when doing weeding and abstract syntax tree creation, we want to know exactly which rule is a certain tree node created from (is it class declaration, or binary expression), and depending on its type we may want to access its 2nd or 4th children because that is where the relevant subtree is stored.

One solution is to store this information somewhere on the generic tree, and have some helper functions to retrieve the correct subtrees depending on the rule. However, note how everything becomes no longer statically typed (our implementation language is C++) and we imagined we will probably be retrieving subtrees using string parameters, and there is just lots of frictions in doing that.

We want to fully utilize the power of a statically typed language where the compiler catches typos for types. We realized this is a perfect candidate for automatic code generation: if we generate C++ struct definition for all possible tree types, and store pointer to subtrees as regular fields inside structs, how easy and error-free would that be when writing weeders and abstract syntax tree conversion code. As a result, we wrote code that generates all possible tree definitions from context free grammar description, as well as code that generates code that correctly manipulates tree stack and assigns subtree pointers to corresponding fields in tree structures. Not only does this prevents us from typing strings or children indices incorrectly, it also enables IDEs (integrated development environment) to perform auto-completion and syntax highlighting, which greatly increases productivity. This will not be possible without baking in all tree definitions as regular C++ code that can be checked at compile time.

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.