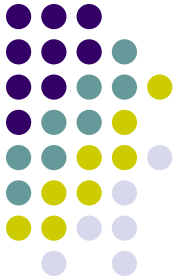
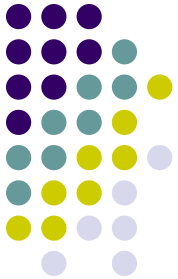
Multi-Symbol Words - Lexical Analysis



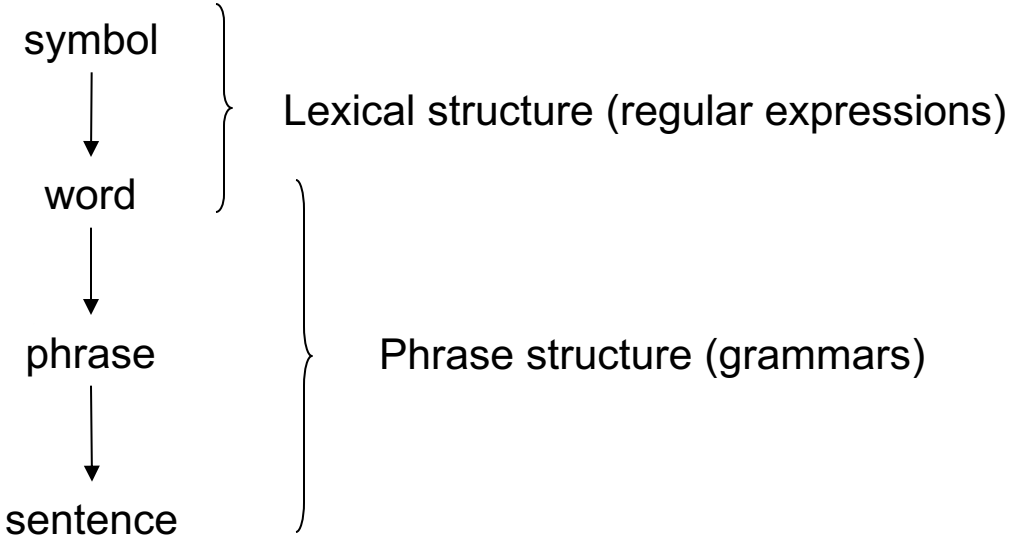
- In our exp0 programming language we only had words of length one
- However, most programming languages have words of lengths more than one
- The lexical structure of a programming language specifies how symbols are combined to form words
  - Not to be confused with the phrase structure which tells us how words are combined to form phrases and sentences
- The lexical structure of a programming language can be specified with regular expressions
  - whereas the phrase structure is specified with grammars.
- The "parser" for the lexical structure of a programming language is called a lexical analyzer or lexer
- The output of a lexer is usually given in terms of tokens.

Read Chap 2

Multi-Symbol Words - Lexical Analysis



- This gives us the following hierarchy:



The Calc Language

EBNF notation stating that exp can appear zero or more times

Listing 2.7: The grammar for the calc language.

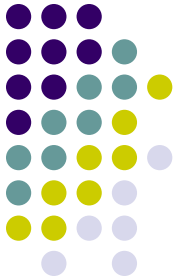
```
1  explist : (exp)*
2
3  exp : NUM
4      | op exp exp
5      | LPAREN op exp exp (exp)* RPAREN
6
7  op : PLUS
8     | MINUS
```

Tokens

Tokens

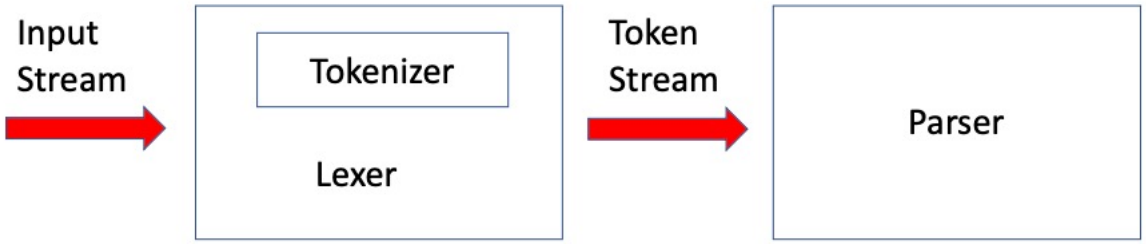
- This language allows us to write expression like,
  - 125
  - + 36 14
  - (+ 1 2 3)
  - Note that actual values and op names are now encoded as tokens in the grammar, e.g. NUM, PLUS

