High level strategies:

We can either print the jasmin instructions directly, or we can kind of make an intermediate representation in terms of ocaml objects, and then convert that internal representation to direct jvm code. The second approach (using an IR) is nice and would allow us to do all kinds of things like late binding of variables to locals array, computing the stack / locals limit after we generate all the statements for a method, rearrange / delete methods for optimizations etc, and also delegate the job of printing with proper tabs and labels to the converter of IR to Jasmin. For this milestone, the jasmin code is directly printed as what we did in prettyPrinter, just so we get an idea which parts are more difficult. The final code generation will likely be done according to the following design.

We decided to implement, for the last milestone, a jasmin ast. This ast will allow us to do a second pass of the code to be generated, so we can appropriately calculate the size of the local stack as well as the amount of variables needed. Jasmin assembler code is, for the moment, being directly printed at code generation in a way similar to that of the pretty printer.

All top level variable declarations become public static fields
   - Variable initialization will go into the static initialization block

All top level type declarations except for structs will be omitted.
   -There is no reason to preserve custom type information in runtime, as we do not support reflection type widening / narrowing. The utility of custom types ceases after the type checker phase. For example, If we have

```
type point int;
var x point = 23;
```

for all intents and purposes we will treat x as an int.

   - Struct types will need special treatment, and their type information will be relevant at runtime. We can generate a class for every struct or we may be able to implement it using a hashmap.

- Slice types will need special handling. We can write our own slice class in java and compile it to jasmin. We can also use ArrayList.

Statements:

Operand stack height before and after execution each statement should be zero.
Computing the number of local variables: See the variable declaration section.

- Empty statement

- generate no-op for now. If possible get rid of it later.

(We need to find out if no-op can simply be skipped, as there might be issues with how we handle labels.)

- Expression statement

- Evaluate the expression. This should increase the stack height by one (see the expressions section.) Pop the expression off stack to preserve stack height invariant.

- Void function calls need some thought. We can either evaluate that expression in a special way, or add a special “void” value on the stack (this could be java null) and then pop it off.

- Assignment Statement

- The lvalue is an expression, whose type is cached in the ast node, so you should be able to pattern match on the type and make appropriate decisions.

- Different instruction depending on whether variable is a top level variable or not (if top level then it’s just a static field of the class. Otherwise it’s probably on the locals stack)

- Type Declaration Block

- Should be handled pretty much like top level type declarations.

- If we *are* generating separate classes for each struct, we need to have the sometime indicating the scope of the declaration in the type name to ensure uniqueness. For example, the following code defines two very different point types.

```
type point struct {
  x int;
  y int;
}

func main() {
  type point struct {
```

```
    x int;
    y int;
  }
}
```

as evidenced here: https://play.golang.org/p/Sj3RzDAf9o

- Short Variable Declaration

- A combination of variable declaration and assignment. Similar design principles should apply.

- Variable Declaration

- There needs to be a mapping from local variables to the *locals* array on the stack. As we encounter new variable declarations, we need to add them to the mapping. Since a function body can have child scopes containing different variable of the same name (just like we had different types of the same name above, we need to be careful that we do not choose the name of string of the variable as the key of our mapping. A symbol table entry should be a good choice - but we will have to check if it's hashable by default. The number of local variables in a function/method can then just be the size of this map.

- If it contains an initialization, the same design principles as assignments apply.

- Print Statement

- Generate the right print statement depending on the type.

- If Statement

- This expects a boolean in the stack (int 1 or 0). It uses a single label to skip to the else portion when required. Labels should be chosen carefully - each of the labels used for an if statement should contain:

    1. some indication that the label was generated as part of if statement
    2. a number that differentiates the labels for this if statement from the labels of other if statements. (For example the labels can look like 'IfStart_0', 'ElseBlock_0' etc.)

- Else block no longer needs to be optional (if no else, then else block just points to the right code that follows)

    -Currently, the labels are generic (Label_X), and implementation of else labels seems unnecessary complicated, given the nesting of else ifs.

- Switch statement

- Needs some thought. Use table switch / lookupswitch maybe.

- For loop

- Needs to handle label jumping carefully.

- Break statement

- Go to end of for loop.

  (This statement is troubling because we need to know which for loop you were in to jump to the right place.)

- Continue statement

- Go to beginning of loop. Should be handled similarly to break statement.

- Block statement

- It starts a new scope and have different variables with same name.

- Return Statement

- Evaluate expression. Return.

- Return expression is optional (i.e. obviously no expression to return in functions of void return type.)

- Always generate a return instruction even for void functions because all methods in java bytecode require a return.

Expressions:

Evaluating each expression should increase the stack height by exactly one. The special case is void function call, and a null value could be added to the stack anyway to maintain consistency. The typechecker ensures the result of void function call never gets used.

Computing stack size limit:

unary expressions:

max_stack of child

binary expressions:

max of ((max_stack of left child), (max_stack of right child) + 1)

other expressions:

to be thought about

Lots of statements also require extra stack slots (e.g. print statement pushes System.out on the stack).

For milestone 3 we just put a big number (e.g. 100) for each method's stack limit. When we're done implementing all the statements and expressions, we will know for sure how to calculate this stack limit.

- IdExp:
- If it’s a local variable, push it to stack from the locals array of the method. We will have to use the mapping from variables to locals array elements (see the variable declaration section of statements).
- If id is a variable declared in the top level, get a static field.
- If id is not a variable (e.g. a type or something) scream loudly. The type checker should make sure this never happens.

- Literal Expression

- load constant literal. Note that whenever possible, bipush/sipush is more efficient than ldc

- may finally need to convert our ints from texts to decimals. Test if jvm supports octal/hexadecimal literals)

- Unary Expression

    - If any unary operator is not supported by the jvm write a static function that does the required work. We can even put a collection of all these helper functions in a separate class that will always be included when we generate code. This can be a kind of “runtime environment” for our code.

- Binary expressions

- If any binary operator is not supported out of the box, do a similar thing as unary expressions.

- Function call

- invokestatic! All our functions are static. Void function calls need to be dealt with care.

- Some function call nodes could be type cast nodes in disguise. See section of type cast expressions.

- Append

- We can just write a Slice class is Java to manage this. Or we could also just use ArrayLists, since we seem to have access to the whole java standard library from the jvm.

- TypeCast

- If we do not preserve custom types at runtime. We don’t need to do anything apart from evaluating the argument expression.

- ***Caution:*** we do not change the ast node from function calls to type casts in case of type casting to custom type. We will have to lookup at the function name id’s symbol table entry (the symbol table entry is bundled with the identifier) to determine for sure if the id has type function of type custom type / new type, and decide based on that.

- Index Expression

- If array, then the proper aload instruction

- If slice, then proper slice instructions

- Any other type should have been rejected by the typechecker.

- Select Expressions

- What to do here really depends on how we decide to implement structs. If we define a class for each struct, we can use all kinds of getfield instructions to get non-function fields, and invokevirtual to use methods.