Languages and Paradigms of Programming

Language specification

Filip Binkiewicz (332069)

April 13, 2017

1 BNF Grammar

$$
\begin{aligned}
\mathit{Num} \ni\ & n & ::=\ & 0 \mid 1 \mid -1 \mid 2 \mid \ldots \\
\mathit{Var} \ni\ & x & ::=\ & \texttt{x} \mid \texttt{y} \mid \ldots \\
\mathit{Expr} \ni\ & e & ::=\ & n \mid x \mid e_1 + e_2 \mid e_1 * e_2 \mid e_1 - e_2 \mid e_1 / e_2 \mid \\
& & & f(x) \mid f(f_1) \mid x{+}{+} \mid x{-}{-} \\
\mathit{BExpr} \ni\ & b & ::=\ & \texttt{true} \mid \texttt{false} \mid x \mid e_1 <= e_2 \mid e_1 < e_2 \mid \\
& & & e_1 >= e_2 \mid e_1 > e_2 \mid e_1 == e_2 \mid \\
& & & b_1 \texttt{ and } b_2 \mid b_1 \texttt{ or } b_2 \mid \texttt{not } b \\
\mathit{FExpr} \ni\ & f & ::=\ & x \mid \texttt{lambda (}x\texttt{): } I \\
\mathit{Instr} \ni\ & I & ::=\ & x = e \mid x = b \mid x = f \mid \\
& & & \texttt{while } b \texttt{ do } I \mid I_1; I_2 \mid \\
& & & \texttt{if } b \texttt{ then } I \mid \texttt{if } b \texttt{ then } I_1 \texttt{ else } I_2 \mid \\
& & & \{\ d\ \texttt{;}\ I\ \} \mid \texttt{print } e \mid \\
& & & \texttt{for } x \texttt{ ranging from } e_1 \texttt{ to } e_2 \texttt{ do } I \mid \\
& & & \texttt{return } e \mid \texttt{yield } e \\
\mathit{Decl} \ni\ & d & ::=\ & \texttt{int } x \mid \texttt{bool } x \mid \\
& & & \texttt{function } y\texttt{(}x\texttt{) } I \mid \texttt{function } y\texttt{(ref } x\texttt{) } I \mid \\
& & & \texttt{function } y\texttt{(fun } x\texttt{) } I
\end{aligned}
$$

2 Remarks

- Most of the constructions described in the grammar are standard and won't be explained further — their meaning can be assumed to repli-

cate their C's equivalents'. All non-trivial constructions are explained in the next section.

- This grammar has not yet been written in a format understood by `BNFC`, but it will be done shortly. The format imitates the one used in the Semantics and Verification class — in particular, it does not take account for precedence level in expressions, and I omitted that on purpose, so as to make the grammar more human-readable. Rewriting it might imply some minor changes.

3 Explanation

Most of the features of this language have been taken from the standard list. Below I will describe the interesting features of this language.

- General information.
  - **One shall not use an undeclared variable!** (this is however not a syntax error as it is not a context-free feature)
  - The language is **statically typed**, meaning exactly what the author of the assignment described so verbosely.
  - Runtime errors are explicitly reported.
- `if _ then _ else` - since there are two versions of this statement, the problem that bothered Pascal's developers appears. We assume that every `else` corresponds to the **closest** `if`, i.e. the program

```
if false then
        if true then
                print 1
        else
                print 1
```

  does not print anything. But, of course, I endorse all programers not to take advantage of this too much.
- `function`. This language allows to declare three types of functions (all recursive with static visibility):
  - pass-by-value
  - pass-by-reference

– functions with a function parameter (pass-by-value)

All functions **return an integer value** (0 if a `return` statement is not met int the function's body).

Another important remark is that a function declared as `function f(fun f)` discards the parameter and inside it every call of `f` will be a recursive call. Other than that, this is self-explainatory (I hope!).

- `lambda` denotes an anonymous function which can be assigned to a variable or passed to a function. Lambdas are only pass-by-value and return an integer.
- The Pascal-style `for` — I believe this is self-explainatory as well.
- `print` allows to print an expression (an attempt to print a function variable will raise a ,,compile" error.
- Blocks work like in C, except that there needs to be at least one semicolon - the one separating declarations from instructions. Hence, the following code is fine:

```
{ ; print 10 }
```

- `yield` works like in Python and is the only feature not described in the standard assignment specification. It allows the programmer to define rather stupid generators:

```
function f(x)
{
        int i;
        i = 0;
        while (true)
        {
                yield x + i;
                i = i + 1
        }
}
;
print f(1);             // prints 1
print f(1);             // prints 2
print f(3)              // prints 5
```

A `yield` instruction outside of a function definition raises a runtime error.

- Comments - this will be done by `BNFC` and I think I'm going to stick with the `// comment` and `/* comment */` tradition.