平成24年度 修士論文

静的型付け言語における汎用的なユーザ定義演算子を含む式の構文解析手法

東京工業大学大学院 情報理工学研究科
数理・計算科学専攻

学籍番号 11M37011

市川 和央

指導教員

渡辺 治 教授
千葉 滋 教授

平成25年1月31日

概要

言語内ドメイン専用言語はドメイン専用言語 (DSL) の *composability* が高いため、言語指向プログラミング (LOP) のようなスタイルの開発に有用である。しかし、言語内 DSL は文法がそれを実装する汎用言語 (ホスト言語と呼ばれる) に制限されてしまう。そのため、ホスト言語を拡張可能にすることで文法の制限を緩和する様々な手法が提案されているが、それらの手法の多くは composability を壊してしまう。ユーザ定義演算子は composability を壊さずにホスト言語の構文を拡張することができるが、既存の手法では定義できる演算子の構文に制限があり、また字句規則の拡張はすることができない。

そこで、我々は既存のユーザ定義演算子よりも強力な構文拡張が可能で、ユーザ定義リテラルも実現することのできるようなユーザ定義演算子 *protean operators* とその構文解析手法を提案する。protean operators はオペレータ部分に当たる name part とオペランド部分に当たる hole の列で表現することができるような演算子のクラスで、name part と hole の並び方には制限がなく、name part には任意のトークンを利用することができる。protean operators の name part はトークンの切り方を変えるため、ユーザ定義リテラルを実現することも可能である。我々の提案する構文解析手法は、期待される型の情報を利用した再帰下降構文解析である。本手法は期待される型の情報を利用して有効な演算子を制限することで枝刈りを行いながら解析を進めるといったものである。このような枝刈りに加えて左再帰を許す packrat parsing [31] を利用することで、protean operators を含む式を効率的に解析することを可能とする。

本論文では、protean operators の強力さ・有用性を示すために、その表現力が解析表現文法 (PEG) [8] と同等であることを示した。また、我々の提案した構文解析手法の計算量がほとんどの場合に線形オーダーとなっており、従来の構文解析手法で protean operators を実現しようとした場合と比較して十分に小さいことを示す。我々は、本手法を利用して Java 言語のサブセットに protean operators を追加した言語 ProteaJ を作成した。

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、本研究で考察を行う際に相談に乗って下さった武山先輩に感謝致します。最後に、共に研究活動を行い、多くの助言を頂いた研究室の皆様に感謝致します。

目 次

図 目 次

表 目 次

第1章　はじめに

ドメイン専用言語 (DSL) 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言語内 DSL [10] と呼ぶ。言語内 DSL の利点はそのホスト言語および他の DSL と組み合わせて利用することができる点である。ある DSL の内部で別の DSL を利用することも可能である。このような、ある DSL を他の DSL と組み合わせて利用することができるという性質を DSL の *composability* [23] と呼ぶ。この性質により、多数の小さな DSL を組み合わせて開発を行う 言語指向プログラミング *(LOP)* [30] を容易に実現することができる。しかし、言語内 DSL はホスト言語によって文法が制限されてしまうという大きな欠点を持っている。そのため、DSL の composability を損なわずにホスト言語の文法を拡張する手法が求められている。

ホスト言語を拡張可能にすることで文法の制限を緩和する手法としては、字句マクロ・構文マクロを代表とする多くの手法が存在するが、それらの手法の多くは DSL の composability を壊してしまう。ユーザ定義演算子は composability を壊さずにホスト言語の構文を拡張することができるが、既存の手法では定義できる演算子の構文に制限があり、また字句規則の拡張はすることができない。

そこで我々は、既存のユーザ定義演算子よりも強力な構文拡張が可能で、ユーザ定義リテラルも実現することのできるようなユーザ定義演算子 *protean operators* とその構文解析手法を提案する。protean operators はオペレータ部分に当たる name part とオペランド部分に当たる hole の列で表現することができるような演算子のクラスで、name part と hole の並び方には制限がなく、name part には任意のトークンを利用することができる。protean operators はそれぞれ固有の字句規則を持っており、それが利用されている場所ではその字句規則に従うため、protean operators でユーザ定義リテラルを実現することも可能である。我々の提案する期待される型を利用した再帰下降構文解析は、期待される型情報を利用した型のあわない演算子を利用する解析パスの枝刈りと左再帰を許す packrat parsing [31] を組み合わせることで、このような演算子を含む式の高速な解析を可能とする。

本論文の構成は以下の通りである。第 2 章では、言語内ドメイン専用言語のための既存の言語拡張手法について述べ、それらの問題点を指摘する。第 3 章では protean operators とその解析方法について説明する。第 4 章では、第 3 章で提案した手法に基づいて protean operators を実装した言語 ProteaJ を紹介し、その実装手法と応用例について述べる。第 5 章では protean operators の表現力及び提案する解析手法の計算量に関する議論を行う。最後に、第 6 章では本論文をまとめ、今後の課題を示す。

第2章　言語内ドメイン専用言語のための言語拡張

言語内ドメイン専用言語はドメイン専用言語 (DSL) の *composability* が高いため、言語指向プログラミング (LOP) のようなスタイルの開発に有用である。しかし、言語内 DSL は文法がそれを実装する汎用言語 (ホスト言語と呼ばれる) に制限されてしまう。そのため、ホスト言語を拡張可能にすることで文法の制限を緩和する様々な手法が提案されているが、それらの手法の多くは composability を壊してしまう。ユーザ定義演算子は composability を壊さずにホスト言語の構文を拡張することができるが、既存の手法では定義できる演算子の構文に制限があり、また字句規則の拡張はすることができない。本章では、言語内 DSL のための言語拡張の既存手法を紹介し、それらの問題点を示す。

2.1　言語内ドメイン専用言語

ドメイン専用言語 (DSL) は特定の問題解決に特化したプログラミング言語を指す。例えば、SQL はデータベースの操作に特化した DSL であり、make はプログラムのビルド作業の記述に特化した DSL である。DSL は C 言語 や Java といった汎用言語とは異なり限定された用途にしか利用することができないが、その分簡潔で読みやすいプログラムを書くことができる。

汎用言語の範囲内でライブラリとして DSL を実現したものを 言語内 DSL [10] (または 組み込み DSL [14]) と呼ぶ。言語内 DSL はそれを実装する汎用言語 (ホスト言語 と呼ぶ) の表現力を上手く利用して DSL 風の文法を実現する。言語内 DSL としてよく知られているのが Ruby の Rake [32] で、これは Ruby の中で make 風の文法を実現したものである。

言語内 DSL の利点はそのホスト言語および他の DSL と組み合わせて利用することができる点である。言語内 DSL はホスト言語のライブラリに過ぎないため、ホスト言語のプログラムの一部として DSL のプログラムを記述することができる。そのため、DSL 内部でホスト言語の機能を利用することができ、ある DSL の内部で別の DSL を利用することも可能である。このような、ある DSL を他の DSL と組み合わせて利用すること

ができるという性質を DSL の *composability* [23] と呼ぶ。この性質により、多数の小さな DSL を組み合わせて開発を行う 言語指向プログラミング *(LOP)* [30] を容易に実現することができる。

しかし、言語内 DSL はホスト言語によって文法が制限されてしまうという大きな欠点を持っている。言語内 DSL はホスト言語の範囲内で実装されているため、ホスト言語の文法で表現することができないような DSL は実現することができない。そのため、Scala [19] などの言語は多数のシンタックスシュガーを導入することでホスト言語の表現力を向上させ、様々な DSL を実現できるようにしている。しかし、このような手法では言語設計時に想定していないような文法を持つ DSL は実現することができない。

2.2　言語拡張

言語拡張 はホスト言語を拡張可能にすることで、文法の制限を緩和する試みである。ソースコードにコンパイラに対する命令を記述しておくことで、コンパイル時にコンパイラの動きを変化させ、本来は受理できないような文法を受理できるようにする。このような言語拡張は今まで様々な手法で実現されてきた。

しかし、既存の言語拡張手法の多くは文法の拡張性を得るのと引き換えに DSL の composability を損なってしまう。具体的には、以下の 2 つのいずれかの問題が発生してしまう。

- 2 つの DSL を同時に利用した場合に、一方の DSL がもう一方の DSL を上書きしてしまう
- ある DSL の内部で他の DSL を利用することができない

本節では、既存の言語拡張の手法を簡単に解説し、それらが DSL の composability を損なっていることを示す。

2.2.1　字句マクロ

字句マクロはソースコード上の特定のパターンにマッチした部分を別の文字列に置き換えるテキスト置換処理である。これは通常、字句解析と構文解析の間のプリプロセスと呼ばれるフェーズに処理される。字句マクロは C 言語および C++言語で採用されており、#define ディレクティブにより定義することができる。図 2.1 に示した SQL の select 文を C++言語上で字句マクロを用いて再現すると、図 2.2 のようになる。余分な括弧が

```
select name from students where score <= 500
```

図 2.1: SQL の select 文の例

```
#define select(s) (new Select(#s))
#define from(t) ->from(t)
#define where(c) ->where(#c);

ResultSet* rs =
  select (name) from (students) where (score <= 500)
```

図 2.2: 字句マクロによる SQL の select 文の再現例

付いてしまっているが、C++言語では本来解析することができない SQL 風の文法を実現していることが分かる。

字句マクロは単なる字句の置換であるため、ホスト言語や他の DSL のセマンティクスを上書きしてしまう危険性がある。例えば、図 2.2 では select や from をマクロによって置換してしまっているため、このプログラム内ではこれらの名前を関数名として利用することができない。

2.2.2　構文マクロ

構文マクロはプログラムを構文解析して得られた抽象構文木 (AST) を別の AST に置き換えるシステムであり、Lisp 言語族が実装していることで知られている。構文マクロは字句マクロと異なりそれぞれのトークンの構文上の意味を利用することができるため、字句マクロよりは名前の衝突に強く安全であると言える。実際、Scheme などでは構文マクロが *hygienic* [18] であることが保証されている。これは、構文マクロの定義で使われている変数名がそのマクロを利用している場所の変数名と衝突していた場合でも、一方が一方を上書きすることがないことを保証する。

構文マクロは字句マクロよりは composability が高いが、同じ構文を持つ複数の DSL およびホスト言語を同時に利用した場合、一方がもう一方を上書きしてしまう。例えば、次の図 2.3 は図 2.1 の SQL の select 文を Lisp 方言の一つである Scheme で再現したものである。 ここでは select と <= の 2 つのマクロを定義しているが、このうち <= マクロは通常の比較演算子の <= と同じ構文を持っている。そのため、このプログラムでは <= 比較演算子は <= マクロに上書きされてしまっているため、利用することができない。

また、構文マクロは構文解析後に展開されるため、構文解析が通らないような文法は実現することができない。そのため、構文マクロの表現力

```
(define-syntax select
  (syntax-rules (from where)
    ((select col from table where cond)
     (select-from-where 'col table cond))))

(define-syntax <=
  (syntax-rules ()
    ((<= col value)
     (less-equal 'col value))))

(define result
  (select name from students where (<= score 500)))
```

