開発環境も自動生成するパーサジェネレータ

長　　慎　也[†]　兼　宗　　進[†]

最近の開発環境は，プログラムを編集，コンパイル，実行するという，従来の言語処理系がもつ機能に加えて，コード補完，名称変更などの高度な支援機能を備えているものが多い．しかし，これらの支援機能は，言語ごとに個別に実装されており，利用できる機能がまちまちであるという問題がある．これらの支援機能を，自動的に生成する仕組みがあることが望ましい．一般に，言語処理系はパーサジェネレータを用いて作成されることが多い．このとき，言語開発者は構文やセマンティクスなどの言語仕様を記述する．ここで書かれた言語仕様をもとに，開発環境の支援機能までをも生成するパーサジェネレータ Azuki を提案する．

Azuki には，多数のプログラミング言語の間で再利用可能な支援機能のテンプレートが用意されており，言語開発者は，それらのテンプレートを選択し，言語仕様を入力として与えるだけで，支援機能を備えた開発環境を生成することができる．

Parser Generator which generates IDE

Shinya Cho[†] and Susumu Kanemune[†]

Recent develop environments has many sophisticated support features like code completion and renaming. However, these features have to be implemented by each programming language separately. These features should be generated automatically. Most language processor is generated by parser generator. Language developer defines the language specification like the syntax and semantics of the language. We propose the system named Azukiwhich generats language specification generates develop environments from the langauge specification. Azukiinclude feature templates which generates features for support programming. The only things language developer has to do are writing language specification and select feature templates.

1. はじめに

近年，パーサジェネレータなどの進歩により，自分でオリジナルのプログラミング言語（以下，自作言語）を作成することが容易になってきた．

一方 Eclipse Platform[1)](以下 Eclipse) に代表されるように，プログラム開発のための高度な機能を備えた開発環境が登場してきている．Eclipse の支援機能の代表的なものを**表 1** に挙げる．

また，Eclipse は，プラグインを作成・追加することで，新しいプログラミング言語の開発環境を提供できる仕組みをもっている．自作言語を開発した作者 (言語開発者) も，プラグインを作成すれば，Eclipse 上で，自分の言語の開発を行うことができ，高度な支援機能をも利用することが可能である．

ところが，プラグイン開発を行うには，言語そのものを開発するのに比べてさらに多くのプログラムを作る必要があり，手間がかかる作業である．

自作言語でなくとも，一般に使われれている様々なプログラミング言語においても，Eclipse で開発をしたいという要求は多く，実際多数の言語に対応したプラグインが開発されている．しかし，すべての言語で表 1 のような支援機能が使えるわけではない．次に，いくつかの言語での対応状況を示す．

- **Flash(ActionScript)** コード補完，名前変更などが可能．ただし，コンパイルは，1 個のファイルを保存すると再コンパイルを行うのに時間がかかる．エラーの修正はできない．
- **Haskell** コンパイル，実行が Eclipse 上から行えるが，支援機能の類はほとんど実装されていない．
- **Ruby**，**Perl**，**PHP** 型のない言語である関係上，コード補完は型が明らかな場合のみ，リファクタリングは，ユーザに変更箇所の確認を求める．Ruby のプラグインにおいては，型推論を行う機構を追加して対応する動きもある．

† 一橋大学
Hitotsubashi University

表 1　Eclipse の主な支援機能

構文エディタ	色づけ，インデント，対応する括弧の提示などを行う．
差分コンパイル	プログラムが変更されたときに，再度コンパイルをやりなおすべきモジュールの範囲を特定し，必要最低限の再コンパイルだけを行う．
名前変更	プログラムの意味を変えずに，クラス名，メソッド名などを変更する．
コードアシスト	メソッド名の補完候補などを提示する．
エラー修正	コンパイルエラーが起きたときに，修正候補を提示し，必要に応じて修正する．
検索	文字列検索だけでなく，クラスやメソッドの参照元，参照先を探し出す．
GUI 連携	プログラムの編集に応じて，クラスの構造を GUI に表示したり，逆に GUI の操作に応じて，プログラムを自動的に変更したりする．
ナビゲーション	フィールドやメソッドの呼び出し部分から，その定義部分を表示する．

