SLY (Sly Lex Yacc)

This document provides an overview of lexing and parsing with SLY. Given the intrinsic complexity of parsing, I would strongly advise that you read (or at least skim) this entire document before jumping into a big development project with SLY.

SLY requires Python 3.6 or newer. If you're using an older version, you're out of luck. Sorry.

Introduction

SLY is library for writing parsers and compilers. It is loosely based on the traditional compiler construction tools lex and yacc and implements the same LALR(1) parsing algorithm. Most of the features available in lex and yacc are also available in SLY. It should also be noted that SLY does not provide much in the way of bells and whistles (e.g., automatic construction of abstract syntax trees, tree traversal, etc.). Nor should you view it as a parsing framework. Instead, you will find a bare-bones, yet fully capable library for writing parsers in Python.

The rest of this document assumes that you are somewhat familiar with parsing theory, syntax directed translation, and the use of compiler construction tools such as lex and yacc in other programming languages. If you are unfamiliar with these topics, you will probably want to consult an introductory text such as "Compilers: Principles, Techniques, and Tools", by Aho, Sethi, and Ullman. O'Reilly's "Lex and Yacc" by John Levine may also be handy. In fact, the O'Reilly book can be used as a reference for SLY as the concepts are virtually identical.

SLY Overview

SLY provides two separate classes `Lexer` and `Parser`. The `Lexer` class is used to break input text into a collection of tokens specified by a collection of regular expression rules. The `Parser` class is used to recognize language syntax that has been specified in the form of a context free grammar. The two classes are typically used together to make a parser. However, this is not a strict requirement–there is a great deal of flexibility allowed. The next two parts describe the basics.

Writing a Lexer

Suppose you're writing a programming language and you wanted to parse the following input string:

```
x = 3 + 42 * (s - t)
```

The first step of parsing is to break the text into tokens where each token has a type and value. For example, the above text might be described by the following list of token tuples:

```
[ ('ID','x'), ('EQUALS','='), ('NUMBER','3'),
  ('PLUS','+'), ('NUMBER','42'), ('TIMES','*'),
  ('LPAREN','('), ('ID','s'), ('MINUS','-'),
  ('ID','t'), ('RPAREN',')') ]
```

The SLY `Lexer` class is used to do this. Here is a sample of a simple lexer that tokenizes the above text:

```
# calclex.py

from sly import Lexer

class CalcLexer(Lexer):
    # Set of token names.   This is always required
    tokens = { ID, NUMBER, PLUS, MINUS, TIMES,
               DIVIDE, ASSIGN, LPAREN, RPAREN }

    # String containing ignored characters between tokens
    ignore = ' \t'

    # Regular expression rules for tokens
    ID      = r'[a-zA-Z_][a-zA-Z0-9_]*'
    NUMBER  = r'\d+'
    PLUS    = r'\+'
    MINUS   = r'-'
    TIMES   = r'\*'
    DIVIDE  = r'/'
    ASSIGN  = r'='
    LPAREN  = r'\('
    RPAREN  = r'\)'

if __name__ == '__main__':
    data = 'x = 3 + 42 * (s - t)'
    lexer = CalcLexer()
    for tok in lexer.tokenize(data):
        print('type=%r, value=%r' % (tok.type, tok.value))
```

When executed, the example will produce the following output:

When executed, the example will produce the following output:

```
type='ID', value='x'
type='ASSIGN', value='='
type='NUMBER', value='3'
type='PLUS', value='+'
type='NUMBER', value='42'
type='TIMES', value='*'
type='LPAREN', value='('
type='ID', value='s'
type='MINUS', value='-'
type='ID', value='t'
type='RPAREN', value=')'
```

A lexer only has one public method `tokenize()`. This is a generator function that produces a stream of `Token` instances. The `type` and `value` attributes of `Token` contain the token type name and value respectively.

The tokens set

Lexers must specify a `tokens` set that defines all of the possible token type names that can be produced by the lexer. This is always required and is used to perform a variety of validation checks.

In the example, the following code specified the token names:

```
class CalcLexer(Lexer):
    ...
    # Set of token names.   This is always required
    tokens = { ID, NUMBER, PLUS, MINUS, TIMES,
               DIVIDE, ASSIGN, LPAREN, RPAREN }
    ...
```

Token names should be specified using all-caps as shown.

Specification of token match patterns

Tokens are specified by writing a regular expression rule compatible with the `re` module. The name of each rule must match one of the names of the tokens provided in the `tokens` set. For example:

```
PLUS = r'\+'
MINUS = r'-'
```

Regular expression patterns are compiled using the `re.VERBOSE` flag which can be used to help readability. However, unescaped whitespace is ignored and comments are allowed in this mode. If your pattern involves whitespace, make sure you use `\s`. If you need to match the `#` character, use `[#]` or `\#`.

Tokens are matched in the same order that patterns are listed in the `Lexer` class. Longer tokens always need to be specified before short tokens. For example, if you wanted to have separate tokens for `=` and `==`, you need to make sure that `==` is listed first. For example:

```
class MyLexer(Lexer):
    tokens = { ASSIGN, EQ, ...}
    ...
    EQ     = r'=='       # MUST APPEAR FIRST! (LONGER)
    ASSIGN = r'='
```

Discarded text

The special `ignore` specification is reserved for single characters that should be completely ignored between tokens in the input stream. Usually this is used to skip over whitespace and other non-essential characters. The characters given in `ignore` are not ignored when such characters are part of other regular expression patterns. For example, if you had a rule to capture quoted text, that pattern can include the ignored characters (which will be captured in the normal way). The main purpose of `ignore` is to ignore whitespace and other padding between the tokens that you actually want to parse.

