ECS 140A Homework 3 – Problem 2

1 Python

Step 1: Algorithm/Pseudocode

```
class SimpleParser:
    init(s):
        store string input
        store current character position in string input

    fun_s():
        call get_next_char()
        if char is 'a', handle repeats of 'a'
            check 1 char ahead without moving position
            if peek char is 'a' then recurse
            else call fun_x() because no more 'a's
        else if char is 'b', call fun_x()
        else not 'a' or 'b', call fun_x()

        if fun_x() returned true, print valid string
        else print error

    fun_x():
        call get_next_char()
        if char is 'c' or 'd', try to get next char
            if index error raised, return true because no more char left
        return false if not c or d, or no error raised

    get_next_char():
        increment current char position
        return char in string input at new position

    peek_next_char():
        return char in string input 1 ahead of current position without moving
```

Step 2: Actual Code

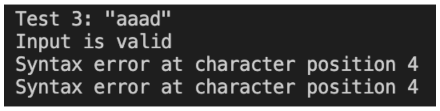
```python
class SimpleParser:
    def __init__(self, str_in):
        self.string = str_in
        self.char_pos = -1

    def fun_s(self):
        is_success = False
        ch = self.get_next_char()
        if ch == 'a': # handle repeats of 'a'
            if ch == self.peek_next_char():
                self.fun_s()
            else:
                is_success = self.fun_x()
        elif ch == 'b':
            is_success = self.fun_x()
        else:
            is_success = self.fun_x()

        if is_success:
            print("Input is valid")
        else:
            print(f"Syntax error at character position {self.char_pos}")

    def fun_x(self):
        ch = self.get_next_char()
        if ch == 'c' or ch == 'd':
            try:
                self.get_next_char()
            except IndexError:
                return True
        return False

    def get_next_char(self):
        self.char_pos += 1
        return self.string[self.char_pos]

    def peek_next_char(self):
        return self.string[self.char_pos + 1]
```

Step 3: Working Code

There were no syntax errors, so the initial working code was the same as the previous step.

Step 4: Debug Process

Bug 1



```
Test 3: "aaad"
Input is valid
Syntax error at character position 4
Syntax error at character position 4
```

This bug occurred because I did not account for additional print statements after returning from the recursive call of `fun_s()`.

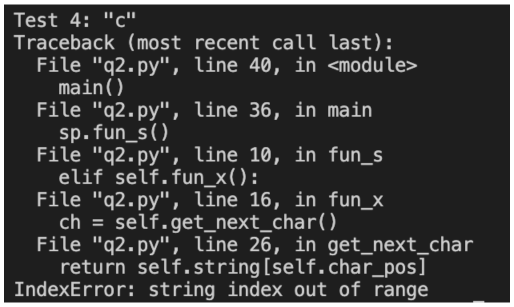
Buggy code:

```python
def fun_s(self):
    is_success = False
    ch = self.get_next_char()
    if ch == 'a': # handle repeats of 'a'
        if ch == self.peek_next_char():
            self.fun_s()
        else:
            is_success = self.fun_x()
    elif ch == 'b':
        is_success = self.fun_x()
    else:
        is_success = self.fun_x()

    if is_success:
        print("Input is valid")
    else:
        print(f"Syntax error at character position {self.char_pos}")
```

Fixed code:

```python
def fun_s(self):
    ch = self.get_next_char()
    if ch == 'a' and ch == self.peek_next_char():
        self.fun_s()
    elif self.fun_x():
        print("Input is valid")
    else:
        print(f"Syntax error at character position {self.char_pos}")
```

Bug 2



```
Test 4: "c"
Traceback (most recent call last):
  File "q2.py", line 40, in <module>
    main()
  File "q2.py", line 36, in main
    sp.fun_s()
  File "q2.py", line 10, in fun_s
    elif self.fun_x():
  File "q2.py", line 16, in fun_x
    ch = self.get_next_char()
  File "q2.py", line 26, in get_next_char
    return self.string[self.char_pos]
IndexError: string index out of range
```

This bug was raised because I did not account for having no repeats of the character `a`. As a result, we got character `c` in `fun_s()` even though it should have been parsed by `fun_x()`. To fix this, if we have not begun parsing the string, we will peek the first character to see if it's `a` or `b`. If it is, we can get the next character normally. Otherwise, we will need to make sure we call `fun_x()` without moving positions.

