Project: Core Compiler Instructions

Course: CS 5363
Instructor: Xiaoyin Wang
Semester: Fall 2015

Overview

For the core compiler project, you will complete a non-optimizing compiler for the TL language as described by the TL 16.0 Specification. The resulting compiler should translate TL source code, producing output at several intermediate stages, into MIPS assembly code executable in a simulator. Note that this documented is revised from the previous CS5363 document generated by Jeff Von Ronne.

Make sure you pay attention to the requirements listed under Input/Output specification below. Submitted programs must comply with those specifications to receive full credit.

Students can form teams of 2 to work on the project, and you can write you code in Python3, Java, or C/C++, but you need to make sure your project compiles and runs well in the computers in the lab.

Scanner

The first deliverable is a Scanner for the TL language. A BNF grammar and a list of lexical items for the TL language is provided. The output of the scanner will be a list of tokens.

In the output file, tokens should be listed in sequence, one token each line.
All the tokens should be in the form as they are defined in the BNF grammar.

For tokens with attributes or multiple values, please print the tokens in the form of: TOKEN(value). For example, the output token stream of
*while SQRT * SQRT <= N do*
is:
WHILE
ident(SQRT)
MULTIPLICATIVE(*)

ident(SQRT)
COMPARE(<=)
ident(N)
DO

Parser

You are required to write, "by hand," a parser for it. After the parser tree is generated, or while the parser tree is being generated, you are required to remove non-essential parts of the parser tree to reduce it to an abstract syntax tree (AST).

For a lexically or syntactically invalid program, your scanner or parser should output an appropriate error message. For a syntactically valid input program, your parser should output a Graphviz DOT file.

Type Checking and Type Annotated AST

Once you have generated the AST. The compiler should traverse the AST to find the type of each sub-expression and determine whether there are any violations of the language's type rules.

In order to type check the program, you will need to create a symbol table to associate each appearance of a variable with that variables declared type. The best way to do this, is probably to create an entry in a hash table indexed by name and containing information about each declared entity. (For TL, it is sufficient to just track the name and type of each declared variable.) And then when the AST generation code comes across a variable in a procedure body, it can look that name up in the symbol table and store a pointer/reference to that symbol table entry in the AST node.

Three Address Code Generation

The next step is for the compiler to traverse the abstract syntax tree and translate the program into basic blocks of three-address code according to the TL language semantics. This three address-code will be another in-memory data structure. Your compiler must output a graphviz DOT file depicting control-flow graph where each node represents a block of instructions and is labeled with those instructions.

It is suggested that you get the CFG output (see below) working as early as possible to aid in testing and debugging. In order to be able to produce the appropriate CFG output and translate into MIPS assembly code, you will probably want to have a class representing blocks (encapsulating a list of instructions and providing operations to iterate over that list and to access the successor blocks) and also a class (or hierarchy of classes) representing individual instructions.

MIPS Assembly Output

As the last step, the "back end" of the compiler should translate the program into executable MIPS assembly code. This involves selecting the appropriate MIPS assembly language instructions, modifying the code so that it executes with a finite number of registers, and outputting an ASCII file containing MIPS assembly code that is executable with SPIM.

The ILOC and MIPS instruction sets are quite similar, and depending on the subset of ILOC which the IR contains, instruction selection may be done through a simple 1:1 substitution of MIPS instructions for ILOC instructions.

Input/Output Specification

The compiler program's main method should expect a TL input file, <basename>.tl, to be specified as command-line argument when it is invoked.

If the input file is not a syntactically-valid program (i.e., if the program contains a lexical element that is not a legal token or does not match the BNF grammar), the compiler should output an error message to standard error stream containing the text "SCANNER ERROR", the text "PARSER ERROR", or the text "SYNTAX ERROR".

If there is no syntax error, the compiler should output an abstract syntax tree in graphviz dot format to the AST output file,<basename>.ast.dot. If there is a type error, the compiler should output an error message to standard error that contains the text "TYPE ERROR".

If there is no syntax or type error, the compiler should also output a control flow graph in graphviz dot format to the file,<basename>.3A.cfg.dot. The control flow graph should be based on a translation of the original program into three-address code.

The final output of the compiler, MIPS assembly code that faithfully implements the semantics of the original source program, should be written to the file, <basename>.s.

So for a correct TL source program, <basename>.tl, the compiler should produce a Type-annotated Abstract Syntax Tree (<basename>.ast.dot), a control flow graph (<basename>.3A.cfg.dot), and a MIPS assembly code file (<basename>.s).

Testing

You are also required to adequately test your compiler and submit your test cases along with your source code and document the current state of your compiler based on your own testing. Include both incorrectly-typed programs and correctly-typed programs that exercise all parts of your compiler.

Grading

The full points for the project is 30 points, accounting for 30% of the course.

The distribution of the points are:

Lexical Analysis (5 points)
Parser, including type annotation of AST (10 points)
Code Generation, including generating assembly code (10 points)
Documentation (5 points)

There are up to 5 points that can be added to your score of the project if you are able to implement constant propagation and the elimination of unreachable code.

Delivery

There are 3 due dates for the project.

Lexer due Feb. 21st
Parser (including type annotation) due Mar. 30th.
Whole Program (including optimization if you plan to work on it) due May 3rd.

The project should be uploaded to Blackboard before the due date.
Format of the file uploaded:
[abc123ID]_[lastname].zip

The uploaded folder should contain your source code, all dependencies, and a sub-folder called 'workdir' where you should put two files 'build.sh', and 'exec.sh'. Running 'build.sh' should build your project, and running 'exec.sh <basename>.tl' should execute your compiler on a source code file in 'workdir', and output all corresponding outputs into 'workdir'.