Handout
MicroML

Interpretation and Compilation of Programming Languages
NOVA FCT
Tomé Dias 60719
Francisco Vasco 61028

10 June, 2024

Contents

1 The MicroML Language

In this project we implement a Typechecker, Interpreter and Compiler for the MicroML language as described in the document. Our projects consists of two executables the Interpreter and the Compiler. Each pipeline includes a parser and a type-checker stage before the execution / code generation stages, using the tools and techniques described throughout the semester.

1.1 File execution

To generate the JAR files for the compiler and interpreter run:

```
./generateJar.sh
```

To compile and run a program run (and then pass the program file path as input):

```
./runCompiler.sh
```

To test the interpreter run:

```
./runInterpreter.sh
```

1.2 Language Description

The following is a grammar for the syntax of MicroML:

$$
\begin{array}{lcl}
E & ::= & \mathsf{Num} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{id} \mid \mathsf{string} \\
& \mid & E + E \mid E - E \mid E * E \mid E/E \mid -E \mid (E) \\
& \mid & E = E \mid E \mathsf{!=} E \mid E > E \mid E < E \mid E \mathsf{<=} E \mid E \mathsf{>=} E \mid \\
& \mid & E \mathsf{\&\&} E \mid E \mathsf{||} E \mid \tilde{E} \\
& \mid & \mathsf{let}\ (x = E)^+\ \mathsf{in}\ E \\
& \mid & \mathsf{new}\ E \mid E := E \mid !E \\
& \mid & \mathsf{if}\ E\ \mathsf{then}\ E\ \mathsf{else}\ E\ \mathsf{end} \mid \mathsf{if}\ E\ \mathsf{then}\ E\mathsf{end} \\
& \mid & \mathsf{while}\ E\ \mathsf{do}\ E\ \mathsf{end} \\
& \mid & \mathsf{println}\ E \mid \mathsf{print}\ E \mid E; E \\
& \mid & \mathsf{fun}\ PL\ \rightarrow\ E\ : T\ \mathsf{end} \mid E(EL?) \\
& \mid & () \\
EL & ::= & E(, E)* \\
PL & ::= & (\mathsf{id} : T)+ \\
T & ::= & \mathsf{unit} \mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{ref}\ T \mid (T)* \rightarrow T
\end{array}
$$

The language is an expression-based language of arithmetic expressions, boolean operators (we write $\tilde{E}$ for negation) and conditionals, relational operators, declaration blocks (with optional type annotations); reference creation (new E), assignment ($E := E$) and dereference ($!E$); while loops (while E do E end); an *overloaded* print and print line primitive function; sequential composition of expressions ($E; E$); functions fun $PL\ \rightarrow\ E\ : T$ end and function application $E(EL?)$. The language also includes the unit type and its unique inhabitant, written $()$.

Declarations can be recursive (allowing for recursive function definitions) and functions can take multiple arguments. For simplicity, there is no partial function application.

The semantics of references and reference types is the one we presented during lectures. The expression $E_1 := E_2$ evaluates to the value of E_2. The expression (while E do E end) is of type unit, being evaluated essentially for its underlying effects on memory. We have also implemented function return types.

1.3 Additional Features

Test suite: We have prepared some *.j files you can compile and even alter, these files contain i.e the Fibonacci sequence and the factorial recursive implementation.

Language feature – Strings: Extend the language and its type system to include strings, featuring string literals and concatenation and any other operations you find appropriate.

Language feature – Type inference: There is some kind of type inference implemented Only function call need to define types.