図 2.3: 構文マクロによる SQL の select 文の再現例

```
(set-dispatch-macro-character #\# #\$ 'read-sql-select)

#$ select name from students where score <= 500
```

図 2.4: リードマクロによる SQL の select 文の再現例

はホスト言語の構文の柔軟さに依存する。Lisp では様々な DSL を構文マクロにより表現することができるが、これは Lisp の構文が S 式と呼ばれる非常に単純で柔軟な構造であるためである。従って、Lisp の構文マクロでも S 式で表現できない文法構造を持つような DSL は実現できない。Lisp 以外の言語でも Dylan [3]、MetaML [21]、Template Haskell [22]、Nemerle [24] などの言語は構文マクロを持っているが、いずれも Lisp と同様にホスト言語の構文自体は拡張することができないという問題を持っている。

2.2.3　リードマクロ

リードマクロは特定の字句を読んだ場合に字句・構文解析器を切り替えるシステムである。Common Lisp のリードマクロはプログラマがマクロ文字と呼ばれる特殊な文字を定義し、コンパイラがその文字を読み込むと字句・構文解析器がそのプログラマの指定したものに切り替わる。SQL の select 文を構文解析する関数 read-sql-select を用意すると、図 2.4 のようにして図 2.1 の select 文を再現できる。 この例では #$ がマクロ文字となっており、#$ 以降が read-sql-select により解析される。

リードマクロは任意の DSL の構文を実現することが可能だが、あるリードマクロを使ったプログラムの中で別のリードマクロを使うことができな

い可能性がある。リードマクロで導入される構文解析関数はユーザが書いたものであるため、その内部で他のリードマクロのマクロ文字がマクロ文字として処理される保証はない。そのため、DSL の composability を保証することができない。

リードマクロと類似したシステムは様々な言語に導入されている。Template Haskell [22] の準クォートや、Converge [28] の DSL block はリードマクロと同様にソースコードの一部をユーザ定義の字句・構文解析器によって解析する。これらも Common Lisp のリードマクロと同様、DSL の composability を損なってしまう。

2.3　ユーザ定義演算子

一部のプログラミング言語はユーザが新しい演算子を定義し言語に追加することができる ユーザ定義演算子 の仕組みを持っている。これは C++ や Ruby、Python などの持つ演算子オーバーロードを発展させたもので、プログラムを簡潔にし、可読性を向上させる。

ユーザ定義演算子は言語拡張の一種と見なすこともできる。ユーザは演算子という形でコンパイラに字句・構文規則の追加を通知し、ホスト言語が通常解析できないような形の式を含む式を解析可能にしている。

ユーザ定義演算子の優れた点は DSL の composability を保つところである。ユーザ定義演算子は異なるライブラリで定義された演算子でも同時に組み合わせて使うことができる。演算子のオペランド部分で別のライブラリで定義された演算子を利用することも可能である。演算子は型によってオーバーロードされるため、同一の構文を持つ演算子が存在してもお互いを上書きしてしまうことはない。

しかし、既存のユーザ定義演算子はその表現力に制限があった。ユーザ定義演算子に利用できる構文は中置二項演算子などの特定のパターンのものに限られる。また、既存のユーザ定義演算子では正規表現のような複雑なユーザ定義リテラルを作ることはできなかった。

本節では、既存のユーザ定義演算子を紹介し、それらには表現力に不足があることを示す。

2.3.1　中置二項演算子

Scala [19] や Haskell [15]、OCaml [13] などの言語ではプログラマが新しい中置二項演算子を定義することができる。中置二項演算子は $\_\ op\ \_$ のような形式の演算子であり、複数組み合わせることで $\_\ op_1\ \_\ op_2\ \_\ \ldots\ \_\ op_n\ \_$ のような構文を実現することができる。ここで、$\_$ は演算子のオ

```
str should startWith (‘‘Hello’’) // (A)
str should endWith regex (‘‘wo.ld’’) // (B)
str should not include regex (‘‘Go*dbye’’) // (C)
```

図 2.5: ScalaTest による単体テストの記述例

ペランドを表現しており、*op* および op_i は演算子のオペレータを表現している。以降、演算子のオペランド部分 _ を *hole* と呼び、演算子のオペレータ部分 op_i を *name part* と呼ぶ。

Scala では中置二項演算子のユーザ定義を活用した言語内 DSL が実際に作られ、利用されている。ScalaTest [2] はそのようなライブラリの一つで、単体テストを自然言語に近い形で記述することができる。図 2.5 は ScalaTest による単体テストの記述例である。 図 2.5 の (A) では should が中置二項演算子となっており、str と startWith(“Hello”) がその引数となっている。(B) では should と regex が中置二項演算子であり、endWith は定数である。 (C) では should と include が中置二項演算子となっている。

中置二項演算子の組み合わせで DSL 風の文法を実現することは可能ではあるが、それには非常に多くの労力がかかる上、ユーザにとって分かりづらい制限がついてしまうことがある。図 2.5 の例において、(A) の startWith は関数であり、(B) の endWith は定数であり、(C) の include は中置二項演算子である。ScalaTest では図 2.5 のような記述を許すために、同じ名前の関数と定数と中置二項演算子を用意しているのである。また、(A)、(B)、(C) のいずれも最後の引数部分に括弧が付いているが、この括弧は (A) および (C) では外すことができず、(B) では外すことができる。このような文法の制限はあまり直感的であるとは言えない。中置二項演算子の組み合わせによる表現は DSL の本来のセマンティクスとは異なる形で構文を組み立てているため、実現に余分な労力を要したり、ユーザが直感的に分かりづらい文法の制限がついてしまうことがある。

2.3.2　mixfix operators

Coq [26] や Agda [1] などの言語では *mixfix operators* [6] と呼ばれる演算子のユーザ定義を許している。mixfix operators は中置二項演算子よりも広範な構文を表現することが可能な演算子のクラスである。mixfix operators は複数の name part を持つことができ、infix、prefix、postfix、および outfix の形式を取ることができる。ここで、infix、prefix、postfix、outfix はそれぞれ次に示すような形式を指す。

infix: _ op_1 _ op_2 _ ⋯ _ op_n _

prefix: op_1 _ op_2 _ ⋯ _ op_n _

```
_ ? _ : _ :: Boolean => Int => Int => Int
if _ then _ else _ :: Boolean => Int => Int => Int
_ [ _ ] :: Array[Int] => Int => Int
| _ | :: Vector[Int] => Int
```

図 2.6: mixfix operators の例

postfix: _ op_1 _ op_2 _ ... _ op_n
outfix: op_1 _ op_2 _ ... _ op_n

name part には任意の識別子を利用することができるので、開き括弧や閉じ括弧を識別子とする言語では配列操作なども演算子で記述できる。図 2.6 は mixfix operators の例である。:: に続く部分は型を表現している。例えば、３つ目の例の _ [_] は Int 型の配列と Int 型の値を受け取り Int 型の値を返す演算子である。

Pure [12] は中置二項演算子に加えて、prefix、postfix、outfix の単項演算子をユーザが定義することを認めている。中置二項演算子の組み合わせにより、_ op_1 _ op_2 _ ... _ op_n _ のような構文を表現することができるため、これと prefix、postfix、outfix の単項演算子を組み合わせることで、この言語も mixfix operators と同等の表現力を持つことができる。

mixfix operators は中置二項演算子よりも強力であり、例えば図 2.5 の例は mixfix operators を利用すると以下のように括弧を外して記述することができる。

```
str should startWith ‘‘Hello’’
str should endWith regex ‘‘wo.ld’’
str should not include regex ‘‘Go*dbye’’
```

また、startWith や endWith、not などを前置演算子として書くことができるため、同じ名前の関数や定数、演算子を複数用意するような余分な手間を掛ける必要がない。これは DSL の本来のセマンティクスと近い形で構文を組み立てられるためである。

mixfix operators は単体テストなどを自然言語に近い形で記述するのには向いているが、一方でシェルスクリプトや正規表現といった簡潔さを重視した形式記述は苦手である。mixfix operators は hole と hole の間に必ず name part が入るため、どうしても表現に冗長さが入ってしまう。例えば、図 2.7 のようなコードを mixfix operators で実現することを考えてみよう。
本来のシェルスクリプトでは正規表現はダブルクォートでくくられているが、今回は正規表現もユーザ定義演算子として記述することを目指すものと

```
grep -i hel*o ~/Documents/Main.java
```

図 2.7: シェルスクリプトの例

```
grep _ _ _ :: GrepOptions => Regex => File => GrepResult
_ _ :: GrepOption => GrepOptions => GrepOptions
"" :: GrepOptions
-i :: GrepOption
_ _ :: Regex => Regex => Regex
_ * :: Regex => Regex
_ :: Letter => Regex
_ / _ :: File => Identifier => File
_ . _ :: File => Identifier => File
~ :: File
```

図 2.8: 図 2.7 の記述に必要な演算子

する。grep はコマンド名、-i はオプション、hel*o と /Documents/Main.java は引数なので、このようなコードを書くためには例えば図 2.8 に挙げるような演算子が必要となる。ここで、"" は空文字列を表現しているものとし、また任意の英数字 1 文字を認識して Letter 型を返す演算子と任意の識別子を認識して Identifier 型を返す演算子は既に定義されているものとする。

grep _ _ _ や _ _ のような演算子は mixfix operators の範囲には含まれない。mixfix operators では連続する hole の間に name part が必要なので、grep _ _ _ は grep _ , _ , _ のように、_ _ は _ - _ のようにしなければならない。そのため、mixfix operators で図 2.7 のコードを表現すると

```
grep -i, h-e-l*-o, ~/Documents/Main.java
```

のようになってしまう。このように、mixfix operators は表現に冗長さを付加してしまうことがある。

また、mixfix operators では演算子を追加しても字句規則は変化しないため、複雑なユーザ定義リテラルを実現することはできない。具体的には以下のような問題が発生してしまう。

- *- や ~/ が一つのトークンとみなされてしまう
- 空白文字をリテラルの切れ目にできない
- 空文字列 "" は演算子に使えないため、文法を変える必要がある
- -i は 2 トークンとみなされるため、別の文字列に変える必要がある

１つ目の問題を解消する容易な方法はスペースを開けることか括弧を付けることである。しかし、それはリテラルを式として表現することなので、プログラムが非常に読みづらくなってしまう。この問題は正規表現のような識別子と衝突するリテラルを作ろうとした場合に特に深刻で、例えば正規表現 hel*o は h e (l*) o と解釈するべきだが、(hel) * o と解釈されてしまう。２つ目の問題は、

```
/home/username/log1.txt /home/username/log2.txt
```

と書いたときに、これを一つの連続したパスとして解析しまうといった問題である。これは

```
(/home/username/log1.txt) (/home/username/log2.txt)
```

のようにそれぞれのパスに括弧を付けることで解消できるが、それはユーザにとって非常に分かりづらい制約となる。３つ目、４つ目の問題は name part に使えるものが限られるというもので、文法などを変更することで対処できるが、こちらも可読性や簡潔さを犠牲にしてしまうことがある。これらを踏まえて mixfix operators で図 2.7 のコードを再現すると、以下のような非常に可読性が低く、とても DSL とは言えないようなコードとなってしまう。

```
grep _i, h-e-(l*)-o, (~)/Documents/Main.java
```