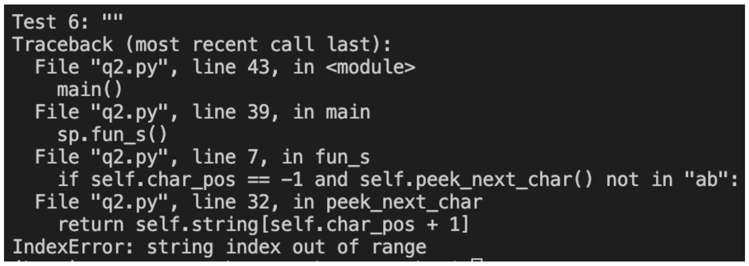
Buggy code:

```python
def fun_s(self):
    ch = self.get_next_char()
    if ch == 'a' and ch == self.peek_next_char():
        self.fun_s()
    elif self.fun_x():
        print("Input is valid")
    else:
        print(f"Syntax error at character position {self.char_pos}")
```

Fixed code:

```python
def fun_s(self):
    if self.char_pos == -1 and self.peek_next_char() not in "ab":
        ch = self.peek_next_char()
    else:
        ch = self.get_next_char()

    if ch == 'a' and ch == self.peek_next_char():
        self.fun_s()
    elif self.fun_x():
        print("Input is valid")
    else:
        print(f"Syntax error at character position {self.char_pos}")
```

Bug 3



```
Test 6: ""
Traceback (most recent call last):
  File "q2.py", line 43, in <module>
    main()
  File "q2.py", line 39, in main
    sp.fun_s()
  File "q2.py", line 7, in fun_s
    if self.char_pos == -1 and self.peek_next_char() not in "ab":
  File "q2.py", line 32, in peek_next_char
    return self.string[self.char_pos + 1]
IndexError: string index out of range
```

This bug was raised because I did not consider having an empty string as input. When peeking the very first character at the beginning of the parsing process in `fun_s()`, it does not account for having nothing to read. To handle having no characters to read or peek, I return `None` instead so I can check if nothing was read easily and handle this case accordingly.

Buggy code:

```python
def fun_s(self):
    if self.char_pos == -1 and self.peek_next_char() not in "ab":
        ch = self.peek_next_char()
    else:
        ch = self.get_next_char()
    if ch == 'a' and ch == self.peek_next_char():
        self.fun_s()
    elif self.fun_x():
        print("Input is valid")
    else:
        print(f"Syntax error at character position {self.char_pos}")

def fun_x(self):
    ch = self.get_next_char()
    if ch == 'c' or ch == 'd':
        try:
            self.get_next_char()
        except IndexError:
            return True
    return False

def get_next_char(self):
    self.char_pos += 1
    return self.string[self.char_pos]

def peek_next_char(self):
    return self.string[self.char_pos + 1]
```

Fixed code:

```python
def fun_s(self):
    if self.peek_next_char() == None:
        ch = None
    elif self.char_pos == -1 and self.peek_next_char() not in "ab":
        ch = self.peek_next_char()
    else:
        ch = self.get_next_char()

    if ch == 'a' and ch == self.peek_next_char():
        self.fun_s()
    elif self.fun_x():
        print("Input is valid")
    else:
        print(f"Syntax error at character position {self.char_pos}")

def fun_x(self):
    ch = self.get_next_char()
    if (ch == 'c' or ch == 'd') and self.get_next_char() == None:
        return True
    return False

def get_next_char(self):
    try:
        self.char_pos += 1
        return self.string[self.char_pos]
    except IndexError:
        return None

def peek_next_char(self):
    try:
        return self.string[self.char_pos + 1]
    except IndexError:
        return None
```

