(19)日本国特許庁(JP)　(12)特　許　公　報(B2)　(11)特許番号
特許第5600301号
(P5600301)
(45)発行日　平成26年10月1日(2014.10.1)　(24)登録日　平成26年8月22日(2014.8.22)

(51)Int.Cl.　F I
G06F 9/45 (2006.01)　G06F 9/44 322M

請求項の数 44 (全 49 頁)

(21)出願番号	特願2010-548175 (P2010-548175)
(86)(22)出願日	平成21年2月27日 (2009.2.27)
(65)公表番号	特表2011-513824 (P2011-513824A)
(43)公表日	平成23年4月28日 (2011.4.28)
(86)国際出願番号	PCT/GB2009/000552
(87)国際公開番号	WO2009/106843
(87)国際公開日	平成21年9月3日 (2009.9.3)
審査請求日	平成24年1月27日 (2012.1.27)
(31)優先権主張番号	08152165.0
(32)優先日	平成20年2月29日 (2008.2.29)
(33)優先権主張国	欧州特許庁 (EP)
(31)優先権主張番号	61/032,547
(32)優先日	平成20年2月29日 (2008.2.29)
(33)優先権主張国	米国 (US)

(73)特許権者　507345538
アイティーアイ　スコットランド　リミテッド
イギリス国　グラスゴー　ウェスト　ジョージ　ストリート　１９１　５階
(74)代理人　100114775
弁理士　高岡　亮一
(72)発明者　リトル，ダグラス
イギリス国，グラスゴー　ジー１　２アールエヌ，１４１　ウエスト　ナイル　ストリート，カレドニアン　スイート，セント　アンドリュー　ハウス，スラム　ゲームス　リミテッド

最終頁に続く

(54)【発明の名称】システム表現およびハンドリング技術

(57)【特許請求の範囲】
【請求項１】
第１のデータ構造の実装を生成するためのコンピュータプログラミングツールであって、前記第１のデータ構造はコンピュータプログラミング言語の少なくとも一部を表し、前記言語は構文規則の組を満たす複数の構文的要素と複数のリンクされたノードを備える前記第１のデータ構造を備え、そのようなノードはルートノード、前記ルートノードに直接リンクされた複数の第１階層のノード、および１つ以上の前記ノードを介して前記ルートノードに間接的にリンクされた複数の下層ノードを備え、前記ノードは前記言語の構文的要素と、前記ノード間の実装およびインターフェースではなく代替可能性の継承の経路を表す前記ノード間のリンクのパターンを表し、前記ツールは、
前記第１のデータ構造またはその記述を記憶するように動作可能なメモリと、
前記第１のデータ構造から、前記第１のデータ構造に対応する第２のデータ構造またはその記述を生成するように構成された少なくとも１つのプロセッサと、を備え、
前記第２のデータ構造は、前記第２のデータ構造の内、前記ルートノードに直接リンクされたルートノード以外の前記第２のデータ構造の全てのノードを有する前記第１のデータ構造のノードに対応するノードを備え、
前記第２のデータ構造の前記ノード間の前記リンクは前記ノード間の代替可能性の継承の経路、実装およびインターフェースを表し、
前記プロセッサはさらに、前記１つの第１のデータ構造のリンクパターンに基づいて、前記第１のデータ構造により表される前記代替可能性の継承への準拠を確立するための前

記第２のデータ構造を使用するその後の処理操作中に、前記第２のデータ構造のノードに関連して強制される前記第１のデータ構造の代替可能性の規則を定義する実装規則を生成するように動作可能であり、
前記第２のデータ構造と前記実装規則は、前記第１のデータ構造の前記１つの実装を形成する、コンピュータプログラミングツール。
【請求項２】
前記第１のデータ構造は少なくとも部分的に不均一なツリー構造である、請求項１に記載のコンピュータプログラミングツール。
【請求項３】
前記第１のデータ構造は少なくとも部分的に有向非巡回グラフ構造である、請求項２に記載のコンピュータプログラミングツール。
【請求項４】
前記言語は複数のトークンから構成され、前記言語の各前記構文的要素は、前記トークンの群または前記トークンの所定の組合せを表わす、請求項１に記載のコンピュータプログラミングツール。
【請求項５】
前記第１のデータ構造は、言語拡張を表わす更に別のリンクされたノードを有し、前記言語拡張は前記コンピュータプログラミング言語に対する拡張である、請求項１に記載のコンピュータプログラミングツール。
【請求項６】
前記第１のデータ構造の前記更に別のノードは、前記言語拡張のそれぞれの構文的要素を表す、請求項５に記載のコンピュータプログラミングツール。
【請求項７】
前記第１のデータ構造の前記更に別のノードの内の少なくとも一つは、前記第１のデータ構造の他のノードの幾つかに対して代替可能として定義される、請求項５に記載のコンピュータプログラミングツール。
【請求項８】
前記実装規則は、前記第１のデータ構造によって表わされる前記第１のデータ構造の前記１つ以上の更に別のノードの前記代替可能性の関係への準拠を確立するために、前記実装を利用するその後の処理操作中に、前記第１のデータ構造の前記１つ以上の更に別のノードに対応する前記第２のデータ構造のノードに関連して強制される代替可能性の規則を定義する、請求項５に記載のコンピュータプログラミングツール。
【請求項９】
前記言語拡張は前記コンピュータプログラミング言語以外の言語の少なくとも一部である、請求項５に記載のコンピュータプログラミングツール。
【請求項１０】
コンピュータプログラミングツールによって第１のデータ構造の実装を生成するための実装方法であって、前記第１のデータ構造はコンピュータプログラミング言語の少なくとも一部を表し、前記言語は構文規則の組を満たす複数の構文的要素と複数のリンクされたノードを備える前記第１のデータ構造を備え、そのようなノードはルートノード、前記ルートノードに直接リンクされた複数の第１階層のノード、および１つ以上の前記ノードを介して前記ルートノードに間接的にリンクされた複数の下層ノードを備え、前記ノードは前記言語の構文的要素と、前記ノード間の実装およびインターフェースではなく代替可能性の継承の経路を表す前記ノード間のリンクのパターンを表し、前記実装方法は、
コンピュータプログラミングツールが備えるプロセッサが、前記第１のデータ構造またはその記述を受取ることと、
前記プロセッサが、前記第１のデータ構造の実装を生成することとを含み、当該生成することは、
前記第１のデータ構造に対応する第２のデータ構造またはその記述を生成することであって、前記第２のデータ構造は、前記第２のデータ構造の内、前記ルートノードに直接