2.3.3　より進んだアプローチ

Isabelle [20] や OBJ3 [11] は mixfix operators に加えて *empty syntax* の演算子のユーザ定義を許している。empty syntax は hole のみからなる _ _ のような構文である。empty syntax の演算子 (以下、*empty operators* と呼ぶ) を利用することで、grep _ _ _ のような mixfix operators のみでは実現できなかった演算子を実現することが可能となる。

empty operators は非常に強力だが、その強力さゆえに構文解析が難しい。empty operators は演算子を識別する部分がないため、導入すると文法に多くの曖昧性と非決定性を生じさせてしまう。そのため、Isabelle では chart parsing [17] と呼ばれる構文解析法によって曖昧性を保持したまま解析を行い、生成された全ての構文木に対して型チェックを行うことで型レベルで曖昧性を排除している。この手法は確かに empty operators を正しく解析することができるが、曖昧さの大きい文法に対しては非効率的で [23]、実用性に欠ける。多くの DSL ライブラリを同時に利用した場合、それぞれの DSL の文法が曖昧性を含まなくても、構文解析器が扱うそれ

らを併合した文法には曖昧性が生じてしまう [16] ため、このような手法は composability が低い。

type-oriented island parsing [23] は構文木を作りながら型チェックを同時に行うことで Isabelle の手法よりも高速に解析を行うことを可能にしている。この手法は island parsing [25] と呼ばれる chart parsing の一種をもとにした解析方法で、強力な構文拡張を DSL の composability を保ちつつ実現できる。しかし、type-oriented island parsing は字句規則の拡張は許すことができない。

Metaborg [5] で利用している構文解析方法 [4] は構文規則に加えて字句規則も拡張することを許している。これは scannerless generalized LR parsing [29] に基づく構文解析手法で、scannerless にすることにより字句規則を構文規則と同様に扱うことを可能としている。しかし、この手法は Isabelle と同じく曖昧性を保持したまま解析を行い、生成された全ての構文木に対して型チェックを行うため、曖昧性や非決定性が高くなるような場合には非効率的で、composablility が低いと言える。

第3章　期待される型を利用した再帰下降構文解析

我々は、mixfix operators 以上に強力な構文拡張が可能で、ユーザ定義リテラルも実現することのできるようなユーザ定義演算子を解析する構文解析手法を提案する。本手法は期待される型の情報を利用して字句・構文規則を切り替えながら構文解析を進める。言い換えると、型を非終端記号とした再帰下降構文解析である。

本章では、まず本手法が取り扱うユーザ定義演算子について説明し、次いで本手法の中心的な部分のアルゴリズムを紹介する。その後、その他の重要な特徴についての詳細な解説を行う。

3.1　protean operators

本手法では mixfix operators よりも広範な構文を持つ演算子を取り扱うことができる。我々は、本手法で取り扱うことのできる演算子のクラスを *protean operators* [1] と名付けた。

3.1.1　protean operators の特徴

protean operators の構文は name part と hole の列により表現する。mixfix operators との違いは name part と hole の並び方に制限がない点である。protean operators は infix, prefix, postfix, outfix のいずれでもないような順序で演算子が並ぶことを許す。例えば、grep _ _ _ や _ _ のような演算子も protean operators として定義することができる。

protean operators では、name part に任意のトークンを利用することができる。例えば、-i のような通常の字句解析器では 1 トークンとして扱われないようなトークンでも name part に利用することができる。空文字列 "" も同様に name part に利用することができる。

protean operators の非常に重要な特徴は、protean operators はそれぞれ個別に字句規則を持つという点である。これは protean operators の

[1] 変幻自在な演算子 の意

```
grep _ _ _ :: GrepOptions => Regex => File => GrepResult
_ _ :: GrepOption => GrepOptions => GrepOptions
"" :: GrepOptions
-i :: GrepOption
_ _ :: Regex => Regex => Regex
_ * :: Regex => Regex
_ :: Letter => Regex
_ / _ :: File => Identifier => File
_ . _ :: File => Identifier => File
~ :: File
```

図 3.1: grep 式を実現する protean operators

name part が字句規則となると言い換えることもできる。例えば図 3.1 のような protean operators が定義されていた場合、次のコード (ホスト言語は Java 風の言語とする)

```
GrepResult r = grep -i hel*o ~/Documents/Main.java;
```

は正しく解析され、-i は 1 つのトークンと、~/ は 2 つのトークンとして解釈される。-i は GrepOption 型を返す protean operator -i の name part なので、-i は 1 つのトークンとして字句解析され、~ と / は それぞれ File 型を返す~ と File 型を返す _ / _ の name part なので、それぞれ別々のトークンとしてみなされる。このように、各 protean operators の式では字句規則がそれらの name part のトークンを含むものに切り替わる。

3.1.2　演算子のオーバーロードと順序

protean operators は他のユーザ定義演算子と同様に型情報を持っており、引数の型によってオーバーロードされる。また、 protean operators は引数の型だけでなく、返り値の型によってもオーバーロードされる。これは構文規則が型によってオーバーロードされるともみなすことができる。例えば、図 3.1 の演算子が存在するとき、

```
Int i = 1;
Int j = -i;
GrepOption opt = -i;
```

とすると、j は Int 型の値 -1 となるが、opt は GrepOption 型のオブジェクトが代入される。このように、protean operators は引数と返り値の型によるオーバーロードを用いて文法の曖昧性を取り除き、DSL の composability を保証している。これは Isabelle [20] や Metaborg [5] などと同様の手法である。

```
[lex] _ _ :: Regex => Regex => Regex
[lex] _ * :: Regex => Regex
[lex] _ :: Letter => Regex
[lex] _ / _ :: File => Identifier => File
[lex] _ . _ :: File => Identifier => File
[lex] ~ :: File
```

図 3.2: lexical operators

protean operators では同じ型を返す演算子の間に順序関係を付けることを要求する。これは文法規則に順序関係を付けることで非決定性を小さく抑え、高速な解析を可能にするためである。多くの場合、異なる DSL は異なる型を用いる [23] ため、同じ型を返す演算子の間に順序関係を付けても DSL の composability は損なわれない。

演算子間の順序関係はどちらの演算子を優先的に利用するかを示すものである。例えば、if _ then _ と if _ then _ else _ の 2 つの演算子があった場合、前者は後者を完全に包含しているため、前者を優先にしてしまうと後者は絶対に利用されなくなってしまう。一般的には、より特殊な演算子を優先するように順序関係を指定する。本論文中では、同一コード上でより上に書かれた演算子を優先とする。

3.1.3　lexical operators

protean operators では、演算子ごとに異なる字句規則を持つことができるため、当然空白文字の扱いも変えることができる。しかし、大抵の場合において空白文字は構文上の意味を持たないため、構文レベルの拡張を行う際には空白文字を無視して記述できるのが好ましい。そこで我々は、通常の protean operators では空白文字を特別扱いするものとし、字句レベルの拡張を行うための演算子 *lexical operators* を用意するものとした。

lexical operators は空白文字を特別扱いせず、通常の文字と同じように扱う。そのため、正規表現のようなユーザ定義リテラルを表現するのに適している。lexical operators は字句レベルの規則を表現するので、lexical operators のオペランドに入るのも lexical operators の式でなければならない。以降、lexical operators は通常の protean operators と区別するために、図 3.2 のように [lex] を付与するものとする。 図 3.2 のような演算子を定義すると、

```
Regex r = hel*o;
File f = ~/Documents/Main.java
```

は正しく解釈されるが、

```
[lex] a :: Letter
[lex] b :: Letter
...
[lex] z :: Letter
[lex] A :: Letter
...
[lex] Z :: Letter
[lex] 0 :: Letter
...
[lex] 9 :: Letter
[lex] _ _ :: Letter => Identifier => Identifier
[lex] _ :: Letter => Identifier
```

図 3.3: Letter 及び Identifier を認識する lexical operators

```
Regex r = he l*o;
File f = ~/Documents /Main.java
```

のように空白文字が入っていると構文解析に失敗する。そのため、前章で示したようなリテラルの切れ目を認識できない問題は発生しない。

任意の英数字 1 文字を認識して Letter 型を返す演算子や任意の識別子を認識して Identifier 型を返す演算子も、図 3.3 のようにすることで lexical operators として定義することができる。 このように、lexical operators を利用することで、様々な構文要素を表現することが可能となる。

通常の protean operators も 0 個の空白文字を認識する lexical operators を定義することで lexical operators で表現することが可能となる。そのため、通常の protean operators は lexical operators の特別な場合と見なすことができる。

3.1.4　まとめ

以上をまとめると、本手法で扱うことのできる protean operators は、

- name part と hole の列からなる
- name part と hole の並び方には制限がない
- name part には任意のトークンを利用できる
- それぞれ個別の字句規則を持つ
- 引数と返り値の型でオーバーロードされる
- 同じ型を返す演算子の間に順序関係が必要

- 空白文字の扱いを変えた 2 種類の演算子が存在

3.2　期待される型を利用した再帰下降構文解析

本手法は再帰下降構文解析に基づく構文解析手法で、期待される型の情報を利用して有効な演算子を制限することで枝刈りを行いながら解析を進めるといったものである。このような枝刈りに加えて、左再帰を許す packrat parsing [31] を利用することで、protean operators を含む式を効率的に解析することを可能とする。

3.2.1　処理の流れ

通常のコンパイラは次のような順序で処理を進める。

1. 字句解析
2. 構文解析
3. 意味解析 (型チェックを含む)
4. コード生成

字句解析で入力文字列をトークン列に変換し、構文解析でトークン列から抽象構文木 (AST) を生成する。生成された AST に対して意味解析を行って AST に型情報などを加え、コード生成により目的コード (バイトコードなど) を生成する。

我々の手法は、これらのうち字句解析・構文解析・型チェックを連携させる。すなわち、型チェックの結果を字句解析や構文解析にフィードバックしながら解析を進めていく。構文解析の処理の流れは以下の通りである。ただし、4 において、演算子が通常の protean operators であれば空白文字を特別扱いして無視するものとし、lexical operators であればその hole 部分では演算子の列挙の際に lexical operators のみを利用するものとする。

1. 型がわからない部分はホスト言語の規則で字句・構文解析を行う。
2. 型がわかる部分まで解析したら型解析を行い、次の式で期待される型を特定する。
3. 期待される型を返す演算子を列挙し、演算子の順序でソートする。

4. 演算子を順番に使って解析を試み、成功した時点で終了する。
   各演算子はそのパターンに沿って以下のように解析する。
   途中で失敗したらバックトラック[2]して次の演算子を試す。

   (a) hole は対応する引数の型を期待される型として 3 へ。
   Fail が帰ってきたら失敗。

   (b) name part は対応するトークンのみを受理する字句解析を行う。字句解析に失敗したら失敗。

5. 解析が成功した場合は AST を返し、失敗した場合は Fail を返す。

ここに示したように、我々の手法では期待される型を利用して演算子を限定し、その後それぞれの演算子の規則で解析を行う。このことにより、引数と返り値の型による演算子のオーバーロードを実現し、また様々な字句・構文規則を持った演算子を実現することが可能となる。

