言語処理系
構文解析 (Parsing) その 2

西川 清史 (電子情報システム工学科)

2.5 return 文の構文解析

- return 文の構文解析
  - 文法の確認
  - ReturnStatement 構造体の定義
  - parseStatement() メソッドの修正
  - parseReturnStatement() メソッドの追加
- let 文と同じ処理
  - キーワード (return) のあとのトークンを調べる
  - 左から右へ走査 (left-to-right scan)

return 文の例

Monkey 言語の return 文の例

```
return 5;
return 10;
return add(15);
```

- return 文は return と式のみから構成される

return 文の構造

```
return <expression>;
```

ReturnStatement 構造体の定義 (p.45)

ast/ast.go

```
type ReturnStatement struct {
    Token token.Token // the 'return' token
    ReturnValue Expression
}
func (rs *ReturnStatement) statementNode() {}
func (rs *ReturnStatement) TokenLiteral() string { return rs.Token.Literal }
```

- ReturnStatement 構造体
  - Node および Statement インターフェースを実装する

LetStatement 構造体と ReturnStatement 構造体

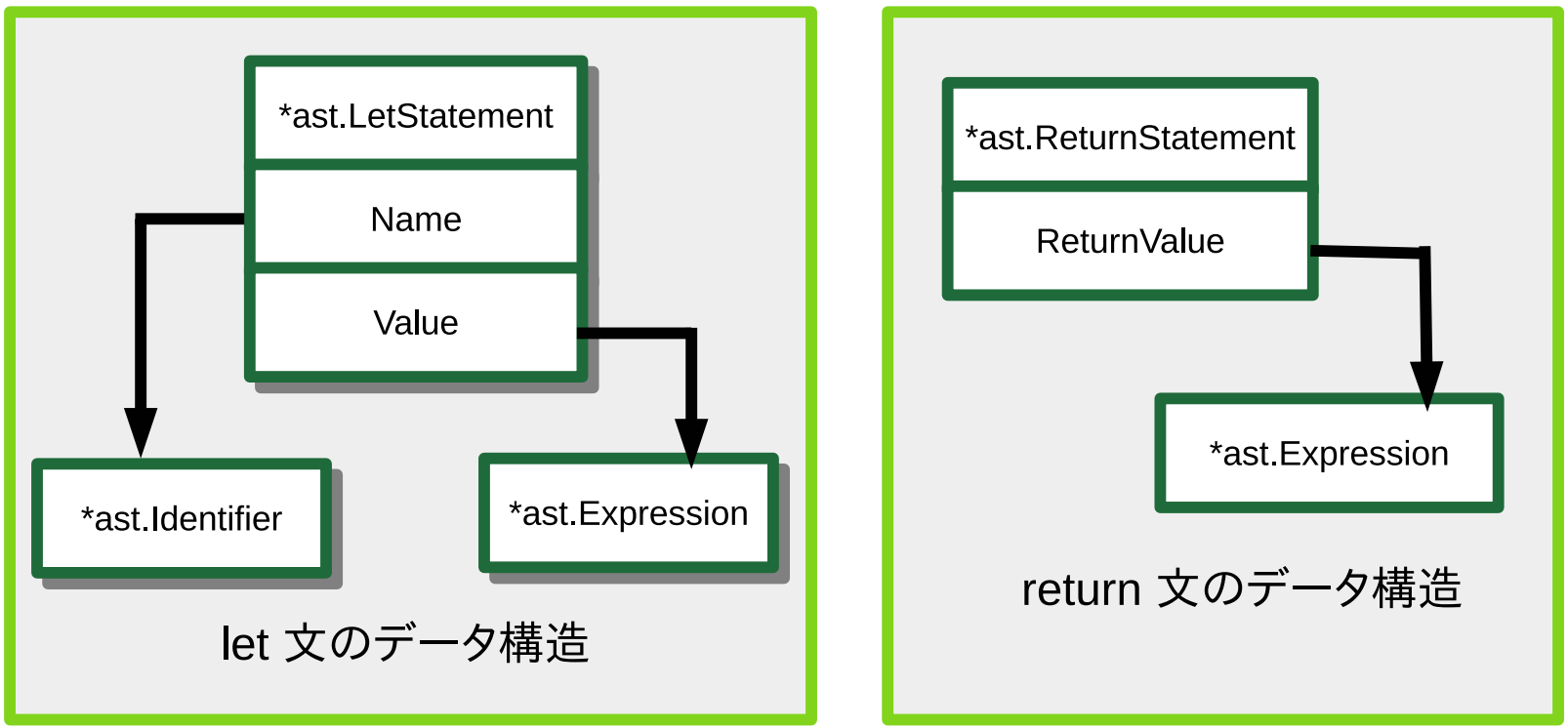

Figure: 文の形の違いのデータ構造による表現

parseStatement メソッドの変更

parser/parse.go

```
func (p *Parser) parseStatement() ast.Statement {
    switch p.curToken.Type {
    case token.LET:
        return p.parseLetStatement()
    case token.RETURN:
        return p.parseReturnStatement()
    default:
        return nil
    }
}
```

parseReturnStatement (p.46)

parser/parser.go

```
 func (p *Parser) parseReturnStatement() *ast.ReturnStatement {
    stmt := &ast.ReturnStatement{Token: p.curToken}

    p.nextToken()

    // TODO: We're skipping the expressions until we
    // encounter a semicolon
    for !p.curTokenIs(token.SEMICOLON) {
        p.nextToken()
    }

    return stmt
}
```

- return 文の処理
  - 右辺の式 (expression) は呼び飛ばしている (TODO の部分)