- **Java** 表 1 のすべての機能をもつ．

例えば，Java の開発環境である JDT は，基本的なパッケージである org.eclipse.jdt.core だけでも 40 万行，15M バイト にわたる非常に巨大なプラグインである．これと同じプログラムを他の言語においても作成することは難しい．

このように，一般に普及している言語ですら，Eclipse におけるプラグイン（開発環境）の整備は十分でないといえる．さらに， 自作言語のように個人的作成された言語においては，プラグインを作成することすら困難といえる．

2. 既存の取り組み

筆者自身が作成したプログラミング言語において，その開発環境に関する課題を挙げる．

2.1 Tonyu System

Tonyu System[2] は，ゲーム開発のための言語と開発環境である．ここでは，GUI エディタ，コード補完についての課題を挙げる

- **GUI エディタ**

  Tonyu System は，ゲームの画面設計を行うための GUI エディタを備えている．本来ならば，ユーザが Tonyu で書いたプログラムを読み込み，GUI エディタに反映させ，GUI エディタで設計した画面設計の情報を Tonyu のプログラムに書き込むということを行いたいが，プログラムの内容と GUI の連携，特に GUI エディタで変更した部分をプログラムに反映させる機能は難しく，実装されていない．そこで，プログラムとは別の独自形式で画面設計の情報を保存しているが，これは一般のユーザには見えないようになっており，プログラムの動作の把握を難しくしている．

- **コード補完**

  Tonyu には**図 2** のようなメソッド名の補完機能を搭載しているが，すべてのメソッド名を提示してしまい，型が異なるなどの理由でその場に書くことができないメソッドも補完の対象として提示してしまう．これは，ユーザが現在書いているプログラムがどの部分で，その場所にある式がどの型かを判定するのが困難であるからである．

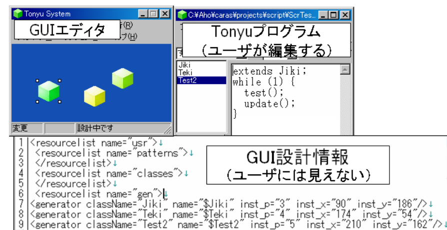

図 1　Tonyu の GUI エディタ

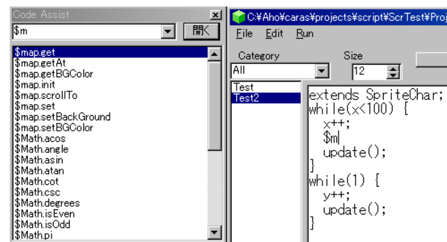

図 2　Tonyu のコード補完

2.2 Aroe

Aroe[3] は，Web ブラウザ上で Web アプリケーションが開発可能な言語とその環境である．ここでは，GUI エディタ，差分コンパイルについての課題を挙げる

- **GUI エディタ**

  Aroe は，Flash アプリケーションの開発機能をもち，Tonyu と同じく GUI エディタによる画面設計を支援する．ただし Tonyu とは異なり，画面設計の情報を Aroe プログラムとして保存している．しかし，GUI エディタの編集結果を，プログラムに出力する機能しかもたず，逆にプログラムを編集したと同時に，編集結果を画面に反映するような機能は実装できていない．

- **差分コンパイル**

  Aroe は，プログラムを書き換えると，AJax を用いてコンパイルを随時行えるような仕組みをもつ (**図 3**)．ただし，実装においては十分な速度が出ず，書き換え

にコンパイルが追いつかない状況が多く見られた．速度を上げるには，プログラムを書き換えたときに，必要最低限の箇所でコンパイルを行うようにする必要がある．このためにプログラム間の依存関係の解決に多数の補助ファイル（キャッシュなど）を確保して速度向上を図ったが，ファイルの管理が煩雑になった．