また、型のあわないような演算子ははじめから利用されないので、その分不要な解析パスが枝刈りされるため効率的に解析を行うことができる。本手法ではバックトラックを繰り返し行う必要があるが、packrat parsing [7] の技術を組み合わせることで、その分のコストも無視することが可能な程度に小さくすることができる。

ただし、本手法では演算子を含む式の解析に期待される型を利用するので、期待される型のわからないような場所では演算子を利用することができない。例えば、代入文の左辺は型が書いていない限り期待される型がわからないため、演算子を利用することができない。そのため本手法はできるだけ期待される型の分かる部分が多いプログラミング言語に導入するべきである。

3.2.2　具体例

例えば、ホスト言語を Java として次のようなコードを解析することを考える。

```
GrepResult r = grep -i hel*o ~/Documents/Main.java;
```

このとき、GrepResult r = まで読むことで、この文がローカル変数を宣言して初期値を代入する文であることを知ることができる。

次に、我々の構文解析器は型解析器と連携して次の式で期待される型を特定する。この例では、 GrepResult r = の GrepResult が型を表現する識別子であることが分かっているので、型解析を行うことで次の式で期待される型が shell.grep.GrepResult 型であることがわかる。

[2]元の位置に戻ること

```
grep _ _ _ :: GrepOptions => Regex => File => GrepResult
_ _ :: GrepOption => GrepOptions => GrepOptions
"" :: GrepOptions
-i :: GrepOption
[lex] _ _ :: Regex => Regex => Regex
[lex] _ * :: Regex => Regex
[lex] _ :: Letter => Regex
[lex] _ / _ :: File => Identifier => File
[lex] _ . _ :: File => Identifier => File
[lex] ~ :: File
```

図 3.4: shell.grep.GrepOperators

このコードの書かれたソースファイルから参照可能な演算子を図 3.4 の演算子とすると、期待される型である GrepResult 型の値を返すような演算子は grep _ _ _ のみなので、この演算子を利用して解析を行う。

grep _ _ _ 演算子のパターンは name part, hole, hole, hole となっているので、まずは 4b のように name part grep の部分を解析する。つまり、grep だけを受理するような字句解析を行う。GrepResult r = の次の部分は grep -i ... なので、字句解析は成功し、GrepResult r = grep まで読み進める。

次の hole 部分は GrepOptions 型が入るので、GrepOptions 型を期待する型として 3 に戻る。GrepOptions 型を返す演算子は _ _ と "" の 2 つがあるが、前者の方が優先なのでまずは前者を利用して解析を行う。

_ _ の 1 つ目の hole は GrepOption 型を期待するので、再び 3 に戻り、-i 演算子で解析を行う。-i 演算子はただひとつの name part -i だけからなるので、-i を受理するような字句解析を行う。この解析は成功し、... grep -i まで読み進む。

_ _ の 2 つ目の hole は GrepOptions 型を期待するので、_ _ か "" で解析を行う。前者が優先なのでまずは前者を利用して解析を行う。_ _ の 1 つ目の hole は GrepOption 型を期待するため、-i 演算子で解析を試みるが、次の入力は -i でないので失敗する。_ _ での解析が失敗したため "" で解析を行うと、これは成功するので、これで grep _ _ _ の 1 つ目の hole 部分の解析が終了する。

以降、同様にして解析を進めていく。正規表現の部分及びファイルパスの部分は lexical operators で定義されているため、ここまでとは異なり字句解析の時に空白文字を無視することができない。しかし、それ以外は同様にして解析を進めることができる。

正規表現の protean operators の規則には左再帰が含まれるが、これは、左再帰を許す packrat parsing [31] のアルゴリズムにより検出して正しく

解析することができる。

3.2.3　左再帰を許す packrat parsing

我々の手法のように、AST の根から葉の方向に進みながら AST を成長させつつ解析を行っていく構文解析手法をトップダウン型の構文解析と呼ぶ。逆に AST の葉から根に向かって小さな部分木をつなぎ合わせるようにしながら解析を行っていく構文解析手法をボトムアップ型の構文解析と呼ぶ。我々の手法は再帰的に構文規則を適用しながらトップダウンで構文解析を進めるため、再帰降下構文解析と呼ばれるトップダウン型の構文解析の一種である。

本手法のアルゴリズムは型のあわない演算子を利用した解析パスを枝刈りすることにより構文解析の高速化を行っているが、バックトラックを繰り返し行うため単純に実装すると指数時間の解析時間がかかってしまう。そこで、packrat parsing [7] を利用することでバックトラックのオーバーヘッドを低減させ、高速な解析を可能とする。

packrat parsing [7] は再帰下降構文解析にメモ化を組み合わせたもので、通常再帰下降構文解析では指数時間かかるような構文解析を線形時間で行うことができる。packrat parsing はメモテーブルと呼ばれる構造体に部分木の位置と構造を記憶させることで、同じ解析を 2 度行うことがないようにしている。そのため packrat parsing はバックトラックに強く、常に線形時間で動作する。

しかし、 packrat parsing は通常の再帰下降構文解析と同様、左再帰を扱うことができない。左再帰とは、

```
_ / _ :: File => Identifier => File
```

のように、最も左に位置する部分が再帰的に自身の規則を参照しうる場合を言う。このような場合、packrat parsing を含む再帰下降型の構文解析は同じルールを参照し続ける無限再帰に陥ってしまう。

そのため、我々は左再帰を許す packrat parsing [31] を利用することで、左再帰を扱うことを可能とした。これは packrat parsing のメモ化の仕方に手を加えたもので、部分木の解析前に `Fail` を書きこんでおくことで、左再帰を検出する。左再帰を検出した場合は、左再帰する規則以外で一度解析を行い、その解析結果を種として同じ部分を繰り返し左再帰を含む規則で解析する。左再帰の部分を再び解析すると、メモテーブルから種となる部分木が発見されるので、無限再帰には陥らずに種を少し成長させた部分木が得られる。その部分木を種として再び解析を行い、それ以上部分木が成長しなくなるまでこれを繰り返す。これにより、左再帰を含む規則を正しく解析することができる。

3.3　演算子の優先順位と結合性

ほとんどのプログラミング言語において、演算子には優先順位が存在する。これは前述した同じ型を返す演算子間の順序関係とは異なるもので、演算子の結合の強さを表現したものである。例えば、1 + 2 * 3 は 1 + (2 * 3) と解釈されるが、これは _ + _ よりも _ * _ のほうが演算子優先順位が高いためである。

以降では、同じ型を返す演算子間の順序関係と演算子の結合の優先順位を区別するために、前者を解析順序 (順位)、後者を結合順序 (順位) と呼ぶこととする。ここで、順序は 2 つの演算子の関係を示すときに使い、順位は比較対象となる演算子全体での位置を示すときに用いるものとする。A より優先順位が高いというのは、A よりも優先することを示している。

演算子の結合順位が存在するとき同時に問題となるのが演算子の結合性である。これは同じ結合順位の演算子を 3 つ以上つなげて使ったとき、左右どちらの結合を優先するかを表す。例えば、7 - 5 - 3 は (7 - 5) - 3 と解釈されるが、これは _ - _ が左結合性を持つためである。

本手法ではこのような演算子の結合順位及び結合性を直接的にサポートしてはいないが、演算子の型情報を書き換えることによって実現することができる。型情報の書き換えは以下のようにして行う。ここでは簡単のため、結合順位は整数値で表現されており、値が小さいほど結合性が高いものとする。

1. 結合順位を含むような型を用意する。
   以降、基本の型が T で 結合順位が X の型を T(X) と書く。
   結合順位の最大値を M とする。

2. 各演算子の持つ型情報を以下のように書き換える。ただし、その演算子の結合順位を P とする。

   - 返り値の型 R を R(P) に書き換える。
   - 各引数の型 A を A(P) に書き換える。
     ただし、以下の例外の場合は A(P - 1) に書き換える。
     - 演算子が左結合でなく、対応する hole が演算子の最も左に位置していている場合
     - 演算子が右結合でなく、対応する hole が演算子の最も右に位置していている場合

3. 以下のような演算子を追加する。これらの解析順位は最低とする。
   `_ :: T(M) => T`
   `_ :: T(X - 1) => T(X)` ($\forall X > 0$)

例えば、次のような演算子が存在する場合を考える。

```
_ + _ :: Int => Int => Int (priority = 2, left association)
_ * _ :: Int => Int => Int (priority = 1, left association)
```

まず、Int 型に結合順位を付与した型 Int2 型、Int1 型、Int0 型を作成する。ここで、Int0 型は Int リテラルのための型である。_ + _ 及び _ * _ は左結合なので、Int2、Int1、Int0 を利用して以下のように書き換える。

```
_ + _ :: Int2 => Int1 => Int2
_ * _ :: Int1 => Int0 => Int1
```

最後に、作成した結合順位付きの型 Int2 型、Int1 型、Int0 型を Int 型と接続する以下の演算子を加える。

```
_ :: Int2 => Int
_ :: Int1 => Int2
_ :: Int0 => Int1
```

最終的に、次のような形に変換される。

```
_ + _ :: Int2 => Int1 => Int2
_ * _ :: Int1 => Int0 => Int1


_ :: Int2 => Int
_ :: Int1 => Int2
_ :: Int0 => Int1
```

このように、演算子の結合順位は型情報として表現される形に書き換えられる。そのため、同じ型を返す演算子であっても結合順位の異なる型であれば解析順序が付いている必要はない。

ちなみに、括弧は結合順位をリセットする効果があるため、これを演算子で表現すると、

```
( _ ) :: Int => Int0
```

となる。上記の変換では演算子の全オペランドの型を書き換えてしまうため、括弧のような演算子をユーザが定義することはできない。そのため、本手法では括弧を表現する演算子を解析時に自動的に導入するようにしている。一方、mixfix operators [6] などでは最左オペランドと最右オペランド以外のオペランドは結合順位の影響を受けないため、括弧も演算子として定義することができる。しかし、protean operators で同様のことを行った場合、それがユーザにとって分かりやすく矛盾がないものとなるかが不明なため、今回は採用しなかった。

3.4　サブタイプ関係

C++ や Java などの多くの言語では、型と型の間にサブタイプ関係を持たせることができるが、我々の構文解析手法はこのような関係を直接的にはサポートしていない。しかし、サブタイプをスーパータイプに関連付ける演算子を追加することによって、サブタイプ関係を表現することができる。ただし、その際結合順位を崩さないようにする必要がある。

Sub 型が Super 型のサブタイプであるとき、このサブタイプ関係は演算子により

```
_ :: Sub => Super
```

のように表現することができる。しかし、これは結合順位の情報を持たないので、演算子の結合順序を崩さないようにするためには、任意の結合順位 P に対して

```
_ :: Sub(P) => Super(P)
```

のようにする必要がある。
これらの演算子はユーザに定義された Super 型を返す通常の演算子よりも解析順位が低くなり、結合順序を示す `_ :: Super(P - 1) => Super(P)` よりも解析順位が高くなるように解析順位を設定しなければならない。

しかし、このようにした場合 Super 型を返す結合順位 `P - 1` の演算子よりも、Sub 型を返す結合順位 `P - 2` の演算子が優先的に選択されてしまう。そのためより正確には、

```
_ :: OP(T, P) => T(P)
```

