平成22年度 学士論文

内部ドメイン専用言語支援のための型に連動した字句・構文ルールの変更機構

東京工業大学 理学部 情報科学科
学籍番号 07-0236-3
市川 和央

指導教員
千葉 滋 教授

平成23年2月7日

概要

Scala をはじめとする多くの新しいプログラミング言語が内部ドメイン専用言語を支援するための仕組みを備えている。内部ドメイン専用言語によってプログラムを記述することはソースコードに高い可読性を与え、生産性・保守性を向上させることができる。

しかし、従来の内部ドメイン専用言語の実現方法には様々な問題がある。まず、ホスト言語が解析できないような文法は利用できないという制限が存在する。例えば、Scheme の構文マクロは構文木に対するパターンマッチであるため、構文解析ができないような文法は実現することができない。また、従来の手法では複数の内部ドメイン専用言語を同時に利用しようとすると、それらが衝突を起こして正常に動作しなくなる危険性がある。例えば、C/C++言語の字句置換マクロによって内部ドメイン専用言語を実現しようとした場合、本来関係の無いはずの部分までマクロによって書き換えられてしまう可能性がある。更に、使いやすい内部ドメイン専用言語の構成と利用には高度な知識が必要なため、多くのプログラマにとって敷居が高いものとなっていた。例えば Scala で内部ドメイン専用言語を実現しようとした場合、implicit conversion や省略規則に精通していて、かつメソッドの優先順位や結合性の決定規則なども知っている必要がある。内部ドメイン専用言語で書かれたプログラムにバグがあった場合、その発見が困難であるというのも非常に大きな問題点である。メタプログラミングを用いて内部ドメイン専用言語を実現した場合、コンパイル時にソースコードが書き換えられてしまうため、バグの追跡を行うためには内部ドメイン専用言語のソースコードを読まなければばならない。

本研究では、型によって字句・構文ルールを切り替える手法により、これらの問題を解決した。字句・構文ルールは N 項演算子として定義し、期待される型によって利用される N 項演算子を選択することで内部ドメイン専用言語を実現する。型によって局所的に字句・構文ルールを切り替えることにより、ホスト言語では本来解析できないような多様な文法の解析を可能とした。字句・構文ルールの変更は構文木が構成されるよりも前に行われるため、たとえホスト言語が構文解析できないような文法でも許容することができる。また、期待される型による字句・構文ルールの切り替えにより、内部ドメイン専用言語ごとに別々の解析ルールが適用されるた

め、他の内部ドメイン専用言語との衝突は自動的に回避される。そのため明示的に衝突の回避を行う必要はなく、複数の内部ドメイン専用言語を同時に利用することができる。本研究では字句・構文ルールを N 項演算子という形で定義するため、従来の手法と比べて単純で分かりやすく、それ程多くの知識を必要としない。N 項演算子という概念自体も直感的で親しみやすい。本システムは型安全であり、構文エラーや型エラーにより内部ドメイン専用言語で記述されたプログラムのバグを発見することができる。そのため、内部ドメイン専用言語を利用して記述したプログラムのコードはそのコード上で検証することができる。

本研究では Java 言語からインターフェースや内部クラスなどを取り除いたサブセットのコンパイラを作成し、そのコンパイラに対してこのアイデアを適用した。実装は Java 言語によって行なった。左再帰カウンタとメモ化を利用したトップダウン構文解析を字句解析および型チェックと連携させることにより実現している。作成したコンパイラを利用して SQL や BNF といったドメイン専用言語を記述できることを確認した。

謝辞

本研究を進めるにあたって、研究の方針や論文の構成法など数多くの助言と指導をして頂いた千葉滋教授に心より感謝致します。また、本研究の仕様や実装、論文の構成方法などについて様々な助言をして頂いた武山文信氏に心から感謝致します。最後に、情報科学に関する多くの知識を与えて下さった千葉研究室の皆様に感謝致します。

目 次

第 1 章　はじめに **9**

第 2 章　内部ドメイン専用言語 **10**

2.1 内部ドメイン専用言語の必要性 10
2.2 従来の内部ドメイン専用言語の実現手法の問題点 11
2.2.1 字句置換マクロ 13
2.2.2 構文マクロ 16
2.2.3 Scala 18

第 3 章　型により制限されたユーザ定義 N 項演算子 **21**

3.1 概観 21
3.1.1 N 項演算子の定義方法 21
3.1.2 readas パラメータ 22
3.1.3 優先度 22
3.1.4 利用方法 23
3.1.5 N 項演算子とメタプログラミング 24
3.2 特徴 24
3.2.1 表現能力 24
3.2.2 衝突の回避 25
3.2.3 分かりやすさ 26
3.2.4 バグの発見 26

第 4 章　実装 **28**

4.1 コンパイラの基礎 28
4.1.1 字句解析 28
4.1.2 構文解析 28
4.1.3 意味解析 29
4.1.4 コード生成 29
4.2 本システムの実装の概要 29
4.2.1 宣言部分と本体部分を分ける 29
4.2.2 字句解析・構文解析・型チェックの連携 30
4.2.3 トップダウン構文解析 30

4.3 宣言部分の解析 30
4.3.1 宣言部分とは 30
4.3.2 字句・構文解析 30
4.3.3 意味解析 31
4.4 本体部分の解析 31
4.5 式部分の解析 32
4.5.1 解析順序 32
4.5.2 括弧で括られた式の解析 32
4.5.3 通常の Java の式 32
4.6 N 項演算の解析 33
4.6.1 N 項演算子の試行 33
4.6.2 左再帰型 N 項演算子 33
4.6.3 メモ化 34
4.6.4 左再帰カウンタ 37
4.7 式部分の解析の例 38

第 5 章 ケーススタディ 45
5.1 四則演算 45
5.2 BNF 46

第 6 章 まとめと今後の課題 50
6.1 まとめ 50
6.2 今後の課題 51
6.2.1 表現力の強化 51
6.2.2 readas の制限 51
6.2.3 コンパイル時のオーバーヘッドの評価 51

図 目 次

2.1 SQL の select 文の例 1 11
2.2 SQL の select 文の例 2 12
2.3 オブジェクト形式マクロを用いた SQL の select 文の再現例 13
2.4 関数形式マクロを用いた SQL の select 文の再現例 15
2.5 Scheme の構文マクロを用いた SQL の select 文の再現例 . . 16
2.6 Scheme の構文マクロを用いた SQL の select 文の再現例（衝突回避版） 18
2.7 Scala による SQL の select 文の再現例 19

3.1 本システムを用いた SQL の select 文の再現例 21
3.2 SQL の select 文を再現する N 項演算子の定義例 22
3.3 返り値の型が求められている型のサブタイプであるような場合まで許容してプログラムの解釈が不可能になってしまう例 23
3.4 図 3.1 の変換後擬似コード 24

4.1 左再帰型 N 項演算子を含む例 34
4.2 バックトラックにより同じ部分の解析が行われる場合の例 35
4.3 表 4.1 の第三行の情報を基に構築した構文木 36
4.4 表 4.5 の第七行の情報を基に構築した構文木 37
4.5 図 4.1 を単純なトップダウン構文解析で解析した場合 . . . 38
4.6 図 2.2 の再現例 39
4.7 左再帰カウンタによる左再帰の停止 40
4.8 :col >= :val の試行終了時の構文木 41
4.9 最も深い位置にある:left and :right の試行終了時の構文木 . 43
4.10 解析結果の構文木 44

5.1 四則演算を実現する N 項演算子の定義例 45
5.2 図 5.1 の N 項演算子の利用例 45
5.3 図 5.2 のコンパイル後のコード 46
5.4 優先順位を全て等しくした場合の図 5.2 のコンパイル後のコード 46

5.5 優先順位を逆転させた場合の図 5.2 のコンパイル後のコード 46
5.6 BNF を表現する N 項演算子の定義例 47
5.7 図 5.6 の N 項演算子の利用例 48
5.8 図 5.7 のコンパイル後の擬似コード 49

表 目 次

4.1 select :col from :table の試行後のメモテーブル 34
4.2 select :col from :table where :cond の試行後のメモテーブル 36
4.3 :col >= :val の試行後のメモテーブル 41
4.4 最も深い位置にある:left and :right の試行後のメモテーブル 42
4.5 select :col from :table where :cond の試行終了後のメモテーブル 43

第1章　はじめに

ドメイン専用言語とはある特定の問題領域における問題解決に特化したプログラミング言語を指し、それを汎用的なプログラミング言語の枠組みの中で実現したものを内部ドメイン専用言語と呼ぶ。内部ドメイン専用言語の利用はソースコードの可読性を向上させ、プログラミングの生産性や保守性を高めることにつながる。

内部ドメイン専用言語を実現する従来の方法には多くの問題点がある。主な問題は以下の四点である。

1. 実現できる文法に制限がある
2. 内部ドメイン専用言語同士の衝突の可能性がある
3. 高度な知識が必要で難しい
4. バグの発見が難しい

そこで、従来の手法の問題点を解決する手法として、本研究では型によって字句・構文ルールを切り替える手法を提案する。字句・構文ルールはN項演算子として記述し、期待される型によってN項演算子を制限することでルールの切り替えを行う。この手法を利用することで、

1. ホスト言語の文法にとらわれない表現能力
2. 型をスコープとした衝突の回避
3. 分かりやすく直感的な利用方法
4. 構文エラーや型エラーによるバグの検出

を得ることができる。

以下、第2章では内部ドメイン専用言語を実現する既存のアプローチの問題点について述べ、第3章では本研究のアイデアである型により制限されたユーザ定義N項演算子を紹介する。第4章では本システムの実装について解説し、第5章では本システムの具体的な利用法を例を用いて説明する。最後に、第6章では本研究のまとめと今後の課題について述べる。

第2章　内部ドメイン専用言語

本章では内部ドメイン専用言語を実現する既存のアプローチの問題点について述べる。第1節では内部ドメイン専用言語の必要性について説明する。第2節では内部ドメイン専用言語を実現する既存の手法を紹介し、それらの問題点を指摘する。

2.1　内部ドメイン専用言語の必要性

ドメイン専用言語とは、ある特定の問題領域における問題解決に特化したプログラミング言語を指す。それとは逆にあらゆる問題を包括的に扱う言語を汎用言語と呼ぶ。具体的には、汎用言語はC言語、Java、Lispなどの言語を指し、ドメイン専用言語はSQL、XML、正規表現などが例として挙げられる。汎用言語はあらゆる問題を扱う必要があるためTuring完全[1]であるが、ドメイン専用言語は必ずしもTuring完全ではなく、例えばSQLやXMLはTuring完全ではない。

