Victoria University of Wellington
School of Engineering and Computer Science

SWEN430: Software Development
Assignment 4 — Machine Code Generation

Due: Friday 1st October @ Mid-night

Overview

This project is concerned with *x86 machine code generation.* The x86 machine architecture presents a number of interesting challenges, such as limited registers with non-uniform, overlapping representations. Furthermore, generating machine code itself is more challenging than for e.g. JVM bytecode. To help with this, the `jx86` library is provided for writing x86 assembly language (which e.g. can then be compiled with `gcc`). You can download this library from the SWEN430 homepage, and you can view the library source at: `http://github.com/DavePearce/jx86`.

You can download the latest version of the compiler from the SWEN430 homepage, along with the supplementary code associated with this assignment. *We recommend that you start from a fresh version of the compiler, rather than any extensions from previous assignments.* Initially, you should find a large number of all tests currently fail and, furthermore, that a number are marked as *ignored.* For the purposes of this assignment, those which are ignored are not being considered. However, the remainder should all pass when the assignment is complete.

NOTE: you should not attempt to modify the `jx86` library in anyway, and your program should compile against the stock version provided on the course homepage.

NOTE: the supplementary code includes a *runtime library*, `whilelang/runtime/runtime.c`, which is written in the C language. You are not permitted to modify this library.

NOTE: you'll need to have `gcc` installed on your machine in order to run the tests. This is because it is needed to compiler the assembly generated from `jx86` into an executable. On MacOS, you can install *Xcode Command Line Tools.* On windows, you can install *MinGW* or *Cygwin.*

NOTE: when running the tests on something other than Linux `X64_64`, you'll need to manually set the `X86Tests.TARGET` variable.

NOTE: you can inspect the output produced for each test case by looking at the `*.s` files in the `tests/valid` directory.

NOTE: various websites allow you to run `gcc` and look at the assembly output (e.g. `gcc.godbolt.org`).

NOTE: to get more debugging output, you can call `fprintf(stderr,...)` from within the `runtime.c` library. You can either add a new method for this and call it via `makeExternalMethodCall()`, or insert it into one of the existing methods (e.g. `intcmp`).

1 Fundamentals (20%)

To help get started, a number of relatively straightforward parts of the WHILE language require implementing:

- **Subtraction / Multiplication**. The implementation for these is currently missing from `translateArithmeticOperator()`, though should be very similar to that for addition.
- **Logical Conjunction**. The implementation for this is missing from `translateCondition()`
- **Unary Expressions**. The implementation of arithmetic negation and logical not operators is currently missing from `translateUNary()`.

Having successfully implemented these, you should find all tests in `X86Tests.part_1` now pass.

2 Control Flow (10%)

The translation of `while`, `do-while` and `for` loops and `switch` statements is not fully implemented in the compiler. The following illustrates a simple example which counts up to a given number `n`:

```
int count(int n) {
  int i = 0;
  while(i < n) { i = i + 1; }
  return i;
}
```

The above program should compile into machine code which resembles the following:

```
wl_count:
    pushq %rbp
    movq %rsp, %rbp
    subq 16, %rsp
    movq 0, %rbx          //
    movq %rbx, -8(%rbp)   // i = 0
label1:
    movq -8(%rbp), %rbx   //
    movq 24(%rbp), %rcx   //
    cmpq %rcx, %rbx       //
    jge label2            // i >= n
    movq -8(%rbp), %rbx   //
    movq 1, %rcx          //
    addq %rcx, %rbx       //
    movq %rbx, -8(%rbp)   // i = i + 1
    jmp label1
label2:
    movq -8(%rbp), %rbx   //
    movq %rbx, 16(%rbp)   //
    jmp label0            // return i
label0:
    movq %rbp, %rsp
    popq %rbp
    ret
```

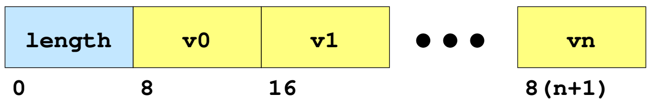
In the above, we have annotated the x86 assembly language to indicate which statements in the While program each instruction corresponds with. **In implementing While loops, you should find the existing implementation of `for` loops to be helpful**. The translation of `break` and `continue` statements is one challenging aspect. The following illustrates a simple example:

```
switch(n) {
  case 0:
    break;
  default:
    n = n + 2;
}
```

To complete this part, break and continue statements should be implemented as *unconditional branching instructions* (i.e. `jmp`). You will need to modify the `Context` to include the necessary targets for `break` and `continue`. Having successfully implemented these, you should find all tests in `X86Tests.part_2` now pass.