文の処理まとめ

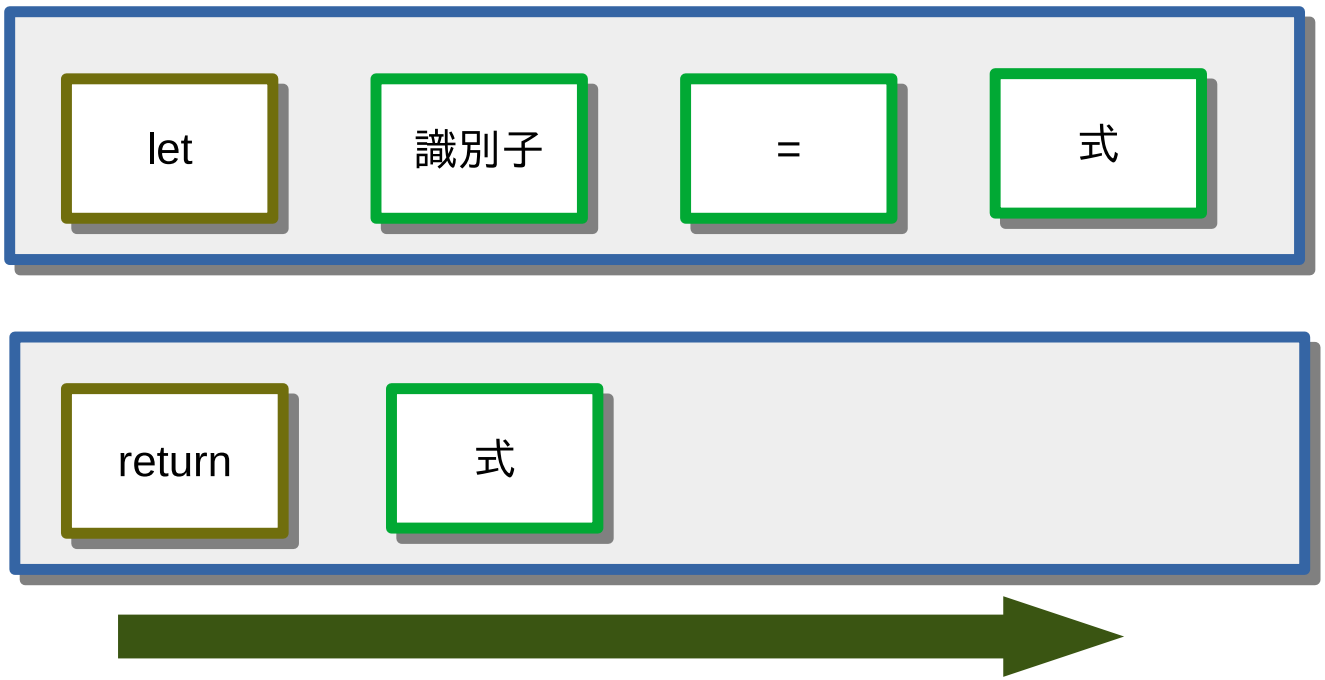

Figure: 文 (statement) の処理のまとめ

文の処理とメソッド

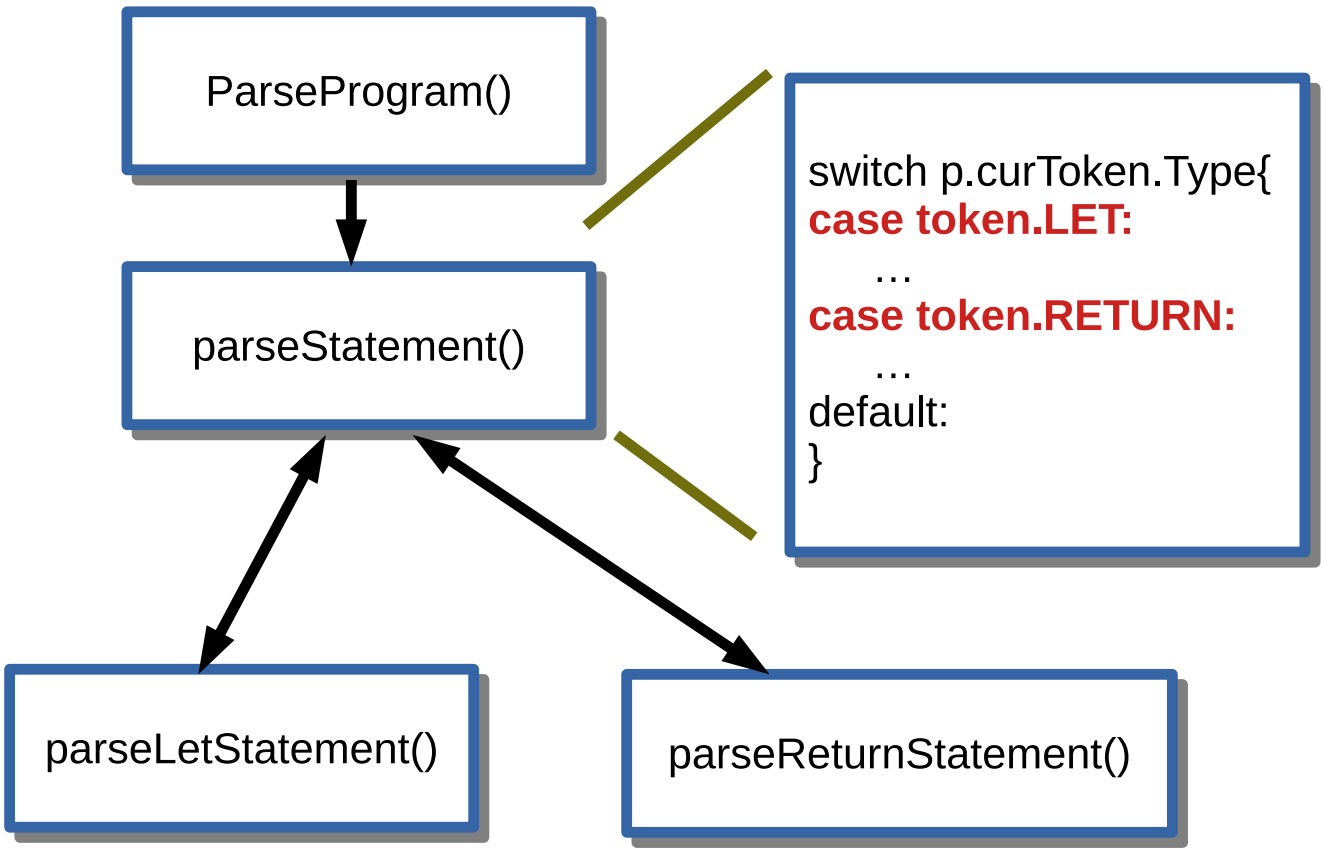

Figure: 文 (statement) の処理の流れ

2.6 式の構文解析 (p.47)

- 文 (statement) の構文解析
  - 比較的理解しやすい
  - トークンを「左から右へ」処理すれば良い
- 式 (expression) の構文解析の難しさ
  - 演算子の優先順位
    - `5*5+10`
    - `5*(5+10)`
  - 同一のトークンを複数の位置に使用することが可能
    - `-5-10`
    - `5*(add(2,3) + 10)`

2.6.1 Monkey における式

前置演算子を使った式

```
-5
!true
!false
```

中置演算子 (二項演算子) を使った式

```
5+5
5-5
5/5
5*5
```

2.6.1 Monkey における式

比較演算子を使った式

```
foo == bar
foo != bar
foo < bar
foo > bar
```

丸括弧を使った式のグループ化

```
5 * (5 + 5)
((5 + 5) * 5) * 5
```

Monkey における式

呼び出し式

```
add(2, 3)
add(add(2, 3), add(5, 10))
max(5, add(5, (5 * 5)))
```

関数リテラル

```
fn(x, y) { return x + y} (5, 5)
(fn(x) { return x} (5) + 10 ) * 10
```

if 式

```
let result = if (10 > 5) { true } else { false };
```

2.6.2 トップダウン演算子順位解析 (Pratt 構文解析)

- ここでの目的
  - 式の構文解析を行い構文木を作成する
- アルゴリズム
  - トップダウン演算子順位解析 (Pratt 構文解析)
  - Vaughan Pratt “Top Down Operator Precedence”
  - トップダウン構文解析法
- Pratt 構文解析
  - 文脈自由文法やバッカスナウア記法に基づく構文解析とは異なる手法
  - 文法ルールに構文解析関数 (`parseLetStatement` などのような) を関連付けない
  - 代わりに各トークンタイプに関数を関連つける
    - 各トークンタイプに対して 2 つの関数が関連付けられる

2.6.3 用語

- **前置演算子** (prefix operator)
  - オペランド (演算対象) の「前」に「置」かれる演算子
  - 例: `--5`
- **後置演算子** (postfix operator)
  - オペランドの「後」に「置」かれる演算子
  - 例: `foobar++`
  - `Monkey` インタプリタには後置演算子はない
- **中置演算子** (infix operator)
  - オペランドの間に置かれる演算子
  - 例: `5 * 8`
  - **二項演算子式** (binary expressions) に現れる
- **演算子の優先順位** (operator precedence, order of operations)
  - 異なる演算子がどの優先順位を持っているかを表す
  - `5 + 5 * 10`

2.6.4 AST の準備

- `Monkey` のプログラム
  - 一連の文の集まり
  - `let` 文、`return` 文
- AST に 3 番目の文の種類を追加する
  - 式文 (expression statements)
  - 1 つの式からなる文
    - `let x = 5;`
    - `x + 10;` // 式文の例

