ECS 140A Homework 4 – Problem 1

1 Python

Step 1: Algorithm/Pseudocode

```
enum TokenType:
    define CONSTANT, OPERATOR, VARIABLE, SPECIAL
    CONSTANT = "constant"
    OPERATOR = "operator"
    VARIABLE = "variable"
    SPECIAL = "special symbol"

class Token:
    init(tok_text, tok_type):
        set text to tok_text
        set token_type to tok_type

get_tokens(input):
    replace all spaces with empty string
    parse input char by char
        if see 0 or 1, create constant token
        if see a, b, c, or d, create variable token
        if see ":=" or ':', create special token
        all else is operator token
            handle relational operators (==, <=, >=, !=)
            if see char in "=<>!", check if next char is '='

print_tokens(toks):
    loop over toks array
        print(token text)
        print(token type)
```

Step 2: Actual Code

```python
# Sources:
# https://docs.python.org/3/library/enum.html

from enum import Enum

class TokenType(Enum):
    CONSTANT = "constant"
    OPERATOR = "operator"
    VARIABLE = "variable"
    SPECIAL = "special symbol"

class Token:
    def __init__(self, tok_text, tok_type):
        self.text = tok_text
        self.token_type = tok_type

# Assume program is syntactically correct (no non-sensical tokens)
def get_tokens(text):
    text.replace(" ", "")
    toks = []
    i = 0
    while i < len(text):
        if text[i] == 0 or text[i] == 1:
            toks.append(Token(text[i], TokenType.CONSTANT))
        elif 'a' <= text[i] <= 'd':
            toks.append(Token(text[i], TokenType.VARIABLE))
        elif text[i] == ';':
            toks.append(Token(";", TokenType.SPECIAL))
        elif text[i] == ':' and text[i+1] == '=':
            toks.append(Token(":=", TokenType.SPECIAL))
            i += 1
        elif text[i] in "=<>!":
            if text[i+1] == '=':
                toks.append(Token(text[i:i+2], TokenType.OPERATOR))
                i += 1
            else:
                toks.append(Token(text[i], TokenType.OPERATOR))
        i += 1
    return toks

def print_tokens(toks):
    for i, tok in enumerate(toks):
        print(f"Token {i} = {tok.text}")
        print(f"Token type: {tok.token_type}")
        print()
```

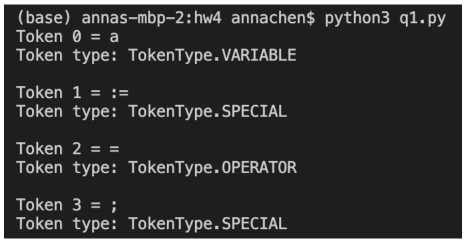
Step 3: Working Code

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

Step 4: Debug Process

Bug 1

Test input: `"a := 0 + 1;"`



```
(base) annas-mbp-2:hw4 annachen$ python3 q1.py
Token 0 = a
Token type: TokenType.VARIABLE

Token 1 = :=
Token type: TokenType.SPECIAL

Token 2 = =
Token type: TokenType.OPERATOR

Token 3 = ;
Token type: TokenType.SPECIAL
```

Not catching the 0 and 1 tokens properly because I was comparing 0 and 1 as integers instead of string literals.

Buggy code:

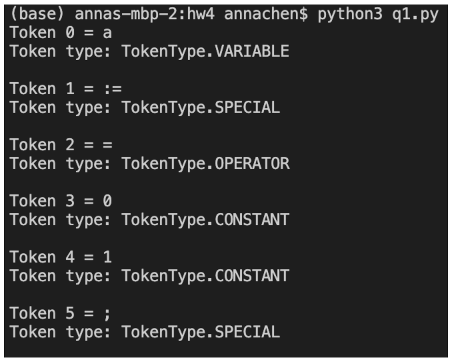
```
if text[i] == 0 or text[i] == 1:
```

Fixed code:

```
if text[i] == '0' or text[i] == '1':
```

Bug 2

Test input: `"a := 0 + 1;"`



```
(base) annas-mbp-2:hw4 annachen$ python3 q1.py
Token 0 = a
Token type: TokenType.VARIABLE

Token 1 = :=
Token type: TokenType.SPECIAL

Token 2 = =
Token type: TokenType.OPERATOR

Token 3 = 0
Token type: TokenType.CONSTANT

Token 4 = 1
Token type: TokenType.CONSTANT

Token 5 = ;
Token type: TokenType.SPECIAL
```

As you can see, the code read in the equals sign twice for the `:=` token. This is because I forgot to increment `i` to skip the equals sign.

