Grammar for simple calculator:

```
<program>    ::=   <stmt_list>  $$
<stmt_list>  ::=   <stmt> <stmt_list> | ε
<program>    ::=   <stmt_list>  $$
<stmt_list>  ::=   <stmt> <stmt_list> | ε
<stmt>       ::=    id := <expr> | read id | write <expr>
<expr>       ::=   <term> <term_tail>
<term_tail>  ::=   <add_op> <term> | ε
<term>       ::=    id | number
<add_op>     ::=   + | −

FIRST sets:
<program>    → {$$, id, read, write}
<stmt_list>  → {ε, id, read, write}
<stmt>       → {id, read, write}
<expr>       → {id,number}
<term_tail>  → {ε, +, −}
<term>       → {id,number}
<add_op>     → {+, −}

FOLLOW sets:
<program>    → ε
<stmt_list>  → {$$}
<stmt>       → {id,read,write,$$}
<expr>       → {id,read,write,$$}
<term_tail>  → {id,read,write,$$}
<term>       → {+,−,id,read,write,$$}
<add_op>     → {id,number}
$$           → {ε}
read         → {id}
write        → {id,number}
id           → {:=,id,read,write,$$,+,−}
+            → {id,number}
−            → {id,number}
:=           → {id,number}
number       → {+,−,id,read,write,$$}

PREDICT sets:
<program>    ::= <stmt_list>  $$      → {id,read,write,$$}
<stmt_list>  ::= <stmt> <stmt_list>   → {id,read,write}
<stmt_list>  ::= ε                    → {$$}
<stmt>       ::=  id := <expr>        → {id}
<stmt>       ::=  read id             → {read}
<stmt>       ::=  write <expr>        → {write}
<expr>       ::= <term> <term_tail>   → {id,number}
<term_tail>  ::= <add_op> <term>      → {+,−}
<term_tail>  ::= ε                    → {id,read,write,$$}
<term>       ::= id                   → {id}
<term>       ::= number               → {number}
<add_op>     ::= +                    → {+}
<add_op>     ::= −                    → {−}
```

```python
"""Correctness parser for calculator grammar. Note that this program
commits a massive style faux pas: the variables token and tokens are
global variables. In other words, they are accessible from anywhere in
the program. I chose to do this because this any attempt to do this
'right' results in a bunch of parameters getting passed around that
obscures the idea that this demonstration program is trying to
convey. Sometimes 'proper' isn't always 'simplest.'"""

import sys

def tokenize(lexemes):
    global tokens
    # Make a copy of lexemes (don't just point tokens at it)
    tokens = list(lexemes)
    for i in range(len(tokens)):
        if tokens[i] not in ['read','write','num',None] and \
                tokens[i].isalnum() and not tokens[i].isdigit():
            tokens[i] = 'id'
        elif tokens[i].isdigit():
            tokens[i] = 'number'

def nextToken():
    """Grab the next token. Set the current token to None if there are
    no more tokens."""
    global token
    #print 'Matched',token
    if len(tokens)>0:
        token = tokens.pop(0)
    else:
        token = None

def match(expected):
    """Match the current token against the expected value. If
    successful, grab the next token. Set the current token to None if
    there are no more tokens."""
    if token==expected:
        nextToken()
    else:
        raise Exception('Parse error',token,expected)

""" Following from here are the functions that correspond to the BNF
grammar. Each non-terminal gets a function. The 'if' statement has a
series of cases, one for each BNF rule with that non-terminal on the
left-hand side. The conditions of the 'if' statement for each case
check if the next token is in the PREDICT set for that rule."""

def program():
    if token in ['id','read','write','$$']:
        stmt_list()
        match('$$')
    else:
        raise Exception('Parse error')

def stmt_list():
    if token in ['id','read','write']:
        stmt()
        stmt_list()
    elif token=='$$':
        pass
    else:
        raise Exception('Parse error')
```

```

def stmt():
    if token=='id':
        match('id')
        match(':=')
        expr()
    elif token=='read':
        match('read')
        match('id')
    elif token=='write':
        match('write')
        expr()
    else:
        raise Exception('Parse error')

def expr():
    if token in ['id','number']:
        term()
        term_tail()
    else:
        raise Exception('Parse error')

def term_tail():
    if token in ['+','-']:
        add_op()
        term()
        term_tail()
    elif token in ['id','read','write','$$']:
        pass
    else:
        raise Exception('Parse error')

def term():
    if token=='id':
        match('id')
    elif token=='number':
        match('number')
    else:
        raise Exception('Parse error')

def add_op():
    if token=='+':
        match('+')
    elif token=='-':
        match('-')
    else:
        raise Exception('Parse error')

"""Open up the file, grab the program, and parse it."""

file = open(sys.argv[1],'r')
data = file.read()
file.close()
lexemes = data.split()
tokenize(lexemes)
token = tokens.pop(0)

program()
print "Success"
```