Description of the language features, PLP

April 11, 2022

Text description of the language

I am implementing an interpreter for an imperative language, roughly a small subset of C with similar syntax. In addition to some fundamental features of imperative languages, it includes Cpp-style const-correctness of variables and variable references as well as try-except blocks enabling to raise and catch named exceptions. The precise syntax and expected semantics of the language's features is shown in the attached sample programs. I provide fully correct programs, programs with syntax errors, and programs inducing typecheck errors. The attached test script demonstrates the expected output of the test programs. Since the only features of the language that are not fully standard for this task are the const-correctness and exceptions, I will briefly describe their semantics. Almost all the features are also described in detail by the language's continuational semantics, which can be found below.

const-correctness

A variable can be declared to be const:

```
const int x = 3;
```

A function argument can also be declared to be const:

```
def int some_function(const int x){ declarations, instructions }
```

The values of const variables cannot be modified, i.e. an assignment to a const variable is not possible, and passing a reference to this variable to a function as an argument which was not declared to be const is also not possible (because otherwise the function call could change the value of the variable). Thus, in this environment, the variable `x` can be passed to `some_function` by reference:

```
int y = some_function(ref x)
```

However, if the function definition looked like this:

```
def int some_function(int x){ declarations, instructions }
```

this would be a violation of const-correctness.

Exceptions

The mechanism is well-known, and in this case the only difference to Java-style or Cpp-style exceptions is that, rather than being objects, they are simply identified by their identifiers. Try-except blocks have non-empty list of exception identifiers, and, if a `raise` instruction with one of these identifiers is encountered while executing the block, program execution moves to the beginning of the `except` instruction. Exception names can be shadowed.

Feature table as described in the task

For 15 pts:

- Three types: int, str, bool
- Literals, arithmetic, comparisons
- variables, assignment
- print
- while, if
- functions with recursion
- passing by value and reference

For 20 pts:

- Shadowing, static binding
- Management of runtime errors
- Returning values from functions

Additionally:

- 4 pts: static typing
- 2 pts: arbitrarily nested functions with static binding
- 1 pt: break, continue
- 2 pts: try-catch blocks with named exceptions
- 1 pt: const-correctness

In total, I hope to be awarded 30 points if all the features are implemented correctly.

Note on argument passing

The function

$$\texttt{def int} f(\texttt{int } x)...$$

Does not know whether an argument will be passed by value:

$$y = f(2137)$$

or by reference:

$$y = f(\texttt{ref } x)$$

Both options are possible, and the appropriate behavior of function call will be ensured at calling.

Language grammar in EBNF

```
 - program
Prog.  Program ::= [FunDecl] Main;

- declarations
DMain.  Main ::= "main" Block ;
DFun.  FunDecl ::= "def" Type Ident "(" [ArgDecl] ")" Block;
DFunDecl.  Decl ::= FunDecl ;
DVarDecl.  Decl ::= Type Ident "=" Exp ";" ;
DConstVarDecl.  Decl ::= "const" Type Ident "=" Exp ";" ;
DArgDecl.  ArgDecl ::= Type Ident ;
DConstArgDecl.  ArgDecl ::= "const" Type Ident ;
separator ArgDecl "," ;

- instructions IBlock.  Block ::= "" [Decl] [Instr] "" ";" ;
IAss.  Instr ::= Ident "=" Exp ";" ;
IIf.  Instr ::= "if" Exp "then" Instr Else "fi" ";" ;
IElseEmpty.  Else ::= "" ;
IElse.  Else ::= "else" Instr ;
IWhile.  Instr ::= "while" Exp "do" Instr ;
IPrStr.  Instr ::= "print" Exp ";" ;
IRet.  Instr ::= "return" Exp ";" ;
IBreak.  Instr ::= "break" ";" ;
IRaise.  Instr ::= "raise" Ident ";" ;
IContinue.  Instr ::= "continue" ";" ;
ITryBlock.  Instr ::= "try" Block "except" Ident [Ident] Block ;
IBlockInstr.  Instr ::= Block ;
IExp.  Instr ::= Exp ";" ;
separator Instr "" ;

- expressions

EOr.  Exp ::= Exp1 "or" Exp ;
EAnd.  Exp1 ::= Exp2 "and" Exp1 ;
EEq.  Exp2 ::= Exp3 "==" Exp3 ;
```

```
ENeq.  Exp2 ::= Exp3 "!=" Exp3 ;
ELeq.  Exp2 ::= Exp3 "<=" Exp3 ;
EGeq.  Exp2 ::= Exp3 ">=" Exp3 ;
ELess.  Exp2 ::= Exp3 "<" Exp3 ;
EGrt.  Exp2 ::= Exp3 ">" Exp3 ;
EPlus.  Exp3 ::= Exp3 "+" Exp4 ;
EMinus.  Exp3 ::= Exp3 "-" Exp4 ;
EConcat.  Exp3 ::= Exp3 "" Exp4 ;
ETimes.  Exp4 ::= Exp4 "*" Exp5 ;
EDiv.  Exp4 ::= Exp4 "/" Exp5 ;
ENeg.  Exp5 ::= "-" Integer ;
ENot.  Exp5 ::= "not" Exp6 ;
EInt.  Exp6 ::= Integer ;
EBool.  Exp6 ::= Bool ;
EStr.  Exp6 ::= String ;
EIdent.  Exp6 ::= Ident ;
EFuncall.  Exp6 ::= Ident "(" [Arg] ")" ;
coercions Exp 6 ;

separator Arg "," ;
AVal.  Arg ::= Exp ;
ARef.  Arg ::= "ref" Ident ;

separator Ident "," ;
separator Decl "" ;
separator FunDecl "" ;

comment "//" ;
comment "/*" "*/" ;


TBool.  Type ::= "bool" ;
TInt.  Type ::= "int" ;
TStr.  Type ::= "str" ;
BTrue.  Bool ::= "true" ;
BFalse.  Bool ::= "false" ;
```

