Introduction to Language Theory and Compilation: Exercises

Session 8: Code generation

ULB

Code generation

Two questions:

1. What code to generate? For which architecture?
   - Low Level Virtual Machine Intermediate Representation (LLVM IR) for an abstract machine because it is intermediate code (see `http://www.llvm.org`)
2. At what point should the code be generated? How can this be specified formally?
   - Attribute grammars

Abstract machine

In this course, we'll use LLVM IR as target language, which acts as a kind of assembly language for an abstract machine.

- LLVM IR is a intermediary language developed (originally) for a Virtual Machine (LLVM), the file extension is `*.ll`
- LLVM IR is can be optimized and compiled to a specific architecture by using LLVM tools.

For the complete documentation, go to
`http://llvm.org/docs/LangRef.html`

LLVM IR – generals

First of all, we present a simplified LLVM because we avoid objects and visibility.

We also avoid to call specific architecture commands like commands to access on registers.

Inline comments starts with a ';' until the end of line.

LLVM IR – identifier

Two types

Global identifiers (*functions*, *global variables*) begin with the **'@'** character

Local identifiers (*register names*, *types*) begin with the **'%'** character

LLVM IR – identifier

Three formats

Named values[a] '[%@][a-zA-Z$._][a-zA-Z$._0-9]*'

Unamed values '[%@]integer' looks like '%0'. It is temporaries values and they are numbered sequentially (using a per-function incrementing counter, starting with 0).

Constants classical form for numeric ('null' for pointers, 'true'/'false' for boolean)

[a] other characters can be surrounded with quotes and special characters may be escaped using '\xx' where 'xx' is the hexadecimal ASCII code.

LLVM IR – primitive types

i*N*	is an integer defined on *N* bits (i.g. *i*1 for boolean, *i*32 for classic integer)
half	16-bit floating point value
float	32-bit floating point value
double	64-bit floating point value
void	does not represent any value and has no size
label	represents code labels
array	[<# elements> x <elementtype>]

LLVM IR – function

Syntax

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

Example: *sum*(*a*, *b*)

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

Simple usage

as an **Interpreter**

1. Produce the byte-code file

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

2. Run the interpreter

```
lli code-source.bc
```

as an **Compiler**

1. Produce the byte-code file

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

2. Run the compiler

```
llc code-source.bc -o=code-source.bin
```

3. Run your program

```
./code-source.bin
```

ULB

Exercise 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.

Operations

The list of all operations are available on `http://llvm.org/docs/LangRef.html`.

The most useful subset is explained in the remainder section of the statement sheet.

Solution for exercise 1

```
@y = constant i32 3

define i32 @exercise1(){
    entry:
        %x = call i32 @readInt()
        %0 = load i32* @y
        %1 = add i32 %x, 3
        %2 = sub i32 9, %0
        %3 = mul i32 %1, %2
        ret i32 %3
}
```

Exercise 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

Solution for exercise 2

```
define void @exercise2(){
    entry:
        ; item 1
         %a = alloca i32
         %b = alloca i32
        ; item 2
        %0 = call i32 @readInt()
        %1 = call i32 @readInt()
        store i32 %0, i32* %a
        store i32 %0, i32* %b
        ; item 3
        %2 = load i32* %a
        %3 = add i32 5, %2
        store i32 %3, i32* %a
        ; item 4
        %4 = load i32* %b
        %5 = sdiv i32 %4, 2
        store i32 %5, i32* %b
```

Solution for exercise 2 (ctd.)

```
    ; item 5
    %6 = load i32* %a
    %7 = load i32* %b
    %8 = icmp sgt i32 %6, %7
    br i1 %8, label %output\_a, label %output\_b
output\_a:
    %9 = load i32* %a
    call void @println(i32 %9)
    ret void
output\_b:
    %10 = load i32* %b
    call void @println(i32 %10)
    ret void
}
```

Exercise 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.

Solution for exercise 3

```
define i32 @readInt() {
    entry:                                          ; create
        variables
        %res   = alloca i32
        %digit = alloca i32
        store i32 0, i32* %res
        br label %read
    read:                                           ;
        read a digit
        %0          = call i32 @getchar()
        %1 = sub i32 %0, 48
        store i32 %1, i32* %digit
        %2 = icmp ne i32 %0, 10
        br i1 %2, label %save, label %exit
```

Solution for exercise 3 (ctd.)

```
    save:                                                    ;
        res<-res*10+digit
        %3 = load i32* %res
        %4 = load i32* %digit
        %5 = mul i32 %3, 10
        %6 = add i32 %5, %4
        store i32 %6, i32* %res
        br label %read
    exit:                                                    ;
        return res
        %7 = load i32* %res
        ret i32 %7
}
```

Exercise 4

Translate this C program in LLVM IR.

```c
#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
```

Exercise 4 (ctd.)

```
        }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;
}
```

Solution for exercise 4 (ctd.)

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

; External declaration for ramdomization
declare i32 @rand()
declare i32 @time(i32*)
declare void @srand(i32)

define i32 @getNumber(){
    entry:
        %0 = call i32 @rand()
        %1 = srem i32 %0, 100
        ret i32 %1
}
```

Solution for exercise 4 (ctd.)

```
define i32 @main() {
    ;initialization of randomizer
    %1 = call i32 @time(i32* null)
    call void @srand(i32 %1)
    %2 = call i32 @getNumber()
    %guess = alloca i32
    store i32 %2, i32* %guess
    %i = alloca i32
    store i32 0, i32* %i
    br label %for_loop
  for_loop:
        %3 = load i32* %i
        %4 = icmp slt i32 %3, 5
        br i1 %4, label %inner_for, label %after_for
```

Solution for exercise 4 (ctd.)

```
inner_for:
    %try = alloca i32
      %5 = call i32 @readInt()
      store i32 %5, i32* %try
    %6 = load i32* %try
    %7 = load i32* %guess
    %8 = icmp sgt i32 %6, %7
    br i1 %8, label %first_cond, label %test_next
test_next:
      %9 = load i32* %try
    %10 = load i32* %guess
    %11 = icmp slt i32 %9, %10
    br i1 %11, label %second_cond, label %else
first_cond:
      ;greater
      call i32 @putchar(i32 45);-
    call i32 @putchar(i32 10);\n
    br label %end_for
```

Solution for exercise 4 (ctd.)

```
second_cond:
    ;lower
    call i32 @putchar(i32 43);+
    call i32 @putchar(i32 10);\n
    br label %end_for
else:
    ;success
    call i32 @putchar(i32 79);O
    call i32 @putchar(i32 75);K
    call i32 @putchar(i32 10);\n
    ret i32 0
    br label %end_for ; never reached
end_for:
      %20 = load i32* %i
      %21 = add i32 %20, 1
      store i32 %21, i32* %i
      br label %for_loop
```

Solution for exercise 4 (ctd.)

```
    after_for:
        ;failure
        call i32 @putchar(i32 75);K
        call i32 @putchar(i32 79);O
        call i32 @putchar(i32 10);\n
        ret i32 0
}
```