Tokens

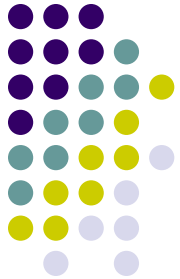


- The definition of Tokens usually has two parts:
  - A token type
  - A token value
- For example, in Calc we have
  - a token type PLUS with a token value of '+'
  - a token type NUM with an integer token value.
- That means lexers turn character/symbols streams into *token streams*
- Token streams is what is read by parsers.

The Syntactic Analysis Phase



The Lexer



- The lexer includes the tokenizer and implements a token stream with the following interface:

pointer – a function that points to the current token in the token stream.

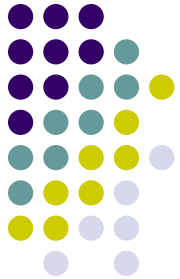
next – a function that advances the pointer to the next token.

match – a function matches a token against the current token in the token stream.

end-of-file – a predicate that returns true if the pointer reached the end of the token stream.

Note: same interface as our earlier InputStream class.

Calc Tokens



`NUM` – a number token describing an actual number in the input stream.

`PLUS` – a token describing the `+` operator.

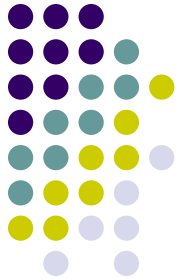
`MINUS` – a token describing the `-` operator.

`LPAREN` – a token describing the left parenthesis.

`RPAREN` – a token describing the right parenthesis.

```
$ python3
Python 3.8.5 (default, Sep  4 2020, 02:22:02)
>>> from calc_lexer import tokenize
>>> for t in tokenize("+ 101 25"):
...     print(t)
...
Token(PLUS,+)
Token(NUM,101)
Token(NUM,25)
Token(EOF,\eof)
>>>
```

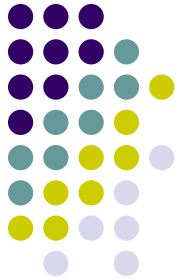
Specifying Tokens



- In our calc_lexer.py file all we need to do is to define the token types and values
- The rest of the code is boiler plate implementing the tokenizer and lexer
- We use regular expression to specify the token values

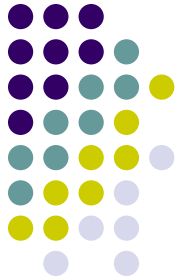
```
token_specs = [
#   type:           value:
    ('NUM',         r'[0-9]+'),
    ('PLUS',        r'\+'),
    ('MINUS',       r'-'),
    ('LPAREN',      r'\('),
    ('RPAREN',      r'\)'),
    ('WHITESPACE', r'[ \t\n]+'),
    ('UNKNOWN',     r'.'),
]
```

Regular Expressions



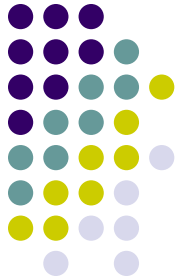
- Each letter A through Z and a through z is a regular expression.
- Each number 0 through 9 is a regular expression.
- Each printable character \(, \), -, \+, etc. is a regular expression.
- If A and B are regular expressions then AB is also a regular expression and represents the concatenation of the two regular expressions.
- If A is a regular expression then (A) is also a regular expression. Parentheses allow us to group regular expressions. Just as in grammars, the use of escaped parentheses in regular expressions is very important because the regular expression (A) is different from the regular expression \(A\). The former is the grouping of regular expression A and the latter is the concatenation of the three regular expressions.
- If A and B are regular expressions then A | B is also a regular expression and represents the choice between regular expression A and regular expression B.
- If A is a regular expression then A? is also a regular expression and specifies the regular expression A as optional.
- If A is a regular expression then A* is also a regular expression and specifies that the regular expression A can appear zero or more times. We use the same operator in the EBNF notation for grammars.
- If A is a regular expression then A+ is also a regular expression and specifies that the regular expression A can appear one or more times. You can think of A+ as a shorthand for AA*.

