

$\neg<><UU$ (notavaCC):

オブジェクト指向抽象構文木を生成するコンパイラ・コンパイラ

小 藤 哲 彦[†] 河 野 健 二^{††} 竹 内 郁 雄^{††}

具象構文と抽象構文は対応関係が強く、これらを別々に記述するのは無駄が多い。この無駄を省くために、具象構文と抽象構文を一体として記述する手法がいくつか提案されている。本論文では、具象構文と抽象構文を一体として記述し、抽象構文木を出力するパーザを生成するコンパイラ・コンパイラ $\neg<><UU$ (notavaCC) の設計と実装について述べる。 $\neg<><UU$ は、継承、ラベル付け、エイリアスの構文記法を用い、オブジェクト指向に基づいた抽象構文を記述することができる。 $\neg<><UU$ は、算術式のような従来の研究では表現できない抽象構文を記述することができ、表現力が高い。本論文では $\neg<><UU$ が実用的な性能を持っていることも示す。

$\neg<><UU$ (notavaCC):

A Compiler Compiler that Generates Object-oriented Abstract Syntax Trees

TETSUHIKO KOTO,[†] KENJI KONO^{††} and IKUO TAKEUCHI^{††}

Because of the similarity between abstract syntax and concrete syntax, it is wasteful to write these syntax descriptions individually. The paper describes the design and implementation of a compiler compiler named $\neg<><UU$ (notavaCC), which has an integrated notation that can represent both of abstract syntax and concrete syntax, and generates a parser that builds an object-oriented abstract syntax tree. Using the notation for *inheritance*, *labeling* and *aliases*, $\neg<><UU$ enables us to generate an object-oriented syntax tree of an arithmetic expression, which existent compiler compilers cannot generate. The paper also describes the practicability of $\neg<><UU$.

1. はじめに

プログラムなどの複雑な構造を表現するのにしばしばテキストが利用される。テキストを用いると、既存のプログラムを利用して編集・検索・印刷といった様々な処理を行うことができ、利便性が高い。そのような、計算機で処理されることを前提に作られた言語を扱うプログラムでは、構文解析の結果として抽象構文木¹⁾ (以下構文木と呼ぶ) を扱うことが望ましい場合が多い。実際、文献 2) の 9.1 節、文献 3) の 5.2 節で示されるように、コンパイラのような複雑な処理を行う場合、具象構文から構文木を作成することがしばしば行われる。

しかしながら、Yacc⁴⁾をはじめとした多くのコンパイラ・コンパイラは、解析木³⁾の構文要素に対してユーザが記述したアクションを起動する形式になっている。このため、構文木を出力するには、ユーザは構文木を表現するための構造体やクラスを別途記述し、アクションに構文木を構築するコードを手作業で記述しなければならない。しかし、解析木と構文木にはしばしば対応関係があり、構文木を表す構造体やクラス (抽象構文と呼ぶ) と解析木を表す文法 (具象構文とも呼ぶ)、それらを結び付けるアクションをを別々に定義するのは無駄が多い。多くの場合、BNF の左辺に現れる非終端記号は構文木のノードに対応し、そのノードの子は BNF の右辺に現れる終端・非終端記号に対応する。たとえば、JavaTM の while 文の具象構文は、BNF で次のようになる。終端記号をアンダーラインで表す。

WhileStatement ::= while (Expression)
Statement

一方、抽象構文における while 文は、Java のクラ

[†] 電気通信大学大学院電気通信学研究科情報工学専攻
Course in Computer Science and Information Mathematics, Graduate School of Electro-Communications, The University of Electro-Communications

^{††} 電気通信大学情報工学科
Department of Computer Science, The University of Electro-Communications

スで、次のように表すことができる。

```
class WhileStatement ... {
    Expression expression;
    Statement statement;
}
```

非終端記号 *WhileStatement* が、構文木のノードを表すクラス *WhileStatement* に対応し、インスタンス変数 *expression* と *statement* が、BNF の右辺の非終端記号 *Expression* と *Statement* に対応する。

このような無駄を省くために、具象構文と抽象構文を一体として記述し、それから具象構文と抽象構文を機械的に生成するコンパイラ・コンパイラがいくつか提案されている。しかしそれらは、2 章で述べるように、図 1 に示した算術式の具象構文と、図 2 に示した抽象構文を一体として記述することはできない。この抽象構文は、オブジェクト指向に従い、すべての演算を式 (*Expr*) の一種として表現し、*Term* や括弧などの中置記法特有の概念を捨象している。

本論文では、構文木を生成することに特化したパーザを出力するコンパイラ・コンパイラである $\neg<><UU$ (notavaCC) について報告する。 $\neg<><UU$ は、EBNF (BNF に対して、右辺で正規表現の記法を使用できるように拡張したもの) に基づく記法から、抽象構文と、構文木を出力するパーザとを自動的に生成する。 $\neg<><UU$ では、Yacc のようにアクションを書いたり、構文木を表現する構造体やクラスを別途書いたりする必要はなく、具象構文と抽象構文を一体として宣言的に記述できる。 $\neg<><UU$ は、上述の算術式のような、従来の手法が扱えなかった構文を扱うことができる。

$\neg<><UU$ は、パーザが出力する構文木の編集を許すので、 $\neg<><UU$ が扱えるような構造を多く含む構文木であれば、たとえそれが $\neg<><UU$ の能力を超えるような構造を部分的に含んでいても、 $\neg<><UU$ によって出力した構文木をユーザが記述したプログラムで書き換えることで対応できる。 $\neg<><UU$ は、曖昧でない任意の文脈自由文法を扱うことができる。生成されるのは、そのような言語を扱う Java プログラムである。

以下、2 章で既存の研究の特徴と問題点を指摘し、3 章で $\neg<><UU$ がどのようにしてその問題を解決するかを述べる。4 章ではいくつかの言語について性能を評価し、 $\neg<><UU$ が実用的であることを示す。5

```
Expr ::= Expr + Term | Term
Term ::= Term * Factor | Factor
Factor ::= NUMBER | ( Expr )
```

図 1 算術式を表す BNF

Fig.1 BNF for arithmetic expressions.

```
abstract class Expr { }
class Add extends Expr {
    Expr op1;
    Expr op2;
}
class Multiplication extends Expr {
    Expr op1;
    Expr op2;
}
class Literal extends Expr {
    String number;
}
```

図 2 算術式を表す抽象構文

Fig.2 Abstract syntax for arithmetic expressions.

```
Tree ::= Name ( ;
              | Tree
              | { Tree* } )
```

```
class Tree {
    Name name;
    Tree[] children;
}
```

図 3 木を表す具象構文 (上) とその抽象構文

Fig.3 Concrete and abstract syntax for trees.

章で本論文をまとめる。

2. 関連研究

具象構文と抽象構文の定義をそれぞれ別々に与えなければならないという煩雑さは、すでに広く認められている。実際、具象構文と抽象構文を一体として記述するコンパイラ・コンパイラとして、Java Tree Builder (JTB)⁵⁾、SableCC^{6),7)}、Front^{8),9)}、ANTLR^{10),11)}、JJTree¹²⁾ などが提案されている。また、抽象構文も定義できるように拡張された EBNF である WBNF¹³⁾ が提案されている。これらはみな、具象構文を記述する EBNF に対して若干の記法を追加して (あるいは追加せずに)、抽象構文を機械的に生成できるようにしたものである。これらのコンパイラ・コンパイラが生成するパーザは、その抽象構文に基づいた構文木を出力する。抽象構文は、パーザを記述するプログラム言語のクラスや構造体で表されるが、以下では、すべて Java のクラスで表現する。public などの修飾子は適宜省略する。

本章では、これらの手法を大きく 2 つに分類し、それらが図 2 の算術式をうまく扱えないことを指摘す

この奇妙な名前の由来は、 $\neg<><UU$ を公開している <http://ne.cs.uec.ac.jp/~koto/notavacc/> を見ればすぐに了解していただけるだろう。

る．また，子をブレースで囲って列挙し，子が 1 つだけのときはブレースを省略できる，図 3 のような木構造を表現する構文を表現できないことを指摘する．また，コンパイラ・コンパイラが扱うことのできる文法の種類に関する問題を指摘する．

2.1 解析木とその変形に基づく手法

2.1.1 JTB

テキストの構造を表すデータ構造を生成することを考えたときに，最も簡単な方法は，解析木を生成することである．プロダクション（EBNF における等式）に対して 1 つクラスを生成し，パーザが出力する木のノードはそのクラスのインスタンスで表される．JTB はこの方式をとっている．

この方式は，*Term* などの非終端記号ごとにクラスを生成するため，図 2 は表せない．また，JTB が図 3 の具象構文に対して出力するクラスは，ブレースが省略されない場合の子を，`{` を表すオブジェクト，子のツリーのリスト，`}` を表すオブジェクトの 3 つを含むリストで表現し，ブレースが省略された場合はその子単体で表現する．これは，図 3 の抽象構文とかけ離れている．

2.1.2 SableCC と Front

SableCC と Front も解析木を出力するが，ノードはプロダクションの右辺の選択肢ごとに異なるクラスで表される．これは，次のような，手続き型のプログラミング言語における文を表す場合に都合がよい．`<While>` などはクラス名を表す拡張であり，具象構文上の意味はない．

```
Stat ::= <While> while ( Expr ) Stat
      | <Print> print Expr*
      | <Assign> Identifier = Expr
      :
```

これから，次の抽象構文が得られる．

```
abstract class Stat { }
class While extends Stat {
    Expr expr;
    Stat stat;
    :
}
class Print extends Stat {
    Expr[] expr;
    :
}
:
```

