FELADATKIÍRÁS

A feladatkiírást a tanszéki adminisztrációban lehet átvenni, és a leadott munkába eredeti, tanszéki pecséttel ellátott és a tanszékvezető által aláírt lapot kell belefűzni (ezen oldal *helyett*, ez az oldal csak útmutatás). Az elektronikusan feltöltött dolgozatban már nem kell beleszerkeszteni ezt a feladatkiírást.

MŰEGYETEM 1782

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

Modular Interpreter Written in JavaScript

Thesis

By
Heilig Benedek

Supervisor
Kundra László János

December 5, 2016

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Heilig Benedek*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, December 5, 2016

Heilig Benedek
hallgató

Kivonat

Jelen dokumentum egy diplomaterv sablon, amely formai keretet ad a BME Villamosmérnöki és Informatikai Karán végző hallgatók által elkészítendő szakdolgozatnak és diplomatervnek. A sablon használata opcionális. Ez a sablon $\LaTeX$ alapú, a *TeXLive* $\TeX$-implementációval és a PDF-$\LaTeX$ fordítóval működőképes.

Abstract

This document is a LaTeX-based skeleton for BSc/MSc theses of students at the Electrical Engineering and Informatics Faculty, Budapest University of Technology and Economics. The usage of this skeleton is optional. It has been tested with the *TeXLive* TeX implementation, and it requires the PDF-LaTeX compiler.

Introduction

The main goal of the thesis is to create a modular interpreter in the native language of the modern Internet, JavaScript. The idea behind an interpreter such as this is that usually for someone new to programming, it is a hassle to download and install the various compilers and IDEs needed to even begin the actual programming phase, and this could be made easier with web technologies.

Nowadays, everyone has a laptop and a phone capable of browsing the web, and all these devices have a browser to do just that. Most dynamic websites use JavaScript as a language to manipulate the page (to make it dynamic), so all these browsers have a JavaScript engine built into them. I intend to use this capability to bring a more seamless first experience to those who want to learn programming and are at the beginning of their journey.

This is why I set out to create an interpreter that is:

- Written in JavaScript, thus available to be used on every kind of device
- Modular, thus making the creators of educational tools easier by abstracting the common things, and providing an easy way to allow execution of code inside the browser in any custom language that has a module created for it

To execute code written in a specific language, either a compiler (that translates directly into machine code), or an interpreter is used (which translate the code to another language higher than machine code). JavaScript is an interpreted language, partly that is why it can be found in so many places and thus the interpreter described in this thesis will be a cross-interpreter - that is, it will translate from an interpreted language to another interpreted language. But before any interpretation can begin, the code has to go through some transformations, mainly lexical analysis and parsing.

Lexical analysis is basically turning the source code into tokens using basic pattern matching rules, which will make the parsing stage easier. A token has a value and a class - the pattern matching pairs the values to their classes. A good and widely used tool for pattern matching is regular expressions, which is available in JavaScript as part of the language, and thus the proposed interpreter shall use. In case the writer of the module needs more complex rules, the lexical analyser should provide a way to use simple states and define transitions between these states.

In the next stage called parsing (or syntactical analysis), the tokens are turned into a parse tree using simple rules. A parse tree represents the hierarchy of the matched rules. One common way to define these rules is BNF (Backus-Naur form), which is a language to describe grammars. The given grammar describes every type of sentence in the language. In the definitions, we should be able to use the token classes from the previous stage (rules should be automatically added to the grammar). The parse tree then can be used to create an AST (Abstract Syntax Tree), which then can be used to interpret the code itself by evaluating the nodes, which basically represent nested parameters, some with fixed values, and some that need some computation to produce a value (and of course all these can have side effects).

To create a language module, the necessary steps are:

1. Create the rules for the lexical analyser
2. Provide the grammar in BNF
3. Parse the code, and iterate through the AST

The last step is the actual interpretation, which can be a simple evaluation of an expression, or a small virtual machine that keeps track of functions, variables, scopes and etc. This is the part that requires most attention when writing a language module.

