BURSA TEKNİK ÜNİVERSİTESİ
MÜHENDİSLİK VE DOĞA BİLİMLERİ FAKÜLTESİ
Bilgisayar Mühendisliği Bölümü

# PROGRAMLAMA DİLLERİ DERSİ

# PROJE ÖDEVİ DOKÜMANI

## REAL-TIME GRAMMER-BASED SYNTAX HIGHLIGHTER WITH GUI

**ERVA NUR BOSTANCI**

22360859060

HAZİRAN 2025

# 1. GİRİŞ

Programlama dünyasında, kodun anlaşılabilirliği ve geliştirici verimliliği hayati öneme sahiptir. Karmaşık yazılım projeleriyle çalışırken, geliştiricilerin kodun yapısını hızlıca kavraması, hataları tespit etmesi ve hata ayıklaması büyük önem taşır. Bu bağlamda, sözdizimi vurgulayıcıları (syntax highlighters)**,** kod düzenleyicilerinin önemli bir parçası haline gelmiştir. Sözdizimi vurgulayıcıları, kaynak kodun farklı bileşenlerini (anahtar kelimeler, operatörler, dizeler, sayılar vb.) renkler ve stil değişiklikleriyle görsel olarak ayırt ederek, kodun okunabilirliğini önemli ölçüde artırır ve sözdizimsel hataların erkenden fark edilmesine yardımcı olur.

Bu projenin temel amacı, gerçek zamanlı bir sözdizimi vurgulayıcıya sahip bir grafik kullanıcı arayüzü (GUI) geliştirmektir**.** Uygulama, kullanıcının yazdığı kodu anında analiz ederek, tanımlı dilbilgisi kurallarına uygun olarak ilgili sözdizimi bileşenlerini vurgulayacak ve dilbilgisi kurallarına uymayan hataları tespit edip kullanıcıya bildirecektir. Bu süreç, sözcüksel analiz (lexical analysis) ve sözdizimsel analiz (syntactic analysis / parsing) olmak üzere iki ana aşamadan oluşmaktadır. Sözcüksel analiz, ham kodu anlamlı birimlere (token'lara) ayırırken, sözdizimsel analiz bu token akışını belirli bir biçimsel dilbilgisine (formal grammar) göre ayrıştırarak kodun yapısal geçerliliğini kontrol eder.

Proje kapsamında, kendi belirlediğimiz basit bir programlama dilinin sözdizimi kuralları tanımlanacak ve bu kurallara uygun bir lexer ile bir top-down recursive-descent parser implementasyonu yapılacaktır. Geliştirilecek GUI, kullanıcı dostu bir arayüz sunarak, kod yazma deneyimini görsel geri bildirim ve hata tespitiyle zenginleştirecektir. Bu rapor, projenin tasarım seçimlerini, teknik detaylarını ve implementasyon metodolojilerini kapsamlı bir şekilde belgelemektedir.

# 2. PROGRAMLAMA DİLİ VE DİLBİLGİSİ SEÇİMİ

Bu projede, gerçek zamanlı bir sözdizimi vurgulayıcı ve ayrıştırıcı geliştirmek amacıyla, karmaşıklığı yönetilebilir düzeyde tutulan, **"MiniLang"** adını verdiğimiz basit bir programlama dili tasarlanmıştır. MiniLang'ın sözdizimi, modern programlama dillerinin temel yapı taşlarını (değişkenler, atamalar, kontrol akışı ifadeleri, fonksiyonlar) içerecek şekilde tasarlanmış olup, C veya JavaScript benzeri bir blok yapısını ({}) benimsemektedir. Bu yaklaşım, karmaşık bir dilin tüm detaylarını uygulamak yerine, sözdizimi analizi süreçlerinin (leksiksel ve sözdizimsel analiz) temel prensiplerini etkin bir şekilde göstermemizi sağlamıştır.

