Compilation Homework

98-317: Hype for Types

Due: 13 November 2018 at 6:30 PM

Introduction

This week we talked about compilation. In this homework you will implement a compiler from an imperative language with rich boolean/arithmetic expressions and control flow features into a simple abstract assembly language with arithmetic instructions and conditional jumps.

You have a lot of freedom in how you choose to compile. The assembly interpreter tracks how many instructions it takes to execute a piece of compiled code. A goal in compiler implementation is to make the generated assembly run as efficiently as possible, and you can use this value returned from the interpreter to measure how efficient your compiled code is. You can use the optimization techniques we covered in class to maximize the efficiency of the code your compiler generates.

This assignment will have a scoreboard on Autolab, on which you can see how your compiler's optimizations stack up to your peers' compilers!

In this assignment you will also have an opportunity to implement a statics-checker for the imperative language, which checks that the program always returns a value and never uses a variable before it has been assigned to.

Turning in the Homework From inside the `hw10` directory, run the command

```
tar cf handin.tar Compile.sml Statics.sml
```

then submit `handin.tar` to Autolab.

VSIMPL

For this assignment we've designed a simple imperative language which we call VSIMPL (short for ~~Vijay's~~ Very Simple IMperative Programming Language). It has two sorts of expressions: arithmetic expressions (*aexp* for short) and boolean expressions (*bexp* for short). Variables in VSIMPL can only store integer values; never booleans. For simplicity, VSIMPL does not have block scoping; you can think of all variables as global. A VSIMPL program takes some number of integers as inputs, and returns an integer. Here is the syntax of VSIMPL:

Arithmetic Expression	*aexp*	::=	x	variable
			c	numerical constant
			aexp `+` *aexp*	addition
			aexp `-` *aexp*	subtraction
			aexp `*` *aexp*	multiplication
			aexp `/` *aexp*	division
Boolean Expression	*bexp*	::=	`true`	true constant
			`false`	false constant
			`!`*bexp*	logical negation
			bexp `||` *bexp*	logical or
			bexp `&&` *bexp*	logical and
			aexp `==` *aexp*	equality
			aexp `!=` *aexp*	inequality
			aexp `<` *aexp*	less than
			aexp `>` *aexp*	greater than
Command	*cmd*	::=	x `=` *aexp* `;`	assignment
			`if` *bexp* `{` *cmds* `}` `else` `{` *cmds* `}`	conditional
			`while` *bexp* `{` *cmds* `}`	loop
			`return` *aexp* `;`	return
Program	*program*	::=	`main` (*params*) `{` *cmds* `}`	

where *cmds* means zero or more instances of *cmd*, and *params* means a comma-separated sequence of zero or more variable names. This syntax is implemented in `VSIMPL.sml`. To understand the syntax better, you can find example VSIMPL programs in the `examples` directory. We've provided a parser for VSIMPL; you can use it through the `Top` module as described later in this document.

The statics of VSIMPL prevent accessing from a variable before it is assigned to, and ensure that a program will never exit before returning. The dynamics of VSIMPL are exactly what you'd expect from experience with imperative languages like C and Python.

Abstract Assembly

The target language of your compiler is 3-address abstract assembly with the following syntax:

Operand	*oper*	::=	x	variable
			c	constant
Instruction	*instruction*	::=	ℓ :	label
			$x \leftarrow oper$	mov
			$x \leftarrow oper + oper$	add
			$x \leftarrow oper - oper$	sub
			$x \leftarrow oper \times oper$	mul
			$x \leftarrow oper \div oper$	div
			$x \leftarrow oper = oper$	eq
			$x \leftarrow oper < oper$	lt
			`JUMP` ℓ	jump
			`IF` *oper* `THEN` ℓ `ELSE` ℓ	conditional jump
			`RET` *oper*	return
Program	*program*	::=	`main` $(params) = instructions$	

Where *params* means a comma-separated sequence of zero or more variable names, and *instructions* means a sequence of zero or more instances of *instruction*. This syntax is implemented in `Assembly.sml`.

All values in this language are integers. The program executes by executing instructions sequentially, starting at the beginning of the program. Label instructions don't do anything during execution and are skipped when encountered. When a jump instruction is encountered, the execution continues from the location of the corresponding label. The conditional jump instruction jumps to the first label if the operand is nonzero, and jumps to the second label if the operand is zero. When a return command is encountered, execution terminates and the value of the operand is the program's output.

An interpreter for this language is implemented in `Assembly.sml`.

Required: Write a Compiler

In `Compile.sml`, implement a compiler from VSIMPL to Assembly as a function

```
compile :  VSIMPL.program -> Assembly.program
```

Your goal is first and foremost to implement a correct compiler, such that for any VSIMPL program you put into it, the behavior of the output Assembly code is consistent with the behavior you'd expect from the source program. Our reference implementation is fewer than 100 lines long.

A secondary goal is for the output Assembly code to be efficient. The Assembly interpreter counts the number of steps it takes to execute a program; this number lets you gauge how efficient your compiler's output is. The goal is to get this number to be as small as possible for any particular source program and inputs.

Testing your compiler We've provided the following functions to test your compiler:

```
Top.compile_and_print :  string -> unit
Top.compile_and_run :  string -> int list -> unit
```

After running `sml -m sources.cm` inside the `hw10` directory,

- `Top.compile_and_print <file>` reads and parses a VSIMPL program stored in `<file>`, parses it, feeds it through your compiler, and prints out the resulting compiled code.
- `Top.compile_and_run <file> <arguments>` reads and parses a VSIMPL program stored in `<file>`, feeds it through your compiler, then runs the resulting Assembly code in the interpreter on the inputs `<arguments>`.

We've provided some example VSIMPL programs in the `examples` directory to help with debugging, and we encourage you to also write your own VSIMPL programs to test on. As an example, here is what we get when we run our reference solution compiler on the program in `fact.vsimpl` with input 5:

```
- Top.compile_and_run "examples/fact.vsimpl" [5];
Finished parsing source code.
Finished checking program statics.
Finished compiling program.
Running compiled program...
Took 31 steps to run.
Program returned 120.
val it = () : unit
```

Not Required: Check Statics

We consider a program to violate the statics of VSIMPL if it could possibly do either of the following:

- use a variable before it is assigned to
- terminate without returning

In `Statics.sml`, implement a function

```
check :  VSIMPL.program -> unit
```

which returns `()` if the given program checks under VSIMPL's statics, and raises `Error` if it violates the statics.

Hint: Since it's impossible to statically predict the value of an expression branched on in an `if` or `while` command, you'll need to be conservative when dealing with them. In particular:

- An `if` command assigns to a variable if both of its branches assign to that variable.
- An `if` command returns if both of its branches return.
- A `while` command doesn't necessarily assign to any variables.
- A `while` command doesn't necessarily return.