Of course, something like this has been attempted many times, there are numerous solutions to creating parsers, although most of them are not written using JavaScript, they are more native tools to generate parsers. The list includes GNU Bison [1], Jison [2] (a clone of Bison in JavaScript), YACC [3] and ANTLR [4]. But out of all of the available works, the most exciting is Syntax [5], which has the same goals (parser in JavaScript) and the same things in mind (educational usage) and my thesis idea, and which was published just a week after I've handed in my thesis proposal. The author of Syntax describes his motives in the article and the main differences from my thesis are the following:

- He only set out to create a modular (language agnostic) parser
- He provides multiple parsing algorithms
- He provides multiple target languages, including JavaScript, but that only means that the parser that is generated will be a JavaScript module (other targets include Python, Ruby and PHP)
- He provides a way to set operator precedence

[1] GNU Bison `https://www.gnu.org/software/bison/`

[2] Jison - Your friendly JavaScript parser generator! `http://zaa.ch/jison/`

[3] Yet Another Compiler-Compiler `http://dinosaur.compilertools.net/`

[4] Another Tool For Language Recognition `http://www.antlr.org/`

[5] Syntax - language agnostic parser generator `https://medium.com/@DmitrySoshnikov/syntax-language-agnostic-parser-generator-bd24468d7cfc`

- He does not provide a state based lexical analyser

In every other sense, the two projects are very similar, although I have only learned about Syntax during later stages of development.

Another similar project is my own project - from which the idea came for this thesis -, which is a Logo interpreter that translates to JavaScript, and which was based on absolutely no research about parsers, and thus is buggy but generally works quite well. It had two iterations, the first of which I built 4 years ago as a hobby project. The second iteration is also a hobby project, but it is much more mature, and can handle Logo source pretty well. The choice of language - Logo - is because it is a good step towards understanding of basic ideas of programming (in both functional and imperative paradigms), and also provides the user with something more materialistic (it is easy to create beautiful and complex drawings), which is great for young children to get a reward for programming, which is a great way to motivate them. The most beautiful things they can draw are fractals (up to a certain level of depth), which also teach them a lot about recursion, which can be hard to understand for new pupils (fractals are a good way to visualize recursion).

Other solutions to running code on the web include cloud based solutions, which means that the code that the user inputs is actually compiled and ran on a remote virtual machine. While this is great in the sense that it is exactly as the "real thing", it has a great drawback - it requires an active internet connection to use. My solution can be bundled into a desktop application (or the website can be downloaded) using tools like the electron [6] framework. This allows the students to practice while on the go (and without an active internet connection), which cloud based solutions can not allow.

During the implementation, I will not be focusing on optimal speeds, since this is not a tool built for performance but rather one that is built to be available everywhere. To achieve modern, well readable source code and portability, I will be using the ECMAScript 6 coding standard, which is available in some modern browsers, but not so many older ones. I will develop my solution in a Node.js module, which can then be converted into code that works on older browsers using Babel [7] and Browserify [8], which are Node.js modules to convert to older standards and to bundle modules into one source file for browsers respectively.

Chapter 1 will outline the basic coding conventions I followed during development. In Chapter 2, I will describe the implemented lexical analyser, and it's inner workings. In Chapter 3, I will write about the parser module, and the implemented parsing algorithm. In Chapter 4, I will describe the two example modules I have created - a math expression parser, and a simple Logo subset interpreter.

[6] Electron `http://electron.atom.io/`

[7] Babel `https://github.com/babel/babel`

[8] Browserify `http://browserify.org/`

Chapter 1

Coding conventions

While JavaScript is a modern language, there are some parts of it that evolved quite slowly compared to other languages. The most prominent part is classes, which have been properly introduced into the standard with ECMAScript version 6. There are shortcomings of this implementation, although it improved on the previous version greatly. Before ECMAScript 6, a programmer who wanted to create an object in JavaScript had to use a weird syntax, since the language did not really have a class keyword, and so anonymous objects had to be used instead of classes. This allowed object oriented development, but only to a certain extent. With ECMAScript 6, a proper class keyword was introduced, alongside with keywords like extend and constructor.