You can also discard more specialized text patterns by writing special regular expression rules with a name that includes the prefix `ignore_`. For example, this lexer includes rules to ignore comments and newlines:

```
# calclex.py

from sly import Lexer

class CalcLexer(Lexer):
    ...
    # String containing ignored characters (between tokens)
    ignore = ' \t'

    # Other ignored patterns
    ignore_comment = r'\#.*'
    ignore_newline = r'\n+'
    ...
```

```
if __name__ == '__main__':
    data = '''x = 3 + 42
                * (s    # This is a comment
                  - t)'''
    lexer = CalcLexer()
    for tok in lexer.tokenize(data):
        print('type=%r, value=%r' % (tok.type, tok.value))
```

Adding Match Actions

When certain tokens are matched, you may want to trigger some kind of action that performs extra processing. For example, converting a numeric value or looking up language keywords. One way to do this is to write your action as a method and give the associated regular expression using the `@_()` decorator like this:

```
@_(r'\d+')
def NUMBER(self, t):
    t.value = int(t.value)   # Convert to a numeric value
    return t
```

The method always takes a single argument which is an instance of type `Token`. By default, `t.type` is set to the name of the token (e.g., `'NUMBER'`). The function can change the token type and value as it sees appropriate. When finished, the resulting token object should be returned as a result. If no value is returned by the function, the token is discarded and the next token read.

The `@_()` decorator is defined automatically within the `Lexer` class–you don't need to do any kind of special import for it. It can also accept multiple regular expression rules. For example:

```
@_(r'0x[0-9a-fA-F]+',
   r'\d+')
def NUMBER(self, t):
    if t.value.startswith('0x'):
        t.value = int(t.value[2:], 16)
    else:
        t.value = int(t.value)
    return t
```

Instead of using the `@_()` decorator, you can also write a method that matches the same name as a token previously specified as a string. For example:

```
NUMBER = r'\d+'
...
def NUMBER(self, t):
    t.value = int(t.value)
    return t
```

This is potentially useful trick for debugging a lexer. You can temporarily attach a method a token and have it execute when the token is encountered. If you later take the method away, the lexer will revert back to its original behavior.

Token Remapping

Occasionally, you might need to remap tokens based on special cases. Consider the case of matching identifiers such as "abc", "python", or "guido". Certain identifiers such as "if", "else", and "while" might need to be treated as special keywords. To handle this, include token remapping rules when writing the lexer like this:

```
# calclex.py

from sly import Lexer

class CalcLexer(Lexer):
    tokens = { ID, IF, ELSE, WHILE }
    # String containing ignored characters (between tokens)
    ignore = ' \t'

    # Base ID rule
    ID = r'[a-zA-Z_][a-zA-Z0-9_]*'

    # Special cases
    ID['if'] = IF
    ID['else'] = ELSE
    ID['while'] = WHILE
```

When parsing an identifier, the special cases will remap certain matching values to a new token type. For example, if the value of an identifier is "if" above, an `IF` token will be generated.

Line numbers and position tracking

By default, lexers know nothing about line numbers. This is because they don't know anything about what constitutes a "line" of input (e.g., the newline character or even if the input is textual data). To update this information, you need to add a special rule for newlines. Promote the `ignore_newline` rule to a method like this:

```
# Define a rule so we can track line numbers
@_(r'\n+')
def ignore_newline(self, t):
    self.lineno += len(t.value)
```

Within the rule, the lineno attribute of the lexer is now updated. After the line number is updated, the token is discarded since nothing is returned.

Lexers do not perform and kind of automatic column tracking. However, it does record positional information related to each token in the token's `index` attribute. Using this, it is usually possible to compute column information as a separate step. For instance, you can search backwards until you reach the previous newline:

```
# Compute column.
#     input is the input text string
#     token is a token instance
def find_column(text, token):
    last_cr = text.rfind('\n', 0, token.index)
    if last_cr < 0:
        last_cr = 0
    column = (token.index - last_cr) + 1
    return column
```

Since column information is often only useful in the context of error handling, calculating the column position can be performed when needed as opposed to including it on each token.

Literal characters

Literal characters can be specified by defining a set `literals` in the class. For example:

```
class MyLexer(Lexer):
    ...
    literals = { '+','-','*','/' }
    ...
```

A literal character is a *single character* that is returned "as is" when encountered by the lexer. Literals are checked after all of the defined regular expression rules. Thus, if a rule starts with one of the literal characters, it will always take precedence.

When a literal token is returned, both its `type` and `value` attributes are set to the character itself. For example, `'+'`.

It's possible to write token methods that perform additional actions when literals are matched. However, you'll need to set the token type appropriately. For example:

```
class MyLexer(Lexer):

     literals = { '{', '}' }

     def __init__(self):
         self.nesting_level = 0

     @_(r'\{')
     def lbrace(self, t):
         t.type = '{'      # Set token type to the expected literal
         self.nesting_level += 1
         return t

     @_(r'\}')
     def rbrace(t):
         t.type = '}'      # Set token type to the expected literal
         self.nesting_level -=1
         return t
```

Error handling

If a bad character is encountered while lexing, tokenizing will stop. However, you can add an `error()` method to handle lexing errors that occur when illegal characters are detected. The error method receives a `Token` where the `value` attribute contains all remaining untokenized text. A typical handler might look at this text and skip ahead in some manner. For example:

```
class MyLexer(Lexer):
    ...
    # Error handling rule
    def error(self, t):
        print("Illegal character '%s'" % t.value[0])
        self.index += 1
```

In this case, we print the offending character and skip ahead one character by updating the lexer position. Error handling in a parser is often a hard problem. An error handler might scan ahead to a logical synchronization point such as a semicolon, a blank line, or similar landmark.

If the `error()` method also returns the passed token, it will show up as an `ERROR` token in the resulting token stream. This might be useful if the parser wants to see error tokens for some reason–perhaps for the purposes of improved error messages or some other kind of error handling.

other kind of error handling.

