Compiler Construction 1
Pepper · Rohloff

Summer term 2014
Handout: May 15, 2014
Submission: June 4, 2014

Milestone 1

Parser for μ-Opal

Assignment Parser
Implement a parser for the language μ-Opal.

a) The EBNF of μ-Opal is written with readability in mind, not implementation. Transform it into an equivalent BNF.

b) Design an abstract syntax for μ-Opal.

c) Annotate your BNF with semantic actions, eliminate left-recursive productions, and perform left-factorization to obtain a grammar suitable for top-down parsing. Include the resulting grammar in your submission as a text or PDF file.

d) Implement a recursive-descent parser (combinator-based or manually written) with *minimal* error handling.

e) Write several test cases to demonstrate the functionality of your parser, especially error detection. Document for each test case the expected result as a comment in the `.mo`-file.

Hints for all milestones
The compiler project consists of the implementation of a parser, context checker, interpreter, and code generator for μ-Opal. A scanner as well as other parts of the compiler infrastructure are available from the ISIS page of the course.
The project is divided into four milestones:

Milestone 1 (June 4, 2014 10am) Implementation of the parser (25 credits)

Milestone 2 (June 18, 2014 10am) Implementation of the context checker (25 credits)

Milestone 3 (July 2, 2014 10am) Implementation of the interpreter (25 credits)

Milestone 4 (July 16, 2014 10am) Implementation of the code generator (25 credits)

For each milestone you are requested to submit an archive file ⟨*group number*⟩`.zip` on ISIS. Please make sure that the archive extracts to a subfolder ⟨*group number*⟩.
To gain full credit, the archive must contain

- all source files to compile that particular milestone with `sbt`, thoroughly commented, and
- several test cases as `.mo`-files containing μ-Opal code and comments on the expected test result.

The `target` and `bin`-subdirectories and other temporary files do *not* belong into the archive.

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 The context-free syntax of µ-Opal is specified by the following EBNF:

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

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 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 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 numbers of booleans).
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 $[x_1 \mapsto val_1, ..., x_n \mapsto val_n]expr$.
- 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.