ULB
INFO-F403 - Introduction to language theory and compiling
Introduction to language theory and compiling

Bastogne Jérôme,
Hereman Nicolas

Academic year 2015-2016 - December 16, 2015

Part 3 - Code Generation

Fixing part 2 mistakes

There was a mistake in the modified grammar rules in the second part of the project. The rule 21 was :

```
[21]    <Factor>         ->       - <ExprArith>
```

which was wrong because it didn't take into account the priority of the "-" operation. An arithmetic expression like $-5+5$ would have been parsed like $-(5+5)$ instead of $(-5)+5$. So it was corrected that way :

```
[21]    <Factor>         ->       - <Factor>
```

Hypothesis on the language

As everything isn't detailed in the statement, some hypothesis have to be made. At first, there is no variable type in this language, so we took the decision to consider them all like integers. By that fact, the division also returns an integer (i.e. $5 \div 2 = 2$).

The second hypothesis concerns the "for" operation. It isn't precised if the condition to enter in the loop is that the variable has to be lower or equal than the objective or if the variable has to be strictly lower. In our implementation, we consider that the variable has to be strictly lower than the objective.

A function to read integers

As there is a read operation in the language, a function to read integers from the standard input had to be implemented. Here is the llvm code which does this :

```
; Defining a function wich read integer
define i32 @readInt() {
entry:
        %res = alloca i32
        %digit = alloca i32
        %mult = alloca i32
         store i32 0, i32* %res
```

```
        store i32 1, i32* %mult
        br label %firstread
firstread:
        %a = call i32 @getchar()
        %b = icmp eq i32 %a, 45
        br i1 %b, label %firstminus, label %firstdigit
firstminus:
        store i32 -1, i32* %mult
        br label %read
firstdigit:
        %c = sub i32 %a, 48
        store i32 %c, i32* %digit
        %d = icmp ne i32 %a, 10
        br i1 %d, label %save, label %exit
read:
        %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
save:
        %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
getminus:
        store i32 -1, i32* %mult
        br label %read
exit:
        %7 = load i32* %res
        %8 = load i32* %mult
        %9 = mul i32 %7, %8
        ret i32 %9
}
```

The function reads each char from the standard input. If the first char is a minus (ascii code : 45) the result is multiplied by -1 to have a negative number. If the function encounter an "end of line" symbol (ascii code : 10), it stops and returns the result. All the others chars parsed are transformed in the corresponding integer by subbing 48 (the ascii code of 0) and added to the already transformed char multiplied by 10.

This code is only included if there is at least one read call.

A function to print integers

As there is a print operation in the language, a function to print an integer on the standard output had to be implemented. Here is the llvm code which do this :

```
; Defining a function wich print integer
define void @putInt(i32 %a) {
entry:
        %size = alloca i32
        %int = alloca i32
        store i32 10, i32* %size
        store i32 %a, i32* %int
        br label %negativetest
negativetest:
        %0 = icmp slt i32 %a, 0
        br i1 %0, label %minus, label %sizecmp
minus:
        call i32 @putchar(i32 45)
        %2 = load i32* %int
        %3 = mul i32 %2, -1
        store i32 %3, i32* %int
        br label %sizecmp
computesize:
        %4 = load i32* %size
        %5 = mul i32 %4, 10
        store i32 %5, i32* %size
        br label %sizecmp
sizecmp:
        %6 = load i32* %size
        %i = load i32* %int
        %7 = icmp ugt i32 %6, %i
        br i1 %7, label %printloop, label %computesize
printloop:
        %j = load i32* %int
        %8 = load i32* %size
        %9 = udiv i32 %8, 10
        store i32 %9, i32* %size
        %10 = udiv i32 %j, %9
        %11 = urem i32 %10, 10
        %12 = add i32 %11, 48
        call i32 @putchar(i32 %12)
        %14 = icmp ugt i32 %9, 1
        br i1 %14, label %printloop, label %exit
exit:
        ret void
}
```

The function starts by testing if the integer is negative. If it is, it prints the minus char. Then it stores the absolute value of the integer. After that, it computes the lowest power of ten bigger than this absolute value, with a minimum of ten to avoid division by zero later, and stores it in a "size" variable.

After that, the function loops while the "size" variable is strictly greater than one. In each loop the "size" variable is divided by ten and stored. The absolute value is divided by the new size. The modulo ten of this division is taken and the result is transformed in the corresponding ascii code by adding 48.

The left-associativity

In the part 2 of the project, the left associativity of the operators was not kept because it was impossible while removing left-reccursion. This problem is fixed in the third part at the level of llvm generation. Here is a pseudo-code of how it is applied with the example of the rule 35 :

```
[35]     <AndCond2>       ->       and <CondTerm> <AndCond2>

left = number of last register // the result of the last comparaison is stored there
Apply <CondTerm> rule
right = number of last register // the result of <CondTerm> is stored there
Write LLVM and between left and right register
Apply <AndCond2> rule
```

As you can see, the llvm code is written before the recursion so it gives a left-associativity.

operation NOT

There is no "not" operation in llvm so a xor with 1 has been used to remplace it. $not1 = 0$, $not0 = 1$, $1xor1 = 0$ and $0xor1 = 1$.