Matematiksel İfade Çözümleyici

1. Proje Kapsamı

1.1 Açıklama

Bu doküman aşağıdaki özellikleri içeren bir uygulamayı açıklamaktadır.

1. Formal bir dil oluşturulmuştur.
2. Bu dil toplama, çıkarma, çarpma, bölme, faktöriyel, üs alma, sin-cos fonksiyonlarını ve parantezleri içerir.
3. İşlemler tam sayılar, negatif sayılar ve ondalıklı sayıları destekler.
4. Kullanıcı tarafından girilen ifade bu dil için tokenlere ayrılır.
5. Tokenler kullanıcıya gösterilir.
6. İfade parse edilir, sözdiziminde ya da anlamsal olarak hata varsa kullanıcıya gösterilir.
7. Parse ağacı kullanıcıya gösterilir.
8. Parse edilen ifade yorumlanarak(interpret) sonuç gösterilir.

1.2 İşlem Öncelikleri

Matematiksel ifadelerin işlenmesinde aşağıdaki öncelik sıralaması[1] esas alınmıştır.

a) Genel Öncelik Sırası

1. Fonksiyon adları (sin cos)
2. Faktöriyel operatörü (!)
3. Üs alma operatörü (^)
4. Çarpma ve bölme operatörleri (* /)
5. Toplama ve çıkarma operatörleri (+ -)

b) Operatörler Arası Seviyeler

- Unary + ve - operatörü * ve / operatörlerinden önceliklidir.
- Farklı parantez seviyeleri arasında, içteki parantez dıştakine göre daha yüksek önceliğe sahiptir.
- Aynı parantez seviyesindeki operatörler soldan sağa doğru işlenir.

1.3 Kısıtlamalar ve Özel Durumlar

- Üs operatörü sağdan sola doğru işlenir. Örnek: 2^3^4 => 2^(3^4)
- Faktöriyel operatörünün operandı yalnızca pozitif ve tam sayılar olabilir.
- Bölme operatörünün sağ operandı sıfır (0) olamaz.

Dili sade tutmak amacıyla yukarıdaki kısıtlar gramerde serbest bırakılmış ancak parser ve interpreter'da hata olarak işlenmiştir. Binaenaleyh bu dil bağlamdan bağımsız bir dildir.

2. Dil

2.1 Gramer

Gramer notasyonunda CFG(Context Free Grammar)'nin **EBNF**(Extended Backus-Naur Form) varyantı kullanılmıştır.

