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.

Translation of the original work description

Modular Interpreter Written in JavaScript

To learn programming, there are some basic tools everyone needs - a compiler, some text editor or an IDE. To acquire and use these can sometimes be non-trivial, especially for the younger generations. Most of the learning material is on the Internet, but to solve the exercises one needs to use the aforementioned programs. By eliminating this extra step and putting the programming environment closer to the materials the life of new pupils can be made easier. The aim of this thesis is to create a modular interpreter in JavaScript, using which anyone can run any kind of code inside the browser (and hence it is modular, it can be easily extended with new languages). The role of the interpreter is to build an Abstract Syntax Tree (AST) from the given source code, and to process the tree to run the equivalent JavaScript code. Using this interpreter anyone could easily build educational websites that can focus on teaching programming, since the learning materials and the environment will be in the same place - in the browser. The requirements are the following:

- Create the core part of the interpreter
- Create an example language module

M Ű E G Y E T E M 1 7 8 2

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 7, 2016

Contents

Kivonat

A programozás megtanulása akadályokkal teli folyamat tud lenni a kezdetekben. A szakdolgozatomban erre a problémára kívánok megoldást javasolni egy JavaScipt alapú moduláris értelmez? elkészítésével, ami bármilyen programozási nyelvet képes értelmezni - amennyiben elkészül a megfelel? modul hozzá - és átfordítani JavaScript kóddá.

A megoldás egy reguláris kifejezés és állapotgép alapú lexikális analizátor modulból - neve Lexer -, és egy szintaktikai elemz? modulból áll, aminek neve Parser és az LALR elemz? algoritmust implementálja, majd ennek segítségével dolgozza fel a bemenetét a BNF-ben megadott nyelvtan szerint.

A szakdolgozatomban leírom a megoldás menetét és elmagyarázom a használt algoritmusok m?ködését - és hogy miért az adott algoritmus mellett döntöttem. Emellett kitérek a használt technológia sajátosságaira hogy a JavaScript mélyebb ismerete nélkül is könnyen értelmezhet? legyen a dolgozatom.

Abstract

Learning how to program can be a problematic experience for new students. In my thesis I propose a solution to the problematic first experiences by creating a modular interpreter that can interpret any kind of programming language - if a module is created for it - which has a module for it, and translate that code to JavaScript.

The solution consists of a regular expressions and state machine based lexical analyser module called Lexer, and a Parser module that implements the LALR algorithm to parse grammars defined in BNF.

In my thesis I describe the solution and explain the used algorithms, and the choice behind them. I also give a brief explanation of the used technologies to make the solution easy to understand even for those who do not know much about JavaScript or parsing algorithms.

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 [4], 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]+(?![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 `(?![0-9]*\.)`, which is a negative lookahead pattern. This means that the whole expression will only match if the inside of this pattern (some numbers and 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. Two special subclasses are always added to the list of classes, which are the EOLTokenClass - it matches the end-of-line characters - and the EOFTokenClass - which matches the end-of-file indicator. 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:

$\langle Sum \rangle$ ::= $\langle Sum \rangle$ '`+`' $\langle One \rangle$
  | $\langle One \rangle$

$\langle One \rangle$ ::= '`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

One of the big choices someone who sets out to write a parser shall make is which parsing algorithm to use. The algorithms can be divided into two groups, namely top-down parsing algorithms and bottom up parsing algorithms. These differ in the following ways:

- Top-down parsers try to fit a parsing tree to the input string by looking at the tokens left to right, and looking for any matching grammar rules. This means that the parse tree is built from the higher level structures, and the most basic rules get parsed the last - hence the name top-down parsing. Since two rules can have similar prefixes, in ambiguous cases all the possible choices are explored. The list of top-down parsers include:
  - Recursive-descent parser - these parsers use recursive functions to parse the input. These parsers closely resemble the grammars they are generated from, but they tend to have performance problems because of the recursion. They can be applied to any grammar, but the algorithm is not guaranteed to terminate.
  - LL(k) parser - an LL(k) parser is a state machine based parser which parses the tokens from left to right and performs leftmost derivation (the first non-terminal from left to right is rewritten), with k lookahead. What this means is that during parsing, the decisions are based on not just the current token, but also the next k tokens. These parsers are generally easy to construct, so many computer languages are built to be LL(*1*) - which means the language's grammar can be parsed by an LL(*1*) parser. LL(*1*) grammars can also be parsed by a recursive-descent algorithm, and are guaranteed to terminate.
- Bottom-up parsers do parsing the other way around - they first find the most basic structures, and build the higher level structures from the already recognized lower level ones. A different name for bottom-up parsers is shift-reduce parsers, since the resulting parses uses these main actions during parsing. Some bottom-up parsing algorithms are:
  - LR(k) parser - similar to the LL(k) parser in the sense that it reads from left to right, it is state machine based and uses k lookahead, but these parsers

perform rightmost derivation (usually in reverse). These parsers can handle the grammars they accept (unambiguous, context-free) very efficiently (in linear time).

