Java 用開発ツールを利用した COBOL 開発環境の改善手法

野村総合研究所
生産技術革新推進部
主任テクニカルエンジニア

長谷川 勇(はせがわ いさむ)

専門はソフトウェア開発ツールの設計・開発。最近は静的解析を用いてソフトウェア開発の品質・生産性向上を目指した研究開発活動や実プロジェクトでの技術支援を行っている。

要旨

近年のツールの需要増加に対して、開発現場へのツールの供給は十分ではない。特に、Java 言語の開発ツールは世の中にあふれている一方で、COBOL 言語の開発ツールはごくわずかである。本稿では、ツールの供給不足の問題を解決するために、純粋構文木による構文解析器の共通化、及び構文木間での木構造の変換を用いることで、Java 言語の開発ツールを COBOL 言語の開発に適用可能にする手法を提案する。本手法では、従来開発が困難であった構文解析器の開発を、純粋な構文解析器の再利用と、一般的な手法による木構造の変換に置き換えることで開発の難易度を下げ、ツールのそのほかの部分をそのまま再利用することで開発工数を 20%以下に抑えることができる。また、実際にオープンソースソフトウェアの Java 言語の開発ツールに本手法を適用して COBOL 言語の開発ツールの開発に適用した実験結果からその有効性を示す。

キーワード：開発ツール、Java、COBOL、純粋構文木、構文解析器

Despite increasing needs for development tools in recent years, supply of the tools to developers is not enough. Especially for COBOL language, development tools are few, while those for Java language can be found everywhere. In this article, in order to solve the insufficiency of tool supply, I will propose a method, that is composed of standardizing parser by utilizing pure syntax tree and transforming tree structure between syntax trees, by which development tools for Java language are applicable to development using COBOL language. This method will decrease the man hour needed to develop a development tool to less than 20 %, by decreasing the difficulty level of development by replacing the development of parser which is usually difficult with reusing pure parser and transforming tree structure by general method, and by reusing the rest of the original tool. This article also indicates its effectiveness by the result of an experiment of developing a development tool for COBOL language using an open source development tool for Java language and applying the method to it.

Keyword : development tools, Java, COBOL, pure syntax tree, parser

1．はじめに

近年、ソフトウェアシステムの巨大化、多様化、高機能化に伴い、プログラムの開発はいっそう困難になってきている。このため、プログラム開発を支援する開発ツールの重要度はますます増している。一方、ツールの需要増加に対して、その供給は十分とは言えない。これはプログラミング言語の多様化によるツール需要の増加や、構文解析器の開発が困難であるためにツール開発の難易度が上がっていることなどの要因がある。特に、金融系基幹システムで依然主要言語の1つであるCOBOL言語の開発ツールはコンパイラベンダー製のツールしかない場合も多く、開発者の要求を満たせていない。このため、今でも汎用的なテキストエディタでコードを書き、コマンドラインからコンパイルし、デバッグメッセージ出力用のコードを埋め込むことでデバッグを行う旧来の手法のままの開発現場もあり、生産性向上を阻害する要因となっている。

我々の目的はツールの供給不足を解決することであり、その手段として、本稿では開発ツールが十分に整備されているJava言語のツールを、ツールの供給が不足しているCOBOL言語に適用する手法を提案する。本手法を採用することで、困難な構文解析器の開発を行うことなくツールを供給できるため、ツール供給不足の問題を解決することができる。

本稿では、まず、2節において本論文の前提知識となる構文解析などの用語に関する説明とともに、構文解析器の開発が困難である点を中心にツール供給不足の原因について述べる。3節では、ツール供給不足の問題を解決するための手法を提案する。まず、我々のアプローチである、構文解析器の開発において、構文木の構築と抽象構文木の構築を明確に分離し、更に構文木間での変換を行うことで、あるプログラミング言語のツールをほかの言語に適用可能にする手法を提案する。次に、構文木間での変換の具体例として、COBOL言語からJava言語への変換例を示す。最後に、4節では、我々が本手法を用いて評価実験を行った結果を述べ、関連手法・研究との比較により評価を行う。

2．開発ツール供給の課題

本節では、開発ツール供給の課題として、開発ツールの需要増大に対して供給が不足している問題を述べる。はじめに、前提知識として、構文解析などの用語に関する簡単な解説を行い、その後で開発ツールの供給が不足する原因を述べる。

(1) 前提知識

① 構文解析

構文解析とは、ある文章に対し、その文章が属する言語において規定された文法に基づき、その文章の構造を解析することである。

特に、計算機科学の世界では、構造が規定された形式言語（プログラミング言語やXMLなど）を対象とすることが多い。形式言語を規定する文法を形式文法と呼ぶ。形式文法はその記述力によって幾つかのクラスに分類することができるが、プログラミング言語は文脈自由文法と呼ばれるクラスの形式文法で定義されることが多い。

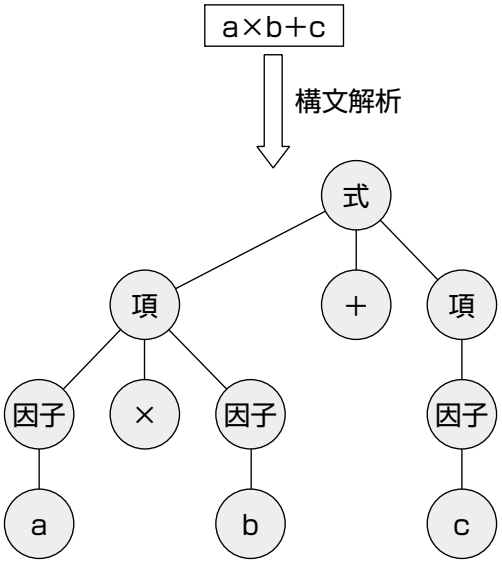

図1　構文木の例

また、構文解析の結果としては、文章の構造を木構造で表現した構文木や抽象構文木（Abstract Syntax Tree：AST）といったデータ構造を使用することが多い。図1に構文解析と出力の構文木の例を示す。この例では言語は「四則演算を持つ数式」であり、文章は「個別の式」である。この言語の文法は次の3つのルールから成る（通常、文法の定義は厳密性のために後述するBNFを使って記述するが、ここでは説明のため文章で記述する）。

- 「式」は単一の「項」、または「項」の加減算から成る。
- 「項」は単一の「因子」、または「因子」の乗除算から成る。
- 「因子」はa、b、c……である。

図1の場合、a×b＋cという「式」はa×bとcという2つの「項」の加算から成り、a×bという「項」はaとbという2つの「因子」の乗算から成る。この構造を表すのが図1の構文木である。

構文解析は、典型的にはコンパイラの実装に使用されるが、それ以外のツールにおいても構文依存の処理（例えば、変数のスコープ解決など）には構文解析が必要である。

② BNF・EBNF

前述の通り、通常文法を定義するときはBNF（Backus Naur Form：バッカスナウア記法）やEBNF（Extended Backus Naur Form：拡張バッカスナウア記法）を使用する。図2にBNFとEBNFを使い、図1の文法を記述した例を示す（ただし、BNFでは繰り返しの記述に再帰が必要なため、文法を多少