リンクされたルートノード以外の前記第２のデータ構造の全てのノードを有する前記第１のデータ構造のノードに対応するノードを備え、前記第２のデータ構造の前記ノード間の前記リンクは代替可能性の継承の経路、前記ノード間の実装およびインターフェースを表す、ことと、

前記第１のデータ構造により表される前記代替可能性の継承への準拠を確立するための前記第２のデータ構造を使用するその後の処理操作中に、前記実装を利用するその後の処理操作中に前記第２のデータ構造のノードに関連して強制される前記第１のデータ構造の代替可能性の規則を定義する実装規則を生成することと、を含む、実装方法。

【請求項１１】

コンピューティングデバイスで実行されたときに、前記コンピューティングデバイスにコンピュータプログラミングツールを用いて第１のデータ構造の実装を生成する方法を実行させる実装コンピュータプログラムを記憶した記憶媒体であって、

前記第１のデータ構造はコンピュータプログラミング言語の少なくとも一部を表し、前記言語は構文規則の組を満たす複数の構文的要素と複数のリンクされたノードを備える前記第１のデータ構造を備え、そのようなノードはルートノード、前記ルートノードに直接リンクされた複数の第１階層のノード、および１つ以上の前記ノードを介して前記ルートノードに間接的にリンクされた複数の下層ノードを備え、前記ノードは前記言語の構文的要素と、前記ノード間の実装およびインターフェースではなく代替可能性の継承の経路を表す前記ノード間のリンクのパターンを表し、前記方法は、

コンピュータプログラミングツールが備えるプロセッサが、前記第１のデータ構造またはその記述を受取ることと、

前記プロセッサが、前記第１のデータ構造の実装を生成することとを含み、当該生成することは、

前記第１のデータ構造に対応する第２のデータ構造またはその記述を生成することであって、前記第２のデータ構造は、前記第２のデータ構造の内、前記ルートノードに直接リンクされたルートノード以外の前記第２のデータ構造の全てのノードを有する前記第１のデータ構造のノードに対応するノードを備え、前記第２のデータ構造の前記ノード間の前記リンクは前記ノード間の実装およびインターフェースではなく代替可能性の継承の経路を表す、ことと、

前記第１のデータ構造により表される前記代替可能性の継承への準拠を確立するための前記第２のデータ構造を使用するその後の処理操作中に、前記実装を利用するその後の処理操作中に前記第２のデータ構造のノードに関連して強制される前記第１のデータ構造の代替可能性の規則を定義する実装規則を生成することと、を含む、記憶媒体。

【請求項１２】

コード部分で動作するコード部分ハンドリングツールであって、

前記コード部分は第１のデータ構造の実装を生成するためのコンピュータプログラミングツールによって生成された実装の第２のデータ構造のインスタンスであり、前記第１のデータ構造はコンピュータプログラミング言語の少なくとも一部を表し、前記言語は構文規則の組を満たす複数の構文的要素と複数のリンクされたノードを備える前記第１のデータ構造を備え、そのようなノードはルートノード、前記ルートノードに直接リンクされた複数の第１階層のノード、および１つ以上の前記ノードを介して前記ルートノードに間接的にリンクされた複数の下層ノードを備え、前記ノードは前記言語の構文的要素と、前記ノード間の実装およびインターフェースではなく代替可能性の継承の経路を表す前記ノード間のリンクのパターンを表し、前記コンピュータプログラミングツールは、

前記第１のデータ構造またはその記述を記憶する操作可能なメモリと、

前記第１のデータ構造から、前記第１のデータ構造に対応する第２のデータ構造またはその記述を生成するように構成された少なくとも１つのプロセッサと、を備え、

前記第２のデータ構造は、前記第２のデータ構造の内、前記ルートノードに直接リンクされたルートノード以外の前記第２のデータ構造の全てのノードを有する前記第１のデータ構造のノードに対応するノードを備え、