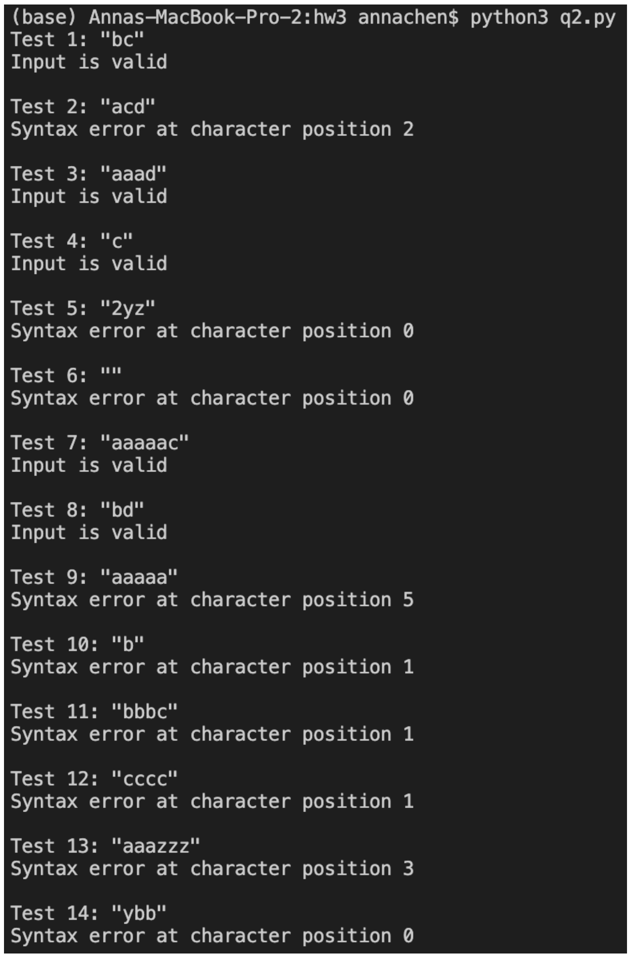
Step 5: Add Documentation

Note: I have modified my code after the debugging stage to use `try-except` for printing error messages when the input string is invalid. I ran the code again with all given and additional test cases passed.

```python
class SimpleParser:
    def __init__(self, str_in):
        '''' Initializer '''
        self.string = str_in # string to parse
        self.char_pos = -1 # current char position in string

    def fun_s(self):
        '''
        Handles grammar rule for S
        Prints message according to if grammar rule is satisfied
        '''
        if self.peek_next_char() == None: # handle if no characters left to read
            ch = None
        # don't move position if string[0] is not 'a' or 'b' and let fun_x() handle
        elif self.char_pos == -1 and self.peek_next_char() not in "ab":
            ch = self.peek_next_char()
        else: # if char is 'a' or 'b'
            ch = self.get_next_char()

        try:
            if ch == 'a' and ch == self.peek_next_char(): # handle repeated 'a' chars
                self.fun_s()
            else:
                self.fun_x()
                print("Input is valid")
        except:
            print(f"Syntax error at character position {self.char_pos}")

    def fun_x(self):
        '''
        Handles grammar rule for X
        Raises an exception if grammar rule is not satisfied
        '''
        ch = self.get_next_char()
        if not ((ch == 'c' or ch == 'd') and self.get_next_char() == None):
            raise

    def get_next_char(self):
        '''
        Moves forward 1 position and returns character at new position
        Returns None if out of bounds
        '''
        try:
            self.char_pos += 1
            return self.string[self.char_pos]
        except IndexError:
            return None

    def peek_next_char(self):
        ''' Returns character 1 position ahead. Returns None if out of bounds '''
        try:
            return self.string[self.char_pos + 1]
        except IndexError:
            return None
```

Step 6: Extra Test Cases Used

- `"aaaaac"` (valid)
- `"bd"` (valid)
- `"aaaaa"` (invalid at 5)
- `"b"` (invalid at 1)
- `"bbbc"` (invalid at 1)
- `"cccc"` (invalid at 1)
- `"aaazzz"` (invalid at 3)
- `"ybb"` (invalid at 0)



