SpoookyC Tokens / Lexeme Examples

Lexeme(s) / example lexemes	**Token**
iff	IF
wyl	WHILE
els	ELSE
:^|	END_IF
:^{	END_WHILE
X^O	ASSIGN
:’{	END_LINE
(:	OPEN_P
:)	CLOSE_P
#num, #swag, #345	IDENT
aprt	PRINT
skn	SCAN
,	COMMA
123, 45.6, “here”, “here it is”, tru, fls	LITERAL
.app, .rmv	LIST_MUTATOR
.len, .get	LIST_ACCESSOR
int, flo, boo, str, dub, lst	TYPE_IDENT
-, not	UN_OP
*, /	BIN1_OP
+	BIN2_OP
<=, >=, ==, and, oor, >, <	BIN3_OP

SpoookyC Grammar Rules

<stmts> →<stmts> <stmt>|<stmt>

<stmt> → <declare_stmt> | <assign_stmt> | <if_stmt> | <while_loop> | <print_stmt> | <scan_stmt> | <list_stmt>

<if_stmt> → IF <exp> <stmts> ENDIF | IF <exp> <stmts> ELSE <stmts> END_IF

<while_loop> → WHILE <exp> <stmts> END_WHILE

<declare_stmt> → TYPE_IDENT IDENT END_LINE
| TYPE_IDENT IDENT ASSIGN <exp> END_LINE

<assn_stmt> → IDENT ASSIGN <exp> END_LINE

<list_mut_stmt> → IDENT LIST_MUTATOR OPEN_P <exp> CLOSE_P END_LINE

<print_stmt> → PRINT OPEN_P <toPrint> CLOSED_P END_LINE

<to_print> → <exp> | <to_print> COMMA <to_print>

<scan_stmt> → SCAN OPEN_P IDENT CLOSED_P END_LIN E

Syntax Rule:

<exp>.1 → <exp2_piece> | <exp>.2 BIN3_OP <exp2_piece>

Predicate:

```
iff <exp>.1 → <exp>.2 BIN3_OP <exp2_piece>
    iff BIN3_OP == ( > | < | >= | <=)
        <exp>.2.type == (string) && <exp2_piece>.type ==  <exp>.2.type
        <exp>.2.type == (|int|float|dub) && <exp2_piece>.type == (|int|float|dub)
    els iff BIN3_OP == ( and | oor )
        <exp>.2.type == boo && <exp2_piece>.type ==  boo
    els iff BIN3_OP == ( == )
        <exp>.2.type == <exp2_piece>.type == boo
```

Production :

```
iff ( <exp>.1 == <exp>.2 BIN3_OP <exp2_piece>)
    <exp>.1.type = boo
els
    <exp>.1.type = <exp2_piece>.type
```

Syntax rule:**<exp2_piece.1>** → <exp1_piece> | <exp2_piece.2> BIN2_OP <exp1_piece>

Predicate rule:

```
iff <exp2_piece.1> → <exp2_piece.2> BIN1_OP <exp1_piece>
    <exp2_piece.2>.type == (int|dub|flo|str) && <exp1_piece>.type == (int|dub|flo|str)
```

Production rule:

```
iff<exp2_piece> → <exp1_piece>
    <exp2_piece>.type = <exp1_piece>.type
end iff
```

```
els iff <exp2_piece> → <exp2_piece> BIN2_OP <exp1_piece>
    iff <exp2_piece>.type == <exp1_piece>.type
        result.type ==<exp1_piece>.type
    else iff <exp1_piece>.type == int && <exp2_piece>.type==flo||<exp1_piece>.type ==flo
    && <exp2_piece>.type==int
        result.type==flo
    else iff <exp1_piece>.type == int && <exp2_piece>.type==dub||<exp1_piece>.type
    ==dub && <exp2_piece>.type==int
        result.type == dub
    else iff <exp1_piece>.type == str && <exp2_piece>.type==dub|int|flo
    ||<exp1_piece>.type ==dub|int|flu && <exp2_piece>.type==str
        result.type==str
    else ERROR
    end iff
end iff
```

