CSSE 404 Compiler Documentation

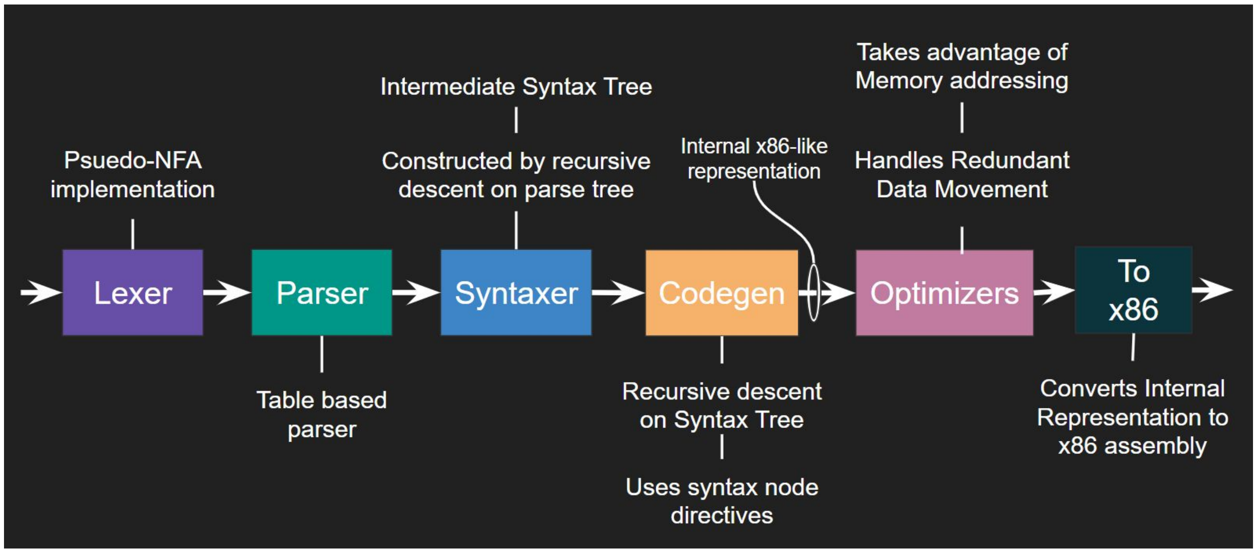
Leela Pakanati, Manoj Kurapati

Overview

Our compiler is made up of 6 total stages:

- Lexer
    - Tokenize input file and determine the function of all symbols
- Parser
    - Parse the input to create a tree that forms the input into the grammar
- Syntax tree generation
    - Generate a syntax tree out of the parse tree for increased usability in code generation
- Code Generation
    - Generate intermediate assembly code from the syntax tree
- Optimization
    - Layers that take a list of instructions and return a new list of more optimized instructions
- x86 Conversion
    - Convert Intermediate code to x86 assembly

The following image shows the general functionality of each of these stages:


Psuedo-NFA implementation
Lexer
Parser
Table based parser
Intermediate Syntax Tree
Constructed by recursive descent on parse tree
Syntaxer
Internal x86-like representation
Codegen
Recursive descent on Syntax Tree
Uses syntax node directives
Takes advantage of Memory addressing
Handles Redundant Data Movement
Optimizers
To x86
Converts Internal Representation to x86 assembly

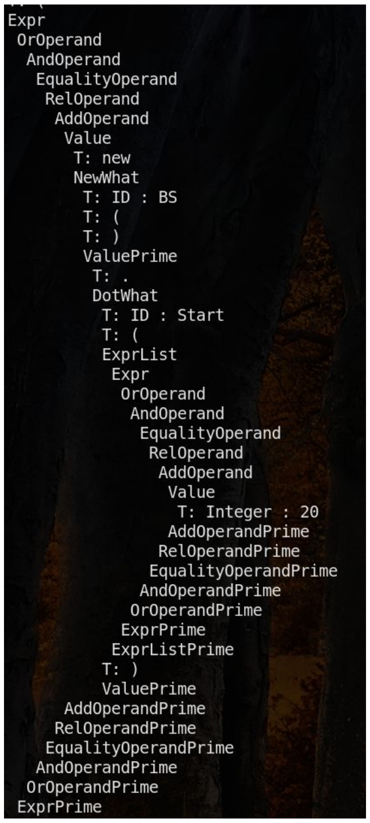
Throughout this process, these are also custom data types used for the SymbolTable and for the intermediate instruction representation.

