COMP 335 – ASSIGNMENT 10

Ethan Benabou
40032543

1 – LEXICAL SPECIFICATIONS

These regular expressions are written in PCRE (Perl Compatible Regular Expression) notation which is a very common syntax for expressing regular expressions in various programming languages.

Id: `[aA-zZ]([aA-zZ]|[0-9]|_)*`

Integer: (0|[1-9][0-9]*)

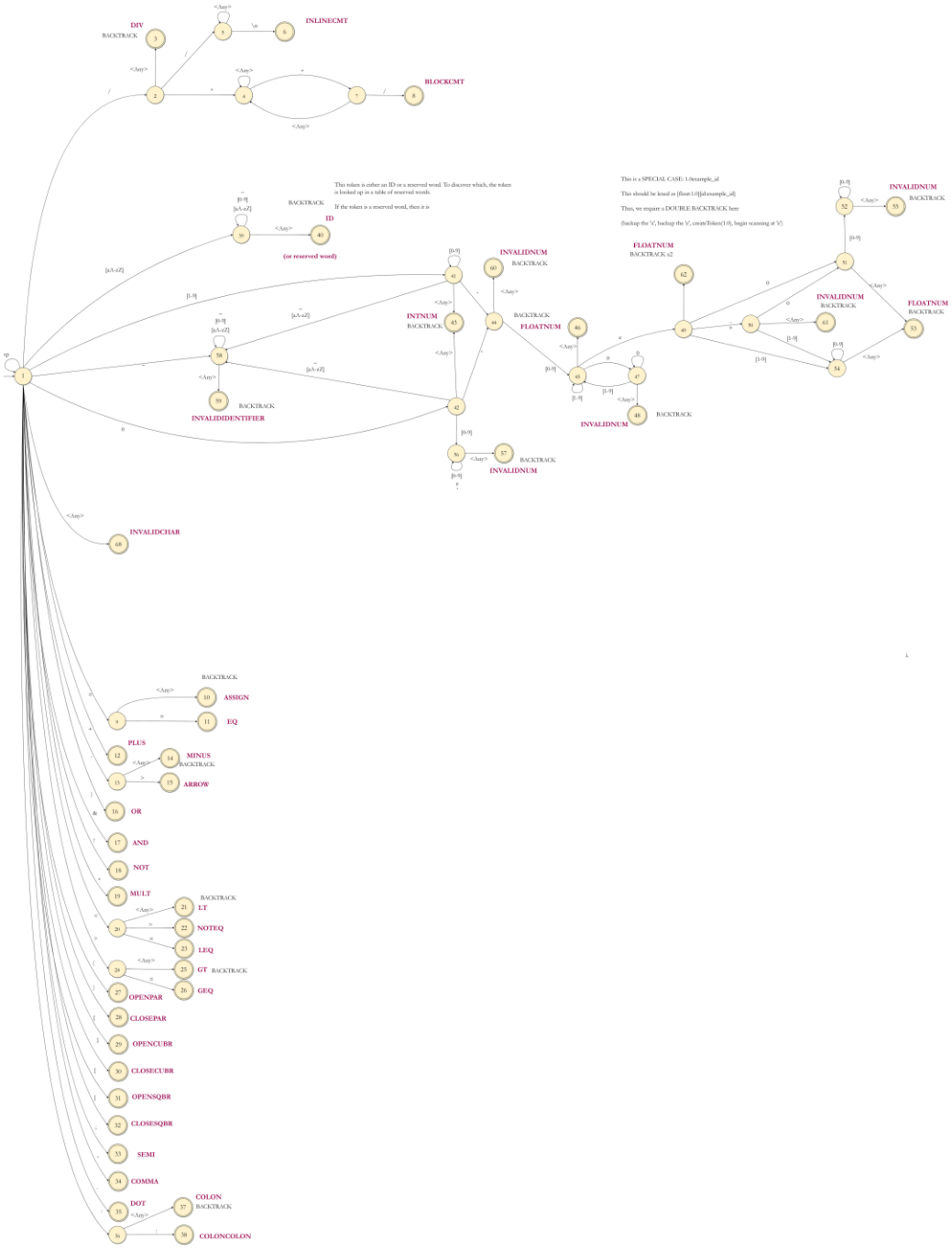
Float: (0|[1-9][0-9]*)\.(0|[0-9]*[1-9])(e(+|-)?(0|[1-9][0-9]*))?

