Compiler Construction 1
Pepper · Lorenzen

Winter term 2014/2015
Handout: Nov 6, 2014
Submission: Nov 12, 2014

# Milestone 1a

Preparation for syntactic analysis

**Assignment** Preparation for syntactic analysis
Prepare the EBNF of μ-Opal for a recursive-descent parser with a lookahead of length 1.

a) The EBNF of μ-Opal (page 3) is written with readability in mind, not implementation. Write down an equivalent BNF.

b) Design an abstract syntax for μ-Opal and implement it as Scala `case` classes in the file `AbstractSyntax.scala` of the template.

c) Annotate your BNF with semantic actions, eliminate left-recursive productions, and perform left-factorization to obtain a grammar suitable for top-down parsing. Write down the resulting grammar. Write down all the director sets of competing productions of your transformed grammar and how you calculated them.

Submit the source tree of your μ-Opal compiler implementation including your implementation of the abstract syntax and a PDF file containing your solution to assignment a) and c).

*File to modify:* `AbstractSyntax.scala`

Carefully read the following pages. They state the rules how you have to prepare your homework, describe the language μ-Opal and the target virtual machine COVM, and how to set up and use the μ-Opal compiler template.

## Hints for the compiler project

The compiler project consists of the implementation of a parser, context checker, interpreter, and code generator for μ-Opal. The implementation language is Scala.
The project is divided into five milestones:

**Milestone 1a (due on Nov 12, 2014, 11:55 PM)** Preparation for lexical analysis (10 credits)

**Milestone 1b (due on Nov 26, 2014, 11:55 PM)** Implementation of the parser (20 credits)

**Milestone 2 (due on Dec 17, 2014, 11:55 PM)** Implementation of the context checker (25 credits)

**Milestone 3 (due on Jan 21, 2015, 11:55 PM)** Implementation of the interpreter (20 credits)

**Milestone 4 (due on Feb 4, 2015, 11:55 PM)** Implementation of the code generator (25 credits)

- You have to implement your solution using the template provided on the ISIS page. A later section (page 5) explains how to compile and run the template. Also consider the hints on the ISIS page on the recommended Scala tools.
- Your must submit your solution for each milestone as a single ZIP archive named ⟨*group number*⟩`.zip` using the links provided on the ISIS page. Generated artefacts like the `target` subdirectory or other temporary files must be excluded from the archive file. Your archive must extract into a source tree subdirectory named ⟨*group number*⟩ which immediately contains the file `build.sbt` from the template.
- You are free to add new source files to your implementation but you may only modify those files indicated in the assignment of the respective milestone. If you change other files your submission will be graded with 0 credits!
- Your submission must compile with the command `sbt compile`. If it gives compile-time errors it will be graded with 0 credits!
- You have to get at least 35 credits in the milestones 1a, 1b, and 2, and at least 25 credits in the milestones 3 and 4.

## The language µ-Opal

µ-Opal is a tiny functional language with first-order recursive functions over the primitive data types natural numbers and booleans. This is a typical µ-Opal program:

```
-- Calculate 1 * 2 * ... * (n-1) * n
DEF fac(n: nat): nat ==
  IF eq(n, 0) THEN 1 ELSE mul(n, fac(sub(n, 1))) FI

DEF MAIN: nat == fac(8)
```

The special definition `DEF MAIN:`*Type* `==` *Expr* specifies the result value of the program.
The string `--` indicates a comment up to the end of the line.

