(19)日本国特許庁(JP)　(12)公開特許公報(A)　(11)特許出願公開番号
特開2010-140407
(P2010-140407A)
(43)公開日　平成22年6月24日(2010.6.24)

(51)Int.Cl.	F I	テーマコード（参考）
G06F 11/36 (2006.01)	G06F 9/06 620M	5B376

審査請求 未請求　請求項の数 5　ＯＬ　（全 12 頁）

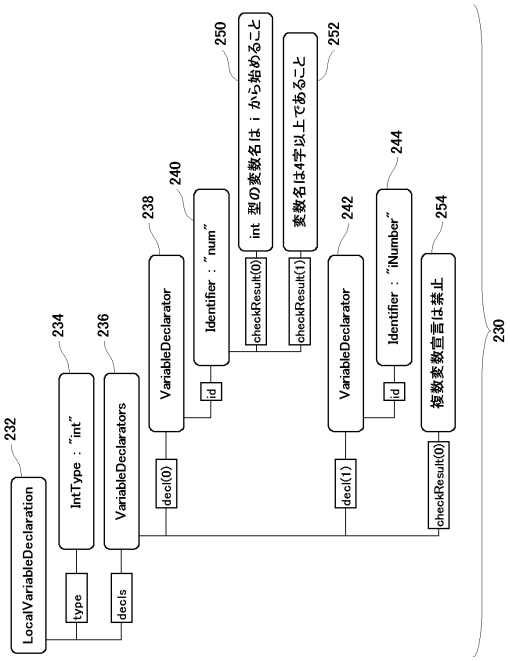
(21)出願番号　特願2008-318345 (P2008-318345)
(22)出願日　平成20年12月15日 (2008.12.15)

(71)出願人　000155469
株式会社野村総合研究所
東京都千代田区丸の内一丁目６番５号
(74)代理人　100105924
弁理士　森下　賢樹
(72)発明者　木村　綾太郎
東京都千代田区丸の内一丁目６番５号　株式会社野村総合研究所内
(72)発明者　藤田　由起
東京都千代田区丸の内一丁目６番５号　株式会社野村総合研究所内
(72)発明者　山城　俊介
東京都千代田区丸の内一丁目６番５号　株式会社野村総合研究所内

Ｆターム(参考)　5B376 BC39

(54)【発明の名称】ソースコード検査装置

(57)【要約】
【課題】ソースコードの解析を支援する。
【解決手段】ソースコード処理装置は、検査対象となるソースコードを取得し、その構文解析を行いＡＳＴ（抽象構文木）を生成する。ＡＳＴは、ソースコードの構成要素に対応するノードをその論理構造にしたがって構造化したモデルである。そしてＡＳＴにおける各ノードを走査し、ソースコードの構成要素についてあらかじめ設定されている検出条件に合致する箇所を特定し、注釈ノードを設定する。
【選択図】図７



【特許請求の範囲】
【請求項１】
ソースコードを取得するコード取得部と、
前記ソースコードの構成要素に対応するノードを抽出し、前記ソースコードにおける前記構成要素の配置にしたがって前記ノードを構造化することにより、前記ソースコードの論理構造を前記ノードの集合体により示す構文モデルを生成する構文モデル生成部と、
前記構文モデルにおける各ノードを走査し、前記ソースコードの構成要素についてあらかじめ設定されている検出条件に合致する箇所を特定する検査部と、
前記特定された箇所に注釈を示すノードである注釈ノードを設定する注釈ノード設定部と、
を備えることを特徴とするソースコード検査装置。
【請求項２】
前記検査部は、所定の命名規則に適合しない名称の変数に対応するノードに前記注釈ノードを設定することを特徴とする請求項１に記載のソースコード検査装置。
【請求項３】
前記検査部は、更に、所定の処理の開始時に記述されるべき構成要素に対応する先端ノードと、前記所定の処理の終了時に記述されるべき構成要素に対応する終端ノードを特定し、先端ノードの数と終端ノードの数が同数となっているかを判定し、
前記注釈ノード設定部は、前記先端ノードの数と前記終端ノードの数が同数となっていないときには、更に、前記先端ノードと前記終端ノードに関するルールが満たされていない旨を示す注釈ノードを設定することを特徴とする請求項１または２に記載のソースコード検査装置。
【請求項４】
前記検査部は、第１のクラスにおいて定義されている第１のメンバ関数が、前記第１のクラスを継承する第２のクラスにおいて定義されている第２のメンバ関数によりオーバーライド（override）されているかを判定し、
前記注釈ノード設定部は、前記第２のメンバ関数が前記第１のメンバ関数をオーバーライドしているとき、前記第２のメンバ関数に対応するノードに前記注釈ノードを設定することを特徴とする請求項１から３のいずれかに記載のソースコード検査装置。
【請求項５】
ソースコードを取得する処理と、
前記ソースコードの構成要素に対応するノードを抽出し、前記ソースコードにおける前記構成要素の配置にしたがって前記ノードを構造化することにより、前記ソースコードの論理構造を前記ノードの集合体により示す構文モデルを生成する処理と、
前記構文モデルにおける各ノードを走査し、前記ソースコードの構成要素についてあらかじめ設定されている検出条件に合致する箇所を特定する処理と、
前記特定された箇所に注釈を示すノードである注釈ノードを設定する処理と、
をコンピュータに実行させることを特徴とするソースコード検査プログラム。
【発明の詳細な説明】
【技術分野】
【０００１】
本発明は、コンピュータプログラムの分析、特に、ソースコードの分析を支援するための技術に関する。
【背景技術】
【０００２】
企業や公共施設などの運用を支える業務システム、いわゆるエンタープライズシステム（Enterprise System）は、今や、大小さまざまな組織の基盤として導入されている。業務システムは、ノード端末やデータベースから得られるデータを集計、蓄積、解析、加工した上でより付加価値の高い情報を出力する。業務システムを構成するプログラムファイルの数は、数十万、時には数百万にも及ぶことがある。業務システムは複雑化する組織マ