Operatör önceliği gramer yapısında her bir ifade türünün daha detaylı bileşenlere ayrılarak alt seviyede tanımlanmasıyla sağlanmıştır. Öncelik sıralaması, daha karmaşık yapılardan daha basit yapılara doğru (örneğin, <expression> → <term> → <power> → <factorial> → <function> → <primary>) hiyerarşik bir şekilde belirlenmiştir. Bu **hiyerarşi**, her işlem grubunun gramerde yalnızca kendinden bir üst seviyede kullanılabilmesini sağlayarak **operatör önceliğini doğal bir şekilde uygular**. Örneğin, 2 + 3 * 4 ifadesinde çarpma işlemi önce yapılır çünkü <term> seviyesindeki çarpma, <expression> seviyesindeki toplama işleminden daha önce değerlendirilir.

```
G = {Σ, T, V, P, S}
V = {expression, term, power, factorial, function, primary,
number, digit} ⊆ Σ
T = {sin, cos, +, −, ∗, /, ^, !, (, ), ., 0, 1, 2, 3, 4, 5,
6, 7, 8, 9} ⊆ Σ
Σ = T ∪ V
S = expression
P:
<expression>  ::= <term> (("+" | "−") <term>)∗
<term>        ::= <power> (("∗" | "/" | ε) <power>)∗
<power>       ::= <factorial> ("^" <power>)∗
<factorial>   ::= <function> ("!")∗
<function>    ::= "sin" "(" <expression> ")"
                |   "cos" "(" <expression> ")"
                |   <primary>
<primary>     ::= <number>
                |   "(" <expression> ")"
<number>      ::= ["−"]<digit>{<digit>}["."<digit>{<digit>}]
<digit>       ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

2.2 Gramer Açıklaması

1. **<expression> ::= <term> (("+" | "-") <term>)***
    - Bir ifade, bir terimle başlar ve ardından sıfır veya daha fazla toplama veya çıkarma işlemi gelir.
    - Örnek: 5 + 3 - 2
2. **<term> ::= <power> (("*" | "/" | ε) <power>)***
    - Bir terim, bir üs ifadesi (<power>) ile başlar ve ardından sıfır veya daha fazla çarpma (*) ya da bölme (/) işlemi gelir.
    - ε örtülü (implicit) çarpmayı temsil eder: 2(3) => 2*3, 2cos(60)=>2*cos(60)
3. **<power> ::= <factorial> ("^" <power>)***

    - Bir üs ifadesi, bir faktöriyel ifadesiyle başlar ve ardından sıfır veya daha fazla üs alma işlemi gelir.
    - Örnek: 2 ^ 3, 3^2^1
4. **<factorial> ::= <function> (“!”)***
    - Bir faktöriyel ifadesi, bir fonksiyon ifadesiyle başlar ve ardından sıfır veya daha fazla faktöriyel işlemi gelir.
    - Örnek: 5!, 3!!, 5!!!
5. **<function> ::= "sin" "(" <expression> ")" | "cos" "(" <expression> ")" | <primary>**
    - Bir fonksiyon ifadesi, sin veya cos fonksiyonlarından biriyle başlar, ardından bir parantez içinde bir ifade gelir veya bir primary ifade olabilir.
    - Örnek: sin(30), cos(45)
6. **<primary> ::= <number> | "(" <expression> ")"**
    - Bir birincil ifade, bir sayı veya parantez içinde bir ifade olabilir.
    - Örnek: 42, (3 + 4)
7. **<number> ::= ["-"] <digit> {<digit>} ["." <digit> {<digit>}]**
    - Bir sayı, negatif işaretle başlayabilir.
    - Ardından bir veya daha fazla rakam gelir.
    - Bir ondalık nokta ve bir veya daha fazla rakamla devam edebilir.
    - Örnek: 123, -45.67
8. **<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"**
    - Bir rakam, 0'dan 9'a kadar olan herhangi bir sayı

2.3 Gramerin Belirsizliği (Ambiguity)
Gramerde iki noktada belirsizlik ortaya çıkabilir:

1. Örtük çarpma durumu: <term> ::= <power> (("*" | "/" | ε) <power>)*
   Burada ε (epsilon) ile gösterilen implicit çarpma, örneğin "2(3)" veya "2sin(x)" gibi ifadelerde belirsizliğe yol açabilir.
2. Üs işlemlerinde sağdan birleşme:
   <power> ::= <factorial> ("^" <power>)*
   “2^3^4” gibi ifadelerde belirsizlik oluşabilir.

Ancak **manuel olarak geliştirilen** Recursive Descent Parser ile bu belirsizlikler kolayca tolere edilebilir düzeydedir.

2.4 Gramerin Sol Özyinelemeliliği (Left Recursion)
Gramerde doğrudan sol rekürsiyon bulunmuyor, dolaylı sol rekürsiyonlar ise implementasyonda belirli koşullar altında çalıştırılarak kontrol altında tutulabilir ve sonsuz döngünün oluşmasının önüne geçebilir şekildedir.

3. Uygulama

Arayüz → Matematiksel İfade → (Lexer:Tokens → Parser:AST → Interpreter:number) → Arayüz → Görselleştirme