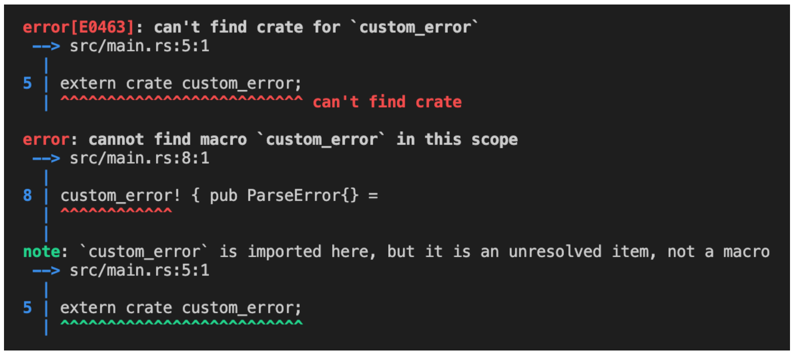
```
(base) Annas-MacBook-Pro-2:hw3 annachen$ python3 q2.py
Test 1: "bc"
Input is valid

Test 2: "acd"
Syntax error at character position 2

Test 3: "aaad"
Input is valid

Test 4: "c"
Input is valid

Test 5: "2yz"
Syntax error at character position 0

Test 6: ""
Syntax error at character position 0

Test 7: "aaaaac"
Input is valid

Test 8: "bd"
Input is valid

Test 9: "aaaaa"
Syntax error at character position 5

Test 10: "b"
Syntax error at character position 1

Test 11: "bbbc"
Syntax error at character position 1

Test 12: "cccc"
Syntax error at character position 1

Test 13: "aaazzz"
Syntax error at character position 3

Test 14: "ybb"
Syntax error at character position 0
```

2 Rust

Step 2: Actual Code

```rust
// Sources:
// https://crates.io/crates/custom_error
// https://docs.rs/custom_error/latest/custom_error/macro.custom_error.html

extern crate custom_error;
use custom_error::custom_error;

custom_error! { ParseError
    Syntax{pos: i32} = "Syntax error at character position {pos}"
}

struct SimpleParser {
    str_in: String,
    char_pos: i32,
}

// Assuming string input has no whitespace
// No null value in Rust; so using ' ' if nothing left in string to parse
impl SimpleParser {
    fn new(s: String) -> SimpleParser {
        return SimpleParser {
            str_in: s,
            char_pos: -1
        };
    }

    fn fun_s(&mut self) {
        let mut ch: char;
        if self.char_pos == -1 && self.peek_next_char() != 'a'
               && self.peek_next_char() != 'b' {
            ch = self.peek_next_char();
        } else {
            ch = self.get_next_char();
        }

        if ch == 'a' && ch == self.peek_next_char() {
            self.fun_s();
        } else {
            match self.fun_x() {
                Ok() => println!("Input is valid"),
                Err(e) => println!("{}", e)
            }
        }
    }
    fn fun_x(&mut self) -> Result<(), ParseError> {
        let ch = self.get_next_char()?;
        if (ch == 'c' || ch == 'd') && self.get_next_char() == ' ' {
            return Ok(); // return success
        }
        return Err(ParseError::Syntax{pos: self.char_pos}.to_string()); // return fail
    }

    fn get_next_char(&mut self) -> char {
        self.char_pos += 1;
        if self.char_pos <= self.str_in.len() {
            return self.str_in.chars().nth(self.char_pos as usize).unwrap();
        }
        return ' ';
    }

    fn peek_next_char(&self) -> char {
        if self.char_pos + 1 <= self.str_in.len() {
            return self.str_in.chars().nth((self.char_pos + 1) as usize).unwrap();
        }
        return ' ';
    }
}
```

Syntax Error 1



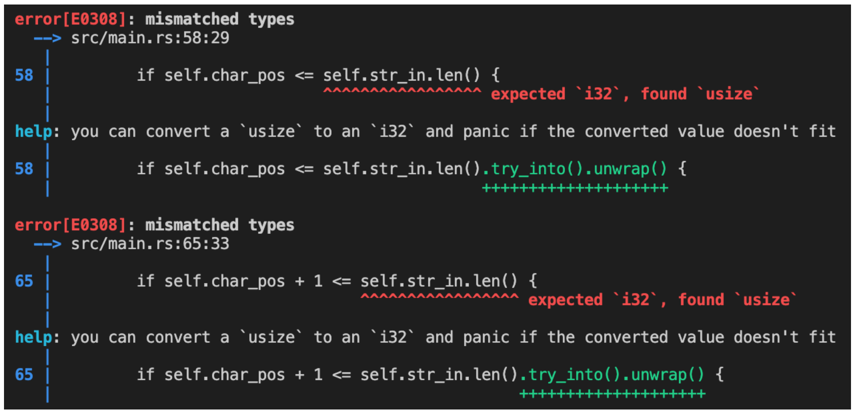
```
error[E0463]: can't find crate for `custom_error`
 --> src/main.rs:5:1
  |
5 | extern crate custom_error;
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^ can't find crate

error: cannot find macro `custom_error` in this scope
 --> src/main.rs:8:1
  |
8 | custom_error! { pub ParseError{} =
  | ^^^^^^^^^^^^
  |
note: `custom_error` is imported here, but it is an unresolved item, not a macro
 --> src/main.rs:5:1
  |
5 | extern crate custom_error;
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^
```

This error was raised because I forgot to add the dependency in the `.toml` file. Before, there were no dependencies listed.