ネジメントをより効率化するために発展を続けている。
【特許文献１】特開２００７－１５６７９１号公報
【発明の開示】
【発明が解決しようとする課題】
【０００３】
業務システムは、開発時だけでなく保守作業時・再構築時においてもソースコードの検査・修正を必要とする。特に、ソースコードの修正においては、修正作業自体よりも、膨大なソースコードの中から修正すべき箇所を特定するための作業の方が負荷が大きい。
【０００４】
また、複数のプログラマが関与する大規模な業務システムの開発においては、コーディングルールを制定することが多い。たとえば、関数や変数の命名、コメントの書き方等について統一性を持たせれば、ソースコードの可読性を高めたり修正ミスを防ぐことができる。しかし、コーディングルールの遵守は、プログラマの遵法意識に依存しているのが現状である。
【０００５】
本発明は、本発明者による上記課題認識に基づいて完成された発明であり、その主たる目的は、ソースコードの効率的な解析を支援するための技術を提供することにある。
【課題を解決するための手段】
【０００６】
本発明のある態様は、ソースコード検査装置に関する。
この装置は、ソースコードを取得し、ソースコードの構成要素に対応するノードを抽出し、構成要素の配置にしたがってノードを構造化することにより、ソースコードの論理構造をノードの集合体により示す構文モデルを生成する。そして、構文モデルにおける各ノードを走査し、ソースコードの構成要素についてあらかじめ設定されている検出条件に合致する箇所を特定し、注釈ノードを設定する。
【０００７】
なお、以上に示した各構成要素の任意の組み合わせ、本発明を方法、システム、記録媒体、コンピュータプログラムにより表現したものもまた、本発明の態様として有効である。
【発明の効果】
【０００８】
本発明によれば、ソースコードの効率的に解析しやすくなる。
【発明を実施するための最良の形態】
【０００９】
図１は、ソースコードの言語変換過程を示す模式図である。
本実施例においては、日立ＣＯＢＯＬで記述されたソースコード（以下、「日立ソース２００」とよぶ）をＩＢＭ・ＣＯＢＯＬで記述されたソースコード（以下、「ＩＢＭソース２０２」とよぶ）に変換する過程を例として説明する。
【００１０】
ソースコードの言語変換を実現するための主たる機能は３つあり、それぞれ、パーサ（parser）、プロセッサ（processor）、ライタ（writer）とよぶ。各機能は、解析処理、変換処理、生成処理をそれぞれ担当する。
解析処理（Ｓ１）：まず、第１ソースコードとして日立ソース２００を取得し、パーサが構文解析し、第１の構文モデルとしての抽象構文木（以下、「ＡＳＴ（Abstract Syntax Tree）」とよぶ）を生成する。ＡＳＴは、演算子やそのオペランドといったソースコードの構成要素にノードを対応づけた木構造として形成される。ＡＳＴは、一般的なコンパイラやインタプリタで採用されている既存技術の応用により生成できる。日立ソース２００のように変換元となるソースコードを「変換元ソース」、変換元ソースから生成されるＡＳＴを「変換元ＡＳＴ」、変換元ＡＳＴに含まれるノードを「基本ノード」、または、「変換元ノード」とよぶ。変換元ＡＳＴは、変換元ソースの言語構造に対応したＡＳＴで

あってもよいし、他のプログラミング言語、たとえば、標準的なＣＯＢＯＬの言語構造に対応した標準モデルとしてのＡＳＴであってもよい。解析処理については、図３に関連して詳述する。

【００１１】

変換処理（Ｓ２）：プロセッサは、変換元ＡＳＴを、第２ソースコードとしてのＩＢＭソース２０２に対応したＡＳＴに変換する。ＩＢＭソース２０２のように変換先となるソースコードを「変換先ソース」、変換先ソースのために生成されるＡＳＴを「変換先ＡＳＴ」、変換先ＡＳＴに含まれるノードを「変換先ノード」とよぶ。第２構文モデルとしての変換先ＡＳＴは、変換先ソースの言語構造に対応する。変換処理については、図３および図４に関連して詳述する。

【００１２】

生成処理（Ｓ３）：ライタは、変換先ＡＳＴから変換先ソースであるＩＢＭソース２０２を生成する。生成処理については、図４に関連して詳述する。

以上に示したように、変換元ソースである日立ソース２００から変換元ＡＳＴを生成し、変換元ＡＳＴから変換先ＡＳＴを生成し、変換先ＡＳＴから変換先ソースであるＩＢＭソース２０２を生成することにより、日立ソース２００がＩＢＭソース２０２に変換される。

【００１３】

言語変換処理を実現するためには、１．変換元ソースから変換元ＡＳＴを生成するためのルール（以下、「第１ＡＳＴルール」とよぶ）、２．変換元ＡＳＴから変換先ＡＳＴを生成するためのルール（以下、「第２ＡＳＴルール」とよぶ）、３．変換先ＡＳＴから変換先ソースを生成するためのルール（以下、「ソース生成ルール」とよぶ）の３種類のルールが必要である。各種のプログラミング言語についてこれらのルールをライブラリとして提供することにより、さまざまなプログラミング言語間における言語変換処理が実現される。