ExpressionStatement 構造体 (p.52)

ast/ast.go

```
type ExpressionStatement struct {
    Token token.Token // the first token of the expression
    Expression Expression
}
func (es *ExpressionStatement) statementNode()  {}
func (es *ExpressionStatement) TokenLiteral() string { return es.Token.Literal }
```

- 式文 (expression statement) を表すための構造体
  - Token フィールド ··· すべてのノードに存在する
  - Expression フィールド ··· 式そのものを保持する
- Statement インターフェースを実装する
  - Statement として処理が可能

式文の処理とメソッドの関係

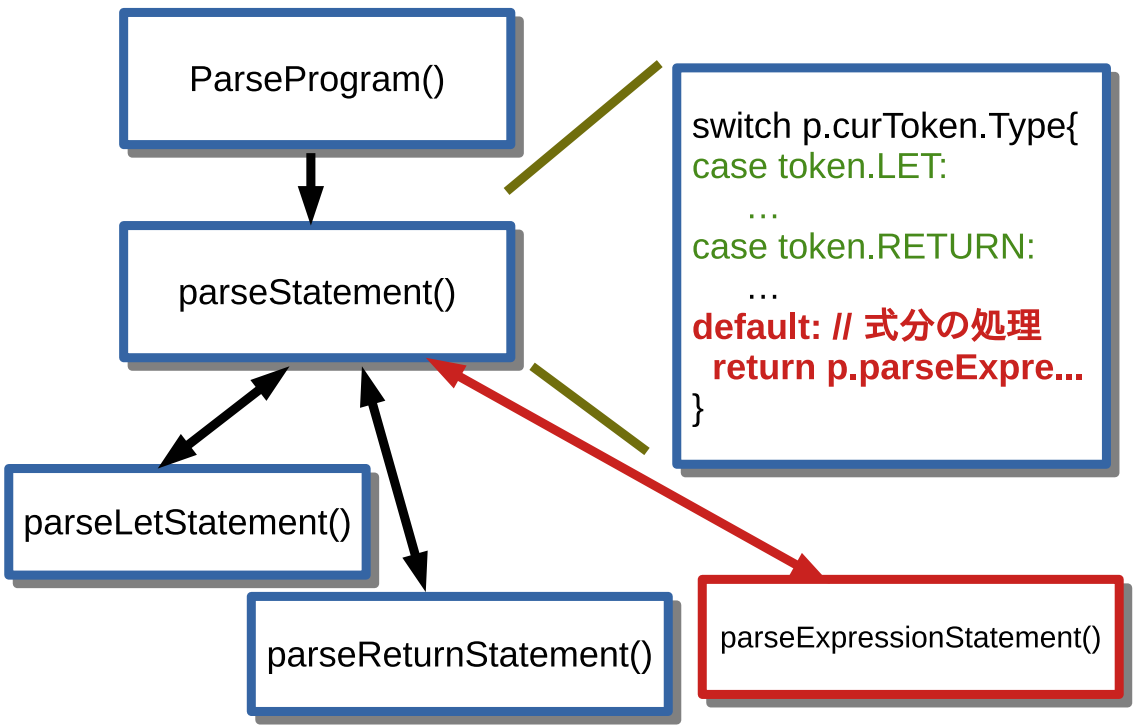

Figure: `parseExpressionStatement()` メソッドによる式文の処理

String() メソッドの AST ノードへの追加 (p.52)

ast/ast.go

```
type Node interface {
    TokenLiteral() string
    String() string      // 追加
}
```

- Node インターフェースのメソッドセットに String() を追加
  - すべてのノードが String() メソッドを実装する必要がある
  - *ast.Program の String() メソッドを呼び出すとプログラム全体を文字列として復元可能となる

これからやることの概要

- 構文解析による構文木の生成
  - Pratt 構文解析
  - 式 (expression) の構文解析
- 概要
  - 式の構成要素のデータ構造
    - 識別子、整数リテラル、前置演算子、中置演算子
  - 式と構文木とデータ構造

識別子のデータ構造

ast/ast.go (p.33)

```
type Identifier struct {
    Token token.Token // the token.IDENT token
    Value string
}
func (i *Identifier) expressionNode() {}
func (i *Identifier) TokenLiteral() string { return i.Token.Literal }
func (i *Identifier) String() string { return i.Value }
```

- Identifier 構造体
  - Expression インターフェースを実装する
  - Expression の一種として扱える

整数リテラルのデータ構造

ast/ast.go (p.33)

```
type IntegerLiteral struct {
    Token token.Token
    Value int64
}
func (il *IntegerLiteral) expressionNode() {}
func (il *IntegerLiteral) TokenLiteral() string { return il.Token.Literal }
func (il *IntegerLiteral) String() string { return il.Token.Literal }
```

- IntegerLiteral 構造体
  - Expression インターフェースを実装する
  - Expression の一種として扱える

式とデータ構造 (識別子と整数リテラル)

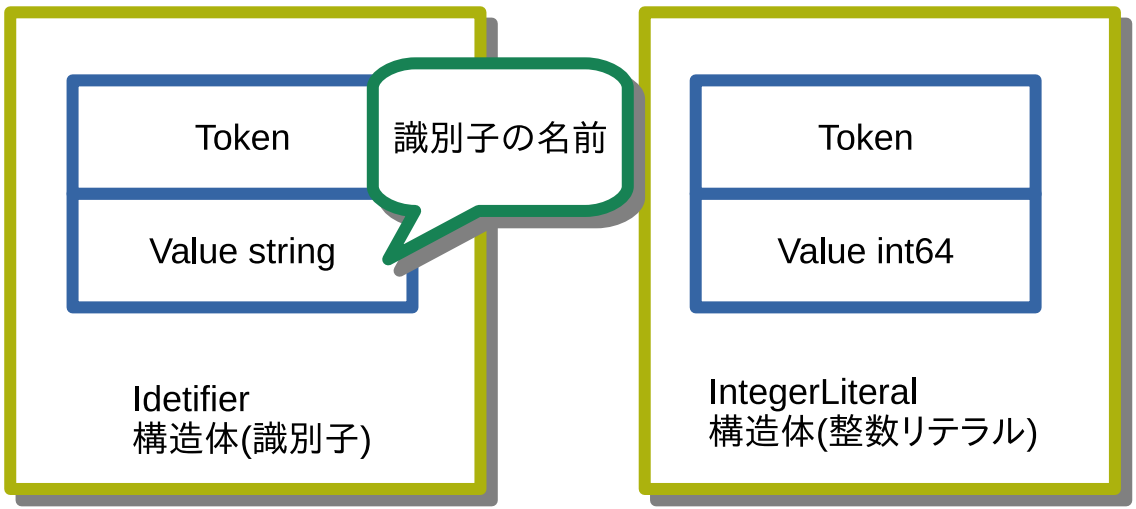

Figure: 識別子と整数リテラルのデータ構造

- トークンと値
  - 識別子の値は文字列で「名前」を保持
  - 整数リテラルの値は `int64` で表現

変数の値は？

- 構文解析
  - プログラムの構造を解析することが目的
  - 変数の領域割当や、値の管理などは行わない
- 評価 (evaluation) ··· 3 章
  - 構文木を評価しプログラムを実行
  - 変数は領域を割り当てられ、値を代入される

前置演算子の構造体

ast/ast.go (p.65)

```
type PrefixExpression struct {
    Token    token.Token // The prefix token, e.g. !
    Operator    string
    Right    Expression
}
func (pe *PrefixExpression) expressionNode() {}
func (pe *PrefixExpression) TokenLiteral() string { return pe.Token.Literal }
func (pe *PrefixExpression) String() string {
    var out bytes.Buffer
    out.WriteString("(")
    out.WriteString(pe.Operator)
    out.WriteString(pe.Right.String())
    out.WriteString(")")
    return out.String()
}
```