Single-line comments: //.*\n

Multi-line comments: /*.**/

2 – FINITE STATE AUTOMATON

The diagram is very large, please view the submitted high-res PNG. A preview image has been inserted below. Note that the implementation differs slightly from the DFA picture – the real program adds support for so-called *imbricated comments* (i.e. nested comments). The nested comments require keeping a stack and I figured it would be simpler to display the DFA as it was before adding support for imbricated comments. This means that the comment branches (both inline and block comment) are not implemented in quite the same was as is shown in the diagram below.



3 – DESIGN

The scanner is implemented in Golang, and the main module is called: `github.com/obonobo/esac`

I begun the implementation by designing a set of small but powerful interfaces and data types for the components that I was going to implement. Here are the interfaces and types I came up with:

- `PACKAGE: github.com/obonobo/esac/core/scanner`
  - `Scanner interface`
  - `CharSource interface`
  - `Token struct`

- `PACKAGE: github.com/obonobo/esac/core/tabledrivenscanner`
  - `This package contains an implementation of the Scanner interface from the previous package.`
  - `Table interface`

The idea is to start the implementation with the `scanner.CharSource` interface. This component must feed UTF-8 characters, possibly buffered, into the scanner. When the scanner component is instantiated, it gets injected with a CharSource implementation (dependency injection). There were a few requirements that I wanted to support with the CharSource implementation:

- **Arbitrary byte source:** the CharSource should support reading raw bytes from any data source, including files as well as in-memory buffers.

- **UTF-8:** CharSource should assume UTF-8 encoding of the data source, using the Go stdlib utf8 package for decoding. This means that the chars fed into the scanner are variable-width codepoints i.e. they use the `rune` data type in Go rather than the `byte` data type.

- **Support for backtracking:** the scanner must be able to feed forward one char at a time as well as backup one char at a time.

- **Reporting of line + col number:** the CharSource must support reporting the current line number where its cursor is located as well as the current column number.

```go
package scanner

// The CharSource interface is kind of like the io.RuneScanner interface except
// that it doesn't report sizes and the `UnreadRune()` method (named
// `BackupChar` in this interface) also returns the unread rune
//
// It also is able to report the current line and column number that it is on.
//
// CharSource should assume UTF-8 encoding
//      Ethan Benabou, 3 days ago • Changed project name to "esac" and added MI
type CharSource interface {
    // Reads the next character in the input
    NextChar() (rune, error)

    // Back up one character in the input in case we have just read the next
    // character in order to resolve ambiguity
    BackupChar() (rune, error)

    // Reports the current line number
    Line() int

    // Reports the current column number
    Column() int
}
```

The implementation of the CharSource interface is in the `github.com/obonobo/esac/core/chuggingcharsource` package. It is called ChuggingCharSource because it simply chugs the entire data source into memory all at once, storing it in a []byte (byte slice). It decodes bytes into UTF-8 codes whenever a char is requested. Finally, as mentioned above, it supports backing up a char any number of times (all the way to the beginning of input), and it contains one method for reporting line number and one method for reporting column number.

Next is the *Scanner* interface. This interface supplies a single method: *Scanner.NextToken()* which decodes the next token present in the char source. This interface can be reused for other implementations, though a table-driven scanner was chosen in this case.

```go
// The Scanner interface is used to represent a parser that tokenizes a
// character stream
type Scanner interface {
    // Extract the next token in the program. This function is called by the
    // syntactic analyzer
    NextToken() (Token, error)
}
```

The Token struct describes the objects that will be generated by the scanner:

```go
type Token struct {
    Id     Kind   // The unique identifier of this token
    Lexeme Lexeme // The exact string that was matched as this token
    Line   int    // The line number on which the token was found
    Column int    // The column number on which the token was found
}

func (t Token) String() string {
    return fmt.Sprintf(
        "Token[Id=%v, Lexeme=%v, Line=%v, Column=%v]",
        t.Id, t.Lexeme, t.Line, t.Column)
}
```

In this file (*token.go*), there are also several constant and utility methods for working with tokens. All the "kinds" of tokens that can be generated (e.g. ASSIGN, ARROW, EQ, PLUS, etc.) are constants in this file. These constant serve as unique IDs for every token generated.