【００１４】

図２は、ソースコード処理装置１００の機能ブロック図である。

ここに示す各ブロックは、ハードウェア的には、コンピュータのＣＰＵをはじめとする素子や機械装置で実現でき、ソフトウェア的にはコンピュータプログラム等によって実現されるが、ここでは、それらの連携によって実現される機能ブロックを描いている。したがって、これらの機能ブロックはハードウェア、ソフトウェアの組合せによっていろいろなかたちで実現できることは、当業者には理解されるところである。

ソースコード処理装置１００は、図１に示したパーサ、プロセッサ、ライタの機能を担う装置である。本実施例においては、ソースコード処理装置１００の主たる機能は、コンピュータ上のソフトウェアとして実現されるものとして説明する。より具体的には、本実施例におけるソースコード処理装置１００の各機能は、主として、ＪＡＶＡ（登録商標）やScalaにより記述され、ＪＶＭ（Java Virtual Machine）上において実行されるコンピュータプログラムにより実現される。

【００１５】

ソースコード処理装置１００は、ＵＩ（ユーザインタフェース）部１１０、データ処理部１２０およびデータ保持部１４０を含む。

ＵＩ部１１０は、ユーザインタフェース処理を担当する。データ処理部１２０は、ＵＩ部１１０やデータ保持部１４０から取得されたデータを元にして各種のデータ処理を実行する。データ処理部１２０は、ＵＩ部１１０とデータ保持部１４０との間のインタフェースの役割も果たす。データ保持部１４０は、各種データを保持するための記憶領域である。

【００１６】

ＵＩ部１１０：

ＵＩ部１１０は、入力部１１２と出力部１１６を備える。入力部１１２は、ユーザの各種入力を検出したり、ファイル等のデータを外部装置から取得する。出力部１１６は、各

種情報を表示したり、ファイル等のデータを外部装置に送信する。入力部１１２は、コード取得部１１４を含む。コード取得部１１４は、図１の日立ソース２００のような変換元ソースを取得する。出力部１１６は、コード出力部１１８を含む。コード出力部１１８は、図１のＩＢＭソース２０２のような変換先ソースを出力する。

【００１７】

データ処理部１２０：

データ処理部１２０は、第１構文モデル生成部１２２、第２構文モデル生成部１２４、コード生成部１２６、検査部１２８および注釈ノード設定部１３０を含む。第１構文モデル生成部１２２は、第１ＡＳＴルール１５２を含む。第１構文モデル生成部１２２は、第１ＡＳＴルール１５２を参照して、変換元ソースから変換元ＡＳＴを生成する。第２構文モデル生成部１２４は、第２ＡＳＴルール１５４を含む。第２構文モデル生成部１２４は、第２ＡＳＴルール１５４を参照して、変換元ＡＳＴから変換先ＡＳＴを生成する。コード生成部１２６は、ソース生成ルール１５６を含む。コード生成部１２６は、ソース生成ルール１５６を参照して、変換先ＡＳＴから変換先ソースを生成する。

【００１８】

検査部１２８は、変換元ＡＳＴ、あるいは、変換先ＡＳＴのノードを走査して、所定の「検出条件」に合致する箇所を特定する。検査部１２８は、コーディングルール１５８を含む。検出条件の全部または一部は、コーディングルール１５８として定義される。検出条件については、図５以降に関連して詳述する。注釈ノード設定部１３０は、検出条件に合致する箇所が検出されたとき、「注釈ノード」をＡＳＴに設定する。注釈ノード設定部１３０は、注釈ノード設定ルール１６０を含む。注釈ノード設定ルール１６０には、検出条件と注釈ノードが対応づけられている。ある検出条件に合致する箇所が検出されたとき、どのような注釈ノードを設定すべきか、はこの注釈ノード設定ルール１６０に設定されている。注釈ノードの設定の詳細については、図６以降に関連して詳述する。検出条件と注釈ノードは、ＡＳＴを対象として所定のコーディングルールが守られているか、プログラミング構造的に不適切な記載箇所がないか、を判断するために導入される。

【００１９】

データ保持部１４０：

データ保持部１４０は、第１構文モデル保持部１４２と第２構文モデル保持部１４４を含む。第１構文モデル保持部１４２は、第１構文モデル生成部１２２が生成した変換元ＡＳＴを保持する。第２構文モデル保持部１４４は、第２構文モデル生成部１２４が生成した変換先ＡＳＴを保持する。

【００２０】

解析処理（Ｓ１）、すなわち、パーサとしての機能は、主として、入力部１１２と第１構文モデル生成部１２２により実現される。変換処理（Ｓ２）、すなわち、プロセッサとしての機能は、主として、第２構文モデル生成部１２４により実現される。生成処理（Ｓ３）、すなわち、ライタとしての機能は、主として、コード生成部１２６や出力部１１６により実現される。

【００２１】

ソースコード処理装置１００は、変換元ソースを変換先ソースに言語変換する「ソースコード変換装置」としての機能のほかに、検出条件を設定することにより、ソースコードの記述内容を検査する「ソースコード検査装置」としての機能も備える。以下、「ソースコード変換装置」としての機能と「ソースコード検査装置」としての機能を分けて説明する。

【００２２】

［ソースコード変換装置としての機能］

図３は、日立ソース２００から変換元ＡＳＴを生成する過程を説明するための模式図である。

同図に示す日立ソース２００は、変数「A」と変数「B」を比較対象とし、「AとBが共に１」という条件が成立するときには、変数「FLAG」に文字列「Y」を設定し、それ以外の

ときには変数「FLAG」に文字列「N」を設定する、という内容を示している。１行目の構文「EVALUATE A B」は、「A」と「B」を比較対象とする、という処理内容を示している。