Lexer

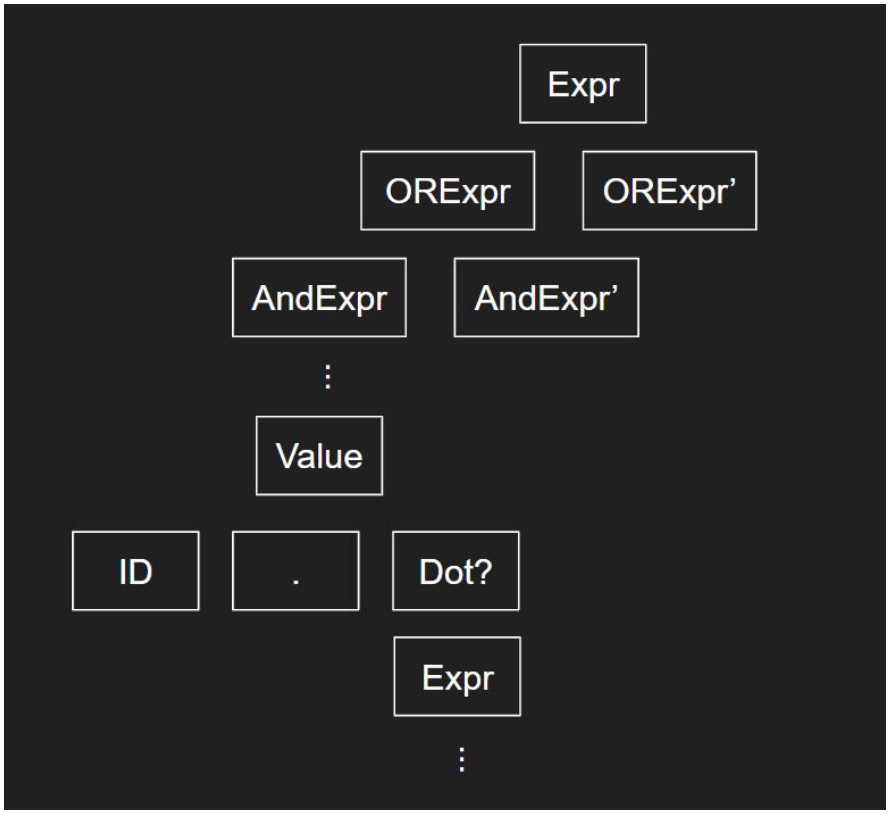
The first step of our compiler is lexing the input file to properly extract the text from the file into a form that distinguishes each component. In this stage, we take an input file and determine each word/symbol and what it represents, We classify each of these tokens one of the following: ReservedWord, Integer, Operator, Delimiter, ID, Whitespace, or Comment. This process is done by using a NFA - like algorithm that sends the words through each 'path' in the right order to allow for preference. We will pass the next character into the test for whitespace first, if it fails, we send it to Operator, then Delimiter, Integer, ReservedWord, ID in that order. Each of these checks will keep pulling in more characters from the input to see if the full token will match. Once one of these types match on the input, it saves to a buffer the identified type, the token, and the line number it was found on.

Parser

The next step of the compiler is the parser. For this stage, we modified the original grammar we were given to take into account order of operations and make it unambiguous and LL1 compliant. Once we had that, we were able to use an online tool to create first and follow sets for each of the tokens. This final grammar can be found in './grammar/grammar.txt'. With this grammar design done, we used a simple table parser algorithm as discussed in class. By taking the output of the lexer, we start at the first token and use the table to follow by each new symbol seen, going to the respective next item in the parse table. A sample portion of an output from this parse tree is shown in its text output representation and an abridged tree representation below:

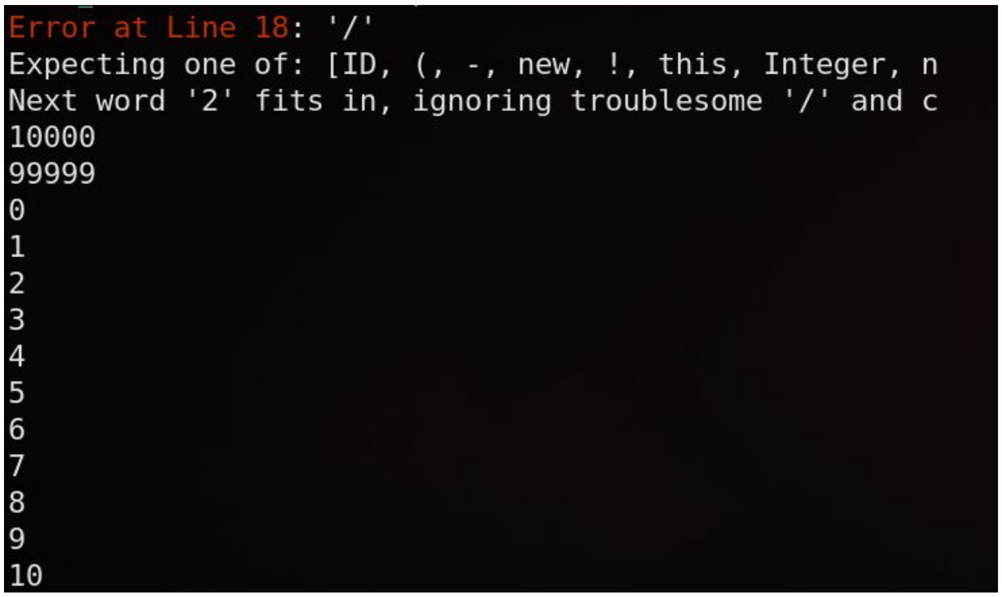


```
Expr
 OrOperand
  AndOperand
   EqualityOperand
    RelOperand
     AddOperand
      Value
       T: new
       NewWhat
        T: ID : BS
        T: (
        T: )
        ValuePrime
         T: .
         DotWhat
          T: ID : Start
          T: (
          ExprList
           Expr
            OrOperand
             AndOperand
              EqualityOperand
               RelOperand
                AddOperand
                 Value
                  T: Integer : 20
                 AddOperandPrime
                RelOperandPrime
               EqualityOperandPrime
              AndOperandPrime
             OrOperandPrime
            ExprPrime
           ExprListPrime
          T: )
          ValuePrime
      AddOperandPrime
     RelOperandPrime
    EqualityOperandPrime
   AndOperandPrime
  OrOperandPrime
 ExprPrime
```

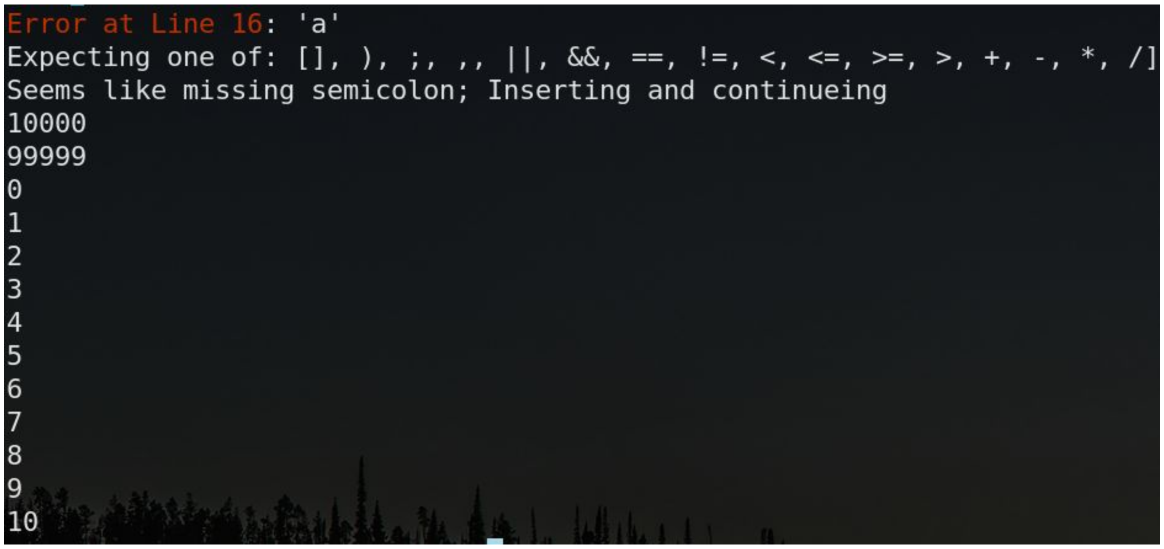


Error Detection:

In the parser, we handle some of the errors associated with the parsing stage. We run into an error in the table parser if the next token isn't in the follow set from the current token. If this occurs, one way we can handle this error is by checking to see if the next token will fit in the set instead. If this works, then we simply ignore the violating character and continue compilation after throwing a warning to the user. Alternatively, if this doesn't work, we check to see if a semicolon is in the current follow set. If so, then we assume that the issue is a missing semicolon. If the current token then works for after an inserted semicolon, we insert the semicolon and continue compilation. An example of each of these situations is shown below where we compile and run a test program.



```
Error at Line 18: '/'
Expecting one of: [ID, (, -, new, !, this, Integer, n
Next word '2' fits in, ignoring troublesome '/' and c
10000
99999
0
1
2
3
4
5
6
7
8
9
10
```

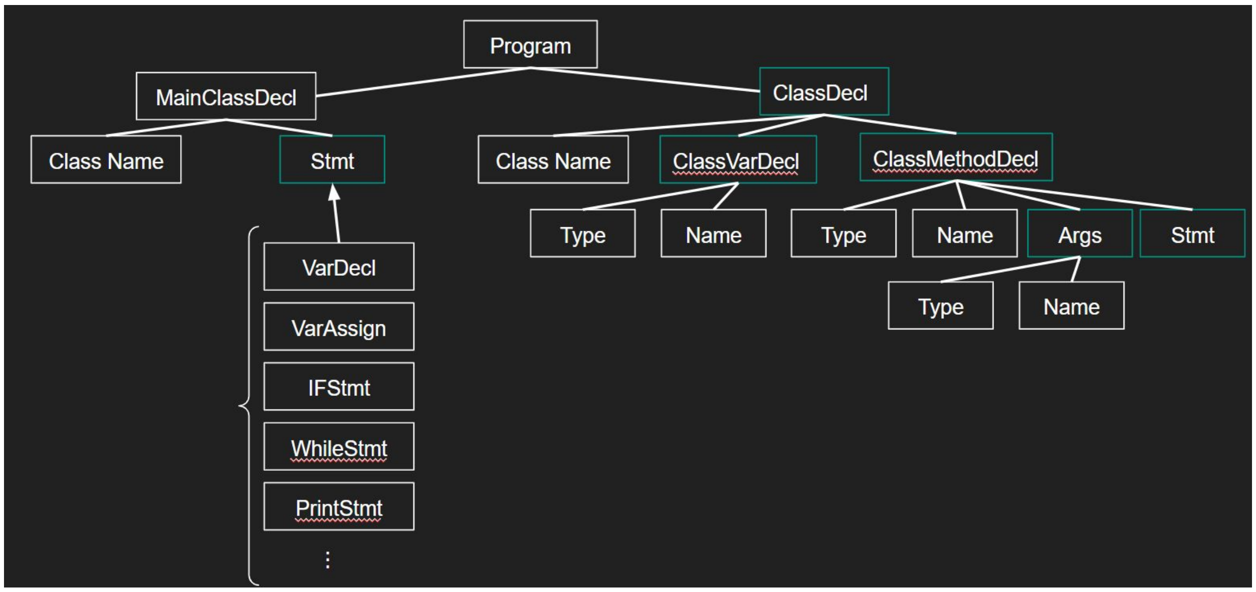


```
Error at Line 16: 'a'
Expecting one of: [], ), ;, ,, ||, &&, ==, !=, <, <=, >=, >, +, -, *, /]
Seems like missing semicolon; Inserting and continueing
10000
99999
0
1
2
3
4
5
6
7
8
9
10
```

Syntax Tree Generation

While the parse tree is a really good way of properly demarcating different parts of the input code, the output is actually unusable for proper code generation. This next stage of the compiler is taking the parse tree and generating a new Abstract Syntax Tree that is able to better encode the information for generating code.

