P0 Scanner

Emil Sekerinski, McMaster University, revised February 2022

The scanner reads the characters of the source consecutively and recognizes symbols they form:

- procedure `init(src)` initializes the scanner
- procedure `getSym()` recognizes the next symbol and assigns it to variables `sym` and `val`.
- procedure `mark(msg)` prints an error message at the current location in the source.

Symbols are encoded by integer constants.

In [ ]:
```
IDENT = 1; NUMBER = 2; TIMES = 3; DIV = 4; MOD = 5; PLUS = 6; MINUS = 7; AND = 8; OR = 9
EQ = 10; NE = 11; LT = 12; GT = 13; LE = 14; GE = 15; SEMICOLON = 16; COMMA = 17
COLON = 18; BECOMES = 19; PERIOD = 20; DOTDOT = 21; NOT = 22; LPAREN = 23; RPAREN = 24
LBRAK = 25; RBRAK = 26; LBRACE = 27; RBRACE = 28; LARROW = 29; RARROW = 30; CARD = 31
COMPLEMENT = 32; UNION = 33; INTERSECTION = 34; ELEMENT = 35; SUBSET = 36; SUPERSET = 37
IF = 38; THEN = 39; ELSE = 40; WHILE = 41; DO = 42; CONST = 43; TYPE = 44; VAR = 45
SET = 46; PROCEDURE = 47; PROGRAM = 48; INDENT = 49; DEDENT = 50; EOF = 51
```

Following variables determine the state of the scanner:

- `(line, pos)` is the location of the current symbol in source
- `(lastline, lastpos)` is used to more accurately report errors
- `ch` is the current character
- `sym` the current symbol, `TIMES` ... `EOF` or `None`
- if `sym` is `NUMBER`, `val` is the value of the number
- if `sym` is `IDENT`, `val` is the identifier string
- `source` is the string with the source program
- `index` is the index of the next character in `source`
- `indents` is a stack with indentations
- `newline` is a boolean indicating the start of a line

The source is specified as a parameter to the procedure `init`:

In [ ]:
```
def init(src):
    global line, lastline, pos, lastpos
    global ch, sym, val, source, index, indents
    line, lastline = 0, 1
    pos, lastpos = 1, 1
    ch, sym, val, source, index = '\n', None, None, src, 0
    indents = [1]; getChar(); getSym()
```

Procedure `getChar()` assigns the next character in `ch`, or assigns `chr(0)` at the end of the source. Variables `line`, `pos` are updated with the current location in the source and `lastline`, `lastpos` are updated with the location of the previously read character.

In [ ]:
```
def getChar():
    global line, lastline, pos, lastpos, ch, index
    if index == len(source): ch, index, pos = chr(0), index + 1, 1
    else:
        lastpos = pos
        if ch == '\n':
            pos, line = 1, line + 1
        else:
            lastline, pos = line, pos + 1
        ch, index = source[index], index + 1
```

Procedure `mark(msg)` prints an error message with the current location in the source. To avoid a cascade of errors, only one error message at a source location is printed and compilation stops.

In [ ]:
```
def mark(msg):
    raise Exception('line ' + str(lastline) + ' pos ' + str(lastpos) + ' ' + msg)
```

Procedure `number()` parses

```
number ::= digit {digit}
digit ::= '0' | ... | '9'
```

If the number fits in 32 bits, sets `sym` to `NUMBER` and assigns to number to `val`, otherwise reports an error.

In [ ]:
```
def number():
```

```
def number():
    global sym, val
    sym, val = NUMBER, 0
    while '0' <= ch <= '9':
        val = 10 * val + int(ch)
        getChar()
    if val >= 2**31: mark('number too large')
```

Procedure `identKW()` parses

```
identKW ::= keyword / identifier
identifier ::= letter {letter | digit}
letter ::= 'A' | ... | 'Z' | 'a' | ... | 'z'
keyword ::= 'div' | 'mod' | 'and' | 'or' | 'if' | 'then' | 'else' | 'while' | 'do' | 'const' |
'type' | 'var' |
                    'set' | 'procedure' | 'program'
```

The longest sequence of character that matches `letter {letter | digit}` is read. If that sequence is a keyword, `sym` is set accordingly, otherwise `sym` is set to `IDENT`.