ドメイン専用言語は内部ドメイン専用言語と外部ドメイン専用言語の二つに大別される[2]。内部ドメイン専用言語とは、ドメイン専用言語を構築する際に用いた汎用言語（以下ホスト言語と呼ぶ）の枠組みの中で、ライブラリなどの形で実現されたドメイン専用言語を指し、外部ドメイン専用言語とは、ホスト言語とは異なる言語で記述するドメイン専用言語を指す。内部ドメイン専用言語はホスト言語の機能や既存のツールをそのまま利用できるが、ホスト言語の文法によって制限されるという欠点を持つ。逆に、外部ドメイン専用言語はホスト言語とは別の言語であるため、自由な形式の文法を許すことができるが、ホスト言語用のツールのサポートを受けることができない。

ドメイン専用言語を利用することの最大の利点はプログラミングの生産性や保守性を向上させるという点である。ドメイン専用言語は扱う問題領域の処理を表現しやすいように設計されているため、ソースコードは無駄を排した非常に可読性の高いものとなっている。そのため効率的なプログラムの作成が可能で、検証や修正も容易に行うことができる。

[1]計算機の理想的なモデルであるTuring機械と同等の計算能力を持っていること

[2]Martin Fowler氏の記事「Language Workbenches: The Killer-App for Domain Specific Languages?」[1]における分類

内部ドメイン専用言語はホスト言語の統合開発環境などと連携することで、ドメイン専用言語のメリットを更に効率的に引き出すことができる。例えば、ホスト言語用のエディタやデバッガを利用することでコードの自動補完やミスタイプの検出、プログラムのステップ実行といった、プログラムの記述や点検を支援する機能を使うことができる。外部ドメイン専用言語でこのような支援を受けるためには専用の開発環境を用意する必要があり、非常に高いコストがかかる。

内部ドメイン専用言語の欠点はホスト言語によって文法が制限されてしまう点であるが、ホスト言語側が内部ドメイン専用言語の記述を支援することによりその欠点を小さくすることが可能である。近年 Scala や Ruby といった内部ドメイン専用言語の記述がしやすい汎用言語が登場して注目を集めており、これは内部ドメイン専用言語の支援機構を備えた汎用言語が求められていることを意味している。しかし、この内部ドメイン専用言語を支援する従来の手法には幾つかの大きな問題点が存在する。次節では、従来の手法にはどのような問題点が存在するかについて述べる。

2.2　従来の内部ドメイン専用言語の実現手法の問題点

従来の内部ドメイン専用言語の実現手法には以下の四つの問題点がある。

- 表現能力の不足
- 内部ドメイン専用言語同士の衝突
- 高度な知識が必要
- バグの発見が困難

使いやすい内部ドメイン専用言語を実現するためはホスト言語による制約を弱めて自由度の高い文法を扱うことができなくてはならない。つまりメソッド呼び出しにおける括弧やドットなどの非本質的な記号を省略できたり、特定の位置に存在する字句をリテラルとみなしたり、サブルーチンの記述方法を変更できたりする必要がある。例えば、次の図 2.1 を参照して欲しい。これは SQL 命令の一種である select 文の非常に簡単な例である。SQL はデータベースを操作することに特化したドメイン専用言語であり、この

```
1 select name from employees where TOEIC_score >= 600
```

図 2.1: SQL の select 文の例 1

select 文は employees というテーブルから TOEIC_score >= 600 という条件を満たしたデータを抽出し、それらのデータの name 列の値を取得する命令である。これを見てわかるように、select 文は select...from...where... という特殊な構造をしており、また、name や TOEIC_score はテーブルの列名であるにもかかわらず、ダブルクォートで囲まれていない。不等号>= も右辺は数値であるのに対して左辺は列名であり、本来の意味と異なる意味で用いられている。

```
select name from employees where TOEIC_score >= 600 and age < 40
```

図 2.2: SQL の select 文の例 2

図 2.2 は「and」を用いて複数の条件を指定した select 文の例である。ここで and は条件同士を結合する二項演算子のような働きをしており、また、不等号>=や<よりも優先順位が低いものとして扱われていることが分かる。以上のことから、図 2.1 及び図 2.2 のような文を実現するためには少なくとも、

- select...from...where... という構造を許す
- ダブルクォートで囲まれていない字句をリテラルとみなす
- 演算子を本来と異なる意味で解釈する
- 二項演算子のようなサブルーチンを定義でき、優先順位も設定できる

といったような条件を満たす必要があることがわかる。従来の内部ドメイン専用言語の実現手法ではこれらの条件をすべて満たすことはできない。

また、内部ドメイン専用言語の文法はそれが必要とされる部分のみで適用され、それ以外の部分では使われないようになっていなければならない。図 2.2 で利用されている and は一般的によく使われる単語であるため、他の内部ドメイン専用言語でも利用されている可能性がある。従来の内部ドメイン専用言語の実現手法ではそのような場合に衝突が発生し、少なくとも一方の内部ドメイン専用言語は正常に動かなくなってしまう。

内部ドメイン専用言語を実現する従来の手法は、プログラムコードの書き換えや省略規則を駆使する必要があり、それらについての深い知識がなければ使いやすい内部ドメイン専用言語を構成することができない。そのため熟練したプログラマー以外は手が出しづらく、素人プログラマーの使

用を阻害していた。

従来の手法のうち、プログラムのソースコードを書き換えが伴う手法では、プログラムのバグの発見が非常に難しいという問題がある。これは、プログラムの意味解析が行われるよりも前にソースコードが変更されてしまうためで、バグの追跡にはその内部ドメイン専用言語自体のソースコードを読まなければならない。

もちろん、既存の手法全てが上で挙げたような問題点を必ず持っているというわけではなく、手法によっては一部の問題点を克服しているものもある。本節の残りの部分では図 2.1 を目的コード例として[3]既存の手法を紹介し、その問題点について確認していく。

2.2.1 字句置換マクロ

字句置換マクロはソースコード上で一致した字句を書き換える C/C++ 言語で採用されているメタプログラミング方式で、C/C++言語の字句置換マクロには単純にテキストを置換するオブジェクト形式マクロと引数を持つことができる関数形式マクロの二種類が存在する。

オブジェクト形式マクロはその単純さゆえに表現能力が低く、影響範囲のコントロールができず、またコードを見ただけでは何を意図したコードかわからないという欠点を持つ。C 言語上でオブジェクト形式マクロを用いて図 2.1 のコードを擬似的に再現したものが図 2.3 である。

```
1  // 実際に呼び出される関数
2  SQLResult* select_from_where(char* col, Table* tbl, char* cnd) {
3      ...
4  }
5
6  // マクロ定義
7  #define select select_from_where(
8  #define from ,
9  #define where ,
10 #define end );
11
12 // 利用例、employeesはTable構造体へのポインタ
13 SQLResult* result =
14     select "name" from employees where "TOEIC_score >= 600" end
```

図 2.3: オブジェクト形式マクロを用いた SQL の select 文の再現例

[3]本来の select 文は where 句が省略できたり、複数の列名を指定できたりするが、簡単のためそのような場合は扱わない。

```
select "name" from employees where "TOEIC_score >= 600" end
```

の select が `select_from_where(` に、from と where が `,` に、end が `);` に置き換わり、

```
select_from_where("name", employees, "TOEIC_score >= 600");
```

となる。条件節全体が文字列として渡されている、図 2.1 には存在しなかったキーワードが書かれている等から表現能力に問題があることがわかる。条件が文字列として渡されているのは、オブジェクト形式マクロでは二項演算子を実現することはできないためで、図 2.1 には存在しなかった end が書かれているのは、select 文を強引に関数呼び出しに書き換えるために終了箇所を示すキーワードが必要であったためである。この例において表現能力以上に問題なのは、定義した select 文と関連がないコード部分でも select や from、where や end といった単語があればこのマクロにより置き換えられてしまう点である。マクロによりコードが書き換えられてしまうので、ユーザは目的のコードが最終的にどのようになるのかを意識しながらプログラムを書かなければならない。また、この問題によりプログラムのバグの発見も非常に困難なものとなってしまう。

関数形式マクロは引数の文字列化という機能によりダブルクォートで囲まれていない字句をリテラルとみなすことができるため、字句置換マクロよりは表現能力が高いと考えられるが、字句置換マクロと同様衝突は容易に発生し、またコードを読む際にマクロによる書き換えを常に意識しなければならない。次の図 2.4 は C++上で関数形式マクロを用いて図 2.1 のコードを再現したものである。

```
select(name) from(employees) where(TOEIC_score >= 600)
```

の `select(name)` が `(new Select("name"))` に展開され、`from(employees)` が `->from(employees)` に、`where(TOEIC_score >= 600)` が `->where("TOEIC_score >= 600");` に展開されるため、

```
(new Select("name"))->from(employees)->where("TOEIC_score >= 600");
```

となる。列名 name は select マクロにより文字列化されるため、ダブルクォートを付ける必要がない。条件句も同様に文字列化して渡しているが、これでは条件句は自身で解析をしなければならない。そのため、条件句の部分は外部ドメイン専用言語になってしまっているといえる。それぞれの引数は非本質的な記号である括弧で括られていることからも、表現能力はあまり高いとは言えない。また、マクロ定義は名称と引数の数が一致すれば適用されるため、同名で引数の数の一致する関数やマクロを定義

```
class SQLResult {
public:
    SQLResult(char* col, Table* tbl, char* cnd) { ... }
};

class From {
private:
    char* col;
    Table* tbl;
public:
    From(char* col, Table* tbl) : col(col), tbl(tbl) {}
    SQLResult* where(char* cnd) {
        return new SQLResult(col, tbl, cnd);
    }
};

class Select {
private:
    char* col;
public:
    Select(char* col): col(col) {}
    From* from(Table* tbl) {
        return new From(col, tbl);
    }
};

// マクロ定義
#define select(s) (new Select(#s))      // #sは引数sを文字列化
#define from(t) ->from(t)
#define where(c) ->where(#c);

// 利用例、employeesはTableクラスのオブジェクトへのポインタ
SQLResult* result =
    select(name) from(employees) where(TOEIC_score >= 600)
```

図 2.4: 関数形式マクロを用いた SQL の select 文の再現例

すると衝突が発生してしまう[4]。更に、ソースコードと実際に実行されるコードが大きく異なるため常にマクロ展開を意識しなければならない。実行されるコードを常に意識していないと、非常に厄介なバグが発生してしまう危険がある。例えば、次のような例が有名である。

```
#define MUL(a,b) a * b
```

のようにマクロを定義した場合、

```
MUL(1 + 2, 3 - 4) / 2
```

は

```
1 + 2 * 3 - 4 / 2
```

[4]図で from 及び where マクロが同名関数と衝突していないのは、マクロ定義が関数定義より後ろにあるためである。マクロは関数呼び出しよりも先に処理されるため、from 関数を通常の方法で使うことはできない

と展開されてしまう。この例の場合はマクロ側にバグがあるが、マクロを利用しているソースコードにバグがある場合も、構文解析や型チェックの前にソースコードが変更されてしまうため、やはりバグの発見は困難なものとなる。

