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.]

For each subset of Python in this course, we will 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. Any symbol not appearing on the left-hand side of a rule is a terminal (e.g., name and decimalinteger). For simple terminals consisting of single strings, we simply use the string and avoid giving names to them (e.g., "+"). 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.

![Figure 5. Context-free grammar for the $P_0$ subset of Python.]

### 2.3. Generating parsers with 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 \( \text{expression} : \text{INT} \), we build a \text{Const} node containing an integer that we obtain from \( t[1] \), and we assign the \text{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'\n value % s."

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.

2.4. The LALR(1) algorithm

To understand the error messages of PLY, one needs to understand the underlying parsing 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 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.
Figure 7. An LALR(1) parse table and a trace of an example run.

- 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 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 \(1 + 2 + 3\). After having consumed \(1 + 2\) 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.

### 2.4.1. 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. Suppose an item has the form \(A := \alpha.X\beta\), where \(A\) and \(X\) are symbols and \(\alpha\) and \(\beta\) 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 \(\gamma\) and \(\kappa\) 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 2.2. 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.

2.4.2. 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.

\[
\text{precedence} = (\text{'nonassoc','PRINT'}, \text{'left','PLUS'})
\]

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:

\[
\text{precedence} = (\text{'left','PLUS','MINUS'}, \text{'left','TIMES','DIVIDE'})
\]

Figure 8 shows the Python code for generating a lexer and parser using PLY.

Exercise 2.3. 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. In addition to handling the grammar in Figure 5, you also need to handle Python-style comments, everything following a \# symbol up to the newline should be ignored. Perform regression testing on your compiler to make sure that it still passes all of the tests that you created for \( P_0 \).
# 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 = ' 	'

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

def t_error(t):
    print 'Illegal character ', 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 ', t.value

import ply.yacc as yacc
yacc.yacc()  

**Figure 8.** Example parser with precedence declarations to resolve conflicts.