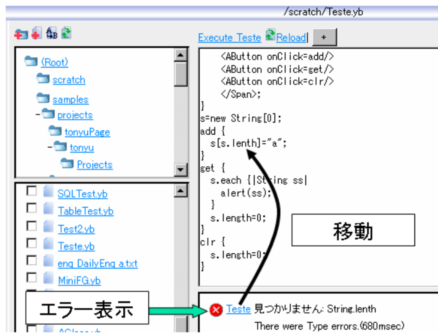

図 3　Aroe の逐次コンパイル

3.　パーサジェネレータ Azuki

前節にあげた機能の実装にまつわる問題点として，次のようなものがある．

- コード補完，差分コンパイルなどは，Java などの他の言語ではすでに実装ずみの支援機能であるのに，それらを流用できない．
- GUI エディタなどは，言語ごとに多少の差異はあっても，異なる言語間で共通に必要となる機能がある．それにもかかわらず，共通化がされていない．

このような問題を解決するため，次のような仕組みを提案する．

- ある言語で実装ずみの支援機能を，他の言語でも利用可能とする．
- 新しい支援機能を作るにあたり，そのために必要な基本的な機能を提供する．
- 支援機能を集めて，プラグインの形でまとめて生成する．

3.1　プラグインの概要

まず，プラグインを生成するにあたり，プラグインがどのような性質のものであるかを述べる．ここでは Eclipse のプラグインに限定して議論する．

Eclipse におけるプラグインとは，Eclipse 本体に追加可能なプログラムであり，Eclipse 自身を用いてプラグインプロジェクトという形で開発される．Eclipse 本体 は Java で記述されており，プラグインプロジェクトも Java のプログラムで構成される．

図 4 は，XML エディタのプラグインプロジェクトの例[4]である．このように，いくつかの Java のクラスから構成されており，各クラスは，色づけ，補完，アウトライン (GUI) 表示などの個別の支援機能に対応している．現状は，これらの機能をプラグイン開発者が手で作成している．

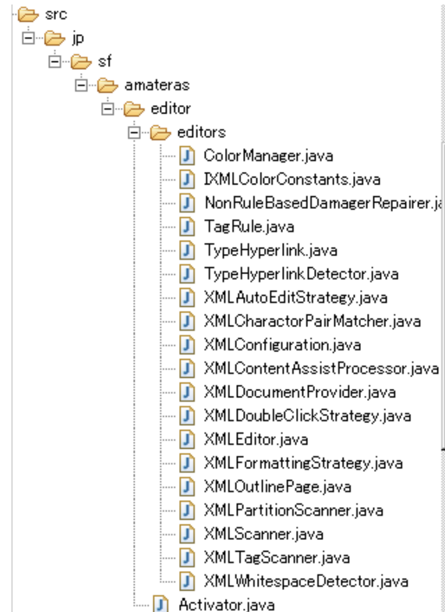

図 4　XML エディタのプラグインプロジェクト

本発表で扱う「プラグイン」は，特に，ある言語に対して次のような機能を実装した Eclipse のプラグインを指す．

- **エディタ** プログラムを編集する機能を最低限もつ．必要に応じて，予約語のハイライト表示や自動インデントなどの補助機能をもつ．
- **コンパイラ** 外部コンパイラ（プラグインプロジェクトに含まれないコンパイラ）を呼び出してコンパイルする機能を最低限もつ．必要に応じて（差分コンパイルを行う場合など），プラグインプロジェクト自身にコンパイラを含めることも可能である．
- **デバッガ** プログラムを実行する機能を最低限もつ．必要に応じて，プログラムを途中で停止したり，値を確認したりする機能をもつ．
- **その他支援機能** 必要に応じて，コード補完，リファクタリングなどの支援機能を提供する．

3.2　言語開発の概要

自作言語に限らず，プログラミング言語を実装する場合

には，パーサジェネレータを利用することが多い．パーサジェネレータは，その言語の構文や，その意味（セマンティクス）を定義すると，パーサを生成する．

従来のパーサジェネレータを用いた開発は**図 5** のようになる．

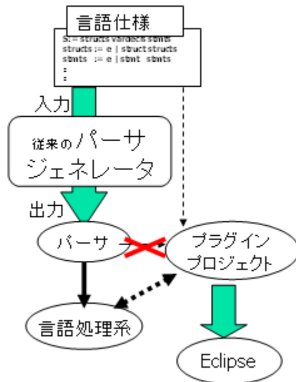

図 5　パーサジェネレータによる言語開発

- 言語開発者は，言語仕様を入力として与え，パーサジェネレータがパーサを出力する．
- パーサを使って，言語開発者が言語処理系を書く．
- 言語開発者はさらに，必要に応じてプラグインを開発する．
- 開発されたプラグインは，プラグインプロジェクトの形で Eclipse に取り込まれる．

