Assignment 2: Parsing with PLY (Python Lex-Yacc)

ECEN 4553 & 5013, CSCI 4555 & 5525
Prof. Jeremy G. Siek

The main ideas covered in this assignment are

- Lexical analysis: regular expressions and using lex
- Parsing: context-free grammars, the LALR(1) shift-reduce algorithm, and using yacc, including shift-reduce errors, reduce-reduce errors, and precedence declarations

In general, the syntax of the source code for a language is called its *concrete syntax*. The concrete syntax of P_0 specifies which programs, expressed as sequences of characters, are P_0 programs. The process of transforming a program written in the concrete syntax (a sequence of characters) into an abstract syntax tree is traditionally subdivided into two parts: *lexical analysis* (often called scanning) and *parsing*. The lexical analysis phase translates the sequence of characters into a sequence of *tokens*, where each token consists of several characters. We discuss lexical analysis in detail in Section 1. The parsing phase organizes the tokens into a *parse tree* as directed by the grammar of the language and then translates the parse tree into an abstract syntax tree. We discuss parsing in detail Section 2.

It is entirely feasible to implement a compiler without doing lexical analysis, instead just parsing. However, scannerless parsers tend to be slower, which mattered back when computers were slow, and sometimes still matters for very large files.

The Python Lex-Yacc tool, abbreviated PLY [1], is an easy-to-use Python imitation of the original `lex` and `yacc` C programs. Lex was written by Eric Schmidt and Mike Lesk [3] at Bell Labs, and is the standard lexical analyzer generator on many Unix systems. YACC stands from Yet Another Compiler Compiler and was originally written by Stephen C. Johnson at AT&T [2]. The PLY tool combines the functionality of both `lex` and `yacc`. In this assignment we will use the PLY tool to generate a lexer and parser for the P_0 subset of Python.

1 Lexical Analysis

The lexical analyzer turns a sequence of characters (a string) into a sequence of tokens. For example, the string

```
'print 1 + 3'
```

will be converted into the list of tokens

```
['print','1','+','3']
```

Actually, to be more accurate, each token will contain the token `type` and the token's `value`, which is the string from the input that matched the token.

With the PLY tool, the types of the tokens must be specified by initializing the `tokens` variable. For example,

```
tokens = ('PRINT','INT','PLUS')
```

Next we must specify which sequences of characters will map to each type of token. We do this using regular expression. The term "regular" comes from "regular languages", which are the (particularly simple) set of languages that can be recognized by a finite automata. A *regular expression* is a pattern formed of the following core elements:

1. a character, e.g. `a`. The only string that matches this regular expression is `a`.

2. two regular expressions, one followed by the other (concatenation), e.g. `bc`. The only string that matches this regular expression is `bc`.

3. one regular expression or another (alternation), e.g. `a|bc`. Both the string `'a'` and `'bc'` would be matched by this pattern.

4. a regular expression repeated zero or more times (Kleene closure), e.g. `(a|bc)*`. The string `'bcabcbc'` would match this pattern, but not `'bccba'`.

5. the empty sequence (epsilon)

The Python support for regular expressions goes beyond the core elements and include many other convenient short-hands, for example `+` is for repetition one or more times. If you want to refer to the actual character `+`, use a backslash to escape it. Section 4.2.1 Regular Expression Syntax of the Python Library Reference gives an in-depth description of the extended regular expressions supported by Python.

Normal Python strings give a special interpretation to backslashes, which can interfere with their interpretation as regular expressions. To avoid this problem, use Python's raw strings instead of normal strings by prefixing the string with an `r`. For example, the following specifies the regular expression for the `'PLUS'` token.

```
t_PLUS =   r'\+'
```

The `t_` is a naming convention that PLY uses to know when you are defining the regular expression for a token.

