TIE-20306 Principles of Programming Languages, autumn 2019

Last change 03.10.2019 08:44.


```
This page change history:
2019-09-30: underscore (_) is no longer valid part of constIDENT
            DOUBLEDOT (..) as a new token
2019-09-25: removed a mention to groups (this project is individual work)
2019-09-24: first public version
```

Project work, phase 1

Lexical Analysis

Basic idea of the lexical phase is to convert a human readable file into a list of tokens. One efficient technique to identify tokens is to use Regular Expressions. Using this quite unreadable "matching" language will quarantee that the input file is read only once and we will get all the defined matches.

Some resources about regex and the Ply

- Learn ReGex the Easy Way
- Online exercises
- PLY regex/lex

TupLang

Definition of Tokens

In the course language TupLang these and only these tokens are recognized from an input stream of characters:

```
# non-tokens

WHITESPACE ::= ** empty space, tabulator(\t) and newline(\n)/linefeed(\r)
               are accepted but ignored in the input (for each newline
               keep a line count to get better error messages) **

COMMENT ::= ** anything between braces { }
               are accepted but ignored **


# reserved words (each identified as a token) are:
define, begin, end, each, select

# one and two letter tokens:
LARROW ::= '<-'
RARROW ::= '->'
LPAREN ::= '('
RPAREN ::= ')'
LSQUARE ::= '['
RSQUARE ::= ']'
COMMA  ::= ','
```

```
DOT    ::= '.'
PIPE   ::= '|'
DOUBLEPLUS ::= '++'
DOUBLEMULT ::= '**'
DOUBLEDOT  ::= '..'
COLON ::= ':'

EQ    ::= '='
NOTEQ ::= '!='
LT    ::= '<'
LTEQ  ::= '<='
GT    ::= '>'
GTEQ  ::= '>='
PLUS  ::= '+'
MINUS ::= '-'
MULT  ::= '*'
DIV   ::= '/'
MOD   ::= '%'


# longer tokens

NUMBER_LITERAL ::= ** one or more numerical digits **

STRING_LITERAL ::= ** any number of characters inside vertical double
                      quotation marks.  E.g.  "merkkijono" **

varIDENT ::= ** a variable name starts with a lowercase letter (a-z) and
             must be followed by at least one character in
             set( 'a-z', 'A-Z', '0-9', '_' ).
             NOTE that this does not allow
             one letter variable names. E.g. valid varIDENT:
             ab, iI, i9_abc, a9 **

constIDENT ::= ** a constant identifier contain one or more
               characters in 'A-Z'. E.g. valid constIDENT:
               N, PIE, ROUND **

tupleIDENT ::= ** a tuple variable starts with '<' and must be followed
               at least one lowercase char ('a-z'). The last char
               must be '>'. E.g. valid tupleIDENT:
               <x>, <mytuple> **


funcIDENT ::= ** a function name starts with an uppercase letter (A-Z) an
              must be followed by at least one character in
              set( 'a-z', '0-9', '_' ). NOTE that this does not allow
              one letter function names. E.g. valid funcIDENT:
              Foo, J00, S_o_m_e **
```

If the input file contains any characters not recognized as tokens or ignored characters then the whole input is invalid.

Python

A character is anything you will read from a file using python3 open()-function with utf-8 encoding.

```
#!/usr/bin/env python3
with open('file.input', 'r', encoding='utf-8') as INFILE:
    data = INFILE.read()
```

This data is the input stream for your lexer (implemented with the PLY module).

a Lexer in PLY

You should map the token definitions used in this document to PLY-tokens. So `each:` in this definition becomes `t_EACH` in PLY. This is not strictly neccessary, but it will save you from confusion/conflicts later on.

PLY documentation has a lot of examples. A simple but fully functional example snippet is provided by the course staff also.

What to submit

Make sure that these are true:

1. Put all your tested SOURCE code (no virtual environments, no *.pyc compiled files) into the prenamed directory in your git repository. ALL the code in same directory (even if this means that you need to copy a file from a previous submission).
2. Write a submission document (README.txt, README.md or README.pdf) which contains the required written part of your submission.

After BOTH of previous items are ready, then submit your git hashID to the course Plussa. More information in the submission page.

Minimal functionality

Your program must start when executed in python by: `python3 main.py` in your submission directory. You can assume that this python interpreter has PLY installed into it.

Your program must understand the following command line arguments:

```
usage: main.py [-h] [--who | -f FILE]
  -h, --help            show this help message and exit
  --who                 print out student IDs and NAMEs of authors
  -f FILE, --file FILE  filename to process
```

(See the course snippet on how to implement this using python standard library.)

Identify tokens using ply.lex and print them out one per line. Use the default ply.lex formatting (you can just say `print( token )` in python). The correct output will be like:

```
LexToken(varIDENT,'message',1,0)
LexToken(LARROW,'<-',1,8)
LexToken(STRING_LITERAL,'Hello popl',1,11)
LexToken(DOT,'.',1,23)
LexToken(funcIDENT,'Print',2,25)
LexToken(LSQUARE,'[',2,30)
LexToken(varIDENT,'message',2,31)
LexToken(RSQUARE,']',2,38)
LexToken(DOT,'.',2,39)
```

Code examples and their expected outputs are available in course repository DO NOT DEPEND solely on those - write your own testcases so that you can be sure that your program works correctly.

The printed fields in these LexTokens are: token.type, token.value, token.lineno, and token.lexpos (automatically filled by the PLY). Only the first item (token.type) should be as shown in the example codes - other fields depend on your processing of values etc. and do not have to be exactly the same as in course examples.

Errors

If your lexer encounters an unknown character it must report the finding with message:
`"Illegal character '{CHAR}' at line {LINE}"`
AND stop the lexer execution immediately.

Required text document

In addition to the lexer code, the you submit a text document in either plain text (.txt), markdown (.md) or pdf (.pdf) format. It should contain answers to the following questions (again, in your **own words**, no copying from other sources):

1. What is lexical analysis and how is it related to other parts in compilation?
2. How is the lexical structure of the language expressed in the PLY tool? I.e., what parts are needed in the code and how are they related to lexical rules of the language?
3. Explain how the following are recognized and handled in your code:
    a. Keywords
    b. Comments
    c. Whitespace between tokens
    d. Operators & delimiters (<-, parenthesis, etc.)
    e. Integer literals
    f. String literals
    g. Function names
    h. Tuple names
4. How can the lexer distinguish between the following lexical elements:
    a. Function names & constant names
    b. Keywords & variable names
    c. Operators - (minus) & -> (right arrow)
    d. String literals & variables names
    e. Comments & other code
    f. Tuple names & two variables compared to each other with <.
5. Did you implement any extras? If so explain them (what and how)
6. What did you think of this assignment? What was difficult? What was easy? Did you learn anything useful?

Extras

In addition to the minimal functionality here are some ideas for extras. Extras do NOT automatically increase your points but they will help... Implementation defails of these items are freely defined by you (what kind of error message etc.)

- Implement comment in a way that it can span multiple lines.
- Accept nested comments (`{ start { inside } out again }`)

You can also invent your own extra features (new error messages, token value checks etc.). DOCUMENT your additions. Only those features which are mentioned in the text document are considered when grading the submission.

Questions with answers

Nothing at the moment.

General practical questions about the course should be sent to popl@lists.tuni.fi.