修正している)。BNFにおいて“→”記号は定義を意味しており、“→”の左に書かれた記号の定義を“→”の右に書く。また、“｜”はorを意味している。例えば、BNFの例の1行目は「式」を定義しており、「式」は「項」か「項“＋”式」か「項“－”式」のいずれかであるという定義である。

EBNFはBNFに幾つかの拡張を行い、記述しやすくした表記法である。例えば、「*」は「0回以上の繰り返し」を意味するため、EBNFの例の1行目の「式」の定義は、「項」に「“＋”項」か「“－”項」を0個以上つなげたものという意味である。

なお、EBNFはあくまで記述を簡単にするためのものであり、その記述力はBNFと等価である(つまり、EBNFで記述可能な文法は、必ずBNFで等価な文法が記述できる)。また、EBNFには処理系によって表記法が異なるという問題がある。

```
式 → 項 ｜ 項 “+” 式 ｜ 項 “−” 式
項 → 因子 ｜ 因子 “×” 項
      ｜ 因子 “÷” 項
因子 → “a” ｜ “b” ｜ “c” ｜ ・・・
```

BNF

```
式 → 項 ( “+” 項 ｜ “−” 項 ) *
項 → 因子 ( “×” 因子 ｜ “÷” 因子 ) *
因子 → “a” ｜ “b” ｜ “c” ｜ ・・・
```

EBNF

図2　BNF・EBNFの例

③ 構文解析器

構文解析器とは構文解析を行うプログラムのことであり、パーサー(parser)とも呼ばれる。

文脈自由文法に対する構文解析器としては、LL法、LALR法など、効率的なアルゴリズムが研究されており(これが多くのプログラミング言語が文脈自由文法で定義される理由の1つである)、それらを使用する構文解析器をそれぞれLL構文解析器、LALR構文解析器と呼ぶ。また、LL法やLALR法で構文解析できる文法をLL(k)文法、LALR(k)文法と呼ぶ。

ここで注意する点として、LL(k)文法やLALR(k)文法はどちらも文脈自由文法のサブセットであり、LL構文解析器もLALR構文解析器もすべての文脈自由文法を解釈できるわけではなく、更にそれらの構文解析器の能力もお互いに等価ではない。このため、「LALR法では構文解析できるがLL法ではできない文法」や「LALR法では構文解析の効率が悪い文法」といったものも存在する。このため、プログラミング言語の構文解析器を開発する場合、採用するアルゴリズムで記述できるように、プログラミング言語の文法を等価な別の形に変更する必要があり、構文解析器の開発を困難にする要因となる。

④ 構文解析器生成系

コンパイラなどの構文解析が必要なプログラムを開発する場合、構文解析器生成系を使

用することが多い。構文解析器生成系（パーサー・ジェネレータ、またはコンパイラ・コンパイラとも呼ばれる）とは、文法を入力として、その文法を解析する構文解析器を自動生成するプログラムである。

構文解析器生成系の入力は、形式はそれぞれ異なるものの、基本的にはEBNFによる文法の定義と、各文法規則に埋め込まれるロジックである。なお、ロジックが必要なのは構文解析結果である抽象構文木を構築するためや、エラーリカバリのためである。

（2）ツールの需要増加

近年のプログラミング言語の増加、特にRubyやPythonなどの軽量プログラミング言語の台頭により、開発現場で使用するプログラミング言語の種類は増加している。このため、使用する言語ごとにツールが必要となり、ツールの需要が増大する原因となっている。更にプログラミング言語の多様化に伴い、全体では需要が増加していても、ある言語単体で見るとツールの需要が相対的に下がっているケースもある。このため、ツールベンダーは需要の大きい言語のツール開発へシフトする傾向があり、ツール供給が不足する一因となっている。例えば、日本における金融系基幹システムなど一部の事業ドメインにおいて、COBOL言語は依然主要な言語だが、ツール市場全体から見ると少数派であり、十分なツールが用意されていない。

（3）構文解析器開発における問題

以前はコンパイラなどの一部のツールのみが構文解析器を必要としていたが、近年ツールの高機能化に伴い、構文解析器を使用するツールも増えてきた。例えば、従来のソースエディタはキーワードのハイライトのために字句解析を行うことはあっても、構文解析は行わなかった。しかし、Eclipseをはじめとする高機能なソースエディタのように、構文依存の処理を行うために内部で構文解析を行う例も増えてきている。本項では、これらのツール開発において必要となる、構文解析器の開発が困難である理由を述べ、結果的にツール開発の難易度が上がりツールの供給が低下している問題を示す。

① 構文解析器生成系の多様化

構文解析器の開発が困難な理由の1つとして、構文解析器生成系の多様化がある。単にプログラミング言語ごとに使用される構文解析器生成系が異なるだけではなく、ある特定の言語に対しても多くの種類の構文解析器生成系が存在する。例えば、Javaで使える実装に限定しても、知名度の高いANTLR[3]、JavaCC[4]以外にも、SableCC、Beaver、RunCC、Grammatica、SJPTなど様々な構文解析器生成系がある[5]。ここで、各構文解析器生成系の記述方法にほとんど差がなく、移行も容易であれば問題とはならない。しかし、実際にはそれぞれの間で記述方法が大きく異

なり、移行は困難である。構文解析器生成系により、生成する構文解析器がLL(k)かLALR(k)かが異なるというだけではなく、構文規則の記法としてそれぞれが独自で拡張したEBNFを使っていることや、コンフリクトが発生した場合の構文規則間の優先順位設定の記述方法が異なること、エラーリカバリの記述方法が異なることなど、様々な差異がある。このため、どれか1つの記述方法を覚えたとしても、そのまま他の構文解析器生成系で記述できるようになるわけではなく、一般的な構文解析に関する理論以外は個別の記述方法を覚え直す必要がある。更に、有名なオープンソースプロジェクトで知名度の低い構文解析器生成系を使用している例もあり、開発するツールごとにそれが使用する構文解析器生成系の記述方法を覚える必要があることも多い。例えば、JavaではANTLRとJavaCCの2つが特に有名だが、Java開発における統合開発環境としてトップクラスのシェアを持つEclipse JDTは内部でJikesPG[6]を採用している。ところがそのほかのプロジェクトにおいてJikesPGの採用例は非常に少ないため、Eclipse JDTの拡張などを考える際にはJikesPGでの記述方法を新たに覚える必要がある。

② ツールにおける構文解析器の多機能化

理想的には、1種類の言語に対しては構文解析器を1つ作ればいいはずだが、実際にはツールごとに異なる構文解析器を作る必要がある。これは、ツールにおける構文解析器が、実際には構文解析の結果の構文木そのものではなく、意味解析を行いやすいように枝狩りや変形を行い、そのほかの属性を付加した抽象構文木（以降AST）を構築しており、このASTがツールにより異なるためである。

```
E → F “+” E | F “−” E | F
F → “a” | “b” | “c”
```