Sometimes you need to do some extra processing for certain kinds of tokens. For example, for the `INT` token it is nice to convert the matched input string into a Python integer. With PLY you can do this by defining a function for the token. The function must have the regular expression as its documentation string and the body of the function should overwrite in the `value` field of the token. Here's how it would look for the `INT` token. The `\d` regular expression stands for any decimal numeral (0-9).

```
def t_INT(t):
    r'\d+'
    try:
      t.value = int(t.value)
    except ValueError:
      print "integer value too large", t.value
      t.value = 0
    return t
```

In addition to defining regular expressions for each of the tokens, you'll often want to perform special handling of newlines and whitespace. The following is the code for counting newlines and for telling the lexer to ignore whitespace. (Python has complex rules for dealing with whitespace that we'll ignore for now.)

```
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

t_ignore  = ' \t'
```

If a portion of the input string is not matched by any of the tokens, then the lexer calls the error function that you provide. The following is an example error function.

```
def t_error(t):
    print "Illegal character '%s'" % t.value[0]
    t.lexer.skip(1)
```

Last but not least, you'll need to instruct PLY to generate the lexer from your specification with the following code.

```
import ply.lex as lex
lex.lex()
```

Figure 1 shows the complete code for an example lexer.

```
tokens = ('PRINT','INT','PLUS')

t_PRINT = r'print'

t_PLUS =   r'\+'

def t_INT(t):
    r'\d+'
    try:
      t.value = int(t.value)
    except ValueError:
      print "integer value too large", t.value
      t.value = 0
    return t

t_ignore  = ' \t'

def t_newline(t):
  r'\n+'
  t.lexer.lineno += t.value.count("\n")

def t_error(t):
  print "Illegal character '%s'" % t.value[0]
  t.lexer.skip(1)

import ply.lex as lex
lex.lex()
```

Figure 1: Example lexer implemented using the PLY lexer generator.

Exercise 1.1. Write a PLY lexer specification for P_0 and test it on a few input programs, looking at the output list of tokens to see if they make sense.

2 Parsing

We start with some background on context-free grammars (Section 2.1), then discuss how to use PLY to do parsing (Section 2.2). To understand the error messages of PLY, one needs to understand the underlying parsing algorithm, so we discuss the algorithm it uses in Sections 2.3 and 2.4. This section concludes with a discussion of using precedence levels to resolve parsing conflicts.

2.1 Background on context-free grammars

A *context-free grammar* consists of a set of *rules* (also called productions) that describes how to categorize strings of various forms. There are two kinds of categories, *terminals* and *non-terminals*. The terminals correspond to the tokens from the lexical analysis. Non-terminals are used to categorize different parts of a language, such as the distinction between statements and expressions in Python and C. The term *symbol* refers to both terminals and non-terminals. A grammar rule has two parts, the left-hand side is a non-terminal and the right-hand side is a sequence of zero or more symbols. The notation `::=` is used to separate the left-hand side from the right-hand side. The following is a rule that could be used to specify the syntax for an addition operator.

(1) `expression ::= expression PLUS expression`

This rule says that if a string can be divided into three parts, where the first part can be categorized as an expression, the second part is the `PLUS` non-terminal (token), and the third part can be categorized as an expression, then the entire string can be categorized as an expression. The next example rule has the non-terminal `INT` on the right-hand side and says that a string that is categorized as an integer (by the lexer) can also be categorized as an expression. As is apparent here, a string can be categorized by more than one non-terminal.

(2) `expression ::= INT`

To *parse* a string is to determine how the string can be categorized according to a given grammar. Suppose we have the string "`1 + 3`". Both the `1` and the `3` can be categorized as expressions using rule 2. We can then use rule 1 to categorize the entire string as an expression. A *parse tree* is a good way to visualize the parsing process. (You will be tempted to confuse parse trees and abstract syntax tress, but the excellent students will carefully study the difference to avoid this confusion.) A parse tree for "`1 + 3`" is shown in Figure 2. The best way to start drawing a parse tree is to first list the tokenized string at the bottom of the page. These tokens correspond to terminals and will form the leaves of the parse tree. You can then start to categorize non-terminals, or sequences of non-terminals, using the parsing rules. For example, we can categorize the integer "`1`" as an expression using rule (2), so we create a new node above "`1`", label the node with the left-hand side terminal, in this case `expression`, and draw a line down from the new node down to "`1`". As an optional step, we can record which rule we used in parenthesis after the name of the terminal. We then repeat this process until all of the leaves have been connected into a single tree, or until no more rules apply.