### Syntax

*This section is especially relevant to milestones 1a and 1b.*

The context-free syntax of µ-Opal is specified by the following EBNF where $\underline{\texttt{\#}}$ indicates the end of the input:

$$\begin{array}{lll}
\mathit{Prog} & ::= & \mathit{Def}^{+}\ \underline{\texttt{\#}} \\
\mathit{Def} & ::= & \underline{\texttt{DEF}}\ \mathit{Lhs}\ \underline{\texttt{==}}\ \mathit{Expr} \\
\mathit{Lhs} & ::= & \underline{\texttt{MAIN}}\ \underline{\texttt{:}}\ \mathit{Type} \\
 & \mid & \underline{\texttt{id}}\ \underline{\texttt{(}}\ [\underline{\texttt{id}}\ \underline{\texttt{:}}\ \mathit{Type}\ (\underline{\texttt{,}}\ \underline{\texttt{id}}\ \underline{\texttt{:}}\ \mathit{Type})^{*}]\ \underline{\texttt{)}}\ \underline{\texttt{:}}\ \mathit{Type} \\
\mathit{Type} & ::= & \underline{\texttt{nat}} \mid \underline{\texttt{bool}} \\
\mathit{Expr} & ::= & \underline{\texttt{number}} \mid \underline{\texttt{true}} \mid \underline{\texttt{false}} \\
 & \mid & \underline{\texttt{id}}\ [\underline{\texttt{(}}\ [\mathit{Expr}\ (\underline{\texttt{,}}\ \mathit{Expr})^{*}]\ \underline{\texttt{)}}] \\
 & \mid & \underline{\texttt{IF}}\ \mathit{Expr}\ \underline{\texttt{THEN}}\ \mathit{Expr}\ [\underline{\texttt{ELSE}}\ \mathit{Expr}]\ \underline{\texttt{FI}}
\end{array}$$

**Primitive functions** A µ-Opal program may use the following primitive functions:

```
DEF add(X: nat, Y: nat): nat     DEF sub(X: nat, Y: nat): nat
DEF mul(X: nat, Y: nat): nat     DEF div(X: nat, Y: nat): nat

DEF eq(X: nat, Y: nat): bool     DEF lt(X: nat, Y: nat): bool

DEF and(X: bool, Y: bool): bool  DEF or(X: bool, Y: bool): bool
DEF not(X: bool): bool
```

### Conditions for context correctness

*This section is especially relevant to milestone 2.*

A context correct µ-Opal program satisfies the following conditions:

- There exists exactly one definition for `MAIN`.
- The names of all defined functions and the primitive functions are disjoint.
- The names of the parameters of the left-hand side of a definition are disjoint.
- The type of right-hand side of a definition is the same as the declared result type of its left-hand side.
- All expressions are well-typed:

- A number is well-typed and has type `nat`.
- `true` and `false` are well-typed and have type `bool`.
- A variable `id` is well-typed if `id` is a parameter in the current context. Its type is the declared type of `id`.
- A function call `id(`$expr_1$`,…,`$expr_n$`)` is well-typed if `id` is a defined or primitive function of $n$ parameters of types $type_1, \ldots, type_n$ and $expr_i$ is of type $type_i$. The type of the function call is the return type of `id`.
- A conditional `IF` $expr_1$ `THEN` $expr_2$ `ELSE` $expr_3$ `FI` is well-typed if $expr_1$ has type `bool` and $expr_2$ and $expr_3$ have the same type. The type of the conditional is the common type of $expr_2$ and $expr_3$.

  An assertion `IF` $expr_1$ `THEN` $expr_2$ `FI` is well-typed if $expr_1$ has type `bool`. The type of the assertion is the type of $expr_2$.

## Evaluation

*This section is especially relevant to milestone 3.*

We describe the evaluation of µ-Opal programs only for context-correct programs. The evaluation of a µ-Opal program is the transformation of an expression to a value (natural number or boolean).
A µ-Opal program returns the value of the right-hand side of the definition `MAIN`.
The value of an expression is defined as follows:

- A number or boolean denotes its own value.
- Assume a function call `id(`$expr_1$`, …, `$expr_n$`)` and let $val_i$ be the values of the argument $expr_i$.
  - If `id` is a primitive function the value of the call is the result returned by the predefined implementation of the primitive function with $val_1, \ldots, val_n$ as input.
  - If `id` is a defined function with right-hand side $expr$ and parameters $x_1,\ldots,x_n$ the value of the function call is $[x_1 \mapsto val_1, \ldots, x_n \mapsto val_n]expr$.
- A conditional `IF` $expr_1$ `THEN` $expr_2$ `ELSE` $expr_3$ `FI` yields the value of $expr_2$ if $expr_1$ has the value `true`. Otherwise, it yields the value of $expr_3$.