のような演算子を追加し、`T(P)` を返すユーザ定義の演算子を `OP(T, P)` を返すように書き換え、サブタイプ関係を

```
_ :: OP(Sub, P) => Super(P)
```

のように表現することで、これを回避する。

例えば、Num 型が Int 型のスーパータイプで、

```
_ + _ :: Int => Int => Int (priority = 2, left association)
_ * _ :: Int => Int => Int (priority = 1, left association)
```

のような演算子がある場合、これは図 3.5 のように変換される。 これは例えば、

```
Num num = 1 + 2 * 3;
```

のように利用することができる。

```
_ + _ :: Int2 => Int1 => IntOp2
_ * _ :: Int1 => Int0 => IntOp1

_ :: NumOp2 => Num2
_ :: NumOp1 => Num1
_ :: NumOp0 => Num0

_ :: IntOp2 => Int2
_ :: IntOp1 => Int1
_ :: IntOp0 => Int0

_ :: IntOp2 => Num2
_ :: IntOp1 => Num1
_ :: IntOp0 => Num0

_ :: Num2 => Num
_ :: Num1 => Num2
_ :: Num0 => Num1

_ :: Int2 => Int
_ :: Int1 => Int2
_ :: Int0 => Int1
```

図 3.5: 結合順位・結合性・サブタイプ関係の変換結果

第4章　ProteaJ

我々は、前章で提案した構文解析手法を利用して、Java 言語のサブセット言語に protean operators を追加した言語 *ProteaJ* を作成した。ProteaJ は Java1.4 相当の機能を持つ。ただし、匿名クラス及び内部クラスはまだ現在のところ実装していない。

ProteaJ はユーザ定義演算子を定義し、モジュール化するための仕組みとして、演算子モジュールというモジュール機構を備える。ユーザはこの演算子モジュールを *using* 節によりインポートすることで言語内 DSL を利用することができる。

本章では、ProteaJ における protean operators の定義方法及び利用方法を解説し、ProteaJ のコンパイラの実装を簡単に紹介する。

4.1　protean operators の定義

ProteaJ では protean operators の定義は図 4.1 のように記述する。 はじめの GrepResult が返り値の型を表し、次の "grep" options regex file が演算子のパターンを表現している[1]。括弧付きの部分 (GrepOptions options,

```
GrepResult "grep" options regex file
  (GrepOptions options, Regex regex, File file): priority = 100
{
  GrepFileStream stream = open(file);
  if(options.ignoreCase()) {
    GrepResult r = stream.grep_ignoreCase(regex);
    stream.close();
    return r;
  }
  ...
}
```

図 4.1: grep 演算子の定義

[1]演算子のパターン部分については設計があまり良くなかったと思われる。パターンを "grep" options regex file と記述するのは grep _ _ _ と記述するのに比べてやや分かりづらい。将来的にはこの部分の再設計を行いたいと考えている。

```
<operator-definition>
  ::= <modifier>* <type> <pattern> <parameters> <throws> <priority> <body>
<modifier>
  ::= "rassoc" | "nonassoc" | "readas"
<pattern>
  ::= ( <name-part> | <hole> | <option> | <repetition> | <predicate> )+
<name-part>
  ::= <string-literal>
<hole>
  ::= <identifier>
<option>
  ::= <identifier> "?"
<repetition>
  ::= <identifier> ( "+" | "*" ) ( "(" <string-literal> ")" )
<predicate>
  ::= ( "&" | "!" ) <type>
<parameters>
  ::= "(" <parameter> ( "," <parameter> )* ")"
<parameter>
  ::= <type> "..."? <identifier> ( "=" <expression> )?
<priority>
  ::= ":" "priority" "=" <int-const>
```

図 4.2: protean operators の定義の文法の形式的な表現

Regex regex, File file) は演算子の引数で、演算子のパターンの hole に当たる部分とそれぞれ対応している。: priority = 100 はこの演算子の結合順位を示している。演算子の結合順位は現在の実装では簡単のため非負の整数値としている。最後の中括弧に囲まれた部分は演算子ボディでメソッドボディと同じように記述する。

protean operators の定義の文法を形式的に記述すると図 4.2 のようになる。これは拡張バッカス・ナウア記法による表現で、a? は a がオプションであることを示している。また、a* は a の 0 回以上の繰り返しを、a+ は a の 1 回以上の繰り返しを表現している。

protean operators には rassoc、nonassoc、readas の修飾子を指定することができる。rassoc 及び nonassoc は演算子の結合性を指定する修飾子で、rassoc は右結合、nonassoc は非結合を示す。どちらの修飾子も指定しなかった場合は左結合の演算子となる。readas 修飾子はこの演算子が lexical operator であることを示す[2]。

ProteaJ では、演算子のパターンとして単純な name part と hole 以外

[2]lexical operators の機能は指定した型のリテラルとして読むとも見なせるため readas という修飾子名を付けた。しかし、この名称は誤解を生む可能性があるため、lexical など他のものに変えることを検討中。

```
GrepResult "grep" options* regex file
  (GrepOption... options, Regex regex, File file)
  : priority = 100 { ... }
```

図 4.3: repetition を利用した protean operators の例

```
readas Regex r "$" (Regex r) : priority = 50 { ... }
readas Regex r1 r2 (Regex r1, Regex r2) : priority = 40 { ... }
readas Regex r "*" (Regex r) : priority = 30 { ... }
readas Regex !Dollar l (Letter l) : priority = 20 { ... }
readas Dollar "$" () : priority = 10 { ... }
```

図 4.4: predicate を利用した protean operators の例

にいくつかの便利な記法を提供している。*option* は値が入っても入らなくても良いような hole を表現しており、デフォルト引数と共に用いる。*repetition* は同じ hole の 0 または 1 回以上の繰り返しを表現しており、可変長引数と対応する。* が 0 回以上、+ が 1 回以上の繰り返しを表現しており、図 4.3 のようにして用いる。 -a 及び -b がそれぞれ GrepOption 型を帰す lexical operators であった場合、図 4.3 の演算子を利用すると以下の式は全て正しいものとみなされる。

```
GrepResult r1 = grep hel*o ~/Documents/Main.java;
GrepResult r2 = grep -a hel*o ~/Documents/Main.java;
GrepResult r3 = grep -a -b hel*o ~/Documents/Main.java;
```

predicate は先読みを表現する発展的な機能である。& のほうを *and predicate*、! のほうを *not predicate* と呼ぶ。and predicate は指定された型を期待される型として解析を試み、解析が成功すれば成功しバックトラックするといったものである。not predicate は逆に指定された型を期待される型として解析を試み、解析が失敗すれば成功しバックトラックする。いずれも入力を消費しないため、次のトークンの条件を指定するような利用法が考えられる。例えば、図 4.4 のような演算子を定義すると、

```
Regex e1 = hel*o;
Regex e2 = hel*o$;
```

は正しい正規表現として解析されるが、

```
Regex e3 = hel$o;
```

は解析に失敗してコンパイルエラーが報告される。

```
operators GrepOperators {
  GrepResult "grep" options* regex file
    (GrepOption... options, Regex regex, File file)
    : priority = 100 { ... }
  GrepOption "-i" () : priority = 0 { ... }
}
```

図 4.5: grep 演算子を定義した演算子モジュール

4.2　演算子モジュール

ProteaJ は protean operators のモジュール化機構として、演算子モジュールを備えている。これは 1 つの機能に関する複数の演算子をまとめたもので、ユーザはこの演算子モジュールを *using* 節によりインポートして利用する。

ProteaJ では全ての protean operators は演算子モジュールの内部で定義する。図 4.5 は grep のための演算子を定義した演算子モジュールの例である。 この演算子モジュールは次のようにして用いる。

```
using GrepOperators;

Regex regex;
File file;
...
GrepResult r = grep -i regex file;
```

同様にして、正規表現及びファイルパスのための演算子モジュールも作成すると、図 4.6 のようになる。ユーザは定義されたこれらの演算子モジュールを組み合わせて利用することができる。以下の例は、RegexOperators と FilePathOperators と GrepOperators の 3 つを組み合わせて利用した例である。

```
using RegexOperators;
using FilePathOperators;
using GrepOperators;

GrepResult r = grep -i hel*o ~/Documents/Main.java;
```

ProteaJ では、演算子モジュールを組み合わせて使うのを支援するために、演算子の結合順序は演算子モジュール内部で閉じるようになっている。複数の演算子モジュールを同時に利用したときの演算子の結合順位は、演算子モジュールを読み込む using 節の順番によって決定する。using 節が

```
operators RegexOperators {
  readas Regex r+ (Regex... r) : priority = 100 { ... }
  readas Regex r "*" (Regex r) : priority = 50 { ... }
  readas Regex l (Letter l) : priority = 0 { ... }
}

operators FilePathOperators {
  readas File dir "/" id
    (File dir, Identifier id) : priority = 100 { ... }
  readas File file "." ext
    (File file, Identifier ext) : priority = 50 { ... }
  readas File "~" () : priority = 0 { ... }
}
```

図 4.6: 正規表現及びファイルパスのための演算子モジュール

```
operators ExRegexOperators extends RegexOperators {
  readas Regex r "+" (Regex r) : priority = 50 { ... }
  readas Regex r "?" (Regex r) : priority = 50 { ... }
}
```

図 4.7: 演算子モジュールの拡張

先に書かれているほど、演算子の結合順位が高くなる。このことから、演算子モジュールの実装者は他の演算子モジュールの演算子の結合順位を知る必要がなく、また演算子モジュールの利用者は同時に利用することを想定していない複数の演算子モジュールを同時に利用できる。

ProteaJ は既存の演算子モジュールに演算子を追加した新たな演算子モジュールを作成する機能を提供している。図 4.7 のように演算子モジュールの宣言に *extends* 節を記述することで、指定した演算子モジュールを拡張した演算子を作成することができる。このとき、演算子の結合順位はもととなる演算子モジュールの結合順位がそのまま引き継がれる。そのため、その結合順位の中に新しい演算子を追加することが可能となる。

protean operators の解析順序は演算子の定義した順番 (先に定義したほうが優先) となる。例えば、図 4.8 のような演算子モジュールを利用すると、

```
Regex e = hel*+o;
```

は `_ *+` 演算子を使った式として解釈される。これは解析順序が

```
_ *+  <  _ *
```

であるためである。もし解析順序が逆であれば、この式は `_ *` 演算子と `_ +` 演算子を利用した式として解釈される。

```
operators RegexOperators {
  readas Regex r+ (Regex... r) : priority = 100 { ... }
  readas Regex r "*+" (Regex r) : priority = 50 { ... }
  readas Regex r "++" (Regex r) : priority = 50 { ... }
  readas Regex r "?+" (Regex r) : priority = 50 { ... }
  readas Regex r "*" (Regex r) : priority = 50 { ... }
  readas Regex r "+" (Regex r) : priority = 50 { ... }
  readas Regex r "?" (Regex r) : priority = 50 { ... }
  readas Regex l (Letter l) : priority = 0 { ... }
}
```

