

Group Assignment

Compilers

Oliver Mikkell Winther Cristensen (klw741)

Mathias Brndum Leroul (swn848)

Sebastian O. Jensen (gjx653)

Turn-in date: 18-12-2015

Contents of the Report

It is largely up to you to decide what you think is important to include in the report, as long as the following requirements are met: Your report should justify all your changes to the compiler modules, in particular, the lexer, parser, interpreter, type checker, machine-code generator, and the optimization modules. All major design decisions should be presented and justified. When evaluating your work, the main focus will be on verifying that your implementation of the language is correct. While we do not put particular emphasis on compiler optimizations in this course, we will also evaluate the quality of the generated code: if there are obvious inefficiencies that could have been easily solved you will be penalized, as they testify either wrong priorities or lack of understanding. You should not include the whole compiler in your report, but you must include the parts that were either added, i.e. new code, or substantially modified. Add them as code listings, and use the appendix if they get too big. Ideally, we should not need to read your source code. Your report should describe whether the compilation and execution of your input/test (FASTO) programs results in the correct/expected behavior. If it does not, try to explain why this is. In addition, (i) it must be assessed to what extent the delivered test programs cover the language features, and (ii) if the implementation deviates from the correct/expected behavior than the test program(s) should illustrate the implementation shortcomings to your best extent. Known shortcomings in type checking and machine-code generation must be described, and, whenever possible, you need to make suggestions on how these might be corrected. The report should not exceed this document in size, and should have an appropriate level of detail. You might be penalized if your report includes too many irrelevant details.

1 Introduction

We're given the grand task of

2 Implementation

Each implementation step is presented in the order, which the code is passed through: Lexer -> Parser -> Interpreter -> Type Checker -> MIPS Code Generator

Notes

- Implemented Boolean Literals: true and false.
Added test files bool.fo, bool.in, bool.out

2.1 True/False

In the first task, we're given the sub-task to implement the boolean operators True and False. We start out by creating the following test files:

File name	Code
bool.fo	<pre>fun int boolTest(int a) = if a == 1 then 1 else 0 fun int main() = let a = read(int) in write(boolTest(a))</pre>
bool.in	1
bool.out	1

Then we implement the True/False token in our parser as well as add True and False to the expression list

File name	Code
Parser.grm	%token <(int*int)> TRUE FALSE
Parser.grm	<pre> TRUE { Constant (BoolVal true, \$1)} FALSE { Constant (BoolVal false, \$1)}</pre>

And finally we add the boolean constants to our MIPS code generator

File name	Code
CodeGen.sml	<pre> Constant (BoolVal b, pos) => if b then [Mips.LI (place, makeConst 1)] else [Mips.LI (place, makeConst 0)]</pre>

2.2 Multiplication and Division

Moving on to the implementation of multiplication and division, we again start out with creating test files.

File name	Code
Mult.fo	<pre>fun int Mult(int a) = a * 2 fun int main() = let a = read(int) in write(Mult(a))</pre>
Mult.in	4
Mult.out	8
Div.fo	<pre>fun int Div(int a) = a / 2 fun int main() = let a = read(int) in write(Div(a))</pre>
Div.in	128
Div.out	64

After creating the test files, we swiftly move on to the lexer

File name	Code
Lexer.lex	<pre> '*' { Parser.TIMES (getPos lexbuf) } '/' { Parser.DIVIDE (getPos lexbuf) }</pre>

In the parser, we add TIMES and DIVIDE to the already created "PLUS MINUS DEQ EQ LTH" token. They're given a proper precedence order as well as added to the expression list.

File name	Code
Parser.grm	%token <(int*int)> PLUS MINUS DEQ EQ LTH DIVIDE TIMES
Parser.grm	%left DIVIDE %left TIMES
Parser.grm	Exp : Exp DIVIDE Exp { Divide (\$1, \$3, \$2) } Exp TIMES Exp { Times (\$1, \$3, \$2) }

Moving on to the type checker, we make sure that we can properly check the used types, when doing multiplication and division

File name	Code
TypeChecker.sml	and checkExp ftab vtab (exp : In.Exp) = case exp of . . . In.Times (e1, e2, pos) => let val (_, e1_dec, e2_dec) = checkBinOp ftab vtab (pos, Int, in (Int, Out.Times (e1_dec, e2_dec, pos)) end In.Divide (e1, e2, pos) => let val (_, e1_dec, e2_dec) = checkBinOp ftab vtab (pos, Int, in (Int, Out.Divide (e1_dec, e2_dec, pos)) end

We go ahead and implement them in our interpreter

File name	Code
Interpreter.sml	<pre> evalExp (Times(e1, e2, pos), vtab, ftab) = let val res1 = evalExp(e1, vtab, ftab) val res2 = evalExp(e2, vtab, ftab) in case (res1, res2) of (IntVal n1, IntVal n2) => IntVal (n1*n2) _ => invalidOperands "Multiplication on non-integral ar end evalExp (Divide(e1, e2, pos), vtab, ftab) = let val res1 = evalExp(e1, vtab, ftab) val res2 = evalExp(e2, vtab, ftab) in case (res1, res2) of (IntVal n1, IntVal n2) => IntVal (n1 div n2) _ => invalidOperands "Division on non-integral args: " [end </pre>