図3　右再帰文法の例

ここでは右再帰による演算子の優先度の問題を示す。例えば、図3の文法を受理するLL(k)構文解析器を考える。

ここで入力a－b－cを構文解析する場合を考える（図4）。これをLL(k)構文解析器でナイーブに実装すると「ツールT_2用AST」の形のASTが構築されることになるが、これをそのまま評価するとa－(b－c)となってしまい、本来のa－b－cと意味が変わってしまう。このためツールによっては(a－b)－cとなるように「ツールT_1用AST」の形のASTを構築するようロジックを作り込むことになる。また、ツールによっては「ツールT_3用AST」の形のASTを構築する場合もある（gccのフロントエンドとして動作するCOBOLコンパイラの実装であるOpenCOBOLでは、実際にこのようなASTを構築する。これはCOBOLのソースコード

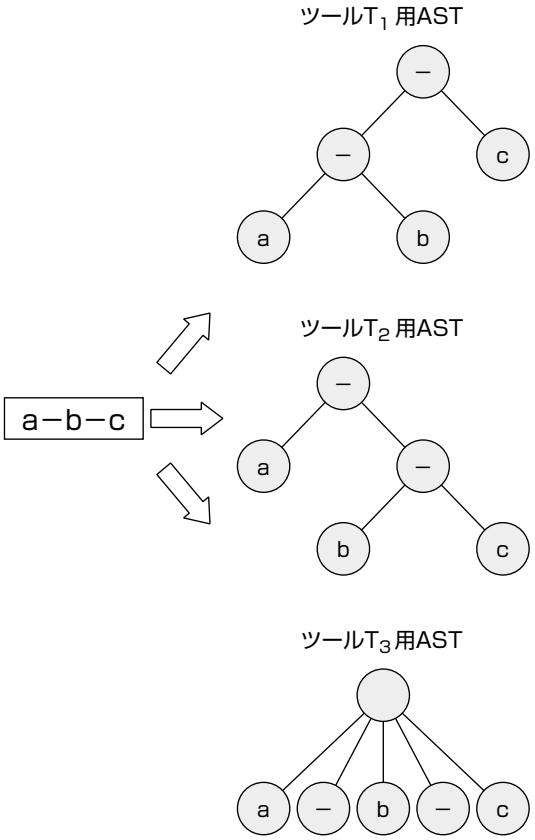

図4　ツール間での異なるAST

を等価なC言語のソースコードに変換するのが目的であり、OpenCOBOL自体は式を評価しないためである)。

また、表1は、幾つかのオープンソースソフトウェアで使用されている、構文解析器生成系の入力となるファイルの行数と、そのファイル中で文法規則を記述している行数と、ロジックを記述している行数の比較である(構文規則の行数とロジックの行数を足しても全体の行数より小さいのは、空行、コメント行、カッコのみの行を除いているためである)。この表より、構文解析器生成系を用いた構文解析器の開発において、文法の記述と同程度かそれ以上のロジックを作り込む必要があることが分かる。

③ 構文解析器生成系を用いた開発が困難

構文解析器の開発が容易であれば、ツールごとに毎回作ったとしても問題はない。しかし、実際には構文解析器の開発は難易度が高く開発要員の確保が困難な場合が多い。

まず、通常構文解析器の開発には構文解析器生成系を使うが、構文解析器生成系を使った開発は誰でもできるわけではない。これはASTを構築するために構文規則に独自の修正を加えたり、ASTを生成するロジックを各構文規則に埋め込んだり、LL(k)、LALR(k)な

ツール名	全体[行]	構文規則[行]	ロジック[行]
Jalopy-1.5b1(java.g)	2479	610	760
Jalopy-1.5b1(java.doc.g)	2129	600	609
checkstyle-4.4(java.g)	1888	723	278

表1　文法定義における構文規則とロジックの割合

どの構文解析器生成系のサポートする文法で書けない部分を処理するために文法を変えロジックで吸収したりする必要があり、構文解析器の開発を行うには、文脈自由文法や構文解析器に関する理論をある程度知っていなければならないためである。また、エラーリカバリを実装するには構文規則とロジック双方への実装が必要なため深いノウハウが求められ、このことも構文解析器の開発を困難にしている。そのほかにも構文解析器生成系を用いた開発が困難な理由として、開発を支援するツールがない点があげられる。構文解析器生成系のためのツールの研究も行われているが[7]いまだ一般には普及していない。

なお、構文規則から素直に実装が可能であり高い生産性を持つパーサー・コンビネータ[8]を用いた構文解析器も研究されているが、関数型言語による開発を行う必要があり、こちらも誰でもできるわけではない。

3．他言語へのツール適用による課題の解決

本節では前述のツール供給不足の問題を解決するために、ある言語の既存ツールを再利用して他言語へ適用する手法を提案する。ここでは、まず我々が対象とするツールに関しての定義を行い、アプローチ方法、適用の手法を述べ、次に具体例として、JavaのツールをCOBOLに適用するために、COBOLからJavaへ木構造の変換を行う場合の例を示す。

(1) 定義

① ツール

まず、我々が対象とするツールを「入力を受け取り、何らかのロジックを実行し、結果を出力するもの」と定義する。次にツールの入出力を次の3通りに分けて考える。

ソース	その言語のソースファイルまたはその一部
バイナリ	コンパイラが生成するバイナリファイル
その他	ソース、バイナリ以外

上記の定義から、ツールの入力と出力が「ソース」「バイナリ」「その他」の3種類のいずれかにより、ツールを次の9通りに分類する。

(i) ソース→ソース
(例：エディタなどソース編集ツール)
(ii) ソース→バイナリ（例：コンパイラ）
(iii) ソース→その他（例：静的チェックなどの解析ツール）
(iv) バイナリ→ソース（例：逆コンパイラ）
(v) バイナリ→バイナリ
(vi) バイナリ→その他
(vii) その他→ソース
(例：ソース生成ツール)
(viii) その他→バイナリ
(ix) その他→その他

このとき(v)、(vi)、(viii)、(ix)は、その対象言語とは独立の入出力なので、対象言語のツールとは考えないこととする。また、

我々の目的は既存のツールを再利用することであるため、ツールの修正が可能であるオープンソースのツールを対象とする。

② ツールの適用

次に、我々のアプローチ方法である「言語G_1のツールT_1を言語G_2に適用する」とは何を指すかを定義する。ここでは、「ツールT_1の入出力のうちソースであるものを言語G_2のソースに対応させること」と定義する。例えば、Java言語のツールをC言語に対応させると次のようになる。

(i) ソース→ソース：Javaソースエディタで C のソースファイルを編集
(ii) ソース→バイナリ：C のソースファイルから Java のクラスファイルを生成するコンパイラ
(iii) ソース→その他：Java の静的チェックで C のソースを静的チェック
(iv) バイナリ→ソース：Java のクラスファイルから C のソースファイルを生成する逆コンパイラ
(v) その他→ソース：Java ソース自動生成で C のソースを生成