SableCC や Front では，算術式は次のように記述される．

```
Expr ::= <Add> Expr ± Term
      | <Bridge1> Term
Term  ::= <Mul> Term * Factor
      | <Bridge2> Factor
Factor ::= <Literal> NUMBER
      | <Paren> ( Expr )
```

図 2 と比べて，Bridge1 などの不要なクラスが生成されてしまう．たとえば式の構成要素に，その値を計算するメソッドを追加する場合，このようなクラスでは，子の式から得られた値を返すだけのメソッドを実装しなければならない．また，メソッドを追加するたびに，このような処理を下請けに出すだけのメソッドを書かなければならない．

また，Front では，インスタンス変数は，EBNF の表現上の終端・非終端記号 1 つに対して 1 つ生成される．図 3 の具象構文に対して，次の出力が得られ，ブレースが省略されるかどうかに応じて，子を保持するインスタンス変数が変わる．

```
class Tree {
    Name name;
    Tree tree1;
    Tree[] tree2;
}
```

同様の問題がコンマで区切られた式

Expression (, *Expression*)*

にもあるが，Front は *A* で区切られた *B* を

B // *A*

と記述する記法を導入することで，*Expression* を表現上一度しか書かずに済ませることができる．これは汎用性に欠ける対応である．

一方，SableCC の EBNF は制限され，図 3 や上のカンマで区切られた式のような表現を受け付けない．

2.1.3 WBNF

WBNF では，算術式を次のように書くことができる．

```
Expr ::= Add | Term ||
Add  ::= Expr ± Term
Term ::= Mul | Factor ||
Mul  ::= Term * Factor
Factor ::= Literal | Paren ||
Literal ::= NUMBER
Paren  ::= ( Expr )
```

ここで，末尾に `||` がついたプロダクション，たとえば *Term* ::= ... は，構文木のノードを作る代わりに，右辺の記号 *Mul* と *Factor* を，左辺の終端記号 *Term* に「分類」する．分類は継承関係によって表現され，得られる抽象構文は次のようになる．

```

abstract class Expr { }
class Add extends Expr {
    Expr expr;
    Term term;
}
abstract class Term extends Expr { }
class Mul extends Term {
    Term term;
    Factor factor;
}
abstract class Factor extends Term { }
class Literal extends Factor {
    Number number;
}
class Paren extends Factor {
    Expr expr;
}

```

加算やリテラルが Expr のサブクラスである点は図 2 と同じだが, Paren などの不要なクラスが生成されている。

WBNF には, 中置記法のための特別な記法が用意されているが, それを用いた場合でも Paren が生成され, さらに Add や Mul を異なるクラスで表現することができなくなる。

また, WBNF では, SableCC と Front と同様, インスタンス変数は終端・非終端記号 1 つに対して 1 つ生成され, 図 3 を表せない。

2.2 操作的な手法

ANTLR と JJTree では, 構文木の構築は操作的である。ANTLR と JJTree の記法は, 本質的には構文木を構築するアクションを簡単に記述できるようにしたものである。これらは, 手続き型のプログラムで記述されたアクションを併記することができ, それによって構文木の構築を自由に制御することができる。

しかし, これらでは, 生成されるクラスが, 子に関する静的な情報を持たない。ANTLR と JJTree では, 子に応じて異なるインスタンス変数が生成されるのではなく, 子のリストを表すインスタンス変数が, すべてのノードの共通の親になるクラスに 1 つ用意されるだけである。このため, 図 2 や図 3 のようなインスタンス変数は生成できない。

ANTLR と JJTree では, ノードの子を得るには添え字を指定する必要があるが, ユーザはこれを即値で指定するか, 子を表現する識別子を手作業で定義するかなければならない。前者では, 文法を変更した場合にそれに合わせてユーザプログラムを変更するのが大変であり, また, 可読性が低下する。後者ではそのような識別子をいちいち定義するのが大変である。

2.3 文法の種類

コンパイラ・コンパイラが扱うことのできる文法の

種類は多いほどよい。たとえば, Java を模した次の文法を考える。ここで, *TypeName* と *VariableName* は, 文法上は同じ形をしているが, 意味的にはまったく異なる構文要素とする。

$$\begin{aligned}
 \textit{TypeName} &::= \underline{\textit{IDENT}} \\
 &\quad | \quad \textit{TypeName} _ \underline{\textit{IDENT}} \\
 \textit{VariableName} &::= \underline{\textit{IDENT}} \\
 &\quad | \quad \textit{VariableName} _ \underline{\textit{IDENT}} \\
 \textit{Expr} &::= \textit{TypeName} _ \underline{\textit{class}} \\
 &\quad | \quad \textit{VariableName} \\
 &\quad \vdots
 \end{aligned}$$

もしコンパイラ・コンパイラが LALR(1)¹⁴⁾ しか扱えないならば, この文法はたとえば次のように変形しなければならない。

$$\begin{aligned}
 \textit{Name} &::= \underline{\textit{IDENT}} \\
 &\quad | \quad \textit{Name} _ \underline{\textit{IDENT}} \\
 \textit{Expr} &::= \textit{Name} _ \underline{\textit{class}} \\
 &\quad | \quad \textit{Name} \\
 &\quad \vdots
 \end{aligned}$$

2.1 節で説明したような (そして $\neg \langle \rangle \langle \cup \cup \rangle$ が使用する), プロダクションやその右辺に対して抽象構文におけるクラスを一意に決定する手法では, 本来意味的に異なる *TypeName* と *VariableName* のような要素が, 文法の変更にもよって 1 つのクラスに集約されてしまう。このような問題を避けるために, コンパイラ・コンパイラが扱うことのできる文法の種類は多いほどよい。

また, Java 1.5 での導入が検討されている, Java に型パラメータを追加した言語¹⁵⁾ では, 次のような関数呼び出しが許される。

```

f(a < b, c, d > e)
f(a < b, c, d > . class)

```

前者は $a < b$ と c と $d > e$ の 3 つの式を引数にとるが, 後者は 1 引数の関数呼び出しであり, 引数は, 3 つの型 b, c, d を引数にとるパラメータ型 a を表すオブジェクトである。このような文法をたとえば LALR(1) で記述すると, 意味的な構造を反映させることは難しい。

SableCC は, LALR(1) を扱い, サポートする文法の種類は比較的少ない。Front は, Yacc のコードを生成するため LALR(1) を扱う。WBNF は, 文法の種類を規定していない。JTB, ANTLR, JJTree は, LL ベースのパーザを生成する。先読みする範囲を広くするなどによって多くの文法をサポートしているが, 左再帰を許さないという問題がある。

3. $\neg<><UU$ の扱う抽象構文

$\neg<><UU$ も、既存の研究と同様に、具象構文に対して若干の記法を追加することで、抽象構文を機械的に生成できるようにしたコンパイラ・コンパイラである。 $\neg<><UU$ は Java で記述されたパーザを生成し、そのパーザは構文解析の結果として構文木を生成する。 $\neg<><UU$ は、簡単なプリミティブを用いて、図 2 や図 3 のような抽象構文を生成することができ、曖昧でない任意の文脈自由文法を扱うことができる。

$\neg<><UU$ は EBNF をベースとする。 $\neg<><UU$ における主要な拡張は、継承関係の記述、ラベル付け、エイリアスの 3 つである。

3.1 継承関係の記述

$\neg<><UU$ にはプロダクションを表す 2 つの記法があるが、そのうちの 1 つは次のものである。

```
left-hand-side { right-hand-side }
```

この形式で定義されたプロダクションに対して、 $\neg<><UU$ はクラス（正確には Java のインタフェース）を 1 つ生成する。この形式のプロダクションだけを用いて文法を記述した場合、パーザが出力するのは解析木になる。

生成するクラスの間に継承関係を設定するのに、*left-hand-side* とブレースの間に次の記法を使う。

```
-> supertypes
```

ここで、*supertypes* は、このプロダクションが生成するクラスの親になるクラスの名前を&で区切って並べたものである。親クラスが指定されなかった場合、すべてのノードのクラスの継承関係の始祖になるクラス Node を継承する。たとえば、次の $\neg<><UU$ の構文記述

```
A { ... }
B -> A { ... }
X -> Y & Z { ... }
```

は、次を生成する。

```
interface A extends Node { ... }
interface B extends A { ... }
interface X extends Y, Z { ... }
```

3.2 ラベル付けによるメソッド生成の制御

3.2.1 ラベル付け

$\neg<><UU$ では、ノードの子にアクセスするために、インスタンス変数ではなくメソッドを用いる。また、メソッドを生成するのに、ラベル付けを用いる。ラベル付けは、生成するメソッドの名前を指定するとともに、どの終端・非終端記号に対してメソッドを生成するかを示す。

たとえば、 $\neg<><UU$ では、while 文を表すプロダクションを次のように書く。

```
WhileStatement {
    "while" "(" expression:Expression ")"
    statement:Statement
}
```

expression と statement はそれぞれ Expression と Statement に対するラベルである。

上のプロダクションに対して生成されるクラスは、

```
interface WhileStatement extends Node {
    Expression expression();
    Statement statement();
};
```

となる。

3.2.2 複数の記号に対する同一のラベル

同名のラベルが構文記述の中に複数存在できる。

```
Tree {
    name:Name ( ";"
    | children:Tree
    | "{" children:Tree* "}" )
}
```