MiniLang'ın temel dilbilgisi kuralları aşağıda Genişletilmiş Backus-Naur Form (EBNF) notation'ı kullanılarak sunulmuştur:

```
# Programın başlangıç noktası: Bir veya daha fazla ifadeden oluşur.
Program       ::= { Statement }

# İfadelerin listesi: Dilin temel yapı taşlarını temsil eder.
Statement     ::= IfStatement
                | WhileStatement
                | ForStatement
                | FunctionDefinition
                | ReturnStatement
                | Assignment
                | AugmentedAssignment
                | FunctionCall
                | Expression

# Koşul ifadeleri: 'if' ve 'while' gibi kontrol akışı yapılarında kullanılır, parantez içine alınır.
Condition     ::= "(" Expression ")"

# If İfadesi: Bir koşul ve bu koşul doğruysa çalışacak bir blok, isteğe bağlı olarak bir 'else' bloğu içerebilir.
IfStatement   ::= "if" Condition Block
                | "if" Condition Block "else" Block

# While Döngüsü: Bir koşul doğru olduğu sürece tekrar eden bir blok.
WhileStatement ::= "while" Condition Block

# For Döngüsü: 'range' fonksiyonu kullanılarak belirli bir aralıkta yinelenen bir döngü.
ForStatement  ::= "for" IDENTIFIER "in" "range" "(" Expression ")" Block
```

```
# Fonksiyon Tanımı: 'def' anahtar kelimesiyle başlayan, bir isim, parametreler ve bir kod bloğu içeren yapı.
FunctionDefinition ::= "def" IDENTIFIER "(" [ ParameterList ] ")" Block
ParameterList      ::= IDENTIFIER { "," IDENTIFIER }

# Fonksiyon Çağrısı: Tanımlanmış bir fonksiyonu argümanlarla çağırma.
FunctionCall   ::= IDENTIFIER "(" [ ArgumentList ] ")"
ArgumentList   ::= Expression { "," Expression }

# Atama İfadesi: Bir değişkene değer atama. Her atama ifadesi noktalı virgül (;) ile sonlanır.
Assignment         ::= IDENTIFIER "=" Expression

# Birleşik Atama İfadesi: Bir değişkene bir işlemi uygulayarak değer atama (örn: x += 5).
AugmentedAssignment ::= IDENTIFIER ("+=" | "-=" | "*=" | "/=") Expression

# Kod Bloğu: Süslü parantezlerle çevrili bir veya daha fazla ifadeden oluşan yapı.
Block          ::= "{" { Statement } "}"
# İfadeler: Operatör önceliğine göre çözümlenen aritmetik ve karşılaştırma ifadeleri.
# Operatör önceliği: *, / > +, - > Karşılaştırma Operatörleri > mantıksal operatörler
Expression ::= LogicalOrExpression
 LogicalOrExpression ::= LogicalAndExpression { "or" LogicalAndExpression }
 LogicalAndExpression ::= Comparison { "and" Comparison }
Comparison ::= Term { ("==" | "!=" | "<=" | ">=" | ">" | "<") Term }
Term ::= Factor { ("+" | "-") Factor }
Factor ::= Primary { ("*" | "/") Primary }
Primary ::= NUMBER
            | FLOAT
            | STRING
            | IDENTIFIER
            | FunctionCall
            | "(" Expression ")"
```