1.1 Problems with ECMAScript 6 classes

While they are a great improvement on the previous situation, ECMAScript 6 classes have some shortcomings - a programmer can not really define a private member for the class, since that keyword is not present in the language. Getters and setters can be used, as well as static variables, but simulating private members is not easy.

Example 1.1: *"A typical class"*

```
class ClassName {
  constructor() {
    this.member = ’x’
    this._private = 1
  }

  get private() {
    return this._private
  }

  doSomething() {
    // do something here
  }
}
```

One of the ways of dealing with the absence of private members is to follow a naming convention - name everything that should be private with an underscore prefix. In fact this is the convention I followed during development, and tried to be consistent with this idea - I tried to avoid using "private" members of other class instances.

1.2 Node.js modules

I developed the solution using Node.js, which allowed me to easily check my solution without even starting a browser. A natural convention arose from this choice, which was to separate things into Node.js modules, which can be included into one another. This is a syntax that browsers can not deal with by themselves, but this problem can be eliminated by using the tools described in the end of the introduction.

Chapter 2

Lexical analysis

Lexical analysis the the process in which an input string is turned into tokens. This is done to make the next step - parsing - easier. For example, the parser can deal with words instead of having to try to match the grammar's rules character-by-character. This can greatly reduce the complexity of the grammar, which means it makes it easier to write a grammar for a specific language.

2.1 Using regular expressions

Regular expressions are a common tool used in lexical analysis [1], since during the analysis, we need to determine what class of tokens does the beginning of the given source code translates to - and regular expressions provide a simple way to write patterns that we can match against our source.

Regular expressions is a well established way of pattern matching, since the concept was created in the 1950s by Stephen Cole Kleene. A regular expression is a string of characters that describe a pattern, which then can be matched against a string to do operations like finding the first matching position, finding all matching positions, replacing said matches with other strings, etc. The language of regular expressions have numerous rules which allows us to create very complicated patterns. The regular expression parser engine in JavaScript is well documented [1], but I will provide some examples:

- `/[0-9]+\.[0-9]+/` match floating point numbers (a dot indicates the decimal point)
- `/[a-z]+[a-z0-9]*/i` match an identifier, a name that must start with a character, has to be at least 1 character long and otherwise can contain alphanumeric characters (lower and upper case too)
- `/[0-9]+(?!\.)/` match an integer (a sequence of numbers that is not followed by a dot)

[1] JavaScript RegExp reference https://developer.mozilla.org/en/docs/Web/JavaScript/Guide/Regular_Expressions

The first pattern makes use of ranges. Instead of having to write out all the numbers, we can just say [0-9], which will match any number. Anything in a pair of square brackets is a group of possible characters in that position, for example [abc] would only match a, b or c. The + sign is a quantifier, expressing that at least one character from the group before it must be present. We can see another quantifier in the second rule, the * means zero or more matches of the group it is attached to. In the first and the last group, we can observe a \. combination. This is required because . has a special meaning in regular expressions (it matches any character), so if we want to only match a dot, we need to escape it, which is done by adding a \ before it. In the second rule, we can see a modifier at the end of the expression. That modifier i means that the matching should be done in a case-insensitive way (the default mode is case-sensitive). In the last rule, we can see one of the more complicated kinds of expressions - at the end of the expression, there is a pattern (?!\.), which is a negative lookahead pattern. This means that the whole expression will only match if the inside of this pattern (a dot in this case) can not be matched after the examined string. The negative lookahead won't be included in the result. Several other tools are available in regular expressions to write complicated patterns (such as positive lookahead, capture groups matching exactly n times, etc.), which can make lexical analysis easier, since the regular expression parser engine is doing some of the work for us.

Because of the expressiveness of regular expressions, my implementation of the lexical analyser will be fairly simple, without the need of explicitly working with lookahead rules when transitioning between states, since those can be handled by carefully writing the regular expressions for the tokens.

2.2 The Lexer module

The lexical analyser module is named Lexer - which is in turn a JavaScript class with several subclasses. Lexer implements a regular expression based matching system with an additional ability to specify states, and transitions between these states. This is however optional, and Lexer can work without keeping track of states.