A More Complete Example

Here is a more complete example that puts many of these concepts into practice:

```
# calclex.py

from sly import Lexer

class CalcLexer(Lexer):
    # Set of token names.   This is always required
    tokens = { NUMBER, ID, WHILE, IF, ELSE, PRINT,
               PLUS, MINUS, TIMES, DIVIDE, ASSIGN,
               EQ, LT, LE, GT, GE, NE }


    literals = { '(', ')', '{', '}', ';' }

    # String containing ignored characters
    ignore = ' \t'

    # Regular expression rules for tokens
    PLUS    = r'\+'
    MINUS   = r'-'
    TIMES   = r'\*'
    DIVIDE  = r'/'
    EQ      = r'=='
    ASSIGN  = r'='
    LE      = r'<='
    LT      = r'<'
    GE      = r'>='
    GT      = r'>'
    NE      = r'!='

    @_(r'\d+')
    def NUMBER(self, t):
        t.value = int(t.value)
        return t

    # Identifiers and keywords
    ID = r'[a-zA-Z_][a-zA-Z0-9_]*'
    ID['if'] = IF
    ID['else'] = ELSE
    ID['while'] = WHILE
    ID['print'] = PRINT

    ignore_comment = r'\#.*'

    # Line number tracking
    @_(r'\n+')
    def ignore_newline(self, t):
        self.lineno += t.value.count('\n')

    def error(self, t):
        print('Line %d: Bad character %r' % (self.lineno, t.value[0]))
        self.index += 1

if __name__ == '__main__':
    data = '''
# Counting
x = 0;
while (x < 10) {
    print x:
    x = x + 1;
}
'''
    lexer = CalcLexer()
    for tok in lexer.tokenize(data):
        print(tok)
```

If you run this code, you'll get output that looks like this:

```
Token(type='ID', value='x', lineno=3, index=20)
Token(type='ASSIGN', value='=', lineno=3, index=22)
Token(type='NUMBER', value=0, lineno=3, index=24)
Token(type=';', value=';', lineno=3, index=25)
Token(type='WHILE', value='while', lineno=4, index=31)
Token(type='(', value='(', lineno=4, index=37)
Token(type='ID', value='x', lineno=4, index=38)
Token(type='LT', value='<', lineno=4, index=40)
Token(type='NUMBER', value=10, lineno=4, index=42)
Token(type=')', value=')', lineno=4, index=44)
Token(type='{', value='{', lineno=4, index=46)
Token(type='PRINT', value='print', lineno=5, index=56)
Token(type='ID', value='x', lineno=5, index=62)
Line 5: Bad character ':'
Token(type='ID', value='x', lineno=6, index=73)
Token(type='ASSIGN', value='=', lineno=6, index=75)
Token(type='ID', value='x', lineno=6, index=77)
Token(type='PLUS', value='+', lineno=6, index=79)
Token(type='NUMBER', value=1, lineno=6, index=81)
Token(type=';', value=';', lineno=6, index=82)
Token(type='}', value='}', lineno=7, index=88)
```

Study this example closely. It might take a bit to digest, but all of the essential parts of writing a lexer are there. Tokens have to be specified with regular expression rules. You

can optionally attach actions that execute when certain patterns are encountered. Certain features such as character literals are there mainly for convenience, saving you the trouble of writing separate regular expression rules. You can also add error handling.

Writing a Parser

The `Parser` class is used to parse language syntax. Before showing an example, there are a few important bits of background that must be covered.

Parsing Background

When writing a parser, *syntax* is usually specified in terms of a BNF grammar. For example, if you wanted to parse simple arithmetic expressions, you might first write an unambiguous grammar specification like this:

```
expr       : expr + term
           | expr - term
           | term

term       : term * factor
           | term / factor
           | factor

factor     : NUMBER
           | ( expr )
```

In the grammar, symbols such as `NUMBER`, `+`, `-`, `*`, and `/` are known as *terminals* and correspond to raw input tokens. Identifiers such as `term` and `factor` refer to grammar rules comprised of a collection of terminals and other rules. These identifiers are known as *non-terminals*. The separation of the grammar into different levels (e.g., `expr` and `term`) encodes the operator precedence rules for the different operations. In this case, multiplication and division have higher precedence than addition and subtraction.

The semantics of what happens during parsing is often specified using a technique known as syntax directed translation. In syntax directed translation, the symbols in the grammar become a kind of object. Values can be attached each symbol and operations carried out on those values when different grammar rules are recognized. For example, given the expression grammar above, you might write the specification for the operation of a simple calculator like this:

```
Grammar                  Action
-----------------------  --------------------------------
expr0   : expr1 + term   expr0.val = expr1.val + term.val
        | expr1 - term   expr0.val = expr1.val - term.val
        | term           expr0.val = term.val

term0   : term1 * factor term0.val = term1.val * factor.val
        | term1 / factor term0.val = term1.val / factor.val
        | factor         term0.val = factor.val

factor  : NUMBER         factor.val = int(NUMBER.val)
        | ( expr )       factor.val = expr.val
```

In this grammar, new values enter via the `NUMBER` token. Those values then propagate according to the actions described above. For example, `factor.val = int(NUMBER.val)` propagates the value from `NUMBER` to `factor`. `term0.val = factor.val` propagates the value from `factor` to `term`. Rules such as `expr0.val = expr1.val + term1.val` combine and propagate values further. Just to illustrate, here is how values propagate in the expression `2 + 3 * 4`:

```
NUMBER.val=2 + NUMBER.val=3 * NUMBER.val=4    # NUMBER -> factor
factor.val=2 + NUMBER.val=3 * NUMBER.val=4    # factor -> term
term.val=2 + NUMBER.val=3 * NUMBER.val=4      # term -> expr
expr.val=2 + NUMBER.val=3 * NUMBER.val=4      # NUMBER -> factor
expr.val=2 + factor.val=3 * NUMBER.val=4      # factor -> term
expr.val=2 + term.val=3 * NUMBER.val=4        # NUMBER -> factor
expr.val=2 + term.val=3 * factor.val=4        # term * factor -> term
expr.val=2 + term.val=12                      # expr + term -> expr
expr.val=14
```

SLY uses a parsing technique known as LR-parsing or shift-reduce parsing. LR parsing is a bottom up technique that tries to recognize the right-hand-side of various grammar rules. Whenever a valid right-hand-side is found in the input, the appropriate action method is triggered and the grammar symbols on right hand side are replaced by the grammar symbol on the left-hand-side.