【００２３】

コード取得部１１４が日立ソース２００を取得すると、第１構文モデル生成部１２２は、まず、構文「EVALUATE A B」を取得する。この構文の構成要素は、「EVALUATE」、「A」、「B」である。第１構文モデル生成部１２２は、まず、第１ＡＳＴルールに基づき、「EVALUATE」に対応する変換元ノードEvaluateStatement２０６を設定し、「EVALUATE」の変数リスト宣言に対応する変換ノードEvaluateTagList２０８を設定する。構成要素とは、ソースコードに記述されている変数や定数、宣言や関数といったステートメントであり、ノードとはそれらの構成要素に対応づけられるＡＳＴの要素である。ノードの中には、EvaluateStatement２０６のように、ソースコード中において明示的に記述されている構成要素に対応して設定されるノードもあれば、EvaluateTagList２０８のように、EvaluateStatement２０６のような他ノードに付随して設定されるノードもある。

【００２４】

「EVALUATE」の引数「A」、「B」に対応して、Expression:"A"２１０とExpression:"B"２１２という２つのノードがEvaluateTagList２０８の子ノードとして設定される。更に、引数「A」、「B」の並置を示すためのノード「None２１４」も設定される。日立ソース２００では、「EVALUATE A B」のように引数をそのまま並べて書くことができるが、ＩＢＭソース２０２では、「EVALUATE A ALSO B」のように引数「A」、「B」の間に「ALSO」という構成要素が記述される。日立ソース２００から変換元ＡＳＴ２０４を生成するときには、Expression:"A"２１０とExpression:"B"２１２だけでなく、None２１４というノードも設定される。より具体的には、「EVALUATE」という表現に対しては、EvaluateStatement２０６、EvaluateTagList２０８および引数に対応するノードと引数をつなぐためのノード（None２１４）が第１ＡＳＴルールにおいて対応づけられている。第１構文モデル生成部１２２はこの第１ＡＳＴルールにしたがって、「EVALUATE」という表現に対応してしかるべき変換元ノードを組み立てる。

変換元ＡＳＴ２０４が標準的・統一的な構文モデルであるならば、変換元ソースのプログラミング言語ごとに第１ＡＳＴルールを用意すれば、任意の変換元ソースから標準的な構文モデルとしての変換元ＡＳＴ２０４を生成できる。

【００２５】

図４は、変換元ＡＳＴから変換先ＡＳＴ、ＩＢＭソース２０２を生成する過程を説明するための模式図である。

まず、第２構文モデル生成部１２４は、変換元ＡＳＴ２０４から変換先ＡＳＴ２２０を生成する。変換先ソースであるＩＢＭソース２０２では、「EVALUATE A ALSO B」のように「EVALUATE」の引数は構成要素「ALSO」によってつなげられる。そこで、第２構文モデル生成部１２４は、変換元ＡＳＴ２０４の変換元ノードNone２１４を、変換先ノードALSO２２２に変換する。変換元ノードと変換先ノードの対応づけは、プログラミング言語ごとに第２ＡＳＴルールとして設定されている。第２構文モデル生成部１２４はこの第２ＡＳＴルールにしたがって変換元ＡＳＴ２０４から変換先ＡＳＴ２２０を生成する。

変換元ＡＳＴ２０４が標準的・統一的な構文モデルであるならば、変換先ソースのプログラミング言語ごとに第２ＡＳＴルールを用意すれば、任意の変換先ＡＳＴへの変換が可能となる。

【００２６】

コード生成部１２６は、各変換先ノードごとにＩＢＭソース２０２の該当構成要素を特定し、特定した構成要素を変換先ＡＳＴ２２０の構造にしたがって組み立てる。このようなルールは、ソース生成ルールとして定義されている。こうして、ALSO表現を含まない日立ソース２００からALSO表現を含むＩＢＭソース２０２が生成される。この結果、ＩＢＭ・ＣＯＢＯＬ用の解析ツールにより日立ソース２００を解析可能となる。

【００２７】

ＩＢＭソース２０２を変換元ソース、日立ソース２００を変換先ソースとする場合にも

基本的な考え方は同様である。ＩＢＭソース２０２から変換元ＡＳＴを作るときには、構成要素「ALSO」からノード「ALSO２２２」を設定する。変換元ＡＳＴを日立ＣＯＢＯＬ用の変換先ＡＳＴに変換するときには、ALSO２２２はNone２１４に変換される。あるいは、ALSO２２２に対応するノードを変換先ＡＳＴには設定しないとしてもよい。いずれにしても、変換先ＡＳＴにおいて、ALSO２２２に対応するノードが無効化されればよい。コード生成部１２６は、None２１４に対応する構成要素を日立ソース２００に記述しない。このような処理方法により、ＩＢＭソース２０２の「EVALUATE A ALSO B」は、日立ソース２００においては「EVALUATE A B」として表現されることになる。

ここでは、日立ソース２００とＩＢＭソース２０２の違いの例としてALSO２２２の有無を挙げたが、このほかのさまざまな言語間の差異についても、第１ＡＳＴルール、第２ＡＳＴルールおよびソース生成ルールを適切に定義することにより変換可能となる。

【００２８】

［ソースコード検査装置としての機能］

図５は、「int num,iNumber」という変数宣言文から生成されるＡＳＴを示す図である。

検査対象となるＡＳＴは、変換元ＡＳＴであっても変換先ＡＳＴであってもよい。ここでは、変換元ＡＳＴを対象として検査を実行するとして説明する。検査の内容はユーザが任意に設定できる。たとえば、業務システムの開発・保守について、以下のようなコーディングルールを設定したとする。

