Compiler Design Document
Semantic Checking and Code Generation
Team: Human suffering (@ajyliew, @emshen, @skoppula, @srobin)

This design document covers the specifics as to how our compiler does semantic checking and code generation, and the data structures that we use. We divide this document into two main parts, the first part detailing semantic checks, and the second code generation. There is large overlap between the two parts (e.g. we re-use structures form semantic checks in the code generation), and we will detail that in the coming sections. The document will also outline the structure of our codebase along the way, and when relevant, refer to specific parts of the code so that the reader is familiarized with our Scala implementation of a Decaf compiler. Please enjoy!

Semantic Checking

High-Level IR Tree

The output of the lexer and parser of our compiler is an ANTLR tree with nodes corresponding to tokens in our grammar. We first convert the entire tree into a high-level IR tree composed of nodes that correspond to classes that we define. An example of the resulting high-level tree after we have processed the ANTLR tree (visualization of our class structure courtesy the `sext` library):

```
IrProgram:
- List:
| - IrCalloutDecl:
| | - random
| | - Node location(line: 1, column: 1)
| - IrCalloutDecl:
| | - srandom
| | - Node location(line: 2, column: 1)
| - IrCalloutDecl:
| | - printf
| | - Node location(line: 3, column: 1)
- List:
| - IrFieldDecl:
| | - IrIntType:
| | | - Node location(line: 8, column: 1)
| | - List:
| | | - IrArrayFieldDecl:
| | | | - A
| | | | - IrIntLiteral:
| | | | | - Some:
| | | | | | - 100
| | | | | - 100
| | | | | - Node location(line: 8, column: 1)
| | | | - Node location(line: 8, column: 1)
| | - Node location(line: 8, column: 1)
```

Figure 1. The high-level IR tree for the callouts and first two lines processing `./tests/semantics/legal/legal-01.dcf`

A complete list of `IrClasses` that we used can be found in `IrClasses.scala`. The conversion of the AST begins with a method call to `constructIR(ast : TokenAST, exceptionGenie : ExceptionGenie)`, and uses a handful or other methods in `IrConstruction.scala` such as `calloutNodeToIrCalloutDecl(ast: AST, exceptionGenie : ExceptionGenie)` to

convert the AST nodes into `IrClass` nodes. This enabled us to modify fields within the nodes (e.g. extending each `IrNode` with a `NodeLocation`), and easily create helpful `IrTree` debug print methods.

Symbol Tables

The next step after construction of the high level IR tree is a DFS traversal of the tree in order to execute type checks and execute variable existence/scope checks. We maintained a tree-like symbol table structure composed of hash tables at each node to keep track of the variables, methods, and their types in every scope. We detail this symbol table structure in Figure 2.

