CSE 304 – Compiler Design
Fall, 2023
Assignment 05 – Intermediate Code Generation

Points: 40
Assigned: Wednesday, 011/15/2023
Due: **Monday, 12/11/2023, at 11:59 PM**

Description:

In this assignment, you will complete intermediate code generation for `Decaf`. Recall the description of `Decaf's` syntax is given in its manual, the AST you built in HW03, and the type checker you built in HW04. This assignment will be based on the AST obtained after type checking and name resolution. From the set of classes in the AST, you will generate intermediate code for a register machine.

As in the past, you will produce a compiler front end that will take a single command-line argument, which is a file name; if that file contains a syntactically valid `Decaf` program, then it will construct an AST and perform type checking and name resolution. If there are no errors detected during the preceding steps, then the compiler will generate code for the target register machine and print that code out (see the section below on printing).

The instructions will fall into the following sections:

1. Abstract Register Machine Specification
2. Storage Allocation
3. Simplifying Assumptions
4. Printing Translated Code
5. Amendments to `Decaf` to Reduce Complexity.
6. Assignment Path
7. Deliverables

1. Abstract Register Machine:

The abstract register machine has:

- **Two** sets of registers:

    1. *Argument Registers*: an unbounded set of registers, ***a0, a1, …, a_n***, for passing arguments to procedures.

       When a procedure is invoked, its arguments should be supplied to ***a0, a1, …, a_n*** (up to the number of arguments, ***n***). The caller should assume that the values in these registers may be updated by the called procedure; so it should not assume that the values in these registers will be preserved. Register ***a0*** will be used to communicate the return value; so the callee should update ***a0*** with the computed return value before returning to the caller.

    2. *Temporary Registers*: an unbounded set of registers, ***t0, t1, …, t_n***, for holding temporary values.

       For procedure calls, the caller should assume that the values in these registers may be updated by the called procedure.

  Any register can hold an integer, a floating point value, or a heap reference.

The machine has instructions to save and restore register values on an internal stack. All arithmetic, comparison, and branch instructions in the machine refer to argument or temporary registers.

In addition to these two sets of registers, the machine has one special-purpose *read-only* register, called the "static area pointer" (**sap**). This register holds the base address of a region for storing all static fields; the static fields themselves are accessed at specific offsets from **sap**. The **sap** register is set by the machine when it loads the program.

- ***Heap***: A heap for statically and dynamically allocated storage. The heap consists of an unbounded sequence of *cells*. Each cell can hold an integer or a floating point value. The machine has instructions to read values from (i.e., load), write values to (i.e., store), and to dynamically allocate heap cells. It also has *directives* to allocate cells for storing static data when the program is loaded.

- ***Control Stack***, ***Data Stack***: The Control stack has return addresses. The Data stack has saved register values. These stacks are not directly manipulated! Call/return changes the Control stack. Register save/store instructions change the Data stack.

Abstract Machine Instructions:

In the following, ***r*** represents an argument register or a temporary register; ***i*** represents a literal integer value; ***f*** represents a literal floating point value; ***L*** represents a symbolic *label*.

Move:

- **`move_immed_i`** *`r, i`* : Set register *r* to integer constant *i*
- **`move_immed_f`** *`r, f`* : Set register *r* to floating point constant *f*
- **`move`** *`r1, r2`* : Set register *r1* to the value of register *r2*

Integer Arithmetic:

The following instructions assume that the source registers contain integer values and perform integer arithmetic operations.

