Write your own compiler - Station #1: the tokenizer

Feb 8, 2017 • Yehonathan Sharvit

The plan

Our journey is made of 4 stations - each of them depending on the previous ones:

1. The tokenizer (aka “Lexical Analysis”): converting an input code - in `LISP` syntax - into an array of tokens.
2. The parser (aka “Syntactic Analysis”): transforming an array of tokens into an Abstract Syntax Tree (AST).
3. The emitter (aka “Code Generation”): string-ifying an AST into `C`-like code.
4. The compiler (aka “You made it”): combining all the pieces together.

(The interactive code snippets are powered by a tool of mine named KLIPSE.)

The tokenizer

The `tokenizer` receives a string of code and breaks it down into an array of tokens.

There are three kinds of tokens:

1. single-character token: `(` and `)`
2. multiple character token: `123` or `abcd`
3. a string: something that starts with a `"` and ends with a `"` (no escaping!)

First, we are going to write a couple of tokenizers for a single token. Each tokenizer receives the code as a string and the current position and returns:

1. the length of the token
2. the token as an object with two keys: `type` and `value`

Single-character token

Let's write a generic function that tokenizes a single character:

```
tokenizeCharacter = (type, value, input, current) ⇒
(value === input[current]) ? [1, {type, value}] : [0, null]
```

```
[Function tokenizeCharacter]
```

Here is the tokenizer for `(`:

```
tokenizeParenOpen = (input, current) ⇒ tokenizeCharacter('paren',
'(', input, current)
```

```
[Function tokenizeParenOpen]
```

```
tokenizeParenOpen('(', 0)
```

```
Array [
  1,
  Object {
    "type": "paren",
    "value": "(",
  },
]
```

And here is the tokenizer for `)`:

```
tokenizeParenClose = (input, current) ⇒ tokenizeCharacter('paren',
')', input, current)
```

```
[Function tokenizeParenClose]
```

```
tokenizeParenClose(')', 0)
```

```
Array [
  1,
  Object {
    "type": "paren",
    "value": ")",
  },
]
```

Multiple character tokens:

We will describe our multi-character token by means of regular expressions:

Here is a generic regexp tokenizer:

```
tokenizePattern = (type, pattern, input, current) ⇒ {
  let char = input[current];
  let consumedChars = 0;
  if (pattern.test(char)) {
    let value = '';
    while (char && pattern.test(char)) {
      value += char;
      consumedChars ++;
      char = input[current + consumedChars];
    }
    return [consumedChars , { type, value }];
  }
  return [0, null]
```

```
}
```

```
[Function tokenizePattern]
```

And here is the `number` tokenizer:

```
tokenizeNumber = (input, current) ⇒ tokenizePattern("number", /[0-9]/, input, current)
```

```
[Function tokenizeNumber]
```

```
tokenizeNumber("123aad", 0)
```

```
Array [
  3,
  Object {
    "type": "number",
    "value": "123",
  },
]
```

And the `name` tokenizer (in our language names are chains of letters):

```
tokenizeName = (input, current) ⇒ tokenizePattern("name", /[a-z]/i, input, current)
```

```
[Function tokenizeName]
```

```
tokenizeName('hello world', 0)
```

```
Array [
  5,
  Object {
    "type": "name",
    "value": "hello",
  },
]
```

String tokenizer

A string is something that starts with a `"` and ends with a `"` (no escaping in our language!):

```
tokenizeString = (input, current) ⇒ {
  if (input[current] === '"') {
```

```
    let value = '';
    let consumedChars = 0;
    consumedChars ++;
    char = input[current + consumedChars];
    while (char !== '"') {
      if(char === undefined) {
        throw new TypeError("unterminated string ");
      }
      value += char;
      consumedChars ++;
      char = input[current + consumedChars];
    }
    return [consumedChars + 1, { type: 'string', value }];
  }
  return [0, null]
}
```

```
[Function tokenizeString]
```

```
tokenizeString('"Hello World"', 0)
```

```
Array [
  13,
  Object {
    "type": "string",
    "value": "Hello World",
  },
]
```

Last thing, we want to skip whitespaces:

```
skipWhiteSpace = (input, current) =>   (/\s/.test(input[current])) ? [1, null] : [0, null]
```

```
[Function skipWhiteSpace]
```

The tokenizer

Let's put all our tokenizers into an array:

```
tokenizers = [skipWhiteSpace, tokenizeParenOpen, tokenizeParenClose, tokenizeString, tokenizeNumber, tokenizeName];
```

```
Array [
  [Function skipWhiteSpace],
  [Function tokenizeParenOpen],
```

```
  [Function tokenizeParenClose],
  [Function tokenizeString],
  [Function tokenizeNumber],
  [Function tokenizeName],
]
```

The code tokenizer is going go over its input and try all the tokenizers and when it finds a match it will:

1. push the token object
2. update the current position

Here is the code:

```
tokenizer = (input) => {
  let current = 0;
  let tokens = [];
  while (current < input.length) {
    let tokenized = false;
    tokenizers.forEach(tokenizer_fn => {
      if (tokenized) {return;}
      let [consumedChars, token] = tokenizer_fn(input, current);
      if(consumedChars !== 0) {
        tokenized = true;
        current += consumedChars;
      }
      if(token) {
        tokens.push(token);
      }
    });
    if (!tokenized) {
      throw new TypeError('I dont know what this character is: ' +
char);
    }
  }
  return tokens;
}
```

```
[Function tokenizer]
```

Let's see our tokenizer in action:

```
tokenizer('(add 2 3)')
```

```
Array [
  Object {
    "type": "paren",
```

```
    "value": "(",
  },
  Object {
    "type": "name",
    "value": "add",
  },
  Object {
    "type": "number",
    "value": "2",
  },
  Object {
    "type": "number",
    "value": "3",
  },
  Object {
    "type": "paren",
    "value": ")",
  },
]
```

Our tokenizer doesn't do any semantic validation. As an example, it can read unbalanced parenthesis:

```
tokenizer('(add 2')
```

```
Array [
  Object {
    "type": "paren",
    "value": "(",
  },
  Object {
    "type": "name",
    "value": "add",
  },
  Object {
    "type": "number",
    "value": "2",
  },
]
```

Let's make sure we can handle nested expressions properly:

```
tokenizer('(add 2 (subtract "314" 2))')
```

```
Array [
  Object {
    "type": "paren",
```

```
    "value": "(",
  },
  Object {
    "type": "name",
    "value": "add",
  },
  Object {
    "type": "number",
    "value": "2",
  },
  Object {
    "type": "paren",
    "value": "(",
  },
  Object {
    "type": "name",
    "value": "subtract",
  },
  Object {
    "type": "string",
    "value": "314",
  },
  Object {
    "type": "number",
    "value": "2",
  },
  Object {
    "type": "paren",
    "value": ")",
  },
  Object {
    "type": "paren",
    "value": ")",
  },
]
```

Hourra!!!

Please take a short rest before moving towards Station #2: The parser.

If you enjoy this kind of interactive articles would you consider a (small) donation on Patreon or at least giving a star for the Klispe repo on Github?