Kitty Grammar

Syntactically well-formed Kitty programs are those derivable from the grammar in Figures 1. Bold names stand for token types. Italicized annotations are comments and not part of the grammar. The grammar in Figure 1 is ambiguous. The ambiguities are removed by the following rules:

Precedence: The precedence of operators from highest to lowest is as follows (operators on the same line have the same precedence):

unary minus (negation)
`*, /`
`+, -`
`<, <=, =, <>, >=, >`
`&`
`|`

Associativity: The operators `*`, `\`, `+`, `-`, `&`, and `|` are all *left-associative*. E.g., 1 – 2 + 3 is parsed as if it were written (1 – 2) + 3. The relational `<`, `<=`, `=`, `<>`, `>=`, and `>` are all *non-associative*. E.g., `1 < 2 = 3` is not a legal expression, even though the explicitly grouped versions `(1 < 2) = 3` and `1 < (2 = 3)` are legal expressions.

Dangling Else: The presence of both if-then and if-then-else expressions in a language introduces an ambiguity as to which `if` expression an `else` clause belongs. The Kitty convention (as in many other languages) is that an else clause belongs to the innermost if expression enclosing it. Thus, the expression

if E_1 then if E_2 then E_3 else E_4

is parsed as if it were written

if E_1 then (if E_2 then E_3 else E_4)

Exp derives Kitty expressions

Exp → `()` *the literal for "no value"*
Exp → **intlit** *as specified by the lexical conventions for integer literals*
Exp → **charlit** *as specified by the lexical conventions for character literals*
Exp → **ident** *as specified by the lexical conventions for identifiers*
Exp → Const
Exp → Nullop `()` *the parentheses are required*
Exp → Unop `(Exp)` *the parentheses are required*
Exp → -Exp *unary minus operator*
Exp → writes(**stringlit**)
Exp → Exp Binop Exp
Exp → **ident** := Exp *assignment*
Exp → if Exp then Exp else Exp
Exp → if Exp then Exp
Exp → let Decs in ExpSeq0 end
Exp → while Exp do Exp
Exp → for **ident** := Exp to Exp do Exp
Exp → (ExpSeq2) *sequence expression, parentheses required*
Exp → (Exp) *grouping via optional parentheses*

ExpSeq0 derives expression sequences with 0 or more expressions

ExpSeq0 → *empty expression sequence*
ExpSeq0 → ExpSeq1

ExpSeq1 derives expression sequences with 1 or more expressions

ExpSeq1 → Exp
ExpSeq1 → Exp ; ExpSeq1

ExpSeq2 derives expression sequences with 2 or more expressions

ExpSeq2 → Exp ; ExpSeq1

Decs derives declaration sequences with 1 or more declarations

Decs → Dec
Decs → Dec;Decs

```
Dec derives variable
declarations
Dec → var ident := Exp
Const derives constants
Const → minint
Const → maxint
Const → true
Const → false
Nullop derives nullary (zeroargument)
operators
Nullop -> readc
Unop derives unary (oneargument)
operators
Unop -> not
Unop -> readi
Unop -> writec
Unop -> writei

Binop derives binary (twoargument)
operators
Arithmetic Binops
Binop -> +
Binop -> -
Binop -> *
Binop -> / integer division
Binop -> % integer modulus
Relational Binops
Binop -> <
Binop -> <=
Binop -> =
Binop -> <> not equals
Binop -> >=
Binop -> >
Logical Binops
Binop -> & short circuit and
Binop -> | short circuit or
```

Figure 1: Kitty Grammar