東京大学
情報理工学系研究科 創造情報学専攻
修士論文

LL(1) 文法の部分集合から flat-chaining および sub-chaining スタイル の fluent interface を 自動生成する領域特化言語

A domain-specific language for generating fluent interfaces supporting flat and sub-chaining styles from LL(1) subsets

山隈 由衣
Yui Yamaguma

指導教員　千葉 滋 教授

2025 年 1 月

概要

Fluent interface は，method chain と呼ばれるメソッド呼び出しの連鎖からなるインターフェースで, EDSL の設計などに利用される．Fluent interface は手動開発が難しく, 様々な文法クラスに対応する fluent interface 生成器が開発されている．Fluent interface の表現方法の一種である sub-chaining は, 複雑な表現をする際に有用であるが, 既存の sub-chaining の fluent interface 生成アルゴリズムは正確性が保証されていないなどの課題がある．

本論文では，LL(1) 文法から flat-chaining および sub-chaining の fluent interface を生成するツールの提案をおこなう．Python 上の埋め込み DSL を使って開発者が BNF 風の文法規則を記述すると Java で書かれた fluent interface を生成する生成器の設計とその生成アルゴリズムについて述べる．提案する生成アルゴリズムは，Greibach 標準形の LL(1) 文法から flat-chaining の fluent interface を生成する手法を，sub-chaining に拡張したものである．入力文法を Greibach 標準形に変換し, sub-chaining の表現を埋め込んで fluent interface 生成アルゴリズムを適用する．また, method chain を入力文法にしたがって解析した AST を構築する．このツールにより ε ルールを除去できる LL(1) 文法からであれば，確実に正確な flat-chaining および sub-chaining の fluent interface の生成が可能になる．

Abstract

Fluent interface is an interface that composed by chain of method calls (called method chain). Fluent interface is helpful for design EDSL. Manually developing fluent interface is difficult. Fluent interface generators for various grammar class are developed. Sub-chaining, a style of method chaining is useful to describe complex chain. However, existing algorithms to generate fluent interface for sub-chaining have problems that correctness is not guaranteed. In this research, we propose tool to generate fluent interface for both flat-chaining and sub-chaining from LL(1) grammar. Developer describes grammar rule like BNF using embedded DSL on Python. Then fluent interface written by Java is generated. In this paper we describe the generator' s design and generation algorithm. The algorithm is expansion of method to generate fluent interface for flat-chaining from LL(1) in Greibach normal form. This tool converts input grammar to Greibach normal form, embeds sub-chaining style and applies algorithms to generate fluent interface. In addition to that, it analyzes method chain according to input grammar and constructs AST This tool enables to certainly generate correct fluent interface for both flat-chaining and sub-chaining from LL(1) grammar that can be removed ε-rules.

目次

第 1 章

はじめに

Fluent interface は，method chain からなるインターフェースで, 近年需要が高まっている．Method chaining の各メソッドはオブジェクトを返し，そのオブジェクトから次のメソッドが呼び出される．Fluent interface を使って method chain を記述でき、埋め込み型領域特化言語 (embedded domain specific languages, EDSLs) のようなライブラリの設計に有用である．Fluent interface の文法エラーはホスト言語の静的な型エラーとして検出できる．静的型検査を行う仕組みは，文法に対応するプッシュダウンオートマトンのスタックを表現する型を定義して構文検査を行う．このようなインターフェースの開発はたくさんの型定義を必要とするので, 手作業では難しい．Fluent interface の複雑な実装をサポートするコード生成器が開発されている．

さまざまな文法クラスに対応する fluent interface 生成器が提案されている．Fluent interface の表現方法には，flat-chaining と sub-chaining の 2 種類がある．Sub-chaining は意味的なまとまりを理解しやすく, 長いあるいは複雑な連鎖の表現に有用である．Flat-chainging と sub-chaining の両方に対応する fluent interface 自動生成器で，対応可能な fluent interface の文法クラスが明らかになっているものはない．

本論文では，ε ルールを除去可能な LL(1) 文法から flat-chaining および sub-chaining の fluent interface を文法エラーの静的検査付きで生成するツール GreenChain を提案する．GreenChain は Python 上の EDSL である．本論文は GreenChain のために fluent interface を自動生成するための新しいアルゴリズムを示す．このアルゴリズムは，ε ルールを除去可能という制限があるものの LL(1) 文法から flat-chaining および sub-chaining の fluent interface を文法エラーの検出機能つきで生成する現実的なアルゴリズムである．実用的なツールを開発した．GreenChain は与えられた LL(1) 文法から ε ルールを除去し，これを GNF の LL(1) に変換する．変換後の文法に sub-chaining の表現を埋め込んだ文法を flat-chaining の fluent interface 生成器に入力し, flat-chaining と sub-cahining の両方を表現できる fluent interface を得る．Fluent interface 生成器では, 入力文法を対応するオートマトンに変換し, これをクラス定義にエンコードして文法を表現している．このツールは method chain を入力文法にしたがって解析した AST を構築する．GreenChain で対応範囲内外の入力文法を記述し、動作の検証を行った．

本研究の貢献は大きく以下のつである.

1. LL(1) 文法の部分集合から flat-chaining と sub-chainig の両方に対応する fluent interface を生成するアルゴリズムを提案した.
2. 実用的なツールを開発した.
3. 対応可能な文法の範囲を明確にした.

本論文の構成について説明する. 第 2 章では, fluent interface の自動生成の背景について述べ, 関連研究を紹介する. 第 3 章では, ツール GreenChain の EDSL としての使い方と flunet interface 生成のアルゴリズムについて詳細に説明する. 第 4 章では, GreenChain の flat-chaining と sub-chaining それぞれに対応した利用例と構築される AST を紹介する. 第 5 章では, 本論文のまとめと今後の課題について述べる.

第 2 章

背景

Fluent interface[1] は，メソッド呼び出しの連鎖からなるインターフェースである．近年，高機能なライブラリの利用インタフェースとして採用されることが増えている [2]．ホスト言語の型システムを利用することでメソッドの並び順の間違いを静的に検出することも可能だが，インタフェースの実装が複雑になる．このため，複雑な実装を支援するために，インタフェースの実装のひな形を出力するコード生成器の開発が重要である．Fluent interface の表現方法には, flat-chaining と sub-chaining の 2 種類がある．Flat-chaining の fluent interface 生成器を，sub-chaining の生成器に応用する手法が研究されている．本論文では，LL(1) のより大きい部分集合から sub-chaining と flat-chaining の fluent interface を生成するツールを提案する．このツールを用いると Graibach 標準形 (GNF) で書ける LL(1) から flat-chaining の静的型検査付き fluent interface を生成する手法があれば，ε ルールを除去できる LL(1) 文法から flat-chaining および sub-chaining の fluent interface の生成が可能になる．

2.1 Method chain

メソッド呼び出しの連鎖を method chain と呼ぶ．近年 method chain の利用が増加している．より多くのプロジェクトでより長い method chain が使われるようになっている [3]. Method chain の表現方法には, flat-chaining と sub-chaining の 2 種類がある．Flat-chaining は，以下のように一つの method chaining からなる．

```
1 # flat-chaining による表現
2 i(9).add().i(2).multiple().i(3)
```

上記のコードは式 $9 + 2 \times 3$ を method chain で表現している．Flat-chaining は，短いあるいは単純な連鎖の表現に有用である．複雑な method chain を flat-chaining で表現すると, 理解が難しくなる．これに対して sub-chaining は複数の method chaining で表現する．Sub-chaining では，以下のように，あるメソッド呼び出し連鎖を他の連鎖に引数として渡す．

```
1 # sub-chaining による表現
2 add(i(9), i(2).multiple().i(3))
```

上記の例は flat-chaining の例と同じ意味を表現しているが, 加算する二つの項を表す method chain `i(9)` と `i(2).multiple().i(3)` をメソッド `add()` に引数として渡していおり, + (9, 2 × 3) の形になっている. Sub-chaining は意味的なまとまりを理解しやすく, 長いあるいは複雑な連鎖の表現に有用である.

2.2 EDSLs

EDSLs(embedded domain specific languages, 埋め込み型領域特化言語)[4] は, 領域特化言語を汎用プログラミング言語を通して使用できる言語である. 例えば, JOOQ[5] は領域特化言語の一つである SQL 文を表現する Java のコードを生成する. 以下のような SQL クエリに対して,

```
1 /* SQL クエリ */
2 SELECT TITLE
3 FROM BOOK
4 WHERE BOOK.PUBLISHED_IN = 2011
5 ORDER BY BOOK.TITLE
```

以下のクエリが Java で生成される.

```
1 \\ SQL 文を表現する Java の EDSL を用いて記述した method chain
2 create.select(BOOK.TITLE)
3         .from(BOOK)
4         .where(BOOK.PUBLISHED_IN.eq(2011))
5         .orderBy(BOOK.TITLE);
```

EDSLs を使って専門的な知識がなくても領域特化言語を利用できる. この他にも, Java で文字列を操作する StringBuilder や, Spring でセキュリティを構成する httpSecurity, Java のコードで HTML を生成する j2html[6] などさまざまな fluent interface が利用されている.

2.3 Fluent interface

Fluent interface は, method chain からなるインターフェースである. method chaining の各メソッドはオブジェクトを返し, そのオブジェクトから次のメソッドが呼び出される. fluent interface は EDSLs のようなライブラリの設計に有用である.

