FORMAL LANGUAGES AND COMPILERS PROJECT

FIGMA MAGIC BANNER -

AN API JSON TO HTML TRANSPILER

Date: 15.06.2020

Academic Year: 2019/2020

Students:

De Candido Gabriele 16117

Palma Giada 15574

TABLE OF CONTENT

1. General explanation of the code

We developed a transpiler that converts a JSON file, exported through an API of the FIGMA software[1] into an HTML code snippet.

The FIGMA software lets you create templates for the design and styling of applications. Through an API the template data can be exported into JSON format (for more information see: https://www.figma.com/developers/api).

The JSON export is a file json composed of objects, arrays, and key-value pairs, in a nested way.

We implemented a lex file called *json.l*, which generates a Lexer. This file is a tokenizing program able to read the input and convert strings in the source to tokens. With regular expressions we have specified patterns to lex, so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to lex has an associated action returning a token that represents the matched string for subsequent use by the *json.y* file (the parser).

The lex file is divided into 3 sections:

- The **definition section**

  Here we specify with the *%option* to change the prefix of the yy.lex.c output file into *json.lex.c.*

  Then we include libraries in the block {% … %} and define regular expressions for recognition of numbers, strings and booleans.

  %%

- The **rules section**

  Here we define rules that, if matched in the input file, execute the associated C code. Namely, if a STRING is matched it will call the *strndup( )* function, which returns a pointer to the duplicate of the matched string. Otherwise, if a NUMBER or a BOOLEAN is matched it will call similarly as above a *strdup( )* function, which returns a pointer to a copy of the matched number or boolean.

  Then curly and square brackets, commas and columns can be matched and their associated token constant will be returned.

  White spaces, tabs and newlines will be ignored.

  If some input is not matched, the lexical analyzer will return an error and exit the program.

  %%

[1] see: https://www.figma.com/

- The **subroutines**

  In our lexer we do not specify any subroutine, because we define the actions to be taken when an input is matched in the yacc file.

We implemented a file called *json.y,* which analyses a series of tokens with their values. Similarly to the lex file the yacc is divided into 3 sections:

- The **definition section**

  Here we include C libraries and the json.lex.c file.

  Then we define all the types that will populate our symbol table.

  And we implement a struct, which creates a hash map composed by key-value pairs, where the keys are json key strings, contained in the input json file, and the value is the numeric value, defined by the constant types above.

  Finally, we define 4 explicit functions that decode key-value pairs, objects, arrays and tags for the target html language. And a *generatedOutput* variable to pass the values.

  %%

- The **rules section**

  Here we define the grammar, the left-hand side of a production is a nonterminal symbol followed by a colon. The right-hand side are actions associated with a rule entered in curly braces. Our grammar is able to recognise:

  - objects in the following form { member }
  - members composed by pairs of key-value followed by a comma
  - pairs of the form key:value, where value can be a string, a number, a boolean, an object or an array
  - arrays in the form [elements]
  - elements which can be string, number, boolean or again an object followed by a comma, or an array followed by a comma
  - values, which as stated above, can be a string, a boolean, or a number

  whenever a nonterminal is recognised and an action is executed, some memory is allocated in order to store a string, containing the value associated to the non terminal.

  %%

- The **subroutines**

  In this section the core of the program is implemented, namely the *main()* method used to inform the user if the program execution succeeded or if some errors occurred. If everything is fine a file with .html extension will be generated, and can be opened with a web browser to see the result.
  In this section, we also implemented the *lookup()* function, used to search for the keys in the symboltable and returned the associated constant.
  The *decodeKV()* function makes a call to the *lookup()* function and returns accordingly the generated value (i.e html tags or css keys) that has to be printed in the final result. The same works for the *decodeObj()*, *decodeArray()* and *checkDIV()* functions.

2. Grammar of the language

The grammar of the language can be found in the json.output file with all the terminals and non terminals symbols and the state machine of the parser.
The grammar is the following:

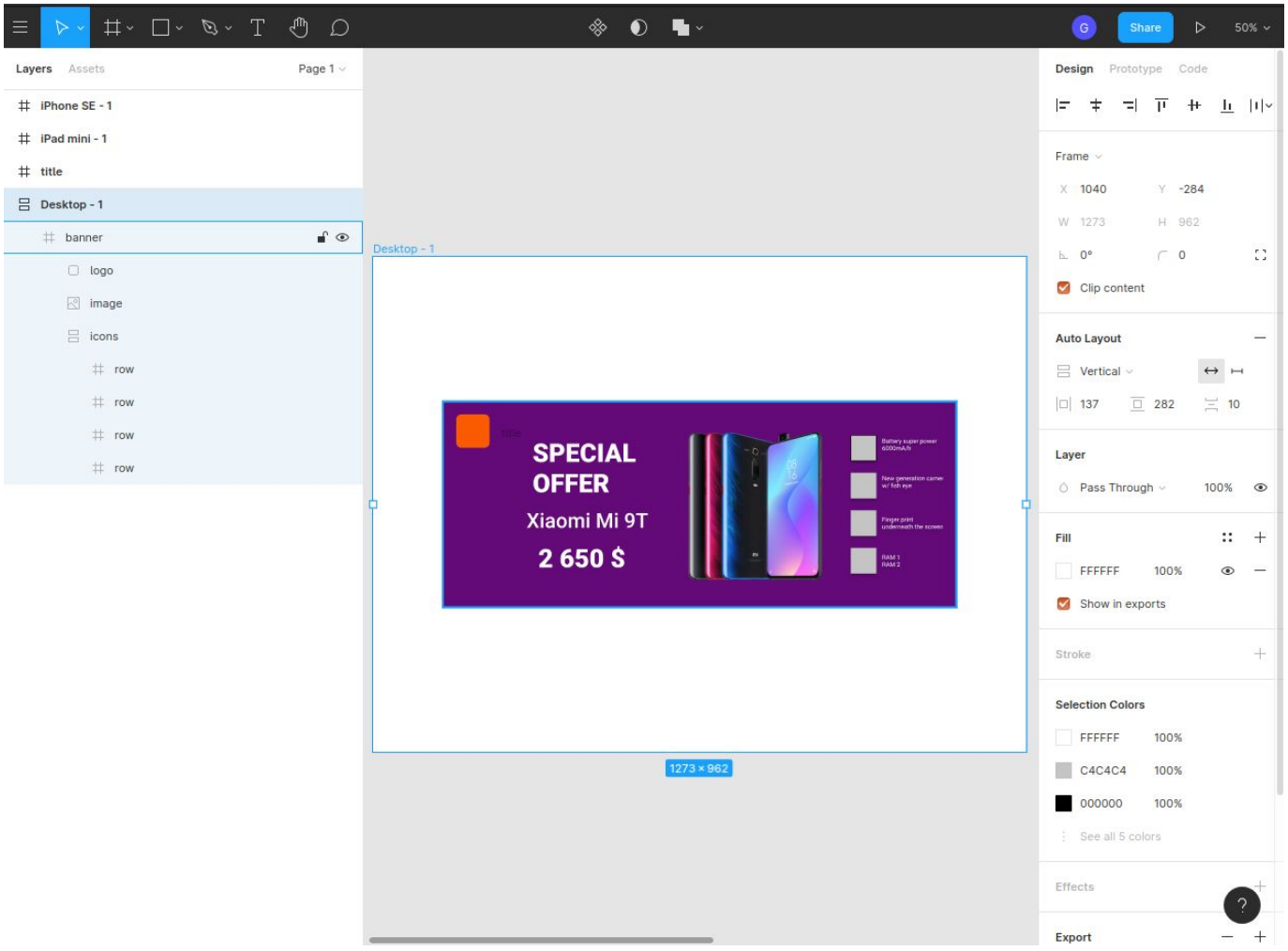
Grammar

```
0 $accept: start $end

1 start: object

2 object: OBJECT_BEGIN members OBJECT_END

3 members: pair
4        | pair COMMA members

5 pair: STRING COLON value
6     | STRING COLON object
7     | STRING COLON array

8 array: ARRAY_BEGIN elements ARRAY_END

9 elements: value
10         | object
11         | array
12         | value COMMA elements
13         | object COMMA elements
14         | array COMMA elements

15 value: STRING
16      | NUMBER
17      | BOOLEAN
```

3. Description of the input

The input is a file json, exported with an API from the FIGMA software used to design templates for applications.

The following is a snippet of the FIGMA software where we have designed the template for a web banner.



The exported file has a well defined structure, so that programmers can easily understand what the tags and properties of the html and css files for developing a website should be. As an input file however, we use a cleaned-up form of the exported json, containing just what we needed to show the banner. (i.e file in github repository called /files/xiaomi-banner.json)


```
{
    "name": "banner",
    "style": {
        "display": "flex",
        "justify-content": "space-between",
        "font-family": "Roboto",
        "absoluteBoundingBox": {
            "height": 398.0,
            "width": 999.0,
            "top": 282.0,
            "left": 137.0
        },
        "fill": {
            "type": "solid",
            "background-color": "#630978"
        }
    },
    "characters": "",
    "children": [
        {
            "name": "logo",
            "style": {
                "absoluteBoundingBox": {
                    "height": 65.0,
                    "width": 65.0,
                    "top": 25.0,
                    "left": 25.0
                },
```


4. Instructions to run the code

Instructions to run the code can be found in the README.md of our github repository.
https://github.com/GiadaPa/Formal-Languages-and-Compiler

However, they can be easily explained here:
To run the code just download the project folder from the given repository.
Open the terminal in the just downloaded folder and run the following command *make all JSON="files/xiaomi-banner"*.
The output html file can be found in the files folder, named the same as the json input file, i.e *xiaomi-banner.html*