Ｒ１．int型の変数名は、文字「i」から始まること

Ｒ２．変数宣言においては、１回の型宣言で複数の変数を定義しないこと

Ｒ３．変数名は４文字以上であること

【００２９】

コーディングルールの制定は、ソースコードの可読性を高めたり、修正ミスを防ぐ上で重要である。特に、複数のソフトウェアエンジニアが関わる大規模な業務システムの開発において適切なコーディングルールを制定することは重要である。

【００３０】

同図の宣言文「int num,iNumber」において、変数「num」は、その先頭文字が「n」であるため、上記のルールＲ１を満たしていない。また、一つのint型宣言で「num」と「iNumber」という２つの変数をまとめて定義しているため、ルールＲ２も満たしていない。更に、変数「num」の文字数は３文字しかないため、ルールＲ３も満たしていない。ソースコード処理装置１００は、コーディングルールが実際に守られているか、また、コーディングルールが守られていない箇所はどこか、を検査できる。

【００３１】

同図に示すＡＳＴ２３０は、宣言文「int num,iNumber」に対応する。第１構文モデル生成部１２２は、変数の宣言文に出会うと、まず、第１ＡＳＴルールに基づき、LocalVariableDeclaration２３２をＡＳＴ２３０に設定する。子ノードとして、int型を示すノードであるIntType２３４と変数リストを示すノードであるVariableDeclarators２３６が設定される。変数「num」、変数「iNumber」に対しては、VariableDeclarator２３８とIdentifier:"num"２４０、VariableDeclarator２４２とIdentifier:"iNumber"２４４がそれぞれ設定されている。

【００３２】

図６は、ルールＲ１に対応して注釈ノードが設定されたＡＳＴ２３０を示す図である。

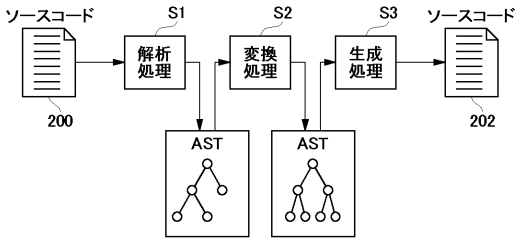
検査部１２８には、上述したルールＲ１～Ｒ３が検出条件として設定される。検査部１２８は、まず、ルールＲ１「int型の変数名は、文字「i」から始まること」に違反している変数ノードをＡＳＴ２３０から検索する。検査部１２８は、ＡＳＴ２３０を走査して、IntType２３４を見つけると、その親ノードであるLocalVariableDeclaration２３２の子ノードの中からidentifierノードを探し、変数名をチェックする。変数「num」はルールＲ１に違反している。注釈ノード設定部１３０はIdentifier:"num"２４０に注釈ノード２５０を設定する。

【００３３】
注釈ノード設定部１３０は、最終的に、ＡＳＴ２３０中の注釈ノード２５０を収集し、リスト化して出力部１１６に出力させる。また、コード生成部１２６は、注釈ノード２５０を含むＡＳＴ２３０からソースコードを生成してもよい。コード生成部１２６はソースコード中において、注釈ノード２５０の対応する箇所に注釈内容を示すコメントを付記してもよい。
【００３４】
図７は、更に、ルールＲ２、Ｒ３に対応して注釈ノードが設定されたときのＡＳＴ２３０を示す図である。
検査部１２８は、次に、ルールＲ２「変数宣言においては、１回の型宣言で複数の変数を定義しないこと」に違反している箇所を検索する。検査部１２８は、ＡＳＴ２３０を走査して、LocalVariableDeclaration２３２を見つけると、その子ノードの中からidentifierノードを探し、宣言されている変数の数をチェックする。ステートメント「int num,iNumber」はルールＲ２に違反している。注釈ノード設定部１３０はVariableDeclarators２３６に注釈ノード２５４を設定する。
【００３５】
検査部１２８は、ルールＲ３「変数名は４文字以上であること」に違反している箇所を検索する。検査部１２８は、ＡＳＴ２３０を走査して、LocalVariableDeclaration２３２を見つけると、その子ノードの中からidentifierノードを探し、宣言されている変数の文字数をチェックする。変数「num」はルールＲ３に違反しているため、注釈ノード設定部１３０はIdentifier:"num"２４０に注釈ノード２５２を設定する。
【００３６】
検出条件は、コーディングルールに関わる条件に限らない。たとえば、コンピュータプログラムの構造として満たされるべき条件を検出条件として設定してもよい。以下、２例を示す。
【００３７】
例１．特定の処理の開始時に記載されるステートメントと終了時に記載されるべきステートメントの確認
たとえば、Ｃ＋＋の場合、「new」演算子によりメモリの動的確保がなされる。メモリを解放する演算子は「delete」である。new演算子で確保したメモリは、使用後には、deleteによって解放しなければならない。また、「open」演算子によりファイルディスクリプタを開いたあとは、「close」演算子により同ファイルディスクリプタを閉じなければならない。このように、「メモリの動的確保」や「ファイルディスクリプタの使用」といった特定の処理の前後においては、所定のステートメントが一対として記述されなければならない。以下、newやopenのように特定の処理の開始時に記述されるべき構成要素を「開始要素」、終了時に記述されるべき構成要素を「終了要素」とよぶ。１つのソースファイルにおいては開始要素と終了要素の数は同数となるべきである。開始要素に対応するノードを「先端ノード」、終了要素に対応するノードを「終端ノード」とよぶことにすると、ＡＳＴにおいては、先端ノードの数と終端ノードの数は同数となるべきである。
このような観点から、ソースコード処理装置１００は先端ノードと終端ノードに基づいてプログラム構造上の不備がないかを検査してもよい。
【００３８】
ここでは、openを開始要素、closeを終端要素として説明する。まず、検査部１２８はＡＳＴを走査し、openに対応する先端ノードを検出すると、先端ノード数をインクリメントする。また、closeに対応する終端ノードを検出すると、終端ノード数をインクリメントする。ＡＳＴの全ノードを走査したとき、先端ノード数と終端ノード数が同数でなかったときには、ソースコードの記述に欠陥があるのかもしれない。先端ノード数が終端ノード数よりも多いときには、開いたまま閉じられていないファイルディスクリプタが存在する可能性がある。このような場合、注釈ノード設定部１３０その旨を警告するための注釈ノードを設定する。

