Assignment 2: Generating Intermediate Representations

Compiler Construction (Fall '24)

Hand-In Date: Nov, 27

In this assignment you should practice and demonstrate that you can generate an intermediate representation (**IR**) given an AST. Given an IR it is possible to optimize the original source code to eventually execute it faster on the target machine. Furthermore, given a certain IR, it requires only another translation step to generate machine code.

Similar to programming languages, IR come at different levels of abstractions and may be high-level (closer to the source code) or low-level (closer to the eventual machine code, e.g. byte code of virtual machines). In this task, you should implement a code generator that produces **three-address code** (**TAC**). In a second transformation step, the IR could be further converted into a low-level IR or already in an machine code instruction set (but you are not supposed to perform that second step).

1. Implementing an IR Code Generator for SPL

We assume that we still have the structured programming language SPL (cf. Listing 1) which is inspired by Lox and does not include handling of functions.

(a) Extend the given SPL compiler front-end to emit correctly ordered TAC.

To keep the task feasible, we assume

- the SPL code has been parsed correctly and does not contain lexical or syntactic errors.
- the source code is semantically correct (no scope or type checks needed).
- `true` and `false` are not used as literals. (They do not exist in TAC.)

You should translate the following grammar elements

- Variable declarations, Literals, Arithmetic and Logical Expressions,
- If- and While- Statements to represent the control flow.
- blocks only if they represent an if-branch or the body of a while-loop

You do **not** have to translate

- groupings of expressions
- print statements
- functions, function calls, etc (which are actually not even part of the grammar)

Given this information, your task is it to implement a TAC generator which

- receives a list of correctly parsed SPL statements.
- visits each statement to emit the corresponding instructions in **TAC**
- writes the generated three-address instructions in the correct order into a single `text-file` for each given program file.

TAC is not standardized. We assume a similar format as it is described at Standford university. At the end of the linked document, you can find several examples of how high-level code translates to TAC. Key properties of TAC are as follows:

- TAC can directly use variables (i.e., their names)
- TAC can use a set of temporary variables (ensure to restrict their number!)
- Constants (i.e., numbers and Strings) have to be loaded into temporary variables to be used in expressions.
- `true` and `false` do not exist as Boolean literals
- the available arithmetic operators are `+`, `-`, `*`, `/`, `%`
- the available comparison operators are `==`, `<`, `&&`, `||`
- The keyword `IfZ` checks whether a conditions evaluates to true.

- TAC can use a set of labels. By convention they are numbered and start with `"_"`, e.g., `_L1:`, `_L2:`, etc.

You are given a template which includes a scanner (lexer) and a parser without error handling. You do not have to change them but only should implement the TACGenerator. It store the eventual **TAC in a text file**.

Some recommendations for the implementation:

- The names of temporaries and labels should start with an `"_"` (e.g., `"_t1"`) to easily distinguish them from variables.
- Ensure that temporaries are released when they are not in use anymore to be used in later instructions
- Start simple: transform first literals and variables, continue with simple binary expressions, etc.
- the grammar (and thus, the AST) already respects the precedence of operators, you just need to take care of ordering the temporary computations correctly.

(b) Add a `readme.md` file to your implementation. It should explain how you performed the TAC emission. Furthermore, in the readme, describe in a few sentences how you (could) optimize your generated three-address code?

Hand-in rules

By November 27, hand-in

- a zip-file of your source code which should be an executable Maven project.
- particularly the zipfiel should include the TACGenerator and a Readme explaining the implementation and potential optimizations.

in the respective assignment on ITSLearning.

This assignment must be passed to successfully complete this course.

```
program         := declaration* EOF ;

declaration     := varDecl | statement ;

varDecl         := "var" IDENTIFIER ( "=" expression )? ";" ;

statement       := exprStmt
             | whileStmt
             | ifStmt
             | printStmt
             | block ;

exprStmt        := expression ";" ;

ifStmt          := "if" "(" expression ")" statement ( "else" statement )? ;

printStmt       := "print" expression ";" ;

whileStmt       := "while" "(" expression ")" statement ;

block           := "{" declaration* "}" ;

expression      := assignment ;

assignment      := IDENTIFIER "=" assignment
| logic_or ;

logic_or        := logic_and ( "or" logic_and )* ;

logic_and       := equality ( "and" equality )* ;

equality        := comparison ( ( "!=" | "==" ) comparison )* ;

comparison      := term ( ( ">" | ">=" | "<" | "<=" ) term )* ;

term            := factor ( ( "-" | "+" ) factor )* ;

factor          := unary ( ( "/" | "*" ) unary )* ;

unary           := ( "!" | "-" ) unary | primary;

primary         := "true" | "false" | NUMBER | STRING | IDENTIFIER
                 | "(" expression ")";
```

Listing 1: "A grammar for SPL"