Buggy code:

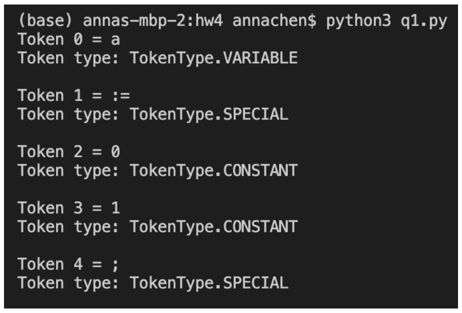
```
while i < len(text):
    ...
    elif text[i] == ':' and text[i+1] == '=':
        toks.append(Token(":=", TokenType.SPECIAL))
    ...
    i += 1
```

Fixed code:

```
while i < len(text):
    ...
    elif text[i] == ':' and text[i+1] == '=':
        toks.append(Token(":=", TokenType.SPECIAL))
        i += 1
    ...
    i += 1
```

Bug 3

Test input: `"a := 0 + 1;"`



```
(base) annas-mbp-2:hw4 annachen$ python3 q1.py
Token 0 = a
Token type: TokenType.VARIABLE

Token 1 = :=
Token type: TokenType.SPECIAL

Token 2 = 0
Token type: TokenType.CONSTANT

Token 3 = 1
Token type: TokenType.CONSTANT

Token 4 = ;
Token type: TokenType.SPECIAL
```

The `+` operator token was not created because I forgot to include an else condition in the outer if-else block.

Buggy code:

```
while i < len(text):
    ...
    elif text[i] in "=<>!":
        if text[i+1] == '=':
            toks.append(Token(text[i:i+2], TokenType.OPERATOR))
            i += 1
        else:
            toks.append(Token(text[i], TokenType.OPERATOR))
    i += 1
```

Fixed code:

```
while i < len(text):
    ...
    else:
        if text[i] in "=<>!" and text[i+1] == '=':
            toks.append(Token(text[i:i+2], TokenType.OPERATOR))
            i += 1
        else:
            toks.append(Token(text[i], TokenType.OPERATOR))
    i += 1
```

Bug 4

The fix for Bug 3 led to spaces being treated as extra tokens. This is because I did not use the string `replace` method correctly and thought the replacement was done in-place.

Buggy code:

```python
text.replace(" ", "")
```

Fixed code:

```python
text = text.replace(" ", "")
```

Bug 5

To print the value associated with the enumerated type TokenType instead of `TokenType.some_type`, I had to use the `value` attribute.

Buggy code:

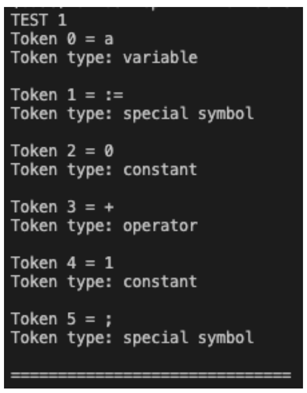
```python
def print_tokens(toks):
    for i, tok in enumerate(toks):
        print(f"Token {i} = {tok.text}")
        print(f"Token type: {tok.token_type}")
        print()
```

Fixed code:

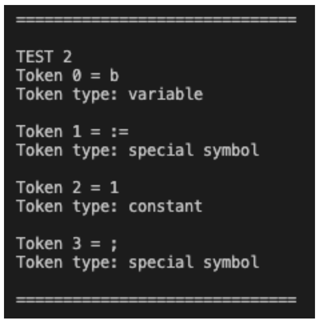
```python
def print_tokens(toks):
    for i, tok in enumerate(toks):
        print(f"Token {i} = {tok.text}")
        print(f"Token type: {tok.token_type.value}")
        print()
```

After fixing these bugs, all given and additional test cases passed.