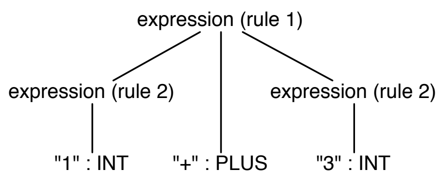


Figure 2: The parse tree for "`1 + 3`".

There can be more than one parse tree for the same string if the grammar is ambiguous. For example, the string "`1 + 2 + 3`" can be parsed two different ways using rules 1 and 2, as shown in Figure 3. In Section 2.5 we'll discuss ways to avoid ambiguity through the use of precedence levels and associativity.

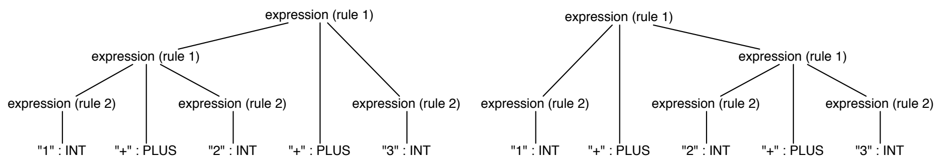


Figure 3: Two parse trees for "`1 + 2 + 3`".

The process describe above for creating a parse-tree was "bottom-up". We started at the leaves of the tree and then worked back up to the root. An alternative way to build parse-trees is the "top-down" *derivation* approach. This approach is not a practical way to parse a particular string but it is helpful for thinking about all possible strings that are in the language described by the grammar. To perform a derivation, start by drawing a single node labeled with the starting non-terminal for the grammar. This is often the `program` non-terminal, but in our case we simply have `expression`. We then select at random any grammar rule that has `expression` on the left-hand side and add new edges and nodes to the tree according to the right-hand side of the rule. The derivation process then repeats by selecting another non-terminal that does not yet have children. Figure 4 shows the process of building a parse tree by derivation. A *left-most derivation* is one in which the left-most non-terminal is always chosen as the next non-terminal to expand. A `right-most derivation` is one in which the right-most non-terminal is always chosen as the next non-terminal to expand. The derivation in Figure 4 is a right-most derivation.

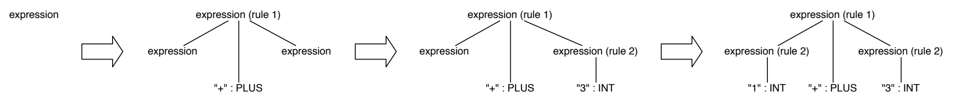


Figure 4: Building a parse-tree by derivation.

In the assignment descriptions for this course we specify which language features are in a given subset of Python using context-free grammars. The notation we'll use for grammars is Extended Backus-Naur Form (EBNF). The grammar for P_0 is shown in Figure 5. This notation does not correspond exactly to the notation for grammars used by PLY, but it should not be too difficult for the reader to figure out the PLY grammar given the EBNF grammar.

```
program ::= module
module ::= simple_statement+
simple_statement ::= "print" expression ("," expression)*
unary_op ::= "+" | "-"
binary_op ::= "+" | "-" | "*" | "/" | "%" | "**"
expression ::= integer
             | unary_op expression
             | expression binary_op expression
             | "(" expression ")"
             | "input" "(" ")"
```

Figure 5: Context-free grammar for the P_0 subset of Python.

2.2 Using PLY

Figure 6 shows an example use of PLY to generate a parser. The code specifies a grammar and it specifies actions for each rule. For each grammar rule there is a function whose name must begin with `p_`. The document string of the function contains the specification of the grammar rule. PLY uses just a colon `:` instead of the usual `::=` to separate the left and right-hand sides of a grammar production. The left-hand side symbol for the first function (as it appears in the Python file) is considered the start symbol. The body of these functions contains code that carries out the action for the production.