- **`iadd`** *`r1, r2, r3`* : Set register *r1* to the sum *`r2 + r3`*
- **`isub`** *`r1, r2, r3`* : Set register *r1* to the difference *`r2 - r3`*
- **`imul`** *`r1, r2, r3`* : Set register *r1* to the product *`r2 * r3`*
- **`idiv`** *`r1, r2, r3`* : Set register *r1* to the quotient *`r2 / r3`*
- **`imod`** *`r1, r2, r3`* : Set register *r1* to the remainder *`r2 % r3`*
- **`igt`** *`r1, r2, r3`* : Set register *r1* to 1 if *`r2 > r3`*; 0 otherwise
- **`igeq`** *`r1, r2, r3`* : Set register *r1* to 1 if *`r2 >= r3`*; 0 otherwise
- **`ilt`** *`r1, r2, r3`* : Set register *r1* to 1 if *`r2 < r3`*; 0 otherwise
- **`ileq`** *`r1, r2, r3`* : Set register *r1* to 1 `if` *`r2 <= r3`*; 0 otherwise

Floating Point Arithmetic:

Analogous to the above instructions, there is a set of floating point arithmetic instructions (prefixed with 'f' instead of 'i') that assume that the source registers contain floating point values and perform floating point arithmetic operations. The exception is **`imod`** which has no floating point analog.

- **`fadd`** *`r1, r2, r3`* : Set register *r1* to the sum *`r2 + r3`*

- **fsub** *r1, r2, r3* : Set register *r1* to the difference *r2 - r3*
- **fmul** *r1, r2, r3* : Set register *r1* to the product *r2 * r3*
- **fdiv** *r1, r2, r3* : Set register *r1* to the quotient *r2 / r3*
- **fgt** *r1, r2, r3* : Set register *r1* to 1 if *r2 > r3*; 0 otherwise
- **fgeq** *r1, r2, r3* : Set register *r1* to 1 if *r2 >= r3*; 0 otherwise
- **flt** *r1, r2, r3* : Set register *r1* to 1 if *r2 < r3*; 0 otherwise
- **fleq** *r1, r2, r3* : Set register *r1* to 1 if *r2 <= r3*; 0 otherwise

Conversions:

- **ftoi** *r1, r2* : Set register *r1* to an integer value by truncating (taking the floor of) the floating point value in *r2*
- **itof** *r1, r2* : Set register *r1* to a floating point value corresponding to the integer value in *r2*.

Branches:

- **bz** *r1, L* : Branch to the label *L* if the value in *r1* is zero
- **bnz** *r1, L* : Branch to the label *L* if the value in *r1* is not zero
- **jmp** *L* : Branch to the label *L* unconditionally

Heap Manipulation:

- **hload** *r1, r2, r3* : A heap cell is addressed by the pair *r2, r3* : base address in *r2* and offset in *r3*. Register *r1* is set to the value in this cell.
- **hstore** *r1, r2, r3* : A heap cell is addressed by the pair *r1, r2* : base address in *r1* and offset in *r2*. This cell is set to the value in register *r3*.
- **halloc** *r1, r2* : This instruction assumes that *r2* contains an integer value; that value specifies the number of cells to be created on the heap. Register *r1* is set to the base address (i.e., the smallest address) of this group of cells.

Procedure Call and Return:

- **call** *L* : Label *L* is treated as the entry point of a procedure, and that procedure is called. The return address is pushed onto the Control Stack.

  There is a special label attached to the first instruction in each method. This label will be of the form "**M_%s_%d**", where the **%s** argument will be the name of the method, and **%d** will be the unique id of the method. Note that method ids are unique across the entire program. For example, consider a method **fie** with id **12**. Then the first instruction of the method will have label **M_fie_12**, and any call to the method will generate the instruction **call M_fie_12**.

- **ret** : The top of the Control Stack has the return address; the Control Stack is popped and the control transfers to the return address.
- **save** *r* : Value of register *r* is pushed on to the Data Stack.
- **restore** *r* : The Data Stack is popped and register *r* is set to the popped value.

2. Storage Allocation:

Parameters and Locals:

Each formal parameter value is in a distinct argument register. Assume that parameters are associated with argument registers in order. For static methods, the value of the first parameter is in ***a0***, second is in ***a1***, and so on. For instance methods, the implicit object (accessed using "this") is in ***a0***, the first parameter value in ***a1***, the second in ***a2***, and so on.

For the purposes of this homework, you may assume that static methods do not contain references to "this" in the method body.