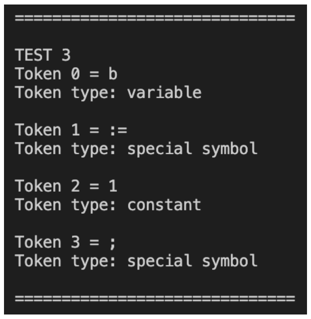
Testing output:



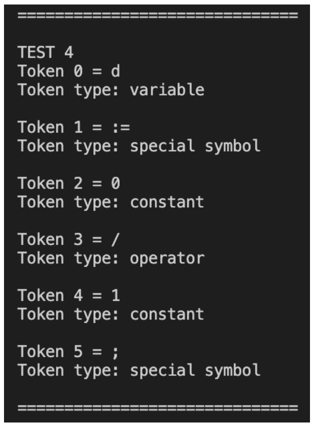
```
TEST 1
Token 0 = a
Token type: variable

Token 1 = :=
Token type: special symbol

Token 2 = 0
Token type: constant

Token 3 = +
Token type: operator

Token 4 = 1
Token type: constant

Token 5 = ;
Token type: special symbol

==============================
```



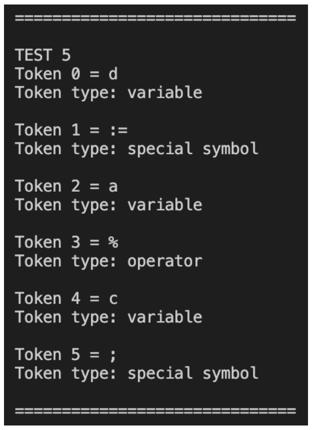
```
==============================

TEST 2
Token 0 = b
Token type: variable

Token 1 = :=
Token type: special symbol

Token 2 = 1
Token type: constant

Token 3 = ;
Token type: special symbol

==============================
```



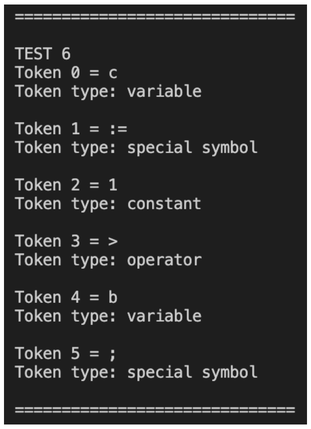
```
==============================

TEST 3
Token 0 = b
Token type: variable

Token 1 = :=
Token type: special symbol

Token 2 = 1
Token type: constant

Token 3 = ;
Token type: special symbol

==============================
```



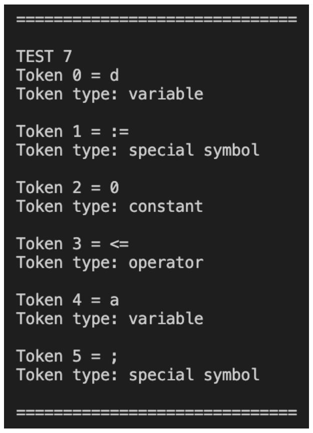
```
==============================

TEST 4
Token 0 = d
Token type: variable

Token 1 = :=
Token type: special symbol

Token 2 = 0
Token type: constant

Token 3 = /
Token type: operator

Token 4 = 1
Token type: constant

Token 5 = ;
Token type: special symbol

==============================
```