Typically, what you want to do in the actions is build an abstract syntax tree, as we do here. The parameter `t` of the function contains the results from the actions that were carried out to parse the right-hand side of the production. You can index into `t` to access these results, starting with `t[1]` for the first symbol of the right-hand side. To specify the result of the current action, assign the result into `t[0]`. So, for example, in the production `expression : INT`, we build a `Const` node containing an integer that we obtain from `t[1]`, and we assign the `Const` node to `t[0]`.

```
from compiler.ast import Printnl, Add, Const

def p_print_statement(t):
  'statement : PRINT expression'
  t[0] = Printnl([t[2]], None)

def p_plus_expression(t):
  'expression : expression PLUS expression'
  t[0] = Add((t[1], t[3]))

def p_int_expression(t):
  'expression : INT'
  t[0] = Const(t[1])

def p_error(t):
  print "Syntax error at '%s'" % t.value

import ply.yacc as yacc
yacc.yacc()
```

Figure 6: First attempt at writing a parser using PLY.

The PLY parser generator takes your grammar and generates a parser that uses the LALR(1) shift-reduce algorithm, which is the most common parsing algorithm in use today. LALR(1) stands for Look Ahead Left-to-right with Rightmost-derivation and 1 token of lookahead. Unfortunately, the LALR(1) algorithm cannot handle all context-free grammars, so sometimes you will get error messages from PLY. To understand these errors and know how to avoid them, you have to know a little bit about the parsing algorithm.

2.3 The LALR(1) Algorithm

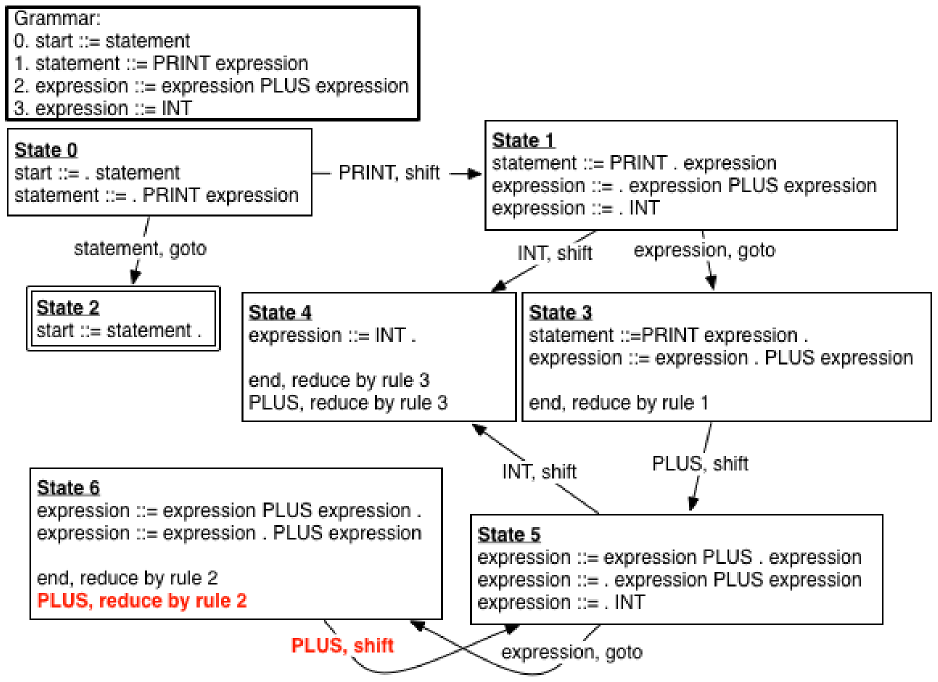
The LALR(1) algorithm uses a stack and a finite automata. Each element of the stack is a pair: a state number and a symbol. The symbol characterizes the input that has been parsed so-far and the state number is used to remember how to proceed once the next symbol-worth of input has been parsed. Each state in the finite automata represents where the parser stands in the parsing process with respect to certain grammar rules. Figure 7 shows an example LALR(1) parse table generated by PLY for the grammar specified in Figure 6. When PLY generates a parse table, it also outputs a textual representation of the parse table to the file `parser.out` which is useful for debugging purposes.

