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

This design document covers the implementation details of our Decaf compiler's semantic checking and code generation, and the data structures of our internal representations (IR). We divide this document into three sections, the first part detailing semantic checks, the second, intermediate code generation, and finally, converting intermediate code into assembly. There is significant overlap between these parts- for example, the symbol table is generated during semantic checking and also used for many aspects of code generation. Throughout the document, we also highlight the structure of our codebase, and when relevant, specific parts of the code.

1. Semantic Checking

a. High-level IR tree

The output of the lexer and parser of our compiler is a custom parse tree, subclassed from the default ANTLR output, such that nodes correspond to tokens in our grammar and have line and column number information.

To create the high-level IR tree, we created several case classes that represent abstract syntax components. A complete list of `IrClasses` that we used can be found in `IrClasses.scala`. Next, we convert the parse tree into the abstract syntax tree by calling `constructIR(ast : TokenAST, exceptionGenie : ExceptionGenie)`, which recursively travels the parse tree and converts branches of tokens into IR nodes. Figure 1 illustrates an example high-level IR tree, visualized using 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 resulting from the callouts and first two lines of `./tests/semantics/legal/legal-01.dcf`

b. 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 variable existence/scope checks. We maintain a tree-like symbol table structure composed of hash tables to keep track of the variables, methods, and their types in every scope. The *global field table* is the symbol table of global variables, and the *callout manager* simply maintains a set of declared callouts. The root of the structure is the *methods table*, which is keyed on method name and returns the method descriptor, which encompasses the symbol table for the method's scope (parameter table) and the method's return type. The *parameter table* maps the method arguments to their type descriptor in order, and additionally stores any local variables declared in the method. This is depicted below in Figure 2, and the corresponding code can be found in `src/compile/symboltables` and `src/compile/descriptors`.

