metin, yazı tipi, ekran görüntüsü, grafik içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

PROGRAMLAMA DİLLERİ

Java Syntax Highlighter

22360859088

KORAY GARİP

# 1. Dil ve Dilbilgisi Seçimi

Bu proje, modern JavaScript dilini analiz edebilen bir sözdizimi vurgulayıcı ve ayrıştırıcı (parser) sistemini tarayıcı üzerinde gerçekleştirmeyi hedefler. Sistem, kullanıcı tarafından yazılan JavaScript kodlarını anlık olarak yorumlayarak sözdizimsel çözümleme ve görsel vurgulama yapar.

1.1 Neden JavaScript?

JavaScript seçmemin başlıca nedenleri şunlardır:

* Yaygınlık: Hem istemci (frontend) hem de sunucu (backend) tarafında geniş kullanım alanına sahiptir.
* Zengin Sözdizimi: if-else, switch-case, function, try-catch, return, while, for gibi birçok yapıyı içeren, ifade yönünden güçlü bir dil yapısına sahiptir.
* Çoklu Paradigma Desteği: Fonksiyonel, nesne yönelimli ve prosedürel programlama stillerini destekler.
* Belgelendirme: ECMAScript standartları üzerinden erişilebilen resmi ve açık bir dil tanımına sahiptir.

1.2 Dilbilgisi Yapısı

Kullanılan dilbilgisi, bağlamdan bağımsız dilbilgisi (Context-Free Grammar - CFG) modeline göre tanımlanmıştır. Bu CFG yapısı:

* recursive descent parser (özyinelemeli ayrıştırıcı) yöntemiyle analiz edilir,
* Her yapı için parseX() formatında fonksiyonlara bölünmüştür (örneğin parseIfStatement, parseFunctionDeclaration),
* Genişletilebilir, açık uçlu bir biçimde tasarlanmıştır.

Program ::= Statement\*

Statement ::= VarDeclaration | FunctionDeclaration | IfStatement

| WhileStatement | ForStatement | ExpressionStatement

| BlockStatement | TryStatement | ReturnStatement

| SwitchStatement | DoWhileStatement

VarDeclaration ::= ("let" | "const" | "var") IDENTIFIER ("=" Expression)? ";"?

FunctionDeclaration ::= "function" IDENTIFIER "(" ParameterList? ")" BlockStatement

IfStatement ::= "if" "(" Expression ")" Statement ("else" Statement)?

# 2. Sözdizimi Analiz Süreci

Bu sistemin temel işleyişi, yazılan JavaScript kodunu anlamlı yapılara ayırma ve görsel olarak sunma sürecine dayanır. Tüm analiz süreci aşağıdaki altı temel adımdan oluşur:

Aşamalar:

1. Kaynak Kod  
   Kullanıcının editöre girdiği ham JavaScript kodudur.
2. Lexical Analyzer (Sözcükleyici / Lexer)  
   Kaynak kod, karakter karakter taranarak anlamlı birimlere (token) ayrılır.
3. Token Stream (Token Akışı)  
   Elde edilen tüm tokenlar bir dizi hâlinde sıralanır. Her token şu bilgileri içerir:
   * Tür (örneğin: KEYWORD, NUMBER)
   * Değer (örneğin: let, 5)
   * Satır ve sütun konumu
   * Uzunluk (karakter sayısı)
4. Parser (Ayrıştırıcı)  
   Token dizisi, önceden tanımlanmış dilbilgisi kurallarına göre kontrol edilir.  
   Hatalı ya da eksik yapılar işaretlenir. Geçerli yapılar Abstract Syntax Tree (AST) şeklinde modellenir.
5. AST (Abstract Syntax Tree)  
   Kodun mantıksal ve yapısal temsilidir. Örneğin bir if yapısı, test ve consequent gibi düğümlere ayrılır.
6. Visualizer (Görselleştirici)
   * Token'lar renkli biçimde kullanıcıya sunulur.
   * AST ağaç görünümünde görselleştirilir.
   * Hatalar ayrı bir sekmede detaylı olarak listelenir.

# 3. Sözcüksel Analiz Detayları (Lexical Analysis)

Lexer (veya sözcükleyici), kaynak kodu satır satır ve karakter karakter tarayarak dilin sözdizimsel birimlerine ayrılmasını sağlar. Bu birimlere token adı verilir.

Her token, belirli bir kategoriye (tür) aittir ve bir regex (düzenli ifade) ile tanımlanır.