Regular Expression



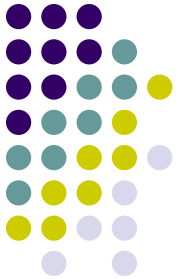
- The regular expression [A-Z] represents a single character between A and Z. Similarly for [a-z] and [0-9].
- The special characters \n, \t, and \r are also regular expressions representing the newline character, the TAB character, and the carriage return character, respectively.
- The dot operator . is a regular expression that represents any single printable character. Most importantly, it does not represent the newline character \n.
- The ˆ operator computes the complement of a set. For example, if we have the regular expression [abc] matching either a,b or c, then the complement [ˆabc] will match any character other than a, b, or c. This is useful in conjunction with character classes. For example, the regular expression [A-Z][ˆA-Z] specifies a word structure that starts with a capital letter followed by a single character that is not a capital letter.

Parsing with Lexers



- Good news: the techniques of building top-down parser we have looked at so far apply to parsers that use lexers!
- Instead of using lookahead symbol we will now use lookahead tokens.

Parsing with Lexers

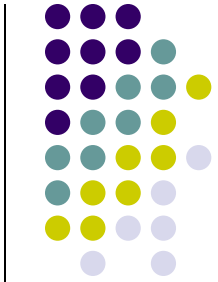


- Consider our calc language again
- We compute the lookahead sets in terms of tokens

Listing 2.9: The grammar for the calc language with lookahead sets.

```
1  explist : ({NUM,PLUS,MINUS,LPAREN} exp)*
2
3  exp : {NUM} NUM
4      | {PLUS,MINUS} op exp exp
5      | {LPAREN} LPAREN op exp exp ({NUM,PLUS,MINUS,LPAREN} exp)* RPAREN
6
7  op : {PLUS} PLUS
8     | {MINUS} MINUS
```

Parsing with Lexers



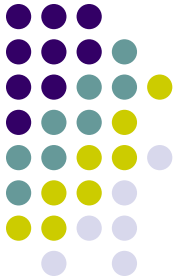
- Now it is straightforward to build the parser

```
def explist(stream):
    while stream.pointer().type in ['NUM', 'PLUS', 'MINUS', 'LPAREN']:
        exp(stream)
    return
```

```
def exp(stream):
    token = stream.pointer()
    if token.type in ['NUM']:
        stream.match('NUM')
        return
    elif token.type in ['PLUS','MINUS']:
        op(stream)
        exp(stream)
        exp(stream)
        return
    elif token.type in ['LPAREN']:
        stream.match('LPAREN')
        op(stream)
        exp(stream)
        exp(stream)
        while stream.pointer().type in ['NUM', 'PLUS', 'MINUS', 'LPAREN']:
            exp(stream)
        stream.match('RPAREN')
    else:
        raise SyntaxError("syntax error at {}".format(token.type))
```

```
def op(stream):
    token = stream.pointer()
    if token.type in ['PLUS']:
        stream.match('PLUS')
        return
    elif token.type in ['MINUS']:
        stream.match('MINUS')
        return
    else:
        raise SyntaxError("syntax error at {}".format(token.type))
```

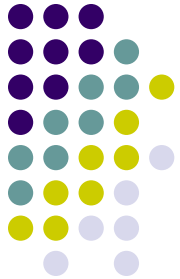
Parsing with Lexers



- Top-level driver function

```
def parse():
    from calc_lexer import Lexer
    from sys import stdin
    try:
        char_stream = stdin.read() # read from stdin
        token_stream = Lexer(char_stream)
        explist(token_stream) # call the parser function for start symbol
        if token_stream.end_of_file():
            print("parse successful")
        else:
            raise SyntaxError("bad syntax at {}"
                              .format(token_stream.pointer()))
    except Exception as e:
        print("error: " + str(e))
```

Parsing with Lexers



- Running the parser

```
$ python3 calc_parser.py
+ 10 25
^D
parse successful
```

```
$ python3 calc_parser.py
(+ 1 2 3)
^D
parse successful
```

```
$ python3 calc_parser.py
+ (+ 1 2 3) 4
^D
parse successful
```