Fixed dependency:

```
...
[dependencies]
custom_error = "1.9.2"
```

Syntax Error 2



```
error[E0308]: mismatched types
  --> src/main.rs:58:29
   |
58 |         if self.char_pos <= self.str_in.len() {
   |                             ^^^^^^^^^^^^^^^^^ expected `i32`, found `usize`
   |
help: you can convert a `usize` to an `i32` and panic if the converted value doesn't fit
   |
58 |         if self.char_pos <= self.str_in.len().try_into().unwrap() {
   |                                              ++++++++++++++++++++

error[E0308]: mismatched types
  --> src/main.rs:65:33
   |
65 |         if self.char_pos + 1 <= self.str_in.len() {
   |                                 ^^^^^^^^^^^^^^^^^ expected `i32`, found `usize`
   |
help: you can convert a `usize` to an `i32` and panic if the converted value doesn't fit
   |
65 |         if self.char_pos + 1 <= self.str_in.len().try_into().unwrap() {
   |                                                  ++++++++++++++++++++
```

This error was raised because I did not match types in the comparison.

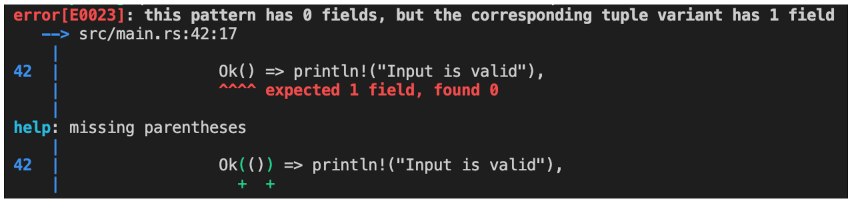
Code with syntax errors:

```
fn get_next_char(&mut self) -> char {
    self.char_pos += 1;
    if self.char_pos <= self.str_in.len() {
    ...
}

fn peek_next_char(&self) -> char {
    if self.char_pos + 1 <= self.str_in.len() {
    ...
}
```

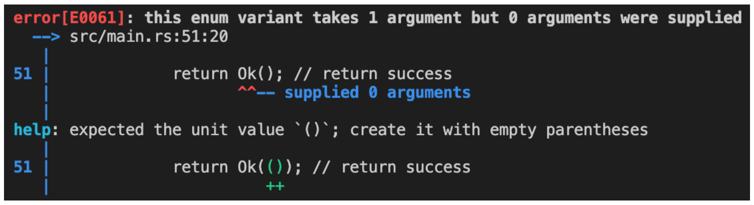
Fixed code:

```
fn get_next_char(&mut self) -> char {
    self.char_pos += 1;
    if self.char_pos <= self.str_in.len() as i32 {
    ...
}

fn peek_next_char(&self) -> char {
    if self.char_pos + 1 <= self.str_in.len() as i32 {
    ...
}
```

Syntax Error 3



```
error[E0023]: this pattern has 0 fields, but the corresponding tuple variant has 1 field
  --> src/main.rs:42:17
   |
42 |                 Ok() => println!("Input is valid"),
   |                 ^^^^ expected 1 field, found 0
   |
help: missing parentheses
   |
42 |                 Ok(()) => println!("Input is valid"),
   |                   +  +
```



```
error[E0061]: this enum variant takes 1 argument but 0 arguments were supplied
  --> src/main.rs:51:20
   |
51 |             return Ok(); // return success
   |                    ^^-- supplied 0 arguments
   |
help: expected the unit value `()`; create it with empty parentheses
   |
51 |             return Ok(()); // return success
   |                       ++
```

These errors were raised because `Ok()` expects an argument but I did not provide one. Because upon success, I don't expect anything specific to be returned, I will use the unit value `()` as a fix.