2.2.1 Lexer subclasses

Lexer.Token

The Token class represents a token - it has a value, which is the parsed string that belongs to the token, and a class, which is a reference to the TokenClass the token belongs to.

Lexer.TokenClass

The TokenClass represents a class of tokens - it has a name, an associated regular expression, and a state, which defaults to the DefaultState. It also has a match method that tries to match against a given string, and returns the consumed part of the string alongside with the remaining part. If the regular expression can not be successfully matched against the string, the consumed part will have a value of false, indicating no successful match. A special subclass is always added to the list of classes, which is the EOLTokenClass - it matches the end-of-line characters. Another special subclass is the StatelessTokenClass, which associates the AnyState with itself, which means it will match in any non-strict state.

Lexer.TokenClassGroup

The TokenClassGroup is a container for TokenClasses. It can be used to define transitions if any of the classes in it matched. It has an interface that consists of a push and pop method, a mapping function to transform all elements and a find function that checks if a TokenClass is part of the group or not.

Lexer.State

The State class represents a state in stateful mode - it has a name, and a strict parameter, which is false by default. Two special subclasses exist, one is the AnyState, which is used for tokens that shall match in any state that is non-strict, and other is the DefaultState, which is the one the state machine starts in, and which all classes belong to if no state parameter is given to them.

Lexer.StateTransitionBase

This class is the common core of the state transition classes - it has a from and to member, which represent the states to transition from and to, and it also has a member that takes the Lexer, and checks and retrieves the states for the two members (initially, the to and from members contain the name of the states only).

Lexer.StateTransition

The StateTransition class is the traditional state transition, which has a token class member alongside the inherited to and from. The instances can be created with two or three parameters - in the case of two parameters, the from state will be derived from the state associated with the token class, while if the class and the two states are provided, the from state will be explicitly set. This allows state transitions based on stateless token classes.

Lexer.GroupStateTransition

The GroupStateTransition is similar to the StateTransition class, but it defines a transition that happens when any of the group's token classes match in a certain state. The three parameters are the TokenClassGroup, the from and the to states.

2.2.2 Stateless mode

In stateless mode, Lexer only uses the default state, and does not transition into any other state - and thus the token classes do not need to be initialised with data about which state they belong to.

2.2.3 Stateful mode

In stateful mode, Lexer takes advantage of the rules defined by the States, StateTransitions and GroupStateTransitions given by the user. Lexer keeps track of the current state, and prioritizes group based transitions when deciding what state to transition to.

2.2.4 Skipping whitespaces

To make the lexical analysis easier, Lexer can be told to ignore whitespace characters. This is a very useful option, and thus the default value is true - but some languages require us to keep track of the indentation (e.g. Python), in which case the rules have to deal with the whitespace characters.

2.2.5 The Lexer class

The Lexer class is the core of this module. It provides facilities to add states, transitions, token classes, and has a member that runs the analysis on the given string. This member is called tokenize, and it follows an iterative algorithm - it looks for the first rule that matches the beginning of the string, creates a token based on that rule, does the necessary state transitions and starts the process all over again. The return value is an dictionary with three values:

- success - indicates whether the whole string was consumed or not
- tokens - the list of tokens the process generated
- rest - the rest of the string that was not parsed, if any

2.3 Possible enhancements

In the current state, the Lexer module is easily usable - however, the user needs to define first the states, then the token classes and groups and finally the transitions. This is a quite verbose way of defining the rules for the analysis - which could be done by using a custom data format that describes the rules. This data could be read from a file and preprocessed, which could make the process of defining the rules more robust. It could also be a more compact way of describing the rules.

Currently, the Lexer module can not transform the value of the tokens it creates. This could also make parsing easier, since e.g. integers and floats could be transformed into their numerical values during this stage, rather than having to deal with this process during the traversal of the AST.

