Lexical analysis and parsing

A. **Reserved words**

Reserved words are not reserved when used as fields. So `return=1` is illegal, but `foo.return=1` is fine.

B. **Round, square, and curly brackets**

In Python and in C it's simple -- function are called using parentheses, and array subscripted with square brackets. In Matlab and in Fortran both are done with parentheses, so there is no visual difference between function call `foo(x)` and indexing array `foo`.

C. **Handling the white space.**

There are four kinds of ws, each recognized by a dedicated rule: NEWLINE, COMMENT, ELLIPSIS, and SPACES. Only NEWLINE returns a token, the three others silently discard their input. NEWLINE collects adjacent `\n` characters and returns a single SEMI token:

```
def t_NEWLINE(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
    if not t.lexer.parens and not t.lexer.braces:
        t.value = ";"
        t.type = "SEMI"
        return t
```

Comments come in two flavors -- regular and MULTILINE.

The regular comments are discarded, while MULTILINE become doc strings, and are handled as expr statements. regular consume everything from % or # up to but not including n:

```
(%|\#).*
```

Multiline COMMENT rule drops leading blanks, then eats everything from `%` or `#` up to the end of line, *including* newline character. COMMENT leaves one newline character, or else funny things happen:

```
@TOKEN(r"(%|\#).*")
def t_COMMENT(t):
    if not options.do_magic or t.value[-1] != "!":
        t.lexer.lexpos = t.lexer.lexdata.find("\n",t.lexer.lexpos)
```

Multiline comments:

```
(^[ \t](%|\#).*\n)+
```

The pattern for multi-line comments works only if re.MULTILINE flag is passed to ctor.

Comments starting with `%!` have a special meaning TBD

ELLIPSIS is the matlab way for line continuation. It discards everything between `...` and the newline character, including the trailing `\n`:

```
def t_ELLIPSIS(t):
    r"\.\.\..*\n"
    t.lexer.lineno += 1
    pass
```

SPACES discards one or more horisontal space characters. Not clear what escape sequence is supposed to do:

```
def t_SPACES(t):
    r"(\\\n|[ \t\r])+"
    pass
```

D. **p_TRANSPOSE**

a. A quote, immediately following a reserved word is always a `STRING`. Implemented using inclusive state *afterkeyword*:

```
t.type = reserved.get(t.value,"IDENT")
if t.type != "IDENT" and t.lexer.lexdata[t.lexer.lexpos]=="'":
    t.lexer.begin("afterkeyword")
```

b. A quote, immediately following any of: (1) an alphanumeric charater, (2) right bracket, parenthesis or brace, or (3) another `TRANSPOSE`, is a `TRANSPOSE`. Otherwise, it starts a string. If the quote is separated from the term by line continuation (...), matlab starts a string, so these rules still holds.:

```
def t_TRANSPOSE(t):
    r"(?<=\w|\]|\)|\})((\.')|')+"
    # <---context ---><-quotes->
    # We let the parser figure out what that mix of quotes and
    # dot-quotes, which is kept in t.value, really means.
    return t
```

E. **Keyword end -- expression and statement**

Any of: `endwhile`, etc. are `END_STMT`. Otherwise, if `end` appears inside parentheses of any kind, it's `END_EXPR`. Otherwise, `end` is illegal.

F. **Optional end statement as function terminator**

Inconsistency between Matlab and Octave, easily solved if the lexer effectively handles the whitespace:

```
function : FUNCTION
         | END_STMT SEMI FUNCTION
```

This usage is consistent with the other cases -- (1) statements start with a keyword and are terminated by the SEMI token, and (2) the lexer combines several comments, blanks, and other junk as one SEMI token. Compare parse.py rule for RETURN statement.

G. **Semicolon as statement terminator, as column separator in matrices.**

Comma, semicolon, and newline are statement terminators. In matrix expressiions, whitespace is significant and separates elements just as comma does.

H. **Matrix state**

In matrix state, consume whitespace separating two terms and return a fake `COMMA` token. This allows parsing `[1 2 3]` as if it was `[1,2,3]`. Handle with care: `[x + y]` vs `[x +y]`

Term T is:

```
(1) a name or a number
(2) literal string using single or doble quote
(3) (T) or [T] or {T} or T' or +T or -T
```

Terms end with:

```
(1) an alphanumeric charater \w
(2) single quote (in octave also double-quote)
(3) right parenthesis, bracket, or brace
(4) a dot (after a number, such as 3.
```

The pattern for whitespace accounts for ellipsis as a whitespace, and for the trailing junk.

Terms start with:

```
(1) an alphanumeric character
(2) a single or double quote,
(3) left paren, bracket, or brace and finally
(4) a dot before a digit, such as .3  .

TODO: what about curly brackets ???
TODO: what about dot followed by a letter, as in field
[foo  .bar]

t.lexer.lineno += t.value.count("\n")
t.type = "COMMA"
return t
```

Class matlabarray

Matlab arrays differ from numpy arrays in many ways, and class matlabarray captures the following differences:

A. **Base-one indexing**

Following Fortran tradition, matlab starts array indexing with one, not with zero. Correspondingly, the last element of a N-element array is N, not N-1.

