Introduction to Language Theory and Compilation Exercises

Session 8: Code generation

Reminders

Identifier

An identifier can be either a named value (**[a-zA-Z$._][a-zA-Z$._0-9]***) or a unamed value (**[0-9]+**) or a constant (i.g. *null*). The named value are used as a variable or a function unlike an unamed value which is a temporary value (like an intermediate calculation step or a storage of a value). The scope of an identifier is determined by its first character where `@` means *global* and `%` means *local*.

You also have to determine the type of your identifier: `i32` for integer, `double` for real, `label` represents code labels and `void` does not represent any value and has no size. You can also use array: `array [<# elements> x <elementtype>`.

Function

A function respect the following signature where *entry:* is the required label.

```
define <ResultType> @<FunctionName> ([argument list]){
   entry:
      ...
      ret <type> <value>
}
```

For instance

```
define i32 @add1(i32 %a, i32 %b){
   entry:
      %varTmp1 = add i32 %a, %b
      ret i32 %varTmp1
}
```

and you can call this function with its signature

```
%1 = call i32 @add1(i32 %myFirstInt, i32 %mySecondInt)
```

Operation

For operations on numbers, a float version is available with the prefix `f`. For instance `fadd` for `add`.

The binary operations are: add, sub, mul, sdiv (the prefix `s` for signed and `u` for unsigned), srem (remainder of a division, s/u for the sign). You can also use the bitwise operations: and, or, xor.

In order to use named variable and keep data in memory, you can allocate (a garbage collector will auto-clean the memory) with the operation *allocate*. The other operations are *store* which stores a value into a pointer made by *allocate* and *load* which loads a value from a pointer.

For instance

```
%a = allocate i32      ; we allocate an integer called 'a'
store i32 1, i32* %a   ; we store the value 1 into the pointer 'a'
%1 = load i32* %a      ; we load the value pointed by 'a'
%2 = add i32 1,1       ; we put %1+1 into a unamed variable
```

Input/Output

You can use functions from the standard library (stdlib). The usual functions of input/output are

```
int getchar(void);
int putchar(int c);
```

The declaration of these function in LLVM IR is

```
declare i32 @getchar()
declare i32 @putchar(i32);
```

Condition

A condition is characterized by a test and a jump. The jump is made by calling one of the two signature of the *br* operator:

Unconditional jump br label %myLabel

Conditional jump br i1 %boolValue, label %myLabelIfTrue, label %myLabelIfFalse

The boolean value can be evaluated by using one of boolean operators: eq ($==$), ne ($\neq$), sgt (s/u for sign, $>$), sge (s/u for sign, $\geq$), slt (s/u for sign, $<$), sle (s/u for sign, $\leq$) and by casting this evaluation into a boolean value with *icmp*.

For instance

```
def i32 compareTo(i32 %a, i32 %b){
   entry:
      %cond = icmp slt i32 %a,%b
      br i1 %cond, label %lower, label %greaterORequals
   lower:
      ret i32 -1
   greaterORequals:
      %1 = sub i32 %a,%b
      ret i32 %1
}
```

Practical aspect

For more information, go to `http://llvm.org/docs/LangRef.html`. In order to run the interpreter, you have to produce byte code and then interpret it:

```
llvm-as code-source.ll -o=code-source.bc
        lli code-source.bc
```

Exercises

Ex. 1. Assuming that you have defined these functions:

```
define i32 @readInt()
define void @println(i32 %value)
```

Write a LLVM function that computes and outputs the value of:

$$(3+x)*(9-y)$$

where x is a value read on input and y is a global variable.

Ex. 2. Assuming that you have defined these functions:

```
define i32 @readInt()
define void @println(i32 %value)
```

Write a function that:

- Allocates memory for two variables we will call a and b
- Initializes a and b with values read on input
- Adds 5 to a
- Divides b by 2
- If $a > b$, output a, else output b

Ex. 3. Define this function

```
define i32 @readInt()
```

which reads an integer of the form `[0-9]+` in base 10 by using

```
; External declaration of the getchar function
declare i32 @getchar()
```

Remember that the character 0 is the ASCII code 48.

Ex. 4. Translate this C program in LLVM IR.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int getNumber(void){
   return rand() % 100;
}

int main(void){
   //initialization of randomizer
   srand(time(NULL));
   int guess = getNumber();
   int i;
   for(i=0;i<5;i++){
      int try;
      scanf("%d",&try);
      if(try > guess){//greater
         putchar(45);//-
         putchar(10);//\n
      }else if(try < guess){//lower
         putchar(43);//+
         putchar(10);//\n
      }else{//success
         putchar(79);//O
         putchar(75);//K
         putchar(10);//\n
         return 0;
      }
   }
   //failure
   putchar(75);//K
   putchar(79);//O
   putchar(10);//\n
   return 0;
}
```