# Terminal Semboller (Tokenlar): Lexer tarafından tanınan temel birimler.

# KEYWORDs: if, else, while, for, def, return, in, range, and, or

# OPERATORs: =, ==, !=, <, >, <=, >=, +, -, *, /, +=, -=, *=, /=, (, ), {, }, ,, ;

# NUMBER: Ondalık nokta içermeyen rakam dizisi (örn: 123)

# FLOAT: Ondalık nokta içeren rakam dizisi (örn: 12.34)

# STRING: Çift tırnaklar arasına alınmış karakter dizisi (örn: "merhaba")

# IDENTIFIER: Harf veya alt çizgi ile başlayan, harf, rakam veya alt çizgi içeren isimler (örn: myVar, calculate_sum)

# COMMENT: '#' karakteri ile başlayan satır sonuna kadar devam eden metin.

# WHITESPACE: Boşluk, tab ve yeni satır karakterleri.

# UNKNOWN: Tanımlı hiçbir kurala uymayan karakterler.

# 3. SÖZCÜKSEL ANALİZ (LEXICAL ANALYSIS)

Sözcüksel analiz, programlama dilinin ilk aşaması olup, kaynak kodu karakter akışından token adı verilen anlamlı birimlere dönüştürmekten sorumludur. Bu süreç, dilin kelimelerini ve sembollerini tanır. Projemizde bu aşama, lexer.py dosyası içinde uygulanan bir "Durum Diyagramı ve Program Uygulaması" yaklaşımıyla gerçekleştirilmiştir. Bu yaklaşım, her bir token türü için düzenli ifadeler (regular expressions) tanımlanmasına ve bu desenleri kaynak kod üzerinde adım adım eşleştirmeye dayanır.

## 3.1. Yaklaşım Seçimi: Durum Diyagramı ve Program Uygulaması

Bu yaklaşım, her bir token türünü temsil eden düzenli ifadelerin doğrudan program kodunda (Python'ın re modülü aracılığıyla) tanımlanmasına ve bu desenlerin kaynak kodu üzerinde bir "durum makinesi" gibi ileriye doğru hareket ederek eşleşmeler aramasından oluşur. Düzenli ifadelerin bir sözlük içinde gruplandırılması ve belirli bir öncelik sırasına göre işlenmesi, doğru tokenizasyonun anahtarını oluşturur.

## 3.2. Token Türleri ve Düzenli İfadeler

MiniLang için tanımlanan token türleri ve bunlara karşılık gelen düzenli ifadeler TOKEN_TYPES sözlüğünde belirtilmiştir:

- **KEYWORD (Anahtar Kelime):** Dilin özel anlamı olan ayrılmış kelimeleri (örn: if, else, while, for, def, return, in, range, and, or). Düzenli ifadede \b kelime sınırları, tam kelime eşleşmesi sağlar.

- **OPERATOR (Operatör):** Aritmetik, karşılaştırma, atama ve yapısal semboller (örn: ==, !=, +, -, =, (, ), {, }, ;, ,, +=, -=, *=, /=). Çok karakterli operatörlerin (örn: ==) tek karakterli olanlardan (örn: =) önce tanımlanması, doğru eşleşmeler için kritik öneme sahiptir.
- **FLOAT (Ondalıklı Sayı):** Ondalık noktası içeren sayılar (örn: 12.34).
- **NUMBER (Sayı):** Tam sayılar (örn: 123). FLOAT deseninden sonra gelmesi önemlidir.
- **STRING (Dize):** Çift tırnaklar içine alınmış metinler (örn: "merhaba dünya").
- **IDENTIFIER (Tanımlayıcı):** Değişken, fonksiyon isimleri gibi kullanıcı tanımlı adlar. Bir harf veya alt çizgi ile başlar, ardından harf, rakam veya alt çizgi içerebilir.
- **COMMENT (Yorum):** # ile başlayan satırın sonuna kadar devam eden metinler.
- **WHITESPACE (Boşluk Karakteri):** Boşluk, tab ve yeni satır karakterleri. Bu token'lar genellikle ayrıştırıcı tarafından göz ardı edilir ancak kodun akışını ve token konum bilgilerini doğru takip etmek için leksiksel analizde önemlidir.
- **UNKNOWN (Bilinmeyen):** Tanımlı hiçbir düzenli ifadeye uymayan karakterleri yakalar, bu da olası bir leksiksel hatayı işaret eder.