B. **C_CONTIGUOUS and F_CONTIGUOUS data layout**

Matlab matrix elements are ordered in columns-first, aka F_CONTIGUOUS order. By default, numpy arrays are C_CONTIGUOUS. Instances of matlabarray are F_CONTIGUOUS, except if created empty, in which case they are C_CONTIGUOUS.

matlab	numpy

`> reshape(1:4,[2 2])`
`1 3`
`2 4`
	`>>> a=matlabarray([1,2,3,4])`
	`>>> a.reshape(2,2,order="F")`
	`1 3`
	`2 4`
	
	`>>> a.reshape(2,2,order="C")`
	`1 2`
	`3 4`

```
>>> a=matlabarray([1,2,3,4])
>>> a.flags.f_contiguous
True
>>> a.flags.c_contiguous
False
```

```
>>> a=matlabarray()
>>> a.flags.c_contiguous
True
>>> a.flags.f_contiguous
False
```

C. **Auto-expanding arrays**

Arrays are auto-expanded on out-of-bound assignment. Deprecated, this feature is widely used in legacy code. In smop, out-of-bound assignment is fully supported for row and column vectors, and for their generalizations having shape

[1 1 ... N ... 1 1 1]

These arrays may be resized along their only non-singular dimension. For other arrays, new columns can be added to F_CONTIGUOUS arrays, and new rows can be added to C_CONTIGUOUS arrays.

matlab	numpy

`> a=[]`
`> a(1)=123`
`> a`
`123`
	`>>> a=matlabarray()`
	`>>> a[1]=123`
	`>>> a`
	`123`

D. **Create by update**

In matlab, arrays can be created by updating a non-existent array, as in the following example:

```
>>> clear a
>>> a(17) = 42
```

This unique feature is not yet supported by smop, but can be worked around by inserting assignments into the original matlab code:

```
>>> a = []
>>> a(17) = 42
```

E. **Assignment as copy**

Array data is not shared by copying or slice indexing. Instead there is copy-on-write.

F. **Everything is a matrix**

There are no zero or one-dimensional arrays. Scalars are two-dimensional rather than zero-dimensional as in numpy.

G. **Single subscript implies ravel.**

TBD

H. **Broadcasting**

Boadcasting rules are different

I. **Boolean indexing**

TBD

J. **Character string constants and escape sequences [ffd52d5fc5]**

In Matlab, character strings are enclosed in single quotes, like `'this'`, and escape sequences are not recognized:

```
matlab> size('hello\n')
1   7
```

There are seven (!) characters in `'hello\n'`, the last two being the backslash and the letter n.

Two consecutive quotes are used to put a quote into a string:

```
matlab> 'hello''world'
hello'world
```

In Octave, there are two kinds of strings: octave-style (enclosed in double quotes), and matlab-style (enclosed in single quotes). Octave-style strings do understand escape sequences:

```
matlab> size("hello\n")
1   6
```

There are six characters in `"hello\n"`, the last one being the newline character.

Octave recognizes the same escape sequnces as C:

```
\"  \a  \b  \f  \r  \t  \0  \v  \n  \\ \nnn \xhh
```

where n is an octal digit and h is a hexadecimal digit.

Finally, two consecutive double-quote characters become a single one, like here:

```
octave> "hello""world"
hello"world
```

Data structures

A. **Empty vector [], empty string "", and empty cellarray {}**

matlab	numpy

`> size([])`
`0 0`
 	
`> size('')`
`0 0`
 	
`> size({})`
`0 0`
	`>>> matlabarray().shape`
	`(0, 0)`
	
	`>>> char().shape`
	`(0, 0)`
	
	`>>> cellarray().shape`
	`(0, 0)`

B. **Scalars are 1x1 matrices**

matlab	numpy

`> a=17`
`> size(a)`
`1 1`
	`>>> a=matlabarray(17)`
	`>>> a.shape`
	`1 1`

C. **Rectangular char arrays**

Class char inherits from class matlabarray the usual matlab array behaviour -- base-1 indexing, Fortran data order, auto-expand on out-of-bound assignment, etc.

matlab	numpy

`> s='helloworld'`
`> size(s)`
`1 10`
`> s(1:5)='HELLO'`
`> s`
`HELLOworld`
`> resize(s,[2 5])`
`HELLO`
`world`
	`>>> s=char('helloworld')`
	`>>> print size_(s)`
	`(1,10)`
	`>>> s[1:5]='HELLO'`
	`>>> print s`
	`HELLOworld`
	`>>> print resize_(s,[2,5])`
	`HELLO`
	`world`

D. **Row vector**

matlab	numpy
`> s=[1 2 3]`	`>>> s=matlabarray([1,2,3])`

E. **Column vector**

matlab	numpy

`> a=[1;2;3]`
 	
`> size(a)`
`3 1`
	`>>> a=matlabarray([[1],`
	`[2],`
	`[2]])`
	`>>> a.shape`
	`(3, 1)`

F. **Cell arrays**

Cell arrays subclass matlabarray and inherit the usual matlab array behaviour -- base-1 indexing, Fortran data order, expand on out-of-bound assignment, etc. Unlike matlabarray, each element of cellarray holds a python object.

matlab	numpy

`> a = { 'abc', 123 }`
`> a{1}`
`abc`
	`>>> a=cellarray(['abc',123])`
	`>>> a[1]`
	`abc`

G. **Cell arrays of strings**

In matlab, cellstrings are cell arrays, where each cell contains a char object. In numpy, class cellstring derives from matlabarray, and each cell contains a native python string (not a char instance).

matlab	numpy

`> a = { 'abc', 'hello' }`
 	
`> a{1}`
`abc`
	`>>> a=cellstring(['abc',`
	`'hello'])`
	`>>> a[1]`
	`abc`

Data structures

All matlab data structures subclass from matlabarray

Structs

TBD

Function pointers

Handles @

String concatenation

Array concatenation not implemented

```
>>> ['hello' 'world']
helloworld
```