Temporary registers are used to hold all local variables as well as all intermediate values computed while evaluating an expression.

Objects and Arrays:

Objects are allocated on heap. The fields of an object are laid out in consecutive locations on the heap.

Consider an object ***a*** of class ***C***. Object ***a*** contains a unique location for all non-static fields in class ***C*** as well as all of its superclasses. The object layout is as follows. Let ***B*** be a base class (with no super classes), with two non-static fields ***w*** and ***x***. Then ***w*** and ***x*** are given unique offsets, starting at 0, so that the fields can be accessed from the base address of ***B*** objects. Let the offsets of ***w*** and ***x*** be 0 and 1 respectively. Whether the fields ***w*** and ***x*** are public or not does not matter.

Let ***C*** be derived from ***B*** and have an additional non-static field ***y***. Fields defined in ***C*** are given offsets that follow all fields in ***B*** (regardless of whether they are public or not). Hence ***y*** should be at offset 2.

Moving further, let ***D*** be derived from ***C*** with two new fields, one called ***z***, and another, coincidentally, called ***x***. ***D***'s fields will have offsets following all fields of ***C***. Hence field ***z*** will be at offset 3 and field ***x*** of ***D*** will be at offset 4. For computing the offsets and determining the layout, it does not matter whether a field is private or public.

Static Fields:

Static fields of all classes will be allocated on the heap, in a special area called the "static area". This area will be allocated before the abstract machine program is run, and its allocation is specified by an abstract machine directive "`.static_data n`", where ***n*** is the total number of static fields to allocate.

The abstract machine has a special-purpose register called **sap**, the static area pointer, that holds the reference to the base of static area. All static fields should be allocated in the static area, with each field having a unique offset. It does not matter if the field is public or private for the purposes of allocation.

For instance, let there be two classes ***B***, a base class, and ***C***, a class derived from ***B***. Let ***B*** have two static fields ***x*** and ***y*** defined in it; and ***C*** have two static fields ***y*** and ***z*** defined in it. Then the four fields will be allocated in the static area with directive "`.static_data 4`"; ***x*** and ***y*** of ***B*** will be at offsets 0 and 1, respectively; and ***y*** and ***z*** of ***C*** will be at offsets 2 and 3, respectively.

Continuing with the above example, let all the static fields in the example be public, and consider an assignment in a source program of the form "`C.x = C.y + C.z;`". The following abstract machine code will be a valid translation of the assignment:

```
move_immed_i t0, 2     # Offset of y in C
hload t1, sap, t0      # t1 = C.y
move_immed_i t2, 3     # Offset of z in C
hload t3, sap, t0      # t3 = C.y
iadd t4, t1, t3
move_immed_i t5, 0     # Offset of x in C (which is x in B)
hstore sap, t5, t4
```

In the above, "#..." is used to represent comments in order to explain the code.

Constructors:

Each constructor will be compiled into an initializer function.

For instance, consider a constructor of the form **`A(int i)`**. This constructor will be compiled to an initializer function that takes a reference to a newly created object as ***a0***, the value of formal parameter ***i*** as ***a1***, executes the code corresponding to the body of the constructor. The first instruction in this sequence will be labeled as **`C_%d`**, where **`%d`** is the unique id of the constructor. Recall that constructor ids are unique across the whole program. Initializer functions return nothing.

Code for new object creation will have two parts. The first part should create an object of the correct size on the heap (using the **`halloc`** instruction). The second should call the initializer method with appropriate arguments. For instance, let the instances of the above class ***A*** have **5** instance fields (note: this includes non-static fields of superclasses, if any). Let the id of the matching constructor be **7**. Then the expression "**`new A(2)`**" may be translated to the following sequence of machine instructions:

```
move_immed t0, 5
halloc t0, 5           # allocate 5 cells for object
move_immed_i t1, 2     # Actual argument
save  a0               # save current argument registers
save  a1
save  t0               # save reference to new object
move   a0, t0
move   a1, t1
call   C_7             # call initializer function
restore t0             # restore original register values
restore a1
restore a0
```

Note: the above code is to illustrate the use of the initializer function; the saves and restores may be necessary only in certain contexts.

