Các công đoạn: Đọc document ANTLR, Làm tutorial, Đọc specification BKIT, Làm btl

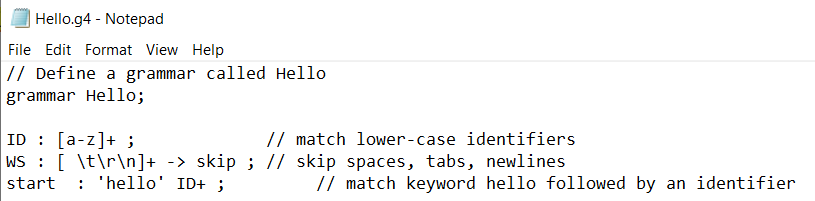
* File batch là file kịch bản, sẽ thực thi các dòng lệnh đc quy định sẵn trong file (giống .sh trong linux)
* ^D là ctrl+D tương ứng với kết thúc file (<EOF>) trong linux, trong win phải là ctrl+Z
* Chú ý trong các file hướng dẫn: các câu đc bắt đầu bằng kí hiệu $ mới là câu lệnh, còn lại là cmt
* Alias: là kí hiệu thay thế, hay còn gọi là biệt danh
* Path là đường dẫn để cài môi trường, có thể thực thi file ở bất cứ đâu trên hệ thống mà ko cần phải vào thư mục gốc vì đã có path dẫn đường

-3 bước thực hiện: antlr4 để sinh file .java, javac sinh ra file class từ file java, grun để show kết quả

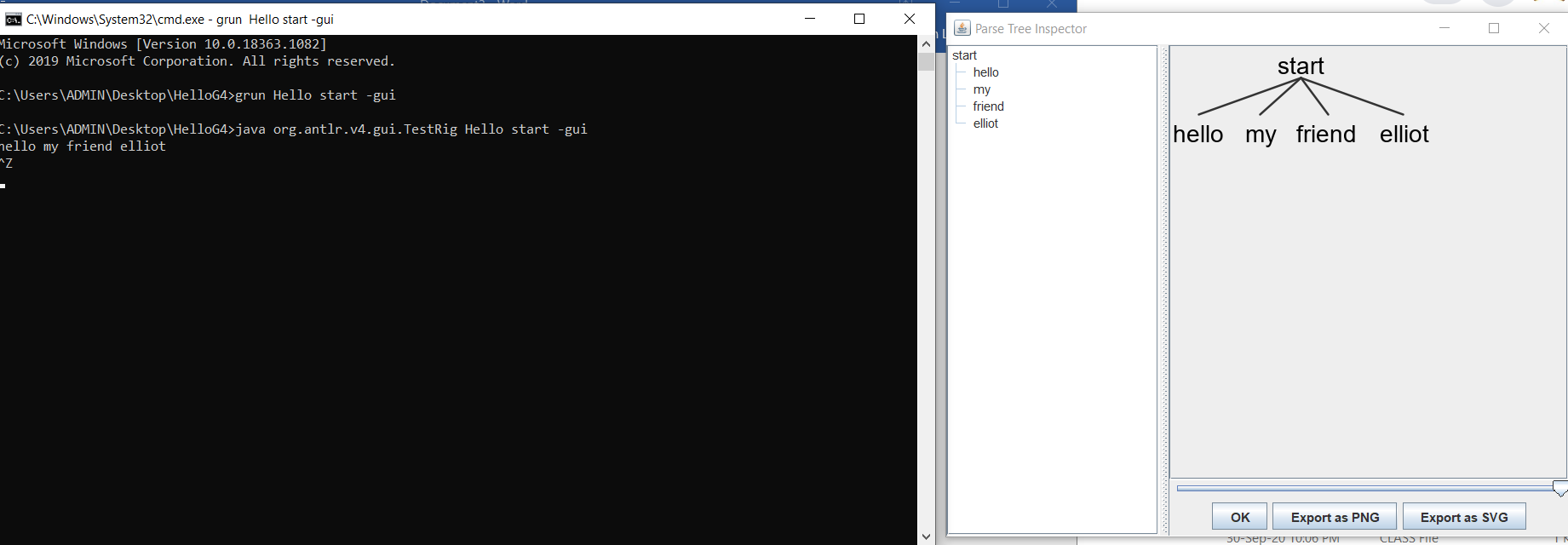
-Grun có 4 phần:

Grun GrammarName startRuleName [-tokens] [-tree] [-gui]