This Syntax Tree holds much of the information closer to the constraints of the original grammar and exposes the general code structure better. Using the output of the parse tree it forms a tree out of nodes that hold the data relevant to that part of the code. For example, a ClassDeclaration node object holds the name of the class, a list of Formal nodes that hold the type and name of each class variable, and a list of MethodDeclaration nodes. These nodes can be generated through recursive descent, passing around parts of the parse tree as relevant to each child. For example, for the ClassDeclaration node discussed earlier, the parse tree would have different children that relate to each of the different properties of the ClassDeclaration node. The node would then take the child that stores the name, hold it, then send the parse tree children related to the class variables to the constructor of Formal. Similarly, it would send the children related to the list of MethodDeclarations to the constructor of the MethodDeclaraction node object. The objects returned by these constructors are held in the parent node. For Node types Statement and Expr, the type can be one of many different subclass nodes so rather than a constructor, these use a factory method called 'getInstance' which acts like a constructor that returns the right type of node that corresponds with the parse tree sent in. An abreviated example structure of the syntax tree can be seen below (green box means a list of that object type):

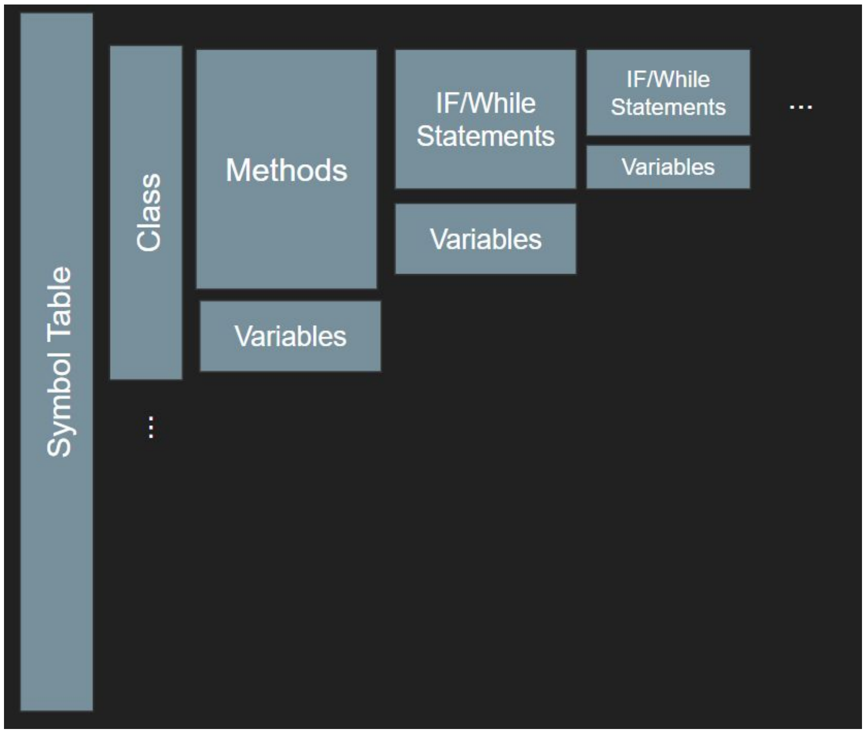


Symbol Table

During the generation of the Syntax Tree, we also start building out the Symbol Table. We have a singleton type class for the SymbolTable. We have an abstract type Symbol which is extended by ClassSymbol, MethodSymbol, VarSymbol, IfSymbol, and WhileSymbol. Each symbol has a name and map of string to symbol for its children symbols. MethodSymbols additionally hold their return type and input arguments. VarSymbols additionally have their type, an enum that determines if it is a class variable, local variable, or input argument variable, and an integer describing the index of the variable (in the order of which the variables are instantiated in their respective scopes).

The SymbolTable object holds a list of ClassSymbols which acts as the uppermost level of the tree of symbols that makes up the table. While and If statements are also tracked as symbols to allow for limiting scope to their children when applicable. A global counter is kept for IDs for the If and While symbol names. The SymbolTable object also holds a list of Symbols which acts as its scope. An example state of the scope could be like this: <ClassSymbol: Class1> , <MethodSymbol: Method2> , <IfSymbol: If3>
This would show that what we are currently generating is inside of Class1, within Method2, and in the If statement that was assigned an ID of 3.