Chapter 3

Parsing (Syntactical analysis)

Parsing (or syntactical analysis) is the process of turning the tokens that the lexical analysis produces into a parse tree by using a grammar. The lexical analysis phase only determines the types of things in the source code, while during parsing the higher level structures are revealed. In natural language, the lexical analysis would only reveal whether the word is a noun, verb, adverb etc., while syntactical analysis would turn these words into sentences, paragraphs, chapters and so on.

The input of the parser is the grammar and the tokens, while the output is a hierarchical structure that represents the structure of the source code (or the tokens, in this case).

3.1 Describing the grammar

There are well established ways to describe grammars - one of them is a notation called Backus?Naur form, or BNF. This notation is widely used, and provides an easy way to describe context-free grammars. Context-free grammars are a subset of formal grammars which do not use context to determine which rule to use.

A grammar consists of basic production rules which can be one-to-one, one-to-many or one-to-none rules, for example:

$\langle \mathit{Sum} \rangle$ ::= $\langle \mathit{Sum} \rangle$ ‘+’ $\langle \mathit{One} \rangle$
  | $\langle \mathit{One} \rangle$

$\langle \mathit{One} \rangle$ ::= ‘1’

This grammar describes a language of only two tokens (or terminals), ’+’ and ’1’, and describes two rules:

- A Sum can be a Sum plus (’+’) a One, or a One on it’s own
- A One is a ’1’ character

Or in short, an infinite amount of ones added to other ones is valid in this grammar's rule set, and the sum of the ones can be easily calculated recursively after parsing. The '|' symbol means 'or', other rules are referenced with their name written in '<>'s, and string literals are represented in quotes.

BNF has been extended over it's existence, the two notable extensions is EBNF and ABNF, where the first characters stand for extended and augmented respectively. The parser implemented in this thesis uses the original BNF syntax.

3.2 The parsing algorithm

3.3 The Parser module

Köszönetnyilvánítás

Ez nem kötelező, akár törölhető is. Ha a szerző szükségét érzi, itt lehet köszönetet nyilvánítani azoknak, akik hozzájárultak munkájukkal ahhoz, hogy a hallgató a szakdolgozatban vagy diplomamunkában leírt feladatokat sikeresen elvégezze. A konzulensnek való köszönetnyilvánítás sem kötelező, a konzulensnek hivatalosan is dolga, hogy a hallgatót konzultálja.

Bibliography

[1] M. E. Lesk and E. Schmidt. *Lex - A Lexical Analyzer Generator*. Bell Laboratories, July 1975. URL: `http://hpdc.syr.edu/~chapin/cis657/lex.pdf`.