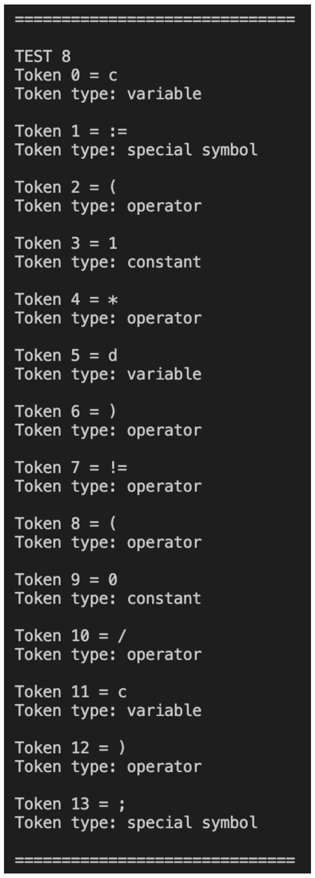
```
==============================

TEST 5
Token 0 = d
Token type: variable

Token 1 = :=
Token type: special symbol

Token 2 = a
Token type: variable

Token 3 = %
Token type: operator

Token 4 = c
Token type: variable

Token 5 = ;
Token type: special symbol

==============================
```



```
==============================

TEST 6
Token 0 = c
Token type: variable

Token 1 = :=
Token type: special symbol

Token 2 = 1
Token type: constant

Token 3 = >
Token type: operator

Token 4 = b
Token type: variable

Token 5 = ;
Token type: special symbol

==============================
```



```
==============================

TEST 7
Token 0 = d
Token type: variable

Token 1 = :=
Token type: special symbol

Token 2 = 0
Token type: constant

Token 3 = <=
Token type: operator

Token 4 = a
Token type: variable

Token 5 = ;
Token type: special symbol

==============================
```



```
==============================

TEST 8
Token 0 = c
Token type: variable

Token 1 = :=
Token type: special symbol

Token 2 = (
Token type: operator

Token 3 = 1
Token type: constant

Token 4 = *
Token type: operator

Token 5 = d
Token type: variable

Token 6 = )
Token type: operator

Token 7 = !=
Token type: operator

Token 8 = (
Token type: operator

Token 9 = 0
Token type: constant

Token 10 = /
Token type: operator

Token 11 = c
Token type: variable

Token 12 = )
Token type: operator

Token 13 = ;
Token type: special symbol

==============================
```

Step 5: Add Documentation

```python
# Sources:
# https://docs.python.org/3/library/enum.html
# https://stackoverflow.com/questions/24487405/getting-value-of-enum-on-string-conversion

from enum import Enum

class TokenType(Enum):
    ''' Enum types given string values for easy token type printing '''
    CONSTANT = "constant"
    OPERATOR = "operator"
    VARIABLE = "variable"
    SPECIAL = "special symbol"

class Token:
    def __init__(self, tok_text, tok_type):
        ''' Initializer '''
        self.text = tok_text
        self.token_type = tok_type

def get_tokens(text):
    '''
    Assumes program is syntactically correct (no non-sensical tokens)
    Creates tokens (constant, variable, special, or operator) while parsing text
    and returns an array of the tokens created
    '''
    text = text.replace(" ", "") # replace all spaces in text
    toks = []
    i = 0
    while i < len(text): # parse char by char
        if text[i] == '0' or text[i] == '1':
            toks.append(Token(text[i], TokenType.CONSTANT))
        elif 'a' <= text[i] <= 'd':
            toks.append(Token(text[i], TokenType.VARIABLE))
        elif text[i] == ';':
            toks.append(Token(";", TokenType.SPECIAL))
        elif text[i] == ':' and text[i+1] == '=':
            toks.append(Token(":=", TokenType.SPECIAL))
            i += 1 # skip '=' to not double count it
        else:
            if text[i] in "=<>!" and text[i+1] == '=':
                toks.append(Token(text[i:i+2], TokenType.OPERATOR))
                i += 1 # skip '=' to not double count it
            else:
                toks.append(Token(text[i], TokenType.OPERATOR))
        i += 1
    return toks

def print_tokens(toks):
    ''' Prints token text and token type for every token in token array '''
    for i, tok in enumerate(toks):
        print(f"Token {i} = {tok.text}")
        print(f"Token type: {tok.token_type.value}")
        print()
```

Step 6: Extra Test Cases Used

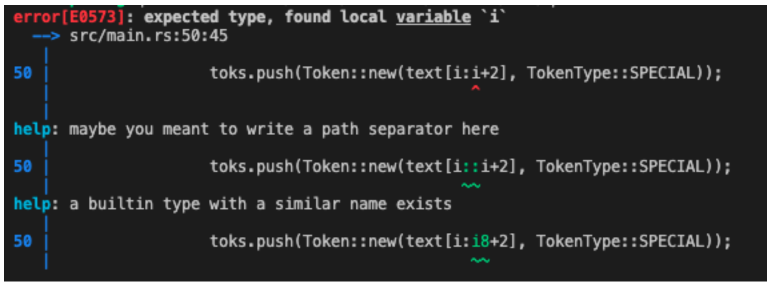
- `"b := 1;"`
- `"d := 0 / 1;"`
- `"d := a % c;"`
- `"c := 1 > b;"`
- `"d := 0 <= a;"`
- `"c := (1 * d) != (0 / c);"`