In [ ]:
```
KEYWORDS = \
    {'div': DIV, 'mod': MOD, 'and': AND, 'or': OR, 'if': IF, 'then': THEN,
     'else': ELSE, 'while': WHILE, 'do': DO, 'const': CONST, 'type': TYPE,
     'var': VAR, 'set': SET, 'procedure': PROCEDURE, 'program': PROGRAM}

def identKW():
    global sym, val
    start = index - 1
    while ('A' <= ch <= 'Z') or ('a' <= ch <= 'z') or \
          ('0' <= ch <= '9'): getChar()
    val = source[start : index - 1]
    sym = KEYWORDS[val] if val in KEYWORDS else IDENT
```

Procedure `comment()` parses

```
comment ::= '//' {character – '\n'}
```

A comment is skipped over.

In [ ]:
```
def comment():
    if ch == '/': getChar()
    else: mark('// expected')
    while chr(0) != ch != '\n': getChar()
```

Procedure `getSym()` parses

```
symbol ::= { ' ' | comment} ( { '\n' {' ' | comment} } | identKW | number | '×' | '+' | '-' |
                    '=' | '≠' | '<' | '≤' | '>' | '≥' | ';' | ',' | ':' | ':=' | '.' | '¬' |
'(' |  ')' | '[' | ']' |
                    '←' | '→' | '{' | '}' | '#' | 'C' | '∪' | '∩' | '∈' | '⊆' | '⊇')
```

If a valid symbol is recognized, `sym` is set accordingly, otherwise an error is reported. The longest match is used for recognizing operators. Blanks that are not at the beginning of a line are skipped. A stack, `indents`, is used to keep track if blanks at the beginning of a line are either ignored or recognized as `INDENT` or `DEDENT`. On the first symbol of a line, `newline` is set to `True` if the indentation is the same as that of the previous line; for all subsequent symbols, `newline` is set to `False`. At the end of the source, `sym` is set to `EOF`.

In [ ]:
```
def getSym():
    global sym, indents, newline
    if pos < indents[0]:
        indents = indents[1:]; sym = DEDENT
    else:
        while ch in ' /':
            if ch == ' ': getChar() # skip blanks between symbols
            else: comment()
        if ch == '\n': # possibly INDENT, DEDENT
            while ch == '\n': # skip blank lines
                getChar()
                while ch in ' /':
                    if ch == ' ': getChar() # skip indentation
                    else: comment()
            if pos < indents[0]: sym, indents = DEDENT, indents[1:]; return
            elif pos > indents[0]: sym, indents = INDENT, [pos] + indents; return
        newline = pos == indents[0]
```

```
        if 'A' <= ch <= 'Z' or 'a' <= ch <= 'z': identKW()
        elif '0' <= ch <= '9': number()
        elif ch == '×': getChar(); sym = TIMES
        elif ch == '+': getChar(); sym = PLUS
        elif ch == '-': getChar(); sym = MINUS
        elif ch == '=': getChar(); sym = EQ
        elif ch == '≠': getChar(); sym = NE
        elif ch == '<': getChar(); sym = LT
        elif ch == '≤': getChar(); sym = LE
        elif ch == '>': getChar(); sym = GT
        elif ch == '≥': getChar(); sym = GE
        elif ch == ';': getChar(); sym = SEMICOLON
        elif ch == ',': getChar(); sym = COMMA
        elif ch == ':':
            getChar()
            if ch == '=': getChar(); sym = BECOMES
            else: sym = COLON
        elif ch == '.':
            getChar();
            if ch == '.': getChar(); sym = DOTDOT
            else: sym = PERIOD
        elif ch == '¬': getChar(); sym = NOT
        elif ch == '(': getChar(); sym = LPAREN
        elif ch == ')': getChar(); sym = RPAREN
        elif ch == '[': getChar(); sym = LBRAK
        elif ch == ']': getChar(); sym = RBRAK
        elif ch == '{': getChar(); sym = LBRACE
        elif ch == '}': getChar(); sym = RBRACE
        elif ch == '←': getChar(); sym = LARROW
        elif ch == '→': getChar(); sym = RARROW
        elif ch == '#': getChar(); sym = CARD
        elif ch == 'C': getChar(); sym = COMPLEMENT
        elif ch == '∪': getChar(); sym = UNION
        elif ch == '∩': getChar(); sym = INTERSECTION
        elif ch == '∈': getChar(); sym = ELEMENT
        elif ch == '⊆': getChar(); sym = SUBSET
        elif ch == '⊇': getChar(); sym = SUPERSET
        elif ch == chr(0): sym = EOF
        else: mark('illegal character')
```