1.parseExpression() は 2 つのパートに分けることが可能である。それぞれのパー トの処理内容に関してまとめよ。

Part(1)―中間演算子以外のトークンを処理（必ず実行）

前置演算子の処理

識別子、リテラルを対応するデータ構造に変換

必ず実行される部分

Part(2)―中間（２項）演算子を処理（forの条件を満足する間）

For文による繰り返し処理；実行されない場合もある。

優先順位が高い場合のみ実行

2. parseInfixExpression() による中置演算子構造体の生成に関してまとめよ。

中間演算子のノードを生成、left,rightがあり、leftは関数の引数として与えられる。

RightはparseExpression()の呼び出しにより生成。

3. parseExpression() に渡す引数 (precedence) の働きに関してまとめよ

演算子の優先順位の制御:

precedence 引数は、与えられた式内の演算子の優先順位を制御します。優先順位は、演算子がどの程度結合するかを示し、式の評価順序に影響します。

式を解析する際に、precedence 引数は現在の演算子の優先順位を決定し、その演算子の左右のオペランドを正しく結合します。

演算子の結合方向の考慮:

演算子の優先順位には、結合方向も含まれます。例えば、演算子が左結合性を持つ場合、左のオペランドから順に結合されます。右結合性を持つ場合、右のオペランドから結合されます。

precedence 引数は、演算子の結合方向を正しく扱うために使用されます。優先順位が同じ場合、結合方向によって演算子のオペランドの組み合わせが変わります。

再帰的な構造の管理:

parseExpression() 関数は再帰的に呼び出され、式全体が分割されます。この際、各部分式の演算子は、その部分式の precedence よりも高い優先順位を持つ場合にのみ、その部分式内で結合されます。

precedence 引数は、再帰的な構造を管理し、式の部分式が適切に結合されることを保証します。

AST (Abstract Syntax Tree) の構築:

引数 precedence は、AST の構築プロセスにも影響を与えます。優先順位に基づいて式が正しく解析され、適切な AST ノードが構築されます。

総括すると、parseExpression() 関数に渡される引数 precedence は、式の演算子の優先順位と結合方向を制御し、再帰的な構造の管理を行います。これにより、与えられた式が適切に解析され、正確な意味を持つ AST が生成されます。

4. parseExpression() の引数のデフォルトを LOWEST よりも優先順位が高いもの (例えば SUM) に変更すると何が起きるか。

`parseExpression()` の引数のデフォルトを LOWEST よりも優先順位が高いもの（例えば SUM）に変更すると、次のような影響があります：

1. \*\*演算子の結合順序が変わる\*\*:

- `parseExpression()` の引数 `precedence` を SUM などより高い優先順位の演算子に設定すると、その演算子が他の演算子よりも優先されます。これにより、その演算子が最初に評価され、その結果が他の演算子のオペランドとして使用されます。

- 例えば、`3 + 4 \* 2` のような式では、通常は乗算が加算よりも優先されますが、引数 `precedence` を SUM に設定すると、加算が優先されます。

2. \*\*式の評価順序が変わる\*\*:

- 優先順位が高い演算子が先に評価されるため、式全体の評価順序が変わります。これは、意図しない結果をもたらす可能性があります。

- 例えば、`3 + 4 \* 2` の場合、通常は乗算が先に評価されますが、引数 `precedence` を SUM に設定すると、加算が先に評価され、結果が異なる可能性があります。

3. \*\*構文解析の制御が変わる\*\*:

- 優先順位の変更により、`parseExpression()` 関数内での構文解析の制御が変わります。より高い優先順位の演算子に遭遇した場合、その演算子に対する構文解析が優先されます。

- これは、AST (Abstract Syntax Tree) の構築や、式の意味論的な解釈に影響を与える可能性があります。

したがって、引数のデフォルトを LOWEST よりも優先順位が高い演算子に変更すると、式の解析と評価に影響を与える可能性があります。プログラムの意図に合わせて、適切な優先順位を設定する必要があります。

5. 以下の式を parseExpression() で解析する際の、関数 (メソッド) 呼び出しの関 係を示せ。 – -a + b \* 3 – a + 5 + -10 – 3 + 5 \* 2 \* 3 – 3 + 5 \* 2 \* 3 + 10 例: -a + b \* 3 parseExpression(LOWEST) → parsePrefixExpression() → parseExpression(PREFIX) → parseInfixExpression(left) → parseExpression(SUM) → parseInfixExpression(left) → parseExpression(PRODUCT)

以下は、各式を `parseExpression()` で解析する際の関数（メソッド）呼び出しの関係を示します：

1. `-a + b \* 3`:

- `parseExpression(LOWEST)`

→ `parsePrefixExpression()` (単項演算子の解析)

→ `parseExpression(PREFIX)` (単項演算子のオペランドの解析)

→ `parseInfixExpression(left)` (2項演算子の解析)

→ `parseExpression(SUM)` (加算の解析)

→ `parseInfixExpression(left)` (2項演算子の解析)

→ `parseExpression(PRODUCT)` (乗算の解析)

2. `a + 5 + -10`:

- `parseExpression(LOWEST)`

→ `parseExpression(SUM)` (加算の解析)

→ `parseInfixExpression(left)` (2項演算子の解析)

→ `parseExpression(LOWEST)` (再帰的に `parseExpression()` を呼び出す)

→ `parseExpression(SUM)` (加算の解析)

→ `parseInfixExpression(left)` (2項演算子の解析)

→ `parseExpression(PREFIX)` (単項演算子の解析)

3. `3 + 5 \* 2 \* 3`:

- `parseExpression(LOWEST)`

→ `parseExpression(SUM)` (加算の解析)

→ `parseInfixExpression(left)` (2項演算子の解析)

→ `parseExpression(PRODUCT)` (乗算の解析)

→ `parseInfixExpression(left)` (2項演算子の解析)

→ `parseExpression(PRODUCT)` (乗算の解析)

4. `3 + 5 \* 2 \* 3 + 10`:

- `parseExpression(LOWEST)`

→ `parseExpression(SUM)` (加算の解析)

→ `parseInfixExpression(left)` (2項演算子の解析)

→ `parseExpression(PRODUCT)` (乗算の解析)

→ `parseInfixExpression(left)` (2項演算子の解析)

→ `parseExpression(PRODUCT)` (乗算の解析)

→ `parseInfixExpression(left)` (2項演算子の解析)

→ `parseExpression(SUM)` (加算の解析)

これらの関数呼び出しの関係により、与えられた式が `parseExpression()` を通じて適切に解析され、AST (Abstract Syntax Tree) が構築されます。