ECOTE preliminary report: Top-down parser with backtracking

Michał Szopiński 300182

May 11, 2022

1 General overview and assumptions

The goal of this project is to write a program to parse and produce a syntax tree for an arbitrary input file using an arbitrary grammar.

The parsing is to be implemented using a top-down recursive descent algorithm, i.e. one that attempts to find a combination of productions matching the input token sequence, starting from the root production. Backtracking means that the algorithm may abandon previously chosen productions if it discovers that they cannot lead to a match.

Because a parser operates on tokens, which are produced during the lexical analysis stage, the program must have a built-in lexer utility. To reduce complexity, the lexeme recognition algorithm is hard-coded and not customizable. The built-in lexer recognizes tokens that are common to popular C-like languages.

As mentioned before, the program checks arbitrary inputs against arbitrary grammars. This implies that the user supplies two files, one containing the input and one containing a description of the grammar.

The program tokenizes both files using the built-in lexer and parses the grammar description file using a hard-coded grammar description meta-language. The produced syntax tree is then validated and transformed into a grammar descriptor object, which is in turn used to parse the input file. As such, the same parser may be used to process both input files.

The program implements rudimentary diagnostics and error handling. In particular, the user may receive lexical, parse and semantic errors during each stage of processing. Changes in the syntax tree are also displayed as they occur.

2 Functional requirements

The programming language of choice for this project is Python. Its dynamic typing makes it suitable for straightforward operations on complex data types. The previous proposal of using C/C++ has been withdrawn.

2.1 Lexical analysis

Because the lexical analyser is hard-coded, it must strive to resemble the lexical ruleset of mainstream C-like languages, so as to match user expectations. A set of popular token categories is defined:

Category	Examples	Description
Identifier	`hello_world123`	Used for variable names and keywords.
Operator	`$ ++ ===`	Used to define multiple-character non-identifier entities.
Separator	`, ; ( }`	Used to define single-character non-identifier entities, typically neighboring each other.
String literal	`"can't"` `'won\'t'`	Incorporates rules for string enclosure and escaping.
Number literal	`123 +1.0`	Incorporates rules for digit sequences, sign prefixes and decimal points.
Comment	`//hello` `/* world */`	Incorporates rules for single-line and multi-line comments.
Invalid	`123abc "hello`	Marks lexical errors. Used for diagnostics.
End of file		Denotes the end of the input file. Used for grammar description.

2.1.1 Scanning and evaluation

Most of the above tokens are produced during the scanning phase. The end-of-file token is appended at the end of the token sequence during the evaluation phase. Comment tokens are removed from the sequence before they reach the parser. The presence of invalid tokens prevents the program from progressing to the parsing phase.

2.2 Grammar description meta-language

Once the grammar description file has been tokenized using the universal lexer, the program applies a predefined meta-grammar to parse the file into a syntax tree for further processing.

At the top level, the meta-language is a set of definitions describing each production in the language. The fundamental building blocks for definitions are binary **compound expressions** and **terminal expressions**.

Compound expressions are the framework for backtracking recursive descent logic. They accept two arguments and define the logical relation between them. Three such expressions are defined:

1. **Concatenation** - accepts if both arguments accept.
2. **Optional concatenation** - accepts if either both or only the second argument accepts.
3. **Alternative** - accepts if either argument accepts.

Terminal expressions are used to describe the terminal symbols of the language. Three kinds of such tokens may be discerned:

1. **String literal** - accepts a token of any category whose value is equal to that enclosed in the literal.
2. **Identifier**
   (a) **Reserved identifier** - identifier belonging to the set `identifier string_literal number_literal end_of_file`. Accepts a token of any value belonging to the matching category.
   (b) **Arbitrary identifier** - resolves to a different definition in the grammar.