  If the `ELSE`-branch is missing and $expr_1$ has the value `false` this is an error.

## The µ-Opal compiler template

Here are some instructions how to set up your source tree for the µ-Opal compiler.

1. Download the µ-Opal compiler template from the ISIS page of the course.
2. Extract the archive and rename the resulting directory `MOCTemplate` into ⟨*group number*⟩.
3. To compile the source tree issue the following command in the directory created in the previous step ($ represents the shell prompt):

   ```
   $ sbt compile
   ```

   You need a working installation of the Simple Build Tool (SBT) for this. The ISIS page provides the necessary information.

   Make sure that you have a working internet connection since SBT has to download several dependencies.
4. Create the Eclipse project files for the Scala IDE:

   ```
   $ sbt eclipse
   ```

   Launch your installation of the Scala IDE and import the µ-Opal compiler into your workspace. To check that everything went fine, try to start the µ-Opal compiler. Open the file `Main.scala` and hit the *Run* button. You should see the following output in the *Console* view:

   ```
   ERROR at global: no file found

   usage: <moc> [ -d ] [ -i ] [-S ] <source>.mo
     -d  enables the debug option
     -i  enables the interpreter option (the compiler will only start the interpreter)
     -S  the coder writes the code for the machine in a the file <source>.S
   ```
5. You can start the µ-Opal compiler from the command line. First generate a start script by

   ```
   $ sbt start-script
   ```

   You can now run the compiler by issuing `target/start` in the root directory. You can pass compiler options to this script. For example, you can try to run the program `fac.mo` contained in the template:

   ```
   $ target/start src/test/resources/parser/fac.mo
   ```

   Since the parser is not yet implemented you get the following answer:

   ```
   ERROR at global: Parser not yet implemented
   ```

*Note:* Instead of giving commands to SBT as arguments you can also start it by typing `sbt` and enter the commands at the SBT shell. This saves you the startup time of SBT.

### The test framework

The µ-Opal compiler contains a test framework using ScalaTest. You can uses it for your own test cases. The directory `src/test/resources` contains a subdiretory for each phase of the compiler where you can deposit your tests.
A test consists of two files: an input file ⟨*test*⟩`.mo` containing a valid or invalid µ-Opal source program and a file ⟨*test*⟩`.mo.expected` containing the expected output.
The format of the `.mo.expected` file depends on the phase to check:

**Parser, context checker** If the input file is correct and should be accepted by your parser/context checker, the `.mo.expected` file only contains `SUCCESS`. If the input file contains syntax/context errors, the `.mo.expected` file contains the exact error message(s) your parser/context checker returns.

**Interpreter, code generator** If the evaluation (by the interpreter) or the execution (of the generated code) produces a valid result, the `.mo.expected` file contains two lines

`SUCCESS`

⟨*result*⟩

For example, the file `src/test/resources/interpreter/fac.mo.expected` contains these lines:

```
SUCCESS
3628800
```

If the source program leads to a runtime error, the `.mo.expected` file contains that exact error message.

**Running the tests** You can run all your tests by the command

```
$ sbt test
```

If you try this on the plain template you obtain a report like the following:

```
[info] ParserTest:
[info] MOpal Parser
[info] - Parsing of fac.mo should succeed *** FAILED ***
[info]   Received the error
[info]   ERROR at global: Parser not yet implemented (ParserTest.scala:54)

[...]

[info] ContextCheckerTest:
[info] MOpal Context Checker
[info] - Context checking of fac.mo should succeed *** FAILED ***
[info]   Received the error
[info]   ERROR at global: Parser not yet implemented (ContextCheckerTest.scala:54)
[info] Run completed in 612 milliseconds.
[info] Total number of tests run: 4
[info] Suites: completed 4, aborted 0
[info] Tests: succeeded 0, failed 4, canceled 0, ignored 0, pending 0
[info] *** 4 TESTS FAILED ***
```

Since nothing is implemented nothing can succeed…
If you just want to run the tests of a certain phase, you can use the `testOnly` command:

```
$ sbt "testOnly de.tuberlin.uebb.comp1.moc.tests.InterpreterTest"
```

(Mind the quotes, they are important.)
Replace `Interpreter` by `Parser`, `ContextChecker`, or `CodeGenerator` to run the other collections of tests.