Tương ứng với: grun, Hello, start, -gui

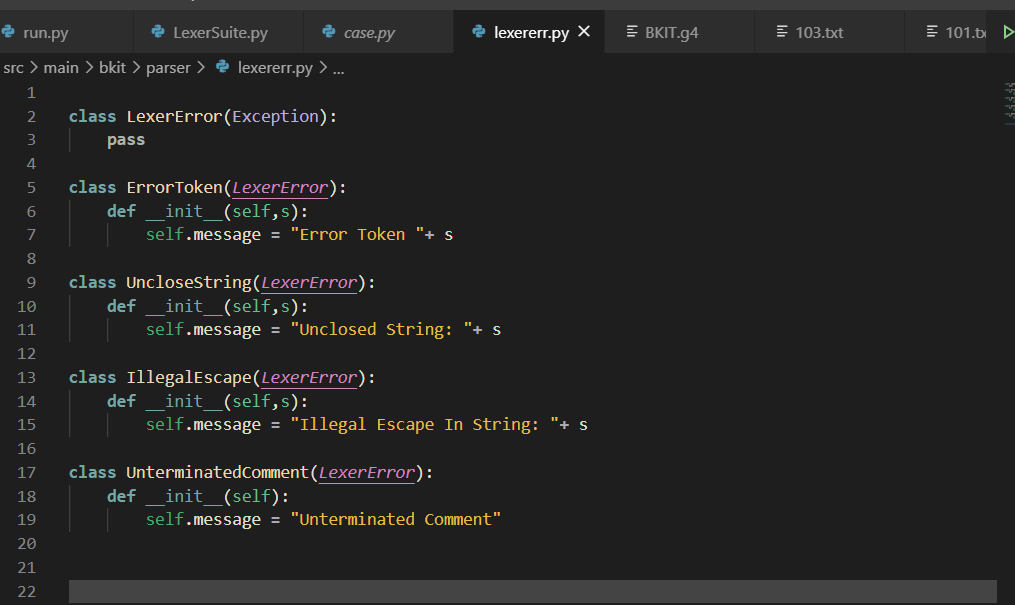


Kết quả:

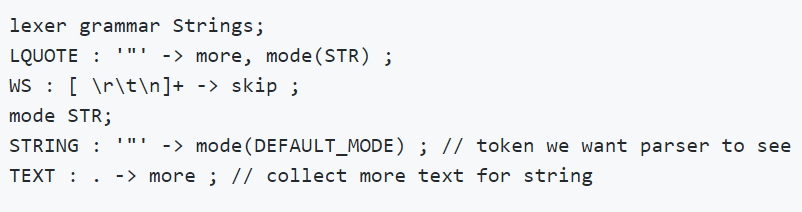


-Mục đích Check LexerSuite: tách các token phân biệt, so sánh result(solution.txt) với expect(LexerSuite.py), nếu == thì đúng còn != thì báo lỗi (error expect là 4 error định nghĩa trong btl)

-Lexer là những gì xét lỗi khi viết code, Parser là những gì xét lỗi khi compile



-Kiềm chế: +? Thay bằng \*, \*? Thay bằng \*



-Có 2 chế độ trong đoạn code: DEFAULT\_MODE và STR mode, khi nhận dạng token LQUOTE gặp command more bắt buộc phải tìm ra kí tự tiếp để hoàn thành token, ko sẽ bị lỗi, gặp command mode(STR) sẽ chuyển sang dò các Lexer phía dưới mode STR. Khi gặp lexer STRING mới chính thức hoàn thành chuỗi hợp lệ, chuyển sang DEFAULT\_MODE, bất cứ khi nào gặp LQUOTE lại tiếp tục vòng lặp đó.

-Có thể thêm hàm sinh testcase mà ko cần phải gọi ở đâu, nhờ vào parameter self

-~~Khi nhận dạng token sẽ lấy đơn vị đầu tiên nhận dạng đc ko phải đơn vị dài nhất~~ (rồi đem so sánh các đơn vị token với các lexer khác lấy token dài nhất)

COMMENT: '\*\*' .\*? '\*\*' -> skip;// .\*? matches anything until the first \*\*

\*?: lấy đầu tiên (không tham lam)

\*: lấy dài nhất (tham lam)

-Ưu tiên phép toán concat hơn phép |

FLOAT: ((DIGIT+ '.' DIGIT\*) | (DIGIT\* '.' DIGIT+)  (EXPONENT)?)

        | (DIGIT+  EXPONENT) ;