本稿では前者3パターン（編集系、コンパイラ、解析系）を対象として、入力であるソースへの対応方法を述べる。

(2) アプローチ

ツールの供給は不足しているが、それはすべての言語で不足しているわけではなく、供給量は言語間で偏っている。特に、最近では Java のツールは十分に供給されており、種類も多く、同一機能を目的とするツールの選択肢も多い。

そこで、例えば Java のようなツールの豊富な言語からツールを持ってきて、ツールの少ない言語に適用することができれば、ツールの供給不足を補うことができる。このとき、我々の目的はツールの機能の100%を利用可能にすることではない。世の中にある Java ツールのうち前述の定義に合うものを選び、その機能の80%を利用可能にできれば良い。ただし、そのときのツール適用の工数は、そ

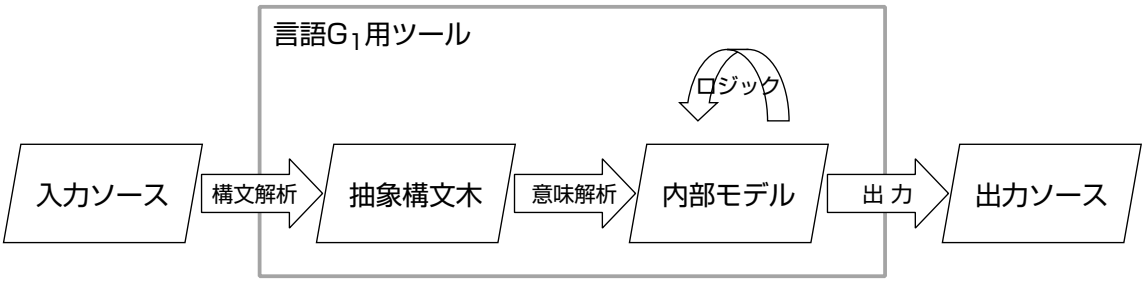

図5　一般的なツールの処理パターン

のツールをスクラッチから開発する工数の20%以下を目標とする。

まず、言語G_1を対象とするツールの一般的な処理フローを考える（図5）。ここで想定する処理フローは、次の処理から成る。

(i) 入力である言語G_1のソースを構文解析し、ASTを構築する。
(ii) ASTに意味解析を行い、内部モデルを構築する。
(iii) ロジックを実行し、内部モデルの分析、編集を行う。
(iv) 内部モデルから出力ソースを生成する。

ここで示した処理手順を持つツールをその他の言語G_2に適用する場合、一般的には次の手順でツールへの修正や追加実装を行う（図6）。

(i) 言語G_2の構文解析器及びASTの実装を新規に追加する。
(ii) 内部モデルから元々の対応言語G_1に依存した部分を取り除き言語独立する。
(iii) 内部モデルの修正に合わせてロジックを修正する。
(iv) 言語G_1用の意味解析処理をG_2のASTに対応するよう修正した意味解析処理を追加する。
(v) 言語独立内部モデルから新しく対応する言語G_2に対応した出力を新規に追加する。

この手順は、新規にツールを開発するのに比べれば工数が少ないとはいえ、元々のツールの持つコンポーネントに相当する数のコンポーネントを修正、または新規に実装する必要があるため、依然その負荷は大きい。

ここで、図6に示した処理のうち、その入力処理を簡略化し、入力である言語G_2のソースから直接言語G_1のASTを構築する場合を考える（図7）。すると、一般的な他言語への適用手法では(i)～(v)の5ステップ必要だったのが、次の2ステップで他言語適用が可能となる。

(i) 新規に対応する言語G_2の構文解析器を開発する。ただし構築するASTは、既存の対応言語G_1のASTを直接構築する。
(ii) 内部モデルから新しく対応する言語G_2に対応した出力を新規に追加する。