以上のことから、字句置換マクロは表現能力が不足しており、衝突が容易に発生し、常にマクロ展開を意識する必要があり、バグが発見しづらいという問題点があるといえる。

2.2.2 構文マクロ

構文マクロとは構文木に対してパターンマッチを行いマッチした部分木を置換するメタプログラミング手法で、比較的高い表現能力を持ち、内部ドメイン専用言語同士の衝突の回避も可能であるが、その定義は非常に難しく高度な知識を必要とし、バグも発見しづらい。次の図 2.5 は Lisp 方言の一つである Scheme 上で構文マクロを用いて図 2.1 のコードを再現したものである。ただし簡単のためこのコードはマクロ名の衝突の回避については考慮していない。define-syntax がマクロ定義で、ここでは select マ

```
1  ;; 実際に呼び出される関数
2  (define (select_from_where col table condition)
3    ...)
4
5  ;; 条件判定を行うクロージャを返す
6  (define (greater-equal col val)
7    ...)
8
9  ;; マクロ定義
10 (define-syntax select
11   (syntax-rules (from where)
12     ((select col from table where condition)
13      (select_from_where 'col table condition))))
14
15 (define-syntax >=
16   (syntax-rules ()
17     ((>= col val)
18      (greater-equal 'col val))))
19
20 ;; 利用例、employeesはテーブル
21 (define result
22   (select name from employees where (>= TOEIC_score 600)))
```

図 2.5: Scheme の構文マクロを用いた SQL の select 文の再現例

クロと>=マクロの二つが定義されている。syntax-rules の第一引数はマクロ内で利用される予約語のリストで、第二引数はパターンとその展開結果のペアのリストである。そのため、select マクロは from と where を予約語として、(select col from table where condition) を (select_from_where

’col ’table condition) に変換し、>=マクロは (>= col val) を (greater-equal ’col val) に変換する。

構文マクロは字句情報に加えて構文木の情報を利用できるため、字句置換マクロに比べて表現能力は強力で、(select col from table where condition) のような新しい構文を許すことができる。関数形式マクロと同様に、引数をリテラルのように読むことも可能である。例えば、列名である name や TOEIC_score はマクロ展開時にクォートを付加することでシンボルに変換される。ただし、構文マクロを適用するためには構文木を作成する必要があるため、構文解析ができないような文法は表現できない。例えば、Scheme の構文は前置記法の S 式でなければならないため、それに違反するような、例えば中置記法の文法などは表現することができない。

図 2.5 のコードはマクロをグローバルなスコープで定義しているため、>=は本来の比較演算の意味を失ってしまう。そこで、Scheme では例えば次の図 2.6 のようにすることでこのようなマクロ名の衝突の発生を回避する。let-syntax は局所的なマクロ定義で、第一引数は定義するマクロのリスト、第二引数が定義したマクロのスコープである。ここでは、,(syntax->datum (syntax (begin expression ...))) でのみ有効なマクロとして、select マクロと>=マクロの二つを定義している。これにより sql-syntax マクロ内部でのみ select 文の構文が定義される。

Scheme の構文マクロは健全性を持っており、意図しない変数名の衝突が発生しない。これは、マクロ内部で導入された変数とそのマクロの展開先のスコープに存在する変数とが衝突しないよう、自動的に変数名の改名が行われるためである。これにより、sql-syntax 内部で定義した select マクロは sql-syntax の展開先における select と一致しなくなってしまう。そのため datum->syntax により静的なスコープ規則を曲げてあたかもその変数がもともとスコープに存在していたかのように振舞わせることで、図 2.6 のような sql-syntax を実現している。

以上のようにすることでマクロ名の衝突を回避することができるが、そのためには使う文法を明示するマクロを作成して利用する必要があり、後からその文法を拡張するのは容易ではない。またそのようなマクロの定義は datum->syntax などの高度な知識が必要なため、非常に複雑で難しいものとなってしまう。今回の例は単純なため比較的簡単だが、例えば>=マクロを where 句内のみで有効になるよう変更すると、その複雑性は大きく増加する。

Scheme マクロは構文マクロであるため、その展開時に型情報は考慮されない。そのため、タイプチェックが行われるのはマクロ内部であり、型エラーによるバグが発見しづらいという欠点も持っている。

```
 1 ;; 実際に呼び出される関数
 2 (define (select_from_where col table condition)
 3   ...)
 4
 5 ;; 条件判定を行うクロージャを返す
 6 (define (greater-equal col val)
 7   ...)
 8
 9 ;; マクロ定義
10 (define-syntax (sql-syntax x)
11   (syntax-case x ()
12     ((sql-syntax expression ...)
13      (with-syntax
14        ((expr (datum->syntax
15          (syntax k)
16          `(let-syntax
17            ((select
18               (syntax-rules (from where)
19                 ((select col from table where condition)
20                  (select_from_where 'col table condition))))
21             (>=
22               (syntax-rules ()
23                 ((>= col val)
24                  (greater_equal 'col val)))))
25
26             ,(syntax->datum (syntax (begin expression ...)))))))
27        (syntax expr)))))
28
29 ;; 利用例、employeesはテーブル
30 (define result
31   (sql-syntax
32     (select name from employees where (>= TOEIC_score 600))))
```

図 2.6: Scheme の構文マクロを用いた SQL の select 文の再現例（衝突回避版）

2.2.3 Scala

Scala[3] は字句置換マクロや構文マクロのようなメタプログラミングを用いる方法と異なり、構文の省略規則の活用と implicit conversion という型変換の機能を利用して内部ドメイン専用言語を実現する。Scala ではレシーバが明示されている引数が一つまたはないメソッドならば、メソッド呼び出しのドット及び引数の括弧を省略できる。そのため、メソッドをまるで（後置）単項演算子または二項演算子のように記述することができる。implicit conversion は暗黙的な型変換を行う仕組みで、この機能により型の一致しないものを代入できたり、本来その型には存在しないメソッドを呼ぶといったことが可能となる。図 2.7 は Scala により図 2.1 のコードを再現したものである。

```
sql select "name" from employees where "TOEIC_score" >=~ 600
```

は構文の省略のルールにより

```
1  class Column(col: String) {
2      def >=~(value: Int) = new SQLCond_GEQ(this, value)
3  }
4
5  abstract class SQLCond
6
7  class SQLCond_GEQ(col: Column, value: Int) extends SQLCond
8
9  class SQLResult(col: Column, table: Table, cond: SQLCond)
10
11 class From(col: Column, table: Table) {
12     def where(cond: SQLCond) = new SQLResult(col, table, cond)
13 }
14
15 class Select(col: Column) {
16     def from (table: Table) = new From(col, table)
17 }
18
19 object sql {
20     implicit def stringToColumn(str: String) = new Column(str)
21     def select (col: Column) = new Select(col)
22 }
23
24 import sql._
25 val result
26     = sql select "name" from employees where "TOEIC_score" >=~ 600
```

図 2.7: Scala による SQL の select 文の再現例

```
sql.select("name").from(employees).where("TOEIC_score".>=~(600))
```

と解釈される。ここで、sql.select の引数は Column 型であり String 型ではないため、sql.stringToColumn として定義された implicit conversion により"name"は Column 型に変換される。"TOEIC_score"についても、String 型は`>=~`メソッドを持たないため、同じく implicit conversion によって Column 型に変換され、Column 型の`>=~`メソッドを呼び出す。

Scala の内部ドメイン専用言語の表現能力はやや不足している。省略規則を利用するためにはレシーバオブジェクトがなければならないなどの条件があるため、今回の例では sql という図 2.1 では存在しなかったオブジェクト名を記述する必要があった。また、Scala はメタプログラミングを利用しないため、文字列化のようなことはできず、列名である name や TOEIC_score はダブルクォートで括らなければならない。Scala はメソッド名の終端の文字によって優先順位が決まるため、`>=~`は他のメソッドより優先される。

implicit conversion は内部ドメイン専用言語同士の衝突を引き起こす可能性がある。図 2.7 で`>=`ではなく`>=~`が使われているのはまさにこの衝突が原因となっている。これは String という汎用的な型に対する implicit conversion を定義したためである。省略規則の活用は、ドメイン専用言語

風の記述方法をとっていてもその実態はメソッド呼び出しなので、型によって呼び出されるメソッドが決定されるため、それによる衝突は発生しない。

Scala の手法は新たな構文を作る訳ではなく、省略規則等を応用して目的の構文を模倣しているため、その定義はあまり直感的でないものとなる。implicit conversion も多用すると衝突したり意図しない挙動を引き起こす可能性があるため、使いどころの見極めが難しい。また、内部ドメイン専用言語らしく書けるようにするためには、「メソッド名の終端の文字によって優先順位が決まる」等の知識が必要である。例えば図 2.7 の`>=~`を`>=:`や`>==`に変更することはできない。これは、`:`で終わるメソッドはそのメソッドの右辺がレシーバ、左辺が引数となり、`=`で終わるメソッドは優先順位が最低となる（例外あり）ためである。

Scala の手法は省略規則などの活用で内部ドメイン専用言語を実現しているため、内部ドメイン専用言語を利用する際も通常通り意味解析が行われる。そのため、型エラーなどによるバグの発見が可能である。

第3章　型により制限されたユーザ定義N項演算子

本章では本研究のアイデアである型により制限されたユーザ定義N項演算子を紹介し、それがどのように前章で示した従来の手法の問題点を解決するかを示す。

3.1　概観

本システムは端的に言えばユーザに自由にN項演算子を定義することを許し、定義したN項演算子をその返り値の型が求められる場所でのみ利用できるようにする、というものである。例えば、本システムを利用して図2.1のコードを再現すると次の図3.1のようになる。この例からわかるように、本システムでは文末を示すセミコロン以外には非本質的な記号を必要とせず、ほとんど外部ドメイン専用言語と同様の記述が可能となっている。この文において不等号>=は通常と異なる意味で解釈されているが、この文の前後や内部で>=を通常の比較の意味で用いることも可能である。これは、本システムがその場所で求められている型によってN項演算子を選択しているためで、例えばwhere句内ではSQLの条件を表す型が求められるのでSQLの条件を返す二項演算子>=が利用される。

3.1.1　N項演算子の定義方法

N項演算子の定義はメソッド定義とほぼ同じ形式で行うが、メソッド名にあたる場所には幾つかのオペレータとオペランドからなるパターンが記述される。図3.2は図3.1のコードを実現するためのN項演算子の定義例である。オペレータはセパレータ・オペランド及び空白文字・コメントを

```
1  SQLResult result =
2      select name from employees where TOEIC_score >= 600;
```

図3.1: 本システムを用いたSQLのselect文の再現例

```
1 operators SQLOperators {
2     SQLResult select :col from :table where :cond
3         (readas Column col, Table table, SQLCond cond)
4          : priority = 100 { ... }
5
6     SQLCond :col >= :val(readas Column col, int val)
7          : priority = 200 { ... }
8 }
```

