XPLN language specification

METU CENG 444, Fall 2018

Cem Bozşahin, November 1, 2018

Recall the simple language I called XP in class, which I've been using for introducing many concepts:

S →	ID := E
E →	E - T | E + T | T
T →	T * F | T / F | F
F →	ID | NUM | (E)
ID →	L ID′
ID′ →	(L| D) ID′ | ε
NUM→	D NUM | D

It is clear by now that rules ID, ID′ and NUM only concern the lexical analyzer, whereas other rules are syntactic.

XP may be good enough for a calculator, but it doesn't seem like a programming language for a computer.

We extend XP to XPLN to get a programming language worthy of the name, by allowing sequencing, conditionals and loops. Conditionals require conditions; i.e. things that can be true or false. We add this feature with the simplest kind of conditionals in arithmetic: the relational operators, which in our simplistic case compare two arithmetic expressions. The conditional statement starts and ends with a keyword, and it may have an else part. In fact all statements that start with a keyword end with a keyword in XPLN.

One conditional statement and one loop statement are enough to achieve Turing-completeness. Additional stuff would be syntactic sugar.

One more thing and we have a genuine language: function calls. Let us assume XPLN is lexically scoped, and that there are no nested definitions.

For simplicity, we assume XPLN is not typed. Since all we do is arithmetic, we don't have to have types if we assume all variables can take on an integer or floating point value. All variables used are considered local. There is no variable definition. Function call is call-by-value. I/O is limited to reading in and printing out one number from standard input/output.

In summary, an XPLN program is a sequence of semi-colon terminated assignment statements, definitions, conditionals and loops. Execution starts with the first statement, and it returns the value of the first executed `return` statement in the main body. Function definitions are not statements. Function calls are expressions. Order of function definitions is not important.

A covering grammar for XPLN can then be written as follows (`S`, `E` and `ID` are same as before, with one more rule for `F` and `NUM`; also note slight change in `S` rule, where termination is left to the variable `Start`):

```
Start  →  Stmt ; (Entry ;)+
Entry  →  Stmt | Def
Stmts  →  (Stmt ;)+
Stmt   →  S | If | Wh | Ret | IO
If     →  if Cond Stmts [else Stmts] endi
Wh     →  while Cond Stmts endw
Ret    →  return E
IO     →  (input | output) ID
Def    →  fun ID Plist  Stmts endf
Plist  →  '(' comma-separated arguments ')'
F      →  ID Plist (function call)
NUM    →  a floating point number (no exponents)
Cond   →  E Bop E | Cond Lop Cond | Uop Cond
Bop    →  one of the relational operators <, <=, ==, >, >=
Lop    →  and | or
Uop    →  ! (not)
```

Conventions:

1. A token in lowercase is a keyword in the grammar above.
2. XPLN is not case sensitive.
3. Keywords cannot be defined as variables (because they are invariables).
4. It is up to an implementation to issue a warning for multiply defined functions. The last definition counts.
5. Some rules include informal descriptions of a functionality.
6. Functions can have multiple `return` statements. At least one is required for the main body and every function.
7. I/O is simple port-IO. The port is tied to standard input for `input` and standard output for `output`. Port I/O is supported by the Virtual Machine we translate into.