A general overall symbol table structure is shown below:



When looking for the symbol of a variable, we first look at the last object in the scope list. If its not in the VarSymbol list of this object, then we look at the object 1 prior to it in the scope and check its VarSymbol list. This continues either until the symbol is found or we get to the first item in the scope at which point an error is thrown that the symbol isn't in scope.

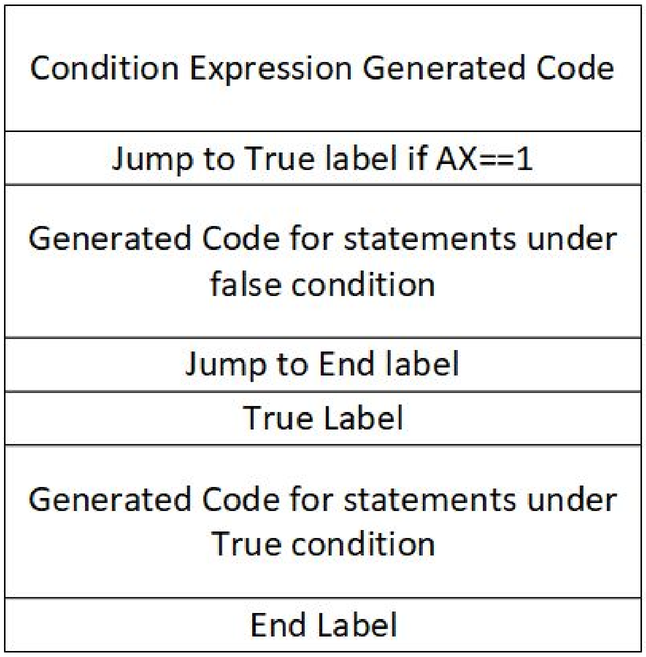
During construction of the Syntax Tree, the symbol table is populated with all ClassSymbols, MethodSymbols, and VarSymbols for class variables. All other variables, if statements, and while statements are done during the Code Generation step.

Code Generation

With the built Syntax Tree, we are able to then generate code with it. This is done by calling 'CodeGen' on the top level of the syntax tree nodes which then works as a recursive descent calling into the rest of the tree, returning a list of the type 'Instruction'. This generated code is very similar to x86 but is more suited to intermediary manipulation and is outlined in the arch package. This is discussed in the architecture section.

The ‘CodeGen’ function for each syntax tree node primarily works by collecting the outputs from running ‘CodeGen’ on its children nodes and arranging them with additional relevant instructions into a list to return. For example code generation for a method declaration adds a label for the method, adds instuctions regarding setting the base and stack pointers, adds the outputs from calling ‘CodeGen’ on each of the Stmt nodes it holds, and instructions to reset the stack and base pointers as well as a return instruction. One thing to note is that upon entering classes, methods, and if/while statements, the scope of the lookup table is appended to allow for subsequent variable lookups to be in the proper scope. At every variable declaration, variables are added to the symbol table as well and it is referenced whenever a variable is accessed to check its existence and type.

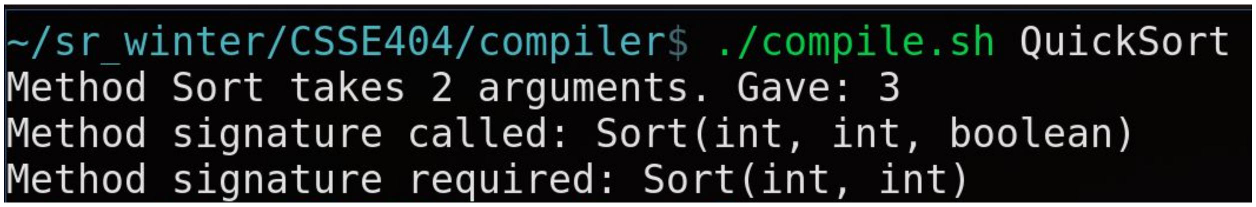
Another interesting example of code generation is for If Statements. These create a label for the true condiion, and the end of the statement. It then adds instructions for evaluating the condition, a jump to the true label if it is true, the list of instructions for the generated code of its statements under the false condition, a jump to the end label, and then the list of instructions for the generated code of its statements under the true condition, and then the end label. This ends up looking like the following figure.