Sẽ hiểu là (DIGIT\* ‘.’ DIGIT+) (EXPONENT)? được tính toán trước

=>Cách đặt dấu |

FLOAT: ((DIGIT+ '.' DIGIT\* | DIGIT\* '.' DIGIT+)  (EXPONENT)?)

        | (DIGIT+  EXPONENT) ;

-Phải đặt tên def trong LexerSuite là test\_... mới đc tính

-Tất cả testcase đều phải “”” testcase””” thì mới ghi mấy escape \ bình thường đc, ko thì đang viết thông qua ngôn ngữ khác (thêm \ phức tạp lắm)

\n\t\\ -> skip

[\\n](file:///\\n) [\\t](file:///\\t) \\\\ -> escape\_string

\conccac\\ -> illegal\_escape

**Trong testcase:**

[\\k](\\\\k): hiểu là dấu ‘\’ + chữ k

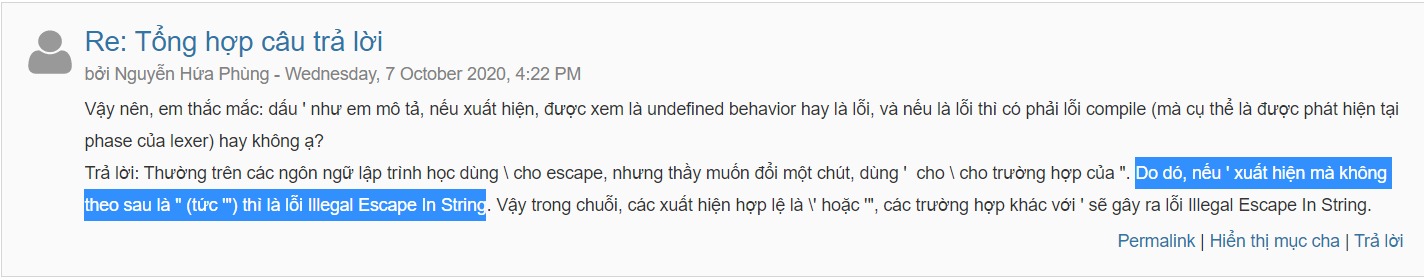
\k: cũng hiểu là dấu ‘\’ + chữ k

\n: hiểu là enter

[\\n](file:///\\n): hiểu là dấu ‘\’ + chữ n

#Để ý màu code (màu đỏ là 1 kí tự đặc biệt đc kết hợp từ 1 hoặc 2 kí tự): \n, \t…

Trong BKIT: escape\_string là \t\n\b\f\r\’ và ‘” (thầy ko thích dùng \” mà thích ‘”)



Trong BKIT ‘ xuất hiện theo sau là “ (‘”) giống Trong python \ xuất hiện theo sau là “

Trong BKIT \ xuất hiện theo sau là t,n,b,f,r,’ giống trong python

\ hay ‘ xuất hiện mà ko có kí tự hợp lệ theo sau sẽ escape string

* \ phải theo sau là bfrnt’\
* ‘ phải theo sau là “

Parser: xây dựng lên cây cấu trúc bắt nguồn từ program->…-> xây dựng tới đâu test tới đấy (có thể có mấy phần tử xây dựng rồi nhưng còn rời rạc, phải gắn vào cây)

-exp là bất cứ gì trong đó miễn trả về con số, chuỗi, {}

-operand và operator: ví dụ như em viết 1 + foo() thì operand của + là 1 và foo()

=> foo() sẽ có operator là foo và operand là phần còn lại

-Phân biệt parser với ngữ nghĩa: luôn đặt câu hỏi “trong cấu trúc này có những gì, cú pháp thế nào, những gì phải có”, ko đặt câu hỏi “cái này nằm ở đâu, đặt ở đây có đúng ko, break có đặt trong while ko?...”

-Các phép toán cộng trừ nhân chia là giống với ngoài đời vì có tính kết hợp trái (làm từ trái sang phải), phép funcall với == ko có tính kết hợp nên bắt buộc phải có dấu () mỗi bên exp (vd: a == b == c ko chấp nhận, chỉ có a == (b==c)..)

Chanel(), type(), pushpop(): <https://github.com/antlr/antlr4/blob/master/doc/lexer-rules.md>

Huynhsama: <https://github.com/huynhsamha/cse-ppl/tree/master/03_Object-Oriented-Programming>

Tutorial: <https://tomassetti.me/antlr-mega-tutorial/>