図 3.2: SQL の select 文を再現する N 項演算子の定義例

除いた任意の文字列で、N 項演算子を利用する際にキーワードとなるようなものを言う。図 3.2 の例では select, from, where, >=がそれにあたる。オペランドは語頭にコロンが付与された識別子で、N 項演算子を利用する際にはその N 項演算子の引数により置き換えられる。図 3.2 の例では:col, :table, :cond, :val がそれにあたる。オペランドは仮引数と前から順に対応しており、利用時には対応する仮引数の型の引数が入る。例えば、:table は Table 型の仮引数 table と対応しているため、図 3.1 では:table の場所に Table 型のオブジェクト employees が書かれており、これが引数として渡される。セパレータは各種括弧、シングルクォート、ダブルクォート、カンマ、セミコロンを指す。各種括弧、シングルクォート、ダブルクォートは対応関係を崩してしまうおそれがあり、カンマ及びセミコロンは式の終了部分を知るために必要なので、これらの記号はオペレータとして利用することができない。

3.1.2　readas パラメータ

図 3.2 において:col に対応する仮引数の型名は readas Column となっている。これは引数として与えられたトークンを Column 型として読むことを表している。具体的には、トークンを String リテラルとして読み、それを Column 型のコンストラクタに渡すことで Column 型のインスタンスを作り、それを引数として渡す。ただし、引数が Column 型を返す Java の式であった場合、それをそのまま引数として渡す。図 3.1 の場合、select 文の:col の位置に存在するトークンは name なので、select 文の第一引数として渡されるのは new Column(”name”) となる。

3.1.3　優先度

各 N 項演算子には仮引数のリストの後、メソッドボディの前に

```
: priority = 優先度
```

のように記述することで優先度を指定することができる。指定がない場合、自動的に優先度は最低となる。今回は簡単のため、優先度は非負整数で指定する。例えば、図 3.2 の例では select :col from :table where :cond の優先度は 100 であり、:col >= :val の優先度は 200 である。優先度は通常の数式の計算規則と同様、N 項演算子の結合順位を決めるパラメータで、優先度が高い N 項演算子のほうが先に計算される。ただし、括弧で括られた式は優先度に関わらず優先して計算される。本システムで定義した N 項演算子は左結合性を持つため、パターンの最も右に現れるオペランドには自身の優先度未満の N 項演算子のみ出現でき、それ以外のオペランドには自身の優先度以下の N 項演算子が出現可能となる。

3.1.4 利用方法

N 項演算子は前述したように、返り値の型がその場所で求められている型と一致したときにのみ利用される。ここで注意したいのは、返り値の型が求められている型のサブタイプであっても利用されないという点である。これを許してしまうと次の図 3.3 のような問題が発生する。図 3.3 のコードは排他的論理和を表す二項演算子^と指数を表す二項演算子^を同時に定義したものである。BitSet が求められるときには排他的論理和を表す^が、Double が求められるときには指数を表す^が利用されるが、その共通のスーパータイプ Object が求められているときにはどちらを利用すればよいか判断できない。

```
1  operators BitOperators {
2      // Exclusive or
3      BitSet :left ^ :right (int left, int right) { ... }
4  }
5
6  operators MathOperators {
7      // Exponentiation
8      Double :left ^ :right (int left, int right) { ... }
9  }
10
11 BitSet xor = 3 ^ 4;      // 7
12 Double exp = 3 ^ 4;      // 81
13 Object obj = 3 ^ 4;      // ?
```

図 3.3: 返り値の型が求められている型のサブタイプであるような場合まで許容してプログラムの解釈が不可能になってしまう例

3.1.5　N 項演算子とメタプログラミング

N 項演算子定義の実体はメタプログラミングであり、型情報を参照しながら適用の可否を判断している。つまり、本システムは型情報を利用したメタプログラミングを N 項演算子という概念によりラップしたものと考えられる。実際、図 3.4 のように、各 N 項演算子は解析後 static メソッドに変換され、それらを利用した演算は static メソッド呼び出しに変化する。また、readas が付与された引数は文字列リテラルとして読み込まれ、対応する型のコンストラクタ呼び出しに変換される。

```
1 SQLResult result =
2     SQLOperators.select_from_where(readasColumn("name"), employees,
3       SQLOperators.greaterEqual(readasColumn("TOEIC_score"), 600));
```

図 3.4: 図 3.1 の変換後擬似コード

3.2　特徴

本システムは従来の手法の問題点を克服しており、従来の手法と比べて容易に内部ドメイン専用言語の構築及び利用ができる。本節では従来の手法の問題点をどのように克服しているか、一つずつ確認していく。

3.2.1　表現能力

本システムは式の解析に字句・構文・型の情報を合わせて利用し書き換えるため、これらのうちの一つないし二つを利用する従来の手法よりも強力な表現能力を持つ。

字句情報の書き換えにより、特定の字句をキーワードとしたり、特定の位置に存在するトークンをリテラルとして扱うなどといったことが可能となる。例えば図 3.2 で言えば、select, from, where, **>=**をオペレータとして扱ったり、:col に対応する引数を Column 型のリテラルとして読むことを可能とする。これは字句置換マクロや構文マクロでもほぼ同様の機能が存在する。

また、構文情報の利用によりドメイン専用言語独自の構文を許すことができる。例えば図 3.2 では、

```
select :col from :table where :cond
:col >= :val
```

の二つの構文を定義し、図 3.1 はこの二つの構文により記述されている。Scala などの言語における省略規則を活用した手法は、メソッド呼び出しを二項演算子のように見せることはできるが、select :col from :table where :cond のような構文を許すことはできない。構文マクロを利用すればほとんど同様のことが可能だが、本システムの場合、構文解析を終えた構文木を操作するのではなく、N 項演算子による演算の構文木を直接構築するため、ホスト言語の文法では構文解析できないような文法でも問題なく許容できる。

更に、期待される型により選択する文法を変えるため、部分ごとに適切な N 項演算子を選択できる。例えば図 3.2 を見ると、:cond の部分には SQLCond 型が期待されることがわかる。そのため、図 3.1 の:cond に当たる部分では、SQLCond 型を返す:col ¿= :val がチェックされるので、where 句を括弧で括るなどの必要はない。

N 項演算子のオペレータを終端記号とし、各型を非終端記号と考えることにより、N 項演算子は BNF に書き換えることができる。ただし、各非終端記号 Tn は対応する型の変数を表す終端記号を vn として、規則 Tn → vn を持つ。逆に任意の BNF は、各非終端記号 Tn に対して規則 Tn → vn（vn は終端記号で変数を表す）を加えれば、vn 以外の終端記号をオペレータ、非終端記号を型と考えることにより N 項演算子に書き換えることができる。このことから、N 項演算子は BNF とほぼ同等の表現能力を持つといえる。

3.2.2　衝突の回避

本システムは型をスコープとすることで N 項演算子同士の衝突を回避している。期待される型によって利用する N 項演算子を判断するため、同一パターンの N 項演算子でも返り値の方が異なれば別の N 項演算子となり、それぞれそれが適切である場合にのみ呼び出される。そのため N 項演算子同士の衝突は発生せず、また N 項演算子の実態は static メソッドであるため局所変数名などの衝突も起こらず、健全性が保障される。

本システムの型によるスコープは Scheme の静的な構文スコープと異なり、衝突の回避を行う際に利用する文法を明示する必要がなく、また後から内部ドメイン専用言語の文法を拡張することも容易に行える。本システムは期待される型が一致する N 項演算子から利用されているものを捜すため、他の N 項演算子との衝突はそもそも考慮する必要がない。Scheme のように利用する文法ごとにマクロを作る必要はなく、単に N 項演算子を追加するだけで文法の拡張も可能である。そのため、複数の内部ドメイン専用言語を型を通じて連携させることも可能である。

本システムでは解釈が二通り以上存在するような曖昧な文がある場合、それを発見してユーザに通知する。これは内部ドメイン専用言語の衝突と言えるが、このような衝突は容易に回避できる。内部ドメイン専用言語独自の型を返す N 項演算子は衝突を引き起こさないので、衝突が発生するのは汎用的な型を返すような N 項演算子を定義した場合である。そのため、内部ドメイン専用言語を作成する際は、N 項演算子の返り値の型をその内部ドメイン専用言語独自の型にして、汎用的な型を返す N 項演算子を定義しないようにすれば良い。

3.2.3　分かりやすさ

本システムは、字句・構文・型の情報を利用したメタプログラミングを N 項演算子という高級な概念でラップすることにより、直感的に分かりやすい形で内部ドメイン専用言語を定義することを可能にしている。字句置換マクロや構文マクロはソースコードの改変を意識しながらプログラムを記述する必要があるため、マクロ定義やその使用方法が非直感的で分かりづらいものにならざるを得ない。本システムで利用するメタプログラミングは型情報を用いる分構文マクロ以上に強力であるため、そのまま利用すれば非常に難しく扱いづらいものとなってしまう。そこで、メタプログラミングを高級な概念により隠蔽することで分かりやすさ・扱いやすさの向上を図った。

N 項演算子による演算は実際には N 引数を持つメソッド呼び出しに変換されて実行されるが、N 項演算子の定義時及び利用時にそれを意識する必要はない。なぜならば、N 項演算子による演算を変換した結果は、必要な N 項演算子が定義された文法で構文解析した結果をホスト言語のコードに変換したものに過ぎないためである。メタプログラミングによって演算の優先順位が変化したり、名前衝突が発生するようなことも起こらないので、ソースコードが改変されることを意識する必要は全くない。また、N 項演算子の定義自体もメソッド定義と似た形式であるため、メソッド定義のような感覚で N 項演算子を定義することができる。

3.2.4　バグの発見

本システムは型によって N 項演算子を選択するため、型安全である。そのため、内部ドメイン専用言語で書かれたソースコードにバグがある場合、型エラーや構文エラーによって検出することが可能である。型が一致しないような N 項演算子は構文解析に利用されないため、N 項演算子による型エラーは発生しない。間違った N 項演算子を利用していた場合、構

文解析に失敗して構文エラーが発生するため、バグを発見することができる。また、N 項演算子による演算以外の部分も構文解析と同時に型チェックが行われるため、型エラーによるバグの発見が可能となっている。

第4章　実装

本研究では Java 言語からインターフェースや内部クラス、演算子及び一部の制御構造などを除いた Java サブセットのコンパイラを作成し、それに対して本研究のシステムを組み込んだ。本章ではまずコンパイラの基礎知識について述べ、その後本システムの実装について詳しく解説する。

4.1　コンパイラの基礎

簡単なコンパイラは通常、字句解析・構文解析・意味解析・コード生成の 4 つのフェーズに分かれる。字句置換マクロによる置き換えは字句解析と構文解析の間で行われ、構文マクロは構文解析と意味解析の間で処理される。多くのコンパイラは意味解析とコード生成の間で最適化を行う。