## 3.3. Token Sınıfı (Token)

Her bir tanınan anlamlı birim, aşağıdaki niteliklere sahip bir Token sınıfı nesnesi olarak temsil edilir:

- type: Token'ın kategorisi (örn: KEYWORD, IDENTIFIER).
- value: Token'ın orijinal metin değeri (örn: "if", "myVar", "123").
- start, end: Token'ın kaynak kodda başladığı ve bittiği karakter indeksleri.
- line, column: Token'ın başladığı satır numarası (1-tabanlı) ve sütun numarası (0-tabanlı). Bu konum bilgileri, hata raporlama ve sözdizimi vurgulama için hayati öneme sahiptir.

```python
class Token:
    def __init__(self, type, value, start, end, line=None, column=None):
        self.type = type
        self.value = value
        self.start = start
        self.end = end
        self.line = line
        self.column = column

    def __repr__(self):
        # Hata mesajları için faydalı olan token bilgilerini içerir
        return f"Token(type='{self.type}', value='{self.value}', start={self.start}, end={self.end}, line={self.line}, column={self.column})"
```

### 3.4. Lexer Sınıfı (Lexer)

Lexer sınıfı, tokenizasyon sürecini yönetir. __init__ metodunda tüm düzenli ifadeler re.compile() ile derlenir ve işlenme sırasına göre bir listeye alınır.

```
def __init__(self):
    self.token_patterns = []
    # Token tiplerinin işlenme sırası önemlidir (örn: FLOAT, NUMBER'dan önce gelmeli)
    ordered_token_types = [
        'KEYWORD', 'OPERATOR', 'FLOAT', 'NUMBER', 'STRING',
        'IDENTIFIER', 'COMMENT', 'WHITESPACE', 'UNKNOWN'
    ]
    for token_type in ordered_token_types:
        pattern = TOKEN_TYPES[token_type]
        self.token_patterns.append((token_type, re.compile(pattern)))
```

tokenize(code) metodu, leksiksel analizin ana fonksiyonudur:

1. Girdi kodunu karakter karakter tarar.
2. Mevcut konumdan başlayarak, tanımlı düzenli ifadelerle mümkün olan en uzun eşleşmeyi arar.
3. Bir eşleşme bulunduğunda, ilgili Token nesnesini oluşturur ve token listesine ekler.
4. WHITESPACE ve COMMENT token'ları, parser'a gönderilmeden önce filtreleneceği için özel olarak işlenir; ancak bunların konumları ve satır/sütun bilgileri doğru bir şekilde ilerletilir.
5. Hiçbir desen eşleşmezse, tek bir karakteri UNKNOWN token'ı olarak işaretleyerek hata durumunu bildirir.
6. Tüm kod tarandıktan sonra, oluşturulan Token nesnelerinin listesini döndürür.

# 4. SÖZDİZİMSEL ANALİZ (SYNTAX ANALYSIS/PARSING)

Sözdizimsel analiz (parsing), leksiksel analizden gelen token akışını alarak, dilin biçimsel dilbilgisi kurallarına göre bir hiyerarşik yapı (genellikle bir **Soyut Sözdizimi Ağacı - AST**) oluşturur. Bu aşama, token'ların doğru bir sırada ve geçerli bir yapı oluşturup oluşturmadığını kontrol eder. Projemizde bu süreç, parser.py dosyası içinde implemente edilen bir **"Top-Down Recursive-Descent Parser"** yaklaşımıyla gerçekleştirilmiştir.

## 4.1. Yaklaşım Seçimi: Top-Down Recursive-Descent Parser

Bu parser türü, dilbilgisinin üretim kurallarını doğrudan ayrıştırma metodlarına dönüştürerek çalışır. Her bir non-terminal sembol (örn: Statement, Expression, IfStatement) parser sınıfında bir metoda (statement(), expression(), if_statement()) karşılık gelir. Bu metodlar, girdiyi (token akışını) yukarıdan aşağıya (en genel kuraldan en spesifik kurala) doğru tarayarak ve beklentilere uygun token'ları tüketerek ayrıştırma ağacını implicit veya explicit olarak inşa eder.