と書けば、図 3 の抽象構文が得られる。children は、ブレースが省略されているかどうかにかかわらず、その子のリストを返す。

ラベルは、終端記号・非終端記号だけでなく、任意の EBNF の式をラベル付けできる。これは、ラベル付けされている EBNF の式に含まれる、すべての終端記号・非終端記号を個別にラベル付けするのと同じである。たとえば、

```
x:( A | B ( C D )+ )
```

は、

```
( x:A | x:B ( x:C x:D )+ )
```

を意味する。

3.2.3 単一の記号に対する複数のラベル

1 つの終端・非終端記号に、複数のラベルが付いてもよい。たとえば、次は、 $\neg<><UU$ における字句定義に使われる構文記述の一部である。

```
CharacterRange {
    lower:CharacterLiteral
    ".."
    upper:CharacterLiteral
}
$private Character -> CharacterRange {
    lower:upper:CharacterLiteral
}
```

CharacterRange は文字コードの範囲を表すための構文要素で、2 つの文字リテラルを 2 つのピリオドで区切ったものである。上限と下限が同じ場合、略記として、その文字コードを表す文字リテラルを 1 つだけ

を書く方法も用意されており、それが `Character` である。`$private` は、生成されるクラスをパーザ以外からアクセスできないようにし、次のクラスが生成される。

```
interface CharacterRange extends Node {
    CharacterLiteral lower();
    CharacterLiteral upper();
}
private interface Character
    extends CharacterRange {
    CharacterLiteral lower();
    CharacterLiteral upper();
}
```

抽象構文において、`Character` は `CharacterRange` の一種であり、`lower()` と `upper()` は同じ値を返す。`Character` は `CharacterRange` として扱うことができ、`lower()` で下限を、`upper()` で上限を得ることができる。

3.2.4 メソッドの戻り値の型

メソッドの戻り値の静的な型は、ラベル付けされるインスタンスがたかだか 1 つの場合、そのメソッドが返す値を表現できる型の中で最も具体的なものである。ここで継承関係における子を、親よりも具体的とする。複数回出現しうる場合、戻り値の型はリストであり、その要素の型は、リストの要素になりうるオブジェクトを表現できる型のうち、最も具体的なものである。たとえば

```
A { ... }
B -> A { ... }
C -> A { ... }
D -> B & C { ... }
X { label1:( B | C ) }
Y { label2:( C D ) }
```

と書くと、`X`、`Y` に対して、次のクラスが生成される。

```
interface X extends Node {
    A label1();
}
interface Y extends Node {
    List<C> label2();
}
```

ここで、`List<C>` は Java 1.5 での導入が検討されている記法¹⁵⁾で、`C` のリストを表す。`→<><UU` は、Java 1.4 以前のために、これを型パラメータのない `List` で置き換えて生成することもできる。

`label1` では、構文解析されるテキストに `B` か `C` のいずれか 1 つだけが出現するので、戻り値はリストではない。一方 `label2` では、構文解析されるテキストに `C` と `D` の 2 つが出現するので、戻り値はリストである。`label1` の戻り値は、`B` と `C` の両方を表現できる型でなければならないので、`Node` か `A` のいずれか

になるが、`A` がより具体的である。`label2` のリストの要素は、`C` と `D` の両方を表現できる型でなければならないので、`Node`、`A`、`C` のいずれかとなり、`C` が最も具体的である。

後述のエイリアスを用いた場合、メソッドの戻り値を決定するアルゴリズムは自明ではない。メソッドの戻り値を決定するアルゴリズムについては、付録で説明する。

3.3 エイリアス

`→<><UU` のプロダクションの記法の 2 つめは、次のものである。

left-hand-side = *right-hand-side* ;

この形で定義される非終端記号をエイリアスと呼ぶ。エイリアスはクラスを生成せず、構文木のノードにならない。エイリアスを用いることで、最終的には図 2 のような算術式の抽象構文を生成することができるが、ここでは簡単な利用法から順に説明していく。

3.3.1 マクロ

多くの言語で、関数の引数をはじめとして、多くの箇所で式をカンマで区切ったリストが構文要素として現れる。しかし、式のリストは、抽象構文のノードとして表すよりは、リストを返すメソッドとして各クラスに存在したほうが良い。エイリアスは、このようなメソッドを実現するマクロとして扱うことができる。

```
expressionList = expressions:Expression
    ( "," expressions:Expression ) * ;
```

プロダクションの右辺に `expressionList` が現れた場合、あたかもその場所に、上のプロダクションの右辺に示される EBNF の式が記述されているかのように扱われ、式のリストを返すメソッド `expressions` が生成される。話を簡単にするために、まだエイリアスの自己埋め込みは考えないことにする。

これを用いると、`Paren` は含むが `Term` などには含まない抽象構文を次のように表現できる。

```
Example      { expression:Expr }
expr          = Add | term ;
Add -> Expr { op1:Expr "+" op2:term }
term          = Mul | factor ;
Mul -> Expr { op1:term "*" op2:factor }
factor        = Literal | Paren ;
Literal -> Expr { number:NUMBER }
Paren -> Expr { "(" expr:Expr ")" }
$abstract Expr { }
```

`$abstract` は予約語であり、`Expr` がクラスを生成するが、具象構文上利用されてはいけないことを示す。`$abstract` をともなって定義された非終端記号は、`$abstract` ではないプロダクションの右辺で使用することはできない。

エイリアスを展開すると次のようになる．

```
Example {
  expression:
    ( Add | Mul | Literal | Paren )
}
Add -> Expr {
  op1:( Add | Mul | Literal | Paren )
  "+"
  op2:( Mul | Literal | Paren )
}
Mul -> Expr {
  op1:( Mul | Literal | Paren )
  "*"
  op2:( Literal | Paren )
}
Literal -> Expr {
  number:NUMBER
}
Paren -> Expr {
  "("
  expr:( Add | Mul | Literal | Paren )
  ")"
}
$abstract Expr { }
```

これは確かに算術式の文法を表しており，これから次のクラスが生成される．

```
interface Example extends Node {
  Expr expression();
}
interface Add extends Expr {
  Expr op1();
  Expr op2();
}
interface Mul extends Expr {
  Expr op1();
  Expr op2();
}
interface Literal extends Expr {
  Token number();
}
interface Paren {
  Expr expression();
}
```

Token はトークンを表す．

3.3.2 引数付きのマクロ

式のリストを表すメソッドの名前を，状況によって変えたい場合もある．たとえば，Java の for 文は，初期化のための式のリストと，変数を更新するための式のリストをパラメータにとることができるが，これらは別の 2 つのメソッドで表現されるべきである．この問題を，ラベルを引数にとるエイリアスを用いることで解決する．引数はそのエイリアスに対するラベル付けとして表し，仮引数は予約語 \$label で表す．式のリストを次のように表せば，initializers:expressionList と書くことで，メソッド initializers がこの式のリ

ストを返すようになる．

```
expressionList = $label:Expression
( "," $label:Expression ) * ;
```

3.3.1 項の利用法との整合性をとるために，右辺に \$label が含まれていなければ，引数のラベルは右辺に含まれるすべての終端・非終端記号をラベル付けすることにする．

3.3.3 自己埋め込みと算術式

エイリアスを用いて，Paren も Term も含まない抽象構文を表現することを考える．Paren を引数付きのエイリアスで表現する．

```
Example { expression:expr }
expr = Add | term ;
Add -> Expr { op1:expr "+" op2:term }
term = Mul | factor ;
Mul -> Expr { op1:term "*" op2:factor }
factor = Literal | paren ;
Literal -> Expr { number:NUMBER }
paren = "(" $label:expr ")" ;
$abstract Expr { }
```

エイリアスを展開すると，自己埋め込みがあるため，プログラクシオンの右辺は無限に長くなり，たとえば Example は次のようになる．

```
Example { expression:Add
| expression:Mul
| expression:Literal
| "(" expression:Add ")"
| "(" expression:Mul ")"
| "(" expression:Literal ")"
| "(" "(" expression:Add ")" ")"
| "(" "(" expression:Mul ")" ")"
| "(" "(" expression:Literal ")" ")"
:
:
}
```

しかし，これらは，直観的には正しく算術式を表している．

→<><UU は，これを生かすために，エイリアスを，コンパイル時に展開されるシンタックスシュガーではなく，構文解析時にノードの削除を行う特殊な非終端記号として扱う．エイリアスの仕様は次のとおりであり，これまで説明したマクロとしての使用法と整合性を持っている．

→<><UU では，パーザはまず，入力されたテキストから解析木を作る．この段階では，エイリアスとそうでない非終端記号の区別はなく，エイリアスもノードを作る．たとえば， $1 * (2 + 3)$ の解析木は，図 4 のようになる．その後，エイリアスによるノードを取り除く．取り除かれたノードの子は，取り除かれたノードの親の子になる（図 5）．取り除かれたノードや，その子に対するラベル付けは，次のように解決する．ま

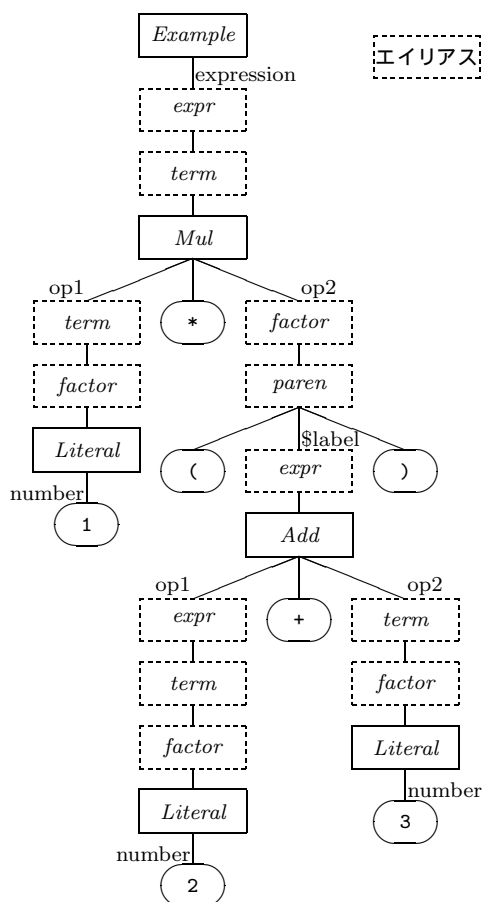


図 4 エイリアスを用いた場合の $1 * (2 + 3)$ の解析木
Fig. 4 Concrete syntax tree of $1 * (2 + 3)$ with aliases.