Fluent interface の利用者は誤ったメソッドの並びを書いてしまうことがあるが, そのような誤った並びは文法エラーと見なすことができる. この fluent interface の文法エラーはホスト言語の静的な型エラーとしてコンパイルときにエラーとすることができる. 例えば, 以下の method chain

```
1 // 誤ったメソッド呼び出し
```

```
2 create.select(BOOK.TITLE).from(BOOK).from(BOOK)
```

の最後の from メソッドは構文的に誤った呼び出しである．このようなものを型検査により静的に検出できる．また, 現在の型から次のメソッドの候補がわかるため, コード補完が利用できる．method chain の途中まで記述すると次に呼び出せるメソッドが補完候補に表示される．また, 構文的な誤りを静的に検出できることから, 実行時のエラーを避けられる点も実用的である．これらのことから, fluent intarface を通してより簡単に EDSLs を利用できる．

2.4 Fluent interface 生成器

このような fluent inferaface の文法エラーを型検査で判別するようなインターフェースを手作業で開発するのは難しい．なぜなら，ホスト言語上でたくさんの型を定義する必要があるからである．そこで, method chain の並びの文法を入力すると自動で fluent interface を生成する手法が研究されている．

Fluent interface 生成器では, 意図する method chain の並びを文法で表現し, これを入力とする．例えば, 加算と乗算の前置記法を表現する method chain の規則は, BNF の文法

終端記号 (T, terminal symbol)：i, add, multiple
非終端記号 (N, non-terminal symbol)：E, T, F
生成規則 (δ , production rules)：
E → add T E |T
T → multiple F t |F
F → i
開始記号 (S, start symbol)：E

で表現できる．Fluent interface 生成器は, 入力された文法で表現された method chain を受理し, 構文的に誤りがある method chain を検出するインターフェースを生成する．入力文法を対応するオートマトンに変換し, それを表現するホスト言語のクラスを定義する．[7] 上記の文法は LL(1) 文法と呼ばれる文法に該当する．LL(1) 文法は, 次の一つのラベルを先読みすることで適用する生成規則が一意に決まる文法で, 正規文法よりも大きく, 文脈自由文法よりも小さい文法クラスである．LL(1) 文法は以下のようにしてオートマトンに変換できる．

各記号を展開して得られる終端記号列の最初になり得る記号の集合である first セットと, 各記号の直後になり得る記号の集合 follow セットを作成する (図 2.1)．first セットと follow セットをもとに, 解析表を作成する (図 2.2)．文法の各規則 A → s について

1. a ∈ first(s) なら A → s を M[A, a] に加える．
2. b ∈ follow(A) なら A → s を M[A, b] に加える．

LL(1) 文法の解析表の要素が重複することはない．LL(1) 文法は次に読む一語がわかっていれば適用する規則が一意に定まる文法のためである．空でない M[A, a] に対して, A に対応するスタックがトップのときに a が読まれると M[A, a] の二語目以降の記号に対応するスタック

をプッシュする遷移規則を定義して, 文法に対応するオートマトンが得られる．このオートマトンを表現するクラスを定義することで fluent interface が生成されるが, 大きい文法が入力されると生成されるクラスの数は膨大になる．これが手動生成が難しい原因である．

記号	first セット	follow セット
E	add, multiple, i	
T	multiple, i	add, multiple, i
F	i	add, multiple, i

図 2.1. first セットと follow セット

	add	multiple	i
E	add T E	multiple F T	i
T		multiple F T	i
F			i

図 2.2. 非終端記号とメソッドの解析表

EDSLs をサポートするコードを生成する EriLex[8] では, 入力文法を状態を持たない決定性実時間プッシュダウンオートマトン (single-state realtime deterministic pushdown automaton, ssrDPDA) に変換し, その遷移規則を表すクラスの集合を生成する．プッシュダウンオートマトンとはスタック付きの有限オートマトン, 実時間とは ε 遷移を含まないことである．ラベルとスタックのトップの記号から適用する遷移規則を決定する．文法とオートマトンには対応関係があることが知られている (図 2.3).

文法	オートマトン
無制限文法	チューリングマシン
文脈依存文法	線形拘束オートマトン
文脈自由文法	プッシュダウンオートマトン
正規文法	有限オートマトン

図 2.3. 文法とオートマトンの対応

ssrDPDA は図 2.4 のように表される．遷移規則 a, Y / AB は, 図 2.5 のように, スタックのトップが Y のときにラベル a が読まれると Y をポップして A と B をプッシュすることを表す.

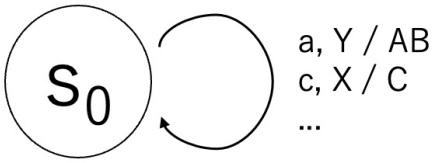

図 2.4. srrDPDA

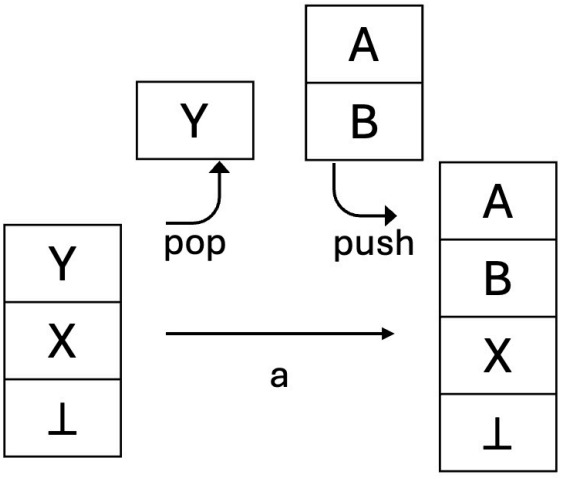

図 2.5. ssrDPDA におけるスタックの操作

静的型検査をおこなう仕組みは，文法に対応するプッシュダウンオートマトンのスタックを表現する型を定義して構文検査を行う．文法の規則が多いと，使うスタックの種類も多くなる．

文法をオートマトンに変換する過程では, 入力文法の非終端記号をスタック記号とする. 生成規則 $nt \rightarrow t\ nt_1\ ...\ nt_n$ を, t をラベルとして nt をポップし, $nt_1\ ...\ nt_n$ をプッシュする遷移規則とみなす. 遷移規則と生成規則は一対一対応している. 例えば, 非終端記号 z, z_1, z_2 に対して生成されるクラスを $ez < \kappa >, ez_1 < \kappa >, ez_2 < \kappa >$(κ は型パ ラメータ) とする. 型 $ez < ez_1 < ez_2 < \perp >>>$ は z, z_1, z_2 をスタックに持つ PDA 構成を表す. メソッド名はラベルを表す. 次に, 各非終端記号に対して一つの型パラメータを持つ総称型クラスを生成する. 型パラメー タは PDA のスタックを表す. $\perp$クラスは空のスタックを表すクラスである. メソッド呼び出しの連鎖は PDA の遷移の連鎖を, メソッド連鎖の接頭辞の型は, その遷移の結果としての PDA の構成を表す. 例えば, $ez < \kappa >$ が以下の 2 つのメソッドを持つ場合：1. public κ a(); 2. public $ez_1 < ez_2 < \kappa >>$ b(); メソッド a は遷移規則 $z \rightarrow$ a(ラベル a で z をポップする) を表す. メソッド b は遷移規則 $z \rightarrow$ bz_1z_2(ラベル b で z を ポップ して, z_1z_2 を プッシュ する) を表す. PDA の元の構成が $ez < \perp >$ で表されるとすると, メソッド a の呼び出しで PDA は構成$\perp$に, メソッド b の呼び出しで構成 $ez_1 < ez_2 < \perp >>$ に遷移する. EriLex のような fluent interface 生成器は, ライブラリの設計者の利用を想定している. EriLex が生成した fluent interface を使って EDSLs の利用者が method chain を記述できる. このような入力文法に対応したオートマトンを表すクラス定義により, method chain の構文的な誤りを型検査を通して検出できる.

さまざまな文法に対応するこのような fluent interface の生成手法が提案されている. Gil らは LR 文法から flat-chaining の fluent interface を生成する手法を提案した [9][10]. LR 文法は LL(1) 文法よりも大きく文脈自由文法よりも小さい文法クラスである. 中間のオートマトンとしてジャンプスタック実時間決定性プッシュダウンオートマトン (jump-stack realtime deterministic pushdown automaton, JRDPDA)[11] を用いている. Grigore らが提案したチューリングマシンを介した任意の文脈自由文法に対応する fluent interface 生成器 [12] は, 型検査にかかる実行とき間が非常に長く実用的でない. このように, 大きい文法クラスに対応する flat-chaining の fluent interface 生成器が提案されている. Java 以外の言語についても, Scala の DSL を作るライブラリである ScaLALR[13] などの生成器が開発されている.

2.5　Sub-chaining に対応した fluent interface 生成器

Sub-chaining の fluent interface も開発されている [14][15]. Flat-chaining の fluent interface 生成手法を応用して, sub-chaining を表現できる fluent interface を生成する手法も研究されている. Silverchain[16] は LL(1) の部分集合から sub-chaining と flat-chaining を生成できるが，部分集合が不明である.

Yamazaki らによって，LL(1) から flat-chaining のインターフェースを生成するアルゴリズムがあれば，LL(1) から flat-chaining と sub-chaining の両方を生成するアルゴリズムが提案されている [17].

文法	fluent interface 生成器	中間のオートマトン
LL(1)の部分集合	EriLex[3] Sub-chaining にも対応： SilverChain[4] Yamazaki らの手法[5]	ssrDPDA
LR 文法	Gil らの手法[8][9]	JRDPDA 等
文脈自由文法	Grigore らの手法[10]	チューリングマシン

図 2.6. 各文法に対応する fluent interface 生成器

しかし該当する LL(1) 文法から flat-chaining への fluent interface 生成器は我々の知るかぎり存在しない．本論文は GNF で書ける LL(1) の flat-chaining の fluent interface 生成器が存在する，というより現実的な仮定を置いて提案をおこなっている．このアルゴリズムでは, flat-chaianing の文法を G = Σ, N, δ, Ainit, sub-chaining の表現を埋め込む以下のような書き換え関数 f を用いる.

$f(G) = \Sigma', N, \delta', A_init,$

$\Sigma' = \Sigma \cup subchain(A)jA \in N,$

$\delta'(A) = subchain(A) \cup \delta\ (A).$

f(G) を入力として, flat-chaining の fluent interface 生成器から sub-chaining を表現する fluent interface を生成できる．この手法では LL(1) から flat-chaining のインターフェースを生成するアルゴリズムを前提としていたが, そのようなアルゴリズムは知られていない．知られているのは GNF(グライバッハ標準形，Graibach Normal Form)[18] の LL(1) 文法に対応する flat-chaining のインターフェースを生成するアルゴリズム [8] だけである．GNF は全ての生成規則の右辺が終端記号から始まる文法の形である．LL(1) 文法の中には等価な GNF が存在しないものがある [19] ので，GNF で書ける LL(1) 文法から flat-chaining のインタフェースを生成できても，任意の LL(1) から falt-chaining のインタフェースを生成できるとは限らない．例えば，言語 $a^n(b^kd + b + cc)^n | n >= 1(k >= 1)$ を認識する以下の文法は，ε ルールを除去できない [19].