LR parsing is commonly implemented by shifting grammar symbols onto a stack and looking at the stack and the next input token for patterns that match one of the grammar rules. The details of the algorithm can be found in a compiler textbook, but the following example illustrates the steps that are performed when parsing the expression `3 + 5 * (10 - 20)` using the grammar defined above. In the example, the special symbol `$` represents the end of input:

```
Step Symbol Stack           Input Tokens            Action
---- ---------------------  ---------------------   -----------------------
1                           3 + 5 * ( 10 - 20 )$    Shift 3
```

```
2    3                       + 5 * ( 10 - 20 )$    Reduce factor : NUMBER
3    factor                  + 5 * ( 10 - 20 )$    Reduce term   : factor
4    term                    + 5 * ( 10 - 20 )$    Reduce expr : term
5    expr                    + 5 * ( 10 - 20 )$    Shift +
6    expr +                    5 * ( 10 - 20 )$    Shift 5
7    expr + 5                    * ( 10 - 20 )$    Reduce factor : NUMBER
8    expr + factor               * ( 10 - 20 )$    Reduce term   : factor
9    expr + term                 * ( 10 - 20 )$    Shift *
10   expr + term *                 ( 10 - 20 )$    Shift (
11   expr + term * (                 10 - 20 )$    Shift 10
12   expr + term * ( 10                 - 20 )$    Reduce factor : NUMBER
13   expr + term * ( factor             - 20 )$    Reduce term : factor
14   expr + term * ( term               - 20 )$    Reduce expr : term
15   expr + term * ( expr               - 20 )$    Shift -
16   expr + term * ( expr -               20 )$    Shift 20
17   expr + term * ( expr - 20             )$    Reduce factor : NUMBER
18   expr + term * ( expr - factor         )$    Reduce term : factor
19   expr + term * ( expr - term           )$    Reduce expr : expr - te
20   expr + term * ( expr                  )$    Shift )
21   expr + term * ( expr )                 $    Reduce factor : (expr)
22   expr + term * factor                   $    Reduce term : term * fa
23   expr + term                            $    Reduce expr : expr + te
24   expr                                   $    Reduce expr
25                                          $    Success!
```

When parsing the expression, an underlying state machine and the current input token determine what happens next. If the next token looks like part of a valid grammar rule (based on other items on the stack), it is generally shifted onto the stack. If the top of the stack contains a valid right-hand-side of a grammar rule, it is usually "reduced" and the symbols replaced with the symbol on the left-hand-side. When this reduction occurs, the appropriate action is triggered (if defined). If the input token can't be shifted and the top of stack doesn't match any grammar rules, a syntax error has occurred and the parser must take some kind of recovery step (or bail out). A parse is only successful if the parser reaches a state where the symbol stack is empty and there are no more input tokens.

It is important to note that the underlying implementation is built around a large finite-state machine that is encoded in a collection of tables. The construction of these tables is non-trivial and beyond the scope of this discussion. However, subtle details of this process explain why, in the example above, the parser chooses to shift a token onto the stack in step 9 rather than reducing the rule `expr : expr + term`.

Parsing Example

Suppose you wanted to make a grammar for evaluating simple arithmetic expressions as previously described. Here is how you would do it with SLY:

```
from sly import Parser
from calclex import CalcLexer

class CalcParser(Parser):
    # Get the token list from the lexer (required)
    tokens = CalcLexer.tokens

    # Grammar rules and actions
    @_('expr PLUS term')
    def expr(self, p):
        return p.expr + p.term

    @_('expr MINUS term')
    def expr(self, p):
        return p.expr - p.term

    @_('term')
    def expr(self, p):
        return p.term

    @_('term TIMES factor')
    def term(self, p):
        return p.term * p.factor

    @_('term DIVIDE factor')
    def term(self, p):
        return p.term / p.factor

    @_('factor')
    def term(self, p):
        return p.factor

    @_('NUMBER')
    def factor(self, p):
        return p.NUMBER

    @_('LPAREN expr RPAREN')
    def factor(self, p):
        return p.expr

if __name__ == '__main__':
    lexer = CalcLexer()
    parser = CalcParser()

    while True:
        try:
            text = input('calc > ')
```

```
            result = parser.parse(lexer.tokenize(text))
            print(result)
        except EOFError:
            break
```

In this example, each grammar rule is defined by a method that's been decorated by `@_(rule)` decorator. The very first grammar rule defines the top of the parse (the first rule listed in a BNF grammar). The name of each method must match the name of the grammar rule being parsed. The argument to the `@_()` decorator is a string describing the right-hand-side of the grammar. Thus, a grammar rule like this:

```
expr : expr PLUS term
```

becomes a method like this:

```
@_('expr PLUS term')
def expr(self, p):
    ...
```

The method is triggered when that grammar rule is recognized on the input. As an argument, the method receives a sequence of grammar symbol values in `p`. There are two ways to access these symbols. First, you can use symbol names as shown:

```
@_('expr PLUS term')
def expr(self, p):
    return p.expr + p.term
```

Alternatively, you can also index `p` like an array:

```
@_('expr PLUS term')
def expr(self, p):
    return p[0] + p[2]
```

For tokens, the value of the corresponding `p.symbol` or `p[i]` is the *same* as the `p.value` attribute assigned to tokens in the lexer module. For non-terminals, the value is whatever was returned by the methods defined for that rule.

If a grammar rule includes the same symbol name more than once, you need to append a numeric suffix to disambiguate the symbol name when you're accessing values. For example:

```
@_('expr PLUS expr')
def expr(self, p):
    return p.expr0 + p.expr1
```

Finally, within each rule, you always return a value that becomes associated with that grammar symbol elsewhere. This is how values propagate within the grammar.

