CS 444
Assignment 1

Mark Rada (marada)
Riel Fuller (r3fuller)
Michael Forsyth (mfforsyt)

This document outlines the design of a scanner, parser, and weeder for the Joos 1W language as specified by the Java Language Specification 2.0 (JLS) and the course website `https://www.student.cs.uwaterloo.ca/~cs444/joos.html`. It also discusses some challenges faced while implementing the scanner and parser, as well as our strategy for testing correctness.

Code is written in Ruby, and our implementation code is located in `lib/` of the submitted code. A copy of the Marmoset tests is located in `test/`, while our own tests are located in `spec/`. Miscellaneous tasks are handled by the Ruby `make` (`rake`) system, which is organzied in `rakelib/`.

Design

The design of this phase of the compiler is broken up into five sections: token data structures, the scanner, the concrete syntax tree (CST) classes, the parser, and the weeder.

Tokens

The interface between the scanner and the parser is an `Array` of `Joos::Token` objects. `Joos::Token` is an abstract base class which encodes the original value of the token, the type of the token, and metadata related to where the token originated in the source file.

Concrete subclasses are created for each keyword, operator, separator, and literal type, as defined in the JLS. A concrete class for identifiers is also created in the `Joos::Token` namespace. Classes may `include` mixins which attach extra metadata to a token, though the name of the concrete class specifies the type of the token.

For instance, a token representing the keyword `while` will be contained by an instance of the `Joos::Token::While` class, and a token representing a litertal `true` will be contained by an instance of the `Joos::Token::True` class. Identifiers are wrapped by `Joos::Token::Identifier`, literal integers by `Joos::Token::Integer`, and so on.

Each token contains metadata that can be used for diagnostic purposes including the path to the source file where the token originated, as well as line and column numbers where the token begins. Additional metadata is attached to some token classes, such as `IllegalToken` modifier which is used to mark the types of tokens that are legal Java tokens, but are not used in Joos.

Tokens marked as illegal can be immediately weeded out and a helpful error message can be printed.

Additional weeding functionality is added to specific token classes as needed. In the case of string and character literals, we check that all escape sequences in the string are legal escape sequences.

Lexer

The lexical analysis portion of our compiler consists of 3 classes: `Scanner`, which simply feeds lines into the `tokenize` function; `DFA`, a general deterministic finite automaton class; and `ScannerDFA`, a subclass of `DFA` which contains the Joos-specific lexing rules.

`DFA` contains the structure of a finite automaton: a transition table and set of accepting states, with an implicit start symbol named `:start`. The transition table maps a state symbol to an array of `(predicate, state)` pairs. The `transition` function looks up a transition from a state on a given input character by classifying the character to map the character to the DFA's alphabet.

Invalid transitions return the state `:error`. Though, not all valid states need be in the table — we have accepting states like `:line_comment` that contain no transitions, as well as error states like `:float` for catching certain illegal tokens. Non-ASCII characters have no transition and will be caught during scanning.

To perform tokenization, `DFA` has a function `tokenize` which runs the maximal munch algorithm against an input string. The algorithm doesn't implement backtracking, since the only potential backtracking case, octal escape sequences, is handled during literal string validation.

`tokenize` returns 2 values: an array of `DFA::Tokens`, which are simple structures that later get mapped to full `Joos::Token` objects, and an `AutomatonState`, which represents the final state of the algorithm. `AutomatonState` is a nested class that holds the internal state of a run through the DFA — the current state symbol and input seen so far. Tokenization is performed line-by-line. The returned `AutomatonState` 'continuation' is passed to the next call to `tokenize` to handle the case of block comments, which may span multiple lines.

The use of a continuation state, however, requires us to implement two additional `ScannerDFA` methods: `raise_if_illegal_line_end!` and `raise_if_illegal_eof!`. These methods check that we haven't ended on any incomplete tokens.

`DFA` also has a few convenience methods for initializing the transition table. The principal method is `state` which provides a Ruby DSL (implemented in the `StateBuilder` nested class) for concisely specifying a state and its transitions.

One simplifying design choice we made is to scan keywords and identifiers using the same rule and only differentiate between them when later constructing `Tokens`.

Concrete Syntax Tree

The CST is defined by the grammar. The grammar is specified in Ruby code using literal array and symbol syntax. By walking through the structure, we can generate classes for each non-terminal as well as convenience methods for looking up terminals and non-terminals that will be children in the syntax tree. Generated classes all inherit from the `Joos::CST` base class, which provides general functionality for the CST.