Continuation-style denotational semantics of the language

This semantics is given in a simplified form, assuming the only handled/returned values are numerals. In addition, some mental shortcuts have been made, so it is not fully formal. The `print` instruction is omitted altogether.

Semantic domains

$$\mathrm{State} = \mathrm{Loc} \longrightarrow (\mathrm{Int} \cup \mathrm{Bool} \cup \mathrm{Str})$$

$$\text{Ans} = \text{Error} \times \text{State}$$

Variable environment:

$$\text{Env} = \text{Var} \longrightarrow \text{Loc}$$

Function environment:

$$\text{FEnv} = \text{FName} \rightharpoonup \text{Fun}$$

where

$$\text{Fun} = \text{Arg} \longrightarrow \text{Cont}_E \longrightarrow \text{Env} \longrightarrow \text{FEnv} \longrightarrow \text{EEnv} \rightharpoonup \text{Cont}$$

And the type Arg, denoting an argument passed at function call, is:

$$\text{Arg} \longrightarrow \epsilon \mid \text{Arg}, \text{Arg} \mid \texttt{Expr} \mid \texttt{ref Var}$$

Exception environment:

$$\text{EEnv} = \text{EName} \rightharpoonup \text{Cont}$$

Continuation types

Instruction continuation:

$$\text{Cont} = \text{State} \rightharpoonup \text{Ans}$$

Arithmetic expression continuation:

$$\text{Cont}_E = \text{Int} \longrightarrow \text{State} \rightharpoonup \text{Ans}$$

Boolean expression continuation:

$$\text{Cont}_B = \text{Bool} \longrightarrow \text{State} \rightharpoonup \text{Ans}$$

String expression continuation:

$$\text{Cont}_S = \text{Str} \longrightarrow \text{State} \rightharpoonup \text{Ans}$$

Declaration continuation:

$$\text{Cont}_D = \text{Env} \longrightarrow \text{FEnv} \longrightarrow \text{State} \rightharpoonup \text{Ans}$$

Function argument declaration continuation:

$$\text{Cont}_A = \text{Env} \longrightarrow \text{State} \rightharpoonup \text{Ans}$$

Types of semantic functions

Instructions:

$$\mathcal{I}[\![]\!] : \text{Instr} \longrightarrow \text{Env} \longrightarrow \text{FEnv} \longrightarrow \text{EEnv} \longrightarrow \text{Cont} \longrightarrow \text{Cont}_E \longrightarrow \text{Cont} \longrightarrow \text{Cont} \longrightarrow \text{State} \rightharpoonup \text{Ans}$$

Four continuations are passed here: the first two are responsible for 'semicolons' and handling the `return` statement, and the other two are respectively responsible for the `break` and `continue` instructions.

