Syntax analysis/parsing intro

Note: We learned earlier that a scanner produces each token as a pair -- a string and its token type. For simplicity, the parsers we write will ignore the token types and just use the strings, which we will call tokens all by themselves.

Software that performs *syntax analysis* for a grammar is called a *parser*.

The parser's job is to receive tokens from a scanner and either simply recognize whether the token sequence can be produced by a grammar or build a parse tree for that sequence of tokens.

A parser that builds and returns a parse tree based on a sequence of tokens is called a *parse tree generator*. A parser that only reports whether a sequence of tokens *can* be parsed is called a *recognizer*.

We will begin with recognizing token sequences rather than building trees.

On vocabulary: A compiler breaks a program into tokens and those tokens are often multiple characters long, like 'while' and '=='. But, to present the theory of parsing, we will use small, simple grammars which use single characters as their terminal symbols rather than multi-character tokens. Conceptually these are the same thing; symbols presented to the parser that aren't to be broken down any further. When talking about a grammar I tend to use the word "terminal" instead of "token" and they tend to be single lower-case characters. When talking about a parser, I tend to use the word "token" instead of "terminal", even if our token stream is just a sequence of characters. Because most of our examples deal with single-character tokens, I sometimes use the phrase "input character" instead of "token" or "terminal".

PDA Parsing Strategy

We saw that a context-free grammar (CFG) could be converted to an equivalent pushdown automata (PDA) mechanically. The PDA has three states s0, s1, s2, with s0 being the start state and s2 being an accept state. A transition from s0 to s1 is labelled (λ,∅,S∅) which puts the grammar's start symbol onto the PDA stack, and a transition from s1 to s2 is labelled (λ,∅,∅) to let the PDA get to an accept state only if the stack is empty. There are then transition arrows from s1 to s1 with a triple of the form (x,x,λ) for each possible terminal x and a triple of the form (λ,A,ω) for each production in the grammar A→ω (ω is pronounced "omega" because it's that lower-case Greek letter and is used here to represent whatever is on the right-hand side of the production).

For example the following grammar contains three terminal symbols (a, b, and x) and three productions (S→aSa, S→bSb, and S→x):

S → aSa | bSb | x

The s1 to s1 transitions would thus be labelled (a,a,λ), (b,b,λ), (x,x,λ), (λ,S,aSa), (λ,S,bSb), and (λ,S,x).

Every string in the language that S generates (often abbreviated as L(S)) has a path to s2 that consumes the entire input and every string not in L(S) does not have such a path.

The behavior of this PDA can be simulated in computer code. The code at the end of this page uses a Python list as a stack (Python uses "append" instead of "push" when using a list as a stack) and manipulates it exactly as the PDA would. The code throws an exception if an error occurs during parsing. If the input is in the language of the grammar then the function completes without error.

Predictive parsing

You'll notice in the parsing code that when S is on top of the stack we use the next input token to decide which production to use next. This is called *predicitve parsing* because we are using the next input token to predict which of the production choices is the right one to use.

We can use predictive parsing here because there is no overlap in which first token each choice of production can evolve into. All strings with a derivation beginning S → aSa will begin with an 'a'. All strings with a derivation beginning S → bSb will begin with a 'b'. All strings with a derivation beginning S → x will begin with an 'x'. Since the first characters each production can evolve into do not overlap, we know that if S is on the top of the stack we can replace it with aSa if the next token is 'a', bSb if the next token is 'b', and x if the next token is 'x'. If the next token is none of these three, then there is no production that will work and the string is not parsable.

This simple predictive parsing strategy is often enough to parse an input, but some situations don't allow it.

- Left recursive. If the grammar has any rules which would cause an infinite parsing loop, then this method won't work. For example S → Sa can't be used because replacing S with Sa repeatedly would cause an infinite number of a's to be pushed on the stack.
- Ambiguous first token. If more than one production can begin with the same token, then predictive parsing cannot be applied. For example, with the productions S → aAb | aBc we cannot predict which to use if the next token is 'a'.

If the grammar is not left recursive and doesn't have ambiguous first tokens then the grammar is suitable for this method. If a grammar is not suitable, sometimes it can be rewritten to be suitable. And, if not, there are more powerful parsing methods that may do the trick, but this class does not cover them.

Sample Code

Here is sample code for recognizing whether an input sequence is generated by the context-free grammar S → aSa | bSb | x. This code also includes a simple class to emulate a scanner's match and next functions.

```
class scanner:
    # toks[i] must evaluate to the i-th token in the token stream.
    # Assumes toks does not change during parsing
    def __init__(self,toks):
        self._toks = toks
        self._i = 0

    # If no more tokens exist or current token isn't s, raise exception.
    # Otherwise pass over the current one and move to the next.
    def match(self,s):
        if (self._i < len(self._toks)) and (self._toks[self._i] == s):
            self._i += 1
        else:
            raise Exception

    # If any tokens remain return the current one. If no more, return None.
    def next(self):
        if self._i < len(self._toks):
            return self._toks[self._i]
        else:
```

```
            return None


# Input can be any type where len(input) is defined and input[i] yields a
# string (ie, string, list, etc). Raises Exception on a parse error.
def parse(input):
    toks = scanner(input)
    stack = ['S']
    while len(stack) > 0:
        top = stack.pop()      # Always pop top of stack
        tok = toks.next()      # None indicates token stream empty
        if tok == None:        # Running out of tokens is error in this grammar
            raise Exception
        elif top in ('a', 'b', 'x'):     # Matching stack top to token
            toks.match(top)
        elif top == 'S' and tok == 'a':  # S -> aSa must be the next
            stack.append('a')            # production to follow here
            stack.append('S')
            stack.append('a')
        elif top == 'S' and tok == 'b':  # S -> bSb must be the next
            stack.append('b')            # production to follow here
            stack.append('S')
            stack.append('b')
        elif top == 'S' and tok == 'x':  # S -> x must be the next
            stack.append('x')            # production to follow here
        else:
            raise Exception    # Unrecognized top/tok combination
    if toks.next() != None:
        raise Exception

try:
    parse("aabxbaa")
except:
    print("Reject")
else:
    print("Accept")
```