図 4.8: 演算子の定義順序が意味を持つ例

ProteaJ では、通常の protean operators と lexical operators を強く区別している。全ての lexical operators の結合順位は通常の protean operators よりも必ず高いものとみなされ、Java のメソッド呼び出しなどよりも結合順位が高いものとみなされる。通常の protean operators と lexical operators の結合順位は同じ演算子モジュールの中でも明確に区別されており、同じ値を用いたとしても通常の protean operators と lexical operators が同じ結合順位となることはない。そのため、lexical operators の引数には lexical operators の式以外は入ることができない。これにより字句レベルと構文レベルがはっきりと区別されるため、DSL の実装がより容易になる。

4.3　演算子を利用できる場所

protean operators は期待される型が分かるような場所でのみ利用することができる。そのため、代入文の左辺などでは protean operators を利用することはできない。これは、Java 言語では代入文の左辺を記述するよりも前にその部分の型を知ることはできないためである。

これは一見非常に大きい制約だが、複雑な式が現れるような場所のほとんどでは期待される型を知ることができるため、実用上はそれほど問題とならない。以下の表 4.1 は ProteaJ において式が現れる場所と期待される型の対応を示している。 これから分かるとおり、式が使われる場所では代入文の左辺とメソッド呼び出しのレシーバを除くすべての部分で期待される型を知ることができる。代入文の左辺では複雑な式が現れることはほとんど無いため、これも実用上は問題とならない。メソッド呼び出しのレシーバは複雑な式が現れうるが、メソッド呼び出しのレシーバ部分にユーザ定義演算子による式を利用するような場合は、そのメソッド呼び出しもユーザ定義演算子で表現すれば良い。ちなみに、コンパイラが自動的

場所	期待される型 (わからない場合は x)
代入文の左辺	x
代入文の右辺	左辺の型
メソッド呼び出しのレシーバ	x
メソッド呼び出しの引数	対応する仮引数の型
演算子の引数	対応する仮引数の型
if, for, while の条件式	boolean
switch の引数	char または int
case の条件式	char または int
throw の引数	throws に書かれた型または catch されている型
return の引数	返り値の型
フィールド初期値	フィールドの型
option のデフォルト引数	対応する仮引数の型
式文の式	void

表 4.1: 式が現れる場所と期待される型

に Java のメソッドを演算子に変換することも原理的には可能だが、クラスパスの通った全てのクラスのメソッドを演算子に変換するのはコストがかかりすぎるため事実上不可能である。

4.4　コンパイラの実装

我々は ProteaJ のコンパイラを Java 言語によりスクラッチから実装した。本手法は字句・構文解析と型チェックを組み合わせた特殊な構文解析方法を用いるため、既存のコンパイラフレームワークなどを利用することはできなかった。プログラムサイズはコンパイラ全体で約 12000 行、そのうち構文解析部分が約 5000 行となっている。

ProteaJ のコンパイラの動作は大きく 2 段階のフェーズに分かれる。1 段階目が宣言部分の解析フェーズで 2 段階目が本体部分の解析フェーズである。ProteaJ は Java 言語をもとにしているため、メソッドなどをそれが定義されるよりも前に用いる、前方参照が許されている。しかし、我々の手法では式部分の構文解析を行うのに型やメソッド、演算子などのメタ情報を利用するため、メソッドボディ等を解析するよりも前にそれらの情報を収集する必要がある。そのため、コンパイラはまずメソッドボディやフィールド初期値を除いたクラスやメソッド、演算子の宣言部分のみを解析してメタ情報を収集する。これが 1 段階目の宣言部分の解析フェーズである。2 段階目の本体部分の解析フェーズでは、宣言部分の解析フェーズ

で集めたメタ情報を利用して、メソッドボディやフィールド初期値を解析する。

本体部分の解析は前章の手法に沿って行う。ただし、ProteaJ ではそれに加えて Java の文法も解釈する必要があるため、解析手法に多少の工夫を加えている。ProteaJ では式を演算子を利用して解析するのに失敗した場合、その部分を通常の Java のルールで解析する。これは ローカル変数参照やメソッド呼び出しなどの演算子によらない式を解析するためである。通常の Java のルールによる解析も失敗した場合、コンパイラは lexical operators による解析を試みる。これも失敗した場合、コンパイラはコンパイルエラーを報告する。

4.5　ケーススタディ

本節では、ProteaJ を用いて我々が実際に実装した DSL を紹介し、protean operators が高い表現力を持っていることを示す。

4.5.1　標準出力

ProteaJ では `void` を返すような演算子を定義することも許している。`void` を返す演算子も他の演算子と同様にして定義することが可能で、例えば次のようにして定義することができる。

```
operators OutputOperators {
  void "p" msg (String msg): priority = 0 {
    System.out.println(msg);
  }
}
```

この演算子は以下のようにして利用することができる。

```
using OutputOperators;
...
p "Hello, world!"
```

`p` は `String` を引数に取り、その文字列を出力する。上記の例では、`Hello, world!` という文字列が表示される。

ProteaJ では式文の式は `void` を返すものとみなされる。そのため、`void` を返す演算子を作ることでプログラマはユーザ定義文を作ることができる。

```
operators RegexOperators {
  readas Regex l "|" r (Regex l, Regex r) : priority = 200
  readas Regex rs+ (Regex... rs) : priority = 100
  readas nonassoc Regex r "?+" (Regex r) : priority = 50
  readas nonassoc Regex r "*+" (Regex r) : priority = 50
  readas nonassoc Regex r "++" (Regex r) : priority = 50
  readas nonassoc Regex r "??" (Regex r) : priority = 50
  readas nonassoc Regex r "*?" (Regex r) : priority = 50
  readas nonassoc Regex r "+?" (Regex r) : priority = 50
  readas nonassoc Regex r "?" (Regex r) : priority = 50
  readas nonassoc Regex r "*" (Regex r) : priority = 50
  readas nonassoc Regex r "+" (Regex r) : priority = 50
  readas nonassoc Regex r "{" n "}" (Regex r, Nat n) : priority = 50
  readas Regex "[" es+ "]" (RegClsElem... es) : priority = 40
  readas RegClsElem f "-" t (RegLetter f, RegLetter t) : priority = 30
  readas RegClsElem l (RegLetter l) : priority = 20
  readas Regex "." () : priority = 10
  readas Regex l (RegLetter l) : priority = 10
  readas RegLetter l (Letter l) : priority = 0
  readas RegLetter d (Digit d) : priority = 0
}
```

図 4.9: 正規表現 DSL

4.5.2　正規表現

lexical operators を活用することにより、ProteaJ では非常に複雑なユーザ定義リテラルを作ることができる。例えば、図 4.9 は正規表現を実現する演算子モジュールで、次のようにして用いることができる。

```
using OutputOperators;
using RegexOperators;
...
Regex stnumber = [0-9]{2}(B|M|D)[0-9]{5};
Matcher m = regex.matcher(text);
if(m.find()) {
  p "match : " + m.group();
}
```

正規表現リテラルが利用されているのは 4 行目の右辺である。この正規表現リテラルは _ _ や [_]、_ { _ } や_ - _ などの様々な演算子からなっており、学籍番号にマッチするような正規表現を表している。ここで、(_) は ProteaJ が提供している括弧で、演算子の結合順位をリセットする機能を持つ。

```
operators StringOperators {
  String left "+" right (String left, String right) : priority = 50 {
    return left.concat(right);
  }
}
operators ExStringOperators extends StringOperators {
  String buf (StringBuilder buf) : priority = 150 {
    return buf.toString();
  }
  StringBuilder left "+" right
  (StringBuilder left, String right) : priority = 100 {
    left.append(right);
    return left;
  }
  StringBuilder s1 "+" s2 "+" s3
  (String s1, String s2, String s3) : priority = 0 {
    StringBuilder buf = new StringBuilder();
    buf.append(s1).append(s2).append(s3);
    return buf;
  }
}
```

図 4.10: String の足し算とそれを最適化する演算子モジュール

4.5.3　簡単な最適化

protean operators の別の活用方法として、protean operators を利用した最適化が挙げられる。例えば、String の足し算 _ + _ は多くの String を接続する場合には非効率であり、そのようなときは StringBuilder を利用する。ProteaJ の protean operators はこのようなケースで利用することができる。

図 4.10 は String の足し算を表現する演算子モジュールとそれを最適化する演算子モジュールの例である。 StringOperators は通常の String の足し算 _ + _ を表現しており、ExStringOperators はその足し算が 3 回以上連続する場合に StringBuilder を使うように StringOperators を拡張している。これらの演算子をまとめると以下のようになっている。

```
_ :: StringBuilder => String (priority = 4)
_ + _ :: StringBuilder => String => StringBuilder (priority = 3)
_ + _ :: String => String => String (priority = 2)
_ + _ + _ :: String => String => String => StringBuilder (priority = 1)
```

我々の手法では、演算子の結合順位は型情報に置き換えられるため、これらの演算子は次のように書き換わる。

```
_ :: StrBld3 => Str4
_ + _ :: StrBld3 => Str2 => StrBld3
_ + _ :: Str2 => Str1 => Str2
_ + _ + _ :: Str1 => Str1 => Str0 => StrBld1

_ :: Str4 => String
_ :: Str3 => Str4
_ :: Str2 => Str3
_ :: Str1 => Str2
_ :: Str0 => Str1

_ :: StrBld4 => StringBuilder
_ :: StrBld3 => StrBld4
_ :: StrBld2 => StrBld3
_ :: StrBld1 => StrBld2
_ :: StrBld0 => StrBld1
```

そのため、`"foo" + "bar" + "baz"` のような 3 つ以上の `String` の足し算の場合、`String -> Str4 -> StrBld3 -> StrBld2 -> StrBld1` の順序で解析が進むため、`_ + _ + _` 演算子を用いた形に解析される。しかし、`"foo" + "bar"` のような 2 つの `String` の足し算の場合、この `_ + _ + _` 演算子の適用に失敗するため、通常の `_ + _` 演算子により足し算が行われる。

4.5.4　SQL

ProteaJ では、非常に複雑な DSL を実現することも可能である。我々は SQL を表現する DSL を 2 つの演算子モジュール `SQLOperators` と `FilePathOperators` として実装した。これらの演算子モジュールを利用することで、例えば図 4.11 のようなプログラムを記述することができる。

```
import java.sql.*;

using FilePathOperators;
using SQLOperators;
using OutputOperators;
using ExStringOperators;

public class Main {
  static boolean existTable(String tbl) throws Exception {
    ResultSet tables = select tablename from sys.systables
                       where tablename = tbl.toUpperCase();
    return tables.next();
  }

  static void insertMember(int id, String name) throws Exception {
    insert into members ( user_id, name ) values ( id, name );
  }

  public static void main(String[] args) throws Exception {
    connect to ./database.db;
    if(existTable("members")) drop table members;
    create table members (
      user_id int not null primary key,
      name varchar(64) not null
    );

    if(existTable("posts")) drop table posts;
    create table posts (
      id int not null generated always as identity,
      date timestamp default current timestamp,
      user_id int,
      comment long varchar
    );

    insertMember(123, "ichikawa");
    insertMember(345, "ohtani");
    insertMember(567, "hiramatsu");
    insert into posts ( user_id, comment )
               values ( 123, "Ohayo!" );

    ResultSet rs = select * from members;
    while(rs.next()) {
      p rs.getInt(1) + " " + rs.getString(2);
    }
    commit;
    disconnect;
  }
}
```