There are many other kinds of things that might happen in a rule though. For example, a rule might construct part of a parse tree instead:

```
@_('expr PLUS term')
def expr(self, p):
    return ('+', p.expr, p.term)
```

or it might create an instance related to an abstract syntax tree:

```
class BinOp(object):
    def __init__(self, op, left, right):
        self.op = op
        self.left = left
        self.right = right

@_('expr PLUS term')
def expr(self, p):
    return BinOp('+', p.expr, p.term)
```

The key thing is that the method returns the value that's going to be attached to the symbol "expr" in this case. This is the propagation of values that was described in the previous section.

Combining Grammar Rule Functions

When grammar rules are similar, they can be combined into a single method. For example, suppose you had two rules that were constructing a parse tree:

```
@_('expr PLUS term')
def expr(self, p):
    return ('+', p.expr, p.term)

@_('expr MINUS term')
def expr(self, p):
    return ('-', p.expr, p.term)
```

Instead of writing two functions, you might write a single function like this:

```
@_('expr PLUS term',
   'expr MINUS term')
```

```
    def expr(self, p):
        return (p[1], p.expr, p.term)
```

In this example, the operator could be `PLUS` or `MINUS`. Thus, we can't use the symbolic name to refer to its value. Instead, use the array index `p[1]` to get it as shown.

In general, the `@_()` decorator for any given method can list multiple grammar rules. When combining grammar rules into a single function though, all of the rules should have a similar structure (e.g., the same number of terms and consistent symbol names). Otherwise, the corresponding action code may end up being more complicated than necessary.

Character Literals

If desired, a grammar may contain tokens defined as single character literals. For example:

```
    @_('expr "+" term')
    def expr(self, p):
        return p.expr + p.term

    @_('expr "-" term')
    def expr(self, p):
        return p.expr - p.term
```

A character literal must be enclosed in quotes such as `"+"`. In addition, if literals are used, they must be declared in the corresponding lexer class through the use of a special `literals` declaration:

```
    class CalcLexer(Lexer):
        ...
        literals = { '+','-','*','/' }
        ...
```

Character literals are limited to a single character. Thus, it is not legal to specify literals such as `<=` or `==`. For this, use the normal lexing rules (e.g., define a rule such as `LE = r'<='`).

Empty Productions

If you need an empty production, define a special rule like this:

```
    @_('')
    def empty(self, p):
        pass
```

Now to use the empty production elsewhere, use the name 'empty' as a symbol. For example, suppose you need to encode a rule that involved an optional item like this:

```
    spam : optitem grok

    optitem : item
            | empty
```

You would encode the rules in SLY as follows:

```
    @_('optitem grok')
    def spam(self, p):
        ...

    @_('item')
    def optitem(self, p):
        ...

    @_('empty')
    def optitem(self, p):
        ...
```

Note: You could write empty rules anywhere by specifying an empty string. However,writing an "empty" rule and using "empty" to denote an empty production may be easier to read and more clearly state your intention.

Dealing With Ambiguous Grammars

The expression grammar given in the earlier example has been written in a special format to eliminate ambiguity. However, in many situations, it is extremely difficult or awkward to write grammars in this format. A much more natural way to express the grammar is in a more compact form like this:

```
    expr : expr PLUS expr
         | expr MINUS expr
         | expr TIMES expr
         | expr DIVIDE expr
         | LPAREN expr RPAREN
         | NUMBER
```

Unfortunately, this grammar specification is ambiguous. For example, if you are parsing

the string "3 * 4 + 5", there is no way to tell how the operators are supposed to be grouped. For example, does the expression mean "(3 * 4) + 5" or is it "3 * (4+5)"?

When an ambiguous grammar is given, you will get messages about "shift/reduce conflicts" or "reduce/reduce conflicts". A shift/reduce conflict is caused when the parser generator can't decide whether or not to reduce a rule or shift a symbol on the parsing stack. For example, consider the string "3 * 4 + 5" and the internal parsing stack:

```
Step Symbol Stack  Input Tokens       Action
---- ------------- ----------------   --------------------------------
1    $                    3 * 4 + 5$   Shift 3
2    $ 3                    * 4 + 5$   Reduce : expr : NUMBER
3    $ expr                 * 4 + 5$   Shift *
4    $ expr *                 4 + 5$   Shift 4
5    $ expr * 4                 + 5$   Reduce: expr : NUMBER
6    $ expr * expr              + 5$   SHIFT/REDUCE CONFLICT ????
```

In this case, when the parser reaches step 6, it has two options. One is to reduce the rule `expr : expr * expr` on the stack. The other option is to shift the token `+` on the stack. Both options are perfectly legal from the rules of the context-free-grammar.

By default, all shift/reduce conflicts are resolved in favor of shifting. Therefore, in the above example, the parser will always shift the `+` instead of reducing. Although this strategy works in many cases (for example, the case of "if-then" versus "if-then-else"), it is not enough for arithmetic expressions. In fact, in the above example, the decision to shift `+` is completely wrong—we should have reduced `expr * expr` since multiplication has higher mathematical precedence than addition.

To resolve ambiguity, especially in expression grammars, SLY allows individual tokens to be assigned a precedence level and associativity. This is done by adding a variable `precedence` to the parser class like this:

```
class CalcParser(Parser):
    ...
    precedence = (
       ('left', PLUS, MINUS),
       ('left', TIMES, DIVIDE),
    )

    # Rules where precedence is applied
    @_('expr PLUS expr')
    def expr(self, p):
        return p.expr0 + p.expr1

    @_('expr MINUS expr')
    def expr(self, p):
        return p.expr0 - p.expr1

    @_('expr TIMES expr')
    def expr(self, p):
        return p.expr0 * p.expr1

    @_('expr DIVIDE expr')
    def expr(self, p):
        return p.expr0 / p.expr1
    ...
```

This `precedence` declaration specifies that `PLUS`/`MINUS` have the same precedence level and are left-associative and that `TIMES`/`DIVIDE` have the same precedence and are left-associative. Within the `precedence` declaration, tokens are ordered from lowest to highest precedence. Thus, this declaration specifies that `TIMES`/`DIVIDE` have higher precedence than `PLUS`/`MINUS` (since they appear later in the precedence specification).