Condition Expression Generated Code
Jump to True label if AX==1
Generated Code for statements under false condition
Jump to End label
True Label
Generated Code for statements under True condition
End Label

One thing to note is that we generate the code in a very x86 manner with regards to register usage. We use the EAX, EBX, ECX, and EDX registers as intended. For Expr type tree nodes which are intended to evaluate a value to be used, the final value is stored in the EAX (the accumilator) register such that it can immediately be operated on easily. The EBX register is used primarily for holding addresses when refering to memory operations; ECX is used primarily for array indexing; EDX is used primarly as the second register for arithmetic and other dual operand operations.

Type Checking

While generating the code, we also do a good amount of type checking. Whenever we generate code for a statement that assigns an expression to a variable, we check with the variable symbol to see the type the variable and check that the expression it is getting set to matches. With arrays, we check that the index and the assigned value both evaluate to integers. Furthermore, for method calls, we validate the call signature with number and type of arguments and check that they match the number and type of arguments it is being called with. An example of the method signature error output is shown below.



```
~/sr_winter/CSSE404/compiler$ ./compile.sh QuickSort
Method Sort takes 2 arguments. Gave: 3
Method signature called: Sort(int, int, boolean)
Method signature required: Sort(int, int)
```

Architecture

For Code Generation purposes, we hold a few details about the architecture. This package includes an enum for Register which has a type for each of the main x86 registers and a '.label' value that returns the 32 bit version of each of these. We also have an Operation enum which holds a list of all of the x86 instructions we use.

The main part of this package however is the Instruction objects. We have an abstract Instruction type which holds the general instruction components like the Operation, a few Registers named 'RS1' and 'RS2', and a immediate integer value. This is then extended by each instruction type which is able to implement a proper constructor for the types it supporters and has a function to convert the instruction into x86 assembly. Each of these instruction types also has a toX86 function that is used in the x86 conversion stage.

For example, a JumpOp is a type for all of the jump operations. This type additionally holds a jump label. It is constructed with either a condition (which determines a conditional jump operation as its Operation) and a label, a label and a boolean (indicating an unconditional jump and whether or not it is a function call), or an empty constructor (indicating a return from a function).

Optimization

Once we have generated the list of instructions for the program, we go on to run it through a couple optimization layers. We handle optimizations through having an abstract type Optimizer that has a function 'optimize' which takes in a list of Instructions and returns an optimized version of the list of Instructions. We then have a list of 'Optimizer' objects that then each run on the instruction list.

Redundant Memory Movement

The first of our optimizers removes instances of redundant memory movement where a value is moved to a regiser only to be moved again to another location. In these instances the register is unnecessarily used as an intermediate and thus, the instructions can be combined. This is done simply by checking each mov instruction where the destination is an instruction and the following instructions is also a mov with its source as teh same register. We can then simply combine these instructions and acheive some optimization.

Combining Immediate Arithmetic

The second of our optimizers combine sequential arithemtic of the same type done on a single register. This primarily comes up when we do subsequent variable declarations; we have several ‘Sub REG 4’ instructions. This optmization checks the code for instances of multiple arithmetic operations on a single register with an integer operand and simply combines them all into one instruction.

PushOp on Memory

One of the greatest locations of inneficiency found in our code is that of using intermediate registers when working on memory when the function can instead operate directly on that memory. This optimization is a fix on one of these instances. Whenever we push from a memory location to a register and then push the register onto the stack, we can combine this into one instruction of pushing the memory location onto the stack. This optimization looks for these instances and changes it to the single push instruction operating on memory. Our code actually has many more similar situations that can easily be optimized with a very similar scheme.

X86 Conversion

This final stage is a simple conversion of the intermediate instruction representation to proper x86 assembly strings. As discussed above, each of the Instruction type objects has a 'tox86' function that converts itself into x86 assembly. For most instructions this step is pretty simple and follows the general arrangement of: 'Op Destination Sources'. For example, for an ArithOp which represents an add operation of EAX and the immediate value 5, the output would be: “add EAX, 5”.