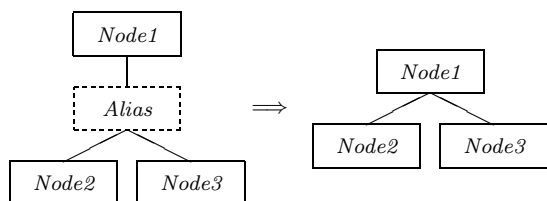


図 5 ノードの取り除き
Fig. 5 Alias removal.

ず、子に対するラベル付けは、それが \$label でない限り、そのまま存続する (図 6)。取り除かれるノードに対するラベル付けは、子に対するラベル付けに \$label が含まれる場合、その \$label を持つ子だけに引き継がれ、\$label は削除される (図 7)。そうでない場合すべての子に引き継がれる (図 8)。エイリアスを取り除くことによって、図 4 から図 9 が得られる。エイリアスを取り除く順番は、結果に影響を与えない。

→<><UUは、このような変換によって作られる構文木を表現できるように、クラスを生成する。得られ

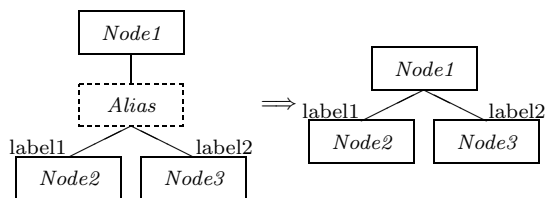


図 6 子に対するラベル付け
Fig. 6 Labels for children.

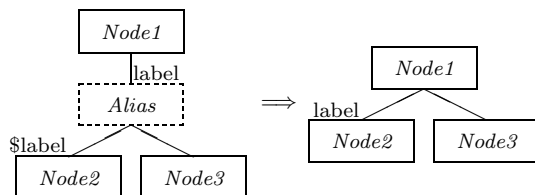


図 7 \$label によるラベルの引継ぎ
Fig. 7 Label inheritance with \$label.

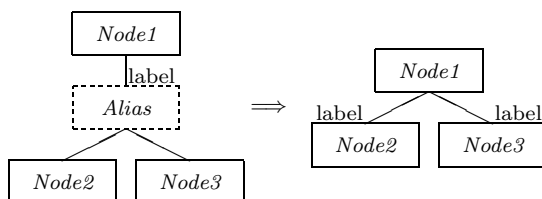


図 8 \$label を用いないラベルの引継ぎ
Fig. 8 Label inheritance without \$label.

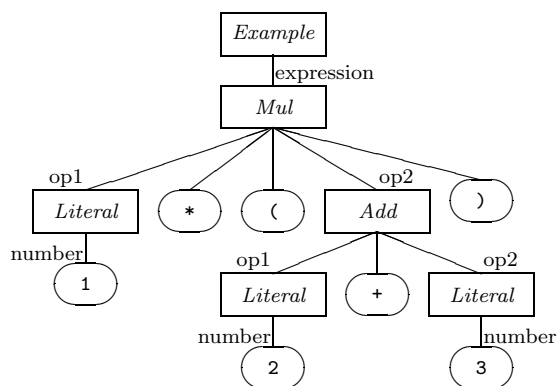


図 9 $1 * (2 + 3)$ の構文木
Fig. 9 Abstract syntax tree of $1 * (2 + 3)$.

るクラスは、次のようになり、これは図 2 を表現している。

```
interface Example extends Node {
    Expr expression();
}
interface Add extends Expr {
    Expr op1();
    Expr op2();
}
```

```

interface Mul extends Expr {
    Expr op1();
    Expr op2();
}
interface Literal extends Expr {
    Token number();
}
interface Expr extends Node {
}

```

3.4 トラバース

→<><UUでは、1つの終端・非終端記号に複数のラベルが付いてもよい。トラバースにおいて、各メソッドから得られる子を1度ずつ訪問すると、複数のラベル付けを持つインスタンスを2度以上訪問してしまう。→<><UUでは、そのノードの子すべてのリストを返すメソッド getChildNodes を、クラス Node に用意することで、この問題を解決する。このメソッドが返す子を1度ずつ訪問すれば、すべてのノードを1度ずつ訪問することができる。

getChildNodes は、どのラベルからもラベル付けされていない子も含む。これには、コメントやホワイトスペースなどの、構文上は無視されるトークンも含まれる。このため、構文木をトラバースしながらすべてのトークンを拾っていくことで、パーザへ入力されたテキストを復元することができる。この機能は、ソースプログラムの一部を機械的に変形するような処理や、ソースプログラムを色やリンクのついた HTML に変換するといった処理において、有用である。→<><UUでは、メモリの消費量を抑えるために、ラベル付けされていない子を構文木から取り除くこともできる。

3.5 構文解析アルゴリズムと曖昧性

→<><UUでは「ツリー構造化スタックを用いた一般化 LR 構文解析アルゴリズム」¹⁶⁾を使用する。これは、無限個の解析木を作るような曖昧性を持たない、任意の文脈自由文法を扱うことのできる構文解析アルゴリズムである。このアルゴリズムは、曖昧な文法に基づいて構文解析を行った場合、1つの入力に対して、その入力を表現可能なすべての構文木を求める。しかし、そのような構文木を効率良く表現するためには、抽象構文に特別な拡張をしなければならない。構文木の数は入力の長さに応じて指数関数的に増加するので、構文木の一部を共有するといった対応が必要になるからである。→<><UUは、計算機で処理することを前提に設計された言語を第1の対象としており、そのような言語はたいてい曖昧ではない。このため、→<><UUは曖昧な文法は扱わないことにした。構文解析時に複数の構文木が生成できる場合、実行時エラーが発生する。

このアルゴリズムは、複数の LR 構文解析器³⁾を並列的に実行するアルゴリズムであり、次のように要約できる。

- 初期状態では、LR 構文解析器が1つだけ用意される。
- トークンを1つ受け取り、すべての LR 構文解析器へ入力することを繰り返す。ただし、その入力が競合³⁾をもたらす場合、LR 構文解析器のコピーをいくつか作成し、競合しているシフト動作や還元動作をそれぞれに割り当て実行させる。また、入力をエラーと見なす LR 構文解析器は捨てる。
- 最後に LR 構文解析器が1つ残った場合、その解析器が正しい構文解析結果を与える。1つも残らなかった場合は入力が文法に従っていない。2つ以上残った場合は、文法が曖昧であり、異なる複数の解析木が構築可能であることを意味する。
- LR 構文解析器のコピーにかかる時間は $O(1)$ である。

→<><UUでは、LR 表の作成には LALR(1) を用いたため、LR 表のサイズは LALR(1) の LR 表と同じである。構文解析の計算量は、入力のテキストの長さ n に対して、最悪の場合指数関数的に増加するが、文法が LALR(1) 文法の場合、若干のオーバーヘッドを除いて LALR(1) 構文解析アルゴリズムと等しい ($O(n)$ であり、係数も近い)。また、並列的に実行される解析器の個数が定数個以下のような文法に対しても $O(n)$ である。文献 17) の 2~15 章で定義される Java の文法がこれにあたる。

また、計算機で処理することを前提に設計された言語には、ほとんどの場合、LR 構文解析器の数を1つ（あるいは定数以下の個数）に絞るような地点（たとえば、手続き型のプログラム言語における文の終端）が頻繁に出現する。それらの地点の間に含まれるトークンの数は、多くの場合、テキストの長さにかかわらず定数以下であることが期待できる。このような場合、並列的に実行される解析器の個数は定数個以下になり、計算量は $O(n)$ になる。

4. 性能評価

Java 2 SDK Standard Edition 1.4.1 (Sun Microsystems) を使い、Pentium III 933 MHz、メモリ 512 MB、Linux の環境において、性能を測定した。以下、1 MB = 1024 KB = 1024² B である。次の理由から、他のコンパイラ・コンパイラとの比較は行わなかった。JTB、ANTLR、JJTree は左再帰を許さない

表 1 コンパイラ・コンパイラの性能

Table 1 Performance of the compiler compiler.