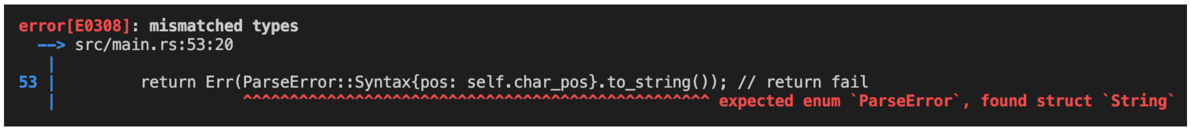
Code with syntax errors:

```
fn fun_s(&mut self) {
...
        match self.fun_x() {
            Ok() => println!("Input is valid"),
            Err(e) => println!("{}", e)
        }
...
fn fun_x(&mut self) -> Result<(), ParseError> {
    let ch = self.get_next_char()?;
    if (ch == 'c' || ch == 'd') && self.get_next_char() == ' ' {
        return Ok(); // return success
    }
    return Err(ParseError::Syntax{pos: self.char_pos}.to_string()); // return fail
}
```

Fixed code:

```
fn fun_s(&mut self) {
...
        match self.fun_x() {
            Ok(()) => println!("Input is valid"),
            Err(e) => println!("{}", e)
        }
...
fn fun_x(&mut self) -> Result<(), ParseError> {
    let ch = self.get_next_char()?;
    if (ch == 'c' || ch == 'd') && self.get_next_char() == ' ' {
        return Ok(()); // return success
    }
    return Err(ParseError::Syntax{pos: self.char_pos}.to_string()); // return fail
}
```

Syntax Error 4



```
error[E0308]: mismatched types
  --> src/main.rs:53:20
   |
53 |         return Err(ParseError::Syntax{pos: self.char_pos}.to_string()); // return fail
   |                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected enum `ParseError`, found struct `String`
```

This error was raised because `ParseError` was expected but I had transformed the value into `String` instead. To fix, I converted the value to `String` when printing the error instead.

Code with syntax errors:

```
fn fun_s(&mut self) {
    ...
        match self.fun_x() {
            Ok(()) => println!("Input is valid"),
            Err(e) => println!("{}", e)
    ...
}

fn fun_x(&mut self) -> Result<(), ParseError> {
    ...
    return Err(ParseError::Syntax{pos: self.char_pos}.to_string()); // return fail
}
```

Fixed code:

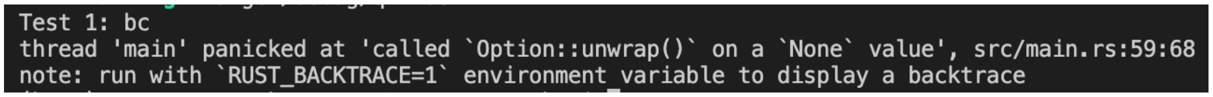
```
fn fun_s(&mut self) {
    ...
        match self.fun_x() {
            Ok(()) => println!("Input is valid"),
            Err(e) => println!("{}", e.to_string())
    ...
}

fn fun_x(&mut self) -> Result<(), ParseError> {
    ...
    return Err(ParseError::Syntax{pos: self.char_pos}); // return fail
}
```

Step 3: Working Code

```rust
// Sources:
// https://crates.io/crates/custom_error
// https://docs.rs/custom_error/latest/custom_error/macro.custom_error.html

extern crate custom_error;
use custom_error::custom_error;

custom_error! { ParseError
    Syntax{pos: i32} = "Syntax error at character position {pos}"
}

struct SimpleParser {
    str_in: String,
    char_pos: i32,
}

// Assuming string input has no whitespace
// No null value in Rust; so using ' ' if nothing left in string to parse
impl SimpleParser {
    fn new(s: String) -> SimpleParser {
        return SimpleParser {
            str_in: s,
            char_pos: -1
        };
    }

    fn fun_s(&mut self) {
        let ch: char;
        if self.char_pos == -1 && self.peek_next_char() != 'a'
              && self.peek_next_char() != 'b' {
            ch = self.peek_next_char();
        } else {
            ch = self.get_next_char();
        }

        if ch == 'a' && ch == self.peek_next_char() {
            self.fun_s();
        } else {
            match self.fun_x() {
                Ok(()) => println!("Input is valid"),
                Err(e) => println!("{}", e.to_string())
            }
        }
    }
    fn fun_x(&mut self) -> Result<(), ParseError> {
        let ch = self.get_next_char();
        if (ch == 'c' || ch == 'd') && self.get_next_char() == ' ' {
            return Ok(()); // return success
        }
        return Err(ParseError::Syntax{pos: self.char_pos}); // return fail
    }

    fn get_next_char(&mut self) -> char {
        self.char_pos += 1;
        if self.char_pos <= self.str_in.len() as i32 {
            return self.str_in.chars().nth(self.char_pos as usize).unwrap();
        }
        return ' ';
    }

    fn peek_next_char(&self) -> char {
        if self.char_pos + 1 <= self.str_in.len() as i32 {
            return self.str_in.chars().nth((self.char_pos + 1) as usize).unwrap();
        }
        return ' ';
    }
}
```