2 Rust

Step 2: Actual Code

```rust
// Sources:
// https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html
// https://stackoverflow.com/questions/36928569/
//     how-can-i-create-enums-with-constant-values-in-rust

enum TokenType {
    CONSTANT,
    OPERATOR,
    VARIABLE,
    SPECIAL
}

impl TokenType {
    fn value(&self) -> String {
        match *self {
            TokenType::CONSTANT => "constant",
            TokenType::OPERATOR => "operator",
            TokenType::VARIABLE => "variable",
            TokenType::SPECIAL => "special symbol"
        }
    }
}

struct Token {
    text: String,
    token_type: TokenType,
}

impl Token {
    fn new(tok_text: String, tok_type: TokenType) -> Token {
        return Token {
            text: tok_text,
            token_type: tok_type
        };
    }
}

fn get_tokens(text: String) -> Vec<Token> {
    let mut ch: char;
    let mut toks = Vec::new();
    let mut i = 0;
    while i < text.len() {
        ch = text.chars().nth(i as usize).unwrap();
        if ch == '0' || ch == '1' {
            toks.push(Token::new(ch.to_string(), TokenType::CONSTANT));
        } else if 'a' <= ch && ch <= 'd' {
            toks.push(Token::new(ch.to_string(), TokenType::VARIABLE));
        } else if ch == ';' {
            toks.push(Token::new(ch.to_string(), TokenType::SPECIAL));
        } else if ch == ':' && text.chars().nth(i+1 as usize).unwrap() == '=' {
            toks.push(Token::new(":=".to_string(), TokenType::SPECIAL));
            i += 1;
        } else {
            if "=<>!".contains(ch) && text.chars().nth(i+1 as usize).unwrap() == '=' {
                toks.push(Token::new(text[i:i+2], TokenType::SPECIAL));
                i += 1;
            } else {
                toks.push(Token::new(ch.to_string(), TokenType::OPERATOR));
            }
        }
        i += 1;
    }
    return toks;
}

fn print_tokens(tokens: Vec<Token>) {
    for (i, tok) in tokens.iter().enumerate() {
        println!("Token {} = {}", i, tok.text);
        println!("Token type: {}", tok.token_type.value());
        println!();
    }
}
```

Syntax Error 1



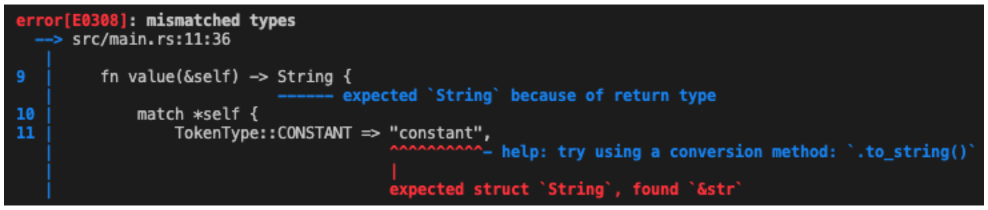
```
error[E0573]: expected type, found local variable `i`
  --> src/main.rs:50:45
   |
50 |                 toks.push(Token::new(text[i:i+2], TokenType::SPECIAL));
   |                                             ^
   |
help: maybe you meant to write a path separator here
   |
50 |                 toks.push(Token::new(text[i::i+2], TokenType::SPECIAL));
   |                                            ~~
help: a builtin type with a similar name exists
   |
50 |                 toks.push(Token::new(text[i:i8+2], TokenType::SPECIAL));
   |                                             ~~
```

I used the wrong substring / slicing syntax here, and need to convert to String type.

Fixed code:

```
toks.push(Token::new(text[i..i+2].to_string(), TokenType::SPECIAL));
```

Syntax Error 2



```
error[E0308]: mismatched types
  --> src/main.rs:11:36
   |
9  |     fn value(&self) -> String {
   |                        ------ expected `String` because of return type
10 |         match *self {
11 |             TokenType::CONSTANT => "constant",
   |                                    ^^^^^^^^^^- help: try using a conversion method: `.to_string()`
   |                                    |
   |                                    expected struct `String`, found `&str`
```

I forgot to convert the string literal to String type here.