Appendices

S.1 Source code for the Lexer module

```
class Lexer {
  constructor() {
    this._classes = []
    this._classGroups = []
    this._state = Lexer.DefaultState
    this._stateful = false
    this.skipWhitespace = true

    this.addTokenClass(new Lexer.EOLTokenClass())
  }

  get stateful() {
    return this._stateful
  }

  set stateful(value) {
    if (value) {
      if (this._states === undefined) this._states = []
      if (this._stateTransitions === undefined) this._stateTransitions = []
    }
    this._stateful = value
  }

  get state() {
    if (!this.stateful) throw Error(’Lexer is not in stateful mode’)
    return this._state
  }

  set state(value) {
    if (!this.stateful) throw Error(’Lexer is not in stateful mode’)
    let state = this._states.find(s => { s.name === value })
    if (state)
      this._state = state
    else throw TypeError(’"’ + value + ’" is not a a valid state’)
  }

  addState(name, strict = false) {
    if (name === null) return
    if (!this.stateful) throw Error(’Lexer is not in stateful mode’)

    if (name instanceof Lexer.State) {
      if (this._states.indexOf(name) < 0)
        this._states.push(name)
      return name
    } else {
      let state = this._states.find(s => { return s.name === name })
      if (state === undefined) {
        state = new Lexer.State(name, strict)
        this._states.push(state)
      }
      return state
    }
  }

  addStates(states) {
    states.forEach(st => { this.addState(st) })
  }

  _findState(name) {
    if (name instanceof Lexer.State) {
      if (name === Lexer.AnyState || name === Lexer.DefaultState)
        return name
      return this._states.find(st => { return st === name })
    } else {
      let state = this._states.find(s => { return s.name === name })
      if (state !== undefined)
        return state
    }
  }

  addStateTransition(stateTransition) {
    if (!this.stateful) throw Error('Lexer is not in stateful mode')

    if (stateTransition instanceof Lexer.StateTransition) {
      let tc = this._findTokenClass(stateTransition.class)
      if (tc === undefined)
        throw ReferenceError('"' + stateTransition.class + '" is not a valid' +
                             ' Lexer.TokenClass name')
      stateTransition.class = tc
      stateTransition.checkFromTo(this)
      this._stateTransitions.push(stateTransition)
    } else if (stateTransition instanceof Lexer.GroupStateTransition) {
      let tcg = this._findTokenClassGroup(stateTransition.group)
      if (tcg === undefined)
        throw ReferenceError('"' + stateTransition.group + '" is not a valid' +
                             ' Lexer.TokenClassGroup name')
      stateTransition.group = tcg
      stateTransition.checkFromTo(this)
      this._stateTransitions.push(stateTransition)
    } else throw TypeError('"' + stateTransition +
                           '" is not a Lexer.StateTransition' +
                           ' or Lexer.GroupStateTransition')
  }

  addStateTransitions(stateTransitions) {
    stateTransitions.forEach(st => { this.addStateTransition(st) })
  }

  addTokenClass(tokenClass) {
    if (tokenClass instanceof Lexer.TokenClass)
      this._classes.push(tokenClass)
    else throw TypeError('"' + tokenClass + '" is not a Lexer.TokenClass')

    if (tokenClass.state !== Lexer.DefaultState) {
```

```
    let st = this.addState(tokenClass.state)
    tokenClass.state = st
  }
}

addTokenClasses(tokenClasses) {
  tokenClasses.forEach(tc => { this.addTokenClass(tc) })
}

_findTokenClass(name) {
  return this._classes.find(tc => { return tc.name === name })
}

addTokenClassGroup(tokenClassGroup) {
  if (tokenClassGroup instanceof Lexer.TokenClassGroup) {
    tokenClassGroup.map(cn => {
      let tc = this._findTokenClass(cn)
      if (tc === undefined)
        throw ReferenceError('"' + cn + '" is not a valid' +
                             ' Lexer.TokenClass name')
      return tc
    })
    this._classGroups.push(tokenClassGroup)
  } else
    throw TypeError('"' + tokenClass + '" is not a Lexer.TokenClassGroup')
}

addTokenClassGroups(tokenClassGroups) {
  tokenClassGroups.forEach(tcg => { this.addTokenClassGroup(tcg) })
}

_findTokenClassGroup(name) {
  return this._classGroups.find(tcg => { return tcg.name === name })
}

// This is the main function of the Lexer
tokenize(str) {
  if (this.stateful) {
    // Sort transitions to prioritize group transitions
    this._stateTransitions = this._stateTransitions.sort((a, b) => {
      let x = (a instanceof Lexer.GroupStateTransition) ? 0 : 1
      let y = (b instanceof Lexer.GroupStateTransition) ? 0 : 1

      return x < y ? -1 : x > y ? 1 : 0
    })
  }

  let tokens = []
  while (str !== undefined && str.length > 0) {
    let tl = tokens.length

    this._classes.some(c => {