4.1.1　字句解析

字句解析ではプログラムのソースコードを読み込み、それぞれをトークンと呼ばれる断片に分割する。トークンはプログラムのソースコード上で意味を持つコード断片の最小単位で、識別子やリテラル、括弧等の記号などがそれにあたる。字句解析器は通常有限オートマトンにより表現できる。

4.1.2　構文解析

構文解析ではトークン列を読み込みプログラムの構造を分析して、抽象構文木を呼ばれる木構造を作成する。抽象構文木を構成するための構文の文法規則は通常文脈自由文法により記述される。構文解析の手法はトップダウン構文解析とボトムアップ構文解析に大別される。

トップダウン構文解析は文法規則の開始記号から出発して生成規則を順に適用していく手法で、構文木の根から葉に向かって順に解析が進む。この手法は構文解析器の構成が比較的容易であるが、単純なトップダウン構文解析では左再帰を含む文法の解析ができない。左再帰とは、非終端記号 A に数回文法規則を適用することで再び左端に A が現れることを言う。

このような左再帰を含む文法を単純なトップダウン構文解析で解析すると無限再帰に陥ってしまう。トップダウン構文解析の代表例としては LL 法が挙げられる。

ボトムアップ構文解析はトップダウン構文解析とは反対に終端記号から出発して生成規則を逆向きに適用し、最終的に開始記号を導く手法で、構文木の葉から根に向かって解析が進む。この手法は表現できる文法の制限が少なく様々な言語を記述することができるが、構文解析器を人手で構築するのは困難で、yacc などの構文解析器生成系を利用する必要がある。ボトムアップ構文解析の例としては、LALR 法や演算子順位法が知られている。

4.1.3　意味解析

意味解析では抽象構文木を受け取り、型チェックによる不正プログラムの検出や変数名の名前解決等を行う。そのためこの時点で型やフィールド等の情報が揃っている必要がある。

4.1.4　コード生成

コード生成では解析結果を出力する言語に変換する。通常のコンパイラでは意味解析の結果を中間コードとして出力し、それに対して最適化を行った後にコード生成を行う。

4.2　本システムの実装の概要

4.2.1　宣言部分と本体部分を分ける

本システムは通常のコンパイラと異なり、プログラムの宣言部分と本体部分を分けて解析する必要がある。本システムでは求められる型を利用して N 項演算子を選択するため、式部分の解析前にその場所で求められる型が分からなければならない。そのため、型やメソッド・フィールド等を式部分の解析前にすべて知っておく必要があるので、本体部分のコンパイルより先に宣言部分のコンパイルを終えておき、全てのクラスおよび N 項演算子についての情報を揃えておかなければならない。今回の実装では本体部分を解析せずに一旦読み飛ばし、後から本体部分を解析するという手法を取ったが、C/C++言語のように宣言部分をヘッダファイルに記述することで本体部分と分割する手法も考えられる。

4.2.2　字句解析・構文解析・型チェックの連携

本システムでは式部分の解析時に字句解析・構文解析・型チェックが連携して動作する。式部分では字句解析と構文解析の連携により、N 項演算子のオペレータをキーワードとして扱ったり、トークンの文字列化を行うなどの構文による字句ルールの変更を行う。構文解析と型チェックの連携により期待される型に応じた N 項演算子の選択を行う。

4.2.3　トップダウン構文解析

本体部分の構文解析はトップダウン構文解析でなければならない。本システムでは式を解析するためにその式の型を利用するので、局所変数定義では初期値よりも先にその変数の型が、メソッドや N 項演算子の呼び出しでは引数の値よりも先にメソッドや N 項演算子が分からなければならない。そのため、構文木の根から葉に向かって解析していく、トップダウン構文解析が必要になる。

4.3　宣言部分の解析

4.3.1　宣言部分とは

宣言部分とはプログラムの実際の挙動を示すメソッドボディ等の本体部分を除いた部分のコードを指す。具体的には、パッケージ宣言・インポート宣言全体とクラス宣言からメソッド・コンストラクタの本体とフィールド初期値を除いた部分、及び N 項演算子の定義からその本体を除いた部分のことである。

これらを本体部分の解析前に解析することで、本体部分で現れるクラスやメソッド・N 項演算子の解決が可能となる。

4.3.2　字句・構文解析

原則的には LL 法により構文解析を行う。メソッド・コンストラクタの本体とフィールド初期値の部分は解析せずにひとつのトークンのように扱う。そのため、メソッド・コンストラクタ・フィールドの構文解析を行う際は字句解析も連携して解析を行う。

本体部分をまとめてひとつのトークンのようにして扱うためにはその終了箇所が分からなければならない。フィールド初期値の終了箇所はセミコロンで、メソッドボディの終了箇所は開始箇所に存在する開き中括弧に対応する閉じ中括弧であるため、括弧やシングルクォート・ダブルクォート

の対応関係をチェックしながら終了箇所を探す必要がある。そのため、対応関係を崩すような記号は N 項演算子のオペレータとして利用できない。（参照：3.1.1 節）また、コメント中のセミコロンや中括弧は終了箇所とはなり得ず無視するため、コメントの開始または終了を表す文字列を N 項演算子のオペレータとして利用することを許すと対応関係が崩れるおそれがある。そのため、コメントの開始または終了を表す文字列はオペレータに含めることができない。（参照：3.1.1 節）ここで、コメントは本体部分の解析を行う際にもやはり不要なので、無視すると同時に削除しておく。また、複数連続した空白文字も本体部分の解析時には無意味なので、ここで排除しておく。

4.3.3　意味解析

宣言部分の構文解析終了後、全てのクラスの情報を登録したクラスプールと全ての N 項演算子を登録したオペレータプールを作成し、ソースファイルごとにパッケージ宣言及びインポート宣言を基にして、型名を解決する TypeNameResolver を作成する。TypeNameResolver はクラスプールへの参照を持ち、そこから必要な情報を検索する。本体部分の解析では、この TypeNameResolver を基にした NameResolver を作成し、そこに局所変数などの情報を登録して名前解決を行う。N 項演算子はオペレータプールに登録するのと同時に static メソッドとしてクラスプールにも登録する。これは、N 項演算子を最終的に static メソッドに変換するためである。

4.4　本体部分の解析

本体部分の構文は式部分を除けば単純なトップダウン構文解析により解析できる。式部分は他の部分と解析手法が異なるため、別の解析器に分離する。構文解析が式部分に到達すると、その終了地点とそこで要求される型を探して、式部分の解析器に解析を任せる。

式部分の解析には局所変数や仮引数の名前及び型の情報を利用するため、名前解決を行う NameResolver にそれらの情報を登録し、式部分の解析を行う際にその解析器に NameResolver を渡す。NameResolver はその本体部分のコードが記述されたソースファイルの TypeNameResolver が基となっており、また、スコープごとに異なる NameResolver が作られる。

4.5 式部分の解析

式部分は大きく分けて次の三つの種類に分けられる。

- N 項演算
- 括弧で括られた式
- 通常の Java の式

ユーザが定義した N 項演算子を用いた式が N 項演算である。N 項演算は期待される型及び優先度を利用して解析を行う。N 項演算の解析は特殊なため、次節にてその解説を行う。本節ではそれ以外の式の解析について述べる。

4.5.1 解析順序

式部分の解析を行う場合はまず N 項演算を試し、その解析が失敗した場合、括弧で括られた式を試し、それも失敗した場合に通常の Java の式を試す。そのため、通常の Java の式と N 項演算のどちらにでも解釈できる場合は N 項演算が優先される。

4.5.2 括弧で括られた式の解析

括弧で括られた式を解析する場合、新しい解析器にその部分で期待されている型と閉じ括弧の位置を渡して処理を任せる。これにより N 項演算子の優先度が初期化されるため、この括弧は通常の数式における括弧と同様に振舞う。

4.5.3 通常の Java の式

本システムの実装では、通常の Java の式は型を見ながらボトムアップ構文解析により構文解析を行った。その理由は二つあり、一つはメソッド呼び出しやフィールドアクセスが左再帰的なので、トップダウン構文解析による解析が難しいためである。もう一つの理由のほうがより重要で、通常の Java の式の解析では、期待される型の情報を利用しないので、トップダウン構文解析である必要がないためである。

メソッドやクラスインスタンス生成式の引数は式であるため、ここは引数として要求される型とその引数の終了地点を探して、新しい解析器に渡して処理を任せる。引数の型を知るためにはどのようなメソッドが呼ばれ

ているかを知る必要があるため、型情報が必要となる。引数の終了地点はカンマまたは閉じ括弧により判断するため、カンマと閉じ括弧はオペレータに含めることができない。（参照：3.1.1 節）

通常の Java の式は他の二つの式と異なり、構文解析が終了した後で期待される型との型チェックを行う。そのため期待される型の情報は構文解析時には利用しないが、解析が終わった部分の型情報は構文解析に利用する。前述したように、これはメソッド呼び出しなどの引数部分を解析する際に、引数の型が必要になるため、どのような型のどのメソッドが呼ばれているか判断できる必要がある。そのため、常に構文解析が終了している部分については型情報も持っている必要がある。

4.6　N 項演算の解析

4.6.1　N 項演算子の試行

本システムでは期待される型によって使用される N 項演算子を制限する。そのため、宣言部分で作成したオペレータプールから返り値の型が指定された型と一致するものを取得し、その中から N 項演算子を特定する。

全ての N 項演算子は優先度を持ち、それによって N 項演算子の結合の仕方が変化する。例えば優先度が 100 である N 項演算子の各項には優先度が 100 未満の N 項演算子による式が入ることはない。そのため、N 項演算の解析の際には最低優先度を指定する必要がある。優先度が最低優先度未満の N 項演算子は適用されないので、期待される型を基にしてオペレータプールから取得した N 項演算子のうち、優先度が最低優先度を超えないものは試行を行わない。

N 項演算子の試行では、パターンを前から順にチェックしていく。オペレータが現れた場合、字句解析器に働きかけて次に現れるトークンがそのオペレータかどうかをチェックする。オペランドならば、それと対応する型を返すような式が現れるはずなので、再帰的にその式を解析する。ただし、readas 修飾子が付いていた場合は次のトークンをリテラルとして読む。

4.6.2　左再帰型 N 項演算子

N 項演算子には左再帰型のものが存在する。N 項演算子のパターンは幾つかのオペレータとオペランドの列であればよいので、パターンの最左部分がオペランドで、そのオペランドの対応する型が N 項演算子の返り値の型と一致するような左再帰型のパターンも許される。次の図 4.1 の:left and :right は左再帰型 N 項演算子の例である。

```
1  operators SQLOperators {
2      SQLCond :col >= :val(readas Column col, int val)
3          : priority = 200 { ... }
4
5      SQLCond :col < :val(readas Column col, int val)
6          : priority = 200 { ... }
7
8      SQLCond :left and :right(SQLCond left, SQLCond right)
9          : priority = 150 { ... }
10 }
11
12 SQLCond condition
13     = TOEIC_score >= 600 and age < 40;
```