中置演算子の構造体

ast/ast.go (p.70)

```
type InfixExpression struct {
    Token    token.Token // The operator token, e.g. +
    Left     Expression
    Operator string
    Right    Expression
}
func (ie *InfixExpression) expressionNode(){}
func (ie *InfixExpression) TokenLiteral() string { return ie.Token.Literal }
func (ie *InfixExpression) String() string {
    var out bytes.Buffer
    out.WriteString("(")
    out.WriteString(ie.Left.String())
    out.WriteString(" " + ie.Operator + " ")
    out.WriteString(ie.Right.String())
    out.WriteString(")")
    return out.String()
}
```

式とデータ構造 (前置演算子と中置演算子)

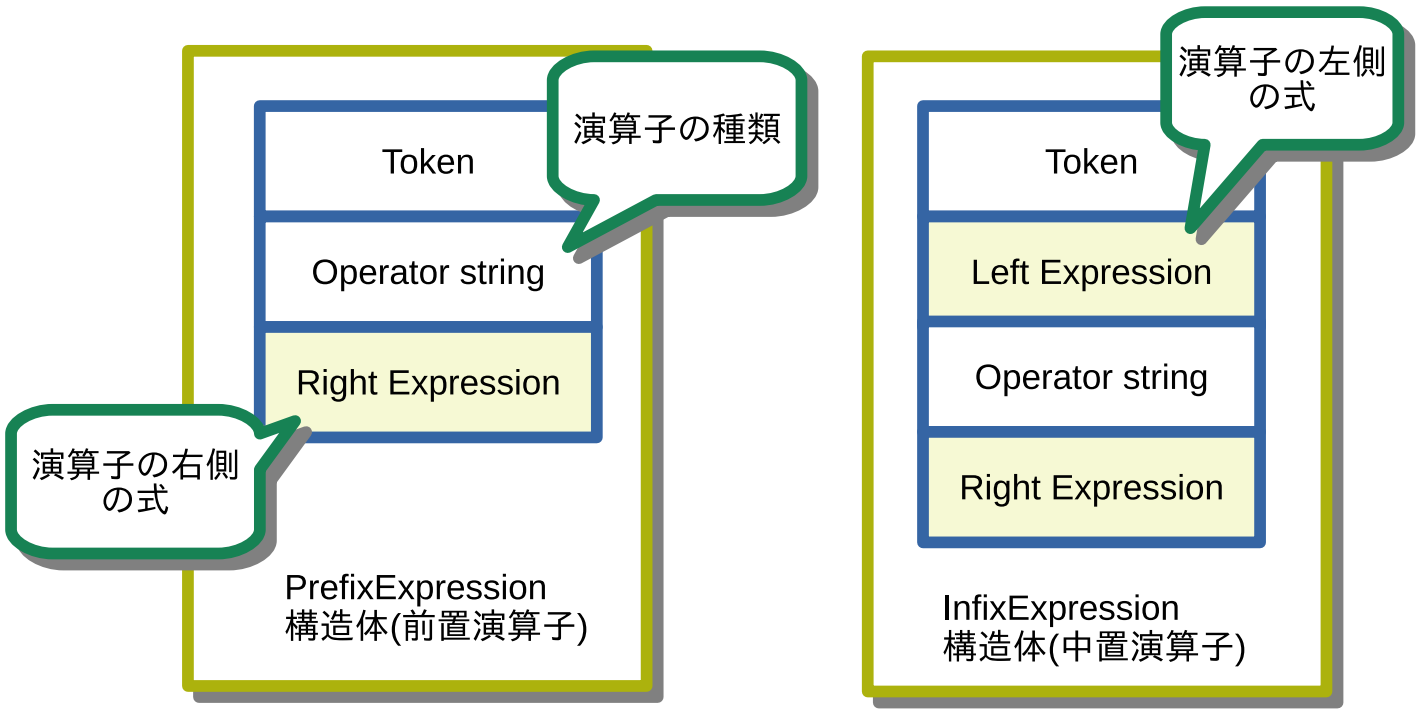

Figure: 演算子のデータ構造

- `Expression` インターフェースを実装

式を構成するデータ構造 (2.6 章)

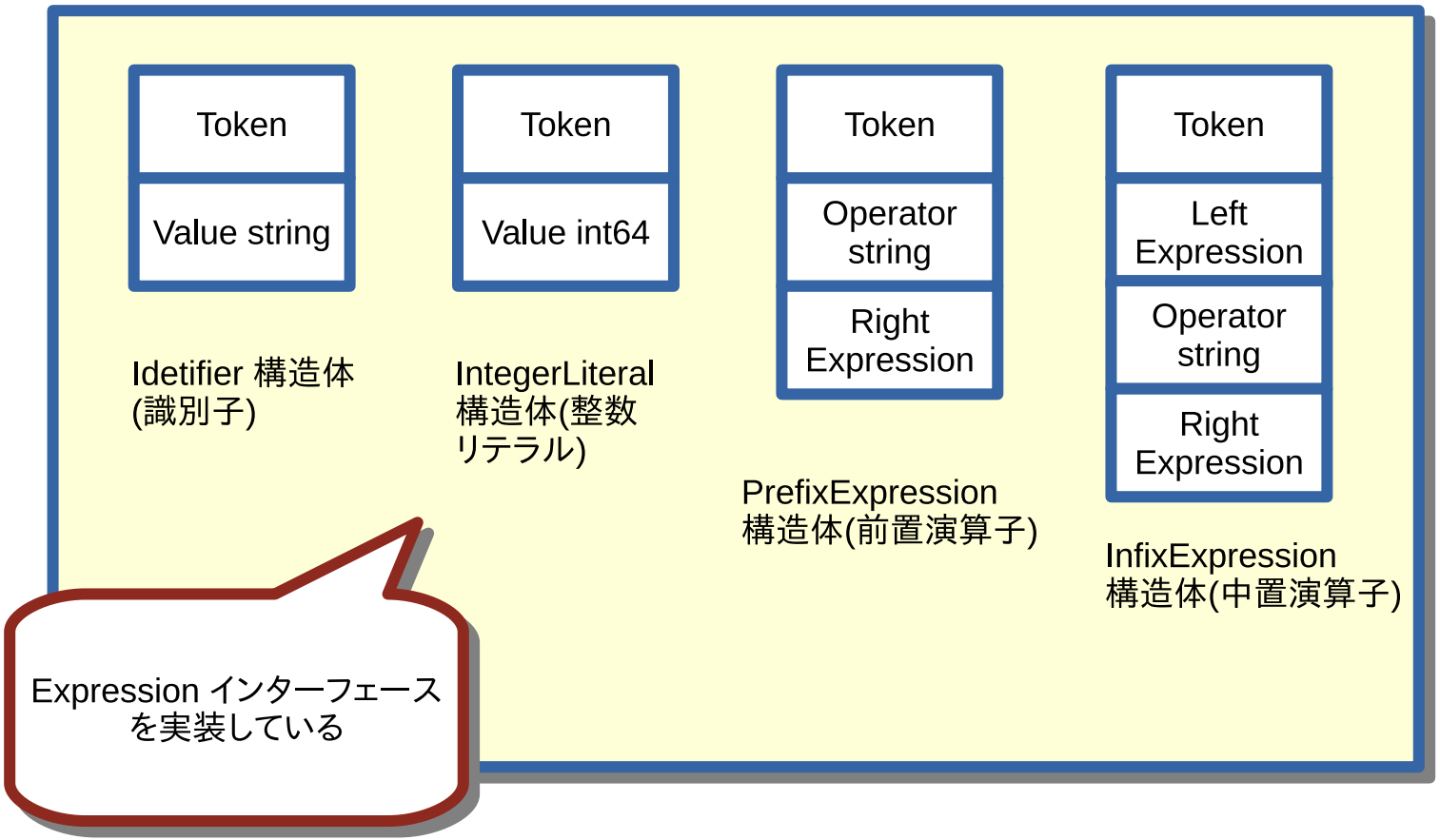

Figure: 式のデータ構造の違い

式 (expression) の表現

- 式
  - 識別子、整数リテラル、前置演算子、中置演算子の結合として表現される (2.6 では)
  - 「2.8 構文解析器の拡張」で別の要素を追加 (真偽値リテラル、グループ化された式、`if` 式など)
- 式の表現
  - 構文木の生成
  - データ構造による表現