プラグインを開発する場合は，3.1 の「最低限」のプラグインを開発するのであれば，外部の言語処理系を呼び出す機能だけを実装すればよい．しかし，より高度な機能（「必要に応じて」以降の機能）を必要とする場合は，それらの機能の追加を手で書いて行う．

ここで，プラグインに高度な機能を追加する場合には，しばしばもとの言語仕様を参照しながら実装を行うことになるが，このときパーサジェネレータが生成したパーサは役に立たない場合が多い．なぜなら，パーサは，与えられたプログラムを解析することはできても，プログラムを変更したり，補完を行ったりする機能をもっていないからである．

3.3　Azuki とは

そこで，**図 6** のように，従来のパーサの出力に加えて，プラグインプロジェクトも自動的に出力するようなパーサジェネレータを提案する．この仕組みを Azuki と呼ぶ．

プラグインプロジェクトは，様々な支援機能から構成されている．Azuki は，これらの支援機能を生成しなければならない．そのためには，支援機能を生成するためのプログラムが必要になる．これを「機能テンプレート」と呼ぶ．機能テンプレートは，言語開発者，すなわち Azuki のユーザが作成するものではなく，Azuki の開発者によって提供されるものであり，言語開発者によって言語仕様が与えられたときに，その仕様をもとに支援機能を生成するようにしておく．この仕組みにより，対象となるプログラミング言語によらず支援機能を生成することができる．

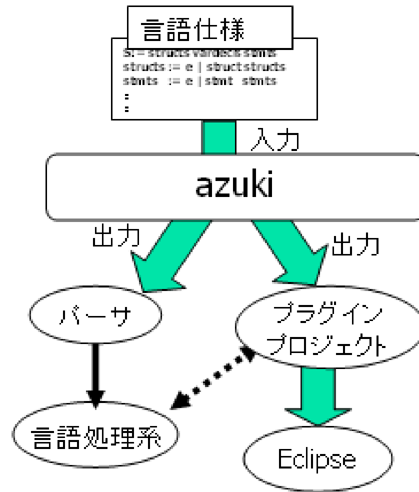

図 6　azuki による言語開発

4.　Azuki の構成

Azuki の構成を**図 7** に示す．

- **言語仕様** 4.2 に定める書式にしたがい，言語開発者が記述する．
- **機能テンプレート** 支援機能を生成するプログラムの集まり．言語仕様にもとづき，プラグインプロジェクトを構成する Java ソースファイルを生成する．
- **パーサ** 構文解析，意味解析の結果を取得する機能をもつ．プラグインの開発とは独立に言語処理系を開発したい場合に用いる．
- **プラグインプロジェクト** Eclipse に対してプラグインの追加を行う．