The precedence specification works by associating a numerical precedence level value and associativity direction to the listed tokens. For example, in the above example you get:

```
PLUS      : level = 1,  assoc = 'left'
MINUS     : level = 1,  assoc = 'left'
TIMES     : level = 2,  assoc = 'left'
DIVIDE    : level = 2,  assoc = 'left'
```

These values are then used to attach a numerical precedence value and associativity direction to each grammar rule. *This is always determined by looking at the precedence of the right-most terminal symbol.* For example:

```
expr : expr PLUS expr           # level = 1, left
     | expr MINUS expr          # level = 1, left
     | expr TIMES expr          # level = 2, left
     | expr DIVIDE expr         # level = 2, left
     | LPAREN expr RPAREN       # level = None (not specified)
     | NUMBER                   # level = None (not specified)
```

When shift/reduce conflicts are encountered, the parser generator resolves the conflict by looking at the precedence rules and associativity specifiers.

1. If the current token has higher precedence than the rule on the stack, it is shifted.
2. If the grammar rule on the stack has higher precedence, the rule is reduced.
3. If the current token and the grammar rule have the same precedence, the rule is reduced for left associativity, whereas the token is shifted for right associativity.

4. If nothing is known about the precedence, shift/reduce conflicts are resolved in favor of shifting (the default).

For example, if `expr PLUS expr` has been parsed and the next token is `TIMES`, the action is going to be a shift because `TIMES` has a higher precedence level than `PLUS`. On the other hand, if `expr TIMES expr` has been parsed and the next token is `PLUS`, the action is going to be reduce because `PLUS` has a lower precedence than `TIMES`.

When shift/reduce conflicts are resolved using the first three techniques (with the help of precedence rules), SLY will report no errors or conflicts in the grammar.

One problem with the precedence specifier technique is that it is sometimes necessary to change the precedence of an operator in certain contexts. For example, consider a unary-minus operator in `3 + 4 * -5`. Mathematically, the unary minus is normally given a very high precedence--being evaluated before the multiply. However, in our precedence specifier, `MINUS` has a lower precedence than `TIMES`. To deal with this, precedence rules can be given for so-called "fictitious tokens" like this:

```
class CalcParser(Parser):
    ...
    precedence = (
        ('left', PLUS, MINUS),
        ('left', TIMES, DIVIDE),
        ('right', UMINUS),            # Unary minus operator
    )
```

Now, in the grammar file, you write the unary minus rule like this:

```
@_('MINUS expr %prec UMINUS')
def expr(p):
    return -p.expr
```

In this case, `%prec UMINUS` overrides the default rule precedence--setting it to that of `UMINUS` in the precedence specifier.

At first, the use of `UMINUS` in this example may appear very confusing. `UMINUS` is not an input token or a grammar rule. Instead, you should think of it as the name of a special marker in the precedence table. When you use the `%prec` qualifier, you're telling SLY that you want the precedence of the expression to be the same as for this special marker instead of the usual precedence.

It is also possible to specify non-associativity in the `precedence` table. This is used when you *don't* want operations to chain together. For example, suppose you wanted to support comparison operators like `<` and `>` but you didn't want combinations like `a < b < c`. To do this, specify the precedence rules like this:

```
class MyParser(Parser):
    ...
    precedence = (
        ('nonassoc', LESSTHAN, GREATERTHAN),  # Nonassociative operators
        ('left', PLUS, MINUS),
        ('left', TIMES, DIVIDE),
        ('right', UMINUS),            # Unary minus operator
    )
```

If you do this, the occurrence of input text such as `a < b < c` will result in a syntax error. However, simple expressions such as `a < b` will still be fine.

Reduce/reduce conflicts are caused when there are multiple grammar rules that can be applied to a given set of symbols. This kind of conflict is almost always bad and is always resolved by picking the rule that appears first in the grammar file. Reduce/reduce conflicts are almost always caused when different sets of grammar rules somehow generate the same set of symbols. For example:

```
assignment :  ID EQUALS NUMBER
           |  ID EQUALS expr

expr       : expr PLUS expr
           | expr MINUS expr
           | expr TIMES expr
           | expr DIVIDE expr
           | LPAREN expr RPAREN
           | NUMBER
```

In this case, a reduce/reduce conflict exists between these two rules:

```
assignment  : ID EQUALS NUMBER
expr        : NUMBER
```

For example, if you're parsing `a = 5`, the parser can't figure out if this is supposed to be reduced as `assignment : ID EQUALS NUMBER` or whether it's supposed to reduce the 5 as an expression and then reduce the rule `assignment : ID EQUALS expr`.

It should be noted that reduce/reduce conflicts are notoriously difficult to spot simply looking at the input grammar. When a reduce/reduce conflict occurs, SLY will try to help by printing a warning message such as this:

```
WARNING: 1 reduce/reduce conflict
```

```
WARNING: reduce/reduce conflict in state 15 resolved using rule (assignment
WARNING: rejected rule (expression -> NUMBER)
```

This message identifies the two rules that are in conflict. However, it may not tell you how the parser arrived at such a state. To try and figure it out, you'll probably have to look at your grammar and the contents of the parser debugging file with an appropriately high level of caffeination (see the next section).