式とデータ構造 (識別子と整数リテラル)

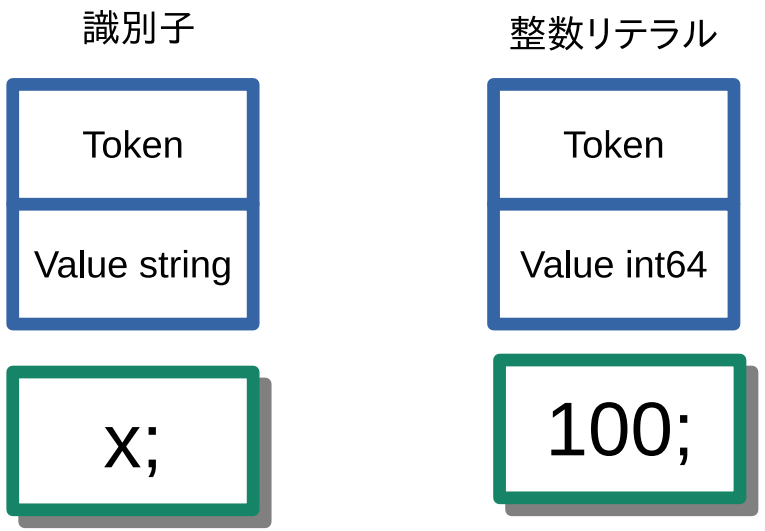

Figure: 識別子と整数リテラルのデータ構造

- 識別子と整数リテラルは単独のデータ構造として表現
  - トークンおよび値 (識別子の名前)

式とデータ構造 (前置演算子)

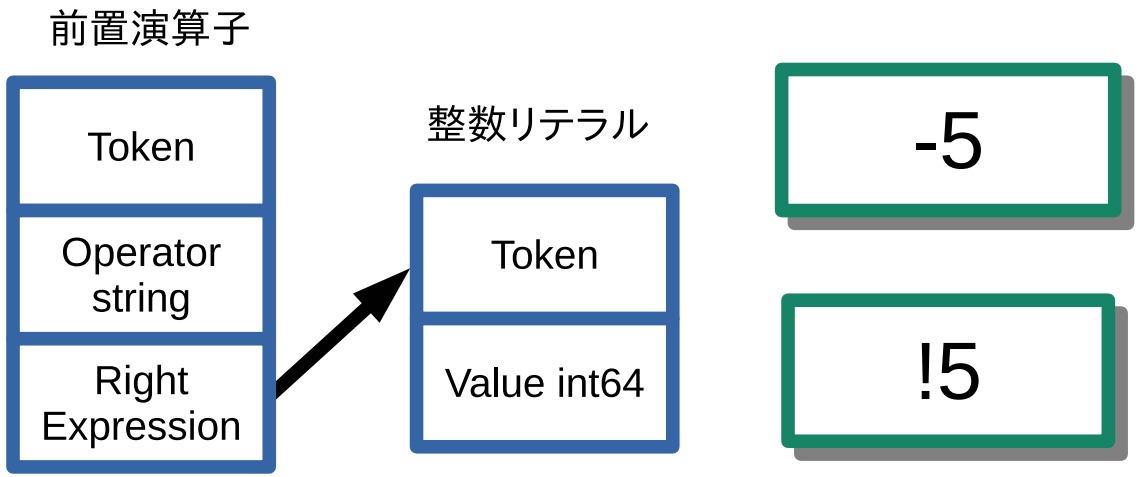

Figure: 前置演算子のデータ構造

- 前置演算子 (prefix operator)
  - -, !
  - データ構造に右辺の情報 ··· `Expression` インターフェース型

式とデータ構造 (中置演算子)

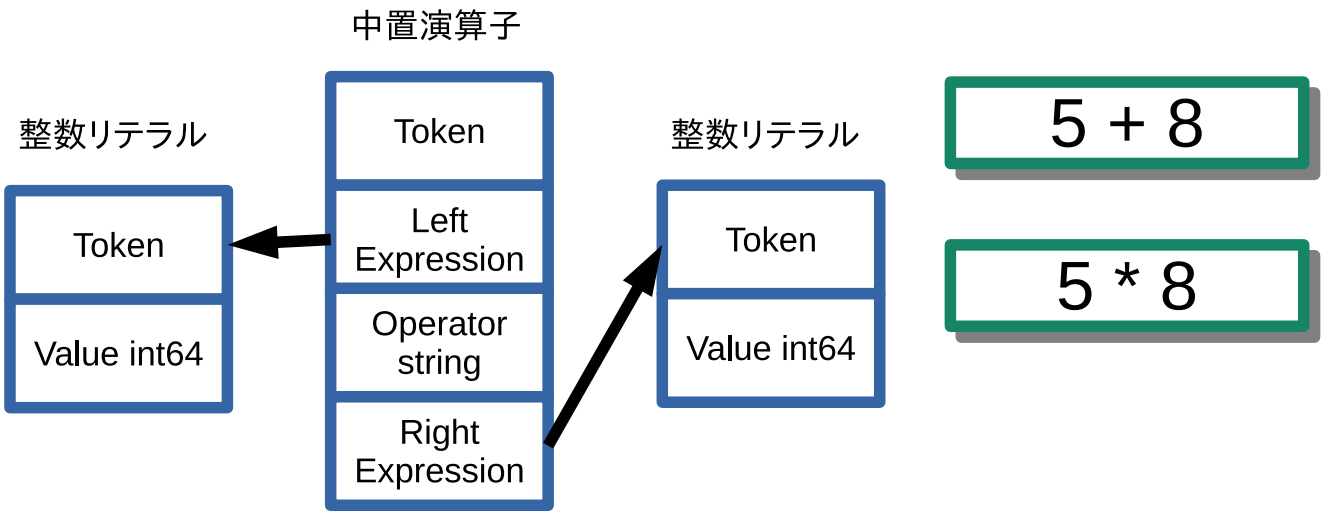

Figure: 式と構文木

- 中置演算子 (infix operator)
  - +, -, *, /
  - データ構造に左辺と右辺の情報 ··· `Expression` インターフェース型