3 Primitive Compounds (40%)

Implementing compound data types in While on the x86 architecture is challenging. For now, we restrict ourselves to *primitive* records and arrays (e.g. `int[]`, `{int f}`, etc). Since arrays correspond to *dynamically sized* structures, they cannot be allocated on the stack. Instead, we allocate them on the heap and the method `allocateSpaceOnHeap()` is provided for this. **For simplicity, we also implement records on the heap (even though this is not strictly necessary).** All compound data types are given a uniform representation where the first word is reserved for holding its *length*:



Following the length field, we have n slots used for storing array elements or record fields. On `x86_64`, each word occupies 8 byte of memory. Hence, the nth field is located $8(n+1)$ bytes from the base. The following illustrates a simple example:

```
int f({int x, int y} r) { return record.y; }
```

This shows an single read of the field `y` from variable `r`. The above program should compile into machine code which resembles the following:

```
wl_f:
    pushq %rbp
    movq %rsp, %rbp
    movq 24(%rbp), %rax // load record pointer into rax
    movq 16(%rax), %rax // load offset 16 from record
    movq %rax, 16(%rbp) // store result into return value
    jmp label0
label0:
    movq %rbp, %rsp
    popq %rbp
    ret
```

We have provided a mostly complete implementation for records, and this provides an important reference. In addition, the implementation for array accesses is also provided. To proceed, we recommend the following steps:

1. **Array Initialisers**. These are necessary to perform any operations on arrays. An appropriate amount of memory must be allocated, and then every element in the initialiser evaluated and assigned into the array. The method `compoundInitialiser()` is provided to help.

2. **Array Length**. This is a simple operation which should read out the length of an array from its location in the underlying data structure.

3. **Array Generators**. At this point, implementing array generators should be relatively straightforward. To help with this, a method `intfill()` is provided in the runtime library. Note that the array length must be calculated dynamically.

4. **Compound Equality**. In order for many assertions to pass, you will need to implement compoound equality for arrays and records. To help with this a method `compoundEquality()` is provided.

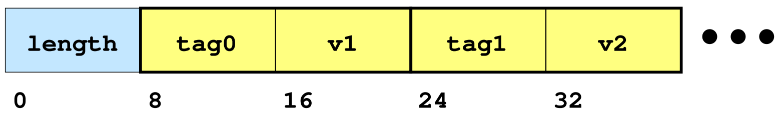
5. **Compound Assignment**. Assigning an element into an compound data type (e.g. record or array) is made challenging because the semantics of WHILE dictate that arrays have *value semantics*. Therefore, when assigning into an array, you should *clone the entire array*. To help with this a method `compoundCopy()` is provided.

NOTE: Although arrays are allocated on the heap, we will not attempt to deallocate them. This is challenging in a language like WHILE which has no explicit deallocation operator. Instead, one could use *reference counting* — but this is beyond the scope of this assignment!

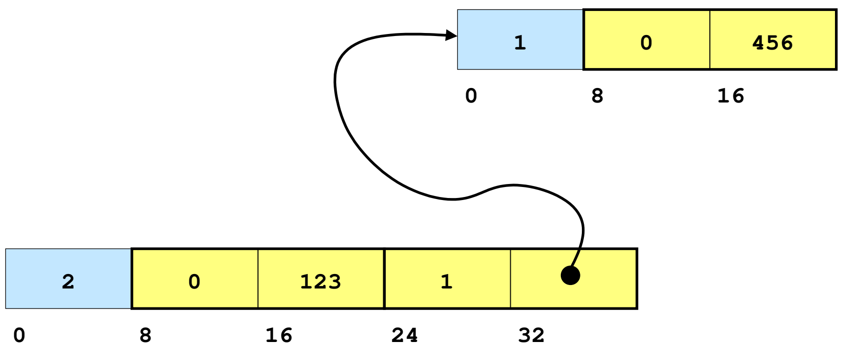
Having successfully implemented these, you should find all tests in `X86Tests.part_3` now pass.

4 Nested Compounds (30%)

Having implemented primitive compounds the next step is upgrade your implementation to handle *nested* compounds. To make this work requires a more complex representation for compound data types. Specifically, you must update your implementation to support the following representation:



Here, we see that each slot is now represented using *two* words: one for the *tag* field; and one for the *payload* field. The tag is used to indicate whether the payload represents a primitive or compound type. For example, the literal `{id:123, data:[456]}` is represented like this:



To help implement this part of the assignment, additional methods in `runtime.c` are provided: `objcmp()`, `objcpy()` and `objfill()`. These replace their primitive counterparts (e.g. `intcmp()`, etc). Having successfully implemented these, you should find all tests in `X86Tests.part_4` now pass.

Submission

Your assignment solution should be submitted electronically via the *online submission system*, linked from the course homepage. You must ensure your submission meets the following requirements (which are needed for the automatic marking script):

1. **Your submission is packaged into a jar file, including the source code**. *Note, the jar file does not need to be executable.* See the following Eclipse tutorials for more on this:

   `http://ecs.victoria.ac.nz/Support/TechNoteEclipseTutorials`

2. **The names of all classes, methods and packages remain unchanged**. That is, you may add new classes and/or new methods and you may modify the body of existing methods. However, you may not change the name of any existing class, method or package. *This is to ensure the automatic marking script can test your code.*

3. **The testing mechanism supplied with the assignment remain unchanged.** Specifically, you cannot alter the way in which your code is tested as the marking script relies on this. However, this does not prohibit you from adding new tests. *This is to ensure the automatic marking script can test your code.*

4. **You have removed any debugging code that produces output, or otherwise affects the computation.** *This ensures the output seen by the automatic marking script does not include spurious information.*

Note: Failure to meet these requirements could result in your submission being reject by the submission system and/or zero marks being awarded.

Assessment

Marks for this assignment will be awarded based on the number of passing tests, as determined by the *automated marking system*. **NOTE:** the test cases used for marking will differ from those in the supplementary code provided for this assignment. However, the intention of these hidden tests is that they are largely equivalent to those provided.