

SCANNERS

Introduction

In this lab you will manually design and implement a scanner for a simplified version of the Lisp programming language.

Lisp

Lisp is a programming language first described in the late 1950s. It is built entirely around **s-expressions**,¹ which can be either an **atom** (a number or a symbol) or a **list** (a sequence of s-expressions contained in parentheses).

For our purposes, numbers in Lisp are integers with an optional minus sign before them. Symbols are sequences of letters, numbers, and hyphens that begin with a letter (we will only use lowercase letters to keep things simple). Here are some examples:

- 0123: valid number
- -3210: valid number
- 1.23: invalid number
- abcd: valid symbol
- ef1-2: valid symbol
- 8hij: invalid symbol
- -d34: invalid symbol
- (a b 2): valid list

Task: Design the lexical grammar for our simplified Lisp: determine how many token types there need to be and construct regular expressions to describe each one.

Designing the automaton

As covered in the lecture, scanners are based on finite automata, so you will need to construct a DFA capable of representing your lexical grammar.

Task: Use Thompson's construction to build an NFA capable of recognising all the regexes in your lexical grammar.

Task: Use subset construction to transform your NFA into a DFA. Optionally, you can then try to minimise your DFA but remember not to combine your final states.

Implementing the scanner

We will implement the scanner in Java. As you go, remember the different kinds of classes available to you:

¹ <https://en.wikipedia.org/wiki/S-expression>

- **Enums** are for listing a fixed set of values of a particular type;
- **Records** are for simple data classes that don't need to be mutated;
- **Sealed classes** allow you to control who can extend your classes;
- **and so on.**

You may not need all of these, but you should reflect on the most suitable way of representing each element of your scanner.

Task: Create a new Java project, and create a class called `LispScanner`. It should conform to the following class diagram:

LispScanner
<ul style="list-style-type: none"> - Map<Char, CharClass> charClasses - Map<State, Map<CharClass, State>> transitionTable
<ul style="list-style-type: none"> + LispScanner() + List<Token> scan(String) - Token nextToken(StringStream)

Googling “java 21 <classname>” should lead you to the relevant documentation for any of the standard Java classes.

The constructor should initialise the character class table and the transition table with all the relevant data. The scan method takes an input string, scans it, and returns the list of scanned tokens. `nextToken` finds the next token from the given stream – `nextToken` uses a `StringStream` so that the input can be modified.

Task: Create the `CharClass` and `Token` classes appropriately.

Scanning loop

The pseudocode in Figure 2.12 in the textbook (p. 64) gives a sketch of how your scanner should function. The loop shown in the pseudocode corresponds to your `nextToken` method, and `scan` is responsible for repeating this loop until all the input is scanned or there is an unrecoverable error.

Task: Implement `nextToken` by following the pseudocode in the textbook. You can either directly encode the token kinds that correspond to different states or add a third table to your class to look them up.

Task: Implement `scan` to repeatedly call `nextToken` until the input is consumed.

Task: Create a main method to test your scanner on various input strings. For example:

- **Input:** (a b 12c)
Expected output: (“(”, LPar), (“a”, Symbol), (“b”, Symbol), (“12”, Number), (“c”, Symbol), (“)”, RPar)

Optional task: Reimplement `nextToken` to use the ‘maximal munch’ algorithm described in Figure 2.13 (p. 66).

Optional task: Java doesn't have goto statements, but can you use separate methods to transform your table-driven scanner into a direct-coded scanner?