Fixed code:

```
impl TokenType {
    fn value(&self) -> String {
        match *self {
            TokenType::CONSTANT => "constant".to_string(),
            TokenType::OPERATOR => "operator".to_string(),
            TokenType::VARIABLE => "variable".to_string(),
            TokenType::SPECIAL => "special symbol".to_string()
        }
    }
}
```

Step 3: Working Code

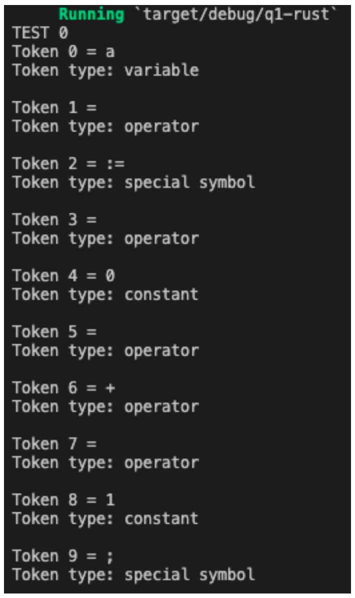
```
// Sources:
// https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html
// https://stackoverflow.com/questions/36928569/
//     how-can-i-create-enums-with-constant-values-in-rust

enum TokenType {
    CONSTANT,
    OPERATOR,
    VARIABLE,
    SPECIAL
}

impl TokenType {
    fn value(&self) -> String {
        match *self {
            TokenType::CONSTANT => "constant".to_string(),
            TokenType::OPERATOR => "operator".to_string(),
            TokenType::VARIABLE => "variable".to_string(),
            TokenType::SPECIAL => "special symbol".to_string()
        }
    }
}

struct Token {
    text: String,
    token_type: TokenType,
}

impl Token {
    fn new(tok_text: String, tok_type: TokenType) -> Token {
        return Token {
            text: tok_text,
            token_type: tok_type
        };
    }
}
```

```
fn get_tokens(text: String) -> Vec<Token> {
    let mut ch: char;
    let mut toks = Vec::new();
    let mut i = 0;
    while i < text.len() {
        ch = text.chars().nth(i as usize).unwrap();
        if ch == '0' || ch == '1' {
            toks.push(Token::new(ch.to_string(), TokenType::CONSTANT));
        } else if 'a' <= ch && ch <= 'd' {
            toks.push(Token::new(ch.to_string(), TokenType::VARIABLE));
        } else if ch == ';' {
            toks.push(Token::new(ch.to_string(), TokenType::SPECIAL));
        } else if ch == ':' && text.chars().nth(i+1 as usize).unwrap() == '=' {
            toks.push(Token::new(":=".to_string(), TokenType::SPECIAL));
            i += 1;
        } else {
            if "=<>!".contains(ch) && text.chars().nth(i+1 as usize).unwrap() == '=' {
                toks.push(Token::new(text[i..i+2].to_string(), TokenType::SPECIAL));
                i += 1;
            } else {
                toks.push(Token::new(ch.to_string(), TokenType::OPERATOR));
            }
        }
        i += 1;
    }
    return toks;
}

fn print_tokens(tokens: Vec<Token>) {
    for (i, tok) in tokens.iter().enumerate() {
        println!("Token {} = {}", i, tok.text);
        println!("Token type: {}", tok.token_type.value());
        println!();
    }
}
```

Step 4: Debug Process

Bug 1

Test input: `"a := 0 + 1;"`



```
     Running `target/debug/q1-rust`
TEST 0
Token 0 = a
Token type: variable

Token 1 =
Token type: operator

Token 2 = :=
Token type: special symbol

Token 3 =
Token type: operator

Token 4 = 0
Token type: constant

Token 5 =
Token type: operator

Token 6 = +
Token type: operator

Token 7 =
Token type: operator

Token 8 = 1
Token type: constant

Token 9 = ;
Token type: special symbol
```

Spaces were treated as tokens. This is because I forgot to get rid of all the spaces first and replace them with the empty string.