- Uygulama tamamen tarayıcıda çalışacak şekilde geliştirilmiştir.
- Girdi metni anlık olarak değerlendirilir.
- **Hataların yakalanması** Lexer, Parser ve Interpreter düzeyinde sağlanmıştır.
- Lexing ve parsing işlemleri **O(n) karmaşıklığında** gerçekleştirilir.

3.1 Kullanılan Teknolojiler

Uygulama **ReactJS** kütüphanesiyle **TypeScript** dilinde kodlanmış ve **Tailwind CSS** ile stilize edilmiştir. TypeScript tip güvenliği sağlayan ve JavaScript'e transcompile edilen bir dildir.

3.2 Tarayıcı (Scanner - Lexer)

Lexer, girdi metnini token'lara ayırarak kaynak kodu daha küçük ve anlamlı parçalara böler. Sayılar, operatörler, parantezler ve fonksiyon isimleri gibi belirli karakter dizilerini tanır ve bunları uygun token türlerine dönüştürür. Beklenmeyen karakterlerle karşılaştığında hata mesajları üretir.

- Kaynak kodda **Lexer.ts**, girdi metnini daha küçük ve anlamlı parçalara böler. Bu parçalar token olarak adlandırılır ve **Token arayüzü** ile temsil edilir.
- Lexer, **tokenize()** metodunda girdi metnini yalnızca bir kez tarar ve token'lara ayrıştırır. Bu, performansı artırır ve bellek kullanımını azaltır.
- Lexer, sayılar (number() metodu), operatörler (scanToken() metodundaki switch ifadesi), parantezler ve fonksiyon isimleri (**identifier()** metodu) gibi belirli karakter dizilerini tanır ve bunları uygun token türlerine dönüştürür.
- Lexer, boşlukları, satır sonlarını ve diğer **boşluk karakterlerini göz ardı eder**, böylece yalnızca anlamlı token'lar üretilir.
- Lexer, beklenmeyen karakterlerle karşılaştığında (scanToken() metodundaki default durumu) hata mesajları üretir.

3.3 Ayrıştırıcı (Parser)

Parser, **top-down parsing** yaklaşımının bir türü olan **Recursive Descent Parsing** tekniğini kullanır. RDP **sol özyinelemeli olmayan** gramerlerle çalışır, yani bir kuralın sol tarafında kendisine **doğrudan** referans verilmez. Bu, parser'ın sonsuz döngüye girmesini önler ve ayrıştırma işlemini daha verimli hale getirir.

- Kaynak kodda **Parser.ts** dosyasında her gramer kuralı için bir ayrıştırma fonksiyonu tanımlanmıştır (**expression**(), **term**(), **power**(), **factorial**(), **function**(), **primary**()).

- Bu fonksiyonlar, girdi token'larını analiz ederek ilgili tipte bir ASTNode oluşturur.
- Temel bir **ASTNode** arayüzü tanımlanmış ve **NumberNode**, **BinaryNode**, **UnaryNode**, **FunctionNode** bu arayüzden türetilmiştir.
- Böylece gramer kurallarına karşılık gelen fonksiyonların tamamı ASTNode türünden dönüş değerine sahip olabilmiştir.
- Parser, **match**() ve **check**() gibi yardımcı fonksiyonlar kullanarak bir sonraki token'a bakar ve doğru ayrıştırma fonksiyonunu çağırır.

3.4 Yorumlayıcı (Interpreter)
Kaynak kodda **Interpreter.ts**, Parser tarafından oluşturulan AST düğümlerini değerlendirerek matematiksel ifadeleri hesaplar. **interpret** metodu, AST'nin kök düğümünü alır ve **evaluate** metodunu çağırarak düğüm türüne göre uygun değerlendirme metodunu seçer. **evaluateBinary** metodu iki operandlı işlemleri, **evaluateUnary** metodu tek operandlı işlemleri ve **evaluateFunction** metodu fonksiyon çağrılarını değerlendirir. Her bir düğüm türü için uygun hesaplamalar yapılır ve sonuç döndürülür.

Not: JavaScript **eval** fonksiyonu tarayıcılar tarafından güvenli bulunmadığından AST'nin JavaScript sözdizimine çevrilip çalıştırılmasından kaçınılmış; bunun yerine tüm AST düğümleri analiz edilerek gömülü matematiksel fonksiyonlar çalıştırılmıştır.

3.2 Kaynak Kodlar ve Erişim
Kaynak kod: https://github.com/sirridemirtas/MathExpressionParser
Uygulama: https://mathexpressionparser.onrender.com/

4. Kaynakça

[1]: Alavi Milani, M. M. R., 2015. *Design and applications of grammar-based methodologies for automatic generation and step-by-step solving of mathematical expressions*. Yayımlanmamış doktora tezi, Karadeniz Teknik Üniversitesi. 3.2.1.3. Conventional structure of algebraic expressions