Arithmetic expressions:

$$\mathcal{E}[\![]\!] : \text{Expr} \longrightarrow \text{Env} \longrightarrow \text{FEnv} \longrightarrow \text{EEnv} \longrightarrow \text{Cont}_E \longrightarrow \text{State} \rightharpoonup \text{Ans}$$

Boolean expressions:

$$\mathcal{B}[\![]\!] : \text{BExpr} \longrightarrow \text{Env} \longrightarrow \text{FEnv} \longrightarrow \text{EEnv} \longrightarrow \text{Cont}_B \longrightarrow \text{State} \rightharpoonup \text{Ans}$$

String expressions:

$$\mathcal{S}[\![]\!] : \text{SExpr} \longrightarrow \text{Env} \longrightarrow \text{FEnv} \longrightarrow \text{EEnv} \longrightarrow \text{Cont}_S \longrightarrow \text{State} \rightharpoonup \text{Ans}$$

Declarations:

$$\mathcal{D}[\![]\!] : \text{Decl} \longrightarrow \text{Env} \longrightarrow \text{FEnv} \longrightarrow \text{EEnv} \longrightarrow \text{Cont}_D \longrightarrow \text{State} \rightharpoonup \text{Ans}$$

Argument declarations:

$$\mathcal{A}[\![]\!] : \text{Argdecl} \longrightarrow \text{Arg} \longrightarrow \text{Env} \longrightarrow \text{Cont}_A \longrightarrow \text{Env} \longrightarrow \text{FEnv} \longrightarrow \text{EEnv} \longrightarrow \text{State} \rightharpoonup \text{Ans}$$

Programs:

$$\mathcal{P}[\![]\!] : \text{Prog} \longrightarrow \text{Ans}$$

Denotations

Declarations

$$\mathcal{D}[\![\text{int } x = e]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_D\ s = \mathcal{E}[\![e]\!]\ \rho\ \rho_F\ \rho_E\ (\lambda\ n.\lambda\ s'.\ \kappa_D\ \rho[x \mapsto l]\ \rho_F\ \rho_E\ s'[l \mapsto n])\ s$$

where

$$l = \text{newloc } s'$$

$$\mathcal{D}[\![d1; d2]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_D\ s = \mathcal{D}[\![d1]\!]\ \rho\ \rho_F\ \rho_E\ (\lambda\ \rho'.\ \lambda\ \rho'_F\ \lambda\ s'.\ \mathcal{D}[\![d2]\!]\ \rho'\ \rho'_F\ \rho_E\ \kappa_D\ s'\ )\ s$$

$$\mathcal{D}[\![\text{def } f(\texttt{argdecls})\ \ (I)]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_D\ s = \kappa_D\ \rho\ \rho_F[f \mapsto F]\ s$$

where

$$F = \lambda\text{args}.\ \lambda\kappa_E.\ \lambda\rho''.\ \lambda\rho''_F.\ \lambda\rho''_E.\ \ \lambda s''.\mathcal{A}[\![\texttt{argdecls}]\!]\ \text{args}\ \rho\ \text{ACont}\ \rho''\ \rho''_F\ \rho''_E\ s''$$

where

$$\text{ACont} = \lambda\rho'.\ \lambda s'.\ \mathcal{I}[\![I]\!]\ \rho'\ \rho_F[f \mapsto F]\ \rho''_E\ \kappa_{ERR}\ \kappa_E\ \kappa_{ERR}\ \kappa_{ERR}\ s'$$

where

$$\kappa_{ERR} = \lambda s.\ \text{"}error\text{"}$$

What is also important to note here is that the function will be executed in the exception environment passed at calling. The only thing that remains is to rewrite this into a fixed-point definition. We have, in full form:

$$F = \lambda \text{args}.\ \lambda\kappa_E.\ \lambda\rho''.\ \lambda\rho''_F.\ \lambda\rho''_E.\ \ \lambda s''.$$

$$\mathcal{A}[\![\texttt{argdecls}]\!]\ \text{args}\ \rho\ (\lambda\rho'.\ \lambda s'.\ \mathcal{I}[\![I]\!]\ \rho'\ \rho_F[f \mapsto F]\ \rho''_E\ \kappa_{ERR}\ \kappa_E\ \kappa_{ERR}\ \kappa_{ERR}\ s')\ \rho''\ \rho''_F\ \rho''_E\ s''$$

And therefore, $F = fix(\Phi)$ for

$$\Phi(X) = \lambda \text{args}.\ \lambda\kappa_E.\ \lambda\rho''.\ \lambda\rho''_F.\ \lambda\rho''_E.\ \ \lambda s''.$$

$$\mathcal{A}[\![\texttt{argdecls}]\!]\ \text{args}\ \rho\ (\lambda\rho'.\ \lambda s'.\ \mathcal{I}[\![I]\!]\ \rho'\ \rho_F[f \mapsto X]\ \rho''_E\ \kappa_{ERR}\ \kappa_E\ \kappa_{ERR}\ \kappa_{ERR}\ s')\ \rho''\ \rho''_F\ \rho''_E\ s''$$