Buggy code:

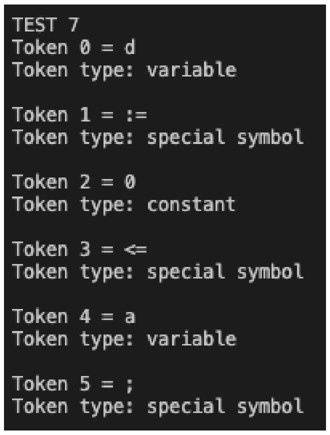
```rust
fn get_tokens(text: String) -> Vec<Token> {
    let mut ch: char;
    let mut toks = Vec::new();
    let mut i = 0;
    ...
    return toks;
}
```

Fixed code:

```rust
fn get_tokens(mut text: String) -> Vec<Token> {
    let mut ch: char;
    let mut toks = Vec::new();
    let mut i = 0;
    text = text.replace(" ", "");
    ...
    return toks;
}
```

Bug 2

Test input: `"d := 0 <= a;"`



```
TEST 7
Token 0 = d
Token type: variable

Token 1 = :=
Token type: special symbol

Token 2 = 0
Token type: constant

Token 3 = <=
Token type: special symbol

Token 4 = a
Token type: variable

Token 5 = ;
Token type: special symbol
```

The `<=` token is incorrectly classified as a special symbol instead of an operator. This is because I accidentally created tokens as the special type for `==`, `<=`, `>=`, `!=`.

Buggy code:

```
if "=<>!".contains(ch) && text.chars().nth(i+1 as usize).unwrap() == '=' {
    toks.push(Token::new(text[i..i+2].to_string(), TokenType::SPECIAL));
    i += 1;
}
```

Fixed code:

```
if "=<>!".contains(ch) && text.chars().nth(i+1 as usize).unwrap() == '=' {
    toks.push(Token::new(text[i..i+2].to_string(), TokenType::OPERATOR));
    i += 1;
}
```

After fixing these bugs, all given and additional test cases (can be found in Python section) outputted as expected.

Step 5: Add Documentation

```rust
// Sources:
// https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html
// https://stackoverflow.com/questions/36928569/
//     how-can-i-create-enums-with-constant-values-in-rust

enum TokenType {
    CONSTANT,
    OPERATOR,
    VARIABLE,
    SPECIAL
}

impl TokenType {
    // For ease of printing token type, assign string values to the enums
    fn value(&self) -> String {
        match *self {
            TokenType::CONSTANT => "constant".to_string(),
            TokenType::OPERATOR => "operator".to_string(),
            TokenType::VARIABLE => "variable".to_string(),
            TokenType::SPECIAL => "special symbol".to_string()
        }
    }
}

struct Token {
    text: String,
    token_type: TokenType,
}

impl Token {
    // Initializer
    fn new(tok_text: String, tok_type: TokenType) -> Token {
        return Token {
            text: tok_text,
            token_type: tok_type
        };
    }
}
```

```rust
// Assumes program is syntactically correct (no non-sensical tokens)
// Creates tokens (constant, variable, special, or operator) while parsing text
// and returns a vector of the tokens created
fn get_tokens(mut text: String) -> Vec<Token> {
    let mut ch: char;
    let mut toks = Vec::new();
    let mut i = 0;
    text = text.replace(" ", ""); // replace all spaces in text
    while i < text.len() { // parse text char by char
        ch = text.chars().nth(i as usize).unwrap();
        if ch == '0' || ch == '1' {
            toks.push(Token::new(ch.to_string(), TokenType::CONSTANT));
        } else if 'a' <= ch && ch <= 'd' {
            toks.push(Token::new(ch.to_string(), TokenType::VARIABLE));
        } else if ch == ';' {
            toks.push(Token::new(ch.to_string(), TokenType::SPECIAL));
        } else if ch == ':' && text.chars().nth(i+1 as usize).unwrap() == '=' {
            toks.push(Token::new(":=".to_string(), TokenType::SPECIAL));
            i += 1; // skip '=' to not double count it
        } else {
            if "=<>!".contains(ch) && text.chars().nth(i+1 as usize).unwrap() == '=' {
                toks.push(Token::new(text[i..i+2].to_string(), TokenType::OPERATOR));
                i += 1; // skip '=' to not double count it
            } else {
                toks.push(Token::new(ch.to_string(), TokenType::OPERATOR));
            }
        }
        i += 1;
    }
    return toks;
}

// Prints token text and token type for every token in token vector
fn print_tokens(tokens: Vec<Token>) {
    for (i, tok) in tokens.iter().enumerate() {
        println!("Token {} = {}", i, tok.text);
        println!("Token type: {}", tok.token_type.value());
        println!();
    }
}
```