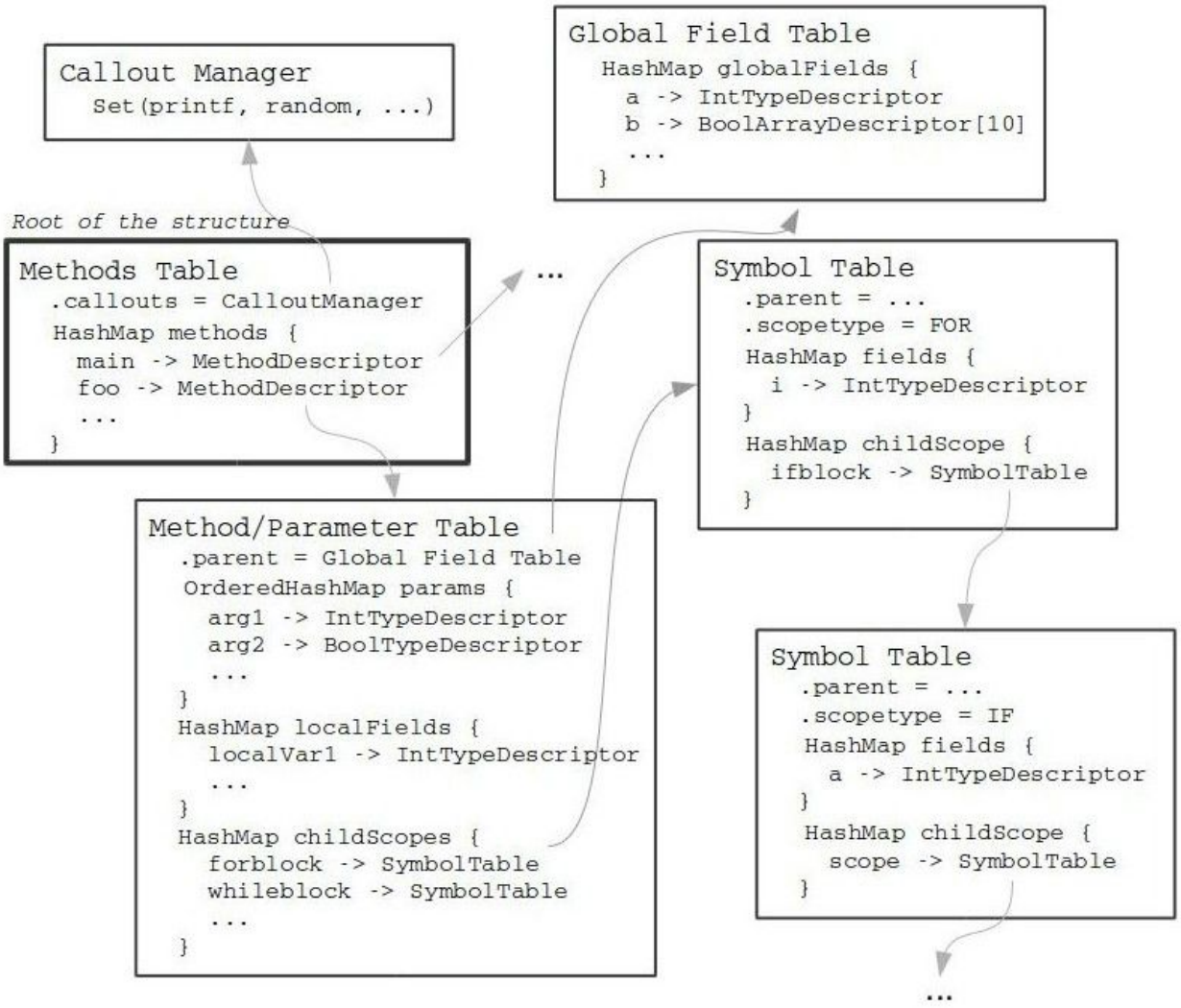


Figure 2. Our tree-based symbol table structure to keep track of variable scope and declarations. When a `lookupID(name : String)` query is called, the `SymbolTable` type recursively calls its .parent and returns the resulting `TypeDescriptor`.

When parsing the `IRTree`, we add layers of `SymbolTables` to the structure, and insert any declarations into the current scope. We keep track of the current scope by maintaining a `Stack<SymbolTable>`, the top of which represents the current scope. Note that the top of this stack could be of type `Method/ParameterTable` as well. The symbol table structure performs checks as it is created, throwing exceptions of various types corresponding to semantic checks relating to variable existence and scope. Some examples include `IdentifierAlreadyExistsException` or `MethodAlreadyExistsException`. A full list of exceptions that can be thrown by our compiler (and subsequently caught by `ExceptionGenie` and reported back to the user) can be found in `Exceptions.scala`.