Parser Debugging

Tracking down shift/reduce and reduce/reduce conflicts is one of the finer pleasures of using an LR parsing algorithm. To assist in debugging, you can have SLY produce a debugging file when it constructs the parsing tables. Add a `debugfile` attribute to your class like this:

```
class CalcParser(Parser):
    debugfile = 'parser.out'
    ...
```

When present, this will write the entire grammar along with all parsing states to the file you specify. Each state of the parser is shown as output that looks something like this:

```
state 2

    (7) factor -> LPAREN . expr RPAREN
    (1) expr -> . term
    (2) expr -> . expr MINUS term
    (3) expr -> . expr PLUS term
    (4) term -> . factor
    (5) term -> . term DIVIDE factor
    (6) term -> . term TIMES factor
    (7) factor -> . LPAREN expr RPAREN
    (8) factor -> . NUMBER
    LPAREN          shift and go to state 2
    NUMBER          shift and go to state 3

    factor                         shift and go to state 1
    term                           shift and go to state 4
    expr                           shift and go to state 6
```

Each state keeps track of the grammar rules that might be in the process of being matched at that point. Within each rule, the "." character indicates the current location of the parse within that rule. In addition, the actions for each valid input token are listed. By looking at these rules (and with a little practice), you can usually track down the source of most parsing conflicts. It should also be stressed that not all shift-reduce conflicts are bad. However, the only way to be sure that they are resolved correctly is to look at the debugging file.

Syntax Error Handling

If you are creating a parser for production use, the handling of syntax errors is important. As a general rule, you don't want a parser to simply throw up its hands and stop at the first sign of trouble. Instead, you want it to report the error, recover if possible, and continue parsing so that all of the errors in the input get reported to the user at once. This is the standard behavior found in compilers for languages such as C, C++, and Java.

In SLY, when a syntax error occurs during parsing, the error is immediately detected (i.e., the parser does not read any more tokens beyond the source of the error). However, at this point, the parser enters a recovery mode that can be used to try and continue further parsing. As a general rule, error recovery in LR parsers is a delicate topic that involves ancient rituals and black-magic. The recovery mechanism provided by SLY is comparable to Unix yacc so you may want consult a book like O'Reilly's "Lex and Yacc" for some of the finer details.

When a syntax error occurs, SLY performs the following steps:

1. On the first occurrence of an error, the user-defined `error()` method is called with the offending token as an argument. However, if the syntax error is due to reaching the end-of-file, an argument of `None` is passed. Afterwards, the parser enters an "error-recovery" mode in which it will not make future calls to `error()` until it has successfully shifted at least 3 tokens onto the parsing stack.
2. If no recovery action is taken in `error()`, the offending lookahead token is replaced with a special `error` token.
3. If the offending lookahead token is already set to `error`, the top item of the parsing stack is deleted.
4. If the entire parsing stack is unwound, the parser enters a restart state and attempts to start parsing from its initial state.
5. If a grammar rule accepts `error` as a token, it will be shifted onto the parsing stack.
6. If the top item of the parsing stack is `error`, lookahead tokens will be discarded until the parser can successfully shift a new symbol or reduce a rule involving `error`.

Recovery and resynchronization with error rules

The most well-behaved approach for handling syntax errors is to write grammar rules

The most well-behaved approach for handling syntax errors is to write grammar rules that include the `error` token. For example, suppose your language had a grammar rule for a print statement like this:

```
@_('PRINT expr SEMI')
def statement(self, p):
    ...
```

To account for the possibility of a bad expression, you might write an additional grammar rule like this:

```
@_('PRINT error SEMI')
def statement(self, p):
    print("Syntax error in print statement. Bad expression")
```

In this case, the `error` token will match any sequence of tokens that might appear up to the first semicolon that is encountered. Once the semicolon is reached, the rule will be invoked and the `error` token will go away.

This type of recovery is sometimes known as parser resynchronization. The `error` token acts as a wildcard for any bad input text and the token immediately following `error` acts as a synchronization token.

It is important to note that the `error` token usually does not appear as the last token on the right in an error rule. For example:

```
@_('PRINT error')
def statement(self, p):
    print("Syntax error in print statement. Bad expression")
```

This is because the first bad token encountered will cause the rule to be reduced--which may make it difficult to recover if more bad tokens immediately follow. It's better to have some kind of landmark such as a semicolon, closing parentheses, or other token that can be used as a synchronization point.

Panic mode recovery

An alternative error recovery scheme is to enter a panic mode recovery in which tokens are discarded to a point where the parser might be able to recover in some sensible manner.

Panic mode recovery is implemented entirely in the `error()` function. For example, this function starts discarding tokens until it reaches a closing '}'. Then, it restarts the parser in its initial state:

```
def error(self, p):
    print("Whoa. You are seriously hosed.")
    if not p:
        print("End of File!")
        return

    # Read ahead looking for a closing '}'
    while True:
        tok = next(self.tokens, None)
        if not tok or tok.type == 'RBRACE':
            break
    self.restart()
```

This function discards the bad token and tells the parser that the error was ok:

```
def error(self, p):
    if p:
         print("Syntax error at token", p.type)
         # Just discard the token and tell the parser it's okay.
         self.errok()
     else:
         print("Syntax error at EOF")
```

A few additional details about some of the attributes and methods being used:

- `self.errok()`. This resets the parser state so it doesn't think it's in error-recovery mode. This will prevent an `error` token from being generated and will reset the internal error counters so that the next syntax error will call `error()` again.
- `self.tokens`. This is the iterable sequence of tokens being parsed. Calling `next(self.tokens)` will force it to advance by one token.
- `self.restart()`. This discards the entire parsing stack and resets the parser to its initial state.

To supply the next lookahead token to the parser, `error()` can return a token. This might be useful if trying to synchronize on special characters. For example:

```
def error(self, tok):
    # Read ahead looking for a terminating ";"
    while True:
        tok = next(self.tokens, None)           # Get the next token
        if not tok or tok.type == 'SEMI':
            break
        self.errok()
```

```
        # Return SEMI to the parser as the next lookahead token
        return tok
```

When Do Syntax Errors Get Reported?

In most cases, SLY will handle errors as soon as a bad input token is detected on the input. However, be aware that SLY may choose to delay error handling until after it has reduced one or more grammar rules first. This behavior might be unexpected, but it's related to special states in the underlying parsing table known as "defaulted states." A defaulted state is parsing condition where the same grammar rule will be reduced regardless of what valid token comes next on the input. For such states, SLY chooses to go ahead and reduce the grammar rule *without reading the next input token*. If the next token is bad, SLY will eventually get around to reading it and report a syntax error. It's just a little unusual in that you might see some of your grammar rules firing immediately prior to the syntax error.