3. Simplifying Assumptions:

For this HW:

- assume that your programs contain no string constants.
- Assume that each method has a "return" statement on every control flow path.
- Assume that constructors do not have a "return" statement.
- Assume that static methods do not contains references to "this" or "super" in their method bodies.

4. Printing the Abstract Machine Instructions:

The output of your compiler will be a text file filled with the abstract machine program. The program shall be printed to the file as follows.

Print one instruction per line. Each instruction will be its opcode, followed by the sequence of arguments. Include at least one space between opcode and the arguments. Arguments shall be comma separated. The entire instruction--- opcode and all arguments--- should be placed in a single line; an instruction shall not spill over two lines.

Labels of instructions, if any, shall be printed on a line of their own. Label of an instruction shall be followed by a ":". For instance, an "iadd" instruction with label "L12" will be printed as:

```
L12:
   iadd t3, t1, t2
```

If an instruction has more than one label, each label shall be printed on a line of their own.

Abstract machine programs may have single-line comments, starting with "#" and ending at the end of line. Comments may appear at end of instructions or as a line of their own.

Program:

The translated program will be:

1. A set of functions corresponding to methods and initializer functions
2. A single line static data directive.

Each function will begin with the label of the function, followed by the instructions in the function body. Functions may be separated by one or more empty lines for readability.

5. Amendments to `Decaf` to Reduce Complexity:

We are generating code for a subset of the `Decaf` language. It does not include the following:

- Arrays: there are no arrays in our subset of `Decaf`
- Implicit Field Accesses: all field accesses in our subset of `Decaf` will specify the object explicitly
- Implicit Method Calls: all method calls in our subset of `Decaf` will specify the object explicitly
- No Overloaded Methods/Constructors: Assume that in each class, there is at most one constructor and at most one method with a given name. Also assume that the names of methods defined in a class are distinct from method names in its super classes. *You do not need to check for these constraints. Your type checker can safely assume that they are true.*

6. Assignment Path:

1. Start with a working AST builder and type checker.
2. Define an interface to create abstract machine instructions and construct the output program.
3. Assume a single class with no fields (!!), no method invocations, constructors, or field accesses. Write a code generator that works on this restricted subset of programs. You may want to work on even smaller subsets than these.
4. Proceed by adding field accesses first, then method invocations, and then finally constructors.

7. Deliverables:

Your AST checker for `Decaf` should be organized into the following files:

1. **Lexer:** `decaf_lexer.py` – PLY/lex scanner specification file. (*unchanged from A02*)

2. **Parser:** `decaf_parser.py` – PLY/yacc parser specification file. (*unchanged from A03*)

3. **AST:** `decaf_ast.py` – table and class definitions for `Decaf`'s AST. (*unmodified from A04 after adding type-related fields and definitions*)

4. **Type Checker:** `decaf_typecheck.py` – definitions for evaluating the type constraints and for name resolution.

    - **Note:** This may be folded into decaf_ast.py as long as readability is maintained.

5. **Code Generator:** `decaf_codegen.py` – definitions for generating code

    - **Note:** This may be folded into decaf_ast.py as long as readability is maintained.

6. **Abstract Machine:** `decaf_absmc.py` – definitions for the abstract machine and for manipulating abstract programs (e.g., printing)

7. **Main:** `decaf_compiler.py` – containing the main python function that ties the modules together, takes the command-line argument of the input file name, and processes the `Decaf` program in that file.

    The translated machine instructions will be written to a file with the same base name as the input file, but with a ".ami" extension.

    Note the change to the file name of Main: We are no longer checking; we are now compiling.

8. **Documentation:** `README.txt` which documents the contents of all the other files.

We will be looking for a submission with the six files specified above, with the specified names. Deviating from these specifications may entail loss of points.