2.2.1 Formal description of the meta-language

The following is a formal description of the above rules, written as a grammar description object using Python syntax:

```
meta_grammar = {
    "root": Alternative(
        "definitions",
        Terminal("end_of_file")
    ),
    "definitions": Concatenation(
        "definition",
        Alternative(
            "definitions",
            Terminal("end_of_file")
        )
    ),
    "definition": Concatenation(
        "definition_key",
        Concatenation(
            Terminal("operator", "="),
            Concatenation(
                "definition_expression",
                Terminal("separator", ";")
            )
        )
    ),
    "definition_key": Terminal("identifier"),
    "definition_expression": "expression",
    "expression": Alternative(
        "concat_expression",
        Alternative(
            "opt_concat_expression",
            Alternative(
                "alt_expression",
                Alternative(
                    "expr_identifier",
                    "expr_string_literal"
                )
            )
        )
    ),
    "expr_identifier": Terminal("identifier"),
    "expr_string_literal": Terminal("string_literal"),
    "concat_expression": Concatenation(
        Terminal("identifier", "concat"),
        "argument"
    ),
    "opt_concat_expression": Concatenation(
        Terminal("identifier", "opt_concat"),
        "argument"
    ),
    "alt_expression": Concatenation(
        Terminal("identifier", "alt"),
        "argument"
    ),
    "argument": Concatenation(
        Terminal("separator", "("),
        Concatenation(
            "expr_arg1",
            Concatenation(
                Terminal("separator", ","),
                Concatenation(
                    "expr_arg2",
                    Terminal("separator", ")")
                )
            )
        )
    ),
    "expr_arg1": "expression",
    "expr_arg2": "expression"
}
```

There are two additional semantic constraints: (1) there must be a definition named `root`, and (2) there mustn't be any definitions whose names belong to the set of reserved identifiers.

2.3 Top-down parser

The parser is the core feature of the software. It takes the root production of the given grammar and attempts to find a set of productions stemming from the root which could accept all the tokens in the sequence. It does so by implementing the logical rules of the three compound expressions discussed earlier.

Each step of the parser is a recursive call to a function which processes a single binary or terminal production. If it is determined that the set of logical rules for that production can not yield a combination of productions to parse the entire token sequence, the function generates an exception and returns control to its caller.

Exceptions don't originate at compound productions, they are merely propagated upwards by them. All exceptions stem from terminal productions at the leaves of the production tree. A terminal symbol matches the current token in the sequence against its signature and either increments the token iterator (”accepts” the token), or raises an error to be handled by the logic of compound productions higher in the syntax tree.

Backtracking is achieved by remembering the state of the token iterator at the initialization of a compound production. If one path fails to parse the token sequence, the iterator is reset and a different path is tried. If neither path succeeds, the error from the later path is propagated upwards, where backtracking may occur as well. If both paths are exhausted at the root level, the token tree is declared unparseable.

The above algorithm merely checks the validity of the token sequence against the grammar. To build a parse tree, each call to the parsing function may additionally result in the addition of a node to a data structure mirroring the history of chosen productions. Backtracking rules apply.

2.4 Grammar generator

Parsing the grammar description file against the meta-grammar yields a syntax tree containing named and anonymous nodes corresponding to various productions. The grammar generator searches this tree for definitions and recursively parses them to build a dictionary of named productions (a grammar description object) for the input file.

3 Implementation

3.1 General architecture

The program is divided into the entry point script and several modules, each providing a separate layer of functionality.

Module	Description
Entry point	Handles user interaction, file I/O and data flow between the main modules of the program.
Diagnostic	Contains functions for displaying data, visualizing data structures and printing diagnostic messages.
Lexer	Implements a finite-state machine to parse the raw input into tokens.
Lexer handlers	Defines the delta function of the finite-state machine.
Meta-language	Contains the hard-coded grammar description object for the meta-language.
Productions	Defines classes for compound and terminal productions.
Parser	Utilities for initializing a top-down recursive descent.
Parser handlers	Logical rules for parsing productions.
Grammar	Syntax tree analysis and grammar description object generation.

3.2 Data structures

3.2.1 Productions

Four classes are defined to describe the three non-terminal and the single terminal production types: `Concatenation`, `OptionalConcatenation`, `Alternative` and `Terminal`.

The non-terminal productions hold two slots for their children nodes. They are separate because the parser function looks at the type of the production to invoke the appropriate handler.

