Exercise 5 documentation

General flow

Our code is run by the class Compiler.java which holds the main function.

We receive as arguments a file path leading to an IC program, and optionals such as a path to a Library file (-L*filepath*), a choice to print the AST to System.out (-print-ast), a choice to print the symbol+type tables to System.out (-dump-symtab), and a choise to print the lir representation of the code (-print-lir).

If the user requested a Library file in his arguments, we send the file to its respective lexer and then its parser.

Next, we send the main program file to the lexer and then to its parser. If a library was supplied, we make sure that its class is named “Library”, and if so append the Library class to the program’s class list.

Now that we have the initial AST structure for the program, we run it through a type-table builder. The type-table builder is a visitor that visits every node in the tree and builds the program’s type-table that we will later use.

After we have built the type table, we run the AST through the symbols-table builder which again, visits every node in the AST and builds a symbols table tree in which every scope has its appropriate symbols-table which is linked to the symbols table of its parent and children.

Then, we run the AST through a type validator that validates all typing rules are correct; and print the results according to the user’s arguments.

Lastly, we send the AST Program root to the TranslationVisitor, which traverses the AST and converts it to a list of lir instructions.

Everything is done within a try-catch scope that catches all lexical, syntax, semantic and general errors.

Class hierarchy

- IC – Package for the main class (Compiler) and misc.
    - Compiler – the class that contains the main function and controls the flow of the compiler program
- IC.AST – Package containing all AST construction nodes.
    - ASTNode – main node class. All other AST nodes implement this class directly or indirectly.
    - PrettyPrinter – a visitor that traverses the AST and prints it in readable format.
- IC.Parser – Package containing all lexering + parsing logic.
    - Lexer – the lexer for the main IC program.
    - LibLexer – the lexer for the library signature file.
    - Parser – the parser for the main IC program.
    - LibParser – the parser for the library signature file.
- IC.SemanticAnalysis – Package for semantic error handling.
- IC.SymbolsTable – Package for all symbol-table building logic.
    - SymbolTable – the data type that represents a symbol table.

    - SymbolsTableBuilder – the visitor that builds the symbols table tree.
- IC.Types – Package for all type-table building logic.
    - TypeTable – the data type representing a type table.
    - TypeTableBuilder – the visitor that builds the type table.

- IC.lir – Package containing all lir translating logic.
    - ClassLayout – Hold a mapping between methods and fields of a class and their offsets.
    - StringLiterals – Holds all literal strings encountered in the program, and can produce a list of StringLiteral object which can be printed at the beginning of the lir program.
    - TranslationVisitor – The visitor that converts the AST into a list of lir instructions.
- IC.lir.Instructions – Package containing all lir Instruction classes.
- IC.lir.CodeGeneration – Package containing all the code generation phase classes.
    - CodeGenerator – A lir intstructions Visitor which get a set of lir instructions, and translates them into an assembly code.
    - AssemblyMethod – A data struct which contains all the relevant information for constructing a method from lir to assembly code. It includes list of the parameter, variables and the amount of virtual lir registers that were used in the method lir implementation. Through this information we can calculate the stack frame size of the function and to determine where to locate each variable, parameter and virtual register inside the stack.

Code Generation implementation

The code generation from lir to assembly is implemented in the lir visitor CodeGenerator.

It receives all the information that was gathered in the lir translation phase and builds from it a single string with the new assembly code. This code will be written afterwards by the Compiler main class into a new .s file.

The lir information includes:

- A list of all the lir instructions.
- A map of all the methods and their relevant information stored in objects of type AssemblyMethod.
- All the string literals
- All the classes and their dv table information stored in a collection of objects of type ClassLayout

The primary function here is generateCode which run the visitor on the lir instructions list and returns the new assembly code in a string.

Important notes:

- For each method, all the virtual lir registers are also stored on the stack along with the method's variables. This allow as to refer all the actual registers as temporaries and not to worry about their previous values in each lir instruction generation. The stack is built from all the virtual lir registers first and then the method's variables.
- The AssemblyMethod class helps to provide all the necessary information about the method for creating a proper stack frame and get stack frame references to all the virtual

lir registers and variables. The data of each method is filled into an AssemblyMethod object in the TranslationVisitor which we create for translating the AST into lir set of instructions in PA04. Then, all the methods new objects are passed to the CodeGenerator.

- The length instruction code generation takes the '-1 byte' of an array, divides its value by 4 and return the result as the array's length. This logic assumes each array cell is taking 4 bytes on the heap. This is almost true except of one exception: when calling the Library function stoa, a new array is returned where each byte represents a character from the input string. Therefore, each array cell in the output is actually taking exactly one byte on the heap. To keep the logic consistent with this specific case, after a stoa call was executed, we set on purpose the -1 byte of the output to be multiplied by 4. Therefore, it doesn't represent the array length value but the value times 4. This change can be problematic only when calling the opposite Library function atos: in this case, before the call, we restore the -1 byte to its original value (dividing by 4) and after the atos call was executed, change its value back to its artificial form (multiply again by 4).