S → a D A

D → a D A | ε

A → c c |b B

B → b^{k-1}d | ε

GNF で書ける文法は ε ルールを含まないことから，この文法は等価な GNF が存在しない．こ

のような文法が入力として与えられた場合, この手法では入力文法に対応する fluent interface は生成できない. また理論のみの提案であり, このアルゴリズムにもとづいたツールの開発は行われていない.

以上のように, sub-chaining にも対応した fluent interface 生成器が提案されている. しかし, 対応可能な文法クラスが明らかになっている現実的なアルゴリズムと生成器はない.

第 3 章

GreenChain の提案

3.1 提案の概要

本論文では，ε ルールを除去可能な LL(1) 文法から flat-chaining および sub-chaining の fluent interface を文法エラーの静的検査付きで生成するツール GreenChain を提案する．GreenChain は Python 上の EDSL である．本論文は GreenChain のために fluent interface を自動生成するための新しいアルゴリズムを示す．このアルゴリズムは，ε ルールを除去可能という制限があるものの LL(1) 文法から flat-chaining および sub-chaining の fluent interface を文法エラーの検出機能つきで生成する現実的なアルゴリズムである．GreenChain の概要は図 3.1 の通りである．

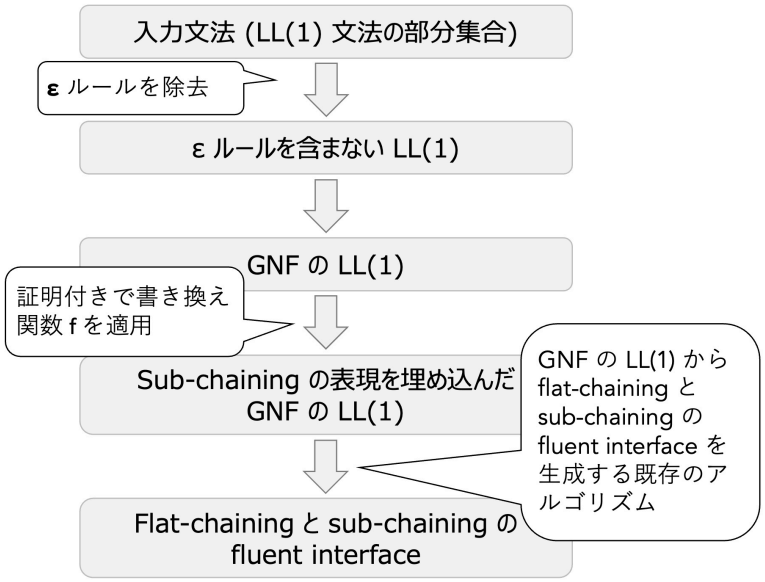

図 3.1. Greenchain の概要

3.2　目的

LL(1) 文法から flat-chaining と sub-chaining の両方に対応する fluent interface を生成するアルゴリズムが提案されているが, LL(1) 文法のどの部分集合から正しい fluent interface を生成できるか明らかでない. 本研究の貢献は以下のとおりである.

1. ε に展開される記号を規則の最後に含まない LL(1) 文法から flat-chaining と sub-chainig の両方に対応する fluent interface を生成するアルゴリズムを提案した.
2. このアルゴリズムで用いる書き換え関数 f[17] は、任意の LL(1) を入力として受け取ることを想定して提案されたが、本論文では GNF の LL(1) 文法を入力とする。そこで GNF の LL(1) 文法を与えても sub-chaining が埋め込まれた GNF の LL(1) 文法が出力されることを形式的に確認した.
3. LL(1) 文法を GNF に変換するアルゴリズムを実装した.
4. 実用的なツールを開発した.

3.3　利用方法

GreenChain の EDSL は BNF 風の文法を記述するための言語で，GreenChain はこの記述から Java で書かれた fluent interface の雛形を生成する．例えば 2.4 節で示した加算と乗算の前置記法を表現する文法を，GreenChain では Python でプログラム 3.1 のように記述する．

```
1  # GreenChain を用いて記述した，文法を入力する method chain
2  fi.Lang
3  .rule().nt('E').arrow().t('add').nt('T').nt('E')
4  .rule().nt('E').arrow().nt('T')
5  .rule().nt('T').arrow().t('multiple').nt('F').nt('T')
6  .rule().nt('T').arrow().nt('F')
7  .rule().nt('F').arrow().t('i')
8  .terminal('i').arg_type('Integer')
9  .start_from('E').end()
```

プログラム 3.1. BNF 風の入力文法の記述

各生成規則は `rule` から始まり，`nt` は非終端記号，`t` は終端記号を表す．各終端記号に対応するメソッドの引数の型は `arg_type` で指定する．`arg_type` を指定しない場合は引数はなしになる．開始記号は `start_from` で与える．展開規則を一つずつ記述する形式で, プログラム 3.1 の 3 行目は規則 E → add T E を表現し, 3 行目と 4 行目を合わせて規則 E → add T E |T を表現している．

GreenChain では method chain を構文解析し, 抽象構文木 (Abstract Syntax Tree, AST) を出力できる．AST は, 文法にしたがって method chain を構文解析した結果を木構造で表現する．規則 E → add T E |T にしたがって展開される場合, E を表すノードが add, T, E を表すノードを子に持つ木で表される．例えば, 以下の method chain

```
1 add().i(9).multiple().i(2).i(3)
```

を入力文法に従って構文解析すると, 図 3.2 のような AST が得られる．

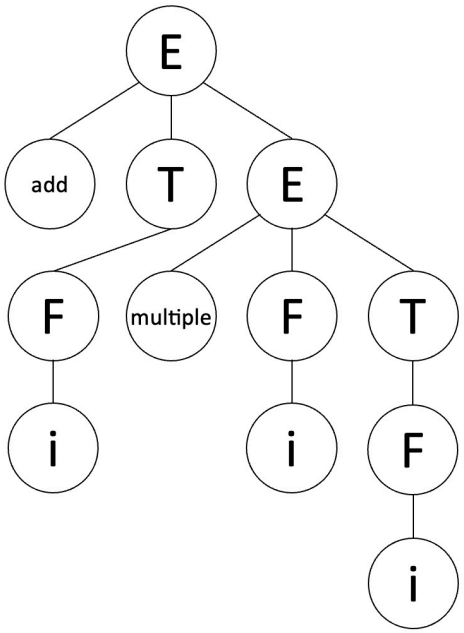

図 3.2. method chain を構文解析して得られる AST

3.4 生成アルゴリズム

GreenChain は与えられた LL(1) 文法からまず ε ルールを除去し，これを GNF で書ける LL(1) に変換し，sub-chaining と flat-chaining の fluent interface を生成する．以下では Python の EDSL で記述した文法を変換して得られた GNF で書ける LL(1) 文法を G = (T, N, δ , S) とする．T は終端記号，N は非終端記号，δ は生成規則，S は開始記号である．

3.4.1　ε ルールの除去

L(A)⊇ ε となる記号を nullable な記号と呼ぶ [19]．ここで L(A) は，A を展開して生成される言語である．以下のようにして nullable な記号の集合を得る．

1. ε ∈ δ (A) となる全ての A を, $nullableSymbols$ に加える．
2. 空のリスト $newNullableSymbols$ を新しく定義する．
   $\forall\ B \in N, \forall\ Z \in \delta\ (B)$
   $Z = Z_1 ... Z_n$
   とする．全ての $1 <= i = n$ について以下が成り立つとき, B を $newNullableSymbols$ に加える．$Z_i \in nullableSymbols$
3. $nullableSymbols = newNullableSymbols$ になるまで, 2. をくり返す.

nullable な記号とその後ろの non-nullable な記号を組み合わせて，新しい記号を作る．A_i を nullable, B_i を non-nullable な記号とすると, 以下の規則

$X \rightarrow B_1 A_2 B_3 B_4 A_5 A_6 B_7$

を以下のように変換する．[] 内は新しい非終端記号を表す.

$X \rightarrow B_1 [A_2 B_3] B_4 [A_5 A_6 B_7]$

例えば以下の規則に対して,

S → a B C

B → b D | ε

C → c D |e

D → d

nullable な記号 B とその後ろに出てくる記号 C を組み合わせた新しい記号 B' を以下のように定義する.

B' → b D C |C

記号 B が含まれる規則 S → a B C を以下の規則に置き換える.

S → a B'

これにより ε ルールが除去できる．ただし，規則の右辺が nullable な記号で終わる場合には適用できない.

3.4.2　ε ルールなしの LL(1) 文法から GNF で書ける LL(1) への変換

この変換では規則の右辺の始めの記号を，終端記号になるまで展開する．$\forall A \in N, \forall X \in \delta$ (A) について，X を $aY_1 ... Y_n (n \geq 1, Y_i \in N \cup T (1 \leq i \leq \mathrm{n}))$ とする．a ∈ N の場合，δ（A）から X を削除して，以下の規則を δ (A) に追加する.

$\forall Z \in \delta$ (a).