4.1　開発の手順

Azuki を用いたプラグインプロジェクトの作成の概要は次のようになる．なお，ここでは言語開発者は言語だけでなく，プラグインも自分で開発するものとする．

- 言語開発者は，言語仕様を記述する

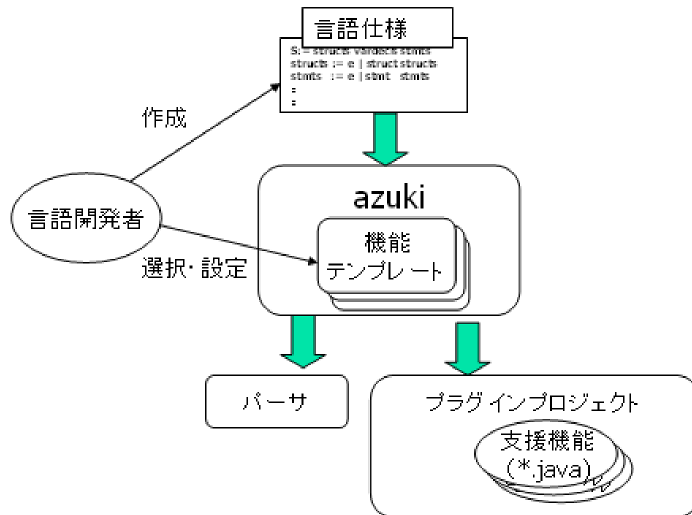

図 7　Azuki の構成

- 言語開発者は，必要な機能テンプレートを選ぶ．機能テンプレートによっては，設定項目（パラメタ）を入力する必要がある．
- Azuki によってパーサとプラグインプロジェクトが生成される．プラグインプロジェクトには，選択した支援機能を実装した Java のソースファイルが含まれている．
- 必要に応じて，言語開発者は生成されたプラグインプロジェクトにさらに自作のプログラムを追加する．

最後の，自作のプラグラムの追加が必要な例には，Tonyu の GUI エディタのように，言語やライブラリに依存する部分，特にグラフィックス表示などのユーザインタフェース部分について言語開発者が機能を手で追加する場合が挙げられる．

4.2 言語定義

ここでは，Azuki を利用する言語開発者が書くべき言語仕様の内容について議論し，そのあと言語仕様の具体的な書き方（言語仕様の言語仕様）を説明する．

- **字句，構文が定義できること**
  まず，パーサジェネレータであるからには，従来のパーサジェネレータがもつ，字句要素，構文要素の記述は当然必要である．後述する TOKENS，GRAMMARS セクションがこれらを定義する．
- **意味を検証できること**
  プラグインにコンパイラを含める場合などには，構文を把握するだけでなく，型やメソッドなどがどのように宣言されているか（言語要素の構成）や，入力されたプログラムが意味的に正しいプログラムか（意味的正しさ）も必要になる．
  そこで，言語仕様の記述には属性文法[5] を採用している．属性文法は，構文に属性を付与させ，それらの属性が満たすべき関係を定義することができる．この仕組みによって，プログラムが意味的に正しいかどうかも検証できる．GRAMMAR セクションには，構文のほかにその構文がもつ属性を定義できる．
- **プログラムが組めること**
  プラグインにコンパイラを含める場合には，コード生成，VM 生成など，高度なプログラミングが求められる．そこで，言語仕様には，RULES というセクションを設け，ここに簡約規則，関数定義を書けるようにしておく．これにより，一種の関数型プログラミングの機能を提供する．
  関数型にしておくことで，「入力の一部が変更されたときに，変更が起きる部分の出力だけを再計算する」仕組み[6] が実装しやすくなり，差分コンパイル機能を生成する場合に有用である．

これらを踏まえ，言語仕様の記述には TOKENS, GRAMMARS, RULES という 3 つのセクションを設けた．次はこれらの書式について説明する．

4.2.1 TOKENS

非終端記号 (字句要素) を定義する．なお，予約語など，内容が一意である字句要素は [GRAMMARS] セクションで直

接書くことも可能である．字句要素の定義を次の形式で並べて書く．

TOKENS := 字句要素定義*
字句要素定義 := 字句要素名 ’:=’ 正規表現
字句要素名 := （英大文字からなる文字列）

4.2.2 GRAMMARS

構文と，構文から生成された構文木がもつ意味式の性質を次の形式で定義する．

GRAMMARS := 構文*
構文 := 構文名 ’:=’ 意味構文 (’|’ 意味構文)*
構文名 := シンボル
意味構文 := 構文要素+ (改行インデント) [意味式 *]
意味式 := 属性 | 制約式
制約式 := 意味素 [’=’ 意味素]
属性 := 意味素 [’=’ 意味素] [where 制約式]
意味素 := 字句要素名 | シンボル | 関数 | 構文素
構文素 := ’$’ シンボル
関数 := シンボル ’(’ 意味素 (’,’ 意味素)* ’)’
シンボル := （英字からなる文字列）