## The target virtual machine COVM

*This section is especially relevant to milestone 4.*

**General Notes** The COVM (Compiler Construction 1 Virtual Machine) is a simple stack machine. Its primitive data types are integers and tuples and it supports conditional branches and function calls. The value stack is an empty or non-empty sequence of words:

| | | | |
|---|---|---|---|
| $ws$ | $::=$ | $[]$ | empty stack |
| | | $w : ws$ | non-empty stack |
| $w$ | $::=$ | $n$ | integers |
| | | $a$ | addresses |
| | | $\langle w, ..., w \rangle$ | tuples |

A word is either an integer $n$, an address of the code segment $a$ (a non-negative integer), or a tuple of words $\langle w_0, ..., w_m \rangle$.
A machine state is a triple of a non-negative integer $ip$ (the instruction pointer), a code segment $C$ and a value stack $ws$:

$$state \subseteq ip \times C \times ws$$

A stack machine programm is an instruction sequence where the first element is the start instruction. The initial state for a code segment $instr_0 \cdots instr_m$ is

$$(0, instr_0 \cdots instr_m, []).$$

A successful final state of machine has a stack of size one. A failure during execution will stop the machine immediately.

### Instructions

The COVM has 19 instructions. Each instruction is described in prose and by the corresponding state transition $state_1 \rightarrow state_2$.
By the notation $C_k[instr]$ for a state $(ip, C, ws)$ we indicate that $instr$ is the $k$-th instruction in the code segment $C$. For example, $(ip, C_{ip}[call], ws)$ means that the instruction pointer references a *call* instruction. In the COVM, the instruction pointer always points to the instruction that will be executed in the next step.

#### Arithmetic instructions

The arithmetic instruction do not influence the linear control flow. Therefore, the instruction pointer is always incremented by one to continue with the next instruction.
All arithmetic instructions expect two integers on top of the stack. This is indicated in the state transitions by the variable names starting with $n$ which stands for integer words.

*add* Add the first two stack words.

$$(ip, C_{ip}[add], n_0 : n_1 : ws) \rightarrow (ip + 1, C, n_1 + n_0 : ws)$$

The machine aborts if there is an integer overflow.

*sub* Subtract the first word from the second word of the stack.

$$(ip, C_{ip}[sub], n_0 : n_1 : ws) \rightarrow (ip + 1, C, n_1 - n_0 : ws)$$

The machine aborts if there is an integer underflow.

*mul* Multiply the first two words of the stack.

$$(ip, C_{ip}[mul], n_0 : n_1 : ws) \rightarrow (ip + 1, C, n_1 \cdot n_0 : ws)$$

The machine aborts if there is an integer overflow.

*div* Divide the second word by the first word of the stack.

$$(ip, C_{ip}[div], n_0 : n_1 : ws) \rightarrow (ip + 1, C, n_1 \div n_0 : ws)$$

The machine aborts if there is a division by zero.

### Push and pop stack words

Similar to the arithmetic instructions all of the following instructions do not modify the linear control flow.