## 4.2. Parser Sınıfı (Parser)

Parser sınıfının ana bileşenleri şunlardır:

- **__init__(self, tokens):** Parser'ı, leksiksel analizden gelen filtrelenmiş (boşluk ve yorumlar hariç) token listesiyle başlatır. current_token, işlenecek mevcut token'ı gösterir.
- **advance():** current_token işaretçisini token akışında bir sonraki token'a ilerletir.
- **eat(token_type, token_value=None):** Ayrıştırma sürecinin en kritik metodudur. current_token'ın beklenen token_type ve isteğe bağlı token_value ile eşleşip eşleşmediğini kontrol eder. Eşleşirse, token'ı tüketir ve advance() çağırır. Eşleşmezse, bir ParserError fırlatarak sözdizimi hatası olduğunu bildirir.

## 4.3. Dilbilgisi Kurallarının Uygulanması (Metodlar)

Parser sınıfındaki her metod, MiniLang'ın belirli bir dilbilgisi kuralını uygulamak üzere tasarlanmıştır:

- **parse():** Ayrıştırma sürecinin giriş noktasıdır ve program() metodunu çağırır.
- **program():** Bir dizi Statement'ı ayrıştırır ve token akışı bitene kadar devam eder.
- **statement():** Mevcut current_token'ın türüne göre farklı ayrıştırma metodlarına (örn: if_statement(), assignment(), function_call()) dallanır. Bu, parser'ın hangi tür sözdizimi yapısını işleyeceğine karar verdiği merkezi bir noktadır.

- **if_statement(), while_statement(), for_statement(), function_definition(), return_statement():** Bu metodlar, ilgili anahtar kelimeleri, parantezleri, süslü parantezleri ve ifadeleri eat() metodunu kullanarak tüketir ve kontrol akışı yapılarını ile fonksiyon tanımlamalarını ayrıştırır.

- **assignment() ve augmented_assignment():** Değişken atamalarını (= operatörü) ve bileşik atamaları (+=, -=, *=, /= gibi operatörler) ayrıştırır.

- **expression(), comparison(), term(), factor(), primary():** Bu metodlar, aritmetik ve karşılaştırma ifadelerini işlerken **operatör önceliğini** doğru bir şekilde uygular. Örneğin, çarpma ve bölme (Factor seviyesinde) toplama ve çıkarmadan (Term seviyesinde) daha yüksek önceliğe sahiptir. Primary ise bir ifadenin en temel birimlerini (sayılar, dizeler, tanımlayıcılar, parantezli ifadeler, fonksiyon çağrıları) ayrıştırır.

- **function_call(), parameter_list(), argument_list():** Fonksiyon çağrılarını ve bunların parametre/argüman listelerini ayrıştırmayı yönetir.

### 4.4. Hata Yönetimi (ParserError)

Sözdizimi hataları, ParserError özel istisnası kullanılarak fırlatılır. Bu sınıf, hatanın mesajını ve hataya neden olan Token nesnesini içerir. Bu sayede GUI, hatanın tam konumunu (satır ve sütun) belirleyebilir ve kullanıcıya okunabilir bir hata mesajı sunabilir.

# 5. VURGULAMA ŞEMASI VE GUI UYGULAMASI

### 5.1. Vurgulama Şeması

Uygulama, kodun farklı token türlerini görsel olarak ayırt etmek için çeşitli renkler ve font stilleri kullanır. Bu sayede kodun okunabilirliği artırılır ve görsel olarak daha güzel bir deneyim sunulur. Vurgulanan başlıca token türleri ve renkleri şunlardır:

- **Anahtar Kelimeler (KEYWORD):** Mavi
- **Operatörler (OPERATOR):** Kırmızı
- **Sayılar (NUMBER, FLOAT):** Mor (float için turuncu)
- **Tanımlayıcılar (IDENTIFIER):** Siyah
- **Yorumlar (COMMENT):** Yeşil ve italik font

- **Dizeler (STRING):** Kahverengi
- **Bilinmeyen Tokenlar (UNKNOWN):** Sarı arka plan üzerinde gri metin (hata sinyali olarak)
- **Sözdizimi Hataları (ERROR):** Kırmızı arka plan üzerinde beyaz metin (hata sinyali olarak)