  – LALR parser - a variation of the LR(*1*) parser which is more memory efficient, since a smaller table is used by the state machine. The downside is that an LALR parser only accept a subset of LR(*1*) grammars. Nevertheless this algorithms is widely used by parser generator tools like Yacc and GNU Bison.

During development I have tried two of these algorithms. One was a naive implementation that was essentially a recursive-descent-like algorithm, while the final version implements an LALR parser construction algorithm.

3.2.1 Naive implementation

My first idea to parse a source string was to build a parser following an intuitive, greedy algorithm. The basic idea was to read the grammar rules, and try to match the beginning of the string with one of the rules by recursively matching the right hand side of each rule. Surprisingly this approach worked quite well in some cases, but had some major flows which led me to abandon this idea and look for an algorithm that was proven to be effective:

- The parser was greedy, it accepted the rule that consumed the most of the input string, so rules that had the same prefix were not handled well by the algorithm.
- Left-recursive rules caused an infinitely deep recursion, since the parser always tried to match the same rule over and over again. This is something that is a common problem in the family of top-down parser, although an algorithm was constructed [2] that is top-down and can handle left-recursive grammars.

After realizing that the naive algorithm has severe limitations, I looked for a better alternative - one that was not too difficult to implement, but was also efficient. My first idea was to use an LL parser, but once I realized that this would not solve the left-recursion problem, I turned to the slightly more complicated LR parsers. After a bit of research, I have found a very easy-to-follow description of an LALR parser construction algorithm, and so I have based my work on this.

3.2.2 LALR - LookAheadLR

The invention of LALR algorithm is attributed to Frank DeRemer [1], but he did not actually create an algorithm to construct these kinds of parsers. Since then multiple algorithms have been created to construct these kinds of parsers. The algorithm I have used - and which is described Stephen Jackson in his "A Tutorial Explaining LALR(1) Parsing" [3] - is an algorithm that first builds a full LR(*1*) table which is then reduced to an LALR table. This is not the most efficient way to construct an LALR parser, since merging of some rules only happens during a late stage, while merging could be done in one of the earlier stages too, with the downside of being slightly more complicated. I have decided to closely follow the tutorial as much as I can, since memory during table construction should not be a huge problem (nowadays there is plenty enough memory in the use-case scenarios of this parser generator, we are not aiming to create a parser for an embedded system with limited memory). The construction of an LALR parser by reducing an LR(*1*) is done in the following steps:

Adding a START rule to the grammar

This is more of a preparation step for the real algorithm - we need to have a specific starting rule for our grammar, which must only have one item on the right hand side. We could let the writer of the grammar deal with this convention, but it's easier to just create a starting rule for ourselves that has the first rule of the grammar on it's right hand side. In my case, this rule gets the special name "#S" which is otherwise not a valid name, so this rule can not be added to later during the grammar definition. This rule is prepended to the grammar, so this rule will be the new first rule of the grammar.

Finding the item sets

The first step is finding the item sets - or to be more precise the canonical collection of LR(*0*) items. These items represent partially parsed rules, with a dot after the last parsed item on the right hand side (or in the case of a rule that has no parsed elements, at the beginning).

$\langle \mathit{Sum} \rangle$ ::= $\langle \mathit{Sum} \rangle$ •'+' $\langle \mathit{One} \rangle$

The above item represents a *Sum* rule that expects a '+' as the next parsed token.

The item sets are constructed using the following algorithm:

1. The first set starts with the starting rule, with the dot at the beginning of the right hand side.
2. We expand the set by adding every rule (with the dot at the beginning) that maps from anything on the right hand side of the rules in the set. We do this until the set does not change any more.
3. We calculate what inputs the set is expecting (what items are after the dot in each rule) and we begin a new set using each of these inputs. We begin a new set by adding each item from the current set that expects the input we chose, but with the dot moved one position forward.
4. We expand each of the newly created sets by jumping to step 2, until we eventually expand all sets with all possible inputs.

Finding the translation table elements

Using the sets we have built in the previous step, we can find the translation table elements. We can treat the sets from now on as the states of the parser, and the translation table will tell us which state is next if we get a certain input. This can be easily derived from the previous step, since any time we create a new set, we know which input lead to that state.

