Programs as Data
Exercise 2.4 and 2.5

Niels Hallenberg

Thursday 2025-08-31

Recap of `Intcomp1.fs`

We have source language `expr`.

The function `eval` interpretates the source language and returns result of evaluation, e.g.:

```
> eval e1 [];;
val it : int = 34
> eval e2 [];;
val it : int = 2217
> eval e3 [];;
val it : int = 100
> eval e5 [];;
val it : int = 14
```

Recap of `Intcomp1.fs`

We have target language `sinstr`.

The function `scomp` compiles source langauge `expr` to target languate `sinstr`, e.g.:

```
> scomp e1 [];;
val it : sinstr list = [SCstI 17; SVar 0; SVar 1; SAdd; SSwap; SPop]
> scomp e2 [];;
val it : sinstr list =
  [SCstI 17; SCstI 22; SCstI 100; SVar 1;
   SMul; SSwap; SPop; SVar 1; SAdd; SSwap; SPop]
> scomp e3 [];;
val it : sinstr list =
  [SCstI 5; SCstI 4; SSub; SCstI 100; SVar 1; SMul; SSwap; SPop]
> scomp e5 [];;
val it : sinstr list =
  [SCstI 2; SCstI 3; SVar 0; SCstI 4; SAdd; SSwap; SPop; SMul]
```

Purpose of 2.4 and 2.5

The purpose of exercise 2.4 is to emit a file representing the target language `sinstr` (stack instructions).

The stack instructions are then to be read from the file by the stack machine `Machine.java`.

The exercise 2.5 is about making the stack machine reading the file. *This is not core to our learning and you are thereby given this file up front, i.e., 2.5 has already been solved.*

Example e1

The example `e1` found in `Intcomp1.fs`.

```
let e1 = Let("z",
             CstI 17,
             Prim("+",Var "z", Var "z"));;
```

From F# interactive you compile to target language

```
> scomp e1 [];;
val it : sinstr list =
  [SCstI 17; SVar 0; SVar 1;
   SAdd; SSwap; SPop]
```

Exercise 2.4

You need to implement a function

```
sinstrToInt : sinstr -> int list
```

that converts a `sinstr` into a list of integers representing the bytecode instruction (`sinstr`) on the file. We use the below table for the conversion.

`SCst` and `SVar` has An extra Parameter.

Sinstr	Value	Parameter
SCst	0	*X*
SVar	1	*X*
SAdd	2	
SSub	3	
SMul	4	
SPop	5	
SSwap	6	

Stack machine state transitions

Instruction	Stack before		Stack after	Effect
RCst i	s	$\Rightarrow$	s, i	Push constant
RAdd	s, i_1, i_2	$\Rightarrow$	$s, (i_1 + i_2)$	Addition
RSub	s, i_1, i_2	$\Rightarrow$	$s, (i_1 - i_2)$	Subtraction
RMul	s, i_1, i_2	$\Rightarrow$	$s, (i_1 * i_2)$	Multiplication
RDup	s, i	$\Rightarrow$	s, i, i	Duplicate stack top
RSwap	s, i_1, i_2	$\Rightarrow$	s, i_2, i_1	Swap top elements

Stack machine instructions,
Figure 2.3 and implementation
Explained in Section 2.8 in PCD.

Example stack machine program

- A simple program, file `is1.txt`:

```
0 17 1 0 1 1 2 6 5
```

Numeric code

```
0 17
1 0
1 1
2
6
5
```

```
0:  SCSTI 17
2:  SVAR 0
4:  SVAR 1
6:  SADD
7:  SWAP
8:  SPOP
```

Symbolic code

- Running the code in file `is1.txt`:

```
% java Machine is1.txt
Result: 34
```

See Section 2.8 in PCD

Exercise 2.4

You need to implement a function

```
assemble : sinstr list -> int list
```

that folds over a list of `sinstr` and use `sinstrToInt` to accumulate the list of integers.

```
> assemble (scomp e1 []);;
val it : int list = [0; 17; 1; 0; 1; 1; 2; 6; 5]
```

You can then use the function intsToFile to create a file with the program:

```
> intsToFile (assemble (scomp e1 [])) "is1.txt";;
val it : unit = ()
```

Exercise 2.5

This has already been done and you simply compile the provided `Machine.java` and run:

```
% javac Machine.java
% java Machine is1.txt
Result: 34
```