このアプローチでは入力から直接本来の対応言語のASTを構築するため、入出力以外の部分はすべて再利用が可能であり工数が非常に小さい。

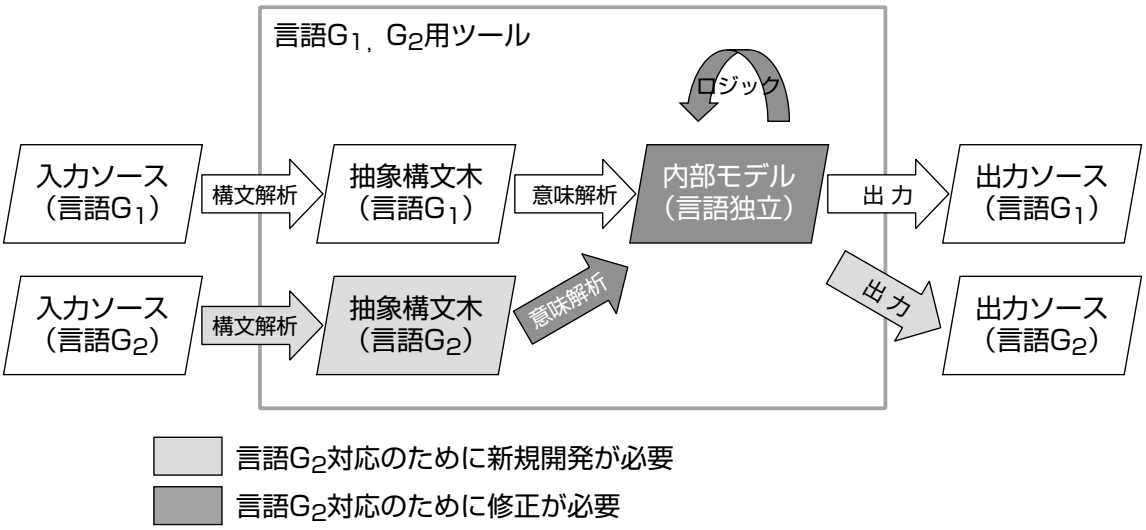

図6　一般的な他言語対応アプローチ

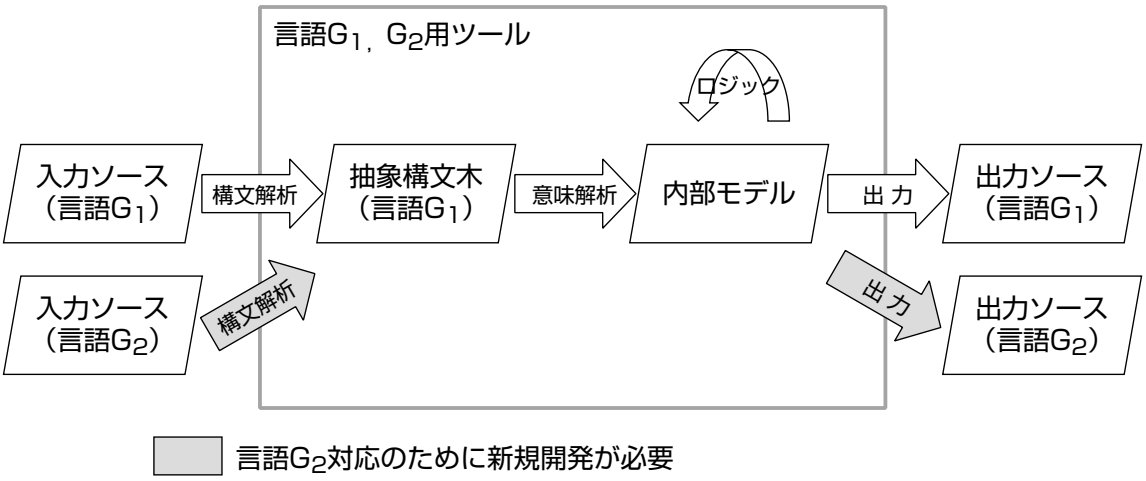

図7　AST変換による他言語対応アプローチ

（3）他言語へのツール適用手法

前述のアプローチを実現するために、まず(i)の入力処理、すなわち新規に対応する言語G_2用の構文解析器として、既存の対応言語G_1のASTを直接構築する構文解析器の構築方法を考える。ここでは、まずその準備として、我々の提案する純粋構文木による構文解析について述べる。次に、この純粋構文木

と木構造の変換による他言語へのツール適用手法を述べる。

なお、(ii) の出力処理は一般的な多言語への適用アプローチと同様なのでここでは省略する。

① 純粋構文木による構文解析

従来のツール開発における構文解析器は構文解析をしながらASTの構築を行っている。これに対し、我々の提案する純粋構文木による構文解析では、入力からASTを構築する処理を以下の2つのステップに明確に分ける。

(a) 入力から構文木の構築
(b) 構文木からASTの構築

ここでステップ (a) で構築する構文木は従来ASTの構築で行う枝狩りや木構造の変換を一切行わない、その文法の構文規則に忠実な構文木である。本稿ではこれを純粋構文木と呼ぶ。

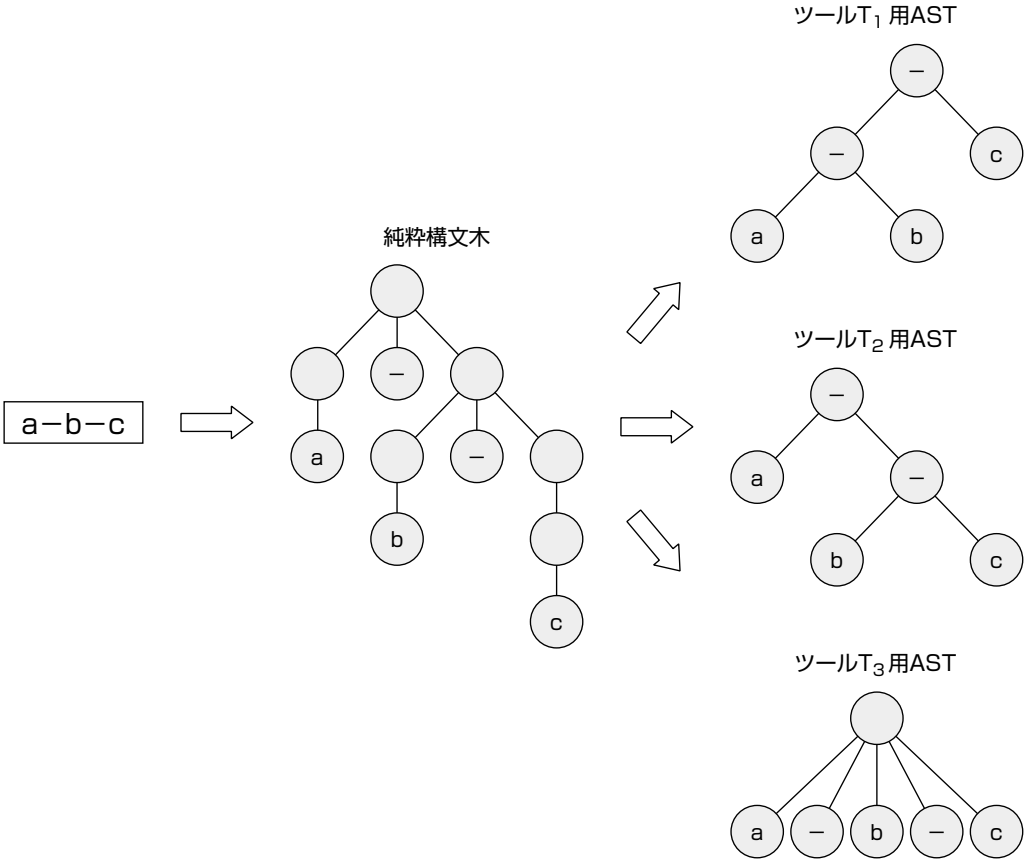

図8　純粋構文木を介したASTの構築

なお、従来の構文解析処理も概念的には、構文解析とASTの構築の2つのステップから成る。しかし、実際には、中間生成物として構文木を作ることはなく、構文解析の処理と並行してASTの生成を行っている。純粋構文木による構文解析では、構文解析処理が終わり純粋構文木の生成が完了してから、ASTの構築を行う点が従来と異なる。

このとき、ステップ（a）で生成した純粋構文木は入力ソースとなる言語にのみ依存するため、一度このような構文解析器を作れば、その他のツールにも再利用できる。すなわち、各ツールは純粋構文木に対して木構造の変換を行うことで、個々のツールの求めるASTを構築すれば良い（図8）。この木構造の変換はXSLTなどの一般的な手法を使えるため、構文解析などの特殊な知識は不要である。このため、2.（3）③で述べた構文解析器の開発要員確保の問題を回避できる。

② 純粋構文木と木構造の変換による他言語へのツール適用

3.（2）で述べた通り、我々のアプローチでは、あるプログラミング言語G_1のツールT_1を別のプログラミング言語G_2のソースコードに適用するために、その入力処理（ステップ（i））でG_2のソースコードからT_1で使用するASTを構築する。

まず、言語G_1のソースコードL_1に対し、言語G_2においてL_1と等価なソースコードをL_2とする。すると、前述の入力処理（ステップ（i））は、次の3ステップで行うことができる（図9）。

（i） 言語G_2のソースL_2を構文解析し純粋構文木A_2を構築する
（ii） 木構造の変換により、L_2と等価なL_1の純粋構文木A_1へ変換する
（iii） L_1の純粋構文木A_1からツールT_1のASTへ変換する

ここで、ステップ（i）と（iii）が、純粋構文木による構文解析のステップ（a）と（b）にそれぞれ対応する。

ただし、ここで述べた「等価なソースコード」の定義は容易ではない。既存のツールで行われている「等価なソースコードへの変換」の多くはセマンティクスの面で等価なソースコードへの変換であった。典型的な例としてはコンパイラのフロントエンドの実装がある。例えば、OpenCOBOL[9]はCOBOLのソースコードを入力とし、それをセマンティクス的に等価なC言語のソースコードへと変換しgccの入力とすることで、COBOLソースコードのコンパイルを行う。一方、ここで提案するアプローチでは、必ずしもセマンティクスの観点での等価性を考慮するわけではなく、シンタックスの観点での等価性を重視する場合もある。例えば、ソースコードエディタの