言語	算術式 * ¹	木構造記述 * ²	「<><UU」	Java * ³	JSR-014 * ⁴
プロダクションの数 (EBNF)	10	9	62	196	220
プロダクションの数 (CFG) * ⁵	10	17	114	361	422
LR 表の状態数	24	34	202	1,218	1,366
コンパイル時間 * ⁶ [秒]	2.4	4.3	6.0	20.8	23.1
最小ヒープサイズ * ⁷ [MB]	1	7	9	41	43
バイナリ サイズ * ⁸ [バイト]	51,949	60,745	206,276	930,202	1,010,280

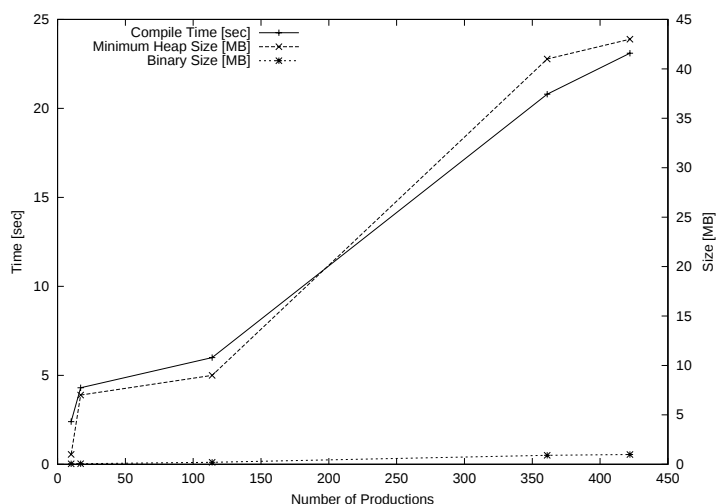
*¹ 四則演算の式を表す言語*² 図 4~9 のような木構造を表現する言語：子の位置を、親からの相対座標で指定する。*³ 文献 17) の 2~15 章において定義されている文法。*⁴ 文献 15) で定義される、Java に List<C>のような型パラメタのサポートを追加したもの。*⁵ EBNF を文脈自由文法に変換した後のプロダクションの数。*⁶ 「<><UU」ソースから Java ファイルを生成するのにかかる、Java VM のブートから終了までの時間。ヒープのサイズは-Xmx64m オプションにより 64 MB に制限した。*⁷ OutOfMemoryError を発生させずにコンパイル可能な-Xmx オプションの最小値 (1 メガバイト単位切り上げ)。*⁸ 生成された Java ファイルをコンパイルして得られるクラスファイルのサイズの総和。

図 10 プロダクション数と使用リソースの関係

Fig. 10 Relation between the number of productions and the resource consumption.

ため、また、SableCC は制限された EBNF を用いるため、測定に使用した文法を大幅に変更しなければならない。Front は LALR(1) を扱うため同様であり、また Yacc のコードを出力するため Java のプログラムを出力する「<><UU」との比較は意味がない。WBNF は、処理系を手でできなかった。

4.1 コンパイラ・コンパイラの性能評価

いくつかの言語の文法を記述し、「<><UU」でコンパイルを行い、性能を測定した (表 1)。文法の大きさの目安として、プロダクションの数と LR 表の状態数を示した。それぞれの言語について、コンパイルに必要な時間とメモリ量を測定した。また、生成されるデータのサイズとして、クラスファイルのサイズを測定した。

文脈自由文法に変換した後のプロダクションの数と、消費されるリソースの関係を図 10 に示した。おおむね線形に増加しており、スケーラビリティがあると考えられる。また、リソースの消費量は十分実用的である。

4.2 生成されたパーザの性能評価

Java 言語仕様¹⁷⁾ の 2~15 章で定義される文法を扱うパーザを用いて、Java 2 SDK Standard Edition 1.4.1 に付属の、ライブラリのソースファイル (src.jar に含まれる拡張子が java のファイル) を構文解析し、構文解析にかかる時間を測定した。ただし、java/util/logging/LogManager.java はこの文法に従っていなかったため、除外した。ヒープのサイズは-Xmx64m オプションにより 64 MB に制限した。

表 2 構文解析にかかる時間

Table 2 Time for parsing.

	全体	ファイルあたり
解析対象のファイルの数	3,887	-
解析対象のサイズ [KB]	38,751	10
Java [秒]	246.5	0.063
JSR-014 [秒]	255.0	0.066

表 3 構文木の表現に必要なメモリ量

Table 3 Memory size for syntax trees.

	全体
構文解析対象のファイルの数	119
構文解析対象のファイルのサイズ [KB]	2,002
Java * ⁹ [KB]	29,696
JSR-014 * ⁹ [KB]	29,696

*⁹ 1 メガバイト単位切り上げ

出力された構文木は保持せず、ガーベジコレクションの対象とさせた。JavaVM のブート後すべてのファイルを 1 度ずつ構文解析するプログラムと、JavaVM のブート後すべてのファイルを 1 度ずつ構文解析し、その後すべてのファイルをもう 1 度ずつ構文解析するプログラムとを用意し、それらの実行時間の差を、構文解析にかかる時間とした。また、これと同様のことを、Java ではなく Java に型パラメータを追加した文法 (JSR-014¹⁵) についても行った。結果は表 2 のようになった。構文解析にかかる時間は 1 ファイルあたり 60~70 ミリ秒と、十分実用的である。

また、src.jar の java/util/ディレクトリ以下に含まれる、LogManager.java 以外のファイルを構文解析し、得られるすべての構文木を保持できる -Xmx オプションの最小値を測定した。ホワイトスペースや、コメントなども保持した。結果は表 3 のようになった。構文木を保持するのに必要なメモリ量は、入力されるテキストとの比は大きいものの、十分実用的である。

5. ま と め

構文木を出力するパーザを生成するコンパイラ・コンパイラ- $\langle \rangle \langle \cup \cup \rangle$ について述べた。 $\langle \rangle \langle \cup \cup \rangle$ は、具象構文と抽象構文を一体として記述でき、構文木を生成するパーザを簡単に素早く作ることができる。 $\langle \rangle \langle \cup \cup \rangle$ は、継承、ラベル付け、エイリアスの構文記法を用い、従来の研究では表現できなかった算術式のような抽象構文を、オブジェクト指向に基づいて記述することができ、これまでのコンパイラ・コンパイラに見られない高い記述性を得ることができた。いくつかの言語における性能を測定し、 $\langle \rangle \langle \cup \cup \rangle$ は実用的な性能を持つことを確認した。

参 考 文 献

- 1) Meyer, B (著), 二木厚吉 (監訳), 酒匂 寛 (訳): プログラミング言語理論への招待, アスキー (1995).
- 2) 中田育男: コンパイラの構成と最適化, 朝倉書店 (1999).
- 3) エイホ, A.V., セシィ, R., ウルマン, J.D.: コンパイラ I 原理・技法・ツール, サイエンス社 (1990).
- 4) Johnson, S.C.: Yacc - Yet Another Compiler Compiler, Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J. (1975).
- 5) Wang, W., Tao, K. and Palsberg, J.: Java Tree Builder. <http://www.cs.purdue.edu/jtb/>
- 6) Gagnon, E. and Hendren, L.: SableCC - an object-oriented compiler framework, Proc. Technology of Object-Oriented Languages Tools 26: August 3-7, 1998 Santa Barbara, California (1998).
- 7) Gagnon, E.: SableCC, an object-oriented compiler framework, Master's thesis, School of Computer Science, McGill University, Montreal (1998).
- 8) Philips Research: Front. <http://www.research.philips.com/InformationCenter/Global/FArticleDetail.asp?lArticleId=2669&lNodeId>
- 9) Augusteijn, L.: *Front: a front-end generator for Lex, Yacc and C*, release 1.0, PHILIPS Research, Information and Software Technology.
- 10) Parr, T.J. and Quong, R.W.: ANTLR: A Predicated-LL(k) Parser Generator, *Software-Practice and Experience*, Vol.25, No.7, pp.789-810 (1995).
- 11) Parr, T. et al.: *ANTLR Reference Manual* (2000).
- 12) Sun Microsystems: JJTree. http://www.webgain.com/products/java_cc/jjtree.html
- 13) Wile, D.S.: Abstract syntax from concrete syntax, *Proc. 19th international conference on Software engineering*, pp.472-480, ACM Press (1997).
- 14) DeRemer, F.: Practical translators for LR(k) Languages, Ph. D. Thesis, M.I.T. (1969).
- 15) Bracha, G., Cohen, N., Kemper, C., Marx, S., Odersky, M., Panitz, S.-E., Stoutamire, D., Thorup, K. and Wadler, P.: Adding Generics to the Java Programming Language: Participant Draft Specification (2001).
- 16) Tomita, M.: An Efficient Augmented-Context-Free Parsing Algorithm, *Computational Linguistics*, Vol.13, No.1-2, pp.31-46 (1987).
- 17) Gosling, J., Joy, B., Steele, G. and Bracha, G.:

The Java Language Specification, 2nd edition,
Addison-Wesley Pub. Co. (2000).