*pushint n* Push an integer constant $n$ onto the stack.

$$(ip, C_{ip}[pushint\ n], ws) \rightarrow (ip + 1, C, n : ws)$$

*pushaddr a* Push an address of an instruction of the code segment onto the stack.

$$(ip, C_{ip}[pushaddr\ a], ws) \rightarrow (ip + 1, C, a : ws)$$

*push i* Copy the $i$-th stack word to the top of the stack.

$$(ip, C_{ip}[push\ i], w_0 : \cdots : w_i : ws) \rightarrow (ip + 1, C, w_i : w_0 : \cdots : w_i : ws)$$

*slide i* Remove $i$ words $w_1, ..., w_i$ from the top of the stack but keep the first one $w_0$. This instruction is handy to remove arguments to a function from the stack.

$$(ip, C_{ip}[slide\ i], w_0 : w_1 : \cdots : w_i : ws) \rightarrow (ip + 1, C, w_0 : ws)$$

*swap* Swap the first two words of the stack.

$$(ip, C_{ip}[swap], w_0 : w_1 : ws) \rightarrow (ip + 1, C, w_1 : w_0 : ws)$$

*pack i* Pack $i$ values into a tuple.

$$(ip, C_{ip}[pack\ i], w_0 : \cdots : w_{i-1} : ws) \rightarrow (ip + 1, C, \langle w_{i-1}, ..., w_0 \rangle : ws)$$

*unpack i* Push the $i$-th component of a tuple onto the stack.

$$(ip, C_{ip}[unpack\ i], \langle w_0, ..., w_i, ..., w_m \rangle : ws) \rightarrow (ip + 1, C, w_i : ws)$$

### Control flow instructions

Control flow instruction like jumps and function calls and returns set the instruction pointer in various ways.

*call* Call the function the top word $a$ of the stack points to (by setting the instruction pointer to $a$) and saves the return address on the stack. The return address is the address of the instruction following the *call*, that is, $ip + 1$.

$$(ip, C_{ip}[call], a : ws) \rightarrow (a, C, ip + 1 : ws)$$

*ret* Return from a function call by setting the instruction pointer to the return address which is the second word on the stack. The result of the function $w$ (the first word of the stack) is kept.

$$(ip, C_{ip}[ret], w : a : ws) \rightarrow (a, C, w : ws)$$

*jmp a* Jump to the absolute address $a$.

$$(ip, C_{ip}[jmp\ a], ws) \rightarrow (a, C, ws)$$

*jz a* Jump to the absolute address $a$ if the top level word is zero. Continue with the next instruction if this word is not zero. The top-most word of the stack is removed in either case.

$$\begin{aligned} &(ip, C_{ip}[jz\ a], 0 : ws) \rightarrow (a, C, ws) && \\ &(ip, C_{ip}[jz\ a], n : ws) \rightarrow (ip + 1, C, ws) && \text{if } n \neq 0 \end{aligned}$$

*jlt a* Jump to the address $a$ if the top level word is less than zero. Continue with the next instruction if this word is greater than or equal to zero. The top-most word of the stack is removed in either case.

$$\begin{aligned} &(ip, C_{ip}[jlt\ a], n : ws) \rightarrow (a, C, ws) && \text{if } n < 0 \\ &(ip, C_{ip}[jlt\ a], n : ws) \rightarrow (ip + 1, C, ws) && \text{if } n \geq 0 \end{aligned}$$

*jgt a* Jump to address $a$ if the top level word is greater than zero. Continue with the next instruction if this word is less than or equal to zero. The top-most word of the stack is removed in either case.

$$\begin{aligned} &(ip, C_{ip}[jgt\ a], n : ws) \rightarrow (a, C, ws) && \text{if } n > 0 \\ &(ip, C_{ip}[jgt\ a], n : ws) \rightarrow (ip + 1, C, ws) && \text{if } n \leq 0 \end{aligned}$$

*stop* Stop the machine. The current state must be a valid final state.

*abort s* Abort execution with an error message $s$.