      if ((!this._state.strict && c.state === Lexer.AnyState)
          || this._state === c.state) {
        let m = c.match(str)
        if (m.consumed !== false) {
          tokens.push(new Lexer.Token(m.consumed, c))
          str = m.rest

          // skip whitespaces if we are allowed
```

```
            if (this.skipWhitespace)
              str = str.replace(new RegExp(’^[ \f\t\v\u00a0\u1680\u180e\u2000-’
                + ’\u200a\u2028\u2029\u202f\u205f\u3000\ufeff]+’), ’’)

            // if we are in stateful mode, check transitions
            if (this.stateful) {
              this._stateTransitions.some(st => {
                let r = st.apply(c, this._state)
                if (r.result) {
                  this._state = this._findState(r.to)
                  return true
                }
              })
            }

            return true
          }
        }
      })

      // no tokens were added because none matched, tokenization failed
      if (tokens.length === tl)
        return { success: false, tokens: tokens, rest: str  }
    }

    // we ran out of characters, everything is tokenized
    return { success: true, tokens: tokens, rest: str }
  }
}

Lexer.State = class {
  constructor(name, strict = false) {
    this.name = name
    // strict state means don’t match things that are assigned to AnyState
    this.strict = strict
  }
}

// special state to assign tokens to which can match in any state
Lexer.AnyState = new Lexer.State(’*’)
// the starting state
Lexer.DefaultState = new Lexer.State(null)

// base class for common stuff
Lexer._StateTransinionBase = class {
  constructor(from, to) {
    this._from = from
    this._to = to
  }

  // sanity check plus turning state names to actual states
  checkFromTo(lex) {
    let f = lex._findState(this._from)
    let t = lex._findState(this._to)

    if (f !== undefined)
      this._from = f
    else throw TypeError(’"’ + f + ’" is not a a valid state’)

    if (t !== undefined)
```

```
      this._to = t
    else throw TypeError(’"’ + t + ’" is not a a valid state’)
  }
}

Lexer.StateTransition = class extends Lexer._StateTransinionBase {
  constructor(tokenClass, from, to) {
    super(from, to)
    this._tokenClass = tokenClass
  }

  get class() {
    return this._tokenClass
  }

  set class(value) {
    this._tokenClass = value
    // convert from default mode (only _from is set)
    if (this._to === undefined) {
      this._to = this._from
      this._from = this._tokenClass.state
    }
  }

  apply(tokenClass, from) {
    if (tokenClass === this._tokenClass && from === this._from)
      return { result: true, to: this._to }
    else
      return { result: false }
  }
}

Lexer.GroupStateTransition = class extends Lexer._StateTransinionBase {
  constructor(tokenClassGroup, from, to) {
    super(from, to)
    this._tokenClassGroup = tokenClassGroup
  }

  get group() {
    return this._tokenClassGroup
  }

  set group(value) {
    this._tokenClassGroup = value
  }

  apply(tokenClass, from) {
    if (this._tokenClassGroup.find(tokenClass) !== undefined
        && from === this._from)
      return { result: true, to: this._to }
    else
      return { result: false }
  }
}

Lexer.Token = class {
  constructor(value, c) {
    this.value = value
    this.class = c
  }
```

```
}

Lexer.TokenClass = class {
  constructor(name, regexp, state = Lexer.DefaultState) {
    // replace regexp with one that only matches at the beginning
    if (regexp instanceof RegExp)
      this.regexp = new RegExp('^' + regexp.source, regexp.flags)
    else
      throw TypeError('"' + regexp + '" is not a RegExp')
    this.name = name
    this.state = state
  }

  match(str) {
    // try to match the beginning of the string
    let m = this.regexp.exec(str)
    if (m === null) {
      return {
        consumed: false,
        rest: str
      }
    } else {
      return {
        consumed: m[0],
        rest: str.replace(this.regexp, '')
      }
    }
  }
}

// shorthand for end-of-line matching, always added
Lexer.EOLTokenClass = class extends Lexer.TokenClass {
  constructor() {
    super('EOL', /[\r\n]+/)
  }
}

// matches in any state
Lexer.StatelessTokenClass = class extends Lexer.TokenClass {
  constructor(name, regexp) {
    super(name, regexp, Lexer.AnyState)
  }
}

// container for classes
Lexer.TokenClassGroup = class {
  constructor(name, classes) {
    this.name = name
    if(classes.constructor !== Array)
      throw TypeError('"' + classes + '" is not an Array')
    this._classes = classes
  }

  map(fn) {
    this._classes = this._classes.map(fn)
  }

  push(classname) {
    this._classes.push(classname)
  }
  pop(classname) {
    this._classes.pop(classname)
  }

  find(tokenClass) {
    return this._classes.find(c => { return c === tokenClass })
  }
}

module.exports = Lexer
```