Compilation Homework

98-317: Hype for Types

Checkpoint Due: 2 April at 6:30 PM
Final Due: 9 April 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.

This homework is split into two parts: For the checkpoint due on 2 April at 6:30 PM, you will implement a compiler for a subset of the language with only "straight-line" code; that is, code without control flow constructs. For the final assignment due on 9 April at 6:30 PM, you will extend your straight-line code compiler to work on the complete language, including control flow constructs. There are no optional parts of this homework.

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!

Turning in the Homework There are two different Autolab assignments to submit to: "Compilers - Checkpoint" and "Compilers - Final". To submit to either of them, submit `Compile.sml`. Do not `tar` it; simply submit the SML file itself.

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 (representing the parameters the program takes as input), and *instructions* means a sequence of zero or more instances of *instruction*. This syntax is implemented in `Assembly.sml`.

A program in this language takes some number of integers as input and returns a single integer.

All values in this language are integers. The program executes by executing instructions sequentially, starting at the beginning of the program until it reaches a return command. When a return command is encountered, execution terminates and the value of the operand is the program's output.

We've implemented an interpreter for this language `Assembly.sml`. There are instructions on how to use this interpreter later when we talk about how to test your code.

Straight-Line Code (due 2 April at 6:30 PM)

We've designed a simple imperative language which we call System C. For the checkpoint, you'll implement a compiler for a subset of System C with arithmetic expressions and no control flow constructs. Here is the syntax of this subset of System C:

Arithmetic Expression	*aexp*	::=	x	variable
			c	numerical constant
			aexp + *aexp*	addition
			aexp - *aexp*	subtraction
			aexp * *aexp*	multiplication
			aexp / *aexp*	division
Command	*cmd*	::=	x = *aexp*;	assignment
			`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 representing the inputs to the program. This syntax is implemented in `SystemC-checkpoint.sml`. To understand the syntax better, you can find example programs in the `examples-checkpoint` directory. We've provided a parser; you can use it through the `Top` module as described later in this document.

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

In `Compile.sml`, implement a compiler from this subset of System C to Assembly as a function

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

Testing your compiler There are two different CM files to build with depending on which part of the assignment you're working on. Using `checkpoint.cm` builds your code against the straight-line code subset of System C, and using `final.cm` builds your code against all of System C. So, for example, when you're working on the checkpoint you might run

```
sml -m checkpoint.cm
```

to build your code.

We've provided the following functions to help with testing your compiler:

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

where

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

We've also provided some example System C programs to help with debugging. There are some straight-line code programs in the `examples-checkpoint` to help with the checkpoint, and there are some programs in the `examples` directory which use control flow constructs. We encourage you to also write your own System C programs to test on.

As an example, here is what we get when we run our reference solution compiler on the program in `examples/fact.vsimpl` with input 5. We give `compile_and_run` a one-element list because the `fact` program takes one parameter as input.

```
- Top.compile_and_run "examples/fact.sc" [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
```

System C (due 9 April at 6:30 PM)

To go from straight-line code to System C. we add `if` and `while` commands to the language. We also now have boolean expressions (in addition to arithmetic expressions as before) to use in the conditions of `if` and `while` commands. Variables in System C can only store integer values; never booleans. For simplicity, System C does not have block scoping; you can think of all variables as global. Here is the full syntax of System C:

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* }	

In `Compile.sml`, extend your implementation of `compile` to work on all of System C. A nifty way to get started on this is to build your checkpoint code with `final.cm` and see where you get "match nonexhaustive"; this shows you what additional code you need to write.

Your goal is first and foremost to implement a correct compiler, such that for any System C 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.