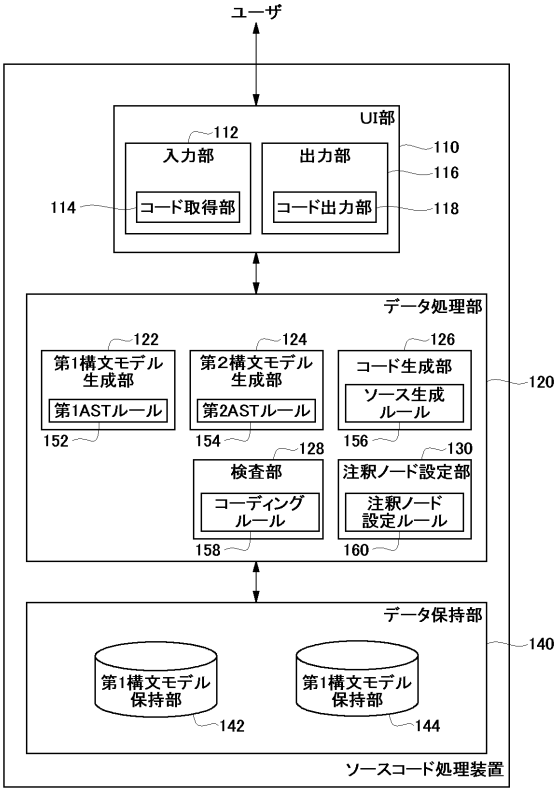
【００３９】
例２．オーバーライドの範囲の確認
あるclassAにおいてそのメンバ関数method1()を定義したとする。このクラスを継承するclassBは、独自にメンバ関数method1()を定義しなくても、classAのメンバ関数method1()を自らのメンバ関数として提供できる。いわば、classBはclassAのメンバ関数を借用することになる。classA.method1()をコールしたときに実行される処理内容と、classB.method1()をコールしたときに実行される処理内容は全く同じとなる。したがって、classAに定義されているmethod1()の内容を修正すると、classA.method1()だけでなく、classB.method1()の挙動も変化する。
【００４０】
一方、classBにおいて、独自に、同名のmethod1()を再定義してもよい。これをオーバーライド（override）とよぶ。この場合、classA.method1()をコールしたときに実行される処理内容はclassAのmethod1()により定義され、classB.method1()をコールしたときに実行される処理内容はclassBのmethod1()により定義される。classAとclassBは、同名のインタフェースmethod1()を提供しつつ、その処理内容をclassAとclassBで異ならせることができる。オーバーライドというプログラミングテクニックは、classBのmethod1()と、classAのmethod1()の処理内容がほとんど同じでありながら若干異なる場合に利用されることが多い。
【００４１】
classBでmethod1()を再定義、すなわち、オーバーライドしている場合、classA.method1()の内容を修正すると、当然ながらclassA.method1()の挙動は変化するが、classB.method1()の挙動は変化しない。classA.method1()の内容の修正にあわせて、classB.method1()も変化させたい場合には、classB.method1()にも、同様の修正を忘れずに施す必要がある。
このような観点から、ソースコード処理装置１００は、各クラスのメンバ関数について、そのオーバーライドされている箇所を特定することにより、ソースコード解析を支援する。
【００４２】
具体的には、検査部１２８は、まず、classAを継承しているサブクラス群を特定し、これらのサブクラスにおいて定義されるメンバ関数を検索する。そして、この中からmethod1()という名前のメンバ関数を特定する。classAのサブクラスにおいてmethod1()が定義されていれば、ここでオーバーライドとなっていることがわかる。注釈ノード設定部１３０は該当箇所に注釈ノードを設定する。classAのmethod1()を修正したときには、このようにして設定された注釈ノードを確認することにより、修正すべきかもしれないメンバ関数の場所を特定できる。
【００４３】
注釈ノード設定部１３０は、最終的に、注釈ノード群をリスト化する。このリストには、注釈ノードの内容、注釈ノードが所属するソースファイル名、行番号、桁位置、行テキスト、親ノード名等、注釈ノードの位置を示す情報も含まれることが好ましい。ユーザは、この注釈ノードのリストにより、膨大なソースコードの中から重点的に確認すべき箇所を把握できるため、ソースコードの開発・保守が容易となる。
【００４４】
以上、実施例に基づいて、ソースコード処理装置１００を説明した。
ソースコード処理装置１００は、「第１ＡＳＴルール」、「第２ＡＳＴルール」、「ソース生成ルール」を用意すれば、さまざまなプログラミング言語で記述されたソースコードを別のプログラミング言語で記述されたソースコードに変換できる。このため、さまざまなプログラミング言語が混在する業務システムにさまざまな解析ツールを適用しやすくなる。ソースコード同士を直接変換するのではなく、ソースコードの論理構造を反映させたＡＳＴ同士の変換であるため、変換元ソースの論理構造を変換先ソースに反映させやすくなる。

