The “Alan TurNing” Programming Language

Description

Alan TurNing is an imperative programming language with a twist.
Major features:

- static typing with boolean and integer
- static binding with variable shadowing
- expression printing
- declaration and statement blocks
- nested, recursive functions
- passing function arguments by reference or by value on caller’s decision
- C-like statement expressions (eg. +=, ++, =)
- **changing direction of code execution**

The last feature is what differentiates TurNing from other programming languages.
It allows programmer to decide whether their code should be executed left-to-right
(or top-down if you will).
The “turn” statement changes current direction, while left and right mark the statements as executable only in the specified path. Even if we had not have recursive functions built in, we could still easily mimic a while loop, making our language Turing (and TurNing) complete.

Notes

The language is going to have pretty complex continuational semantics. Despite my best efforts it may cause problems I am not fully aware of at the moment. I might change some details in intended behaviour during the course of the project.
Pesimistically I would like to have an option to switch to more standard language if I get overwhelmed by TurNing.

Syntax

Below is the syntax of the TurNing language written in LBNF formalism used in BNF Converter. It might contain mistakes and still not be suitable for BNFC, as we are not familiar with type checking and syntax determinisation yet.

```
comment “#” ;
comment “//” ;
comment “/*” “*/” ;

Prog. Prog ::= [Stmt] ;

terminator nonempty Stmt “;” ;
```

Declarations

```
DDef.  Decl  ::= Type Ident ;
DVal.  Decl  ::= Type Ident “=” Exp ;
DFunc. Decl  ::= Type Ident “(“ [Param] “)” “{“ [Decl] [Stmt] “}” ;
Param. Param ::= Type Ident ;

separator  Param “,” ;
terminator Decl “;” ;
```

Types

```
TInt.  Type ::= “int” ;
TBool. Type ::= “bool” ;
```

Statements

```
SSkip.  Stmt ::= “skip” ;
SIf.    Stmt ::= “if” “(“ Exp “)” “{“ [Stmt] “}” ;
SIfte.  Stmt ::= “if” “(“ Exp “)” “{“ [Stmt] “}” “else” “{“ [Stmt] “}” ;
SLeft.  Stmt ::= “left” Stmt ;
SRight. Stmt ::= “right” Stmt ;
STurn.  Stmt ::= “turn” ;
SExp.   Stmt ::= Exp ;
SRet.   Stmt ::= “return” Exp ;
SPrint. Stmt ::= “print” Exp ;
SBlock. Stmt ::= “{“ [Decl] [Stmt] “}”
_.      Stmt ::= “(“ [Stmt] “)”
```

Expressions

```
EAss.    Exp ::= Ident “=” Exp ;
EAddAss. Exp ::= Ident “+=” Exp ;
ESubAss. Exp ::= Ident “-=” Exp ;
EMulAss. Exp ::= Ident “*=” Exp ;
EDivAss. Exp ::= Ident “/=” Exp ;
EModAss. Exp ::= Ident “%=” Exp ;

EIfte. Exp ::= Exp1 “?” Exp “:” Exp ;

EOr.  Exp1 ::= Exp1 “||” Exp2 ;

EAnd. Exp2 ::= Exp2 “&&” Exp3 ;

EEq.  Exp3 ::= Exp3 “==” Exp4 ;
ENEq. Exp3 ::= Exp3 “!=” Exp4 ;

ELT.  Exp4 ::= Exp4 “<” Exp5 ;
ELEq. Exp4 ::= Exp4 “<=” Exp5 ;
EGT.  Exp4 ::= Exp4 “>” Exp5 ;
EGEq. Exp4 ::= Exp4 “>=” Exp5 ;

EAdd. Exp5 ::= Exp5 “+” Exp6 ;
ESub. Exp5 ::= Exp5 “-” Exp6 ;

EMul. Exp6 ::= Exp6 “*” Exp7 ;
EDiv. Exp6 ::= Exp6 “/” Exp7 ;
EMod. Exp6 ::= Exp6 “%” Exp7 ;

ENot. Exp7 ::= “!” Exp8 ;
ENeg. Exp7 ::= “-” Exp8 ;

EFunc. Exp8 ::= Ident “(“ [Arg] “)” ;
AVal.  Arg  ::= Ident ;
ARef.  Arg  ::= “ref” Ident ;

separator Arg “,” ;

EInt.   Exp8 ::= Integer ;
ETrue.  Exp8 ::= “true” ;
EFalse. Exp8 ::= “false” ;

EPreInc.  Exp8 ::= “++” Ident ;
EPostInc. Exp8 ::= Ident “++” ;
EPreDec.  Exp8 ::= “--” Ident ;
EPostDec. Exp8 ::= Ident “--” ;
EVar.     Exp8 ::= Ident ;

coercions Expr 8 ;
```

Examples

Euclides' algorithm:

```
{
	int x = 42;
	int y = 12;
	int gcd (int a, int b) {
		int c;
		left turn;
		if (b != 0) {
			right (
				c = a % b;
				a = b;
				b = c;
			);
			turn;
		};
		return a;
	};
	print (gcd (x, y));
};
```

Overcomplicated recursive factorial:

```
{
	int n = 42;
	bool runFact (int n) {
		int fact (int n) {
			int prev;
			if (n == 0) {
				return 1;
			} else {
				prev = fact (n – 1);
				return prev *= n;
			};
		};
		if (n >= 0) {
			n = fact(n);
			return true;
		} else {
			return false;
		};
	}
	print (runFact (ref n)) ? n : (-1);
};
```

Inspiration

The language was inspired by one of the continuational semantics exercises that were done during the Programs' Semantics and Verification course.