$ZY_1 ... Y_n$

さらに X を Z に置き換えて，a $\notin$ N となるまで繰り返す．左再帰があるとこの繰り返しは停止しないが，LL(1) は左再帰を許さないので，繰り返しは停止する．このようにして ε ルールを含まない全ての LL(1) 文法を GNF に変換できる．

3.4.3 GNF で書ける LL(1) に sub-chaining の表現を埋め込む

ここでは Yamazaki アルゴリズム [17] を応用したアルゴリズムで生成をおこなう．アルゴリズムは，GNF で書ける LL(1) から flat-chaining の fluent interface を生成するツールが存在することを仮定する．まず 3.2.2 の変換で得られた文法 G から sub-chaining の表現を埋め込んだ次のような文法 G' $=$ (T', N, δ', S) を得る．

T' $=$ T $\cup$ $\{A_{subchain}$ |A $\in$ N$\}$

δ'(A) $=$ $\{A_{subchain}\}$ $\cup$ δ (A)

$A_{subchain}$ は，A に対応する sub-chaining を表現する記号である．次に GNF で書ける LL(1) から flat-chaining の fluent interface 生成器で，G' から flat-chaining の fluent interface を生成する．これは G に対応する flat-chaining と sub-chaining の両方を表現する fluent interface である．このアルゴリズムは Yamazaki のアルゴリズムと同じであるが，Yamazaki アルゴリズムは任意の LL(1) から flat-chaining の fluent interface を生成するツールの存在を仮定している．本論文のアルゴリズムは，GNF で書ける LL(1) から flat-chaining の fluent interface を生成するツールが存在することを仮定するので，その正しさを示すためには別途証明が必要である．

以下では次のことを証明する．

G が空列を生成する非終端記号を含まない GNF で書ける LL(1) なら，G' も GNF で書ける LL(1) になる．G $=$ (T, N, δ, S) が GNF で書ける LL(1) 文法のとき，以下の条件が満たされる．

1. $\forall A \in N, \forall X, Y \in$ δ (A).
   $X \neq Y \rightarrow first_G(X) \cap first_G(Y) = \phi$
2. $\forall A \in N$.
   $nullable_G(A) \rightarrow first_G(A) \cap follow_G(A) = \phi$
3. $\forall A \in N, \forall X, Y \in$ δ (A).
   $X \neq Y \rightarrow not(nullable_G(X) and nullable_G(Y))$
4. $\forall\ A \in N$, δ (A) の規則が以下のいずれかになる．
   $A \rightarrow \varepsilon$ $(A = S$ のときのみ$)$
   $A \rightarrow a$
   $A \rightarrow aY_1...Y_n(n >= 1, a \in T, Y_i \in N(1 <= i <= n))$
5. 2. の規則の右辺に開始記号が出現しない．
   $Y_i \neq S$

$first_G(A)$ は文法 G における first セットである．$first_G(A)$ は文法 G における A を展開し

て得られる終端記号列の最初になり得る記号の集合である.

(i) G が 1. を満たすとき，G が空列を生成する非終端記号を含む場合を除いて G' も 1. を満たすことが示されている [17].
(ii) G が 2, 3 を満たすとき, G' もそれぞれ 2, 3 を満たすことが示されている. [17].
(iii) G が 4. を満たすとき,
$A_{subchain}$ が終端記号であることから規則 $A \to A_{subchain}$ は 2. を満たす. 前提から δ (A) の規則も 2. を満たすから, $\delta'(A) = \{A_{subchain}\} \cup \delta\ (A)$ の規則も 2. を満たす.
すなわち G' も 4 を満たす.
(iv) G が 5. を満たすとき,
$A_{subchain}$ は開始記号ではない. 前提から δ (A) の規則も右辺に開始記号を含まないから, $\delta'(A) = \{A_{subchain}\} \cup \delta\ (A)$ の右辺に開始記号が出現しない.
すなわち G' も 5. を満たす.

以上のことから入力文法 G が空列を生成する非終端記号を含まない GNF で書ける LL(1) なら，変換後の文法 G' も GNF で書ける LL(1) になる.

3.4.4　flat-chaining と sub-chaining の両方を表現する fluent interface の生成

Sub-chaining の表現を埋め込んだ文法を,　LL(1) 文法から flat-chaining の fluent interface を生成する生成器に入力し, flat-chaining と sub-cahining の両方を表現できる fluent interface を得る. この flat-chaining の fluent interface 生成器は, GNF で書ける LL(1) から flat-chaining の fluent interface を生成する現実的なアルゴリズムによって変換をおこなう生成器を独自に実装したものを使う. まず, 文法を対応するオートマトンに変換する. 任意の LL(1) 言語は対応する決定性プッシュダウンオートマトンで認識できることが示されている [20]. 次に得られたオートマトンをもとに対応するクラス定義を生成する. これにより flat-chaining と sub-cahining の両方を表現できる fluent interface が生成される.

3.4.5　入力文法が対応範囲外の場合の処理

GreenChain は nullable symbol が右辺の最後に来るような生成規則を含まない LL(1) 文法からであれば正しく fluent interface を生成できる. そこでまず, 入力文法が以下の条件を満たさないかチェックをおこなう.

(i) Nullable symbol が右辺の最後に来るような生成規則を含む
(ii) 解析表のセルの要素が二つ以上になる LL(1) 文法は非終端記号と次に読む一語から適用する規則が一意に定まる文法だが, 解析表の要素が重複する文法はこれに該当しないため LL(1) 文法ではない. 解析表作成ときにこの条件に当てはまるか判定する.

これらの入力を受け取った場合は, 対応する fluent interface を正しく生成できない可能性が

あるため, 処理を中断してエラーを出力し, fluent interface の生成は行わない.

3.5 AST の構築手法

Method chain を入力文法にしたがって構文解析した結果を表す AST は, 一般的な木構造を使って以下のように構築する. まず, 初期状態を表すメソッド内で初期化する. 次に, 呼び出されたメソッドとプッシュされるスタックに対応するノードを, そのクラスのノードの子とする. メソッドを表すノードは葉となる. 例えば, 3.3 節で示した method chain のメソッド `add()` が呼ばれると, 初期状態を表すスタック E のノードの子に, メソッド `add()` およびスタック T, E を表すノードが追加される (図 3.3). このようにして, 図 3.2 のような AST が得られる. スタックに対応するノードは, そのスタックを表すインスタンスに変数として渡す. Method chain の最後のメソッドが構築された AST を返す.

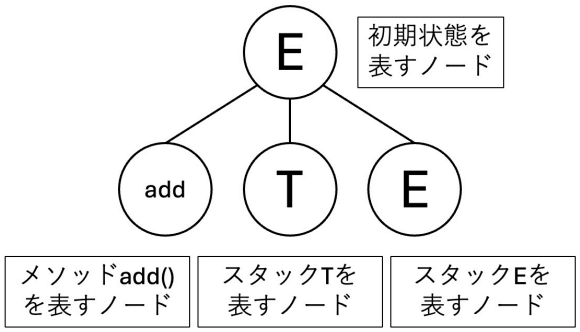

図 3.3. AST のノードの詳細

第 4 章

利用例

GreenChain の利用例を, 以下の要素とともに示す.

(i) Python 上の EDSL で記述した文法
(ii) 入力文法から得られる fluent interface
(iii) 生成された fluent interface を通して method chain を表現する Main 関数
(iv) 構築される AST

図 4.1 の手順で検証を行う．入力文法を EDSL で記述した Python プログラムを実行し, Java の fluent interface を生成する．生成された fluent interface とは別のファイルに Java の Main 関数を作成する．この Main 関数内で, 構文的に正しい method chain を表現し, Java プログラムを実行する．

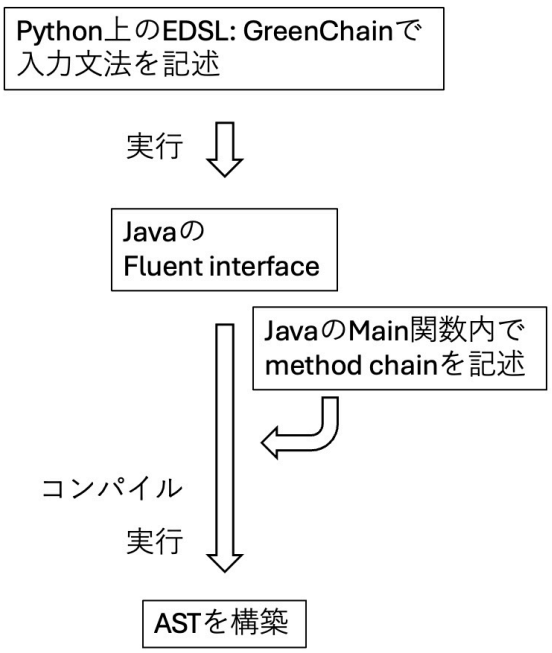

図 4.1. 検証の手順

4.1 Flat-chaining の fluent interface 生成

プログラム 4.1 は, 2.4 節で示した加算と乗算の前置記法を表現する文法から生成される fluent interface である.

```
1  \\ Fluent interface のクラス
2  class FI {
3          ...
4
5          \\ 初期状態を表すメソッド
6          SE<BottomOfE> E() { ... }
7
8          \\ 各スタックを表すクラス
9          static class SE<Rest> {
10                 ...
11
12                 ST<SE<Rest>> add() { ... }
13
14                 SF<ST<Rest>> multiple() { ... }
15
16                 Rest i(Integer arg0) { ... }
17                 ...
18         }
19
20         static class ST<Rest> { ... }
21
22         static class SF<Rest> { ... }
23 }
```

プログラム 4.1. 生成された fluent interface

Fluent interface のクラスは，各スタックを表すクラス，初期状態を表すメソッドを持つ.