式と構文木

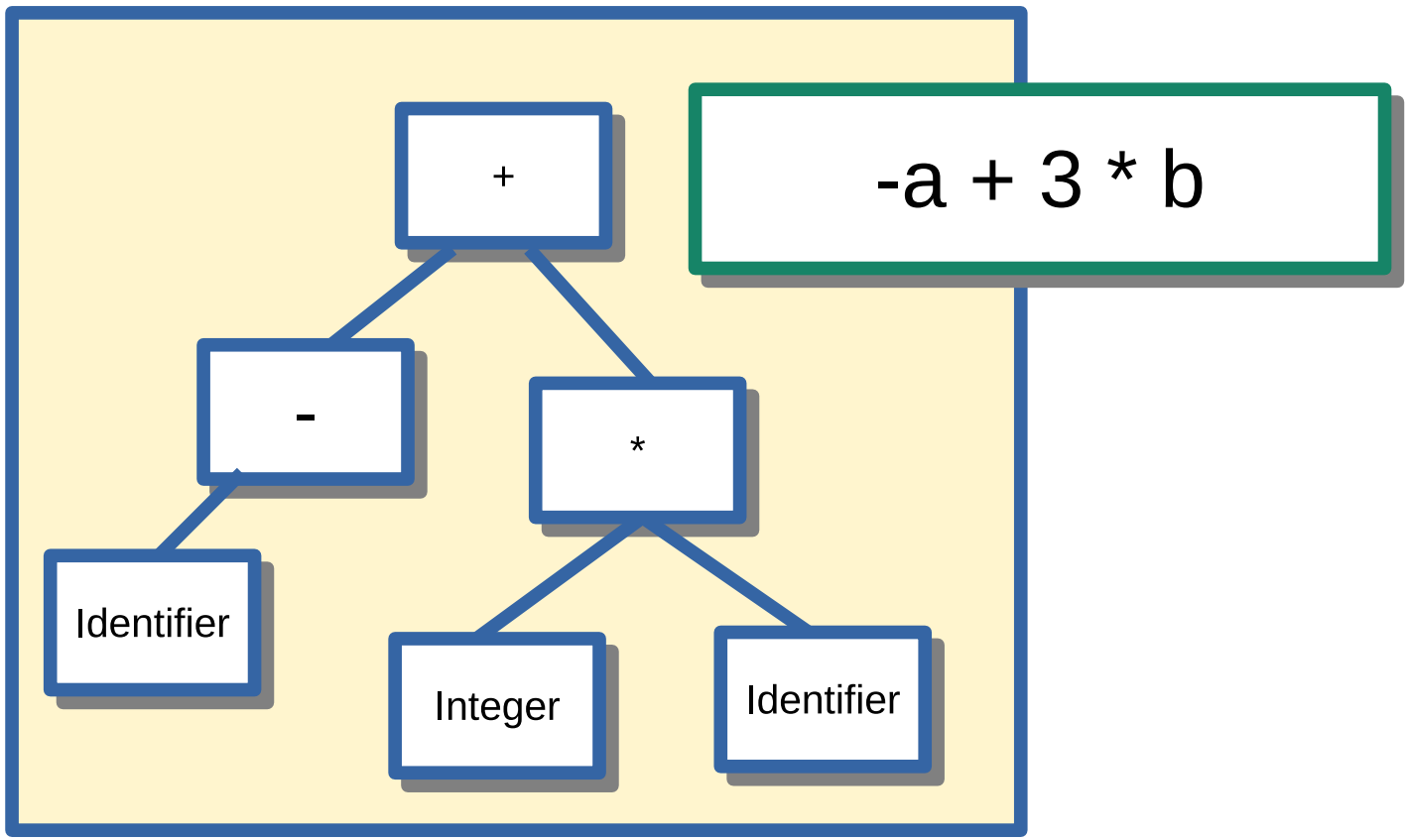

Figure: 式と構文木

式とデータ構造

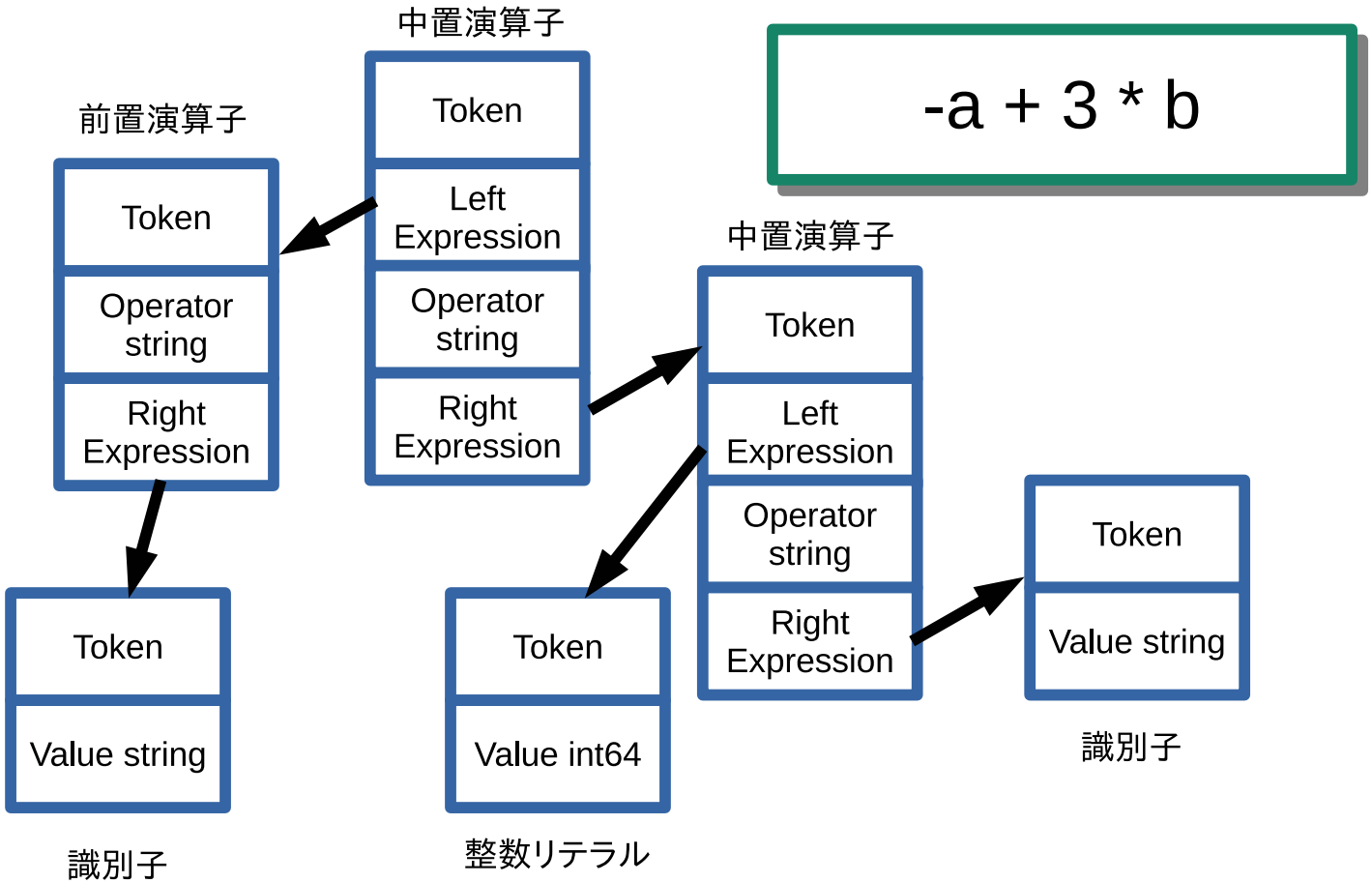

Figure: -a+3*b のデータ構造の例

式とデータ構造

- 式 (expression)
  - 演算子の結合
    - 演算子を中心に右辺・左辺を結合
    - 識別子・整数リテラルは葉
- データ構造
  - 右辺・左辺は `Expression` インターフェース
  - `Expression` インターフェースを実装する任意の構造体