付 録

A.1 メソッドの戻り値の判定

この章では、メソッドの戻り値の型を決定するアルゴリズムについて説明する。その正当性は、A.2 で示す。

メソッドの戻り値の型を決定するにあたり、求めなければならないことは、そのメソッドが2つ以上の子を返すかどうかと、そのメソッドがどのような終端・非終端記号を返すかどうかである。後者を次の関数 C によって求め、前者を関数 $\#$ によって求める。ここでは、ラベルの集合を L 、終端記号の集合を T 、エイリアスの集合を A 、エイリアスではない非終端記号の集合を Y とする。 $Y' = Y \cup \{\text{Token}\}$ とする。 Y は、クラスを生成する非終端記号の集合であり、 Y' は構文木に出現するノードのクラスすべての集合である。 $\neg \langle \rangle \langle \rangle \cup \cup$ では、すべてのトークンはクラス Token で表す。 $B(n)$ は、非終端記号 n を定義するプロダクションの右辺、 ϕ は空集合、 λ は長さ0の列とする。関係は、デカルト積の部分集合であり、2項関係 R に対して、 $R(x, \bullet)$ を次で定義する。

$$R(x, \bullet) = \{y \mid \langle x, y \rangle \in R\}$$

A.1.1 関数 C

C の定義域は Y であり、戻り値は L と Y' の間の2項関係である。 $C(y)(l, \bullet)$ は、クラス y のメソッド l が返す型の集合を表す。メソッド l の戻り値の型は、 $C(y)(l, \bullet)$ に含まれる型すべてを表すことのできる最も具体的な型か、その型のリストである。

C を、次のような、作業メモリを表す補助的な引数を使った再帰関数 C' で定義する。

$$C(y) = C'(B(y), \phi, \phi, \phi)$$

C' の第1~4引数はそれぞれ、現在調べているEBNF、それに対するラベルの集合、EBNF に $\$label$ が含まれていた場合のエイリアス引数、探索を無限ループに陥らせないためのスタックである。

C' の定義は、次のようになる。

- 選択 $e = e_1 | e_2 | \dots | e_n$ 、あるいは、接続 $e = e_1 e_2 \dots e_n$ に対して、

$$C'(e, M, M_0, S) = \bigcup_{i=1..n} C'(e_i, M, M_0, S)$$

特に、接続 $e = \lambda$ に対して、

$$C'(e, M, M_0, S) = \phi$$

- クリーネ閉包 $e = e_0^*$ 、正閉包 $e = e_0^+$ 、あるいは省略 $e = e_0?$ に対して、

$$C'(e, M, M_0, S) = C'(e_0, M, M_0, S)$$

- ラベル付けされた式 $e = label : e_0$ に対して、 $label$ が $\$label$ なら、

$$C'(e, M, M_0, S) = C'(e_0, M \cup M_0, \phi, S)$$

さもなくば、

$$C'(e, M, M_0, S) = C'(e_0, M \cup \{label\}, M_0, S)$$

- 終端記号 t に対して、

$$C'(t, M, M_0, S) = \{\langle l, \text{Token} \rangle \mid l \in M\}$$

- エイリアスではない非終端記号 $y \in Y$ に対して、

$$C'(y, M, M_0, S) = \{\langle l, y \rangle \mid l \in M\}$$

- エイリアス $a \in A$ に対して、 $\langle M, a \rangle \in S$ ならば、

$$C'(a, M, M_0, S) = \phi$$

そうでなくて、 $B(a)$ に $\$label$ が存在すれば、

$$C'(a, M, M_0, S) = C'(B(a), \phi, M, S \cup \langle M, a \rangle)$$

さもなくば、

$$C'(a, M, M_0, S) = C'(B(a), M, \phi, S \cup \langle M, a \rangle)$$

C' が次の構文記述からどのようにラベル付けを抽出するかを示す。

A { x: (B c) }

c = \$label: D y: c? ;

$y: c?$ は省略可能な $y: c$ を表す。大文字はすべて非終端記号とする。 $l \in L, a \in A$ に対して、 $\langle \{l\}, a \rangle$ を、 $l: a$ と書くことにする。 $C(A)$ は次のように計算される。

$$C(A)$$

$$= C'(x: (B c), \phi, \phi, \phi)$$

$$= C'((B c), \{x\}, \phi, \phi)$$

$$= C'(B, \{x\}, \phi, \phi) \cup C'(c, \{x\}, \phi, \phi)$$

ここで、

$$C'(B, \{x\}, \phi, \phi) = \{\langle x, B \rangle\}$$

一方、

$$C'(c, \{x\}, \phi, \phi)$$

$$= C'(\$label: D y: c?, \phi, \{x\}, \{x: c\})$$

$$= C'(\$label: D, \phi, \{x\}, \{x: c\})$$

$$\cup C'(y: c?, \phi, \{x\}, \{x: c\})$$

ここで、

$$C'(\$label: D, \phi, \{x\}, \{x: c\})$$

$$= C'(D, \{x\}, \phi, \{x: c\})$$

$$= \{\langle x, D \rangle\}$$

一方、

$$C'(y: c?, \phi, \{x\}, \{x: c\})$$

$$= C'(y: c, \phi, \{x\}, \{x: c\})$$

$$= C'(c, \{y\}, \{x\}, \{x: c\})$$

$$= C'(\$label: D y: c?, \phi, \{y\}, \{x: c, y: c\})$$

$$= C'(\$label: D, \phi, \{y\}, \{x: c, y: c\})$$

$$\cup C'(y: c?, \phi, \{y\}, \{x: c, y: c\})$$

ここで、

$$\begin{aligned}
& C'(\$label:D, \phi, \{y\}, \{x:c, y:c\}) \\
&= C'(D, \{y\}, \phi, \{x:c, y:c\}) \\
&= \{\langle y, D \rangle\}
\end{aligned}$$

一方,

$$\begin{aligned}
& C'(y:c?, \phi, \{y\}, \{x:c, y:c\}) \\
&= C'(y:c, \phi, \{y\}, \{x:c, y:c\}) \\
&= C'(c, \{y\}, \{y\}, \{x:c, y:c\}) \\
&= \phi
\end{aligned}$$

以上から,

$$\begin{aligned}
& C(A) \\
&= \{\langle x, B \rangle\} \cup \{\langle x, D \rangle\} \cup \{\langle y, D \rangle\} \cup \phi \\
&= \{\langle x, B \rangle, \langle x, D \rangle, \langle y, D \rangle\}
\end{aligned}$$

A.1.2 関数

の定義域は Y であり, 戻り値は L から集合 $F = \{0, 1, 2^+\}$ への写像である. この写像は, ラベルからそのラベルに対応する子の最大個数への関数を表す (2^+ は 2 以上を表す). 非終端記号 $y \in Y$ とラベル $l \in L$ に対して, $\#(y)(l) = 0$ ならば, クラス y はメソッド l を持たない. $\#(y)(l) = 1$ ならば, クラス y はメソッド l を持ち, 戻り値はリストではない. $\#(y)(l) = 2^+$ ならば, クラス y はメソッド l を持ち, 戻り値はリストである.

も C と同様に, 作業メモリを表す補助的な引数を使った再帰関数 $\#'$ で定義する.

$$\#(y) = \#'(B(y), \phi, \phi, \lambda)$$

#' の第 1~4 引数は C' と同様に, 現在調べている EBNF, それに対するラベルの集合, EBNF に \$label が含まれていた場合のエイリアス引数, 探索を無限ループに陥らせないためのスタックである. 第 4 引数は C' とは異なり, 集合ではなく列である. #' では, 探索を打ち切る深さは C' より 1 回深く, 同じ引数とエイリアスのペアが, 第 4 引数に 2 回までは含まれてよい.

#' を次のように定義する. ただし, 関数 $\max$ は, 大小関係 $0 < 1 < 2^+$ に基づいて最大のを返す. F 上の加算 $\oplus$ は $0 \oplus x = x$, $1 \oplus 1 = 2^+$, $2^+ \oplus x = 2^+$, $x \oplus y = y \oplus x$ で定義され, 乗算 $\otimes$ は $0 \otimes x = 0$, $1 \otimes x = x$, $2^+ \otimes 2^+ = 2^+$, $x \otimes y = y \otimes x$ で定義する.

- 選択 $e = e_1 | e_2 | \dots | e_n$ に対して,

$$\begin{aligned}
\#'(e, M, M_0, S) &= \\
&\{\langle l, f \rangle \mid f = \max_{i=1..n} \#'(e_i, M, M_0, S)(l)\}
\end{aligned}$$

- 接続 $e = e_1 e_2 \dots e_n$ に対して,

$$\begin{aligned}
\#'(e, M, M_0, S) &= \\
\{\langle l, f \rangle \mid f &= \bigoplus_{i=1..n} \#'(e_i, M, M_0, S)(l)\}
\end{aligned}$$

特に, $e = \lambda$ のとき,

$$\#'(e, M, M_0, S) = \{\langle l, 0 \rangle \mid l \in L\}$$

- クリーネ閉包 $e = e_0^*$, あるいは正閉包 $e = e_0^+$ に対して,

$$\begin{aligned}
\#'(e, M, M_0, S) &= \\
&\{\langle l, f \rangle \mid f = 2^+ \otimes \#'(e_0, M, M_0, S)(l)\}
\end{aligned}$$

- 省略 $e = e_0?$ に対して,

$$\#'(e, M, M_0, S) = \#'(e_0, M, M_0, S)$$

- ラベル付けされた式 $e = label:e_0$ に対して, $label$ が \$label なら,

$$\#'(e, M, M_0, S) = \#'(e_0, M \cup M_0, \phi, S)$$

さもなくば,

$$\#'(e, M, M_0, S) = \#'(e_0, M \cup \{label\}, M_0, S)$$

- エイリアスではない終端記号 $e \in T$, あるいは非終端記号 $e \in Y$ に対して,

$$\#'(e, M, M_0, S) = (M \times \{1\}) \cup ((L - M) \times \{0\})$$

- エイリアス $a \in A$ に対して, $\langle M, a \rangle$ が S に 2 回以上含まれていれば,

$$\#'(a, M, M_0, S) = \{\langle l, 0 \rangle \mid l \in L\}$$

そうでなくて, $B(a)$ に \$label が存在すれば,

$$\#'(a, M, M_0, S) = \#'(B(a), \phi, M, S \circ \langle M, a \rangle)$$

さもなくば,

$$\#'(a, M, M_0, S) = \#'(B(a), M, \phi, S \circ \langle M, a \rangle)$$

ただし, $S \circ \langle M, a \rangle$ は, 列 S の末尾に $\langle M, a \rangle$ をつなげてできる列である.