Argument declarations

$$\mathcal{A}[\![v]\!]\ e\ \rho\ \kappa_A\ \rho''\ \rho''_F\ \rho''_E\ s'' = \mathcal{E}[\![e]\!]\ \rho''\ \rho''_F\ \rho''_E\ (\lambda n.\ \lambda\ s'''.\ \kappa_A\ \rho[v \mapsto l]\ s'''[l \mapsto n])s''$$

where

$$l = \text{newloc}\ s'''$$

And analogously for declarations of string and bool arguments. Here, the first environment passed represents the environment from the moment of declaration, and the entities with double apostrophes represent the ones from the moment of calling. Those are necessary for calculating argument values (and, once again, the exception environment from calling will be valid while executing the function body).

$$\mathcal{A}[\![v]\!]\ (\texttt{ref}\ x)\ \rho\ \kappa_A\ \rho''\ \rho''_F\ \rho''_E\ s'' = \kappa_A\ \rho[v \mapsto \rho''\ x]\ s''$$

.

$$\mathcal{A}[\![v, \text{argdecls}]\!]\ (\texttt{arg}, \texttt{args})\ \rho\ \kappa_A\ \rho''\ \rho''_F\ \rho''_E\ s'' =$$
$$= \mathcal{A}[\![v]\!]\ \texttt{arg}\ \rho\ (\lambda\ \rho'.\ \lambda\ s'.\ \mathcal{A}[\![\text{argdecls}]\!]\texttt{args}\ \rho'\ \kappa_A\ \rho''\ \rho''_F\ \rho''_E\ s')\ \rho''_F\ \rho''_E\ s''$$

Instructions

$$\mathcal{I}[\![\{d; i\}]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_d\ \kappa_E\ \kappa_b\ \kappa_c\ s = \mathcal{D}[\![d]\!]\ \rho\ \rho_F\ \rho_E\ (\lambda\ \rho'.\lambda\ \rho'_F.\lambda\ s'.\ \mathcal{I}[\![i]\!]\rho'\ \rho'_F\ \rho_E\ \kappa_d\ \kappa_E\ \kappa_b\ \kappa_c\ s')\ s$$

$$\mathcal{I}[\![\{i_1; i_2\}]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_d\ \kappa_E\ \kappa_b\ \kappa_c\ s = \mathcal{I}[\![i_1]\!]\ \rho\ \rho_F\ \rho_E\ (\lambda\ s'.\mathcal{I}[\![i_2]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_d\ \kappa_E\ \kappa_b\ \kappa_c\ s')\ \kappa_E\ \kappa_b\ \kappa_c\ s$$

$$\mathcal{I}[\![x = e]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_d\ \kappa_E\ \kappa_b\ \kappa_c\ s = \mathcal{E}[\![e]\!]\ \rho\ \rho_F\ \rho_E\ (\lambda\ n.\lambda\ s'.\ \kappa_d\ s'[\rho\ x \mapsto n])s$$

$$\mathcal{I}[\![\texttt{if}\ b\ \texttt{then}\ i_1\ \texttt{else}\ i_2]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_d\ \kappa_E\ \kappa_b\ \kappa_c\ s = \mathcal{B}[\![b]\!]\ \rho\ \rho_F\ \rho_E\ (\lambda\ d.\lambda\ s'.\ ifte\ d,\ I_1,\ I_2)s$$

where

$$I_n = \mathcal{I}[\![i_n]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_d\ \kappa_E\ \kappa_b\ \kappa_c\ s'$$

$\mathcal{I}[\![\texttt{if}\ b\ \texttt{then}\ i_1]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_d\ \kappa_E\ \kappa_b\ \kappa_c\ s = \mathcal{B}[\![b]\!]\ \rho\ \rho_F\ \rho_E\ (\lambda\ d.\lambda\ s'.\ ifte\ d,\ I_1,\ \kappa_d s')s$
where I_1 as above.

$\mathcal{I}[\![\texttt{while}\ b\ \texttt{do}\ i_1]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_d\ \kappa_E\ \kappa_b\ \kappa_c\ s = \mathcal{B}[\![b]\!]\ \rho\ \rho_F\ \rho_E\ (\lambda\ d.\lambda\ s'.\ ifte\ d,\ I_1,\ \kappa_d s')s$
where
$I_1 = \mathcal{I}[\![i_1; \texttt{while}\ b\ \texttt{do}\ i_1]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_d\ \kappa_E\ \kappa_d\ (\mathcal{I}[\![\texttt{while}\ b\ \texttt{do}\ i_1]\!])\ \rho\ \rho_F\ \rho_E\ \kappa_d\ \kappa_E\ \kappa_b\ \kappa_c)\ s'$
$\mathcal{I}[\![\texttt{try}\ i_1\ \texttt{except}\ Xs\ i_2]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_d\ \kappa_E\ \kappa_b\ \kappa_c\ s = \mathcal{I}[\![i_1\ ]\!]\ \rho\ \rho_F\ \rho_E[Xs \mapsto I_2]\ \kappa_d\ \kappa_E\ \kappa_b\ \kappa_c\ s$
where