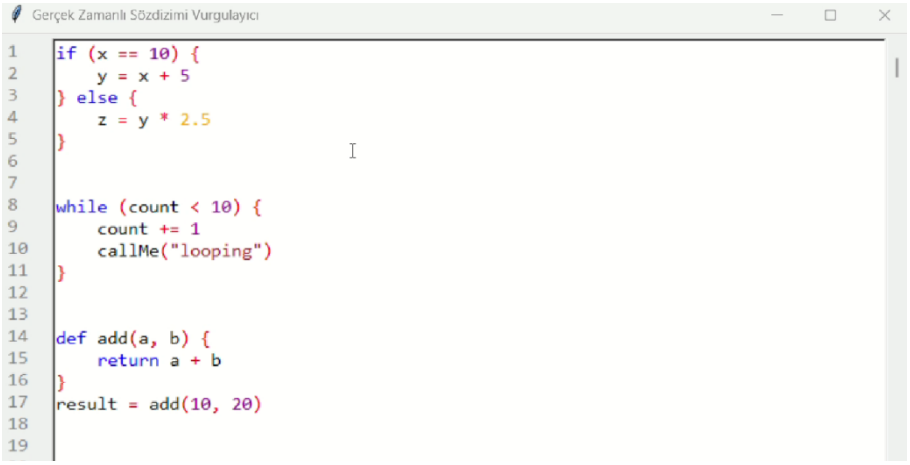
Renk seçimleri, genel bir okunabilirlik ve standart kod düzenleyici temalarıyla uyum sağlama amacı taşımaktadır.

## 5.2. Gerçek Zamanlı İşlevsellik

Uygulama, **gerçek zamanlı (real-time)** sözdizimi vurgulama ve hata tespiti sunar. Bu işlevsellik, Tkinter'ın olay bağlama (event binding) mekanizması aracılığıyla sağlanır:

- self.text_area.bind("<KeyRelease>", self.on_key_release): Kullanıcı metin alanında bir tuşu bıraktığında (<KeyRelease> olayı), on_key_release metodu otomatik olarak tetiklenir.
- on_key_release metodu içinde sırasıyla highlight_syntax() ve parse_and_report_errors() metodları çağrılır. Bu sayede, kullanıcının yazdığı her karakter sonrası kod anında yeniden analiz edilir ve güncel duruma göre vurgulanır veya hatalar bildirilir.

## 5.3. GUI Tasarımı ve Bileşenleri




- **Ana Pencere (tk.Tk):** Uygulamanın ana penceresi Tkinter'ın Tk sınıfı kullanılarak oluşturulur ve "Gerçek Zamanlı Sözdizimi Vurgulayıcı" başlığını taşır.
- **Kod Çerçevesi (tk.Frame):** Satır numaraları ve ana metin alanını düzenli bir şekilde yan yana konumlandırmak için bir tk.Frame kullanılır.
- **Metin Alanı (scrolledtext.ScrolledText):** Kullanıcının kod yazması için çok satırlı, kaydırılabilir bir metin alanı sağlanır. wrap=tk.WORD metnin kelime bazında sarılmasını sağlar. Arka plan rengi beyaz (#FFFFFF) ve imleç rengi siyah olarak ayarlanmıştır. Kenarlık stili tk.SUNKEN (batık) ve bd=2 (2 piksel genişliğinde) olarak belirlenmiştir.