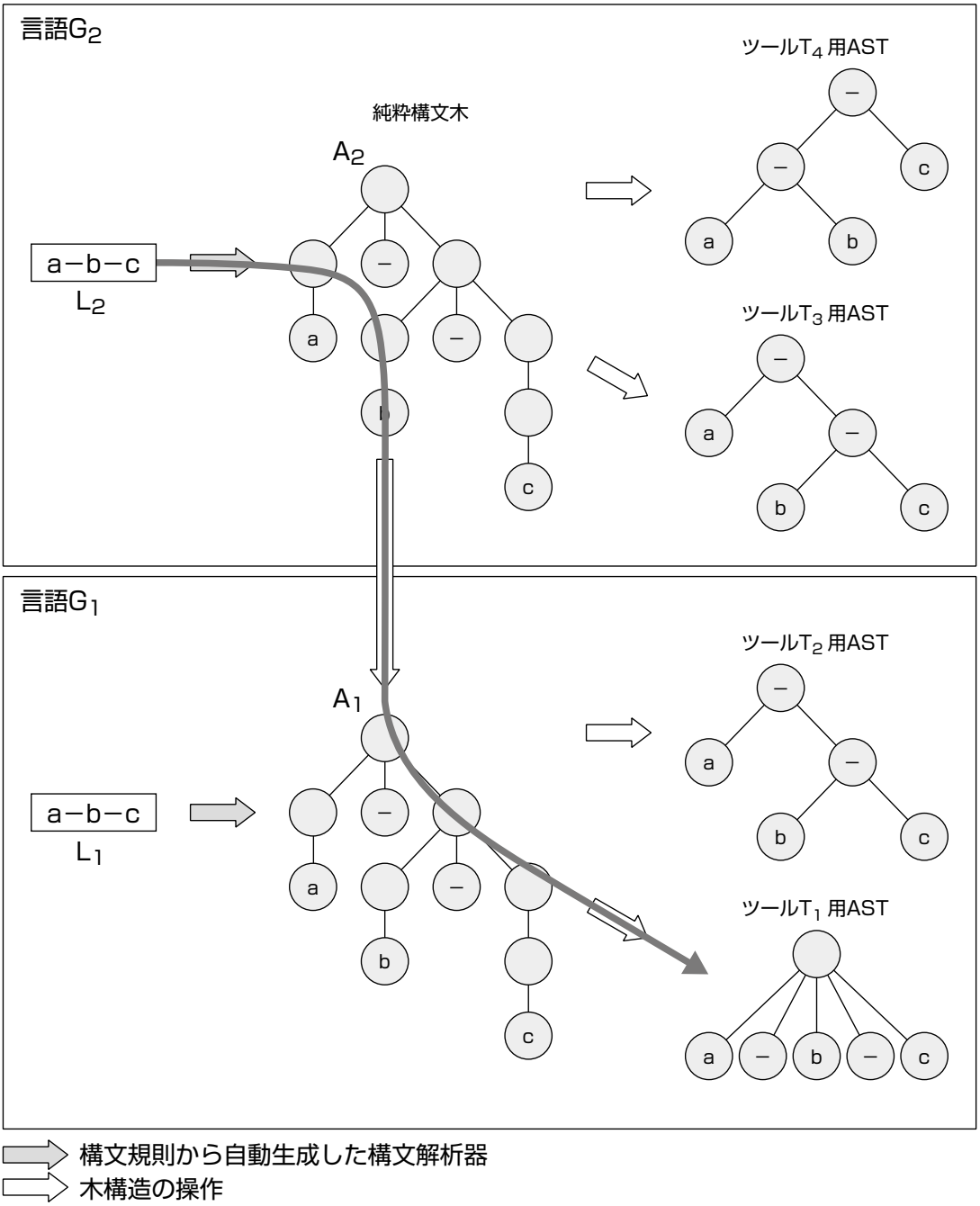

図9　純粋構文木を介したASTの構築

ようなツールを考えた場合、言語間での変換によりセマンティクスを保存することはそれほど重要ではなく、プログラム内のメソッドや関数、ステートメントのネストなどの構造に関する情報の方が重要なためである。

なお、言語間での純粋構文木の変換を考える上で、セマンティクスとシンタックス両方を満足する変換を定義するのは非常に困難である。このため、ここで提案するアプローチでは、どんなツールにも利用できる万能の純粋構文木の変換は定義できないという立場を取り、本手法を適用するツールの特性に合わせて、セマンティクスの等価性を重視した変換や、シンタックスの等価性を重視した変換を使い分けることで、様々なツールに対応する。セマンティクスの等価性重視の変換はOpenCOBOLなど実績があるため、本稿ではシンタックスの等価性重視の変換を行った例を3.(5)で述べる。

(4) COBOLからJavaへの変換概要

ここでは具体例として、Javaのツールを COBOLに適用するためにCOBOLからJavaへの変換を行う場合の例を示す。言語間の変換は、基本的には変換元の言語の各言語要素を変換先の言語の相当する言語要素に変換することで行う。ここで、相当する言語要素とは、シンタックス、セマンティクス双方の観点から同等の言語要素を指す。ただし、シンタックスの観点で等価とは、ソースコードの見た目が同じであることではなく、構文解析を行った結果の構文木の構造が等価であることを指す。

ここでは変換元の言語COBOLの各言語要素が、変換先の言語Javaにおいて、

- 相当する言語要素がある場合
- 相当する言語要素があるが情報が不足する場合
- 相当する言語要素がない場合

の3通りに分けてそれぞれ例を述べる。なお、変換先の言語にはあるが変換元の言語にはない言語要素については、単に使われないだけなので考えない。

(5) 変換先の言語に相当する言語要素がある場合

変換元の言語要素のうち、変換先の言語に相当する言語要素がある場合は、単純にその相当する言語要素への変換を考えればよい。ここでは、COBOLのプログラム、データ項目、集団項目、及びステートメントの変換例を示す。

① プログラム

COBOLでの1プログラムはJavaのクラスに変換する（厳密にはCOBOLの1ファイルに複数のコンパイル単位を記述できるので、1コンパイル単位を1クラスに変換してい

```
001000*
001100 IDENTIFICATION DIVISION.
001200 PROGRAM-ID. PROG001.
002000*
004100 PROCEDURE DIVISION.
009000 END PROGRAM PROG001.
```

```
public class PROG001 extends CobolBase {
        public void procedureDivision() {
        }
}
```

図10　プログラムおよび手続きの変換例

る）。図10ではIDENTIFICATION DIVISIONからEND PROGRAMまでの1プログラムを、Java上の1クラスに変換している。また、PROGRAM-IDであるPROG001を変換先のクラスのクラス名としている。なお、詳細は省略するが、PROCEDURE DIVISIONや手続きの各SECTIONを変換先のクラスのメソッドに変換している。