We do not generate classes for terminal symbols because we keep the existing `Joos::Token` objects in the CST and take advantage of Ruby duck typing to traverse and manipulate the tree.

The CST provides two methods for applying transformations and validation so that weeding can be performed during or after parsing and so that the CST can be transformed into an abstract syntax tree (AST) in place. The first method is the visitor pattern for traversing the tree and performing weeding checks or transformations after parsing has completed. The second method is overriding the default constructor of the base class to perform any weeding checks or transformations that we can during parsing.

We prefer to perform transformations during parsing, but certain operations (those that depend on being able to traverse up the CST), such as checking literal integer values, cannot be done until after parsing because it is impossible to know the signedness of the literal integer until parsing is complete.

Parser

We implemented an LR(1) parser generator which reads from our grammar specification and generates a lookup table which is used as a modified DFA transition function which handles follow sets, and a table of reduction rules.

The parser takes an array of tokens as input and treats the array as a mutable stack while it performs parsing and builds the CST. Our grammar is augmented to include a special `:ENDPROGRAM` which is used to signify a successful parse.

If parsing is successful than a `Joos::CST::CompilationUnit` object will be on the stack and can be returned to the next stage of the compiler. If parsing fails at some point, such as due to an unexpected token, then an error message is printed and parsing is aborted.

Weeder

Weeding operations are performed during both scanning and parsing. Our goal is to weed out issues as early as possible in order to make later stages easier to implement, and so much of the weeding is done during scanning.

All keywords and operators not supported by Joos are caught during scanning and will never be passed to the parser. Literal floating point values and unsupported formats of literal integers are also caught during the scanning phase.

Some grammar requirements of the Joos language are very complicated and would have made the grammar more fragile, take longer to generate, and produce a much larger state machine. To make things easier, we have created skeleton entity classes to mock Joos classes, interfaces, methods, and the like and perform CST transformations to replace some nodes with their appropriate entity. Entity classes are capable of performing weeding checks such as ensuring that `native` methods are also `static`, and any other modifier constraint.

Several other edge cases caused by our grammar are also checked by visiting each node in the CST and checking for node patterns which are not allowed in the Joos specification.

Additionally, literal integers can now be bounds checked to make sure they are within the bounds of a signed 32-bit integer because we know what the final signedness.

Challenges

This phase of the compiler implementation presented many challenges, some of which led to interesting solutions.

Lexer

In the lexer, one challenge was the process of building the transition table. Originally, the `DFA` class had a nested map from states to input symbols to transition states. The `classify` function mapped some characters to character classes (symbols) in order to avoid having many similar transitions. This approach, however, became unwieldy for strings and comments, which needed a transition for essentially every character. Changing the map from inputs to states into an ordered array of `(predicate, state)` pairs greatly simplified the rules for strings and comments by having a few transitions for exceptional cases then a final transition for everything else.

Duplicate String Literals

Literal string duplication is avoided by overriding how `Joos::Token::String` objects are allocated. The `String` object performs translation to binary during initialization in order to check escape sequences and also to have a non-ambiguous format for comparison with other strings. We maintain a global table of strings which is keyed by the binary representation. We then override the allocation method for the class and, before returning the newly allocated object, check if another string has the same binary representation by looking in the global table. If another string already exists then we return the string in the table and throw away the string we are currently allocating. If the string does not exist yet then we add it to the table and return the string object to the caller.

Testing

Testing is done at two scales. The first scale is unit testing, and the second is system testing. Time permitting, code reviews are also performed.

Additionally, we have implemented debugging support into various parts of the compiler for performing operations such as a visual dump of the CST and parser state.

Unit Testing

Unit tests are implemented as we write new classes and methods. When a new file is created, the corresponding test file is also created with a failing test to remind us to actually write tests.

Tests are organized by class and then by nested class and/or method. Our goal is to write tests that cover all code paths and which test class and method specifications given in docstrings in the source code. We use `yard` and `simplecov` in order to test documentation and test coverage respectively.

System Testing

Using a copy of the publicly available tests from Marmoset, we are able to run all the public tests from Marmoset on local machines. To run the tests we parse the test file name to determine if the compiler should successfully parse the soruce code or not. We then run the compiler by forking a new process and waiting for the result and capturing output on standard output and standard error. If a test fails then formatted output is printed.

Due to the large number of tests that Marmoset runs, we run these tests in parallel. First, we determine the number of CPUs available on the system, and then we configure the testing library to create a worker thread for each available core. In the student environment this allows us to run up to 48 tests concurrently, completing the entire test suite in approximately 10 seconds.

We can also run individual system tests manually if needed.