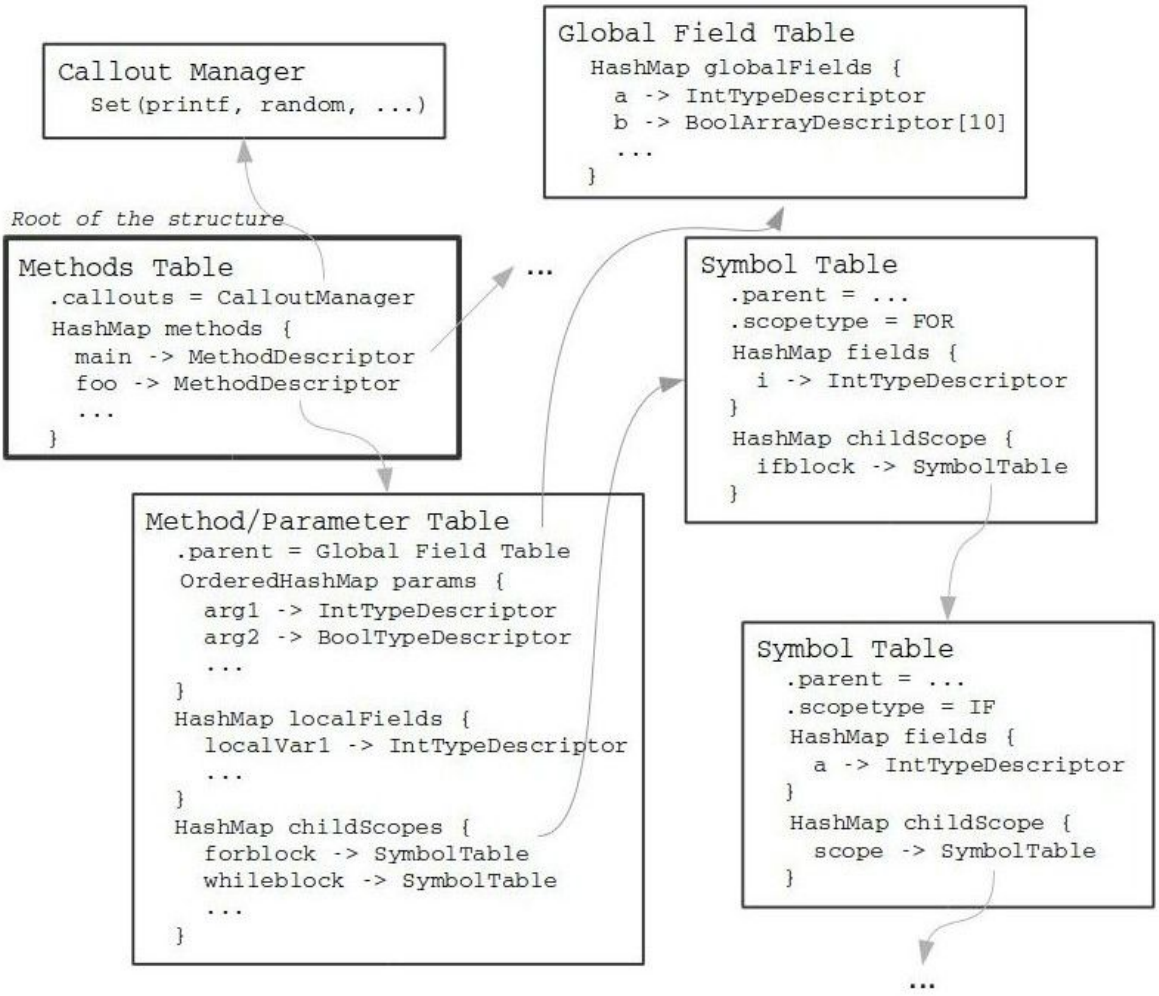


Figure 2. Our symbol table structure to keep track of variable scope. `lookupID(name : String)` is used to recursively check a variable or method's existence in the program. It either returns the corresponding `TypeDescriptor`, or `null` if not found.

We keep track of the current scope by maintaining a `Stack<SymbolTable>`, the top of which represents the current scope, and initialized with the global field table. When walking the `IrProgram` tree, on discovering a new scope (i.e. methods, for, while, or if blocks), we add a `SymbolTable` to the structure by setting its parent to the current `SymbolTable`, then pushing it onto the stack. Then, we insert any declarations.

As the symbol table structure is updated, it performs semantic checks related to variable existence and scope. In particular, it detects when a variable or method has already been declared, or if a method conflicts with a callout. A full list of exceptions that can be thrown by our compiler can be found in `src/compile/exceptionhandling/Exceptions.scala`.

c. IR semantic checking

During the traversal of the high-level IR tree, we not only construct the symbol table structure, but we also run a series of type checks while processing each abstract syntax node. Every node class with 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 = {
        ...
}
```

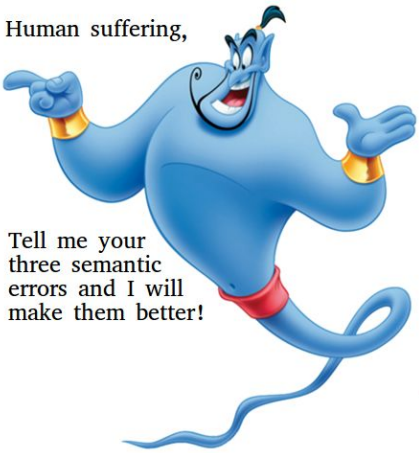
Figure 3. A sample semantic check method signature

An `IrAssignStmt` node has child nodes `IrLocation` and `IrExpr`. Thus, the 'check' function calls `checkIrLocation` and `checkExpr` to ensure that the left and right-hand sides of the assign statement are semantically valid. Then, it additionally checks that the types of location and expression are identical.

d. Error handling

In order for the compiler to return several semantic errors to the user, we implement `ExceptionGenie`, a container (`ExceptionGenie.scala`) that is passed around while traversing the high-level IR tree and stores all semantic errors that we find.

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 errors (theoretically none), also are passed to `ExceptionGenie` and returned to `stderr`.

2. Intermediate Code Generation

a. Introduction

In preparation for generating assembly code, we translate the high-level IR tree into a low-level IR. The translation is performed by traversing the high-level IR tree and recursively converting each node into a sequence of three address codes, defined shortly, which are closer in form to assembly language.

The code can be found in `src/compile/tac/TacGen.scala`, and is structured similarly to our semantic checking code.

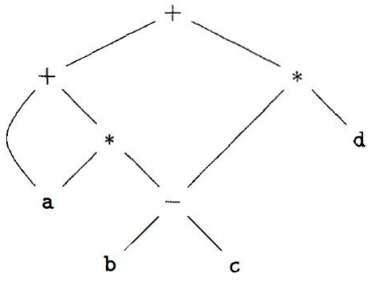
b. Three Address Codes (TACs)

We implement TACs, or pseudo-assembly instructions referencing three or fewer addresses, as our intermediate code format, closely adhering to the design laid out in the Dragon book. We chose this format for simplicity, as it is human-readable and allows us to use only two temporary registers (`%r10`,`%r11`) per operation in assembly.

For an example of a TAC, a binary operation expression may be represented as follows:

```
addr1 = addr2 op addr3
```

Below is an example of the expression `a + a * (b - c) + (b - c) * d` converted to TACs:




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

Figure 4. DAG from Dragon book, p. 359
Our generation is less efficient, as we have no CSE.

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 which represent the result of a single expression, that are inserted into the symbol table structure. After converting the AST into a tree of TACs, we then convert the TACs to assembly in `AsmGen.scala`, which is discussed in Section 3.

c. Symbol tables and Stack allocation

To translate our tree of TAC lists into assembly, we must know each variable's size and offset so that we may allocate adequate space on the stack and obtain correct variable memory references in assembly. To do this, we create a field in each of the four `TypeDescriptor` classes that detail the size and offset of the variable. We also maintain a field in the top level symbol table for each method called `currTotalByteSize`. The field, which represents the max offset from the method's base pointer, allows us to assign the correct offset to any new variable or temporary variable (created from TAC generation) that does not overlap with previously allocated variables. In this way, the symbol tables allow the assembly generator to look up a variable's offset with respect to the current base pointer by querying the variable name in the symbol table structure.

3. Assembly Code Generation

a. General strategy

`AsmGen.scala` contains a function `asmGen(tac: ThreeAddressCode, symbolTable : SymbolTable)` that translates any TAC into a list of corresponding assembly strings, and stores the results in a linked hashmap, of form `LinkedHashMap[Tac, List[String]]`. 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 program which we output to the user.

We designed our TACs such that the mapping between a TAC and its corresponding assembly be relatively straightforward. As such, high-level IR nodes are broken down into several TACs, each of which generally represent basic assembly operations.

For example, a break statement is converted into a `TacGoto(label)` object, which 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.)

b. Managing temporary variables

Our structure for managing temporary variables is `TempVariableGenie`, located in `/src/compile/tac/TempVariableGenie.scala` that is responsible for:

1. Assigning a unique identifier for every TAC.
   Since we are using Scala case classes, two TAC objects of the same class and same arguments would have identical hashes. Thus, in order to store syntactically duplicated TAC-assembly pairs in our `LinkedHashMap`, we must artificially add an additional distinct argument.

2. Produce unique labels for new string literals and control flow. We use a simple system of naming labels with the format `.L###`, where `###` is the number of existing labels, which is monotonically increasing.

3. Produce unique temporary variable names on demand, so that temporary variables do not collide in the symbol table. Similar to above, we name temporary variables with the format `.T###`.

c. Method and callout calls

When traversing a method declaration in the high-level IR tree, we pre-allocate space on the method's stack for the arguments, using the symbol table. Then, when calling a method or callout, we first save the arguments to a list of temporary variables, which is passed as an argument to a method call TAC. During assembly generation, the temporary variables' values are looked up and moved into the appropriate pre-allocated space.

We handle callouts as defined per the `x86` convention. We will not describe the entire convention here, but simply highlight the the two interesting features that we had to deal with: string literals and arrays as callout arguments.

In the high-level IR tree, string literals are scattered throughout, wherever they are passed as arguments to a callout call. However, our assembly construction for string literal constants requires an assembly section beginning with `.section .rodata` and labeling each string literal.

```
        .section        .rodata
.L1:

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

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

Figure 4. A portion of the string literal section generated from `/tests/codegen/input/12-huge.dcf`

Here, we use our `LinkedHashMap` structure to move any `TacStringLiterals` to the front and encapsulate this block of TACs with a `TacStringLiteralStart` and `TacStringLiteralEnd`, which are replaced with `.section .rodata` and `.text`, respectively in assembly generation.

d. Runtime checks

To check for array-out-of-bounds errors, we precede every array index access (`TacArrayLeft` and `TacArrayRight`, in our code) with several 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`.

To check that a value is actually returned when a method's type is not `void`, a jump to a `TacSystemExit(-2)` TAC is appended to every non-void method body. If there is a return statement, then the program will never reach this jump.

e. Miscellaneous features

Initializing local field variables

Decaf locations are initialized to a default value of `0` and `false`, for integers and booleans, respectively, when declared. To implement this, on entering a new scope, we iterate over every local, non-temporary field variable and set the value appropriately. In particular, for arrays, we set the value of every location in the array.

Global variables

We use the very simple `.comm` syntax to define and allocate our global variables at the start of the program (see Figure 5). This way, we can reference any global variable by name in assembly.

```
.comm   b,8,8
.comm   a,8,8
.comm   c,8,8
```

Figure 5. Global variable allocation. `b,8,8` corresponds to the global variable b, of size 8 bytes, and alignment 8.