dOvs - Codegeneration

Group 9
Miran Hasanagić - 20084902
Jakob Graugaard Laursen - 20093220
Steven Astrup Sørensen - 201206081

November 19, 2015

1 Introduction

This report describes the work carried out in order to develop the code generation phase, which is the last component in the compiler for this course. The code generation component takes the Intermediate Representation (IR), and creates assembly code. However, it shall be noted that the IR generated from the previous work, is used as input for the `canon.sml` module, in order to produce a simplified IR according the chapter 8 in the book.

The rest of this report is structured the following way. The two following sections described the two main sml file which have been developed, separately. Afterwards the experience gain and different tests are presented. Finally, the implementation of the code generation component is concluded.

2 Instruction selection

The goal of the code generation is to generated assembly code as specified in the assignment for this course. Hence the target assembly is x86. The `x86gen.sml` takes the output from the canon module, and creates the assembly code. Its main structure is similar to `semant.sml` and `irgen.sml`, in that recursive functions are used to transverse the tree. However, this module does not use "real" registers, but just assumes there are unlimited amount of them. It relies on the support module `x86frame.sml` to create this illusion.

3 The support modules

As mentioned above, the `x86frame.sml` is a support module for the `x86gen.sml` module. The main responsibility of this module is to create the illusion of an infinite amount of registers for `x86gen.sml`. Each time `x86gen.sml` uses a "register", `x86frame` checks if is a real one. Does who isn't real is called temporaries. If a temporary was selected, `x86frame` looks for an available real one, and substitutes it in. If the temporary was needed because it held a value, `x86frame.sml` loads the value from memory into the selected register, then allows the code to run. If the temporary instead was used to store a value, the selected code is allowed to run, then `x86frame.sml` stores the value back in memory from the selected register.

This module both has functions to check if it is a register, and if a specific register is unused currently, meaning that it selects a real register unused by the code.

4 Exponentiation

We were required to implement the runtime function of exponentiation. The only requirment was that if the exponent was positive, we actually calculated the correct amount, ignoring overflow problems.

In short, our method is simple. Everything lifted to 0 gives 1. We accept a signed int as an exponent, but we use it's absolute value as the actual exponent. This means that

$$b^e = b^{-e}.$$

While bizzare, the reason for this choice is twofold. If $b = -1$, then the above gives that $(-1)^e \cdot (-1)^{-e} = 1$, so if b^e has a multiplicative inverse in the integers, then b^{-e} will give it, as long as the values don't overflow.

The second is simply ease of implementation. Instead of dealing with special cases, our code is short and sweet, as seen here

```
int exponent (int base, int expn) {
    int res = 1;

    while (expn != 0) {
        /* Exploit binary representation of expn */
        res *= res;
        if (expn % 2 != 0) {
            res *= base;
        }
        expn /= 2;
    }
    return res;
}
```

5 Problems encountered & experience gained

5.1 General Experience

It was hard to understand how to apply the canon module in the beginning, and also to generate the assembly code. However, the main module provided valuable help for both cases, using the compile method. Afterwards, the simple constructs were made, such a integer expression and binary operators, and tested. In general the approach was to create a tiger program, and then replace the relevant TODO inside both files, described above. Next the more challenging constructs could be implemented. Some of the more challenging are addressed below.

5.2 TODO: Add more if needed

6 5 tiger programs

6.1 test01.tig

```
/*
Test Array, reassignment
*/
let
        type intarray = array of int
        var c := intarray[3] of 43
in
        c[0] := 2;
        c[0] + c[2]
end
```

6.2 test02.tig

```
/* checking that we are doing side effect correctly */
let
        var c := 0
        function increment():int = (c := c+1;1)
        function multiply():int = (c := 2*c;2)
        function test(a:int, b:int):int = c
in
        test(increment(),multiply())
end
```

6.3 test03.tig

```
/*
Checks combination of arrays and record
*/
let
type rectype = {name: string, address: string, id: int, age: int}
type arrtype = array of rectype
var arr := arrtype [5] of
                  rectype{name="aname", address="somewhere", id=0, age=0}
in
    arr[3].name := "kati";
    arr[1].age := 23;
    arr[1].age
end
```

6.4 test04.tig

```
TODO: A test about exponents, that shows it in a good way.
```

6.5 test05.tig

```
/* test for recursive functions
 * current version has an infinite loop for negative cases
*/
let
        even(i:int):int = if i = 0 then 1
                else if i = 1 then 0
                else odd(i-1)
        odd(i:int):int = even(i-1)
in
        even(4) = odd(11)
end
```

7 Conclusion

This report presented the development of the code generator in `x86gen.sml`, which generates assembly instructions from the IR tree. The main functionality was described, and how some modules, like `x86frame.sml`, are used in order to use real registers when generating assembly code. Additionally the gained experience was described. Finally, additional tests provided confidence that this code generator is working correctly.