dOvs - Codegeneration

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

November 18, 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 number of registers. It leave the usage of "real" registers, to the support module `x86frame.sml`, which the described below.

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 usage of real registers. This module is called when real registers need to be used. This module both has functions to check if it is a register, and if a specific register is unused currently. Additionally, it moves relevant values into register in order to perform a operation on these.

4 Problems encountered & experience gained

4.1 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.

4.2 TODO: Add more if needed

5 5 tiger programs

5.1 test01.tig

```
/*
Array test for equal of equal things and unequal for unequal things
*/
let
        type A = array of int
        var b := A[2] of 0
        var c := b
        var d := A[1] of 3
in
        b = c & c <> d
end
```

5.2 test02.tig

```
/*
Test for loops,
that we update control variable correctly,
and that we do not loop to much,
and that we do no enter loops where low > high
*/
let
        var a := 0
in
        for i := 1 to 0 do a := 99; for i := 1 to 3 do a := a + i; a
end
```

5.3 test03.tig

```
/*
Checks if we put the arguments in correct order
*/
let
        function sub(a:int,b:int):int = a - b
in
        sub(3,2)
end
```

5.4 test04.tig

```
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
```

5.5 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) & 0 = odd(4)
end
```

6 Conclusion