3.1 Token Türleri ve Açıklamaları

| Token Türü | Açıklama | Örnekler |
| --- | --- | --- |
| KEYWORD | Anahtar kelimeler | if, return, function |
| IDENTIFIER | Değişken ve fonksiyon adları | x, userName, myFunc |
| NUMBER | Sayısal ifadeler | 42, 3.14, 0xFF |
| STRING | Metin verileri | "merhaba", 'dünya', `x` |
| OPERATOR | İşlem belirten semboller | +, -, \*, =, === |
| PUNCTUATION | Yapısal işaretler | ;, {, }, (, ) |
| COMMENT | Yorum satırları | // yorum, /\* açıklama \*/ |
| UNKNOWN | Tanımlanamayan karakterler | @, #, ~ gibi hatalı semboller |

metin, ekran görüntüsü, yazılım içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.Lexer, token tanımlarını öncelik sırasına göre işler:

3.2 Token Konum Bilgileri

Her token aşağıdaki konumsal verilerle birlikte kaydedilir:

* line: Bulunduğu satır numarası
* column: Satır içindeki başlangıç sütunu
* start, end: Karakter aralığı
* length: Token uzunluğu

Bu bilgiler, hata raporlamasında ve vurgulamada kullanılır.

3.3 Hata Toleransı

Lexer tanımına uymayan karakterler UNKNOWN olarak işaretlenir ve analiz kesilmeden devam eder. Bu sayede:

* Eksik/bozuk kodlarda da analiz yapılabilir,
* Token akışı bozulmaz,
* Parser’a geçerli + hatalı yapı birlikte sunulabilir.

# 4. Ayrıştırma Yöntemi (Parsing Methodology)

Parser (ayrıştırıcı), Lexer tarafından üretilen token akışını alır ve bu tokenları, önceden tanımlanmış dilbilgisi (grammar) kurallarına göre anlamlı yapılara dönüştürür. Bu aşamanın çıktısı, Abstract Syntax Tree (AST) adı verilen ağaç yapısıdır

4.1 Recursive Descent Parser

Sistem, Top-Down Recursive Descent yaklaşımını kullanır. Her grammar kuralı için ayrı bir parsing fonksiyonu:

* Yukarıdan aşağıya (top-down) çalışır,
* Her yapı için özel bir parseX() fonksiyonuna sahiptir,
* Özyinelemeli ve modülerdir

Dilbilgisi Tanımları (Grammar - EBNF Örnekleri)

Sistemdeki grammar tanımları **EBNF (Extended Backus-Naur Form)** biçimine uygundur.

Örnek kurallar:

Program ::= Statement\*

Statement ::= VariableDeclaration | IfStatement | FunctionDeclaration | Block | ExpressionStatement

IfStatement ::= "if" "(" Expression ")" Statement ("else" Statement)?

FunctionDeclaration ::= "function" IDENTIFIER "(" ParameterList? ")" Block

VariableDeclaration ::= ("let" | "const" | "var") IDENTIFIER ("=" Expression)? ";"

Bu kurallara göre parser kodu adım adım yorumlar.

4.2 Grammar Hiyerarşisi

Precedence-based Expression Parsing:

Expression

├── AssignmentExpression (Lowest precedence)

├── ConditionalExpression (? :)

├── LogicalORExpression (||)

├── LogicalANDExpression (&&)

├── EqualityExpression (==, !=, ===, !==)

├── RelationalExpression (<, >, <=, >=)

├── AdditiveExpression (+, -)

├── MultiplicativeExpression (\*, /, %)

├── UnaryExpression (!, -, +, ++, --)

├── PostfixExpression (++ postfix, -- postfix)

└── PrimaryExpression (Highest precedence)

4.3 AST (Abstract Syntax Tree)

AST, kodun mantıksal yapısını temsil eden ağaç yapısıdır. Örneğin:

FunctionDeclaration

├── id: greet

├── params: [name]

└── body:

├── IfStatement

└── ReturnStatement

4.4 Hata Yönetimi ve Kurtarma

Parser, hata anında tamamen durmaz. Bunun yerine:

* Hatalı yapıyı addError() ile raporlar
* Böylece birden fazla hata aynı analizde tespit edilebilir

# 5. Renklendirme Sistemi (Highlighting Scheme)

Renklendirme, sözdizimsel analizden elde edilen token’ların, HTML/CSS kullanılarak görsel olarak ayrıştırılması işlemidir. Amaç, farklı türdeki kod öğelerini (anahtar kelime, sayı, değişken, vs.) kullanıcıya kolay okunur ve ayırt edilebilir şekilde sunmaktır.