図 4.1: 左再帰型 N 項演算子を含む例

4.1.2 節で述べたように、トップダウン構文解析は通常左再帰を含む文法を扱うことができない。これを解決する方法は左再帰性の除去をはじめとして幾つか知られているが、本システムでは [2] のアルゴリズムを採用した。このアルゴリズムはメモ化と左再帰カウンタを利用することで、左再帰を含む文法をトップダウン構文解析で解析する。この手法の利点は左再帰性の除去と異なり文法を変更せずに左再帰を扱うことができるところである。構文の解析を行う際にバックトラックを繰り返すため計算速度は遅いが、このアルゴリズムを利用するのは式部分だけなので、それほど大きなオーバーヘッドはないと考えられる。次節及び次々節ではメモ化と左再帰カウンタについて述べる。

4.6.3　メモ化

メモ化とはトップダウン構文解析において、完成した部分木の情報をメモテーブルと呼ばれる構造体に残すことで、バックトラックにより再び同じ部分の解析が行われた場合にその情報を再利用して再計算を防ぐ手法である。例えば、図 4.2 のコードは二項演算子 select :col from :table の試行が先に行われた場合、:col や:table の部分の解析が二度行われる。

type	priority	begin	end	memo
readas Column	-	28	33	name
Table	-	38	48	employees
SQLResult	100	21	48	select :col from :table ((readas Column, 28, 33), (Table, 38, 48))

表 4.1: select :col from :table の試行後のメモテーブル

select :col from :table の試行が終了した後のメモテーブルは表 4.1 のよ

```
1  operators SQLOps {
2      SQLResult select :col from :table
3          (readas Column col, Table table)
4           : priority = 100 { ... }
5
6      SQLResult select :col from :table where :cond
7          (readas Column col, Table table, SQLCond cond)
8           : priority = 100 { ... }
9
10     SQLCond :col >= :val(readas Column col, int val)
11          : priority = 200 { ... }
12 }
13
14 メソッド {
15     SQLResult result
16         = select name from employees where TOEIC_score >= 600;
17     ...
18 }
```

図 4.2: バックトラックにより同じ部分の解析が行われる場合の例

うになる。メモテーブルの一列目は必要とされる型で、二列目は最低優先度である。これは、本システムが期待される型と優先度により N 項演算子を選択するために記述される。優先度が-となっているのは、N 項演算以外の式であることを示している。メモテーブルの三列目と四列目はそれぞれ式の先頭と終端の位置である。メモテーブルの五列目がメモの本体で、完成した部分木の情報が書かれている。例えば、三行目は SQLResultn 型の優先度 100 の N 項演算が (21, 48) の位置に存在しており、N 項演算子は select :col from :table で、第一引数は readas Column 型でその位置は (28, 33)、第二引数は Table 型でその位置は (38, 48) である事を示している。

メモテーブルの各行はそれぞれ完成した部分木を表している。例えば表 4.1 の第三行の情報を基にして構文木を構築すると図 4.3 のようになる。これは構文木としては正しいが、文の終端に達していないため構文解析の結果としては正しくない。

select :col from :table where :cond の試行を行うとき、表 4.1 のメモテーブルの一行目及び二行目の情報を利用することができる。表 4.5 は select :col from :table where :cond の試行が終了した後のメモテーブルである。

このメモテーブルの七行目の情報から構文木を構成すると図 4.4 のようになる。こちらは文の終端に達しており、構文解析の結果として正しい。

メモ化は高速化だけでなく、記憶領域の縮小の意味合いも持っている。プログラムのコードに対して複数の解釈が可能であった場合、その構文解析を行う場合には、解釈ごとに構文木を作る必要があった。しかし、メモ化を利用すればメモテーブルの行が一つ増加するだけであるため、必要な

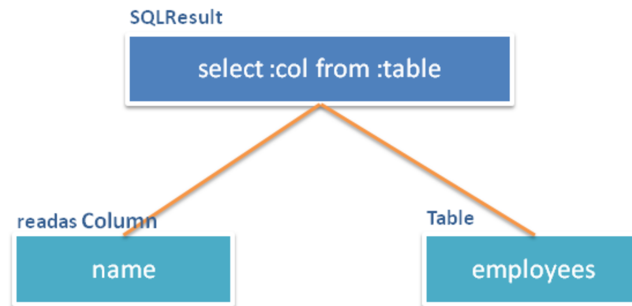

図 4.3: 表 4.1 の第三行の情報を基に構築した構文木

type	priority	begin	end	memo
readas Column	-	28	33	name
Table	-	38	48	employees
SQLResult	100	21	48	select :col from :table ((readas Column, 28, 33), (Table, 38, 48))
readas Column	-	54	66	TOEIC_score
int	-	69	72	600
SQLCond	200	54	72	:col >= :val ((readas Column, 54, 66), (int, 69, 72))
SQLResult	100	21	72	select :col from :table where :cond ((readas Column, 28, 33), (Table, 38, 48) ,(SQLCond, 54, 72))

表 4.2: select :col from :table where :cond の試行後のメモテーブル

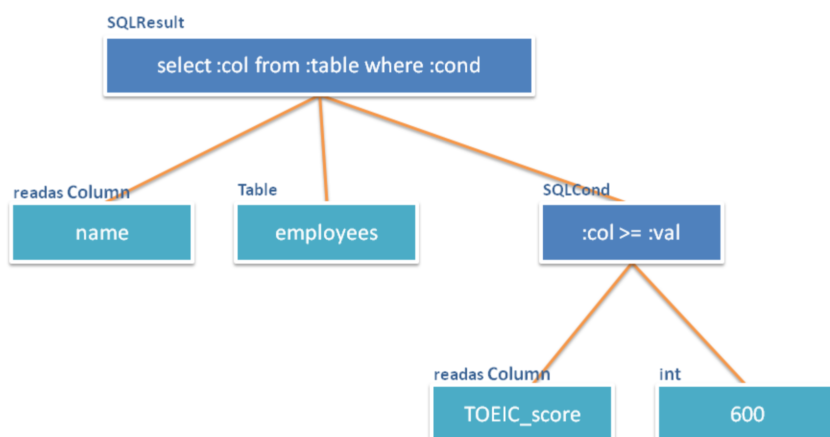

図 4.4: 表 4.5 の第七行の情報を基に構築した構文木

記憶領域は大きく削減される。

このことから、本システムは複数の解釈が存在するような場合も構文解析が可能で、それらの解釈をユーザに伝えることができる。

4.6.4　左再帰カウンタ

トップダウン構文解析において左再帰を扱うことができないのは、左再帰が無限再帰に陥ってしまい、構文解析が停止しなくなってしまうためである。例えば、図 4.1 を単純なトップダウン構文解析で解析しようとした場合、図 4.5 のように無限再帰に陥ることとなる。左再帰をトップダウン構文解析で扱うためには、この無限再帰をどこかで停止させれば良い。

そこで、左再帰カウンタにより左再帰が行われた回数をカウントして、残りのトークンが足りなくなったところで再帰を停止させる。これにより構文解析が停止するため、左再帰もトップダウン構文解析で扱うことが可能となる。この手法は非常に多くのバックトラックが必要となるが、メモ化により再計算が防止されるためそのオーバーヘッドは小さい。本研究では期待される型によって字句ルールも変更されるため、トークンの切り方も変わる可能性がある。そのため、残りのトークン数の代わりに残りの文字数を利用して再帰を停止させる。

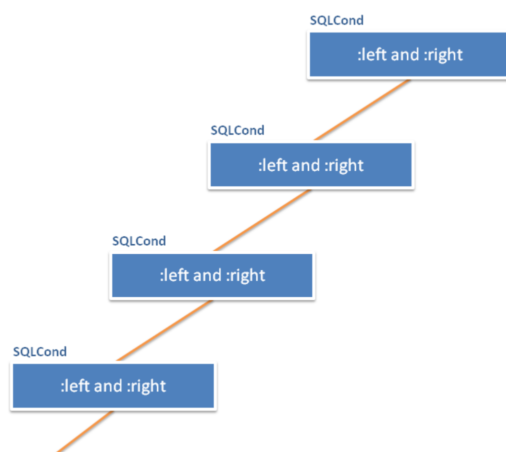

図 4.5: 図 4.1 を単純なトップダウン構文解析で解析した場合

4.7　式部分の解析の例

本節では図 4.6 の

```
select name from employees where TOEIC_score >= 600 and age < 40;
```

の部分の解析がどのように行われるかを追う。

1. 左辺より、式部分の期待する型は SQLResult 型であることがわかっているので、select :col from :table where :cond を試す。
2. select はオペレータなので次に select が現れることを確認し、次へ進む。
3. :col はオペランドであり、対応する型は readas Column なので、次のトークン name を Column 型のリテラル（new Column(”name”)）として読む。結果はメモテーブルに記録しておく。
4. from はオペレータなので次に from が現れることを確認する。
5. :table はオペランドであり、対応する型は Table である。Table 型を返す N 項演算子は存在せず、次のトークンは開き括弧ではないので、通常の Java の式として解析を行い Table 型のオブジェクト employees を得る。結果はメモテーブルに記録する。

```
1  operators SQLOperators {
2      SQLResult select :col from :table where :cond
3          (readas Column col, Table table, SQLCond cond)
4           : priority = 100 { ... }
5
6      SQLCond :left and :right(SQLCond left, SQLCond right)
7           : priority = 150 { ... }
8
9      SQLCond :col >= :val(readas Column col, int val)
10          : priority = 200 { ... }
11
12     SQLCond :col < :val(readas Column col, int val)
13          : priority = 200 { ... }
14 }
15
16 SQLResult result
17  = select name from employees where TOEIC_score >= 600 and age < 40;
```

図 4.6: 図 2.2 の再現例

6. where はオペレータなので次に where が現れることを確認する。

7. :cond は SQLCond 型を要求するオペランドなので、SQLCond 型を期待する式として再帰的に解析する。ここで、select :col from :table where :cond の優先度は 100 であり、:cond はパターンの末尾なので、引数となる式の優先度は 100 より大きいことが要求される。

8. SQLCond 型を返すような N 項演算子は、:left and :right と:col >= :val と:col < :val の三つが定義されている。:left and :right は優先度が 150 であり、:col >= :val と:col < :val は優先度が 200 であるため、まずは:left and :right から試していく。

9. オペランド:left は引数として SQLCond 型を要求し、また優先度は 150 以上であることが求められる。そのため、再び:left and :right を試すこととなる。このようにして左再帰を繰り返すと 8 回目の再帰で残りの文字数が足りなくなり再帰が失敗する。（図 4.7 参照）そこで、失敗した再帰の代わりに:col >= :val を試す。

10. :col は readas Column 型を要求するオペランドなので、TOEIC_score を Column 型リテラルとして読み、メモテーブルに記録する。