各スタックを表すクラス (SE 等) には，スタックの状態を表す変数 rest と，AST のデータを持つ tree, そのスタックがトップにあるときに呼び出し可能なメソッドが含まれている (プログラム 4.2).

```
1         \\ スタックを表すクラス
2         static class SE<Rest> {
3                 \\ スタックの状態
4                 Rest rest;
5                 \\  AST のデータ
6                 TreeNode tree;
7
8                 ...
9
10                \\ 次に呼び出され得るメソッド
11                ST<SE<Rest>> add() {
12                        ...
13                        TreeNode childT = new TreeNode();
14                        TreeNode childE = new TreeNode();
15                        ...
16                        \\ 一つ目のスタックをプッシュした状態を表すインスタンスE
17                        SE<Rest> E = new SE<>(this.rest, childE);
18                        \\ 二つ目のスタックをプッシュした状態を表すインスタンスT
19                        ST<SE<Rest>> T = new ST<>(E, childT);
20                        return T;
21                }
22
23                SF<ST<Rest>> multiple() { ... }
24
25                Rest i(Integer arg0) { ... }
26                ...
27        }
```

プログラム 4.2. 生成された fluent interface のクラス

各メソッドの返り値の型はプッシュ後のスタックの状態を表している. 非終端記号 E を表すスタックがトップにあるときにメソッド add() が呼び出されると, 図 4.2 のスタック操作をおこ

なう．これをクラス内で表現しており，トップのスタックがポップされ，メソッド add() の返り値の型で表現されるスタックがプッシュされる．

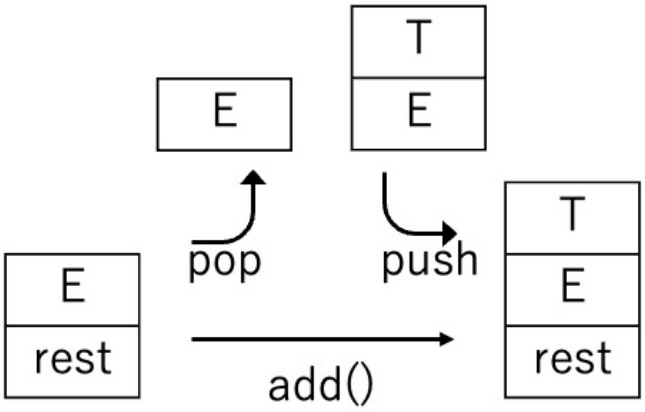

図 4.2. メソッド add() が呼ばれたときのスタックの操作

このときメソッド add() の返り値の型は新しくトップになったスタック T を表す型が一番外にあり，その下に続くスタックを表す型が型パラメータの中で表現されている．スタック T を表すクラスが返されると次はスタック T がポップされ，型パラメータだけが残り，これがこのクラスの Rest に該当する．スタックをプッシュした状態を表す変数の第一引数には，一つ前の状態を渡す．一つ目のスタックをプッシュした状態を表す変数には，rest をそのまま渡す．プログラム 4.2 では E, T をポップしたときの状態としてそれぞれ rest と E を渡している．このような型による文法の表現により，図 4.3 のように method chaining の入力途中で次に選択できるメソッドが表示される．図 4.4 のように直前で呼び出されたメソッド i() の型が SE のため，クラス SE が持つメソッド add(), multiple(), i() が次に呼べるメソッドの候補に表示されている．

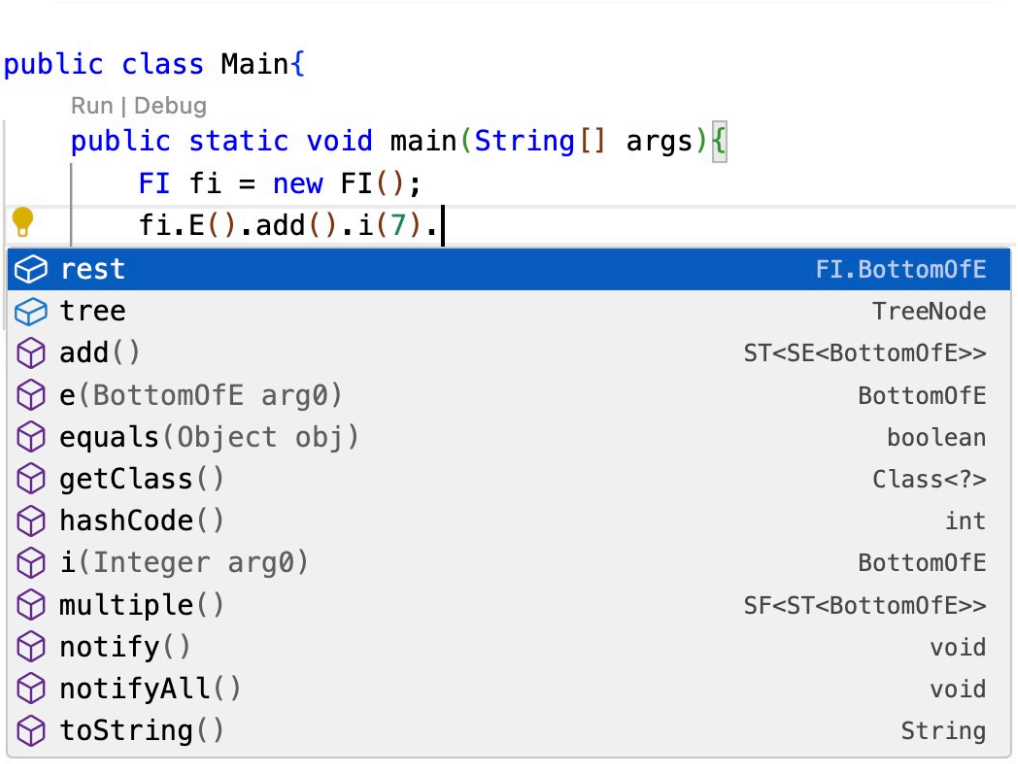

図 4.3. 補完候補のメソッドの表示

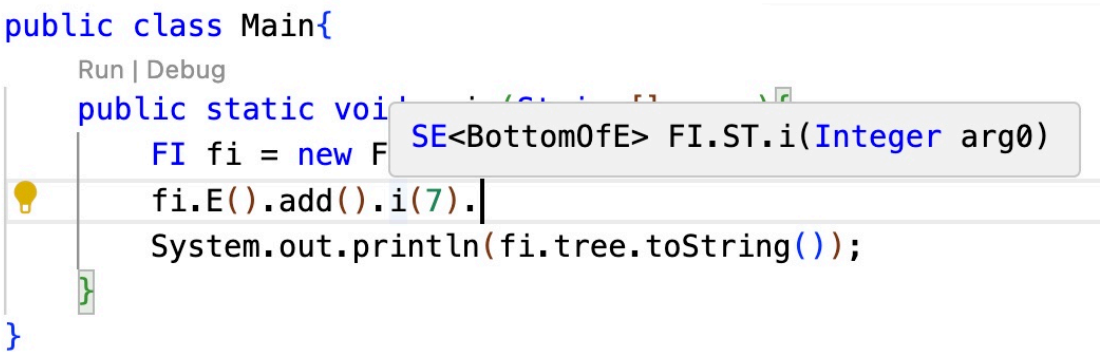

図 4.4. 図 4.3 におけるメソッド `i()` の型

各初期状態に対応するスタックの底を表すクラスでは, AST を返すメソッド `end()` が定義されている (プログラム 4.3).

```
1         class BottomOfE {
2                 \\  AST を返すメソッド end()
3                 TreeNode end() {
4                         return tree;
5                 }
6         }
```

プログラム 4.3. スタックの底を表すクラス

図 4.5 に, 最後から二番目のメソッドの型を示した. この型 ST<BottomOfE> は, 図 4.6 の左のスタックの状態を表している. この状態から, クラス T が持つメソッド `i()` を呼び出す.

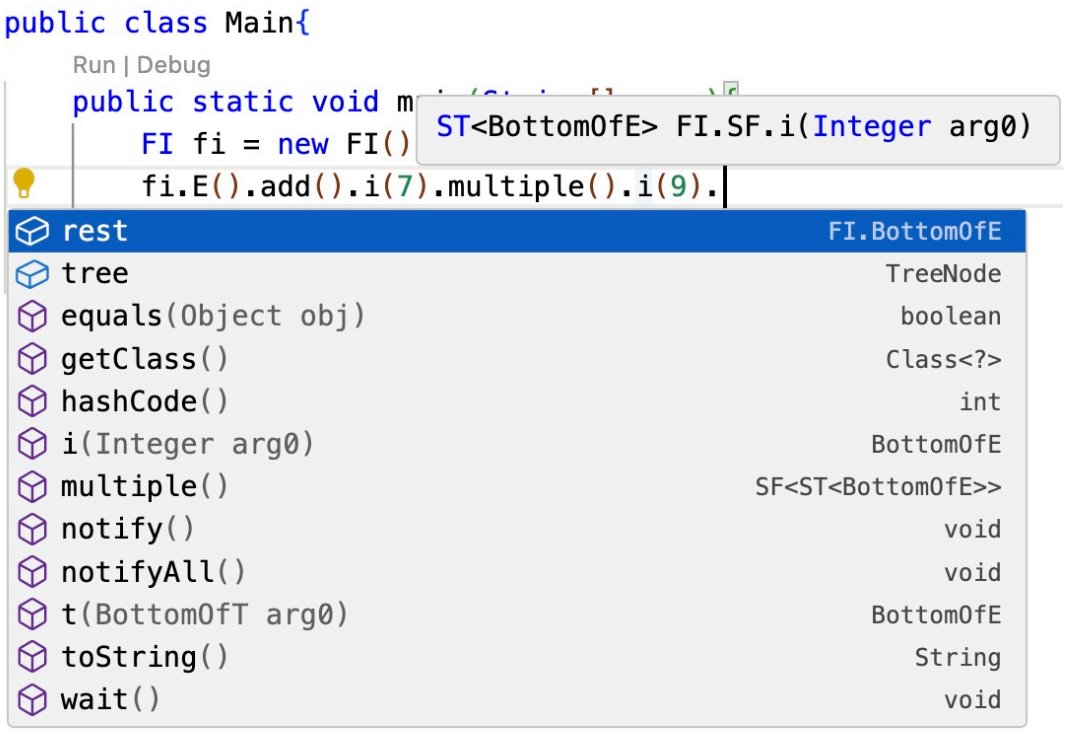

図 4.5. 最後から 3 番目のメソッドの型

プログラム 4.4 のように, このときスタックはプッシュされないため, 図 4.6 の右のスタックの状態になる. この状態は, スタックの底のみであり, method chain を終了できる状態である. 図 4.7 に示したように, この状態はメソッド `end()` を呼び出すことができる. Method chain

の最後に呼び出すメソッド end() は, スタックの底を表すクラスからしか呼び出せない.

```
1         static class ST<Rest> {
2                 ...
3
4                 \\ スタックはプッシュされない
5                 Rest i(Integer arg0) {
6                         ...
7                         return rest;
8                 }
9         }
```

プログラム 4.4. スタック T を表すクラス内のメソッド i()

