Chris Culling

Lab for **assignment 6**

15 Nov 2024

Notes

1. Parsing function per nonterminal
2. Peek-function: identify *statments* and *expressions* from tokens before parsing
3. Consume-function to iterate past no-longer-relevant tokens

Variable names + values = key-value pairs in hash table (symbol table, managed by `AssgStmt`). I used an *unordered_map*.

1. Have file with C@ source code
2. function takes filename as input

- getline for each line
- split each token in each line in and store all those tokens in a vector
- send them to intepreter that interprets one vector at a time

Defining `parse_Statement`

In the beginning, I didn't really see the use in the consume function. My first intuition was to just consume all the tokens in a line after evaluating the next line:

```
void Interpreter::evaluate(const std::vector<std::string>& tokens)
{
    parse_Statement();

    for(i = position; i < tokens.size(); i++)
        consume(tokens.at(i));
}
```

Once I understood the intended implementation and structure of the assignment, things became somewhat straightforward.

I first implemented `parse_Statement` along with three dummy functions `parse_ConfigStatement`, `parse_PrintStatement`, and `parse_AssignStatement` which simply printed to `cout` (without actually making any changes):

CODE

```
void Interpreter::parse_Statement()
{
    string next_token = peek();

    if(next_token == "config")
        parse_ConfigStatement();
    else if(next_token == "print")
        parse_PrintStatement();
    else
        parse_AssignStatement();
}
```

SOURCE

```
config dec
config hex
config bin
print 6
print 2 + 1
print 1 + 5 * 2
test = 1
test2 = 2
```

OUTPUT

```
config parsed
config parsed
config parsed
print parsed
print parsed
print parsed
assign parsed
assign parsed
assign parsed
```

Defining Math Expressions and `parse_PrintStatement`

Then I moved on to implementing all the mathematical expressions. Since both `parse_AssignStatement` and `parse_PrintStatement` required a definition for `MathExpression`, which in turn requires definitions for `SumExpression`, `ProductExpression`, and `PrimaryExpression`, I started with defining `PrimaryExpression`.

Outside consuming at the correct time, all there was to it was defining `Int` and `Variable` as per assignment instructions.

```
bool isInt = regex_match(next_token, regex("-?[0-9]+"));
bool isVariable = regex_match(next_token, regex("[a-zA-Z][a-zA-Z0-9]*"));
```

`SumExpression` and `ProductExpression` looked almost identical.

```
int result = parse_ProductExpression();

string next_token = peek();

while(1)
{
    if(next_token == "+")
    {
        consume("+");
        int newTerm = parse_ProductExpression();
        result = result + newTerm;
    }

// ... and so on

return result;
```

And `MathExpression` was trivial.

```
int Interpreter::parse_MathExpression() { return parse_SumExpression(); }
```

With all these in place, we just needed to make `parse_PrintStatement` actually write to the outstream.

CODE

```
void Interpreter::parse_PrintStatement()
{
    consume("print");

    out_stream << parse_MathExpression() << endl;
}
```

SOURCE

```
config dec
config hex
config bin
print 6
print 2 + 1
print 1 + 5 * 2
test = 1
test2 = 2
```

OUTPUT

```
config set to binary
config set to hexadecimal
config set to decimal
6
3
11
```

Defining Variables and `parse_AssignStatement`

Outside checking for tokens, it was fairly trivial to implement the `parse_AssignStatement` function. I decided to use an `unorderedmap<string, int> variables` to keep track of the variables.

```
// inside parse_AssignStatement
variables[name] = value;
```

Then I just needed to update the code in `parse_PrimaryExpression`:

```
else if(isVariable)
{
    consume(next_token);
    if(variables.find(next_token) == variables.end())
        throw runtime_error("Undefined variable\n");

    value = variables[next_token];
}
```

Now, after updating `source.txt` we had the following success.

SOURCE

```
config dec
config hex
config bin
print 6
print 2 + 1
print 1 + 5 * 2
x = 2 - -2
y = x
```

```
z = y * ( 16 / ( y - 2 ) )
print x
print y
print z
```

OUTPUT

```
config set to decimal
config set to binary
config set to hexadecimal
6
3
11
4
4
32
```

(Note that `config set to` lines still do not actually do anything.)

Implementing `parse_ConfigStatement`

To get `parse_ConfigStatement` to work as intended I simply added an `enum Config { dec, hex, binary }`, and made sure it updates in the `parse_ConfigStatement` function.

```
// inside parse_ConfigStatement
if(next_token == "dec")
{
    config = Config::dec;
    cout << "decimal" << endl;
}
else if(next_token == "hex")
{
    config = Config::hex;
    cout << "hexadecimal" << endl;
}
else if(next_token == "bin")
{
    config = Config::binary;
    cout << "binary" << endl;
}
// etc...
```

And updated print to depend on the `Config` enum.

```
switch (config)
{
```

```
case Config::binary:
    out_stream << bitset<16>(parse_MathExpression()).to_string();
    break;
case Config::hex:
    out_stream << std::hex << showbase << parse_MathExpression();
    break;
default:
    out_stream << parse_MathExpression();
    break;
}
```

Finally, our test on `example-source.txt` was totally successful!

EXAMPLE-SOURCE

```
config dec
print 1 + 1
print 3 + 3 * 3
print ( 3 + 3 ) * 3
x = 2 - -2
y = x
z = y * ( 16 / ( y - 2 ) )
print x
print y
print z
config hex
print z
config bin
print z
```

OUTPUT

```
config set to decimal
2
12
18
4
4
32
config set to hexadecimal
0x20
config set to binary
0000000000100000
```

Finally, I made sure to check that error cases such as `print 1 + - 2` actually throws the correct `runtime_error`, which it does!