**A small example**

Consider this small µ-Opal program:

```
DEF MAIN: nat == foo(1, 5, 3)
DEF foo(x: nat, y: nat, z: nat): nat == add(x, sub(y, z))
```

A possible translation of this program to COVM instructions is the following code. Each instruction is annotated with the stack that is active before the instruction is executed, that is, *ip* points to that instruction. Addresses on the stack are prefixed with `@` to avoid confusion with integers.

```
--  current instruction   stack
--  Code of MAIN
00  pushint 1             -- stack: []
01  pushint 5             -- stack: 1 : []
02  pushint 3             -- stack: 5 : 1 : []
03  pushaddr 7            -- stack: 3 : 5 : 1 : []
04  call                  -- stack: @7 : 3 : 5 : 1 []
05  slide 3               -- stack: 3 : 3 : 5 : 1 : []
06  stop                  -- stack: 3 : []

--  Code of foo
```

```
07  push 3                 -- stack: @5 : 3 : 5 : 1 : []
08  push 3                 -- stack: 1 : @5 : 3 : 5 : 1 : []
09  push 3                 -- stack: 5 : 1 : @5 : 3 : 5 : 1 : []
10  sub                    -- stack: 3 : 5 : 1 : @5 : 3 : 5 : 1 : []
11  add                    -- stack: 2 : 1 : @5 : 3 : 5 : 1 : []
12  ret                    -- stack: 3 : @5 : 3 : 5 : 1 : []
```

The three arguments for the function `foo` are pushed onto the stack in instructions `00` to `02`. The function `foo` starts at address `07` in the code segment. In order to call `foo`, its address `@7` is pushed onto the stack in instruction `03`. The call in instruction `04` jumps to this address and pushes the return address `@5` onto the stack. After the call, the three arguments for `foo` are popped from the stack by the *slide* instruction in `05`. The machine stops with a valid final configuration in instruction `06`.
The function `foo` first pushes `x` onto the stack. `x` is word number 3 of the stack as the return address is the top-most word. Then the arguments `y` and `z` for `sub` are pushed onto the stack. Since the stack grows while pushing arguments, the position of each argument happens to be 3. After execution of `sub` and `add`, the `ret` instruction jumps back to the return address `@5` and removes it from the stack.

## The COVM implementation

The COVM implementation is contained in the µ-Opal compiler template and started by the compiler to execute the generated code. However, it is also possible to run the COVM stand-alone, similar to the Java Virtual Machine implementation `java`.
The ISIS page provides the source distribution of the COVM. You can compile is using SBT and create a start script with the `start-script` SBT command.
The COVM takes a text file containing a sequence of instructions as argument. For example, let `foo.S` be the code of the previous section:

```
$ target/start foo.S
Result: 3
```

If you encounter a `Stack overflow` or an `Out of heap memory`[1] exception you can increase the respective values with the command line switches `-h` and `-s`. The default values are 4096 words for the heap and 512 for the stack.
The COVM implementation offers a trace mode that shows the current instruction pointer (and its target) and the content of the stack. It is enabled by the `-d trace` flag and produces an output similar to the comments in the listing of the previous section.

```
$ target/start -d trace foo.S
ip = 0     { pushint 1 }      []
ip = 1     { pushint 5 }      1 : []
ip = 2     { pushint 3 }      5 : 1 : []
ip = 3     { pushaddr 7 }     3 : 5 : 1 : []
ip = 4     { call }           @7 : 3 : 5 : 1 : []
ip = 7     { push 3 }         @5 : 3 : 5 : 1 : []
ip = 8     { push 3 }         1 : @5 : 3 : 5 : 1 : []
ip = 9     { push 3 }         5 : 1 : @5 : 3 : 5 : 1 : []
ip = 10    { sub }            3 : 5 : 1 : @5 : 3 : 5 : 1 : []
ip = 11    { add }            2 : 1 : @5 : 3 : 5 : 1 : []
ip = 12    { ret }            3 : @5 : 3 : 5 : 1 : []
ip = 5     { slide 3 }        3 : 3 : 5 : 1 : []
ip = 6     { stop }           3 : []
Result: 3
```

[1]You might be surprised to read of a heap since COVM was announced as a stack machine. It is a stack machine as opposed to a register machine but it needs a heap to store the actual values. All the details will be discussed in the memory management chapter of the lecture.