Step 4: Debug Process

Bug 1



```
Test 1: bc
thread 'main' panicked at 'called `Option::unwrap()` on a `None` value', src/main.rs:59:68
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

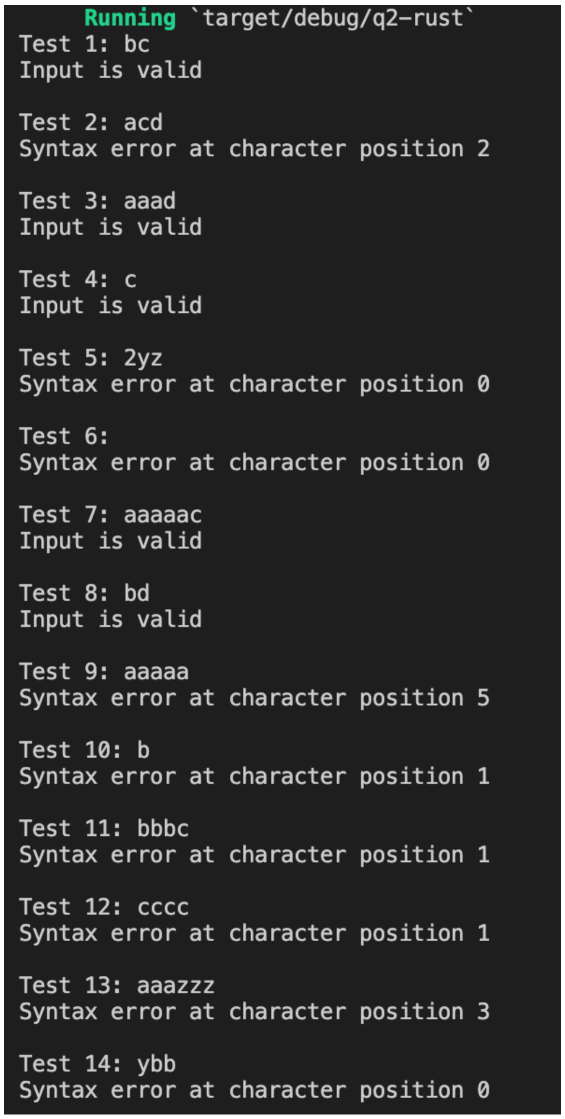
This panic occurred because char_pos reached the end of the string and went out of bounds. I fixed this by using < instead of <= in the comparisons.

Buggy code:

```
    fn get_next_char(&mut self) -> char {
        self.char_pos += 1;
        if self.char_pos <= self.str_in.len() as i32 {
            return self.str_in.chars().nth(self.char_pos as usize).unwrap();
        }
        return ' ';
    }

    fn peek_next_char(&self) -> char {
        if self.char_pos + 1 <= self.str_in.len() as i32 {
            return self.str_in.chars().nth((self.char_pos + 1) as usize).unwrap();
        }
        return ' ';
    }
```

Fixed code:

```rust
fn get_next_char(&mut self) -> char {
    self.char_pos += 1;
    if self.char_pos < self.str_in.len() as i32 {
    ...
}

fn peek_next_char(&self) -> char {
    if self.char_pos + 1 < self.str_in.len() as i32 {
    ...
}
```

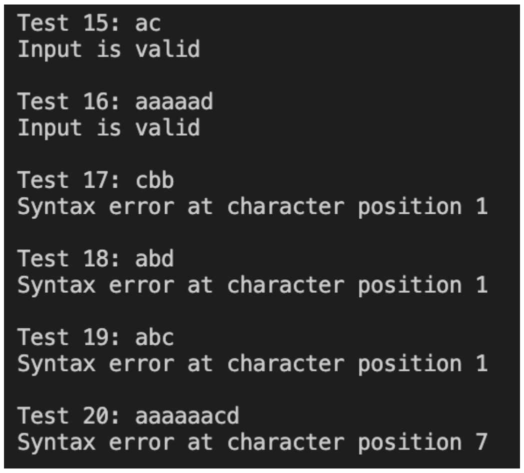