$$I_2 = \mathcal{I}[\![i_2]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_d\ \kappa_E\ \kappa_b\ \kappa_c$$

$$\mathcal{I}[\![\texttt{raise}\ x]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_d\ \kappa_E\ \kappa_b\ \kappa_c\ s = \rho_E\ xs$$
$$\mathcal{I}[\![\texttt{return}\ e]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_d\ \kappa_E\ \kappa_b\ \kappa_c\ s = \mathcal{E}[\![e]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_E\ s$$

$$\mathcal{I}[\![\texttt{break}]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_d\ \kappa_E\ \kappa_b\ \kappa_c\ s = \kappa_b\ s$$

$$\mathcal{I}[\![\texttt{continue}]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_d\ \kappa_E\ \kappa_b\ \kappa_c\ s = \kappa_c\ s$$

$$\mathcal{I}[\![e]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_d\ \kappa_E\ \kappa_b\ \kappa_c\ s = \mathcal{E}[\![e]\!]\ \rho\ \rho_F\ \rho_E\ (\lambda\ n.\ \lambda\ s'.\ \kappa_d\ s')\ s$$

Expressions

I skipped equations fully analogous to the ones presented below.

$$\mathcal{E}[\![f(\mathrm{arg})]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_E\ s = (\rho_F\ f)\ \mathrm{arg}\ \kappa_E\ \rho\ \rho_F\ \rho_E\ s$$

$$\mathcal{B}[\![b_1\ \texttt{or}\ b_2]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_B\ s = \mathcal{B}[\![b_1]\!]\ \rho\ \rho_F\ \rho_E\ (\lambda\ d_1.\ \lambda\ s'.\ \mathcal{B}[\![b_1]\!]\ \rho\ \rho_F\ \rho_E\ (\lambda\ d_2.\ \lambda\ s''.\ \kappa_B\ (d_1 \vee d_2)\ s'')\ s')\ s$$

$$\mathcal{B}[\![e_1 == e_2]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_B\ s = \mathcal{E}[\![e_1]\!]\ \rho\ \rho_F\ \rho_E\ (\lambda\ n.\ \lambda\ s'.\ \mathcal{E}[\![e_1]\!]\ \rho\ \rho_F\ \rho_E\ (\lambda\ m.\ \lambda\ s''.\ \kappa_B\ (n = m)\ s'')\ s')\ s$$

$$\mathcal{E}[\![e_1 + e_2]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_E\ s = \mathcal{E}[\![e_1]\!]\ \rho\ \rho_F\ \rho_E\ (\lambda\ n.\ \lambda\ s'.\ \mathcal{E}[\![e_1]\!]\ \rho\ \rho_F\ \rho_E\ (\lambda\ m.\ \lambda\ s''.\ \kappa_E\ (n+m)\ s'')\ s')\ s$$

$$\mathcal{B}[\![\texttt{not}\ b]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_B\ s = \mathcal{B}[\![b]\!]\ \rho\ \rho_F\ \rho_E\ (\lambda\ d.\ \lambda\ s'.\ \kappa_B(\neg d)\ s')\ s$$
$$\mathcal{E}[\![x]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_E\ s = \kappa_E\ (s(\rho x))\ s$$
$$\mathcal{E}[\![n]\!]\ \rho\ \rho_F\ \rho_E\ \kappa_E\ s = \kappa_E\ n\ s$$

Programs

For a program with function declarations d and main block i:
$\mathcal{E}[\![d, i]\!] = \mathcal{D}[\![d]\!]\ \rho_0\ \rho_{F0}\ \rho_{X0}(\lambda\ \rho.\ \lambda\ \rho_F.\ \lambda\ s.\ \mathcal{I}[\![i]\!]\ \rho_0\ \rho_F\ \kappa_0\ \kappa_{ERR}\ \kappa_{ERR}\ \kappa_{ERR}\ s_0)s_0$
where ρ_0, ρ_{F0}, s_0 - empty functions; ρ_{X0} - an exception environment mapping every exception identifier to κ_{ERR}, and $\kappa_{ERR} = \lambda\ s.$"*error*"