図 4.11: SQL DSL を利用したプログラムの例

第5章　考察

本章では、protean operators の表現力が解析表現文法 (PEG) [8] と同等であることを示すことにより protean operators の有用性を示す。また、我々の提案する構文解析手法がほとんどの場合に線形時間で動作することを示し、従来手法で protean operators を実現しようとした場合との比較検証を行う。

5.1　protean operators の表現力

protean operators は解析表現文法 (PEG) [8] と等価な表現力を持つ。任意の protean operators で書かれた文法は PEG に変換可能で、逆に PEG で書かれた文法規則は protean operators で表現することができる。本節では、まず PEG についての解説を行い、その後 protean operators との相互変換可能性を検証する。

5.1.1　PEG

PEG は文脈自由文法 (CFG) などと同じ形式文法の一種である。PEG における文法の定義は CFG の表現方式の一種であるバッカス・ナウア記法 (BNF) と似ているが、選択を表現する演算子が大きく異なる。BNF では選択を表す演算子 | は「これらのうちどれか」を表現しているが、PEG における選択を表す演算子 \ は「順番に試して最初に成功したものを採用」を表現している。そのため、CFG と異なり PEG は曖昧性を持つ文法を表現できない。

PEG は以下の 3 つ組からなる。

- 非終端記号の有限集合 V_N
- 終端記号の有限集合 V_T
- 解析規則の有限集合 R

ただし $V_N \cap V_T = \emptyset$ でなければならない。各解析規則 $r \in R$ は 非終端記号 $A \in V_N$ と *parsing expression* e の 2 つ組で、$A \leftarrow e$ のように記述

protean operators			PEG
空文字列	""	→	空文字列 ε
name part	a	→	終端記号 a
hole	$\_ : T$	→	非終端記号 T
演算子	$e_1\ e_2\ ...\ e_n$	→	並び $e_1e_2...e_n$
型 T を返す演算子	$op_1 > op_2 > ... > op_n$	→	$T \leftarrow op_1/op_2/.../op_n$

表 5.1: protean operators から PEG への変換規則

する。parsing expressions 全体の集合 P は以下のように帰納的に定義される。$e, e_1, e_2 \in P$ のとき、以下は全て P に含まれる。

1. ε：空文字列
2. $a \in V_T$：任意の終端記号
3. $A \in V_N$：任意の非終端記号
4. e_1e_2：並び
5. e_1/e_2：順序付き選択
6. $e?$：オプション
7. $e*$：０回以上の繰り返し
8. $e+$：１回以上の繰り返し
9. $\&e$：and predicate
10. $!e$：not predicate

このうち、オプション $e?$、０回以上の繰り返し $e*$、１回以上の繰り返し $e+$、and predicate $\&e$、及び not predicate $!e$ はそれ以外の規則を用いて表現できることが知られている [8]。そのため、以降では 1 から 5 までを parsing expressions の生成規則と見なすものとする。

5.1.2　protean operators から PEG への変換

protean operators は引数及び返り値の型によってオーバーロードされるため、各 protean operators は 型を非終端記号とした解析規則と見なすことができる。そのため、表 5.1 のようにして protean operators を PEG に変換することができる。ここで、$op_i > op_j$ は演算子 op_i のほうが演算子 op_j よりも解析順位が高いことを表す。
例えば、次のような protean operators は、

```
grep _ _ _ :: GrepOptions => Regex => File => GrepResult
_ _ :: GrepOption => GrepOptions => GrepOptions
"" :: GrepOptions
-i :: GrepOption
_ _ :: Regex => Regex => Regex
_ * :: Regex => Regex
_ :: Letter => Regex
_ / _ :: File => Identifier => File
_ . _ :: File => Identifier => File
~ :: File
```

以下のような PEG に変換できる[1]。

```
GrepResult ← "grep" GrepOptions Regex File
GrepOptions ← GrepOption GrepOptions / ε
GrepOption ← "-i"
Regex ← Regex Regex / Regex "*" / Letter
File ← Identifier "/" File / Identifier "." File / "~"
```

5.1.3　PEG から protean operators への変換

PEG から protean operators へは基本的には非終端記号を型に置き換えることで変換することができる。ただし、順序付き選択 e_1/e_2 の変換は多少工夫する必要がある。表 5.2 は PEG から protean operators への変換規則を示している。解析規則、空文字列、終端記号、非終端記号、並びは単純な置き換えにより変換することができる。順序付き選択はやや複雑で、e_1/e_2 は以下のような順序で変換する。まず新しい型 X を作成し変換結果を X 型の hole $\_ : X$ とする。e_1 および e_2 を変換した結果のパターンを持つ演算子 op_1 および op_2 を作成し、その返り値の型を X とする。op_1 と op_2 の解析順序を $op_1 > op_2$ とする。
この変換を適用することで、例えば次のような PEG は、

```
Expr ← Expr "+" Expr / Expr "-" Expr / Term
Term ← Term "*" Term / Term "/" Term / Value
```

次のような protean operators に変換できる。

[1] この PEG は左再帰を含んでいる。しかし、左再帰除去を行う手法は広く知られており、また PEG は繰り返しを表現する記法を持っているため、容易に左再帰を含まない well-formed な PEG に変換することができる。

PEG			protean operators
解析規則	$A \leftarrow e$	$\rightarrow$	A を返す演算子 op op のパターンは e
空文字列	ε	$\rightarrow$	空文字列 ""
終端記号	a	$\rightarrow$	name part a
非終端記号	T	$\rightarrow$	hole $\_ : T$
並び	$e_1 e_2 ... e_n$	$\rightarrow$	並び e_1 e_2 ... e_n
順序付き選択	$e_1/e_2/.../e_n$	$\rightarrow$	hole $\_ : X$ ただし X は新しい型で、 演算子 $op_1 > op_2 > ... > op_n$ は X を返す op_i のパターンは e_i

表 5.2: PEG から protean operators への変換規則

```
_ :: Expr2 => Expr
_ + _ :: Expr => Expr => Expr2
_ - _ :: Expr => Expr => Expr2
_ :: Term => Expr2
_ :: Term2 => Term
_ * _ :: Term => Term => Term2
_ / _ :: Term => Term => Term2
_ :: Value => Term2
```

5.2　構文解析の計算量

protean operators のような汎用的なユーザ定義演算子が今まで存在しなかったのは、このような演算子を含む式は解析が難しく、従来の手法では時間がかかりすぎたためである。ユーザ定義演算子による字句・構文規則の追加は文法規則に曖昧性を導入してしまい、それらの曖昧性は文法ではなく型というプログラミング言語のセマンティクスを用いて解決しなければならない。そのため従来はこのような演算子を実現するために、曖昧さを含む文脈自由文法 (CFG) 全体を解析できる scannerless parser によって構文解析を行い、その結果として出力されたありうる構文木全てに対して型チェックを行っていた。この手法は確かに型で曖昧性を解決しているが、構文解析のコストが非常に大きくなってしまう。

本節では、protean operators のようなユーザ定義演算子を解析するための従来の手法を解説し、それらの計算量を示す。その後、本手法の計算量についての評価を行い、従来の手法と比較して十分高速であることを示す。

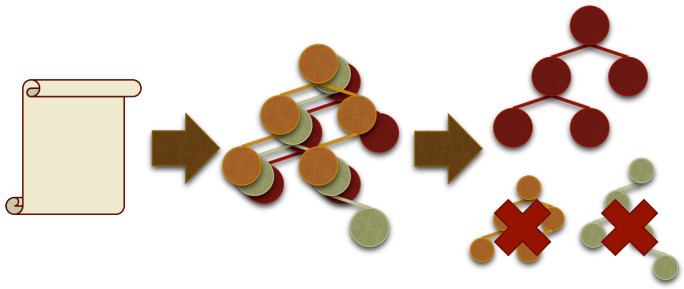

図 5.1: Isabelle や Metaborg の手法のイメージ図

5.2.1　Type-based disambiguation

Isabelle [20] や Metaborg [5] では曖昧性を含む CFG 全体を解析可能な構文解析器を用いてありうる構文木を全て列挙し (*parse forest* を生成し)、それぞれの構文木を型チェックすることにより型レベルでの曖昧性解決を行う。図 5.1 はこの手法の簡単なイメージ図である。この手法は汎用性が高く、ホスト言語を問わず様々な言語で実装することができるが、その反面構文解析のコストが非常に高くなってしまう。

曖昧性を持つ任意の CFG を解析可能な構文解析アルゴリズムは既にいくつも知られているが、それらは曖昧性・非決定性の高い文法に弱く、計算時間が非常に大きくなってしまう。多数の DSL を組み合わせて利用した場合、それぞれの DSL の文法が曖昧性を含まなくても、構文解析器が扱うそれらを併合した文法には曖昧性が生じてしまう [16] ため、そのような構文解析アルゴリズムはコストが高い。

chart parsing [17] は曖昧性を含む CFG 全体を解析可能な構文解析手法として広く知られている。Isabelle の採用している構文解析器もこの chart parser の一種である。chart parser は曖昧性を持つ任意の CFG を解析可能だが、曖昧性・非決定性の高い文法に対しては計算量が $O(|G| * n^3)$ となることが知られている。ここで、$|G|$ は文法サイズ、n は入力長を表す。多数の DSL を組み合わせて利用すると文法に多くの曖昧性が生じるため、計算量は最悪ケースに陥りやすく、また多数の DSL を導入は $|G|$ を大きくしてしまうため、計算時間が肥大化しやすい。また、chart parser は通常 scannerless parser として作られておらず、字句規則の拡張には対応することができない。

Generalized LR (GLR) parsing [27] は chart parsing と並んでよく知られた、曖昧性を持つ任意の CFG を解析可能な構文解析手法である。Metaborg は GLR parsing を拡張した Scannerless GLR (SGLR) parsing [29] を利用している。SGLR parser は字句規則と構文規則を同じように

扱うため、構文規則の拡張と同様にして字句規則を拡張することができる。GLR parser は決定的な文法に対しては入力長の線形時間で動作するが、曖昧性・非決定性の高い文法に対しては chart parser と同様に入力長の 3 乗のオーダーに比例する時間がかかる[2]。
SGLR parser では入力長 n は入力トークン数ではなく入力の文字数であるため、n^3 が非常に大きくなってしまう。そのため、文法の曖昧性・非決定性が大きい場合は解析時間が非常に長くなってしまう危険性がある。多数の DSL の利用は文法に曖昧性・非決定性を導入するが、字句規則を追加する場合は特にそれが顕著なため、計算時間は最悪ケースとなりやすい。これは、字句規則は 1 文字単位で入力を扱うため構文規則よりも文法が衝突しやすいこと、およびその衝突した文法がソースコード中に出現しやすいことによる。

また、曖昧性が非常に大きい場合、型チェックを行うべき構文木の数が非常に多くなってしまうという問題も発生する。特に、多くの部分木に関係するような、例えば識別子と衝突するような、字句規則の追加はありうる構文木の数を爆発的に増大させてしまう。