Finding the extended grammar rules

The extended grammar consists of extended grammar rules - which are basically the original grammar rules with some added notation. Every terminal and non-terminal in the left and right hand side is augmented with some additional information about state transitions (a *from* and a *to set*), following this algorithm:

1. The left hand side's *from set* is the item set the rule belongs to, while the *to set* is read from the translation table of the set - specifically the set that we would transition to if the left hand side was given as input. In the case of the starting rule, this set does not exist, but the starting rule can not appear on the right hand side of any rule, since that is a special rule that we ourselves added.
2. The right hand side is calculated iteratively. The first item's from and *to set* is set similarly to the left hand side, while the following items *from set* will be the previous item's *to set* - basically forming a chain of transitions. The goal here is to determine which is the final set that the rule transitions to.

If we take every item from every item set with the dot at the beginning of the right hand side, and apply this procedure to those items, we get the full extended grammar.

Finding the First and Follow sets

The first sets are calculated for the terminals and non-terminals of the extended grammar with the following algorithm:

1. If x is a terminal, First$(x) = x$
2. If V is not a terminal, then examine the rules that have V as the left hand side, and iterate over the right hand side using the following rules (ϵ represents the empty string):
   (a) If the current item is a terminal, stop the iteration
   (b) If the current item is not a terminal, add the First set of that item to the set of V (minus ϵ). If the current item's First set does not contain ϵ, stop the iteration, otherwise continue with the next item.

(c) If we are currently at the last item of the right hand side and it's first set contains ϵ, add ϵ to the First set of V

This algorithm can be calculated iteratively - using these rules we mutate the First sets until we get to a pass over the sets that does not mutate the sets any more.

The final sets are calculated for the same items, but using the Firsts sets in the algorithm, which is the following:

1. Add the end-of-file token to the Follow set of the starting rule
2. If x is a terminal, Follow(x) $=$
3. If V is not a terminal, then examine the rules that have V appear on the right hand side, and do one of the following (ϵ represents the empty string again):
   (a) If V is not the last item on the right hand side, add everything from the First set of the next item (minus ϵ) to the Follow set of V. If the next item's First set contained ϵ, add everything from the left hand side's Follow set to the Follow set of V too
   (b) If V is the last item on the right hand side, add everything from the left hand side's Follow set to the Follow set of V

The Follow sets can be calculated iteratively the same way we calculated the First sets.

Merging the rules of the extended grammar

We now can merge the extended grammar rules. This is the step that reduces the size of the Action/Goto table, and thus the required memory for the parser.

We can merge two rules if their original rule is the same (that is, they are created from the same original grammar rule, so stripping away the state transition informations would leave us with identical rules) and their final sets are also the same. When we merge two rules, the merged rule's Follow set will be the union of the Follow sets of the merged rules.

Creating the Action/Goto table

Now the core of the parser, the Action/Goto table can be constructed. Using this table, we can then parse any input by applying simple rules - the Actions. We do this by keeping track of the following: a stack, an output array, the input, and an array of parse tree nodes (which is equivalent to the output, but in a tree structure).

The Goto part of the table can be read from the translation table directly - every non-terminal transition is copied as a Goto. The Action part of the table can contain 4 types of actions:

- Accept - this mean that the input has been parsed, and the algorithm is finished
- Error - this represents a syntax error - a token was given as input that was not expected. This action is the default if none specified directly
- Shift - pushes the state associated with it to the parser stack and removes the first token from the input
- Reduce - Pushes the rule associated with it to the parser output, removes as many items from the parser stack as many items the rule's right hand side contains. The same amount of nodes are removed from the nodes list, and a new node is added which has the removed nodes as it's children, and has the rule as it's rule. The next state is deduced from the Goto table - we use the state on top of the stack as a temporary state, we use the left hand side of the rule as an input to the Goto table, and we push the item in the Goto table to the stack.

We start out in state 0, which contains the starting rule. Then we can start parsing by examining the beginning of the input and reading the action we need to perform from the table. Doing this until we hit an Accept or Error action will result in either a syntax error, or a valid parse tree. This tree then can be transformed into an abstract syntax tree by removing unnecessary data - we merge nodes that only have one children with their child, we remove children which have terminal values originally - these contain no information -, and we remove empty strings from the children after reducing them too.

3.3 The Parser module

The Parser module is the one that implements the algorithms and the classes required. While constructing this module, I tried to keep things separated - I've ended up with one private method for the main class for each step of the parsing algorithm, and a parse method to parse code and return the AST.

3.3.1 The Parser subclasses

Parser.RuleTerminalBase

The base class for the following two classes. It defines the isEpsilonRule and isTerminalRule methods.