式と構文木とデータ構造

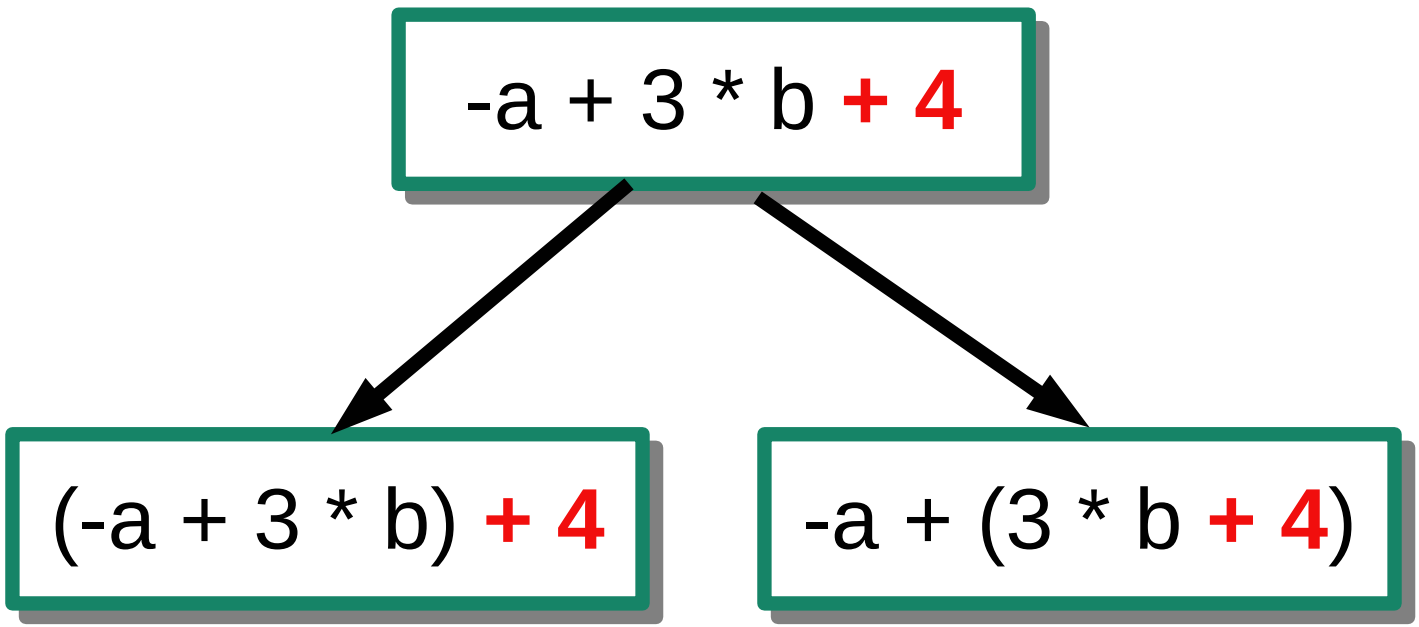

Figure: 式 -a+3*b+4 の表現

式と構文木とデータ構造

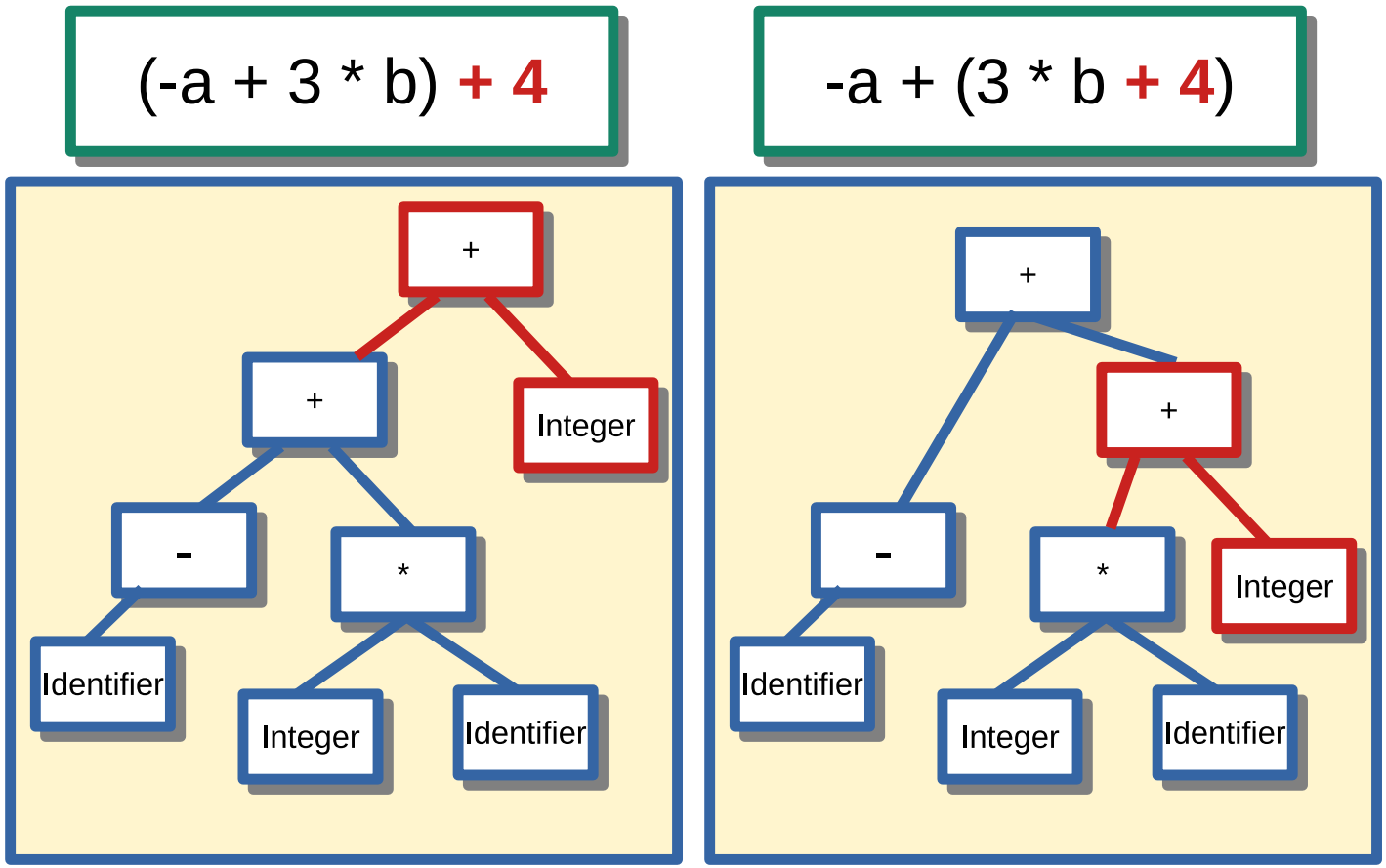

Figure: 式 -a+3*b+4 の構文木による異なる表現の例

式とデータ構造

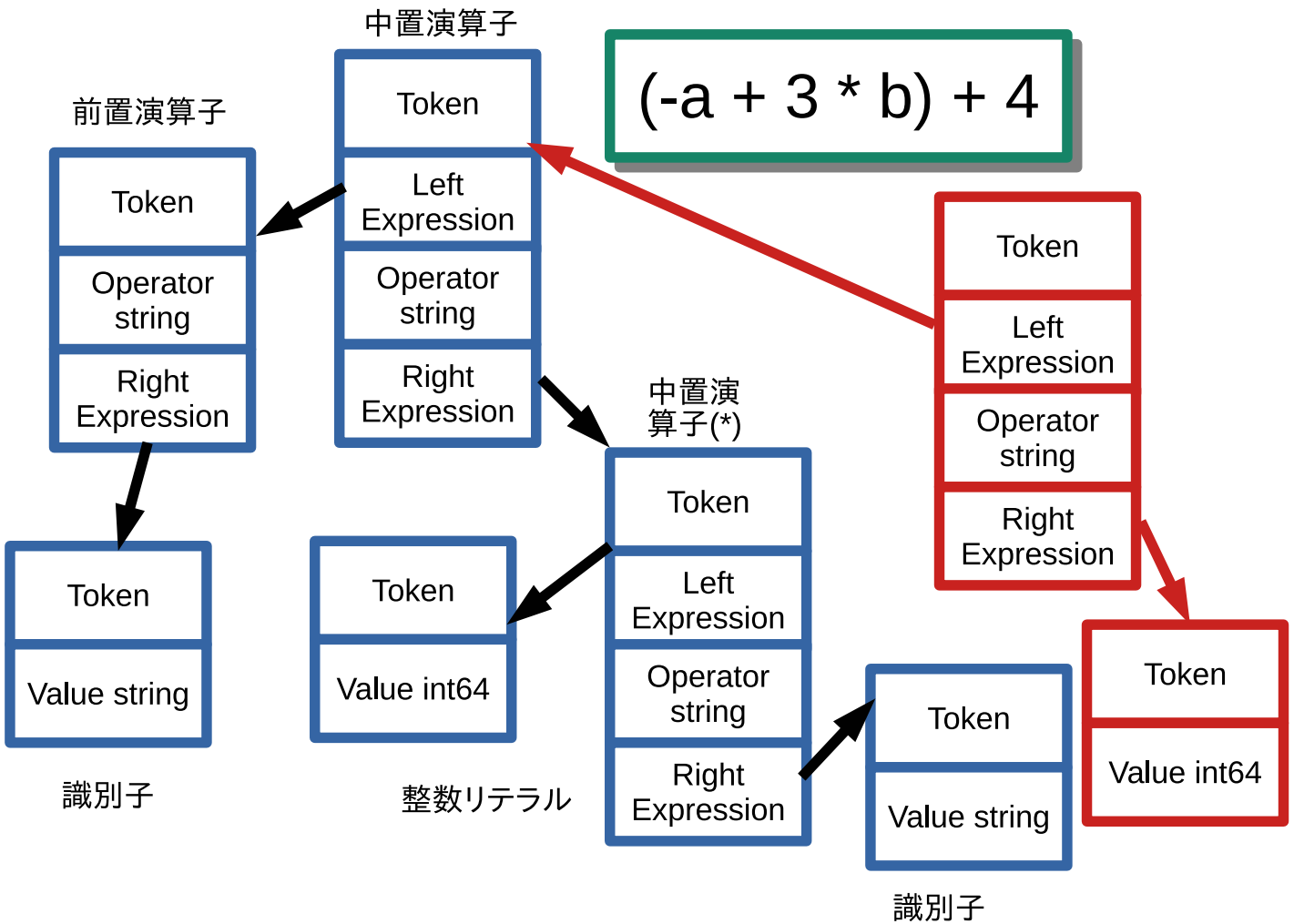

Figure: 式 -a+3*b)+4 のデータ構造の例