5.2.2 Type-oriented island parsing

Type-oriented Island Parsing (TIP) [23] は island parsing [25] と呼ばれる chart parsing の一種をもとにした解析方法で、構文木を作りながら型チェックを同時に行うことで Isabelle の手法よりも高速に解析を行うことを可能にしている。この手法は型チェックを行いながら構文木を組み上げることで、型のあわない部分木の成長を阻害し、効率的に構文解析を行うことができる。

型のあわない部分木の成長の阻害により、TIP の最悪計算量は $|G|$ に比例しなくなり、$O(n^3)$ となる。そのため、TIP は DSL を組み合わせて利用するのに強く、composability が高いと言える。

しかし、TIP は scannerless parser でないため字句規則の拡張を許さず、また TIP のアルゴリズムを scannerless parser に適用することが可能であるかどうかどうかは自明ではない。TIP は 1 トークンの変数または定数を見つけ、そこを起点として解析を行う。しかし、字句規則の拡張を行うことを許した場合はこのような規則をそのまま利用することは難しい。また、scannerless parser は入力長 n が入力の文字数となるため、n^3 のコストは無視できない非常に大きいコストとなってしまう。

[2]GLR parsing の最悪計算量も $O(|G| * n^3)$ だと予想されるが、GLR parsing の計算量と $|G|$ の相関に関する記述を発見できなかったため、$|G|$ に関する議論は行わない。

5.2.3　本手法の計算量

我々の提案手法はほとんどの場合に $O(n)$ で解析を行うことができる。本手法は期待される型に基づいて演算子を限定するため、通常のユースケースでは解析時間は $|G|$ に比例しない。また、左再帰を許す packrat parsing [31] に基づいているため、ほとんどの場合において線形時間での解析が可能となっている。

本手法は期待される型により演算子を限定するので、解析時間はこの限定された演算子の数に比例する。つまり、解析で期待される型として利用される型 T に対して T を返す演算子全体の集合を G_T とし、$|G_T|$ を G_T に含まれる演算子の個数とすると、本手法の解析時間は $max|G_T|$ に比例する。

通常、異なる DSL は異なる型を用いる [23] ため、$max|G_T|$ は非常に小さい値となる。どのため、解析時間は $|G|$ に比例しないとみなして良い。しかし、`Object` 型のような非常に汎用性の高い型を期待される型として利用した場合は、$max|G_T| \simeq |G|$ となってしまう。このことから、 protean operators はできるだけ特殊な型を返すべきであり、ユーザも `Object` 型のような汎用性の高い型ではなく DSL 固有の型を利用するべきであることがわかる。

packrat parsing [7] は再帰下降構文解析にメモ化を組み合わせたもので、任意の左再帰を含まない well-formed な PEG を入力の線形時間で解析することができる。左再帰を許す packrat parsing はこのアルゴリズムに少し手を加えたもので、解析を行う前にメモテーブルに `Fail` を書きこんでおくことで、左再帰を発見して解決する。

左再帰を許す packrat parsing はほとんどの場合は通常の packrat parsing と同様に線形時間で動作するが、メモ化が無駄になるような特定のパターンの構文の場合は線形時間を超える時間がかかってしまう。例えば次のような protean operators があったとき、

```
_ 2 :: Ones  => Start
1 _ :: Start => Start
"" :: Start
_ 1 :: Ones => Ones
1 :: Ones
```

この演算子は 0 個以上の `1` の繰り返しを `Start` 型の値として認識することができるが、その解析には $O(n^2)$ かかる。これは `Ones` 型の式の始点が 1 つずつずれてしまうためメモ化が無駄になってしまい、その分無駄な解析が行われるためである。しかし、このような文法が現実的なプログラミング言語で現れることはほとんどないことが知られている [9]。

第6章　まとめと今後の課題

6.1　まとめ

本研究では、mixfix operators 以上に強力な構文拡張が可能で、ユーザ定義リテラルも実現することのできるようなユーザ定義演算子 protean operators とその構文解析手法を提案した。protean operators は、

- hole と name part の並びからなり、その並べ方には制限がない。
- それぞれ個別の字句規則を持つ。
- 引数及び返り値の型でオーバーライドされる。
- 同じ型を返す演算子の間に順序関係が必要

といった特徴を持つ。我々の提案した構文解析手法は演算子を期待される型の情報を用いて制限することで、型が合わない演算子を利用した解析パスの枝刈りを行う。また、左再帰を許す packrat parsing を組み合せて利用することで、バックトラックのオーバーヘッドを削減し、高速な解析を可能にした。

我々は提案した構文解析を利用して protean operators を持つ言語 ProteaJ を開発し、その実装手法と応用例を紹介した。並べて、我々は protean operators の表現力が解析表現文法 (PEG) と同等であることを示し、本手法の有用性を示した。また、本手法がほとんどの場合において入力長の線形時間で動作することを示し、従来手法で実現した場合に比べて十分高速であることを示した。

6.2　今後の課題

我々の今後の課題の１つは、ジェネリクス及び型の推論を protean operators に取り入れることである。我々の構文解析手法は型情報を利用して解析を進めるため、型情報を知ることができない場所、代入文の左辺などでは利用することができない。しかし、代入文はジェネリクスを利用すれば、

```
_ = _ :: TypeName[T] => T => void
```

のような形で表現できることが考えられる。しかし、このような演算子の解析を行うためには、1 つ目の hole を解析した際に型 T を推論しなければならない。このような推論が従来と同じ型推論手法で実現できるかどうかは明確ではない。そのため、ジェネリックな型と推論の方法を整理し、protean operators に取り入れることが可能であるか検証したい。

今後の課題の 2 つ目は、ローカル変数宣言などのメタレベルのコードを protean operators で表現可能にすることである。protean operators はあくまでも演算子であるため、ローカル変数宣言やクラス定義などのメタレベルの表現はできない。この問題を解決することで、プログラム全体を protean operators で表現することを可能にし、決まった文法を持たないプログラミング言語の開発を可能としたい。

参考文献

[1] : Agda Wiki, `http://wiki.portal.chalmers.se/agda/pmwiki.php`.

[2] : ScalaTest, `http://www.scalatest.org/`.

[3] Bachrach, J. and Playford, K.: D-Expressions: Lisp Power, Dylan Style, Technical report (1999).

[4] Bravenboer, M., Vermaas, R., Vinju, J. and Visser, E.: Generalized type-based disambiguation of meta programs with concrete object syntax, *Proceedings of the 4th international conference on Generative Programming and Component Engineering*, GPCE'05, Berlin, Heidelberg, Springer-Verlag, pp. 157–172 (2005).

[5] Bravenboer, M. and Visser, E.: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions, *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, New York, NY, USA, ACM, pp. 365–383 (2004).

[6] Danielsson, N. A. and Norell, U.: Parsing mixfix operators, *Proceedings of the 20th international conference on Implementation and application of functional languages*, IFL'08, Berlin, Heidelberg, Springer-Verlag, pp. 80–99 (2011).

[7] Ford, B.: Packrat parsing:: simple, powerful, lazy, linear time, functional pearl, *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, ICFP '02, New York, NY, USA, ACM, pp. 36–47 (2002).

[8] Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation, *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, New York, NY, USA, ACM, pp. 111–122 (2004).

[9] Ford, B. and Kaashoek, M. F.: Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking (2002).

[10] Fowler, M.: Language Workbenches: The Killer-App for Domain Specific Languages?, `http://martinfowler.com/articles/languageWorkbench.html` (2005).

[11] Goguen, J. A., Winkler, T., Meseguer, J., Futatsugi, K. and Jouannaud, J.-P.: Introducing OBJ (1993).

[12] Grf, A.: The Pure Programming Language, `http://code.google.com/p/pure-lang/`.

[13] Hickey, J.: *Introduction to Objective Caml*, Cambridge University Press (2007). Forthcoming. An older version of the book is available at `http://files.metaprl.org/doc/ocaml-book.pdf`.

[14] Hudak, P.: Building domain-specific embedded languages, *ACM Comput. Surv.*, Vol. 28, No. 4es (1996).

[15] Jones, S. P.(ed.): *Haskell 98 Language and Libraries: The Revised Report*, http://haskell.org/ (2002).

[16] Kats, L. C., Visser, E. and Wachsmuth, G.: Pure and declarative syntax definition: paradise lost and regained, *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, New York, NY, USA, ACM, pp. 918–932 (2010).

[17] Kay, M.: Readings in natural language processing, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, chapter Algorithm schemata and data structures in syntactic processing, pp. 35–70 (1986).

[18] Kohlbecker, E., Friedman, D. P., Felleisen, M. and Duba, B.: Hygienic macro expansion, *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, New York, NY, USA, ACM, pp. 151–161 (1986).

[19] Odersky, M., Spoon, L. and Venners, B.: *Programming in Scala: A Comprehensive Step-by-step Guide*, Artima Incorporation, USA, 1st edition (2008).

[20] Paulson, L. C.: *Isabelle: a Generic Theorem Prover*, Lecture Notes in Computer Science, No. 828, Springer – Berlin (1994).

[21] Sheard, T.: Using MetaML: a Staged Programming Language, *IN ADVANCED FUNCTIONAL PROGRAMMING*, Springer-Verlag, pp. 207–239 (1999).

[22] Sheard, T. and Jones, S. P.: Template meta-programming for Haskell, *SIGPLAN Not.*, Vol. 37, No. 12, pp. 60–75 (2002).

[23] Silkensen, E. and Siek, J. G.: Well-typed Islands Parse Faster, *CoRR*, Vol. abs/1201.0024 (2012).

[24] Skalski, K., Moskal, M. and Olszta, P.: Meta-programming in Nemerle (2004).

[25] Stock, O., Falcone, R. and Insinnamo, P.: Island parsing and bidirectional charts, *Proceedings of the 12th conference on Computational linguistics - Volume 2*, COLING '88, Stroudsburg, PA, USA, Association for Computational Linguistics, pp. 636–641 (1988).

[26] Team, T. C. D.: The Coq Proof Assistant Reference Manual (2009).

[27] Tomita, M.: An efficient context-free parsing algorithm for natural languages, *Proceedings of the 9th international joint conference on Artificial intelligence - Volume 2*, IJCAI'85, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., pp. 756–764 (1985).

[28] Tratt, L.: Domain Specific Language Implementation via Compile-Time Meta-Programming, *TOPLAS*, Vol. 30, No. 6, pp. 1–40 (2008).

[29] van den Brand, M. G. J., Scheerder, J., Vinju, J. J. and Visser, E.: Disambiguation Filters for Scannerless Generalized LR Parsers, *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, London, UK, UK, Springer-Verlag, pp. 143–158 (2002).

[30] Ward, M. P.: Language Oriented Programming, Software—Concepts and Tools, Vol. 15, pp. 147–161 (1995).

[31] Warth, A., Douglass, J. R. and Millstein, T.: Packrat parsers can support left recursion, *Proceedings of the 2008 ACM SIGPLAN*

symposium on Partial evaluation and semantics-based program manipulation, PEPM '08, New York, NY, USA, ACM, pp. 103–110 (2008).

[32] Weirich, J.: Rake – Ruby Make, `http://rake.rubyforge.org/`.