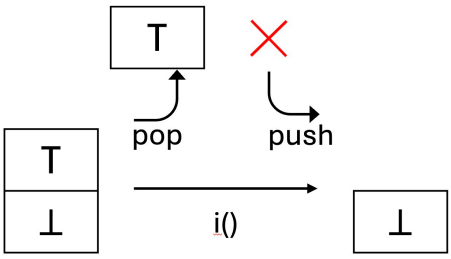

図 4.6. 最後のメソッド i() が呼び出されたときのスタック操作

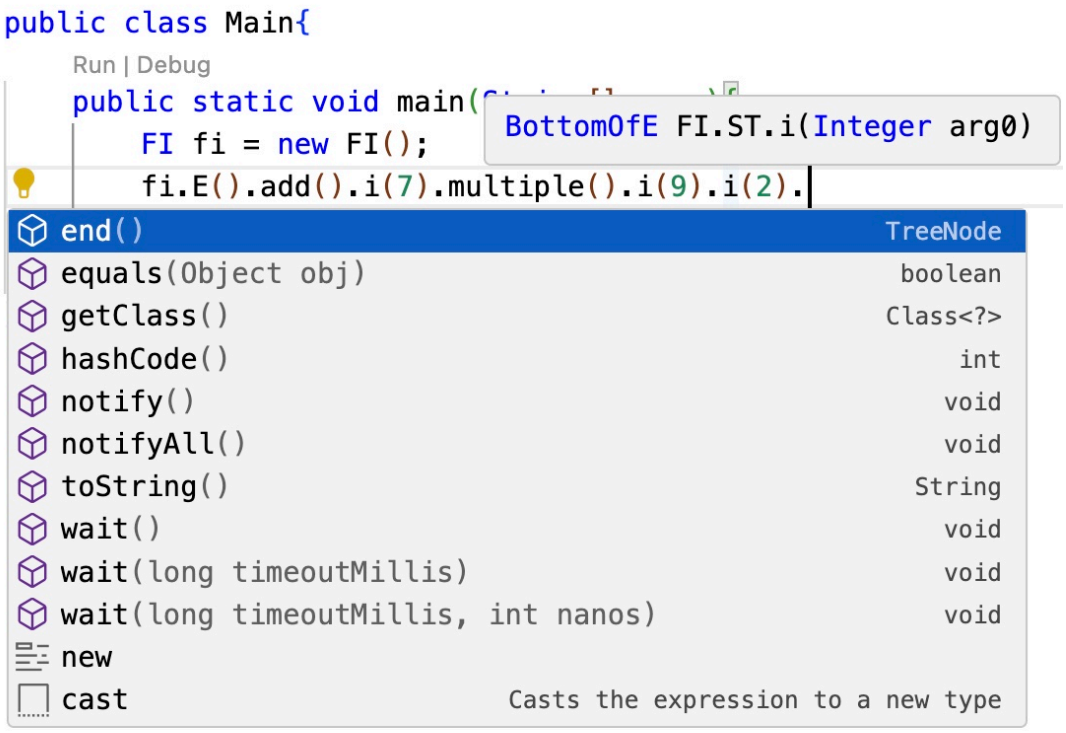

図 4.7. 最後から 2 番目のメソッドの型

初期状態を表すメソッドはプログラム 4.5 のように定義される.

```
1         SE<BottomOfE> E() {
2                 this.tree = new TreeNode();
3                 this.tree.s = "E";
4                 return new SE<BottomOfE>(new BottomOfE(), this.tree);
5         }
```

プログラム 4.5. 初期状態を表すメソッド

返り値の型はスタックの底に E がプッシュされた状態を表す. return するインスタンスの rest にはスタックの底のみの状態を渡す.

4.2　ε ルールを含む文法からの fluent interface 生成

プログラム 4.6 のような ε ルールを含む入力文法が与えられると, 生成規則が以下のように変換される.

```
1 fi.Lang
2 .rule().nt('S').arrow().nt('AA').nt('BB')\
3 .rule().nt('AA').arrow().t('a')\
4 .rule().nt('AA').arrow()\          # εルール
5 .rule().nt('BB').arrow().t('b')\
6 .start_from('S').end()
```

プログラム 4.6. ε ルールを含む入力文法の記述

(i) 入力文法
S → AA BB CC
AA → a | ε
BB → b
CC → c

(ii) ε ルールを除去した文法
S → S'
S' → AA' BB CC |BB CC
AA' → a
BB → b
CC → c

(iii) GNF に変換した文法
S → a BB CC |b CC
AA' → a
BB → b
CC → c

(iv) Sub-chaining の表現を埋め込んだ文法
S → a BB CC |b CC |s
AA' → a |aA'
BB → b |bB
CC → c |cC

最終的に得られた文法を GNF で書かれた LL(1) 文法から flat-chaining の fluent interface 生成器に入力して fluent interface を生成する．図 4.8 では生成した fluent interface を用いて method chaining を記述している．次に呼び出され得るメソッド `a()`, `b()` と非終端記号 S に対応する sub-chaining を表現するメソッド `s()` が補完候補に表示される．

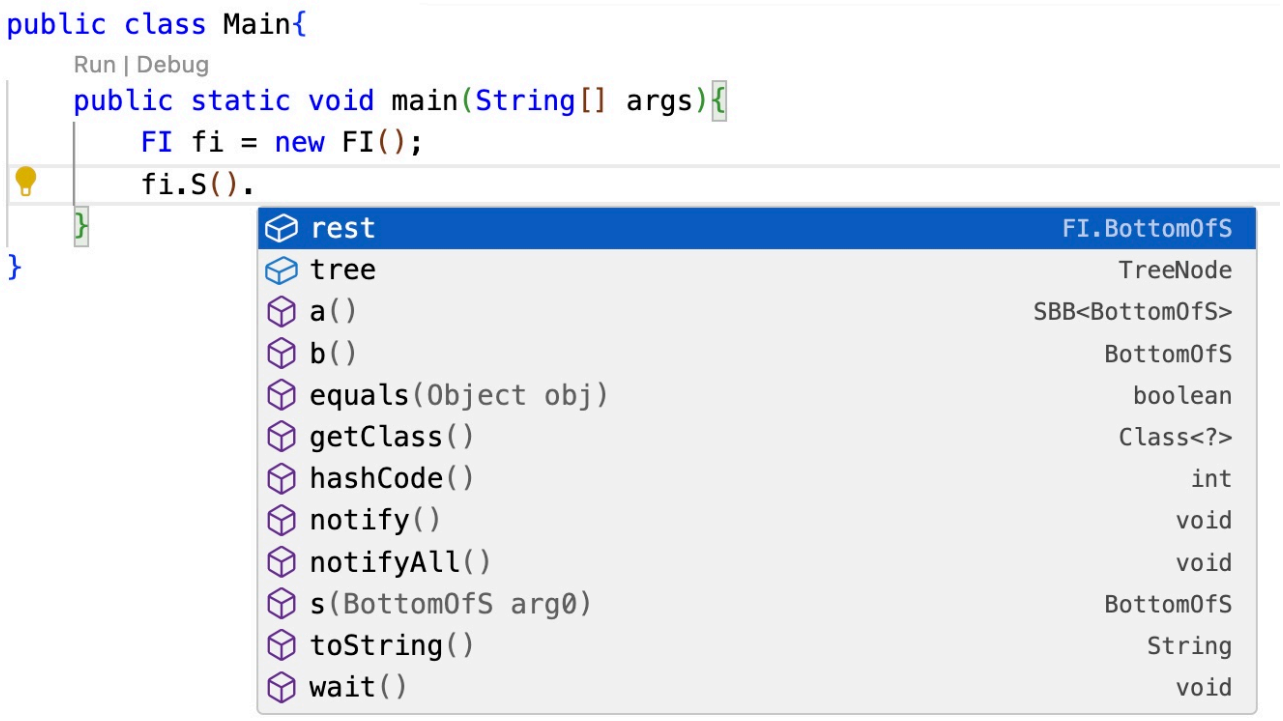

図 4.8. ε ルールを含む文法から生成した fluent interface による補完候補

4.3 Sub-chaining の fluent interface 生成

またプログラム 4.7 のような SQL を表す入力文法から，プログラム 4.8 のような sub-chaining の表現を含む fluent interface が生成される．

```
1 fi.Lang
2 .rule().nt('SelectQuery').arrow().t('selectFrom').nt('WhereClause')
3 .rule().nt('WhereClause').arrow().t('where')
4 .terminal('selectFrom').arg_type('String').arg_type('String')
5 .terminal('where').arg_type('String')
6 .start_from('SelectQuery').end()
```

プログラム 4.7. SQL を表現する入力文法の記述

```
1  class FI {
2          ...
3          \\ Subchain の初期状態に対応するスタックを表すメソッド
4          SSelectQuery<BottomOfSelectQuery> SelectQuery() { ... }
5          SWhereClause<BottomOfWhereClause> WhereClause() { ... }
6          ...
7
8          static class SSelectQuery<Rest> {
9                  ...
10
11                 SWhereClause<Rest> selectFrom(String arg0) { ... }
12
13                 \\ 非終端記号 SelectQuery に対応する sub-chaining
   表現のためのメソッド
14                 Rest selectQuery(FI.BottomOfSelectQuery arg0) {
15                         ...
16                         return rest;
17                 }
18         }
19
20         static class SWhereClause<Rest> {
21                 ...
22                 Rest where(String arg0) { ... }
23
24                 \\ 非終端記号 WhereClause に対応する sub-chaining
   表現のためのメソッド
25                 Rest whereClause(FI.BottomOfWhereClause arg0) {
26                         ...
27                         return rest;
28                 }
29         }
30 }
```

プログラム 4.8. Sub-chaining の表現を含む文法の fluent interface

この fluent interface のクラス FI のインスタンスは subchain の初期状態に対応する全てのスタックを表すメソッドをフィールドの値に持つ．Subchain を受け取るメソッドの引数の型は，引数となる subchain におけるスタックの底に対応する型である．非終端記号

SelectQuery，WhereClause に対応する sub-chaining 表現のための非終端記号に対応するメソッド selectQuery と whereClause が追加される．selectQuery が $SelectQuery_{subchain}$，whereClause が $WhereClause_{subchain}$ に対応する．プログラム 4.9 の Main クラスは生成された fluent interface の利用例である．

```
1 public class Main {
2         public static void main(String[] args) {
3                 FI fi = new FI();
4                 fi.SelectQuery().selectFrom("*").whereClause(
5                         new FI().WhereClause().where("-")
6                 ).end();
7         }
8 }
```

プログラム 4.9. Sub-chaining 表現を使う Main 関数

whereClause メソッドの引数が subchain になっている．sub-chaining の引数の型はプログラム 4.9 の定義と一致している (図 4.9).