Parser.BNFRule

Represents a rule read from the BNF definition. Has a name, and a list of subrules (which corresponds to the right hand side of the rule in a two dimensional array, the alternatives are on the first level and the actual rules on the second). The token classes are represented using this class, in that case a tokenClass member is set, and the subrules list is empty.

Parser.BNFTerminal

This class represents a terminal value read from the BNF definition.

Parser.LR0Item

This class represents an item of the canonical collection of LR(*0*) items. It has a rule, an index to indicate which subrule alternative to use, and a dot value to indicate which item the dot is before.

Parser.LR0ItemSet

It represents a set of LR0Items. It stores the items and the translation table, and has methods used during calculation of the item sets (getting a list of items after dots, creating new items for a new set, expanding the set).

Parser.ExtendedGrammarItem

It represents a terminal or non-terminal of the original grammar, but with the additional information about *from* and *to set*s, and also the First and Follow sets. A set of these items is maintained during parser generation.

Parser.ExtendedGrammarRule

This class is the wrapper for an extended grammar rule, and it keeps track of the left hand side and the right hand side of the original rule. It has methods to check if it is mergeable with another rule, and to get the final set of the rule.

Parser.Action

This is the base class for the actions in the Action/Goto table. For convenience reasons, the Goto class is also a subclass of this. All subclasses implement an execute method that modify the parser's state (the parser is passed as a parameter). It also has a member to store the input in (the token that triggers the action).

Parser.Accept, Parser.Shift, Parser.Reduce, Parser.Goto

These classes represent the items in the Action/Goto table, each implementing the algorithm associated with the action in their execute method.

Parser.Node

This is the class that represents a node in the parse tree and AST. Each node has a reduce method which does the conversion for the node and it's subtree from a parse tree node into an AST node (removing unnecessary data and merging).

3.3.2 The Parser class

The Parser class is the class that encapsulates all the algorithms and does the actual parsing of the source tokens, in it's parse method. This is the class that is exported from the module.

3.4 Possible enhancements

The current implementation runs quite well, can handle LALR grammars with left-recursive rules and generate ASTs from those. A good way to improve the parser would be to implement different algorithms for parsing, which would ease the limitation of only accepting LALR grammars.

Chapter 4

The Language module and two examples

4.1 The Language module

The Language module consists of only one class - the Language class - which encapsulates the Lexer and Parser classes. It has two getters - one for the Lexer and one for the Parser, and a method to build and return the AST from the raw source code string. The module exports the Language class, which has the Lexer and Parser classes as subclasses too.

4.2 Examples

I have written two example language modules to test the modules - one is a math expression parser, and the other is a very small subset of the Logo programming language - a for cycle, and the drawing statements are accepted, and of course the parameters can be math expressions.

4.2.1 The math parser

The way I imagined people could create language modules is by extending the Language class - and this is what I have done in this case. In the constructor I initialize the Lexer token classes, and the parser with the BNF grammar. I added two more methods - an executeOne and an execute method. The execute method builds the AST using the parser's parse method, and calls executeOne on the main node. The executeOne method then recursively executes the subtrees of the nodes - it checks for known types based on the names of the token classes, and parses the integers and floats into their numerical values. If the current node is not a terminal, then it first calls itself on that node to determine it's value. Once all the numerical parameters are ready, it does the calculation, and returns the resulting number. This is done using switch statements to handle each rule in it's own way.

```
const Language = require('../lib/language')
const { Lexer, Parser } = Language

class BasicMath extends Language {
  constructor() {
    super()

    this.lexer.addTokenClasses([
      new Lexer.TokenClass('int', /[0-9]+(?![0-9]*\.)/),
      new Lexer.TokenClass('float', /[0-9]+\.[0-9]+/),
      new Lexer.TokenClass('char', /\S/)
    ])

    this.parser.fromBNF(
      `<S> ::= <S> <SEP> <E> | ""
      <SEP> ::= "," | "" | <Token-EOL>
      <E> ::= <E> <PM> <T> | <T>
      <T> ::= <T> <MD> <H> | <H>
      <H> ::= <H> "^" <F> | <F>
      <PM> ::= "+" | "-"
      <MD> ::= "*" | "/"
      <F> ::= "(" <E> ")" | <Token-int> | <Token-float>`
    )
  }

  executeOne(node) {
    if (node.rule.class !== undefined) {
      switch (node.rule.class.name) {
        case 'int':
          return parseInt(node.rule.value)
          break;
        case 'float':
          return parseFloat(node.rule.value)
          break;
        default:
          return node.rule.value
      }
    }

    let parameters = node.children.map(child => {
      return this.executeOne(child)
    })

    switch (node.rule.name) {
      case 'E': case 'T':
        switch (parameters[1]) {
          case '-':
            return parameters[0] - parameters[2]
            break
          case '+':
            return parameters[0] + parameters[2]
            break
          case '*':
            return parameters[0] * parameters[2]
            break
          case '/':
            return parameters[0] / parameters[2]
            break
          default:
            throw Error('Unknown rule')
          }
          break
        case 'H':
          return Math.pow(parameters[0], parameters[1])
          break
        case 'F':
          return parameters[0]
          break
        case 'S':
          if (parameters.length === 1)
            return parameters[0]
          return parameters[0] + ', ' + parameters[parameters.length - 1]
          break
        default:
          throw Error('Unknown rule')
      }
    }

  execute(code) {
    let ast = this.buildAST(code)
    return this.executeOne(ast[0])
  }
}

module.exports = new BasicMath()
```