ソースファイル中で，構文に一致する部分を見つけると，構文木が生成される．また，構文に意味式が定義されている場合は，意味式が生成され，属性データベースに格納される．

意味式の定義中に constraint: が書いてある場合，それに続く式を 制約式として解釈する．後述の意味チェッカによって，その制約式を満たすかどうかが検証される．

4.2.3 RULES

意味式の計算を補助するための簡約規則を定義する．

RULES := 簡約規則*
簡約規則 := 意味素 ’=’ 意味素 [where 制約式]

簡約規則の左辺 から右辺 への簡約を行うことで意味式の計算を行う．また，where 句を伴う場合，where 以降を満たすような値を，属性データベースに蓄積された意味式から検索することによって求める．

Azuki によって定義された簡単な言語の例を**図 8** に示す．この言語を便宜上 Smpl-lang と呼ぶことにする．

```
[TOKENS]

NAME := [a-zA-Z]+
LITERAL := "[a-zA-Z ]+"
DIGITS := [0-9]+

[GRAMMARS]

S:= structs vardecls stmts ;
structs := e | struct structs ;
stmts   := e | stmt   stmts ;
struct := 'type' NAME '{' vardecls '}'
   isa($struct, Type)
   nameOf($struct) = NAME

vardecls := e | vardecl vardecls;
vardecl  := 'var' NAME 'as' type
   isa ($vardecl, Var)
   typeOf ($vardecl) = $type
   nameOf ($vardecl) = NAME
   ownerOf ($vardecl) = outer(Type)

stmt := var '=' val
   constraint:
      typeOf($var) =  typeOf($val)

var := single | dot

single := NAME
  $single = varRef(undef, NAME)

dot := val '.' NAME
  $dot = varRef($val, NAME)

val := varRead | num | str;

varRead := var
   $varRead = valueOf($var)

str :=  LITERAL
  typeOf($str)= typeByName(String)

num := DIGITS
  typeOf($num) = typeByName(int)

type := NAME
  $type = typeByName(NAME)

[RULES]
typeOf(valueOf(v)) = typeOf(v)
typeOf(undef) = undef
typeOf(varRef(t,n))= var
   where
     isa(var,Var) && nameOf(var)=n
     && ownerOf(var)=typeOf(t)
typeByName(n) = t
   where isa(t,Type) && nameOf(t)=n
```

図 8 Azuki による言語定義

5. 機能テンプレートの例

機能テンプレートには，大きくわけて 2 つの種類がある．

- **単独テンプレート** 「自動生成が簡単な」支援機能を生成する．
- **補助テンプレート** 自動生成が簡単ではない，高度な支援機能の実装を手助けするライブラリを生成する．

「自動生成が簡単な」支援機能には，エディタに色をつける，名前を変更するなどの機能がある（名前の変更が自動生成可能なことは，5.2.3 で示す）．一方，Tonyu の GUI エディ

```
type int {}
type String {}
type Coord {
   var x as int
   var y as int
}
type Equa {
  var x as int
  var expr as String
}
var c as Coord
var e as Equa
c.x=3
c.y=5
e.x=c.x
e.expr="abc"
```

図 9 サンプルプログラム

タなどは，インタフェース部分までをも自動生成するのは困難であるが，補助テンプレートを利用することによって，エディタからソースコードへの反映を行うなどの，必要な機能を簡単に実装できるようになる．

ここでは，具体的な機能テンプレートの例と，その動作原理を示す．

5.1 補助テンプレート

5.1.1 推論エンジン

推論エンジンは，プログラムの意味内容を扱う上での必須の機能を提供する．

- **構文の属性のデータベース化**
  解析されたソースコードの構文木に付与されたすべての属性を，属性データベースと呼ばれる単一のデータベースに格納する．
- **データベースの検索**
  他の支援機能の要求に応じて，格納された属性を検索する．GRAMMAR，RULES セクションに登場する「`where` 制約式」は，属性データベースの中の属性を検索することで，指定された制約式に合う値を見つけ出す．
- **推論**
  GRAMMARS, RULES に書かれている制約式を満たすかどうかを，属性データベースに格納された属性，および，RULES に書かれた簡約規則から求める．また，制約式を充足させる過程を保持しておくことが可能で，名前変更や，差分コンパイルを行う際に有用な情報となる．