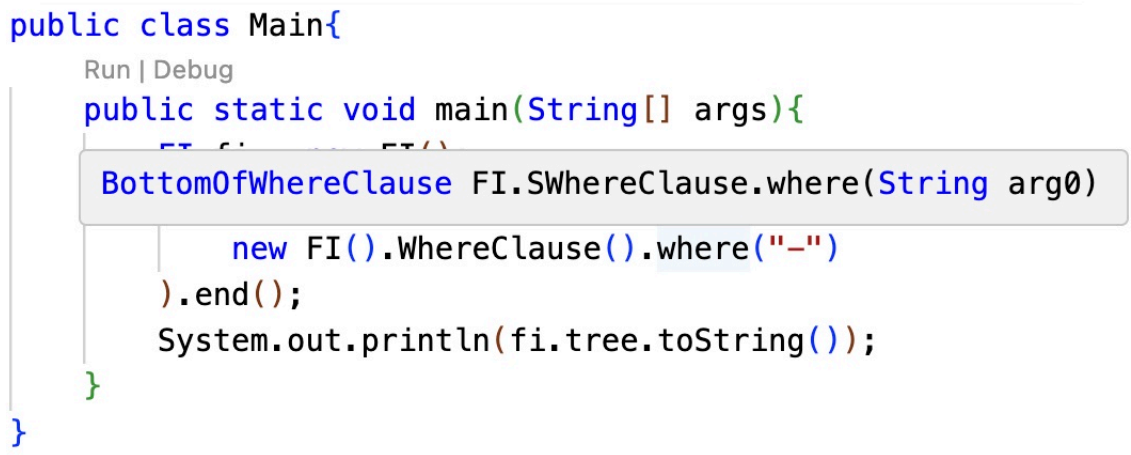

図 4.9. Sub-chaining の引数の型

4.4　入力文法が対応範囲外の場合

2.5 節で示した ε ルールを除去できない LL(1) 文法を入力すると, 以下のエラー文が出力されて処理が停止し, fluent interface は生成されない.

```
1 ERROR    :The input grammar includes production rules that end
2 with nullable symbol.
```

プログラム 4.10 で記述される文法は, 非終端記号 E からメソッド `add()` を呼び出したとき適用する生成規則が一意に定まらないため, LL(1) 文法ではない.

```
1  # 非終端記号 E からメソッド add() を読んだときに適用される生成規則が複数存在する入力文法
2  fi.Lang
3  .rule().nt('E').arrow().t('add').nt('T').nt('E')\
4  .rule().nt('E').arrow().t('add').t('j')\
5  .rule().nt('E').arrow().nt('T')\
6  .rule().nt('T').arrow().t('multiple').nt('F').nt('T')\
7  .rule().nt('T').arrow().nt('F')\
8  .rule().nt('F').arrow().t('i')\
9  .terminal('i').arg_type('Integer')\
10 .start_from('E').end()
```

プログラム 4.10. LL(1) 文法ではない入力文法の記述

このような文法を入力すると, 解析表を作成後に以下のエラー文が出力されて処理が停止する.

```
1 ERROR    :The input grammar is not LL(1) grammar.
```

Fluent interface は生成されない.

4.5　AST の構築

AST は, プログラム 4.11 で定義される TreeNode クラスを使って表す.

```
1  public class TreeNode {
2      \\ 子ノードの配列
3      public ArrayList<TreeNode> children = new ArrayList<TreeNode>();
4      public TreeNode parent = null; \\ 親ノード
5      \\ ノードの名前 ( 非終端記号に対応するノードは記号名または終端記号に対応する
   ノードはメソッド名 )
6      public String s;
7      public Object args; \\ 終端記号に対応するノードの引数
8
9      ...
10
11     \\ AST を出力するメソッド
12     public String toString() { ... }
13 }
```

プログラム 4.11. AST 構築に用いる TreeNode クラス

一般的な木構造を表すクラスに必要な機能を追加したものである. ノードの子の配列と親はそれぞれ変数 `children`, `parent`, メソッド名またはスタック名は変数 `s`, メソッドの引数は変数 `args` に格納される. 各メソッドが呼び出されるごとに, そのメソッドとプッシュされるスタックに対応するノードでを加えて AST を構築する.

4.5.1 GNF の LL(1) を flat-chaining で表現する例

Main 関数内などで生成された fluent interface にしたがって method chain を記述したあと, その method chain を入力文法にしたがって構文解析した AST を参照できる. 4.1 節で示した fluent interface を利用して method chain を表現したプログラム 4.12 の Main 関数を実行すると, 図 3.2 と同様の AST が得られる.

```
1  public class Main{
2          public static void main(String[] args){
3                  FI fi = new FI();
4                  fi.E().add().i(7).multiple().i(9).i(2).end();
5                  System.out.println(fi.tree.toString());
6          }
7  }
```

プログラム 4.12. Fluent interface を利用する Main 関数

この AST は, 初期状態を表すメソッド E() 内で初期化する (プログラム 4.13).

```
1         SE<BottomOfE> E() {
2                 \\ を初期化AST
3                 this.tree = new TreeNode();
4                 \\ スタックを表すクラスに引数として渡すE
5                 return new SE<BottomOfE>(new BottomOfE(), this.tree);
6         }
```

プログラム 4.13. 初期状態を表すメソッド

スタック E を表すクラスのコンストラクタ内でこのノードのデータが格納される (プログラム 4.14).

```
1         static class SE<Rest> {
2                 ...
3                 TreeNode tree;
4
5                 public SE(Rest rest, TreeNode tree) {
6                         ...
7                         this.tree = tree;
8                         this.tree.s = "E";
9                 }
10        }
```

プログラム 4.14. スタックを表すクラスのコンストラクタ内での AST データ格納

プログラム 4.15 に, 次に呼び出される, スタック E を表すクラス (プログラム 4.2) 内のメソッド add() の詳細を示した. メソッド add() およびプッシュされるスタック T, E を表すノードをプログラム 4.14 で格納した現在のノードの子としている.

```
1         static class SE<Rest> {
2                 ...
3
4                 ST<SE<Rest>> add() {
5                         \\ メソッド add() ( 終端記号 ) を表すノード
6                         TreeNode child = new TreeNode();
7                         child.s = "add";
8                         child.parent = this.tree;
9                         \\ 現在のノードの子に加える
10                        this.tree.children.add(child);
11                        \\ 非終端記号 T を表すノード
12                        TreeNode childT = new TreeNode();
13                        childT.parent = this.tree;
14                        \\ 現在のノードの子に加える
15                        this.tree.children.add(childT);
16                        \\ 非終端記号 E を表すノード
17                        TreeNode childE = new TreeNode();
18                        childE.parent = this.tree;
19                        \\ 現在のノードの子に加える
20                        this.tree.children.add(childE);
21                        SE<Rest> E = new SE<>(this.rest, childE);
22                        ST<SE<Rest>> T = new ST<>(E, childT);
23                        return T;
24                }
25        }
```

プログラム 4.15. メソッド内での AST 構築

ここまでの操作で, 図 3.2 の AST の上二段が構築される. これ以降のメソッド呼び出しでも同様の AST の操作を行い, 最後のメソッド end() が完成された図 3.2 の AST を返す.

4.5.2 Sub-chaining を表現する例

Sub-chaining を表現するプログラム 4.8 のメソッド whereClause() 内では, プログラム 4.16 のように AST を構築する.

```
1                   \\ 非終端記号 WhereClause に対応する sub-chaining
    表現に対応するメソッド
2                   Rest whereClause(FI.BottomOfWhereClause arg0) {
3                           \\ 引数の method chain の AST のルートの子
    に含まれるノードを，現在のノードの子に加える
4                           for (TreeNode child : arg0.end().children) {
5                                   child.parent = this.tree;
6                                   this.tree.children.add(child);
7                           }
8                           return rest;
9                   }
```

プログラム 4.16. Sub-chaining 表現の AST 構築

Sub-chaining の引数のメソッド `end()` の返り値は sub-chaining の AST である．これを現在のノードの子として渡す．プログラム 4.17 のような Main 関数を実行すると，同じ二つの図 4.10 のような AST が得られる．

```
1  public class Main{
2          public static void main(String[] args){
3                  FI fi = new FI();
4                  \\ SQL 文を flat-chaining で表現
5                  fi.SelectQuery().selectFrom("*").where("-").end();
6                  \\ AST を出力
7                  System.out.println(fi.tree.toString());
8                  \\ SQL 文を sub-chaining で表現
9                  fi.SelectQuery().selectFrom("*").whereClause(
10                         new FI().WhereClause().where("-")
11                 ).end();
12                 \\ AST を出力
13                 System.out.println(fi.tree.toString());
14         }
15 }
```

プログラム 4.17. Flat-chaining と sub-chaining を表現する Main 関数

同じ内容を flat-chaining で表現しても sub-chaining で表現しても，同様の AST が構築さ

れる.

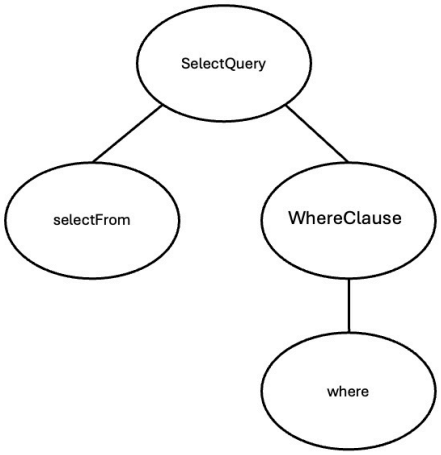

図 4.10. SQL を表現する method chaining を構文解析して得られる AST

4.5.3 入力文法が ε ルールを含む場合の AST

プログラム 4.18 は, ε ルールを含む入力文法である. この文法は ε ルールを除去できる文法である.

```
1 # 生成規則に ε ルール AA →ε を含む
2 fi.Lang
3 .rule().nt('S').arrow().nt('AA').nt('BB').nt('CC')\
4 .rule().nt('AA').arrow().t('a')\
5 .rule().nt('AA').arrow()\
6 .rule().nt('BB').arrow().t('b')\
7 .rule().nt('CC').arrow().t('c')\
8 .start_from('S').end()
```

プログラム 4.18. 除去できる ε ルールを含む入力文法の記述

この入力文法から生成された fluent interface を利用して表現した method chain を含むプログラム 4.19 の Main 関数を実行すると, 図 4.11 のような AST が構築される.

```
1  public class Main{
2          public static void main(String[] args){
3                  FI fi = new FI();
4                  \\ 生成規則 AA →ε が適用される    method chain
5                  fi.S().b().cC(
6                          new FI().CC().c()
7                  ).end();
8                  System.out.println(fi.tree.toString());
9          }
10 }
```

プログラム 4.19. ε ルールを含む文法から生成した fluent interface を使う Main 関数

ε ルール除去の変換は, ε に展開される非終端記号を表すノードがない点を除いて AST に影響はなく, AST で method chain の構文を表現する際には問題ないと考える.