11. >=はオペレータなので次に>=が現れることを確認する。

12. :val は int 型を要求するオペランドなので、ここを再帰的に解析する。int 型を返す N 項演算子は存在せず[1]、開き括弧もないので通常

[1]今回の例では簡単のためプリミティブな演算子はないものとしている。

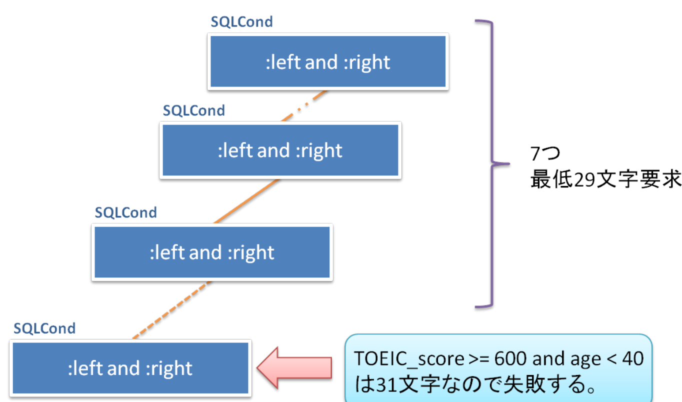

図 4.7: 左再帰カウンタによる左再帰の停止

の Java の式として解析を行う。解析で得られた int 型のリテラル 600 はメモテーブルに記録しておく。

13. :col >= :val の試行が成功したので、結果をメモテーブルに記録して:col >= :val の一つ上の:left and :right に戻る。ここまでのメモテーブルは次の表 4.3 のようになる。ただし、簡単のためこの select 文の文頭を位置 0 とする。構文木は現在、図 4.8 のようになっている。
14. オペレータ and が確認できるので、:right の処理に移行する。
15. SQLCond 型を要求するオペランド:right は、N 項演算子のパターンの末尾であるため、引数の式に対して 150 より大きい優先度を要求する。そのため、優先度が 200 の:col >= :val と:col < :val を試す。まずは:col >= :val から試していく。
16. :col は readas Column 型を要求するオペランドなので、age を Column 型リテラルとして読んでメモテーブルに記録する。
17. オペレータ>=を確認する。次のトークンは>=ではないので、:col >= :val の試行は失敗する。次に:col < :val を試す。
18. :col は readas Column 型を要求するオペランドで、メモテーブルに記録が残っているためそれを利用する。
19. オペレータ<を確認する。
20. :val は int 型を要求するオペランドである。int 型を返す N 項演算子は存在せず、次のトークンは開き括弧ではないので、通常の Java の

type	priority	begin	end	memo
readas Column	-	7	12	name
Table	-	17	27	employees
readas Column	-	33	45	TOEIC_score
int	-	48	52	600
SQLCond	200	33	52	:col >= :val ((readas Column, 33, 45), (int, 48, 52))

表 4.3: :col >= :val の試行後のメモテーブル

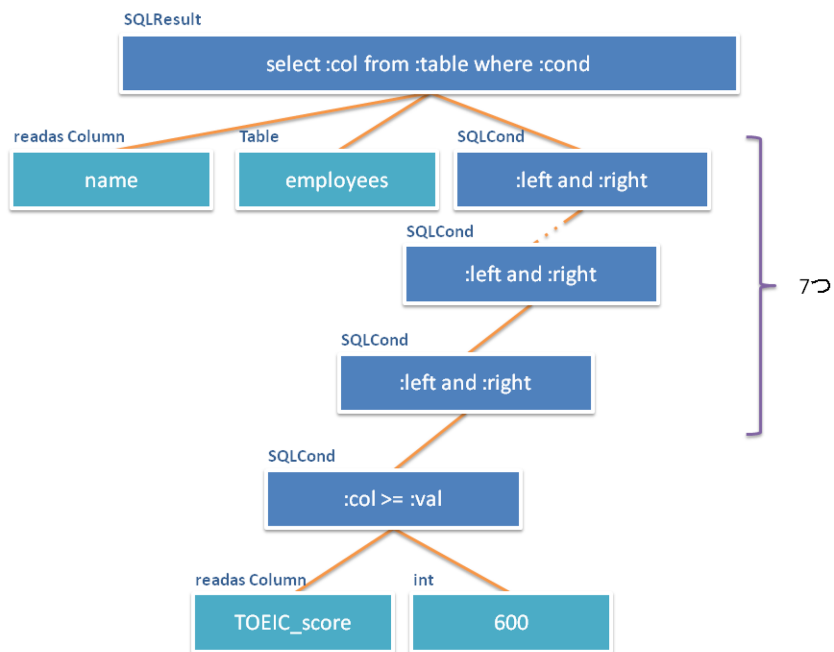

図 4.8: :col >= :val の試行終了時の構文木

式として解析する。結果として得られた int 型のリテラル 40 はメモテーブルに記録しておく。

21. :col < :val の試行が成功したので、結果をメモテーブルに記録して:col < :val の一つ上の:left and :right に戻る。

22. :left and :right の試行が成功したので、結果をメモテーブルに記録してもう一つ上の:left and :right に戻る。ここまでのメモテーブルは次の表 4.4 のようになる。構文木は現在、図 4.9 のようになっている。

type	priority	begin	end	memo
readas Column	-	7	12	name
Table	-	17	27	employees
readas Column	-	33	45	TOEIC_score
int	-	48	52	600
SQLCond	200	33	52	:col >= :val ((readas Column, 33, 45), (int, 48, 52))
readas Column	-	56	60	age
int	-	62	64	40
SQLCond	200	56	64	:col < :val ((readas Column, 56, 60), (int, 62, 64))
SQLCond	150	33	64	:left and :right ((SQLCond, 33, 52), (SQLCond, 56, 64))

表 4.4: 最も深い位置にある:left and :right の試行後のメモテーブル

23. オペレータ and は次のトークンでないため、:left and :right の試行は失敗する。メモテーブルに返り値が SQLCond 型で、優先度 150、位置 33 のものが登録されているので、それを結果の式として一つ上の:left and :right に戻る。これを繰り返し、select :col from :table where :cond まで戻る。

24. select :col from :table where :cond の試行が成功したので、結果をメモテーブルに記録する。この時のメモテーブルは表 4.5 のようになる。

25. メモテーブルを参照し、返り値が SQLResult 型で、この式の開始位置 0 で始まり、終了位置 64（セミコロンの位置から算出）で終わるものを探す。表 4.5 のメモテーブルの十行目がその条件をみたすの

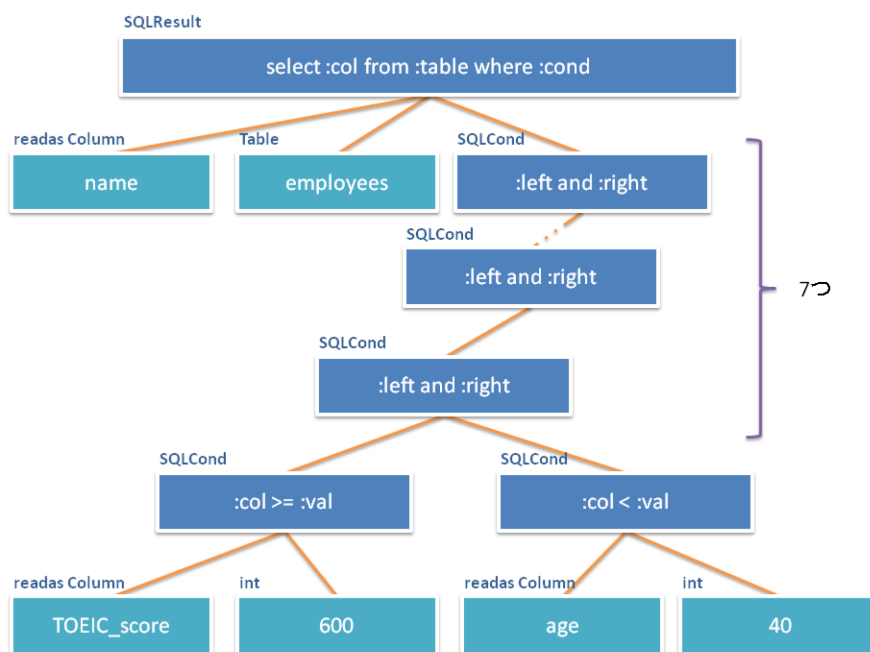

図 4.9: 最も深い位置にある:left and :right の試行終了時の構文木

type	priority	begin	end	memo
readas Column	-	7	12	name
Table	-	17	27	employees
readas Column	-	33	45	TOEIC_score
int	-	48	52	600
SQLCond	200	33	52	:col >= :val ((readas Column, 33, 45), (int, 48, 52))
readas Column	-	56	60	age
int	-	62	64	40
SQLCond	200	56	64	:col < :val ((readas Column, 56, 60), (int, 62, 64))
SQLCond	150	33	64	:left and :right ((SQLCond, 33, 52), (SQLCond, 56, 64))
SQLResult	100	0	64	select :col from :table where :cond ((readas Column, 7, 12), (Table, 17, 27) , (SQLCond, 33, 64))

表 4.5: select :col from :table where :cond の試行終了後のメモテーブル

で、その情報から構文木を復元し、それをこの式の解析結果とする。解析結果の構文木は図 4.10 のようになる。

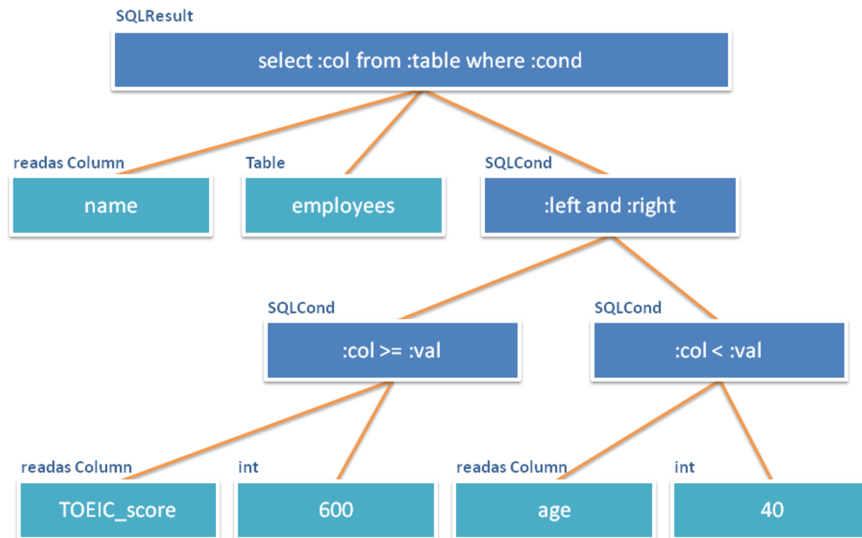

図 4.10: 解析結果の構文木

第5章　ケーススタディ

本章では、本システムの具体的な利用法を例を用いて説明する。前章まではSQLを例として挙げていたので、本章ではそれと異なる例を用いる。

5.1　四則演算

本研究のN項演算子は、当然ではあるが、四則演算を実現することができる。図5.1は四則演算を実現するN項演算子の定義例である。

```
1 operators IntOps {
2     native int :left + :right(int left, int right) : priority = 100;
3     native int :left - :right(int left, int right) : priority = 100;
4     native int :left * :right(int left, int right) : priority = 200;
5     native int :left / :right(int left, int right) : priority = 200;
6 }
```