【００４５】
　また、検出条件にしたがってＡＳＴから該当箇所を特定し、注釈コードを設定することにより、ソースコードの解析が容易となる。ソースコードのテキスト検索ではなく、ソースコードの論理構造を反映するＡＳＴを対象とした検査であるため、膨大なソースコードの中から、コーディングルール違反をしている箇所やコーディングミスの可能性がある箇所を適確に見つけ出しやすくなる。
【００４６】
　以上、本発明について実施例をもとに説明した。実施の形態は例示であり、それらの各構成要素や各処理プロセスの組み合わせにいろいろな変形例が可能なこと、またそうした変形例も本発明の範囲にあることは当業者に理解されるところである。
【図面の簡単な説明】
【００４７】
【図１】ソースコードの言語変換過程を示す模式図である。
【図２】ソースコード処理装置の機能ブロック図である。
【図３】日立ソースから変換元ＡＳＴを生成する過程を説明するための模式図である。
【図４】変換元ＡＳＴから変換先ＡＳＴ、ＩＢＭソースを生成する過程を説明するための模式図である。
【図５】「int num,iNumber」という変数宣言文から生成されるＡＳＴを示す図である。
【図６】ルールＲ１に対応して注釈ノードが設定されたＡＳＴを示す図である。
【図７】更に、ルールＲ２、Ｒ３に対応して注釈ノードが設定されたときのＡＳＴを示す図である。
【符号の説明】
【００４８】
　１００　ソースコード処理装置、　１１０　ＵＩ部、　１１２　入力部、　１１４　コード取得部、　１１６　出力部、　１１８　コード出力部、　１２０　データ処理部、　１２２　第１構文モデル生成部、　１２４　第２構文モデル生成部、　１２６　コード生成部、　１２８　検査部、　１３０　注釈ノード設定部、　１４０　データ保持部、　１４２　第１構文モデル保持部、　１４４　第２構文モデル保持部、　２００　日立ソース、　２０２　ＩＢＭソース、　２０４　変換元ＡＳＴ、　２２０　変換先ＡＳＴ。

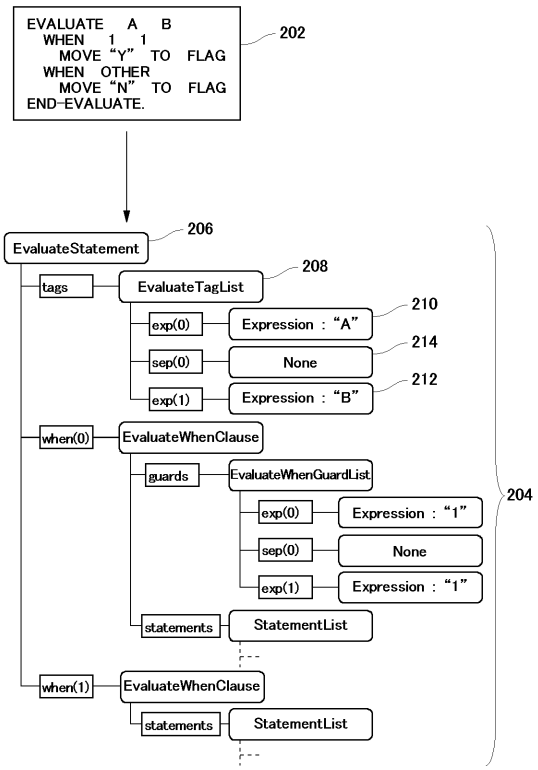
【図１】


ソースコード
S1
解析処理
S2
変換処理
S3
生成処理
ソースコード
200
AST
AST
202

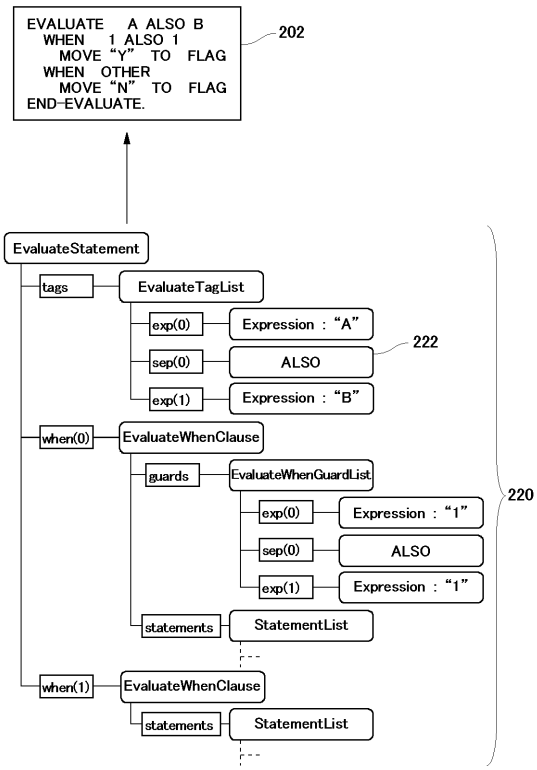
【図２】


ユーザ
UI部
112
入力部
出力部
110
116
114
コード取得部
コード出力部
118
データ処理部
122
124
126
第1構文モデル生成部
第2構文モデル生成部
コード生成部
120
第1ASTルール
第2ASTルール
ソース生成ルール
152
154
156
128
130
検査部
注釈ノード設定部
コーディングルール
注釈ノード設定ルール
158
160
データ保持部
140
第1構文モデル保持部
第1構文モデル保持部
142
144
ソースコード処理装置
100