5.1.2 意味チェッカ

プログラムの意味的正しさを検証するのに用いる．属性データベースの中に格納された属性のうち，制約式であるものについて，それらを充足するかを検証する．検証には推論エンジンを用いるため，検証の過程を保持することが可能である．

5.1.3 構文編集機構

解析ずみのプログラムの構文木を変更する機能を提供する．構文木を変更すると，もとのソースコードの内容もそろって変更される．

5.1.4 探索機構

構文的，意味的に誤っているソースコードを与えると，そのソースコードを正しくする操作として，考えられるものを探索する．後述のコード補完，エラー修正で用いられる．

5.2 単独テンプレート

5.2.1 コード補完

コード補完は書きかけのコード（つまり，構文的に誤っているコード）に対して，メソッド呼び出しなどを補って，構文的，意味的に正しいコードにする操作をプログラマに提示し，プログラマの選択に応じて実行する機能である．この際にどんなメソッド名ならば追加することが可能か，ということを探索機構に問い合わせることによって実現する．

5.2.2 エラー修正

エラー修正は，意味的に正しくないコード，例えば，メソッド呼び出しに書かれているメソッドが存在しなかったり，引数の仕様が違ったりするコードに対して，正しいプログラムに修正するための候補をプログラマに提示し，プログラマの選択に従って実行する機能である．メソッドの追加，削除，引数仕様変更などを行って，意味的に正しいプログラムに書き換えるための操作を探索機構に問い合わせる．

5.2.3 名前変更

変数，メソッド，型などの名称を変更しつつ，かつプログラムの意味を変わることがないようにする機能である．

名前変更を行う際には，名前を変更すべき部分がプログラム上のどの部分になるかを特定しなければならない．これを，内部的に「シンボルの紐付け」と呼ぶ．シンボルの紐付けは，構文に記述された属性を用いて，型の解決（ある変数の型が何で，どのクラスで宣言されているかなど）をする際に，属性間のユニフィケーションを行う過程において行われ，紐付けされたもの同士は，名前変更の際にそろって名前が変わる．

現時点では，名前変更モジュールが実装されている．例えば，Smpl-lang で書かれた**図 9** のプログラムにおいて，型 `Coord` の変数 `x` を `left` に変更する，という指令を名前変更機能に与えると，**図 10** のようなプログラムに置き換わる．図 8 に示した Smpl-lang の言語仕様には，名前変更をどう行うかについては何も書かれていないが，名前変更モジュールをによってその機能が自動的に実装されていることが確認できた．

```
type int {}
type String {}
type Coord {
   var left as int
   var y as int
}
type Equa {
  var x as int
  var expr as String
}
var c as Coord
var e as Equa
c.left=3
c.y=5
e.x=c.left
e.expr="abc"
```

図 10 サンプルプログラム (名前変更後)

6. まとめ

プログラミング言語の開発環境である Eclipse におけるプラグインを，言語仕様を与えるだけで自動生成するパーサジェネレータ Azuki を提案した．現段階の実装においては，与えられた言語仕様に対して，名前変更の機能を自動生成できることを確認した．今後，さらに自動生成可能な支援機能を増やし，Java などの普及した言語と同等レベルの支援機能を，個人的に作成した自作言語においても手軽に利用できる環境を構築していく予定である．

参考文献

1) : Eclipse Platform, http://eclipse.org/.
2) 長慎也 : Tonyu System, http://tonyu.jp/.
3) 長慎也 : Aroe - AJaX-like Rapid Operating Environment, http://aroe.jp/.
4) 竹添直樹 : Eclipse プラグイン開発徹底攻略，毎日コミュニケーションズ (2007).
5) Paakki, J.: Attribute grammar paradigms - a high-level methodology in language implementation, *ACM Comput. Surv.*, Vol. 27, No. 2, pp. 196–255 (1995).
6) Acar, U.A., Blelloch, G.E. and Harper, R.: Adaptive functional programming, *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, ACM, pp.247–259 (2002).