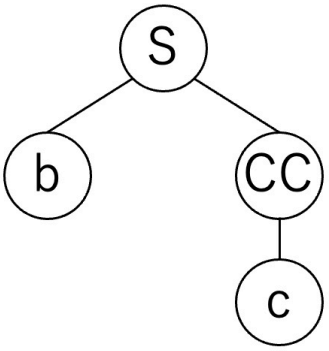

図 4.11. ε ルールを含む文法の表現を構文解析した AST

4.5.4　GNF への変換ときの中間スタックの保存

2.4 節で示した文法では, E → T → F → i という展開があり得るが, ssrDPDA ではトップにスタック E があるときにメソッド `i()` を呼ぶと非終端記号 T, F に対応するクラスを通らない. この場合の AST の構築は, プログラム 4.20 のようにおこなう.

```
1           static class SE<Rest> {
2                   ...
3
4                   Rest i(Integer arg0) {
5                           \\ 中間の非終端記号 T を表すノード
6                           TreeNode middleNodeT = new TreeNode();
7                           middleNodeT.s = "T";
8                           middleNodeT.parent = this.tree;
9                           \\ 現在のノードの子に加える
10                          this.tree.children.add(middleNodeT);
11                          \\ 中間の非終端記号 F を表すノード
12                          TreeNode middleNodeF = new TreeNode();
13                          middleNodeF.s = "F";
14                          middleNodeF.parent = middleNodeT;
15                          \\ 非終端記号 T を表すノードの子に加える
16                          middleNodeT.children.add(middleNodeF);
17                          \\ メソッド i() を表すノード
18                          TreeNode child = new TreeNode();
19                          child.s = "i";
20                          child.args = arg0;
21                          child.parent = middleNodeF;
22                          \\ 非終端記号 F を表すノードの子に加える
23                          middleNodeF.children.add(child);
24                          return rest;
25                  }
26  }
```

プログラム 4.20. GNF への変換ときの中間スタックの保存

展開の中間の非終端記号 T, F を表すノードを, 非終端記号 E を表すノードとメソッド `i()` を表すノードの間につなげる. プログラム 4.21 の Main 関数を実行すると, 図 4.12 のような中間のノードを復元した AST が得られる.

```
1 public class Main {
2         public static void main(String[] args) {
3                 FI fi = new FI();
4                 \\ 文法上の展開は E → T → F → i
5                 fi.E().i(10).end();
6         }
7 }
```

プログラム 4.21. 中間のスタックが発生する method chain を表現する Main 関数

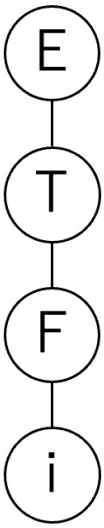

図 4.12. 中間のスタックを復元した AST

第 5 章

結論

5.1 まとめ

本論文は, Python 上の EDSL で BNF 風の文法を記述すると, flat-chaining と sub-chaining の両方を表現する Java の fluent interface を生成するツールを提案した．sub-chaining と flat-chaining を表現する元の入力文法が空列を生成する非終端記号を各生成規則の最後に含まない GNF で書かれた LL(1) なら，GNF で書かれた LL(1) に sub-chaining の表現を埋め込めることを示した.

GreenChain の EDSL は BNF 風の文法を記述するための言語で，GreenChain はこの記述から Java で書かれた fluent interface のを生成する．GreenChain では method chain を解析し, 抽象構文木 (Abstract Syntax Tree, AST) を構築する．入力文法に ε ルールが含まれる場合, nullable な記号とその後ろの non-nullable な記号を組み合わせて，新しい記号を作る．これにより ε ルールが除去できる．ただし，規則の右辺が nullable な記号で終わる場合には適用できない．続いて ε ルールなしの LL(1) 文法を GNF で書かれた LL(1) に変換する．規則の右辺の始めの記号を，終端記号になるまで展開し，ε ルールを含まない全ての LL(1) 文法を GNF に変換する．GNF で書かれた LL(1) に sub-chaining の表現を埋め込み, flat-chaining と sub-chaining の両方を表現する fluent interface を生成する．GNF で書かれた LL(1) から flat-chaining の fluent interface を生成する現実的なアルゴリズムを適用する．まず, 文法を対応するオートマトンに変換する．次に得られたオートマトンをもとに対応するクラス定義を生成する．これにより flat-chaining と sub-cahining の両方を表現できる fluent interface が生成される．Fluent interface を正しく生成できない可能性がある入力文法の場合は, 処理を中断する.

型による文法の表現により method chaining の入力途中で次に選択できるメソッドが表示される．各メソッド内で AST を構築する．method chain の最後に呼び出すメソッドから AST が得られる．入力文法が ε ルールを含む場合も ε ルール除去の変換は AST に影響はない．同じ内容を flat-chaining で表現しても sub-chaining で表現しても, 同様の AST が構築される．その際, 生成規則にしたがった文法展開でスキップされる非終端記号を表すノードを AST に追加する.

ε ルールを除去できない LL(1) 文法や LL(1) 文法ではない文法を入力すると, エラー文が出力されて処理が停止する. Fluent interface は生成されない.

このような LL(1) 文法から flat-chaining と sub-chaining の両方に対応する fluent interface を生成するアルゴリズムを提案した. また, LL(1) 文法のどの部分集合から正しい fluent interface を生成できるかを明らかにした.

5.2　今後の課題

より大きい文法クラスから sub-chaining にも対応できる fluent interface を生成する手法の提案が期待される. 一つの方法として新しい ε ルールの除去のアルゴリズムによって, GreenChain では扱えない文法にも対応できる可能性がある. また, 各メソッドが呼び出されたときに実行する処理を EDSL 設計者 (GreenChain 利用者) が書き込める雛形を作成することで, より実用性を高めることができる. そのためには, AST をさかのぼって処理できるようにする必要がある.

発表文献と研究活動

(1) 山隈由衣, 山崎徹郎, 千葉滋, LL(1) 文法から flat-chaining および sub-chaining の fluent interface を自動生成する DSL の提案にむけて. 日本ソフトウェア科学会第 41 回大会, 2024.09.11

参考文献

[1] Martin Fowler. Fluent interface, December 2005.

[2] A. M. Keshk and R. Dyer. Method chaining redux: An empirical study of method chaining in java, kotlin, and python. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pp. 546–557, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[3] Tomoki Nakamaru, Tomomasa Matsunaga, Tetsuro Yamazaki, Soramichi Akiyama, and Shigeru Chiba. An empirical study of method chaining in java. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, pp. 93–102, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, Vol. 28, Issue 4es, pp. 196–es, December 1996.

[5] jooq: The easiest way to write sql in java., 2009.

[6] Fast and fluent java html5 builder - java html builder., 2017.

[7] Shaula Yemini Robert Strom. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, Vol. SE-12, Issue: 1, pp. 157–171, January 1986.

[8] Hao Xu. Erilex: An embedded domain specific language generator. In Jan Vitek, editor, *Objects, Models, Components, Patterns*, pp. 192–212, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[9] Yossi Gil and Tomer Levy. Formal Language Recognition with the Java Type Checker. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, Vol. 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 10:1–10:27, Dagstuhl, Germany, 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[10] Yossi Gil and Ori Roth. Fling - a fluent api generator. *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, No. Article No. 13;, pp. pp. 13:1–13:25, 2019.

[11] Bruno Courcelle. On jump-deterministic pushdown automata. *Mathematical systems theory*, Vol. 11, pp. 87–109, December 1977.

[12] Radu Grigore. Java generics are turing complete. *POPL 2017 Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. POPL Principles of Programming Languages*, pp. pp. 73–85., 2016.

[13] Kazuhiro Ichikawa. Lalr parser generator for embedded dsls in scala., December 2016.

[14] EricBodden. Efficienthybridtypestateanalysisbydetermining continuation-equivalent states. *ICSE '10: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering*, Vol. 1, pp. 5–14, May 2010.

[15] Eric Bodden. Ts4j: A fluent interface for defining and com- puting typestate analyses. *SOAP '14: Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pp. 1–6, June 2014.

[16] Tomoki Nakamaru, Kazuhiro Ichikawa, Tetsuro Yamazaki, and Shigeru Chiba. Silverchain: a fluent api generator. *SIGPLAN Not.*, Vol. 52, No. 12, pp. 199–211, oct 2017.

[17] Tetsuro Yamazaki, Tomoki Nakamaru, and Shigeru Chiba. Yet another generating method of fluent interfaces supporting flat- and sub-chaining styles. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2022, pp. 249–259, New York, NY, USA, 2022. Association for Computing Machinery.

[18] Sheila A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, Vol. 12, No. 1, pp. 42–52, jan 1965.

[19] D.J. Rosenkrantz and R.E. Stearns. Properties of deterministic top-down grammars. *Information and Control*, Vol. 17, No. 3, pp. 226–256, 1970.

[20] P. M. Lewis. Syntax-directed transduction. *Journal of the ACM (JACM), Issue 3*, pp. 465–488, 1968.

謝辞

本研究を進めるにあたり, ご指導いただきました本学千葉滋教授, 本学山崎徹郎特任助教に心から感謝いたします. 毎週のミーティングではさまざまな視点から助言をいただきました. 論文執筆においても多大なご助力を賜りました. また, 2 年間の研究生活でご意見とご指導をいただきました本学鵜川始陽准教授, 本学千葉研究室の先輩方, 同期の皆様方に感謝いたします. 本当にありがとうございました.