IrChecks

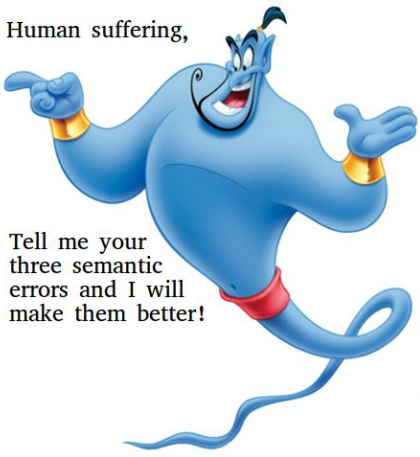
During the traversal of the high-level `IRTree`, we not only construct the symbol table structure, but we also run a series of type checks while processing each node. Every type of node that has a related semantic check has a specific 'check' method in IrCheck.scala that we call to ensure certain type constraints. For example, `checkIrAssignStmt` has the following signature:

```
def checkIrAssignStmt(
    methodsTable: MethodsTable,                  // The entirety of the symbol table structure
    scopeStack: mutable.Stack[SymbolTable],      // The current scope
    stmt: IrAssignStmt,                          // The current IrNode being processed/type-checked
    genie: ExceptionGenie                        // Handler to catch exceptions
  ) : Boolean = {
        ...
}
```

and recursively evaluates `checkIrLocation` and `checkExpr` for the left-hand side and right-hand side of the assign statement, respectively (`IrLocation` and `IrExpr` are the two children `IrNodes` of an `IrAssign` node). It then performs type check equality for both sides once those two calls are done.

Error Handling

In order to return to the user as many semantic errors as we can at one time, we implemented an `ExceptionGenie` container (`ExceptionGenie.scala`) that is passed around and stores all semantic errors that we find while traversing the high-level IR tree or inserting items into the symbol table. At the end of semantic checking, if there are any exceptions in `ExceptionGenie`, we return them to the user so that they are able to fix their Decaf program.



Exceptions that occur because of compiler problems (hopefully none!), also are fed into `ExceptionGenie` and eventually returned to `stderr`.

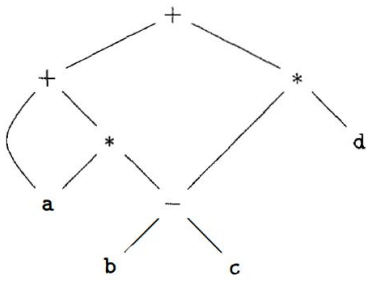
Intermediate Code Generation (Low-Level IR)

In preparation for generating assembly code, we translate the high level IR tree into a lower level intermediate representation. The main flow of the program occurs in `src/Compiler.scala` where the high-level IR tree generated during semantic checking is passed to `IrGen.scala`. `IrGen` handles the translation of our current IR tree into a lower-level IR.

The translation is performed by traversing the high-level IR and converting instructions into a series of three-address codes, which are closer in form to assembly language. For example, a binop expression may be represented as follows:

```
addr1 = addr2 op addr3
```

These three-address codes make our lives simpler when generating assembly, by allowing us to use only two temporary registers (`%r10`,`%r11`) for each operation. This is because there is always at most one operator on the right side of an instruction. Below is an example of the expression `a + a * (b - c) + (b - c) * d` reduced to three address codes:




```
t1 = b - c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4
```

Figure 3. DAG via dragon book p. 359

We represent these in our program by creating case classes to model the structure of three-address code for different types of instructions. These case classes can be found in `src/compile/tac/ThreeAddressCode.scala`. We also produce temporary variables that, as we will describe in more detail later, are inserted into the symbol table structure. After producing a buffer of TACs (three address codes), the conversion of TACs to assembly proceeds in `AsmGen.scala`, which is discussed in the next section.

Preparation for Code Generation: Symbol Tables Rise Again

Before we translate our tree-of-TAC-lists structure into assembly, we must know each variables size and offset to allocate space on the stack in our assembly, and obtain correct variable memory references in assembly code. To do this, we created a field in each of the four `TypeDescriptor` classes that detailed the size and offset of the described variable. We

maintained a field in the top level symbol table for each method called `currTotalByteSize`. The field (representing the max offset from the methods `%ebp`) allowed us to assign the correct offset to any new variable or temporary variable (created from TAC generation) that did not overlap with previously allocated variables. In this way, the symbol tables allowed the assembly generator in `AsmGen.scala` to lookup a variable's offset with respect to the current `%ebp` by querying the variable name in the symbol table structure.