- **Satır Numaraları Alanı (tk.Text):** Metin alanının solunda, tk.Text widget'ı kullanılarak satır numaraları görüntülenir.
    - width=4: Satır numaraları için yeterli sabit bir genişlik sağlar.
    - borderwidth=0: Ana metin alanı ile görsel bütünlük sağlamak için kenarlığı kaldırılmıştır.
    - background="#F0F0F0" ve foreground="gray": Arka plan açık gri, yazı rengi gri tonundadır, böylece koddan ayırt edilebilir ancak dikkat dağıtmaz.
    - state="disabled": Kullanıcının satır numaraları alanına yazmasını engellemek için salt okunur hale getirilmiştir.
- **Senkronizasyon:** Satır numaraları alanı ile ana metin alanı arasındaki kaydırma senkronizasyonu, yscrollcommand seçeneklerinin birbirlerinin yview metodlarına bağlanmasıyla ve update_line_numbers() metodunun düzenli olarak çağrılmasıyla sağlanır. Özellikle update_line_numbers_scroll_position() metodu, her iki alanın dikey kaydırma pozisyonlarını her zaman aynı oranda tutar.
- **Durum Çubuğu (tk.Label):** Uygulamanın altında, status_label adı verilen bir tk.Label widget'ı, sözdizimi geçerliliği veya tespit edilen hatalar hakkında geri bildirim sağlamak için kullanılır.
    - Geçerli sözdizimi durumunda metin yeşil renkte, hata durumunda ise kırmızı renkte görüntülenir.
    - Durum çubuğunun da bd=1 ve relief=tk.SUNKEN gibi hafif bir kenarlık stili bulunmaktadır.
- **Boşluk ve Hizalama:** pack() metoduna eklenen padx ve pady seçenekleri, widget'lar arasında uygun boşluklar bırakarak arayüzün daha ferah ve düzenli görünmesini sağlar.

# 6. DEĞERLENDİRME VE SONUÇ

Bu proje, gerçek zamanlı bir sözdizimi vurgulayıcı ve sözdizimsel hata tespit sistemi oluşturma hedefini başarıyla gerçekleştirmiştir. Projenin temel bileşenleri olan leksiksel çözümleyici, sözdizimsel ayrıştırıcı ve grafik kullanıcı arayüzü, MiniLang adlı özel bir programlama dili için tasarlanmış ve implemente edilmiştir.

- **Eksiksiz Sözdizimi Analizi Akışı:** Ham kaynak koddan anlamlı token'lara ve ardından sözdizimsel geçerliliğin kontrolüne kadar olan tüm sözdizimi analiz süreci başarıyla uygulanmıştır.
- **Gerçek Zamanlı Vurgulama ve Hata Tespiti:** Kullanıcının her tuş basışında kodun anında analiz edilip vurgulanması ve sözdizimi hatalarının eş zamanlı olarak bildirilmesi, projenin temel hedeflerinden biriydi ve başarılı bir şekilde implemente edilmiştir. Bu, geliştirici deneyimini önemli ölçüde artırmaktadır.

- **Kullanıcı Odaklı GUI:** Tkinter kullanılarak oluşturulan GUI, sezgisel bir arayüz sunar. Özellikle satır numaraları özelliğinin eklenmesi ve ana metin alanı ile senkronizasyonunun sağlanması, uygulamanın bir kod düzenleyici olarak kullanılabilirliğini artırmıştır. Hata mesajlarının satır ve sütun bilgisiyle birlikte görüntülenmesi, hata ayıklama sürecini kolaylaştırmaktadır.
- **Modüler ve Anlaşılır Tasarım:** Lexer ve Parser'ın ayrı Python dosyalarında tasarlanması, kodun modülerliğini ve her bir bileşenin sorumluluklarının netliğini sağlamıştır. Her iki bileşenin de iyi belgelenmiş ve takip edilebilir bir yapıya sahip olması, projenin sürdürülebilirliğini artırmaktadır.
- **Biçimsel Dilbilgisi Uygulaması:** MiniLang'ın EBNF ile açıkça tanımlanmış bir dilbilgisine sahip olması, projenin akademik gereksinimlerini karşılamakta ve dilin tutarlı bir şekilde işlenmesini sağlamaktadır. Operatör önceliği gibi karmaşık sözdizimi kurallarının doğru bir şekilde uygulanması, parser'ın sağlamlığını göstermektedir.