5.1 Uygulama Mantığı

1. Lexer’dan elde edilen her token, token türüne göre <span> etiketiyle işaretlenir.
2. Her <span>'a ilgili CSS sınıfı (token-KEYWORD, token-STRING vb.) atanır.
3. CSS dosyasındaki stil tanımları sayesinde token’lar renklendirilir.

5.2 Renk Şeması

Kullanılan renkler, Visual Studio Code gibi modern IDE’lerin koyu temalarına benzer şekilde seçilmiştir.

| Token Türü | CSS Sınıfı | Renk Kodu | Açıklama |
| --- | --- | --- | --- |
| KEYWORD | .token-KEYWORD | #f92672 | Anahtar kelimeler |
| IDENTIFIER | .token-IDENTIFIER | #a6e22e | Değişken/fonksiyon isimleri |
| NUMBER | .token-NUMBER | #ae81ff | Sayılar |
| STRING | .token-STRING | #e6db74 | Metinler |
| COMMENT | .token-COMMENT | #75715e | Yorumlar |
| OPERATOR | .token-OPERATOR | #fd971f | +, -, \*, /, =, vb. |
| PUNCTUATION | .token-PUNCTUATION | #ffffff | Noktalama işaretleri |

5.3 Güvenlik ve Performans

* innerText yerine textContent kullanılır — XSS'e karşı güvenlidir.
* debounce() ile analiz işlemi sınırlanır, hızlı yazımda sistem yavaşlamaz.
* Tüm <span> etiketleri editörden ayrı bölgede çalışır — düzenleme etkilenmez.

# 6. Arayüz Uygulaması (GUI Implementation)

Bu projede geliştirilen grafik arayüz (GUI), analiz motorunun tüm çıktılarının kullanıcıya gerçek zamanlı ve etkileşimli şekilde sunulmasını sağlar. HTML, CSS ve JavaScript teknolojileri ile geliştirilmiş; sade, responsive ve genişletilebilir bir yapıya sahiptir.

6.1 Temel Yapı

Arayüz iki ana bölümden oluşur:

1. Kod Editörü
   * Kullanıcının JavaScript yazdığı kısımdır
   * Genellikle <textarea> ya da özel bir contentEditable alan kullanılır
   * Her tuş hareketinde analiz tetiklenir (oninput / debounce ile)
2. Sekmeli Görünüm Paneli
   * Aşağıdaki sekmeleri içerir:
     + Vurgulama → Kodun renklendirilmiş görünümü
     + Tokenlar → Token listesi (JSON formatında)
     + AST → Abstract Syntax Tree görselleştirmesi
     + Hatalar → Sözdizimi hataları listesi
   * Sekmeler arasında geçiş anında içerik güncellenir

6.2 Teknik Özellikler

| Özellik | Açıklama |
| --- | --- |
| Gerçek Zamanlı İşlem | Kod yazılırken analiz ve vurgulama otomatik yapılır (debounce) |
| HTML5 + Vanilla JS | Ek bağımlılık olmadan, sadece temel web teknolojileriyle çalışır |
| Responsive Tasarım | Mobilde ve masaüstünde uyumlu, CSS Flex/Grid ile tasarlandı |
| Hatalı Kodlarla Uyum | Kod hatalı olsa bile token ve AST üretimi devam eder |
| Ayrı Renkli Panel | Kod ve renklendirme ayrı kutularda sunulur — bu sayede eğitim odaklıdır |
| Hafif & Hızlı | DOM yeniden kullanılabilir yapılmıştır, tek satır değişimden sonra sadece gerekli alanlar güncellenir |

Kod analizinin hemen yanında küçük bir panelde şu bilgiler görüntülenebilir:

* Toplam Token sayısı
* Toplam AST düğümü sayısı
* Hata sayısı
* Kod karakter uzunluğu

# 7. Sonuç ve Genişletme Önerileri

Bu proje ile:

* Kodun sözdizimsel yapısı çözümlenebilir hale gelmiştir
* Hatalar kullanıcıya konumlu ve görsel şekilde sunulmuştur
* AST üretimi, eğitim ve analiz için örnek teşkil etmektedir

Gelecek Genişletmeler:

* Görsel AST editörü (sürükle-bırak)
* JSON çıktısı ile entegrasyon
* Otomatik test üretici modül
* Kod tamamlama (IntelliSense benzeri)