```
     Running `target/debug/q2-rust`
Test 1: bc
Input is valid

Test 2: acd
Syntax error at character position 2

Test 3: aaad
Input is valid

Test 4: c
Input is valid

Test 5: 2yz
Syntax error at character position 0

Test 6:
Syntax error at character position 0

Test 7: aaaaac
Input is valid

Test 8: bd
Input is valid

Test 9: aaaaa
Syntax error at character position 5

Test 10: b
Syntax error at character position 1

Test 11: bbbc
Syntax error at character position 1

Test 12: cccc
Syntax error at character position 1

Test 13: aaazzz
Syntax error at character position 3

Test 14: ybb
Syntax error at character position 0
```

All test cases have passed.

Additional test cases used to check behavior of Rust program:

- `"ac"` (valid)
- `"aaaaad"` (valid)
- `"cbb"` (invalid at 1)
- `"abd"` (invalid at 1)
- `"aaaaaacd"` (invalid at 7)



```
Test 15: ac
Input is valid

Test 16: aaaaad
Input is valid

Test 17: cbb
Syntax error at character position 1

Test 18: abd
Syntax error at character position 1

Test 19: abc
Syntax error at character position 1

Test 20: aaaaaacd
Syntax error at character position 7
```

All additional test cases have passed.

Step 5: Add Documentation

```
// Sources:
// https://crates.io/crates/custom_error
// https://docs.rs/custom_error/latest/custom_error/macro.custom_error.html
// https://doc.rust-lang.org/std/result/

extern crate custom_error;
use custom_error::custom_error;

custom_error! { ParseError
    Syntax{pos: i32} = "Syntax error at character position {pos}"
}

struct SimpleParser {
    str_in: String,
    char_pos: i32,
}

// Assuming string input has no whitespace
// No null value in Rust; so using ' ' if nothing left in string to parse
impl SimpleParser {
    // Initializer
    fn new(s: String) -> SimpleParser {
        return SimpleParser {
            str_in: s, // string input
            char_pos: -1 // current position in string input
        };
    }
    // Handles grammar rule for S
    // Prints message according to if grammar rule is satistfied
    fn fun_s(&mut self) {
        let ch: char;
        // don't move position if string[0] is not 'a' or 'b' and let fun_x() handle
        if self.char_pos == -1 && self.peek_next_char() != 'a'
            && self.peek_next_char() != 'b' {
            ch = self.peek_next_char();
        } else { // if char is 'a' or 'b'
            ch = self.get_next_char();
        }

        if ch == 'a' && ch == self.peek_next_char() { // handle repeated 'a' chars
            self.fun_s();
        } else {
            match self.fun_x() {
                Ok(()) => println!("Input is valid"),
                Err(e) => println!("{}", e.to_string())
            }
        }
    }

    // Handles grammar rule for X
    // Returns an error if grammar rule not satisfied
    fn fun_x(&mut self) -> Result<(), ParseError> {
        let ch = self.get_next_char();
        if (ch == 'c' || ch == 'd') && self.get_next_char() == ' ' {
            return Ok(()); // return success
        }
        return Err(ParseError::Syntax{pos: self.char_pos}); // return fail
    }

    // Move forward 1 position and return char at new position
    // Return ' ' if out of bounds
    fn get_next_char(&mut self) -> char {
        self.char_pos += 1;
        if self.char_pos < self.str_in.len() as i32 {
            return self.str_in.chars().nth(self.char_pos as usize).unwrap();
        }
        return ' ';
    }

    // Returns char 1 position ahead of current
    // Returns ' ' if out of bounds
    fn peek_next_char(&self) -> char {
        if self.char_pos + 1 < self.str_in.len() as i32 {
            return self.str_in.chars().nth((self.char_pos + 1) as usize).unwrap();
        }
        return ' ';
    }
}
```