式とデータ構造

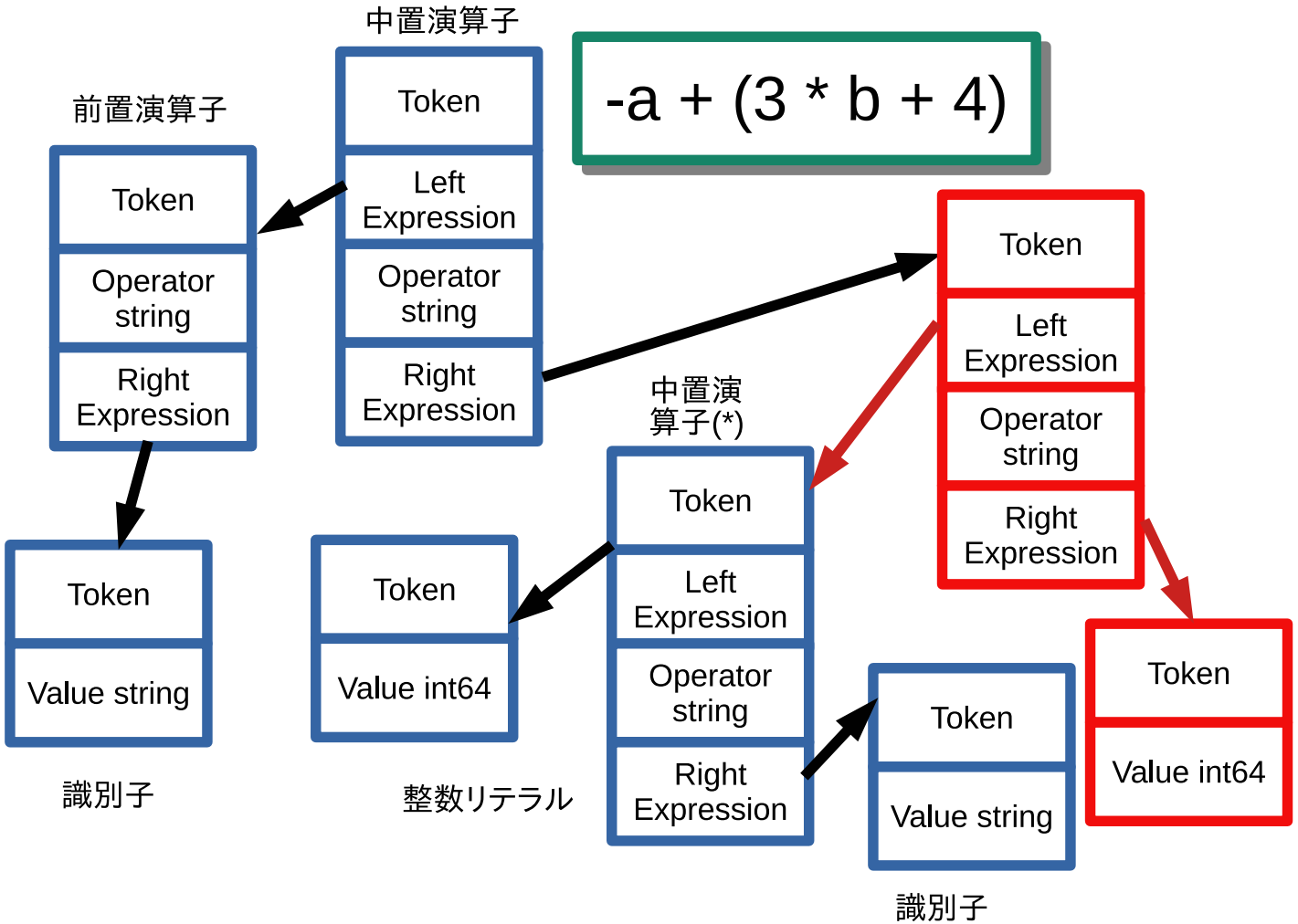

Figure: 式 -a + (3*b+4) のデータ構造の例

構文解析とデータ構造

- 構文解析
  - 式を解析して構文木を生成
  - 複数の演算子の存在
    - 演算子の優先順位を考慮
    - データ構造の接続
- Pratt 構文解析器
  - トークンタイプごとに構文解析関数を関連付け
  - 中置演算子の優先順位にもとづく木構造の生成