The terminal production holds a slot for the category and the value of the token it matches against. Each may be null to disable verification for that field. A method is provided for matching against tokens.

3.2.2 Syntax node

The `Node` class holds a single node of the syntax tree. It has a name field for named productions and a children field. It may hold other nodes, representing compound productions, or tokens, representing terminal productions. Named terminal productions are wrapped in a single-child `Node` object.

To facilitate backtracking, the class exposes methods for adding and removing children without directly accessing the children field.

3.2.3 State classes

The classes `MachineState` and `ParserState` are data aggregates representing the internal state of the lexer and the parser, respectively.

The `MachineState` class contains an assortment of states necessary to provide context for tokenization.

The `ParserState` class holds the token sequence and the grammar that the parser is currently operating on, as well as the token iterator.

3.3 Detailed implementation

3.3.1 Lexer

The lexer is a finite-state machine. The lexing process begins by initializing the state. The input file is then scanned character by character to determine which characters constitute which tokens. On the boundary between tokens and non-tokens (or neighboring tokens), the currently recognized token is appended to the output sequence.

Once the entire input is parsed, an evaluation phase occurs, where transformations are performed on the output sequence. Comments are removed and the end of file token is appended.

3.3.2 Parser

The parser is initialized by creating a “super-root” node and invoking the parser function on the first token in the sequence.

The parser function accepts three arguments:

1. The current parser state, `ParserState`.
2. The prescribed production, either one of the four production types or a string to be resolved from the grammar description object.
3. The parent node, where the parsed production is to be added as a child node.

The root element is parsed by specifying the prescribed production as `"root"` and the parent node as the super-root. Upon exit, the entry point function returns the first child of the super-root, i.e. the root node.

If the production is specified as a string, the main parser function performs name resolution to obtain the corresponding production class. The specified production string then becomes the name for the node to be appended to the parent node. Named productions aid in syntax tree analysis.

Once the production class is resolved, the main function looks up and invokes the appropriate handler for that production.

3.3.3 Terminal handler

Terminal handlers accept input tokens and are the source of syntax errors, crucial to the backtracking mechanism. The root node may be a terminal node, in which case the language only accepts a single token.

The terminal handler resolves the token at the current index and compares it against the production's signature. In case of category or value mismatch, a syntax error is raised and propagated upwards in the call stack.

Upon success, the token iterator is incremented and a token is added to the parent node. If the terminal production is a named production, the token is wrapped in a single-child named node first.

3.3.4 Non-terminal handlers

The concatenation handler parses its two children in sequence. If any of them fails, the error is propagated. No backtracking occurs in this handler.

The optional concatenation handler tries two paths: one where the first child is skipped and one where it is not. If both paths fail, the error from the second child is propagated.

Backtracking is implemented by saving the token iterator before attempting the first path. If the first path fails, the iterator is restored and the second path is attempted. A new node is created for each of the paths. If a path succeeds, the corresponding node is appended to the parent.

The alternative handler is implemented in a similar way, the only difference being the logical rules of the attempted paths.

3.4 Grammar generator

The grammar generator traverses the syntax tree of the parsed grammar file in search of all named nodes corresponding to definitions.

For each definition, it searches nearby descendant nodes for the definition key and expression. The expression is evaluated recursively until all terminal productions are found. Found compound productions are translated into their production classes. String literals are translated into tokens with the given value. Identifiers are translated into tokens of the given category or into references to other definitions.

When definitions are evaluated and prior to exit from the entry point function, semantic rules are validated: the grammar must define a root and it mustn't use reserved identifiers as keys.

4 Test cases

The most important test case validates backtracking. Given the following production:

```
root = concat(
    "alpha",
    opt_concat(
        identifier,
        "beta"
    )
)
```

It must be able to recognize the string `alpha beta`. A naive greedy algorithm would consume the token `beta` as the identifier rather that the token `"beta"`, leaving `opt_concat` unable to consume `beta` as its second child, thus failing the validation.

A more exhaustive test case would be to provide a grammar for JSON and successfully validate a file against it.