図 5.1: 四則演算を実現するN項演算子の定義例

native修飾子は定義したN項演算子がJavaのプリミティブな演算子であることを示す。処理の内容はJavaの演算子によるので、N項演算子のパターンのみ記述して、具体的な処理は記述しない。

図5.2は図5.1のN項演算子を利用したコードの例である。このコードをコンパイルすると図5.3のようなコードになる。このコードを見ると、正しい優先順位に沿って解析できていることがわかる。

```
1 int foo() {
2     int x = 2 * 3 + 4 - 5 / 6;
3     int y = 1 - 2 + 3 - 4 + 5;
4     return x * y;
5 }
```

図 5.2: 図5.1のN項演算子の利用例

図5.1の優先順位を異なる順序に設定すると、図5.2のコンパイル結果は図5.3と異なるものとなる。例えば、定義された四つのN項演算子の優先度を全て等しく設定すると、図5.2をコンパイルした結果は図5.4の

```
1 int foo() {
2     int x = ( ( ( 2 * 3 ) + 4 ) - ( 5 / 6 ) ) ;
3     int y = ( ( ( ( 1 - 2 ) + 3 ) - 4 ) + 5 ) ;
4     return ( x * y ) ;
5 }
```

図 5.3: 図 5.2 のコンパイル後のコード

ようになる。また、:left + :right と:left - :right の優先度と、:left * :right と:left / :right の優先度を逆転させた場合、図 5.2 をコンパイルした結果は図 5.5 のようになる。

```
1 int foo() {
2     int x = ( ( ( ( 2 * 3 ) + 4 ) - 5 ) / 6 ) ;
3     int y = ( ( ( ( 1 - 2 ) + 3 ) - 4 ) + 5 ) ;
4     return ( x * y ) ;
5 }
```

図 5.4: 優先順位を全て等しくした場合の図 5.2 のコンパイル後のコード

```
1 int foo() {
2     int x = ( ( 2 * ( ( 3 + 4 ) - 5 ) ) / 6 ) ;
3     int y = ( ( ( ( 1 - 2 ) + 3 ) - 4 ) + 5 ) ;
4     return ( x * y ) ;
5 }
```

図 5.5: 優先順位を逆転させた場合の図 5.2 のコンパイル後のコード

5.2　BNF

次に本システムを用いて BNF を記述してみよう。BNF はバッカス・ナウア記法の略で、文脈自由文法を表現するためによく利用されるドメイン専用言語の一種である。

BNF は導出規則の集合であり、それぞれの規則は

非終端記号 ::= 式

の形で記述される。規則の右辺の式は終端記号と非終端記号からなる記号列か、またはそのような記号列を or を表す記号|でつないだものとなる。非終端記号は導出規則の左辺に現れる記号を指し、

< 文字列 >

のような形で記述される。終端記号は導出規則の右辺にのみ現れる記号で、そのBNFが表現する文法によって生成される文字列の要素である。

図5.6はBNFを表現するN項演算子の定義例である。この例において、Nonterminal及びTerminalはBNFExprのサブクラスとして定義している。

```
1  operators BNFOps {
2      BNFExpr :term (String term)
3       : priority = 250 {
4          return new Terminal(term);
5      }
6
7      Nonterminal < :name > (readas String name)
8       : priority = 250 {
9          return new Nonterminal(name);
10     }
11
12     BNFExpr < :name > (readas String name)
13      : priority = 250 {
14         return new Nonterminal(name);
15     }
16
17     BNFExpr :sym1 :sym2(BNFExpr sym1, BNFExpr sym2)
18      : priority = 200 {
19         return sym1.append(sym2);
20     }
21
22     BNFExpr :sym1 | :sym2(BNFExpr sym1, BNFExpr sym2)
23      : priority = 150 {
24         return sym1.or(sym2);
25     }
26
27     BNFRule :symbol ::= :expr(Nonterminal symbol, BNFExpr expr)
28      : priority = 100 {
29         return new BNFRule(symbol, expr);
30     }
31 }
```

図5.6: BNFを表現するN項演算子の定義例

このように定義することで、例えば次の図5.7のようなコードの記述が可能となる。makeNumberBNFメソッドによって得られるBNFは非負の整数を生成する文脈自由文法を表現している。

図5.6で< :name >が二つ定義されているのは、N項演算子は返り値の型がその場所で求められている型と一致したときにのみ利用されるためである[1]。BNFRuleの左辺では、< :name >はNonterminalとして求められるが、BNFRuleの右辺では、< :name >はBNFExprとして求められるため、二つ定義する必要がある。

図5.7のコードをコンパイルすると、図5.8のように解釈される。ただ

[1] 3.1.4節参照

```
1  BNF makeNumberBNF() {
2      BNFRule rule1 = <digit_non_zero>
3       ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
4      BNFRule rule2 = <digit>
5       ::= <digit_non_zero> | "0";
6      BNFRule rule3 = <digit_repetition>
7       ::= <digit> | <digit> <digit_repetition>;
8      BNFRule rule4 = <number>
9       ::= <digit> | <digit_non_zero> <digit_repetition>;
10
11     return new BNF(rule1, rule2, rule3, rule4);
12 }
```

図 5.7: 図 5.6 の N 項演算子の利用例

し簡単のため、各 N 項演算子をコンパイルしてできる static メソッドのメソッド名は、パターンのオペランドを_に変え、$を用いて返り値とパターンをつないだものとする。

```
1  BNF makeNumberBNF() {
2      BNFRule rule1 =
3       BNFRule$_::=_(
4        Nonterminal$<_>(readasString("digit_non_zero")),
5        BNFExpr$_|_(BNFExpr$_|_(BNFExpr$_|_(...),
6                                BNFExpr$_("8")),
7                    BNFExpr$_("9")));
8      BNFRule rule2 =
9       BNFRule$_::=_(
10       Nonterminal$<_>(readasString("digit")),
11       BNFExpr$_|_(BNFExpr$<_>(readasString("digit_non_zero")),
12                   BNFExpr$_("0")));
13     BNFRule rule3 =
14      BNFRule$_::=_(
15       Nonterminal$<_>(readasString("digit_repetition")),
16       BNFExpr$_|_(BNFExpr$<_>(readasString("digit")),
17                   BNFExpr$__(
18                    BNFExpr$<_>(readasString("digit")),
19                    BNFExpr$<_>(readasString("digit_repetition")))));
20     BNFRule rule4 =
21      BNFRule$_::=_(
22       Nonterminal$<_>(readasString("number")),
23       BNFExpr$_|_(BNFExpr$<_>(readasString("digit")),
24                   BNFExpr$__(
25                    BNFExpr$<_>(readasString("digit_non_zero")),
26                    BNFExpr$<_>(readasString("digit_repetition")))));
27
28     return new BNF(rule1, rule2, rule3, rule4);
29 }
```

図 5.8: 図 5.7 のコンパイル後の擬似コード

第6章　まとめと今後の課題

6.1　まとめ

プログラムの開発に内部ドメイン専用言語を利用することは、ソースコードの可読性を向上させ、生産性・保守性を高めることにつながる。しかし、内部ドメイン専用言語を実現する従来の手法には多くの問題点があった。従来の手法は実現できる文法に制限があり、ホスト言語が解析できないような構文は使うことができなかった。複数の内部ドメイン専用言語を同時に利用する場合には、それらが衝突して正常に動作しなくなる危険性もあった。また、内部ドメイン専用言語の構築には高度な知識が必要であり、その上内部ドメイン専用言語で記述されたプログラムはバグの発見が困難となるという弊害も存在した。

これらの問題点を解決するために、本研究では期待される型によって字句・構文ルールを切り替える手法を提案・実装した。具体的には、内部ドメイン専用言語の字句・構文ルールを N 項演算子の形で定義し、期待される型によって利用される N 項演算子を選択することによって内部ドメイン専用言語を実現する。本システムは、字句・構文ルールが構文木の構築前に切り替わるため、ホスト言語が解析できない文法でも解析することができる。また、期待される型による字句・構文ルールの切り替えにより、内部ドメイン専用言語ごとに別々の解析ルールが適用されるため、内部ドメイン専用言語同士の衝突回避も同時に実現している。また、N 項演算子による内部ドメイン専用言語の定義は直感的で分かりやすく、高度な知識を必要としない。更に、本システムは型安全で構文エラーや型エラーによって内部ドメイン専用言語で記述されたプログラムのバグを検出することができる。

Java 言語からインターフェースや内部クラスなどを除いた Java サブセットのコンパイラを作成し、そのコンパイラに対して本研究の手法を導入した。作成したコンパイラを利用して SQL や BNF を記述できることを確認した。

6.2　今後の課題

6.2.1　表現力の強化

本システムはホスト言語が解析できない文法の解析が可能であるが、利用できない文字が存在するなど、表現力にはまだ不足している点がある。内部ドメイン専用言語の記述をより容易にするために、オプションや 0 回以上の繰り返しの記述に対応したり、括弧等のセパレータをパターンとして記述できるようにすることが今後の課題である。

また、現在の N 項演算子は代入文を表現することができない。これには色々な問題が関わっているが、そのなかで最も大きい問題となっているのは型による制限が強すぎることであろう。この制限を緩めることで更に強力な表現能力を得ることができると考えられる。

6.2.2　readas の制限

本システムは型によって利用可能な N 項演算子を制限しているが、readas を濫用すると予期せぬ動作を引き起こすおそれがある。そのため、readas に制限を加えるなどの対応が必要となる。例えば、静的な構文スコープを N 項演算子に対して導入するなどの手法が考えられる。問題となるパターンを精査し、最適な解決法を模索したい。

6.2.3　コンパイル時のオーバーヘッドの評価

本システムは式のコンパイル時にバックトラックを繰り返すため、大きなオーバーヘッドがかかる。このオーバーヘッドを測定して本システムが実用に耐えうるかを判断することも今後の課題の一つである。

参考文献

[1] Fowler, M.: Language Workbenches: The Killer-App for Domain Specific Languages?, http://www.martinfowler.com/articles/languageWorkbench.html.

[2] Frost, R. A., Hafiz, R. and Callaghan, P. C.: Modular and efficient top-down parsing for ambiguous left-recursive grammars, *Proceedings of the 10th International Conference on Parsing Technologies*, IWPT '07, Stroudsburg, PA, USA, Association for Computational Linguistics, pp. 109–120 (2007).

[3] The Scala Programming Language, http://www.scala-lang.org/.