I have added some extra rules to verify that the parser can handle optional rules (rules with ϵ as an alternative) and left-recursion. By carefully constructing the grammar I have ensured that the operator precedence is as it should be ($nthpower > */ > +-$).

```
const BasicMath = require('./examples/math')
global.BasicMath = BasicMath
console.log(
  BasicMath.execute(`10 ^ ((10 - 5 + 4) / (6 - 3)), 10 22/10
  432.432/10`)
)
```

Running this piece of code results in the following: '`1000, 10, 2.2, 43.2432`', which is indeed the expected output.

4.2.2 The Logo subset parser

The way the Logo subset parser is constructed is similar to the math expression parser.

It uses the following grammar:

⟨*Program*⟩ ::= ⟨*Program*⟩ ⟨*Expression*⟩
  | ''

⟨*Expression*⟩ ::= ⟨*Command*⟩
  | ⟨*For*⟩

⟨*Command*⟩ ::= ⟨*Command-name*⟩ ⟨*Math*⟩

⟨*Command-name*⟩ ::= ‘`f`’
  | ‘`b`’
  | ‘`l`’
  | ‘`r`’

⟨*For*⟩ ::= ‘`for`’ ⟨*Math*⟩ ‘`[`’ ⟨*Program*⟩ ‘`]`’

⟨*Math*⟩ ::= ⟨*Math*⟩ ⟨*PM*⟩ ⟨*T*⟩
  | ⟨*T*⟩

⟨*T*⟩ ::= ⟨*T*⟩ ⟨*MD*⟩ ⟨*H*⟩
  | ⟨*H*⟩

⟨*H*⟩ ::= ⟨*H*⟩ `^` ⟨*F*⟩
  | ⟨*F*⟩

⟨*PM*⟩ ::= ‘`+`’
  | ‘`-`’

⟨*MD*⟩ ::= ‘`*`’
  | ‘`/`’

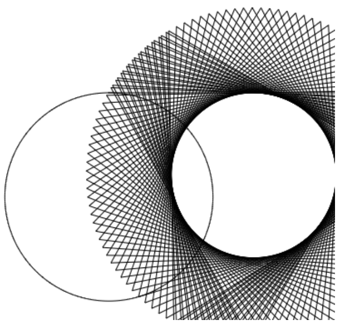
⟨*F*⟩ ::= ‘`(`’ ⟨*Math*⟩ ‘`)`’
  | ⟨*Token-int*⟩
  | ⟨*Token-float*⟩

```
const Logo = require('./examples/logo')
global.Logo = Logo
```

By providing a callback parameter, and implementing simple drawing on a HTML5 canvas in JavaScript, if we bundle the interpreter script and handle the necessary things in the browser, we can actually create drawings fairly easily.

```
logo.execute(‘for 360 [f 2 r 1] for 10^2+50 [f 321 r 121 for 2 [r 1 l 1]]‘)
```

This code generates the following image:



Acknowledgement

I would like to acknowledge my friend Blumenschein Volfram for helping me by checking my work for grammatical errors.

Bibliography

[1] Franklin L DeRemer. *Practical translators for LR (k) languages.* PhD thesis, MIT Cambridge, Mass., 1969.

[2] Richard A Frost, Rahmatullah Hafiz, and Paul C Callaghan. Modular and efficient top-down parsing for ambiguous left-recursive grammars. In *Proceedings of the 10th International Conference on Parsing Technologies*, pages 109–120. Association for Computational Linguistics, 2007.

[3] Stephen Jackson. *A Tutorial Explaining LALR(1) Parsing*, 2009. URL: `http://web.cs.dal.ca/~sjackson/lalr1.html`.

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