Consider state 1 in Figure 7. The parser has just read in a `PRINT` token, so the top of the stack is `(1,PRINT)`. The parser is part of the way through parsing the input according to grammar rule 1, which is signified by showing rule 1 with a dot after the PRINT token and before the expression non-terminal. A rule with a dot in it is called an *item*. There are several rules that could apply next, both rule 2 and 3, so state 1 also shows those rules with a dot at the beginning of their right-hand sides. The edges between states indicate which transitions the automata should make depending on the next input token. So, for example, if the next input token is INT then the parser will push INT and the target state 4 on the stack and transition to state 4. Suppose we are now at the end of the input. In state 4 it says we should reduce by rule 3, so we pop from the stack the same number of items as the number of symbols in the right-hand side of the rule, in this case just one. We then momentarily jump to the state at the top of the stack (state 1) and then follow the goto edge that corresponds to the left-hand side of the rule we just reduced by, in this case `expression`, so we arrive at state 3. (A slightly longer example parse is shown in Figure 7.)

In general, the shift-reduce algorithm works as follows. Look at the next input token.

- If there there is a shift edge for the input token, push the edge's target state and the input token on the stack and proceed to the edge's target state.

- If there is a reduce action for the input token, pop k elements from the stack, where k is the number of symbols in the right-hand side of the rule being reduced. Jump to the state at the top of the stack and then follow the goto edge for the non-terminal that matches the left-hand side of the rule we're reducing by. Push the edge's target state and


Grammar:
0. start ::= statement
1. statement ::= PRINT expression
2. expression ::= expression PLUS expression
3. expression ::= INT
State 0
start ::= . statement
statement ::= . PRINT expression
PRINT, shift
State 1
statement ::= PRINT . expression
expression ::= . expression PLUS expression
expression ::= . INT
statement, goto
INT, shift
expression, goto
State 2
start ::= statement .
State 4
expression ::= INT .
end, reduce by rule 3
PLUS, reduce by rule 3
State 3
statement ::=PRINT expression .
expression ::= expression . PLUS expression
end, reduce by rule 1
INT, shift
PLUS, shift
State 6
expression ::= expression PLUS expression .
expression ::= expression . PLUS expression
end, reduce by rule 2
PLUS, reduce by rule 2
State 5
expression ::= expression PLUS . expression
expression ::= . expression PLUS expression
expression ::= . INT
PLUS, shift
expression, goto

Example parse of 'print 1 + 2'

Stack	Input	Action
[]	'print 1 + 2'	shift to state 1
[(1,PRINT)]	'1 + 2'	shift to state 4
[(1,PRINT),(4,INT)]	'+ 2'	reduce by rule 3 to state 1, goto 3
[(1,PRINT),(3,expression)]	'+ 2'	shift to state 5
[(1,PRINT),(3,expression),(5,+)]	'2'	shift to state 4
[(1,PRINT),(3,expression),(5,+),(4,INT)]	''	reduce by rule 3 to state 5, goto 6
[(1,PRINT),(3,expression),(5,+),(6,expression)]	''	reduce by rule 2 to state 1, goto 3
[(1,PRINT),(3,expression)]	''	reduce by rule 1 to state 0, goto 2
[(2,statement)]	''	accept

Figure 7: An LALR(1) parse table and a trace of an example run.

the non-terminal on the stack.

Notice that in state 6 of Figure 7 there is both a shift and a reduce action for the token `PLUS`, so the algorithm does not know which action to take in this case. When a state has both a shift and a reduce action for the same token, we say there is a *shift/reduce conflict*. In this case, the conflict will arise, for example, when trying to parse the input `print 1 + 2 + 3`. After having consumed `print 1 + 2` the parser will be in state 6, and it will not know whether to reduce to form an expression of `1 + 2`, or whether it should proceed by shifting the next `+` from the input.

A similar kind of problem, known as a *reduce/reduce* conflict, arises when there are two reduce actions in a state for the same token. To understand which grammars gives rise to shift/reduce and reduce/reduce conflicts, it helps to know how the parse table is generated from the grammar, which we discuss next.

