Alex Torres (rkc870)
Jarred Deeley (vaa291)

TL Compiler Project Document

We have created a compiler capable of compiling a TL source file into a MIPS assembly source file, while generating several intermediate representations of the TL source file. These intermediate representations include a token stream file, a diagram of a type-annotated abstract syntax tree, and a control flow diagram.

This compiler was constructed using the Python programming language and was divided into three sequential components- scanner, parser, and code generator. Our compiler also had a “main” component which executed wrapper methods in each compiler component and controlled main flow of the data. For this project we’ve decided to work in parallel and combine ideas from both of our individual programs in order to obtain a final compiler that is a combination of the most robust sections of two individually developed compilers. Also, educational motivations nudged us towards this development process. What I mean by this- we were both interested in constructing all phases of the compiler for learning purposes. *Jarred and Alex both contributed to the “main” component of this project.*

###################################
compiler.py
Main file for the compiler.
Expects a single command line argument which is the file to be compiled.
Calls scanner, parser, and code generator.
###################################

The scanner works by accepting a line from the TL source file and recursively examining the line for tokens. The scanner accepts a TL source file as input and outputs a token stream if the TL source file is lexically valid. Otherwise, the scanner throws a lexical error and terminates compilation. *Jarred constructed the majority of the scanner component of the project.*

###################################
scanner.py
Takes a file and base name as input.
Produces a file named with the base name and of the type tok.

lexer(inputFile, baseName)
Opens or creates the file <baseName>.tok
Reads the inputFile line by line and processes the tokens.

processToken(tok, lineNumber, output)
takes in a token, line number, and output file as input.
Processes a token and determines if it is valid.
Prints to stdout if there was an invalid token. Prints the line number and
possible cause.

findSubTok(tok, subStringLength, output)
Takes a token, length of a substring, and output file as input.
Checks the substring of a token for a valid token.
If a valid substring token is found, then the substring will be written
to the output file as a token and the part of the token that isn't in the
substring is checked to see if it's a valid token.

validToken(tok, output)
Takes a token and output file as input.
Checks if the token is valid according to TL 16 specs.
If the token is valid, then the token is written to the output file.
Returns a boolean value based on if the token is valid or not.
##############

The parser accepts the token stream from the scanner and outputs a type-annotated abstract syntax tree and symbol table if the token stream is syntactically valid. If the token stream is not syntactically valid, or if the token stream encounters a variable not yet declared, then the parser throws a syntax error and terminates compilation. Likewise, if the parser a type mismatch then the parser will throw a type-error and output a type-annotated AST color coding the type-error. If the token stream is syntactically valid then the parser will output a colorless type-annotated AST and a symbol table. *Alex constructed the majority of the parser.*

##############
ParseRv5.py
Class Parser:
Wrapper class containing the main imperative processing of the parser. Calls static helper classes for grammar verification and AST synthesis. Makes use of Node and ASTree objects.

ASTree.py
Class ASTree:
A single ASTree object is used to represent the Abstract Syntax Tree, also contains a helper method for type checking and GraphViz generation.

Node.py
Class Node:
A Node object represents a single node in the AST and contains all necessary information for AST synthesis and type checking.

SyntaxChecker.py
Class Checker:
Contains static methods used to check the syntax of various grammar elements.

AstBuilder.py
Class Builder:
Contains static methods used in AST synthesis. Static methods assume that the input token stream is syntactically valid.
##

The code generator accepts the symbol table from the parser and outputs a control flow graph, and also the MIPS assembly source file for the given TL source file. *The control flow graph and 3-address generator was constructed by Jarred and the MIPS assembly generator was constructed by Alex.*

##
codeGeneration.py
generate_code(ast, base_name)
Takes in an AST and the base name of the tl file.
Produces a CFG and MIPS source file.
Calls three_code_gen(ast) which will take an ast and generate a block structure.

MipsSource.py
class MipsSource
Contains static methods for writing mips source file given a control flow graph of the program.

A block is a structure which has a pointer for entryBlock, thenBlock, and elseBlock.
Each block also has a label (composed of Block + its number) and a list of its instructions.

Code generation is a two-part recursive method, three_code_walk_ast(block, ast, parent='program', symTable = None).
When the method is called, the ast node will call the same method and pass its children as a parameter.
The method checks the children before calling the recursive method to see if more blocks will need to be created. All of the blocks and most of the connections are done at this stage.

Once a node has called all its children with the recursive method, it will handle the current ast node.
The current node will take lists of returned strings from its children and combine it into one list, unless the node is a stmt list, if, or while. These nodes will pack all the instructions it has been given into its current block.
Once this phase is down, the block structure will have all of the MIPS instructions.
##