General comments on error handling

For normal types of languages, error recovery with error rules and resynchronization characters is probably the most reliable technique. This is because you can instrument the grammar to catch errors at selected places where it is relatively easy to recover and continue parsing. Panic mode recovery is really only useful in certain specialized applications where you might want to discard huge portions of the input text to find a valid restart point.

Line Number and Position Tracking

Position tracking is often a tricky problem when writing compilers. By default, SLY tracks the line number and position of all tokens. The following attributes may be useful in a production rule:

- `p.lineno`. Line number of the left-most terminal in a production.
- `p.index`. Lexing index of the left-most terminal in a production.

For example:

```
@_('expr PLUS expr')
def expr(self, p):
    line   = p.lineno      # line number of the PLUS token
    index  = p.index       # Index of the PLUS token in input text
```

SLY doesn't propagate line number information to non-terminals. If you need this, you'll need to store line number information yourself and propagate it in AST nodes or some other data structure.

AST Construction

SLY provides no special functions for constructing an abstract syntax tree. However, such construction is easy enough to do on your own.

A minimal way to construct a tree is to create and propagate a tuple or list in each grammar rule function. There are many possible ways to do this, but one example is something like this:

```
@_('expr PLUS expr',
   'expr MINUS expr',
   'expr TIMES expr',
   'expr DIVIDE expr')
def expr(self, p):
    return ('binary-expression', p[1], p.expr0, p.expr1)

@_('LPAREN expr RPAREN')
def expr(self, p):
    return ('group-expression',p.expr])

@_('NUMBER')
def expr(self, p):
    return ('number-expression', p.NUMBER)
```

Another approach is to create a set of data structures for different kinds of abstract syntax tree nodes and create different node types in each rule:

```
class Expr:
    pass

class BinOp(Expr):
    def __init__(self, op, left, right)
        self.op = op
        self.left = left
        self.right = right

class Number(Expr):
    def __init__(self, value):
        self.value = value

@_('expr PLUS expr',
   'expr MINUS expr',
   'expr TIMES expr',
```

```
    'expr DIVIDE expr')
def expr(self, p):
    return BinOp(p[1], p.expr0, p.expr1)

@_('LPAREN expr RPAREN')
def expr(self, p):
    return p.expr

@_('NUMBER')
def expr(self, p):
    return Number(p.NUMBER)
```

The advantage to this approach is that it may make it easier to attach more complicated semantics, type checking, code generation, and other features to the node classes.

Changing the starting symbol

Normally, the first rule found in a parser class defines the starting grammar rule (top level rule). To change this, supply a `start` specifier in your class. For example:

```
class CalcParser(Parser):
    start = 'foo'

    @_('A B')
    def bar(self, p):
        ...

    @_('bar X')
    def foo(self, p):     # Parsing starts here (start symbol above)
        ...
```

The use of a `start` specifier may be useful during debugging since you can use it to work with a subset of a larger grammar.

Embedded Actions

The parsing technique used by SLY only allows actions to be executed at the end of a rule. For example, suppose you have a rule like this:

```
@_('A B C D')
def foo(self, p):
    print("Parsed a foo", p.A, p.B, p.C, p.D)
```

In this case, the supplied action code only executes after all of the symbols `A`, `B`, `C`, and `D` have been parsed. Sometimes, however, it is useful to execute small code fragments during intermediate stages of parsing. For example, suppose you wanted to perform some action immediately after `A` has been parsed. To do this, write an empty rule like this:

```
@_('A seen_A B C D')
def foo(self, p):
    print("Parsed a foo", p.A, p.B, p.C, p.D)
    print("seen_A returned", p.seen_A])

@_('')
def seen_A(self, p):
    print("Saw an A = ", p[-1])   # Access grammar symbol to the left
    return 'some_value'           # Assign value to seen_A
```

In this example, the empty `seen_A` rule executes immediately after `A` is shifted onto the parsing stack. Within this rule, `p[-1]` refers to the symbol on the stack that appears immediately to the left of the `seen_A` symbol. In this case, it would be the value of `A` in the `foo` rule immediately above. Like other rules, a value can be returned from an embedded action by returning it.

The use of embedded actions can sometimes introduce extra shift/reduce conflicts. For example, this grammar has no conflicts:

```
@_('abcd',
   'abcx')
def foo(self, p):
    pass

@_('A B C D')
def abcd(self, p):
    pass

@_('A B C X')
def abcx(self, p):
    pass
```

However, if you insert an embedded action into one of the rules like this:

```
@_('abcd',
   'abcx')
def foo(self, p):
    pass

@_('A B C D')
def abcd(self, p):
```

```
        pass

    @_('A B seen_AB C X')
    def abcx(self, p):
        pass

    @_('')
    def seen_AB(self, p):
        pass
```

an extra shift-reduce conflict will be introduced. This conflict is caused by the fact that the same symbol `C` appears next in both the `abcd` and `abcx` rules. The parser can either shift the symbol (`abcd` rule) or reduce the empty rule `seen_AB` (`abcx` rule).

A common use of embedded rules is to control other aspects of parsing such as scoping of local variables. For example, if you were parsing C code, you might write code like this:

```
    @_('LBRACE new_scope statements RBRACE')
    def statements(self, p):
```

```
    # Action code
    ...
    pop_scope()        # Return to previous scope

@_('')
def new_scope(self, p):
    # Create a new scope for local variables
    create_scope()
    ...
```

In this case, the embedded action `new_scope` executes immediately after a `LBRACE` ({) symbol is parsed. This might adjust internal symbol tables and other aspects of the parser. Upon completion of the rule `statements`, code undos the operations performed in the embedded action (e.g., `pop_scope()`).