## Karşılaşılan Zorluklar ve Çözümleri:

- **Lexer'da Düzenli İfade Önceliği:** Özellikle çok karakterli operatörlerin (örn: ==) tek karakterli operatörlerden (örn: =) önce eşleştirilmesi veya FLOAT'ların NUMBER'lardan önce tanınması gibi durumlar, düzenli ifadelerin TOKEN_TYPES sözlüğündeki sıralamasının dikkatli bir şekilde yapılmasıyla çözülmüştür.
- **Parser'da Hata Kurtarma:** Basit bir recursive-descent parser'da kapsamlı hata kurtarma mekanizmaları uygulamak karmaşık olabilir. Bu projede, eat() metodunun hatalı token'ı tespit edip anlamlı bir ParserError fırlatmasıyla temel bir hata bildirimi sağlanmıştır. Daha gelişmiş hata kurtarma stratejileri (örn: panik modu) projenin mevcut kapsamı dışında tutulmuştur.
- **GUI Satır Numarası Senkronizasyonu:** Ana metin alanı ve satır numaraları alanı arasında doğru kaydırma senkronizasyonunu sağlamak, yscrollcommand ve manuel yview_moveto çağrılarının hassas bir şekilde birleştirilmesini gerektirmiştir. Bu, fare tekerleği ve manuel kaydırma çubuğu hareketleri sırasında tutarlı bir deneyim sunmak için test edilmiştir.

## Gelecekteki Geliştirmeler:

Bu proje, temel bir sözdizimi vurgulayıcı ve ayrıştırıcı için sağlam bir temel oluşturmaktadır. Gelecekte aşağıdaki özellikler eklenebilir:

- **Daha Zengin Dilbilgisi:** MiniLang'a daha fazla veri tipi (örn: Boolean), kontrol yapıları (örn: switch-case), sınıf tanımları ve modül importları gibi özellikler ekleyerek dilin yetenekleri genişletilebilir.
- **Anlamsal Analiz:** Değişken tanımlarının ve kullanımlarının doğruluğunu, tür uyumluluğunu kontrol eden bir anlamsal analiz aşaması eklenebilir.
- **Otomatik Tamamlama ve Hata Düzeltme Önerileri:** Kullanıcıya kod yazarken anlık öneriler sunan veya basit sözdizimi hataları için düzeltme önerileri getiren özellikler implemente edilebilir.

- **Hata Kurtarma Mekanizmaları:** Parser'ın bir hatadan sonra daha fazla ilerlemesine olanak tanıyan daha gelişmiş hata kurtarma stratejileri geliştirilebilir.
- **Tema Desteği:** Kullanıcıların farklı renk temaları seçebilmesi için bir tema yönetim sistemi eklenebilir.
- **Kaydetme/Yükleme İşlevselliği:** Yazılan kodun dosyalara kaydedilmesi ve dosyalardan yüklenmesi için standart bir menü ve dosya işlemleri eklenebilir.

Sonuç olarak, bu proje ile hem dilbilgisi analizi prensiplerinin teorik anlayışını hem de gerçek dünya uygulamaları için bir GUI içinde bu prensipleri pratikte implemente etme yeteneği gösterilmiştir.

# 7. MAKALE VE TANITIM VİDEOSU

- Makale için: https://medium.com/@enbostanci/ger%C3%A7ek-zamanl%C4%B1-s%C3%B6zdizimi-vurgulay%C4%B1c%C4%B1-kendi-programlama-dilinizi-yaratmak-ve-ayd%C4%B1nlatmak-92b5ece003be

- Tanıtım Videosu için: https://youtu.be/cFAiMkWEI3o

# 8. KAYNAKÇA


- Python Tkinter Dokümantasyonu: https://docs.python.org/3/library/tkinter.html

- Düzenli İfadeler (Regular Expressions) için Python re modülü Dokümantasyonu: https://docs.python.org/3/library/re.html

- EBNF(ExtendedBackus-NaurForm): https://www.freecodecamp.org/news/what-are-bnf-and-ebnf/

- Lexer: https://forum.yazbel.com/t/lexer-yapmak-karsilasacagim-hatalar-secmemi-onerdiginiz-dil-ekstralar/9226

- Sözcüksel Analiz: https://en.wikipedia.org/wiki/Lexical_analysis