② データ項目

COBOLのDATA DIVISION内のデータ項目はJavaのメンバ変数に変換する。図11の例では、COBOL上のデータ項目DATA1、DATA2を、Java上のメンバ変数DATA1、DATA2に変換している（各メンバ変数に付随するアノテーションに関しては後述する）。

```
003300   01 DATA1 PIC 9(02).
003400   01 DATA2 PIC 9(02).
```

```
            @CobolDataDescription(LEVEL="01", PIC="9(02)")
            private CobolData DATA1 = new CobolData();

            @CobolDataDescription(LEVEL="01", PIC="9(02)")
            private CobolData DATA2 = new CobolData();
```

図11　データ項目の変換例

③ 集団項目

COBOLの集団項目による階層構造を表現するためにJavaではインナークラスとインスタンス変数を使用する。図12の例では、COBOLの集団項目GROUP1をJavaのインナークラスGROUP1Class及び、そのクラスのインスタンス変数GROUP1に変換している。また、階層構造を表現するために、集団項目GROUP1内のデータ項目DATA1、DATA2はJava上ではGROUP1Classのメンバ変数に変換している。

このように変換することで、COBOL上のデータ参照DATA1 OF GROUP1は、Java上でGROUP1.DATA1と似通った形式に変換することができる。

なお、COBOLでDATA DIVISION内の各SECTION（WORKING_STORAGE、FILE、LINKAGE）による階層構造も同様の変換規則で表現できる。例えば、WORKING_STORAGE SECTIONは、Java上でWorkingStorageClassクラスとそのインスタンス変数WorkingStorageに変換し、COBOL上のWORKING_STORAGE SECTION内の各データ項目は、WorkingStorageClassクラスのメンバ変数に変換する。

④ ステートメント

COBOLの多くのステートメントはJavaで同名のメソッドを用意することで変換できる。図13にMOVE文の例を示す。ここで、MOVEメソッドは第1引数で指定したデータ項目の値を第2引数で指定したデータ項目にコピーするメソッドである。

```
003300 01 GROUP1.
003400    03 DATA1 PIC XX.
003500    03 DATA2 PIC XX.
```

```
@CobolDataDescription(LEVEL="01")
private GROUP1Class GROUP1 = new GROUP1Class();
private static class GROUP1Class extends CobolData {
            @CobolDataDescription(LEVEL="03", PIC="XX")
            private CobolData DATA1 = new CobolData();

            @CobolDataDescription(LEVEL="03", PIC="XX")
            private CobolData DATA2 = new CobolData();
}
```

図12　集団項目の変換例

(6) 変換先の言語に相当する言語要素があるが情報が不足する場合

前項において、COBOLのデータ項目からJavaのメンバ変数への変換例を示したが、実際にはこの変換例は十分ではない。COBOLのデータ項目は、レベル番号やPICTURE句の文字列など、Javaのメンバ変数にない情報を持つことができるためである。このように、変換先の言語に変換元の言語要素に相当する言語要素があるが、変換先の言語要素で変換元の言語要素の持つすべての情報を持てない場合、変換先の言語においてセマンティクスに影響を与えない言語要素を付加することで必要な情報を持たせる。このようなセマンティクスに影響を与えない言語要素としてはコメントが典型的だが、Javaの場合はアノテーションを用いることで対象のメンバ変数にひも付いた形で情報を付加することができる。

① データ項目

前述の通り、COBOLのデータ項目をJavaのメンバ変数に変換した場合、メンバ変数で表現できない情報が欠落してしまう。このため、各メンバ変数にアノテーションを追加し、メンバ変数が保持できない情報をアノテーションに保持させる。図11で示した変換例では、アノテーションでレベル番号と、PICTURE句の文字列の情報を保持している。

(7) 変換先の言語に相当する言語要素がない場合

変換先の言語に相当する言語要素がない場合、類似の言語要素へ変換することになる。ここで、相当する言語要素、すなわちシンタックスとセマンティクス双方の観点から等価な言語要素がないので、そのどちらかを捨てる必要があり、目的によって判断する必要がある。ここでは例としてセクションとパラグラフの変換例を示す。

① セクション・パラグラフ

COBOLではステートメントのブロックとして、セクションとパラグラフの2段階の単位が用意されており、1つのセクションを複数のパラグラフで構成することができ、更にセクション、パラグラフのどちらもサブルーチンとして扱うことができる。Javaにはこのような複雑な構造はなく、サブルーチンとし

```
004110 MOVE DATA2 OF GROUP1 TO DATA1.
```

```
    _MOVE(WorkingStorage.GRUOP1.DATA2,
         WorkingStorage.GROUP1.DATA1);
```

図13　MOVE文の変換例

```
004000 SECTION1 SECTION.
004010 MOVE DATA1 TO DATA2.
004020 MOVE DATA3 TO DATA4.
004100 PARAGRAPH1.
004110 MOVE DATA5 TO DATA6.
```

```
public void SECTION1() {
   {
       MOVE(WorkingStorage.DATA1, WorkingStorage.DATA2);
       MOVE(WorkingStorage.DATA3, WorkingStorage.DATA4);
   }
   PARAGRAPH1:
   {
       MOVE(WorkingStorage.DATA5, WorkingStorage.DATA6);
   }
}
```

図14　セクション及びパラグラフの変換例

て使用できる言語要素はメソッドのみである。このため、COBOLからJavaへの変換では、セクション・パラグラフどちらもメソッドに変換してしまう方法（この場合パラグラフがセクションの一部というシンタックス上の構造は失われる）や、セクションだけメソッドに変換し、パラグラフはメソッド内のブロックなどに変換する方法（この場合パラグラフはサブルーチンではなくなり、セマンティクス上の互換性は失われる）など、トレードオフの関係にある複数の変換方法があり、ツールの用途によって使い分ける必要がある。

図14に示す変換例ではCOBOLのセクションをJavaのメソッドに変換し、パラグラフをブロック（「{」から「}」で囲まれた範囲）に変換し、パラグラフ名をラベルに変換している。ここで示す変換例は、セマンティクスよりもシンタックスを優先している。例えば、この変換ではパラグラフへのPERFORMが記述できなくなっている。

このような変換規則を採用した理由は、この変換がJavaのコーディングルールチェックツールをCOBOLに適用するために行ったものであり、セマンティクスよりもシンタックスの等価性が重要であったことと、このツールが対象とするCOBOLプログラムはコーディングルールによりパラグラフへのPERFORMの使用が禁止されているため、パラグラフのセマンティクスは重要ではなかったためである。このようにツールの用途により、どのような変換規則を採用するかを考える必要がある。

