TSC – The Small Compiler

Introduction

TSC is a small compiler that is based on LISP like coding, where every expression you give has to be enclosed in parenthesis – or round brackets – unless they are simple integers. You can also add strings – which was applied to the compiler to give options for Boolean expression and strings and identifiers, but identifiers weren't included in the compiler, because I couldn't get the compiler to run infinitely (until an EOF statement was included). And unless the compiler runs infinitely, identifiers would serve no purpose. Thus, you could call this compiler, a single-line interpreter.

I had worked on identifiers originally, including nearly all necessary ingredients of a compiler (I named that compiler VENUS) – but I had made the mistake of doing test files on the original version, and applying new arguments to the compiler step-by-step. Thus, after working on it until the eighth of January, I decided to start from the simple compiler that was discussed in class, and growing it based on my original vision of the compiler.

TSC will run both arithmetic and conditional expressions, all merged into one single statement that would answer the operation based on the first operator included in the statement. Strings will raise an error, unless they are the only expression in the statement, or are avoided from arithmetic operations, but the only purpose of them are to return and print itself. The Output or Error will only be printed after the final closing bracket.

The Scanner

The scanner includes all the tokens that are applied to TSC. There are no comments included, as TSC is only a single-line interpreter. The Scanner also serves the purpose of removing whitespaces, new-lines, and all other separators. All this is included in the file: "scanner.mll".

The tokens included in TSC are:

- Parenthesis:
  - (LEFT PARENTHESIS
  -) RIGHT PARENTHESIS
- Basic arithmetic operators :
  - + ADD
  - – SUBTRACT
  - * MULTIPLY
  - / DIVIDE
- Basic conditional operators :
  - == EQUALITY
  - > GREATER THAN
  - < LESS THAN
- Conditional check:

    - if
- Integer literals
- Strings (mainly to include Boolean operations, but basic string printing is also allowed)

The AST

The AST basically creates a tree – or web – of how the tokens would be handled. TSC has 2 types: one to include all the operators, and one to show how expressions creates the web.

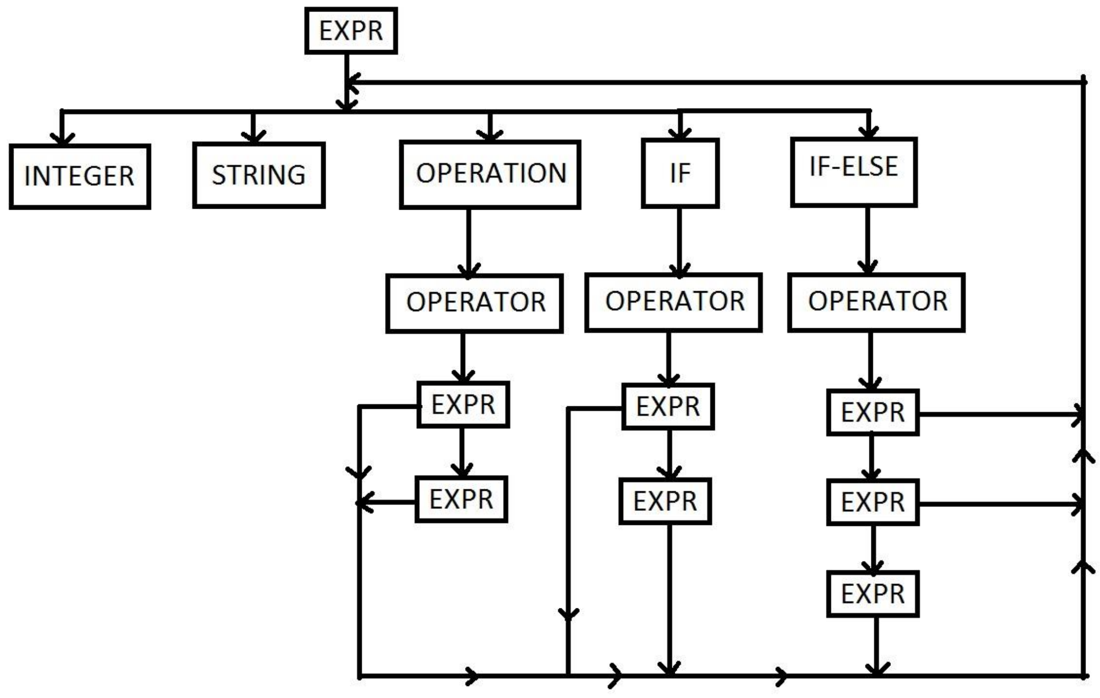
Operators are all the operators mentioned in the scanner (Add, Sub, Mul, Div, Equal, Less, Greater).

The Expression (expr) shows how the web is created: The Primary statement/expression expands to either an integer literal, a string, an operation based on all the operators defined, and If, or and if-else statement.

The operation would itself, expand to an arithmetic operator or a Boolean operation, and 2 expressions referring to the start of the web.

Same way, the If and the If-else statements expands to a boolean token (true or false), and 2 or 3 expressions which – again refers to the start of the web.

The file used is "ast.ml".



The Parser

The parser makes sure that all the proper semantic rules defined for TSC are expressed in readable format, based on the tree that gives information about how the tokens in TSC are used to create proper expressions. The parser makes sure that all expressions are inserted inside curly brackets, and basically how the AST should be analyzed.

The rules created are based on the format for the single-line-expressions of TSC, and are sent to the interpreter which will analyze the information and apply it to the tokens extracted based on the scanner, from the input that the user gives.

The file used is “parser.mly”.

The Interpreter

The interpreter will use all the information that the parser generates, and compile it by applying the syntax of the Ast tree, on the Ocaml compiler.

Using the information, the evaluation function will return a string. It evaluates the input as follows:

- If the expression being evaluated a simple string, then the function leaves it just like that and returns it.
- If the expression is an operation, then it works on the operator as follows:
    - If the operator is a boolean expression, then the function will convert it to 0 or 1 based on if it’s “false” or “true”.
    - If the operator is a string integer, the function will convert it to an integer.
    - In any other case, an error would be raised.

  The expression is then evaluated by the OCaml compiler based on the operator.
- If the expression is an “If” operation, then it works as follows: It first evaluates the second expression, and converts the result from string to bool. It raises an error if the string wasn’t a bool string. It then runs the third expression if the second expression was “true”, and returns “false” otherwise.
- If the expression is an “If-Else” operation, then it works as follows: It first evaluates the second expression, and converts the result from string to bool. It raises an error if the string wasn’t a bool string. It then runs the third expression if the second expression was “true”, and the only difference from “if” is, it runs the fourth expression otherwise instead of just returning “false”.

One important point about the “If” and “If-else” operation is, they both use the same token “if”, and runs based on the number of expressions/arguments following them.

Examples:

- (+ 2 5)
- (= 4 4)
- abcd
- (if (= 4 4) 9)

- (if (= 4 4) 9 10)
- (if (> 1 4) 9 10)
- (* (if (< 3 2) 2) 7)
- (* (if (> 3 2) 2) 7)

My Original Vision / Conclusion

I wanted to include float literals, negative values in integers/float values, identifiers, loops, an interpreter+compiler, and even a few in-built functions to work on graphs that would even generate creating maps, agents, and AI solvers. I tried that on VENUS, but because of not able to sync them, I had to create TSC.