Lecture 7 - Exercises (PSD)

Exercise 7.1 Download `microc.zip` from the book homepage, unpack it to a folder `MicroC`, and build the micro-C interpreter as explained in `README.TXT` step (A).

Run the `fromFile` parser on the micro-C example in source file `ex1.c`. In your solution to the exercise, include the abstract syntax tree and indicate its parts: declarations, statements, types and expressions.

Run the interpreter on some of the micro-C examples provided, such as those in source files `ex1.c` and `ex11.c`. Note that both take an integer `n` as input. The former program prints the numbers from `n` down to 1; the latter finds all solutions to the `n`-queens problem.

Exercise 7.2 Write and run a few more micro-C programs to understand the use of arrays, pointer arithmetics, and parameter passing. Use the micro-C implementation

in `Interp.fs` and the associated lexer and parser to run your programs, as in Exercise 7.1.

Be careful: there is no type checking in the micro-C interpreter and nothing prevents you from overwriting arbitrary store locations by mistake, causing your program to produce unexpected results. (The type system of real C would catch *some* of those mistakes at compile time).

(i) Write a micro-C program containing a function `void arrsum(int n, int arr[], int *sump)` that computes and returns the sum of the first `n` elements of the given array `arr`. The result must be returned through the `sump` pointer. The program's `main` function must create an array holding the four numbers 7, 13, 9, 8, call function `arrsum` on that array, and print the result using micro-C's non-standard `print` statement.

Remember that MicroC is very limited compared to actual C: You cannot use initializers in variable declarations like `"int i=0;"` but must use a declaration followed by a statement, as in `"int i; i=0;"` instead; there is no `for`-loop (unless you implement one, see Exercise 7.3); and so on.

Also remember to initialize all variables and array elements; this doesn't happen automatically in micro-C or C.

(ii) Write a micro-C program containing a function `void squares(int n, int arr[])` that, given `n` and an array `arr` of length `n` or more fills `arr[i]` with `i*i` for $\texttt{i} = 0, \ldots, \texttt{n} - 1$.

Your `main` function should allocate an array holding up to 20 integers, call function `squares` to fill the array with `n` square numbers (where $\texttt{n} \leq 20$ is given as a parameter to the `main` function), then call function `arrsum` above to compute the sum of the `n` squares, and print the sum.

(iii) Write a micro-C program containing a function `void histogram(int n, int ns[], int max, int freq[])` which fills array `freq` the frequencies of the numbers in array `ns`. More precisely, when the function returns, element `freq[c]` must equal the number of times that value `c` appears among the first `n` elements of `arr`, for `0<=c<=max`. You can assume that all numbers in `ns` are between 0 and `max`, inclusive.

For example, if your `main` function creates an array `arr` holding the seven numbers 1 2 1 1 1 2 0 and calls `histogram(7, arr, 3, freq)`, then afterwards `freq[0]` is 1, `freq[1]` is 4, `freq[2]` is 2, and `freq[3]` is 0. Of course, `freq` must be an array with at least four elements. What happens if it is not? The array `freq` should be declared and allocated in the `main` function, and passed to histogram function. It does not work correctly (in micro-C or C) to stack-allocate the array in `histogram` and somehow return it to the `main` function. Your `main` function should print the contents of array `freq` after the call.

Exercise 7.3 Extend MicroC with a for-loop, permitting for instance

```
for (i=0; i<100; i=i+1)
  sum = sum+i;
```

Exercise 7.3 Extend MicroC with a for-loop, permitting for instance

```
for (i=0; i<100; i=i+1)
  sum = sum+i;
```

To do this, you must modify the lexer and parser specifications in `CLex.fsl` and `CPar.fsy`. You may also extend the micro-C abstract syntax in `Absyn.fs` by defining a new statement constructor `Forloop` in the `stmt` type, and add a suitable case to the `exec` function in the interpreter.

But actually, with a modest amount of cleverness (highly recommended), you do not need to introduce special abstract syntax for for-loops, and need not modify the interpreter at all. Namely, a for-loop of the general form

```
for (e1; e2; e3)
  stmt
```

is equivalent to a block

```
{
  e1;
  while (e2) {
    stmt
    e3;
  }
}
```

Hence it suffices to let the semantic action `...` in the parser construct abstract syntax using the existing `Block`, `While`, and `Expr` constructors from the `stmt` type.

Rewrite your programs from Exercise 7.2 to use for-loops instead of while-loops.

Exercise 7.4 Extend the micro-C abstract syntax in `Absyn.fs` with the preincrement and predecrement operators known from C, C++, Java, and C#:

```
type expr =
    ...
  | PreInc of access   (* C/C++/Java/C#  ++i  or  ++a[e]  *)
  | PreDec of access   (* C/C++/Java/C#  --i  or  --a[e]  *)
```

Note that the predecrement and preincrement operators work on lvalues, that is, variables and array elements, and more generally on any expression that evaluates to a location.

Modify the micro-C interpreter in `Interp.fs` to handle `PreInc` and `PreDec`. You will need to modify the `eval` function, and use the `getSto` and `setSto` store operations (Sect. 7.3).

Exercise 7.5 Extend the micro-C lexer and parser to accept `++e` and `--e` also, and to build the corresponding abstract syntax.