Syntax Rule

<exp1_piece.1> → <exp_piece> | <exp1_piece.2> BIN1_OP <exp_piece>

Predicate:

```
iff <exp1_piece.1> → <exp1_piece.2> BIN1_OP <exp_piece>
    <exp1_piece.2>.type == (int|dub|flo) && <exp_piece>.type == (int|dub|flo)
```

Production Rule:

```
iff <exp1_piece.1> → <exp_piece>
    <exp1_piece.1>.type = <exp_piece>.type
els iff <exp1_piece.1> → <exp1_piece.2> BIN1_OP <exp_piece>
    iff <exp1_piece.2>.type == dub ||  <exp_piece>.type == dub
        <exp1_piece.1>.type = dub
    els iff <exp1_piece>.type == flo ||  <exp_piece>.type == flo
        <exp1_piece.1>.type = flo
    els iff
        <exp1_piece>.type = str && <exp2_piece
```

Syntax rule

<exp_piece.1> → LITERAL | IDENT | UN_OP <exp_piece.2>

Predicate:

```
iff <exp_piece.1> → UN_OP <exp_piece.2>
    iff UN_OP== -
        <exp_piece.2>.type  == (int | flo | dub)
    iff UN_OP== not
        <exp_piece.2>.type == boo
```

Production Rule:

```
iff <exp_piece.1> → LITERAL
    <exp_piece.1>.type  = LITERAL.type
iff <exp_piece.1> → IDENT
    <exp_piece.1>.type  = IDENT.type
els iff <exp_piece.1> → UN_OP <exp_piece.2>
    iff UN_OP== not
        <exp_piece.1>.type  = boo
    els iff UN_OP== -
        <exp_piece.1>.type  = <exp_piece.2>.type
```

LITERALs and IDENTs have intrinsic type attributes, which come from outside the parse tree

SpoookyC Example Code

```
int #donald_trump:'{
    skn(:#donaldTrump:):'{
flo #years X^O 0.01 :'{
lst #yearList :'{
iff #donald_trump == 2016 and #years<=4.01
    wyl #years < 4.01
    prt (:"Year ", #years+#donaldTrump, "Still the Donald":) :'{
    #yearList.app(:#years+#donaldTrump:) :'{
    #years X^O #years+1
    :^{
    prt(:"All the years:",#yearList:)
els
    #years X^O #years +- 15
    boo #2much X^O not fls oor tru
prt(:"Not even counting…":)
:^|
lst #yearList :'{
iff #donald_trump == 2016 and #years<=4.01
    wyl #years < 4.01
    prt (:"Year ", #years+#donaldTrump, "Still the Donald":) :'{
    #yearList.app(:#years+#donaldTrump:) :'{
    #years = #years+1
    :^{
    prt(:"All the years:",#yearList:)
els
    #years = #years +- 15
    boo #2much = not fls oor tru
prt(:"Not even counting…":)
:^|

"next line produces 2 correct tokens and 5 error tokens"
int var = 6;if else



int #donald_trump X^O 2016:’{
#donald_trump X^O 2020:’{
int #devil:’{
#devil X^O 666:’{
str #tom_brady X^O “innocent”:’{
prt(#x, “big”):’{
skn(:#x:):’{


iff #donald_trump = 2016
    wyl #year < 2020
        prt (:”gg wp”:) :’}
    :^{
:^|

iff #donald_trump = 2016 wyl #year < 2020 prt “gg wp” :’}:^{:^|


lst #list :’{
int #num X^O #list.get(:4:) :’{ //Get specific item in list
#list.app(:9:) :’{ //Append item to list
#list.rmv(:9:) :’{ //Remove item from list
#list.len(::) :’{ //Get length of list

prt(:big,“x”, 3:):’{ //print out the value of big AND “x3” to console
skn(:x:):’{  //scan things from console and put into variable x
```