再び, 次の例を用いて, # の計算の過程を示す.

```

A { x:(B c) }
c = $label:D y:c? ;

```

ここでは, $\#(A)(y)$ を考える.

$$\begin{aligned}
\#(A)(y) &= \#'(x:(B c), \phi, \phi, \lambda)(y) \\
&= \#'((B c), \{x\}, \phi, \lambda)(y) \\
&= \#'(B, \{x\}, \phi, \lambda)(y) \oplus \#'(c, \{x\}, \phi, \lambda)(y)
\end{aligned}$$

ここで,

$$\#'(B, \{x\}, \phi, \lambda)(y) = 0$$

一方,

$$\begin{aligned}
& \#'(c, \{x\}, \phi, \lambda)(y) \\
&= \#'(\$label:D y:c?, \phi, \{x\}, x:c)(y) \\
&= \#'(D, \{x\}, \phi, x:c)(y) \\
&\quad \oplus \#'(c, \{y\}, \{x\}, x:c)(y)
\end{aligned}$$

ここで,

$$\#'(D, \{x\}, \phi, x:c)(y) = 0$$

一方,

$$\begin{aligned}
& \#'(c, \{y\}, \{x\}, x:c)(y) \\
&= \#'(\$label:D \ y:c?, \phi, \{y\}, x:c \circ y:c)(y) \\
&= \#'(D, \{y\}, \phi, x:c \circ y:c)(y) \\
&\quad \oplus \#'(c, \{y\}, \{y\}, x:c \circ y:c)(y)
\end{aligned}$$

ここで,

$$\#'(D, \{y\}, \phi, x:c \circ y:c)(y) = 1$$

一方,

$$\begin{aligned}
& \#'(c, \{y\}, \{y\}, x:c \circ y:c)(y) \\
&= \#'(\$label:D \ y:c?, \phi, \{y\}, \\
&\quad x:c \circ y:c \circ y:c)(y) \\
&= \#'(D, \{y\}, \phi, x:c \circ y:c \circ y:c)(y) \\
&\quad \oplus \#'(c, \{y\}, \{y\}, x:c \circ y:c \circ y:c)(y)
\end{aligned}$$

ここで,

$$\#'(D, \{y\}, \phi, x:c \circ y:c \circ y:c)(y) = 1$$

一方,

$$\#'(c, \{y\}, \{x\}, x:c \circ y:c \circ y:c)(y) = 0$$

以上から,

$$\begin{aligned}
& \#(A)(y) \\
&= 0 \oplus 0 \oplus 1 \oplus 1 \oplus 0 \\
&= 2^+
\end{aligned}$$

A.2 関数 C と $\#$ の正当性

この章では、関数 C と $\#$ の正当性を簡単に証明する。初めに、それらの証明に必要な準備を行う。

3.3.3 項で示した、エイリアスを展開したプロダクションを考える。

```

Example { expression:Add
        | expression:Mul
        | expression:Literal
        | "(" expression:Add ")"
        | "(" expression:Mul ")"
        | "(" expression:Literal ")"
        | "(" "(" expression:Add ")" ")"
        | "(" "(" expression:Mul ")" ")"
        | "(" "(" expression:Literal ")" ")"
        :
        :
        }

```

ここでは、右辺が(ときには無限の)選択枝の列挙であり、1つの選択枝が(しばしばラベル付けされた) $(Y \cup T)$ の要素の接続であるようなプロダクションを考える。非終端記号をTokenと見なし、1つの選択枝を、 $2^L \times Y'$ の上の文字列と考える。選択枝すべての集合を言語 $\bar{L}$ とする。 $\bar{L}$ を生成する文脈自由文法 $\bar{G} = (\bar{N}, \bar{\Sigma}, \bar{P}, \bar{S})$ を作ることを考える。 $\bar{S}$ は、現在扱っているプロダクションの左辺(ここではExample)とする。 $\bar{\Sigma} = 2^L \times Y'$, $\bar{N} = (2^L \times A) \cup \{\bar{S}\}$ とする。直観的には、 $\bar{P}$ は次のようになる。

$$\begin{aligned}
\bar{P} = \{ & \bar{X} \longrightarrow \bar{\alpha} \mid \bar{X} \in \bar{N}, \\
& \bar{X} = \bar{S} \text{ のとき, } \bar{\alpha} \in \text{Lang}(\bar{e}(\langle \phi, \bar{S} \rangle)) \\
& \bar{X} \neq \bar{S} \text{ のとき, } \bar{\alpha} \in \text{Lang}(\bar{e}(\bar{X})) \\
& \}
\end{aligned}$$

ただし、 $\text{Lang}(\bar{e})$ は正規表現 $\bar{e}$ の表す言語で、 $\bar{e}(\langle \text{labels}, n \rangle)$ は、 $B(n)$ が $\$label$ を含む場合 $B(n)$ の $\$label$ を labels で置き換えてできる正規表現、含まない場合 $B(n)$ 全体を labels でラベル付けして得られる正規表現である。

このようにした場合、 $\bar{P}$ が無限集合になる場合がある。そこで、 $\bar{L}$ を生成する文法として、文脈自由文法を考えるのはやめ、 $\bar{G}$ は、 $\bar{P}$ が無限集合の場合も扱うように拡張された文脈自由文法と考えることにする。 $\bar{G}$ に対して、通常の文脈自由文法と同様に、直接の導出 $\longrightarrow$ が定義できる。 $\longrightarrow$ の反射推移閉包を導出 $\longrightarrow^*$ とする。

関係 $\rightarrow$ を、次で定義する。

$$\begin{aligned}
& \bar{X}_1 \rightarrow \bar{X}_2 \\
& \iff (\exists \bar{\alpha}, \bar{\beta} \in (\bar{N} \cup \bar{\Sigma})^*)(\bar{X}_1 \longrightarrow \bar{\alpha} \bar{X}_2 \bar{\beta})
\end{aligned} \tag{1}$$

ただし、 $\bar{X}_1 \in \bar{N}$, $\bar{X}_2 \in (\bar{N} \cup \bar{\Sigma})$ である。

また、 $\rightarrow$ の反射推移閉包を $\rightarrow^*$ とする。さらに、 $\stackrel{1*}{\rightarrow}$, $\stackrel{2*}{\rightarrow}$ を次で定義する。

$$\begin{aligned}
& \bar{X}_1 \stackrel{1*}{\rightarrow} \bar{X}_n \\
& \iff (\exists \bar{X}_2, \bar{X}_3, \dots, \bar{X}_{n-1} \in \bar{N}) \\
& \quad (\bar{X}_1 \rightarrow \bar{X}_2 \rightarrow \bar{X}_3 \rightarrow \dots \rightarrow \bar{X}_n \\
& \quad \wedge (i \neq j \iff \bar{X}_i \neq \bar{X}_j))
\end{aligned}$$

$$\begin{aligned}
& \bar{X}_1 \stackrel{2*}{\rightarrow} \bar{X}_n \\
& \iff (\exists \bar{X}_2, \bar{X}_3, \dots, \bar{X}_{n-1} \in \bar{N}) \\
& \quad (\bar{X}_1 \rightarrow \bar{X}_2 \rightarrow \bar{X}_3 \rightarrow \dots \rightarrow \bar{X}_n \wedge \\
& \quad (\bar{X}_1, \dots, \bar{X}_n \text{ に同じものが3度以上出現しない}))
\end{aligned}$$

あきらかに、次が成り立つ。

$$\bar{S} \stackrel{1*}{\rightarrow} \bar{X} \implies \bar{S} \stackrel{2*}{\rightarrow} \bar{X} \implies \bar{S} \stackrel{*}{\rightarrow} \bar{X}$$

また、 $\bar{X}_1 \rightarrow \bar{X}_2 \rightarrow \dots \rightarrow \bar{X}_n$ において、 $\bar{X}_i = \bar{X}_j$ であるような $i < j$ が存在するとき、

$$\begin{aligned}
& \bar{X}_1 \rightarrow \bar{X}_2 \rightarrow \dots \rightarrow \bar{X}_{i-1} \\
& \rightarrow \bar{X}_i = \bar{X}_j \rightarrow \bar{X}_{j+1} \rightarrow \dots \rightarrow \bar{X}_{n-1} \rightarrow \bar{X}_n
\end{aligned}$$

であるから、

$$\bar{S} \stackrel{*}{\rightarrow} \bar{X} \implies \bar{S} \stackrel{1*}{\rightarrow} \bar{X}$$

したがって、

$$\bar{S} \stackrel{1*}{\rightarrow} \bar{X} \iff \bar{S} \stackrel{2*}{\rightarrow} \bar{X} \iff \bar{S} \stackrel{*}{\rightarrow} \bar{X} \tag{2}$$

A.2.1 関数 C の正当性

関数 C の正当性をいうには、次を示せばよい。

$$((\exists L_1 \ni l)(\exists \bar{u}, \bar{v} \in \bar{\Sigma}^*)(\bar{S} \longrightarrow^* \bar{u}\langle L_1, y \rangle \bar{v})) \\ \iff y \in C(\bar{S})\langle l, \bullet \rangle$$

C の定義から、次がいえる。

$$y \in C(\bar{S})\langle l, \bullet \rangle \iff (\exists L_1 \ni l)(\bar{S} \xrightarrow{1*} \langle L_1, y \rangle) \quad (3)$$