Code Generation

General Strategy

`AsmGen.scala` contains a function `AsmGen(tac: ThreeAddressCode, symbolTable : SymbolTable)` that converts any TAC into the corresponding assembly, given the current variable scope. This `List[String]` of assembly instructions are mapped to a TAC and lumped into a `LinkedHashMap[Tac, List[String]]` that maintains all our generated assembly. This allows us the flexibility to rearrange segments of assembly, before providing our final output. By iterating across the ordered values of the `LinkedHashMap`, we obtain the final assembly which we output to the user.

We aimed that the mapping between a TAC and its corresponding assembly be relatively straightforward, and not require surrounding TACs to be able to produce correct assembly. As such, TACs generally represent basic assembly operation.

For example, the `TacBinOp` object is converted to assembly by a helper method `binOpToAsm` in `AsmGen.scala`. Similarly, in the `gotoToAsm` method, the `TacGoto` object is converted to a jmp instruction, jumping to the object's stored label. Because we are supporting 64-bit machines, most of our assembly instructions use quadword operations (e.g. `movq`, `addq`, etc.)

One detail of note, essential for understanding our codebase is our so-called `TempVariableGenie`, that is responsible for:

1. Outputting a unique ID for every TAC. Because we're storing TAC-assembly pairs in a hashmap, we could not have different TAC objects collide (e.g. two `TacGoto` with the same label [and hence same hash], but in different part of the code). An astute reader might object that objects of with different memory addresses would not collide, but in Scala's case classes, the hash attribute match when constructor arguments match.

2. Produce new, non-colliding labels on demand, for new string literals and control flow.

3. Produce new, non-colliding temporary variable names on demand, so that temporary variables don't collide in the symbol table.

Method and Callout Calls

When calling a method or callout, we first save the arguments to temporary variables. This list of temporary variable names is stored inside a `TacMethodCallExpr` or `TacMethodCallStmt` (depending on whether the call is an expression or statement), and `asmGen(...)` takes care of passing in the variables to the variable using a predefined allocation of parameters on the stack (offsets of which had been pre-computed by the symbol tables when the high-level IR tree was traversed). Our way of handling callouts is defined as per `x86` convention. To save your increasing boredom with our write-up, we won't describe the convention here, but there are two interesting features that we had to deal with regarding callout arguments: string literals and callout array arguments.

Handling Callout Arguments (String Literals and Arrays)

String literals were scattered throughout callout calls within the high-level IR tree; however, the assembly construction we were using to handle string literal constants required an assembly section with all our string literals defined with labels, beginning with a `.section .rodata`:

```
        .section        .rodata
.L1:

        .string "top of gurp\n"
.L5:

        .string "done j\n"
        .text
```

Here, our LinkedHashMap structure, came in handy. We pulled the `TacStringLiterals` to the front of the linked list and encapsulated this block of TACs is a `TacStringLiteralStart` and `TacStringLiteralEnd.`

Global Variables

We used the very simple `.comm` syntax to define and allocate our global variables at the start of the program (see snippet below!). This way, we were able to reference any global variable by name in assembly.

```
.comm   b,8,8

.comm   a,8,8

.comm   c,8,8
```

Miscellaneous

Runtime Checks

To check for array-out-of-bounds errors, we precede every array index access (the `TacArrayLeft` and `TacArrayRight` TACs) with a roughly four TACs that collectively call the

unary length operator on the array, check whether the evaluated index is within bounds, and jump to a `TacSystemExit(1)` TAC, if the check returns `false`.

Initialization of Local Field Variables
We employ a simple iteration over every local field variable in the current scope without the prefix "`.T`" in its name (indicating the variable is a generated temporary variable). We set these variables (or set of locations, in the case of an array variable) to zero or `false`, depending on the type.