# Handout
# MicroML

Interpretation and Compilation of Programming Languages
NOVA FCT
Bernardo Toninho

2 April, 2024

## Contents



## Changelog

Any changes to the handout will be reported here.

**2 April:** Initial publication of assignment.

## Deadlines

This handout is due on Friday, **June 7**, at 23h59m. There will be a turn-in of **Phase 1** on Friday, **April 26**, at 23h59m. The exact details on how to turn in your solution will be made available at a later date.

## 1 The MicroML Language

In this project you will implement an interpreter and a compiler for the MicroML language, described in this document. Your project must consist of **two** executables, one implementing the interpreter pipeline and another implementing the compiler pipeline. Each pipeline **must** (at least) include a parser and a type-checker stage before the execution / code generation stages, using the tools and techniques described throughout the semester.

The language you will be implementing in this project (MicroML) is a functional and imperative programming language in the style of OCaml.

## 1.1 Language Description

The following is a grammar for the syntax of MicroML:

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

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 $PL \rightarrow E$ 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.

It is part of the challenge to solve any ambiguities in the specification you may find. Feel free to reach out to the teaching staff accordingly.

## 1.2 Milestones

The project will require you to implement the parser, type-checker, interpreter and compiler for MicroML. You should have two executable files, each taking as command line argument a file name, containing the MicroML program to be processed. Both executables should process the file, reporting parsing, lexing or typing errors. If no errors are found, the interpreter executable should evaluate the program and the compiler executable should produce one or more class files as needed to then execute the program (note that you will have to invoke Jasmin directly in your compiler). You may define additional command line options and arguments for your executables, but the default behaviour should be as described above.

**Phase 1:** For Phase 1 you will have to turn in a working interpreter and type-checker for all features of the language with the exception of functions.

**Phase 2:** For Phase 2 you will have to implement all the language features in the interpreter and the compiler, as well as any optional and additional features you choose. With your submission you will also have to turn in a short report, discussing your design.

## 1.3 Additional Features and Grading

A project containing a full implementation of the compiler, interpreter and type-checker for MicroML will be graded for 16/20 points. Not submitting a working **Phase 1** submission will incur in a penalty of 3 points.

The remaining 4 points can be obtained through a combination of the following additional language and project features:

**Extensive test suite (1 pt):** A reasonably extensive test suite of MicroML programs, testing all of the features of the language and the pipeline. The test suite should include both positive and negative tests.

**Program optimization – Constant Folding and Propagation (1 pt):** Implement a program transformation pass, executing between the type-checker and the interpreter/compiler that performs *constant folding* and *constant propagation*.

**Program optimization – Tail Recursion (3 pt):** Implement a program transformation pass, executing between the type-checker and the interpreter/compiler that performs the *tail recursion* optimization. Tail recursion can be achieved by rewriting recursive functions into while loops and/or by having the interpreter/compiler reuse stack frames. Note that this optimization is challenging to implement correctly using our overall framework.

**Language Toolchain – Error Reporting (1.5 pt):** Implement a more robust error reporting system for the language frontend and type-checker. Error reporting should include line number and column information. The type-checker should also strive to report as many type errors as can be found in a given source file.

**Language feature – Multiple numeric types (1 pt):** The MicroML language includes a single numerical type (of integers). Extend the language to also include at least one more non-trivial numerical type such as doubles. Note that you will have to decide how the semantics and typing of arithmetic operations interplay with the presence of these. For instance, the type of $E + E$ will likely depend on the type of its operands. Similarly, compiling such an operation will also require using different JVM instructions.

**Language feature – Strings (1 pt):** Extend the language and its type system to include strings, featuring string literals and concatenation and any other operations you find appropriate.

**Language feature – Omitting type operations from function arguments (2pts):** Writing a function in MicroML requires annotating the formal parameters of the function with their types. However, it is possible to omit type annotations for functions whose declarations already specify a type. For instance, in the following program:

```
let f : int -> int = fun x:int -> x+1 end
 in
   println(f(23))
```

The type annotation for `x` can be omitted if we know that the overall function is meant to have the type `int -> int`. This can be achieved through a technique called *bidirectional typing* (see [DK21], among many other references that can be found online, including tutorials), where typing is split into two *modes*, a *checking* mode, where an expression is *checked* against an expected type, and a *synthesis* mode where the type of an expression is

synthesized (given types for its free variables). In the above program, since the type for `f` is explicitly given, we can type the function `fun x -> x+1 end` by *checking* it against the type `int -> int`, omitting the type annotation for `x`.

**Language feature – Arrays (2pts):** Extend the language with one-dimensional arrays. The extension should provide ways of allocating a new array, indexing into an array and modifying an array element, given its index.

**Language feature – Sum/Variant and Product Types (2pts):** Extend the language with variants (or sums) and product types. See Pierce's Types and Programming Languages book [Pie02, Chapter 11] for a description of these concepts.

**Language feature – Lists (2pts):** Extend the language with a list type. You don't need to define polymorphic lists. You are free to choose how to compile your lists (e.g., you can use Java's lists).

**Language feature – Type definitions (2pts):** Extend the language with a mechanism of type definitions in the style of OCaml's `type` keyword. The usefulness of this mechanism is limited without extending the language with variants and products. You should also consider whether the type names are purely aliases for their definitions or if they should function in the style of Haskell's `newtype` keyword (look it up!).

**Language feature – Algebraic data types (4 pts):** Extend the language with algebraic data types in the style of OCaml or Haskell. This feature is mutually exclusive with **Language feature – Sum/Variant and Product Types**.

**Language feature – Records and Record Types (3 pts):** Extend the language with records and record types. See Pierce's Types and Programming Languages book [Pie02, Chapter 11].

**Language feature – Type inference (4 pts):** Extending the language with type inference, allowing for *all* type annotations to be omitted. You need only restrict yourself to so-called Let-polymorphism [Pie02, Chapter 22].

**Program Optimization – Your choice (1-4 pts):** Implement a (set of) program optimization(s) of your choice, excluding constant folding and propagation. The grade will be a function of the degree of sophistication of the optimization(s).

**Language Feature – Your choice (1-4 pts):** Implement a language feature of your choice that is not already listed above. The grade will be a function of the degree of sophistication of the feature.

# References

[DK21] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Computing Surveys*, 54(5):1–38, May 2021.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.