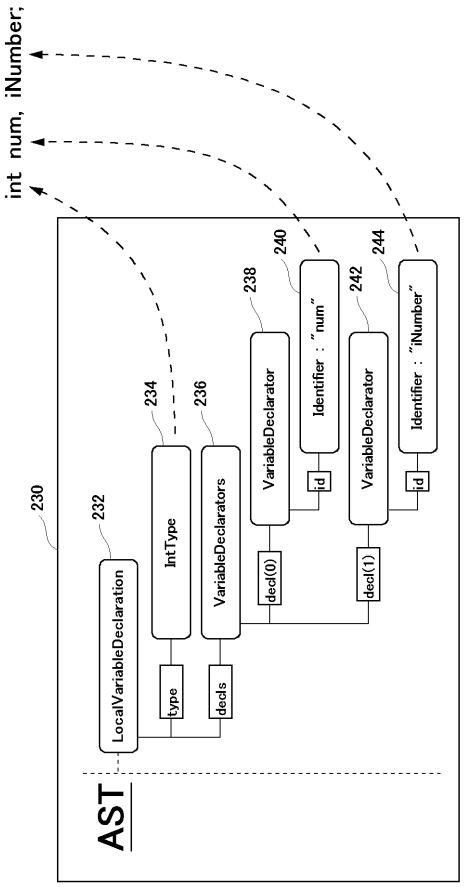
【図３】


EVALUATE A B
WHEN 1 1
MOVE “Y” TO FLAG
WHEN OTHER
MOVE “N” TO FLAG
END-EVALUATE.
202
EvaluateStatement
206
tags
EvaluateTagList
208
exp(0)
Expression : “A”
210
sep(0)
None
214
exp(1)
Expression : “B”
212
when(0)
EvaluateWhenClause
guards
EvaluateWhenGuardList
exp(0)
Expression : “1”
sep(0)
None
exp(1)
Expression : “1”
204
statements
StatementList
when(1)
EvaluateWhenClause
statements
StatementList

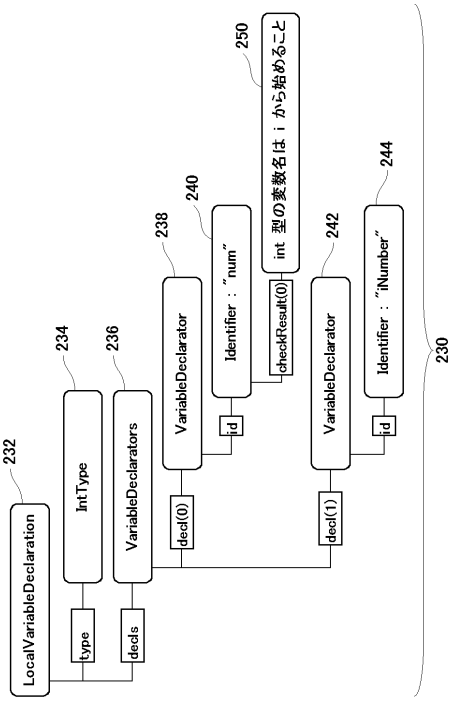
【図４】


EVALUATE A ALSO B
WHEN 1 ALSO 1
MOVE “Y” TO FLAG
WHEN OTHER
MOVE “N” TO FLAG
END-EVALUATE.
202
EvaluateStatement
tags
EvaluateTagList
exp(0)
Expression : “A”
sep(0)
ALSO
222
exp(1)
Expression : “B”
when(0)
EvaluateWhenClause
guards
EvaluateWhenGuardList
exp(0)
Expression : “1”
sep(0)
ALSO
exp(1)
Expression : “1”
220
statements
StatementList
when(1)
EvaluateWhenClause
statements
StatementList

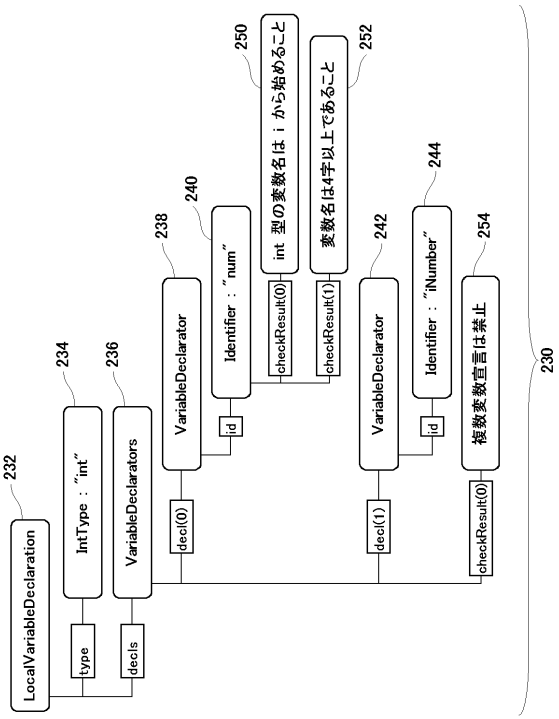
【図５】


int num, iNumber;
AST
230
LocalVariableDeclaration
232
IntType
234
VariableDeclarators
236
VariableDeclarator
238
Identifier : "num"
240
VariableDeclarator
242
Identifier : "iNumber"
244
type
decls
id
decl(0)
decl(1)

【図６】


LocalVariableDeclaration
232
IntType
234
VariableDeclarators
236
VariableDeclarator
238
Identifier : "num"
240
int 型の変数名は i から始めること
250
VariableDeclarator
242
Identifier : "iNumber"
244
230
type
decls
id
decl(0)
decl(1)
checkResult(0)

【図７】


LocalVariableDeclaration
232
IntType : "int"
234
VariableDeclarators
236
VariableDeclarator
238
Identifier : "num"
240
int 型の変数名は i から始めること
250
変数名は4字以上であること
252
VariableDeclarator
242
Identifier : "iNumber"
244
複数変数宣言は禁止
254
230
type
decls
id
decl(0)
decl(1)
checkResult(0)
checkResult(1)