4．実験・評価

(1) 評価実験

本手法を評価するため、次の2種類の実験を行った。

- 本手法がある特定のツールだけではなく広く適用できることを検証する実験
- ツール適用の工数がスクラッチからの開発と比べ十分に小さいことを検証する実験

ここでは、これらの検証実験の概要と結果を示す。

① 適用範囲に関する検証実験

3.で述べた手法により、様々なツールを他言語に適用できることを示すために、Java言語の開発環境であるEclipse JDTを、本稿で提案した手法によりCOBOL言語へと対応させる実験を行った。ただし、本手法がツールの種類に依存せず適用可能であることを示すことが実験の目的であるため、使用した文法はCOBOLのサブセットの文法である。具体的には、データ項目、ステートメント、式、制御構造（IF文）といったプログラミング言語としての最低限の部分と、セクションなどのCOBOL言語特有の構造を持つものである。

この結果、前述した木構造の変換処理を実装し、一部Java言語のトークンに特化した処理（メソッドが「}」という文字で終了することに依存した処理があった）を修正することで、Eclipse JDT上で、COBOLに対応した次の機能を実現することができた。

- Javaソースエディタによる、COBOLプログラムの構造の認識（セクションの折り畳みが可能）
- コンパイラによるCOBOLのソースコードからJavaのクラスファイルの生成
- アウトラインビューによるCOBOLプログラムの構造（データ項目、セクション）の表示
- クロスリファレンスによるデータ項目の定義個所の表示、エディタ上でのフォーカス移動

以上から、ソースコードの構文解析、意味解析、コンパイルといった複数の処理パターンを本手法により他言語へと適用できることを確認できた。

② 工数に関する検証実験

本手法によるツールの適用が、ツールをスクラッチから開発するのに比べ十分に効率的であることを確認するために、Java言語の静的チェックツールPMDを本稿で提案した手法によりCOBOL言語へと対応させ、COBOL対応に要した工数を測定した。結果を表2に示す。

表から、ツール適用の工数は、そのツールをスクラッチから開発する工数の20%以下という目標を達成できたことが分かる。

	行数	ファイルサイズ
PMDソース(Java対象ロジックのみ)	60,000行	1,800KB
COBOL対応修正部分	7,200行	230KB
割合	12%	13%

表2　工数の比較

(2) 関連研究・手法との比較

① テンプレートによる構文解析器開発との比較

構文解析器の開発における工数削減の方法として、あらかじめ用意された文法ファイルのテンプレートを元に構文解析器を開発する方法がある。それらのテンプレートには構文規則だけが用意されており、そこにロジックを追加することで必要な構文解析器を作成できる。

しかし、2.(3) で述べた構文解析器生成系の多様化により、必ずしも採用する構文解析器生成系に対象とするプログラミング言語のテンプレートがあるとは限らない。例えば、ANTLRでもテンプレートはJava、C/C++、Rubyといった一部のメジャーな言語に限られ、COBOL、FORTRANなどのレガシーな言語やScala、PHPなどの近年注目されているがシェア自体は小さい言語はサポートされていない。

また、テンプレートには、構文規則自体は用意されているが、AST構築のための構文規則の修正や、ロジックをどの構文規則に追加するかの判断は依然必要であり、開発に構文解析器に関する知識が必要という問題は解決できない。

これに対し、本手法ではEBNFなどの形式的な言語の定義があれば構文解析器の自動生成も可能であり、テンプレートを用いる手法よりもカバーできる言語の範囲は広い。

② ツール・プラットフォームを用いた開発との比較

構文解析器開発を含め、ツール開発において重複した作業を削減する方法としては、Sapid[10]などのように、あらかじめ構文解析器や内部モデル、APIを用意したツール・プラットフォームを利用する方法がある。ツールをスクラッチから開発する場合、これらの手法を利用することで、効率よく開発することができる。

しかし、我々が目指す既存のツールの再利用を考えた場合、既存のツールはSapidなどで提供するモデルとは異なる内部モデルを使っている場合がほとんどであり、ツール・プラットフォームを利用してもツールの再利用は難しい。

5．おわりに

本稿では、ツールの供給不足の問題を解決するために、純粋構文木による構文解析器の共通化と、構文木間での木構造の変換を用いることで、あるプログラミング言語G_1のツールを他の言語G_2に適用可能にする手法を提案し、具体例としてJavaのツールをCOBOLに適用するための、COBOLからJavaへの変換例を示した。また、評価実験を行い、その結果を示すとともに、関連手法との比較を行い、その効果を示した。

ここで提案した方法は従来の方法と比べ、以下の特長を持つ。

- 純粋な構文解析の処理をツールごとに作り直す必要がなく再利用できる。
- 従来構文規則ごとに埋め込まれていたASTの構築ロジックを分離でき、保守性が高まる。
- 純粋構文木からASTへの木構造の変換や、異なる言語の純粋構文木間の変換はXSLTなどの一般的な手法を使えるため構文解析のような特殊な知識は不要である。
- ツールの構文解析を除く部分を再利用できるため開発工数を20%以下に抑えることができる。

ただし、本稿で提案したプログラミング言語間の純粋構文木の変換において、個々の変換ルールは変換元・先の言語に大きく依存する。本稿では変換の方針とCOBOLからJavaへの変換例の一部を示したのみである。現状では木構造の変換処理の実装は、ノウハウに依存する部分が大きく、本手法をその他の言語に応用する場合には、専門的な知識に依存せず構文解析器の開発を行えるようにするという当初の目的を完全には達成できていない。この課題の解決策としては、木構造の変換ルールを記述する上で、形式的にルールを記述して自動生成する方法や具体的な変換ルールの作成方法を提示する方法が考えられ、これらの検討が今後の課題である。

●参考文献●

[1] Addison-Wesley
“Compilers ： Principles, Techniques, and Tools”
Jeffray.D.Ullman、Alfred V.Aho、Ravi Sethi 著、1986 年

[2] C. M. University
“What is a CASE Environment?”
2007 年
http://www.sei.cmu.edu/legacy/case/case_whatis.html

[3] Antlr parser generator
http://www.antlr.org/

[4] java.net「Javacc home」
https://javacc.dev.java.net/

[5] 「Parser generators」
http://www.java2s.com/Product/Java/Development/Parser-Generators.htm

[6] Jikes
http://jikes.sourceforge.net/

[7] 『情報処理学会ソフトウェア工学研究会研究報告』
「構文解析器の開発支援環境」
笠原史郎、大久保弘崇、粕谷英人、山本晋一郎著、2004-SE-144-8

[8] Springer Berlin / Heidelberg
『Deterministic, error-correcting combinator parsers』S. D. Swierstra and L. Duponcheel 著、1996 年

[9] 「OpenCOBOL - an open-source COBOL compiler」
http://www.opencobol.org/

[10] 『情報処理学会論文誌 Vol.39 No.6』
「細粒度リポジトリに基づいた case ツール・プラットフォーム sapid」
福安直樹、山本晋一郎、阿草清滋著、pp.1990-1998

[11] 『電子情報通信学会技術研究報告 Vol.108 No.173』
「AST 変換を用いた他言語へのツール適用」長谷川勇著、2008 年