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 in our compiler understands the lexical grammer written in a specific format. This format describes everything that is needed in order for scanner to be able to correctly obtain list of tokens from input files.

A brief excerpt of the file format invented that is side input to the scanner:

```
...
IntegerLiteral:
  emit 10
  DecimalIntegerLiteral

DecimalIntegerLiteral:
  DecimalNumeral

DecimalNumeral:
  any 0
  any 123456789
  NonZeroDigit Digits

Digits:
  Digit
  Digits Digit
```

```
Digit:
  any 0123456789

NonZeroDigit:
  any 123456789
...
```

This file format is very similar to the format that describes Java lexical grammar rules as seen in Chapter 3: Lexical Structure of "The Java Language Specification" book. This way, most part of this file can be pulled from the specification directly, reducing possible errors when manually writing regular expressions to match tokens, for example.

Some additional keywords in this file are used by the scanner. "any", for example, 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 is made of multiple accpeting NFA states.

At program start up, the scanner reads in this file, and produces a NFA out of it. Since this file is mostly pulled from the specification describing the lexical grammar of Joos language, the generated NFA correctly recognize list of tokens that are permitted in Joos.

Next, the scanner creates a DFA from this giant NFA using the epsilon-closure algorithm as described in the textbook. Finally, using the DFA, the scanner implements the maximal munch algorithm that tokenize input character stream.

Problems and Resolutions

Some interesting problems arised when we are implementing the part of scanner that performs TXT file -> NFA -> DFA translation.

Sharing NFA States

Consider the following example describing a certain token where x, y and z are atomic characters. A and B are abstract constructs to help with organizing the text file to reduce 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 treats internals of A and B as black boxes and link them using epsilon transitions. In

this case, the starting state of A has an epsilon transition to B, and the accepting state of B is linked with an intermediate state that accepts character 'x'. That state has an epsilon transition to B again, and finally the accepting state of B is linked to accepting state of A using epsilon 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 epsilon 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 duplicates 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.

Efficiency of NFA to DFA translation

The original algorithm we implemented for NFA to DFA is relatively naive: epsilon-closure for a set of NFA states are calculated adhoc every single time. We then realized that it is possible to calculate epsilon-closure for every single NFA state only once at the beginning and cache them. Then the problem of finding out epsilon-closure for a set of NFA states can be done via union operation. Rather than using C++ unordered_set or set, we used bitfields to represent a set of NFA states, as there are only approximately 500 of them and can be represented in 64 bytes. The union operations is then merely bitwise-or of 8 64-bit integers. We are satisfied with memory locality and cache friendliness of this approach.

The above optimization greatly speeded up the process of NFA to DFA construction. This is a long-term benefit as this piece of code is executed each time 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 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 generations: if we genereate 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.

Of course, if there is an error in the generated code, the result will be incorrect. Luckily, this aspect of errors are easy to identify with the help of dynamic_cast and C++ runtime type information. asserts are used extensively in the generated code in order to catch errors at early stage.

Weeder

(@Wei Heng)