To implement the *Scanner* interface, there are several options. I've opted to create a table-driven scanner, whose package is called: `github.com/obonobo/esac/core/tabledrivenscanner.`

To act of scanning via table was split into two tasks, each implemented by a different component. Firstly, the actual *NextToken* function + supporting methods are written on a new struct called *TableDrivenScanner*. This struct implements the *Scanner* interface seen earlier:

The *TableDrivenScanner* has two dependencies that it requires to complete its job: it needs a *scanner.CharSource*, and a *Table* instance (new interface). To ensure loose-coupling, both dependencies are specified by interfaces, not concrete types. Thus, the same *NextToken* algorithm can be reused with a different table. Indeed, the algorithm is quite generic and most changes to the scanner require changes to the implementation of the *Table* interface rather than changes to the *TableDrivenScanner* struct. A new *Table* could be created with completely different final states and the same *TableDrivenScanner* would be able to run its *NextToken* algorithm to generate an entirely different set of tokens.

To support scanning with use of a table, a new *Table* interface was created:

```go
// State transition table
type Table interface {
    // The initial state
    Initial() State

    // Perform a transition
    Next(state State, char rune) State

    // Generates a token given a State
    CreateToken(state State, lexeme scanner.Lexeme, line, col int) (scanner.Token, error)

    // Check if a state requires the scanner to backup
    NeedsBackup(state State) bool

    // Check if a state requires the scanner to backup TWICE
    NeedsDoubleBackup(state State) bool

    // Check if a state is a final state
    IsFinal(state State) bool

    // Checks if a symbol is whitespace
    IsWhiteSpace(char rune) bool
}
```

This interface is the largest yet. The reason why is that I wanted to abstract nearly all the token implementation bits into the *Table* component. By putting this logic in the *Table* rather than in the struct that implements the *Scanner*, the same *NextToken* algorithm can be reused for other tables.

The concrete implementation of the *Table* interface is the `github.com/obonobo/esac/core/tabledrivenscanner/compositetable`. This package contains 2 files: *composite_table.go* and *the_table.go*. The first file contains the struct that implements the *Table* interface, while the second file contains a factory function for creating the actual implementation instance of the table.

The reason why the struct is called *CompositeTable* is because the transition table has been flattened out into a map where the keys are composites of *State* (the current state that we are on) + *rune* (the next character in the input) and the values are *State* types (which is a type alias for *int*). To execute a transition, a *Key* is created containing both the current state as well as the next symbol in the input. The key is hashed into the table to produce the next state.

CompositeTable is the heaviest duty of the structs so far. It contains of course the transition table, but it also controls token creation from final states. Because the table component decides what *State* is next, it also implements the comment stack for processing imbricated comments. It decides what tokens are final, what the initial state is, which characters count as letters, which characters count as digits, as whitespace, etc.

Finally, the file named *the_table.go* contains the actual implementation table that is used by the app. This is basically a giant data structure that lists all the states, transitions, and what have you. This is the implementation that gets wired into the *TableDrivenScanner* component as the *Table* dependency.

4 – USE OF TOOLS

Here is a list of all the tools I used:

- *draw.io* for drawing the DFA
- *regex101.com* for testing out my regular expressions before drawing the DFA
- *vscode* as my text editor
- *golang stdlib* for the code, no external dependencies
- *GNU make* for some simple build commands
- *Github* for code repo + running my CI for testing as I developed

I didn't use any tools to generate the table, I just wrote it out by hand in my code file. For creating the DFA, I did Rabin-Scott powerset construction and those other algorithms by hand in *draw.io*. I did start out drawing the state machine in *JFLAP* (https://www.jflap.org/), but I got frustrated quickly because it doesn't have any copy/paste, so I switched to *draw.io*. I wrote almost all my testing as unit tests so that I could run them automated on GitHub actions as I was pushing commits directly to *main* branch.

My reasoning for this set of tools is that I really wanted to do everything as manually as possible so that I could get a good understanding of the process. I didn't want to use anything auto-generated so I drew the DFA by hand, designed and coded everything by hand, and I wrote tests as I went.