Introduction to Language Theory and Compilation Exercises

Computer Session 2: Code generation

Introduction

LLVM IR (Intermediary Representation) is an Assembly-like low-level language. However, it is hardware-independant, strongly typed, and abstracts away register assignment.

Getting started Go to http://releases.llvm.org/download.html and get the "Clang for x86_64 Ubuntu 16.04" Pre-Build Binaries. You can then execute LLVM from the `bin/` directory.

Identifiers

An identifier can be either a named value (`[a-zA-Z$._][a-zA-Z$._0-9]*`) or an unnamed value (`[0-9]+`), prepended by its scope: `@` for *global* and `%` for *local*. Constants (e.g. *null*) are also allowed. A named value is used as a variable or a function whereas an unnamed value is a temporary value (e.g. an intermediate calculation step).

You have to specify 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 arrays: `[<# elements> x <elementtype>]`.

You can declare a variable to be constant by adding the `constant` keyword, e.g. `@x = constant i32 42`

Functions

A function has the following signature (you can add `entry:` at the beginning of the function body to give an explicit entry point to which you can jump):

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

For instance, the following function computes the sum of its two arguments:

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

You can then call it with its signature:

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

Operations

Operations on numbers are available for both integers (`i32`) and reals (`double`). In the latter case, simply add the prefix `f`[1], e.g. `fadd` adds reals.

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

[1] `f` stands for "`float`", although now `double` are preferred.

In order to use named variables and keep data in memory, you can allocate memory areas (a garbage collector will then automatically clean the memory) with the `alloca` operator. The other operations are `store` which stores a value into a pointer made by `alloca` and `load` which loads a value from a pointer.

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

Input/Output

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

```
int getchar(void);     // gets one character from stdin
int putchar(int c);    // writes one character to stdout
```

The declaration of these function in LLVM IR is

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

Conditionals

A condition is characterized by a test and a jump. The jump is made by calling one of the two signatures 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 the following 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, the following function compares a with b and outputs -1 if $a < b$, $a - b$ otherwise.

```
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    # "as" stands for "assembly"
lli code-source.bc                          # "i"  stands for "interpreter"
```

Exercises

Ex. 1. Using the following functions (provided on Université Virtuelle):

```
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. Using the following functions (provided on Université Virtuelle):

```
define i32 @readInt(){...}
define void @println(i32 %value){...}
```

Write a function that:

- Allocates memory for two integer 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. a. Now, implement this function

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

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

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

Remember that the character 0 has ASCII code 48, and the others are put in order.

b. Test it on your code of Exercises 1 and 2.

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;
}
```