From: Vandana Agarwal . (via Nalanda)
Sent: 28 March 2023 16:36
To: RAMAKANT PANDURANG TALANKAR .
Subject: 11551001318_CS F363_JAN_2023: Stage 2: Implementation updates

11551001318_CS F363_JAN_2023 » Forums » Announcements » Stage 2: Implementation updates

V Stage 2: Implementation updates
by Vandana Agarwal . - Tuesday, 28 March 2023, 4:17 PM

Function Prototypes and File Names

Function prototypes are flexible. Students are advised to use names of data structures such as ast, parseTree, symbolTable etc. appropriately. You can select names of implementation files appropriately from file names ast.c, symbolTable.c, typeExtractor.c, semantics.c, codegen.c etc. You can have additional files, if you need as support, but the name of the file must be indicative of the contents within it.

The function prototype declarations should be in file *.h corresponding to the implementation file name [For example , if you are naming the interface file for the functions in symbolTable.c, name that interface file as symbolTable.h]. The data definitions should also be split in the files appropriately as symbolTableDef.h. etc.

Intermediate Code Generation

Teams can generate assembly language code by first constructing the intermediate code using instructions given in text book or can generate the code directly. [The process of code generation through intermediate code generation is more systematic and produces correct code while direct code generation may be erroneous.] However, there is no extra credit for IR (Intermediate Representation) creation as it is left to the decision of teams as to whether to generate code through IR or by skipping IR . The correctness of the generated code will be of significance.

Instruction Set

Your compiler must generate equivalent assembly code (in file code.asm) with instructions taken from instruction set of the NASM simulator (linux based). Download NASM (Version 2.16.01)/ from https://www.nasm.us/ to verify correctness of the output obtained by executing assembly code generated by your compiler for the user source code in given toy language. Make your code generator module to emit 64-bit code in assembly language for the given test case. As has been mentioned earlier that code generator is not expected to implement any optimizations except for any trivial ones in regard to register assignment or instruction selection etc. Use simple templates for mapping the IR code to equivalent assembly code as was discussed today in the class.

Efficiency

Efficiency (Time and Space) is an expected feature of your compiler code. Design efficient data structure for symbol table etc. Abstract Syntax Tree (AST) is a copy of the user source code in concrete form and all other work including type checking, symbol table, semantic analysis and code generation etc. are expected to be done by traversing the AST only instead of traversing the parse tree. While constructing AST from parse tree, the unused nodes of the parse tree should be freed. The semantic analysis, type checking, symbol table construction and code generation rules should be based on the constructs (sub trees) of the AST.

Compatibility with the Ubuntu and GCC versions specified during stage 1 must be ensured.

See this post in context