2.4 Parse Table Generation

The parse table is generated one state at a time. State 0 represents the start of the parser. We add the production for the start symbol to this state with a dot at the beginning of the right-hand side. If the dot appears immediately before another non-terminal, we add all the productions with that non-terminal on the left-hand side. Again, we place a dot at the beginning of the right-hand side of each the new productions. This process called *state closure* is continued until there are no more productions to add. We then examine each item in the current state I. Suppose an item has the form $A ::= \alpha.X\beta$, where A and X are symbols and α and β are sequences of symbols. We create a new state, call it J. If X is a terminal, we create a shift edge from I to J, whereas if X is a non-terminal, we create a goto edge from I to J. We then need to add some items to state J. We start by adding all items from state I that have the form $B ::= \gamma.X\kappa$ (where B is any symbol and γ and κ are arbitrary sequences of symbols), but with the dot moved past the X. We then perform state closure on J. This process repeats until there are no more states or edges to add.

We then mark states as accepting states if they have an item that is the start production with a dot at the end. Also, to add in the reduce actions, we look for any state containing an item with a dot at the end. Let n be the rule number for this item. We then put a reduce n action into that state for every token Y. For example, in Figure 7 state 4 has an item with a dot at the end. We therefore put a reduce by rule 3 action into state 4 for every token. (Figure 7 does not show a reduce rule for INT in state 4 because this grammar does not allow two consecutive INT tokens in the input. We will not go into how this can be figured out, but in any event it does no harm to have a reduce rule for INT in state 4; it just means the input will be rejected at a later point in the parsing process.)

Exercise 2.1. On a piece of paper, walk through the parse table generation process for the grammar in Figure 6 and check your results against Figure 7.

2.5 Resolving conflicts with precedence declarations

To solve the shift/reduce conflict in state 6, we can add the following precedence rules, which says addition associates to the left and takes precedence over printing. This will cause state 6 to choose reduce over shift.

```
precedence = (
    ('nonassoc','PRINT'),
    ('left','PLUS')
    )
```

In general, the precedence variable should be assigned a tuple of tuples. The first element of each inner tuple should be an associativity (nonassoc, left, or right) and the rest of the elements should be tokens. The tokens that appear in the same inner tuple have the same precedence, whereas tokens that appear in later tuples have a higher precedence. Thus, for the typical precedence for arithmetic operations, we would specify the following:

```
precedence = (
    ('left','PLUS','MINUS'),
    ('left','TIMES','DIVIDE')
    )
```

Figure 8 shows a complete Python program that generates a lexer and parser using PLY, parses an input file (the file specified on the command-line), and prints the AST.

Exercise 2.2. Write a PLY grammar specification for P_0 and update your compiler so that it uses the generated lexer and parser instead of using the parser in the `compiler` module. Perform regression testing on your compiler to make sure that it still passes all of the tests that you created for assignment 1.

References

[1] D. Beazley. PLY (Python Lex-Yacc). `http://www.dabeaz.com/ply/`.

```
################################################################################
# Lexer
tokens = ('PRINT','INT','PLUS')
t_PRINT = r'print'
t_PLUS = r'\+'
def t_INT(t):
    r'\d+'
    try:
      t.value = int(t.value)
    except ValueError:
      print "integer value too large", t.value
      t.value = 0
    return t
t_ignore  = ' \t'
def t_newline(t):
  r'\n+'
  t.lexer.lineno += t.value.count("\n")
def t_error(t):
  print "Illegal character '%s'" % t.value[0]
  t.lexer.skip(1)
import ply.lex as lex
lex.lex()
################################################################################
# Parser
from compiler.ast import Printnl, Add, Const
precedence = (
    ('nonassoc','PRINT'),
    ('left','PLUS')
    )
def p_print_statement(t):
  'statement : PRINT expression'
  t[0] = Printnl([t[2]], None)
def p_plus_expression(t):
  'expression : expression PLUS expression'
  t[0] = Add((t[1], t[3]))
def p_int_expression(t):
  'expression : INT'
  t[0] = Const(t[1])
def p_error(t):
  print "Syntax error at '%s'" % t.value
import ply.yacc as yacc
yacc.yacc()
################################################################################
# Main
import sys
try:
    f = open(sys.argv[1])
    p = yacc.parse(f.read())
    print p
except EOFError:
    print "Could not open file %s." % sys.argv[1]
```

Figure 8: Parser with precedence declarations to resolve conflicts.

[2] S. C. Johnson. Yacc: Yet another compiler-compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, 1979.

[3] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. Technical report, Bell Laboratories, July 1975.