したがって、

$$((\exists L_1 \ni l)(\exists \bar{u}, \bar{v} \in \bar{\Sigma}^*)(\bar{S} \longrightarrow^* \bar{u}\langle L_1, y \rangle \bar{v})) \\ \xrightarrow{(1)} (\exists L_1 \ni l)(\bar{S} \xrightarrow{1*} \langle L_1, y \rangle) \\ \xrightarrow{(2)} (\exists L_1 \ni l)(\bar{S} \xrightarrow{1*} \langle L_1, y \rangle) \\ \xrightarrow{(3)} y \in C(\bar{S})\langle l, \bullet \rangle$$

逆を示す。

$$y \in C(\bar{S})\langle l, \bullet \rangle \\ \xrightarrow{(3)} (\exists L_1 \ni l)(\bar{S} \xrightarrow{1*} \langle L_1, y \rangle) \\ \xrightarrow{(2)} (\exists L_1 \ni l)(\bar{S} \xrightarrow{*} \langle L_1, y \rangle) \\ \xrightarrow{(1)} (\exists L_1 \ni l)(\exists \bar{u}, \bar{v} \in (\bar{N} \cup \bar{\Sigma})^*) \\ (\bar{S} \longrightarrow^* \bar{u}\langle L_1, y \rangle \bar{v})$$

$\neg < > < \cup \cup$ では、死記号の存在する文法はコンパイル・エラーなので、

$$(\exists \bar{\alpha}, \bar{\beta} \in (\bar{N} \cup \bar{\Sigma})^*)(\bar{S} \longrightarrow^* \bar{\alpha}\langle L_1, y \rangle \bar{\beta}) \\ \implies (\exists \bar{u}, \bar{v} \in \bar{\Sigma}^*)(\bar{S} \longrightarrow^* \bar{u}\langle L_1, y \rangle \bar{v})$$

したがって、

$$y \in C(\bar{S})\langle l, \bullet \rangle \\ \implies (\exists L_1 \ni l)(\exists \bar{u}, \bar{v} \in \bar{\Sigma}^*) \\ (\bar{S} \longrightarrow^* \bar{u}\langle L_1, y \rangle \bar{v})$$

A.2.2 関数 $\#$ の正当性

$\#$ の正当性を示す。まず、次が成り立つことを示す。

$$((\exists y \in Y)(\exists L_1 \ni l)(\exists \bar{u}, \bar{v} \in \bar{\Sigma}^*) \\ (\bar{S} \longrightarrow^* \bar{u}\langle L_1, y \rangle \bar{v})) \\ \iff \#(\bar{S})(l) \neq 0$$

$\#$ の定義から、次がいえる。

$$\#(\bar{S})(l) \neq 0 \\ \iff (\exists y \in Y)(\exists L_1 \ni l)(\bar{S} \xrightarrow{2*} \langle L_1, y \rangle) \quad (4)$$

したがって、

$$((\exists y \in Y)(\exists L_1 \ni l)(\exists \bar{u}, \bar{v} \in \bar{\Sigma}^*) \\ (\bar{S} \longrightarrow^* \bar{u}\langle L_1, y \rangle \bar{v}))$$

$$\xrightarrow{A.2.1} C(\bar{S})(l) \neq \phi$$

$$\xrightarrow{(3)} (\exists y \in Y)(\exists L_1 \ni l)(\bar{S} \xrightarrow{1*} \langle L_1, y \rangle)$$

$$\xrightarrow{(2)} (\exists y \in Y)(\exists L_1 \ni l)(\bar{S} \xrightarrow{2*} \langle L_1, y \rangle)$$

$$\xrightarrow{(4)} \#(\bar{S})(l) \neq 0$$

続いて、次を示す。

$$((\exists y_1, y_2 \in Y)(\exists L_1, L_2 \ni l)(\exists \bar{u}, \bar{v}, \bar{w} \in \bar{\Sigma}^*) \\ (\bar{S} \longrightarrow^* \bar{u}\langle L_1, y_1 \rangle \bar{v}\langle L_2, y_2 \rangle \bar{w})) \\ \iff \#(\bar{S})(l) = 2^+$$

ここで、 $\bar{X}_0 = \bar{S}$, $\bar{X}_n = \langle L_1, y_1 \rangle$, $\bar{Y}_m = \langle L_2, y_2 \rangle$ として、

$$\begin{cases} \bar{X}_0 \rightarrow \bar{X}_1 \rightarrow \bar{X}_2 \rightarrow \dots \rightarrow \bar{X}_n \\ \bar{Y}_i \rightarrow \bar{Y}_{i+1} \rightarrow \dots \rightarrow \bar{Y}_m \\ (\exists \bar{\alpha}, \bar{\beta}, \bar{\gamma} \in \bar{\Sigma}^*)(\bar{X}_{i-1} \longrightarrow \bar{\alpha}\bar{X}_i\bar{\beta}\bar{Y}_i\bar{\gamma}) \end{cases} \quad (5)$$

を考える。 $\#$ の定義から、 $\#(\bar{S})(l) = 2^+$ のとき、およびそのときに限り、式 (5) が成り立つような、同じ記号を 3 回以上含まない列

$$\langle \bar{X}_1, \bar{X}_2, \dots, \bar{X}_n \rangle$$

および、同じ記号を 3 回以上含まない列

$$\langle \bar{X}_1, \bar{X}_2, \dots, \bar{X}_{i-1}, \bar{Y}_i, \bar{Y}_{i+1}, \dots, \bar{Y}_m \rangle$$

が存在する。一方、

$$((\exists \bar{u}, \bar{v}, \bar{w} \in \bar{\Sigma}^*) \\ (\bar{S} \longrightarrow^* \bar{u}\langle L_1, y_1 \rangle \bar{v}\langle L_2, y_2 \rangle \bar{w}))$$

$\iff$

$$((\exists i, n, m, \bar{X}_1, \bar{X}_2, \dots, \bar{X}_{n-1}, \\ \bar{Y}_i, \bar{Y}_{i+1}, \dots, \bar{Y}_{m-1})(5) \text{ が成り立つ}))$$

したがって、

$$((\exists y_1, y_2 \in Y)(\exists L_1, L_2 \ni l)(\exists \bar{u}, \bar{v}, \bar{w} \in \bar{\Sigma}^*) \\ (\bar{S} \longrightarrow^* \bar{u}\langle L_1, y_1 \rangle \bar{v}\langle L_2, y_2 \rangle \bar{w})) \\ \iff \#(\bar{S})(l) = 2^+$$

また、式 (2) から、

$$\bar{X}_0 \xrightarrow{*} \bar{X}_{i-1} \wedge \bar{X}_i \xrightarrow{*} \bar{X}_n \wedge \bar{X}_{i-1} \xrightarrow{*} \bar{X}_n \\ \implies \\ \bar{X}_0 \xrightarrow{1*} \bar{X}_{i-1} \wedge \bar{X}_i \xrightarrow{1*} \bar{X}_n \wedge \bar{X}_{i-1} \xrightarrow{1*} \bar{Y}_m$$

であるから、逆も成り立つ。

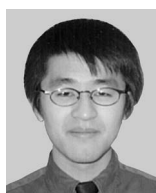
以上により、

$$\begin{aligned}
& ((\exists y_1, y_2 \in Y)(\exists L_1, L_2 \ni l)(\exists \bar{u}, \bar{v}, \bar{w} \in \bar{\Sigma}^*) \\
& \quad (\bar{S} \xrightarrow{*} \bar{u}\langle L_1, y_1 \rangle \bar{v}\langle L_2, y_2 \rangle \bar{w})) \\
& \iff \#(\bar{S})(l) = 2^+ \\
& \neg((\exists y \in Y)(\exists L_1 \ni l)(\exists \bar{u}, \bar{v} \in \bar{\Sigma}^*) \\
& \quad (\bar{S} \xrightarrow{*} \bar{u}\langle L_1, y \rangle \bar{v})) \\
& \iff \#(\bar{S})(l) = 0
\end{aligned}$$

したがって、 $\#(\bar{S})(l) = 1$ のとき、 $\bar{L}$ の要素の文字列には、 l は一度しか含まれない。

(平成 14 年 12 月 24 日受付)

(平成 15 年 7 月 22 日採録)



小藤 哲彦 (学生会員)

1976 年生。2000 年電気通信大学電気通信学部情報工学科卒業。2002 年同大学院電気通信学研究科情報工学専攻前期課程修了。実時間分散協調システム、言語処理系、分散シミュレーションの研究に従事。工学修士。



河野 健二 (正会員)

1970 年生。1997 年東京大学大学院理学系研究科情報科学専攻博士課程中退，同専攻助手に就任。理学博士。2000 年より電気通信大学情報工学科に勤務。現在，同大学同学科講師。1999 年より 2002 年まで科学技術振興事業団さがけ研究 21「情報と知」領域・研究員を兼務。オペレーティングシステム，分散・並列システム，プログラミング言語システム等のシステムソフトウェア全般に興味を持つ。1999 年度論文賞受賞。IEEE/CS，ACM 各会員。



竹内 郁雄 (正会員)

1946 年生。1969 年東京大学理学部数学科卒業。1971 年同大学院理学系研究科数学専攻修士課程修了。同年電電公社武蔵野電気通信研究所に入社。以来，NTT 基礎研究所等の基礎研究部門において，記号処理プログラミングシステムを研究・開発。NTT ソフトウェア研究所を経て，1997 年から電気通信大学情報工学科教授。現在，実時間分散協調システムの研究。独立行政法人防災科学技術研究所地震防災フロンティア研究センター兼務。工学博士。1990 年論文賞受賞。ACM，日本ソフトウェア科学会各会員。