Finally they're added to the MIPS code generator

File name	Code
CodeGen.sml	<pre> fun compileExp e vtable place = case e of . Divide (e1, e2, pos) => let val t1 = newName "divide_L" val t2 = newName "divide_R" in val code1 = compileExp e1 vtable t1 val code2 = compileExp e2 vtable t2 in code1 @ code2 @ [Mips.DIV (place,t1,t2)] end Times (e1, e2, pos) => let val t1 = newName "times_L" val t2 = newName "times_R" in val code1 = compileExp e1 vtable t1 val code2 = compileExp e2 vtable t2 in code1 @ code2 @ [Mips.MUL (place,t1,t2)] end </pre>

2.3 AND and OR

Moving on to the implementation of the boolean literals, AND and OR, we again start out with creating test files.

File name	Code
and.fo	<pre>fun bool FAnd(bool a, bool b) = a && b fun bool main() = let a = read(bool) in let b = read(bool) in let c = write(FAnd(a,a)) in let d = write(FAnd(b,b)) in let e = write(FAnd(a,b)) in write(FAnd(b,a))</pre>
and.in	<pre>1 0</pre>
and.out	<pre>truefalsefalsefalse</pre>
or.fo	<pre>fun bool orTest(bool a, bool b) = a b fun bool main() = let a = read(bool) in let b = read(bool) in let c = write(orTest(b,b)) in let d = write(orTest(a,a)) in let e = write(orTest(a,b)) in write(orTest(b,a))</pre>
or.in	<pre>1 0</pre>
or.out	<pre>falsetruetrue>true</pre>

After creating the test files, we swiftly move on to the lexer

File name	Code
Lexer.lex	<pre> "&&" { Parser.AND (getPos lexbuf) } " " { Parser.OR (getPos lexbuf) }</pre>

In the parser, we add AND and OR to the "TRUE FALSE" token as well as added to the expression list.

File name	Code
Parser.grm	%token <(int*int)> TRUE FALSE AND OR
Parser.grm	Exp : Exp AND Exp { And (\$1, \$3, \$2) } Exp OR Exp { Or (\$1, \$3, \$2) }

Moving on to the type checker, we make sure that we can properly check the used types, when using the boolean literals AND and OR

File name	Code
TypeChecker.sml	<pre> In.And (e1, e2, pos) => let val (t1, e1') = checkExp ftab vtab e1 val (t2, e2') = checkExp ftab vtab e2 in case (t1 = t2, t1) of (false, _) => raise Error ("And cannot take "^ ppType t1 "and "^ppType t2, pos) (true, Array _) => raise Error ("And cannot operate on array", pos) _ => (Bool, Out.And (e1', e2', pos)) end In.Or (e1, e2, pos) => let val (t1, e1') = checkExp ftab vtab e1 val (t2, e2') = checkExp ftab vtab e2 in case (t1 = t2, t1) of (false, _) => raise Error ("Or cannot take "^ ppType t1 "and "^ppType t2, pos) (true, Array _) => raise Error ("Or cannot operate on array", pos) _ => (Bool, Out.Or (e1', e2', pos)) end end </pre>

We go ahead and implement them in our interpreter

File name	Code
Interpreter.sml	<pre> evalExp (And (e1, e2, pos), vtab, ftab) = let val res1 = evalExp(e1, vtab, ftab) val res2 = evalExp(e2, vtab, ftab) in case (res1, res2) of (BoolVal n1, BoolVal n2) => BoolVal (n1 andalso n2) _ => invalidOperands "OR on non-boolean args: " [(Int, Int)] end evalExp (Or (e1, e2, pos), vtab, ftab) = let val res1 = evalExp(e1, vtab, ftab) val res2 = evalExp(e2, vtab, ftab) in case (res1, res2) of (BoolVal n1, BoolVal n2) => BoolVal (n1 orelse n2) _ => invalidOperands "AND on non-boolean args: " [(Int, Int)] end </pre>

Finally we add the short-circuit AND and OR to the MIPS code generator

File name	Code
CodeGen.sml	<pre> And (e1, e2, pos) => let val t1 = newName "and_L" val t2 = newName "and_R" val code1 = compileExp e1 vtable t1 val code2 = compileExp e2 vtable t2 val finish = newName "finish" in code1 @ [Mips.LI (place,"0") , Mips.BEQ (t1, "0", finish)] @ code2 @ [Mips.BEQ (t2, "0", finish) , Mips.LI (place, "1") , Mips.LABEL finish] end Or (e1, e2, pos) => let val t1 = newName "or_L" val t2 = newName "or_R" val code1 = compileExp e1 vtable t1 val code2 = compileExp e2 vtable t2 val finish = newName "finish" in code1 @ [Mips.LI (place,"1") , Mips.BNE (t1, "0", finish)] @ code2 @ [Mips.BNE (t2, "0", finish) , Mips.LI (place, "0") , Mips.LABEL finish] end </pre>

3 Results

4 Conclusion