前記第２のデータ構造の前記ノード間の前記リンクは代替可能性の継承の経路、前記ノード間の実装およびインターフェースを表し、
前記プロセッサはさらに、前記１つの第１のデータ構造のリンクパターンに基づいて、前記第１のデータ構造により表される前記代替可能性の継承への準拠を確立するための前記第２のデータ構造を使用するその後の処理操作中に、前記実装を利用するその後の処理操作中に前記第２のデータ構造のノードに関連して強制される前記第１のデータ構造の代替可能性の規則を定義する実装規則を生成するように操作可能であり、
前記第２のデータ構造と前記実装規則は、前記第１のデータ構造の前記１つの実装を形成し、前記コード部分は前記コンピュータプログラミング言語で表現され、前記コード部分ハンドリングツールは、
前記実装を記憶するためのメモリと、
前記実装に依存して候補コード部分で動作するように構成された少なくとも１つのプロセッサと、を備え、前記候補コード部分は前記第２のデータ構造のノードに対応するインスタンスノードを備える、コード部分ハンドリングツール。
【請求項１３】
前記第２のデータ構造は、そのインスタンスノードおよびその間のリンクが明示的に表現されていない入力形式であり、
前記少なくとも１つのプロセッサは、受け取った前記候補コード部分を、前記インスタンスノードおよびこれらの間のリンクが明示的に表現されている抽象化形式に変換するように構成されている、請求項１２に記載のコード部分ハンドリングツール。
【請求項１４】
前記候補コード部分は、前記コンピュータプログラミング言語で表現されたコード部分であり、
前記入力形式は、前記コード部分のテキスト版であり、
前記抽象化形式は、前記コード部分の抽象構文ツリーまたはグラフ版である、請求項１３に記載のコード部分ハンドリングツール。
【請求項１５】
前記少なくとも１つのプロセッサは、ディスプレイ上で、前記候補コード部分の視覚的表現の生成を命令するように動作可能である、請求項１２に記載のコード部分ハンドリングツール。
【請求項１６】
前記少なくとも１つのプロセッサは、前記候補コード部分の視覚的表現を前記抽象化形式での生成を命令するように動作可能である、請求項１３に記載のコード部分ハンドリングツール。
【請求項１７】
前記少なくとも１つのプロセッサは、前記実装に依存して前記候補コード部分を操作するように動作可能である、請求項１２に記載のコード部分ハンドリングツール。
【請求項１８】
前記少なくとも１つのプロセッサは、前記候補コード部分を前記抽象化形式で操作するように動作可能である、請求項１３に記載のコード部分ハンドリングツール。
【請求項１９】
前記少なくとも１つのプロセッサは、前記実装と照らし合わせてこのような操作を検証し、前記第２のデータ構造および前記実装規則の少なくとも一つに違反する操作を禁止するように動作可能である、請求項１７に記載のコード部分ハンドリングツール。
【請求項２０】
前記少なくとも１つのプロセッサは、前記第２のデータ構造および前記実装規則を遵守する操作を許可するように動作可能である、請求項１９に記載のコード部分ハンドリングツール。
【請求項２１】
前記少なくとも１つのプロセッサは、前記候補コード部分を増やすおよび／または前記

候補コード部分を減らすことを少なくとも含む操作を許可するように動作可能である、請求項１７に記載のコード部分ハンドリングツール。
【請求項２２】
このような操作は、前記候補コード部分に新しいインスタンスノードを追加することを含む、請求項１７に記載のコード部分ハンドリングツール。
【請求項２３】
このような操作は、前記候補コード部分からインスタンスノードを削除することを含む、請求項２２に記載のコード部分ハンドリングツール。
【請求項２４】
このような操作は、前記候補コード部分の特定のインスタンスノードをアノテートすることを含む、請求項２２に記載のコード部分ハンドリングツール。
【請求項２５】
このような操作は、前記候補コード部分の少なくとも一部に所定のプロセスを実行することを含む、請求項１７に記載のコード部分ハンドリングツール。
【請求項２６】
前記所定のプロセスは、前記コード部分ハンドリングツールによってアクセス可能なアクションの組内に定義される、請求項２５に記載のコード部分ハンドリングツール。
【請求項２７】
前記所定のプロセスは、所定の目的のために前記候補コード部分を最適化するように動作可能な最適化プロセスである、請求項２５に記載のコード部分ハンドリングツール。
【請求項２８】
前記少なくとも１つのプロセッサは、前記候補コード部分の前記インスタンスノードに依存してこのような操作を実行するように動作可能である、請求項２２に記載のコード部分ハンドリングツール。
【請求項２９】
前記少なくとも１つのプロセッサは、前記インスタンスノードの特定の型を識別し、前記識別されたインスタンスノードに依存してこのような操作を実行するように動作可能である、請求項２８に記載のコード部分ハンドリングツール。
【請求項３０】
前記第１のデータ構造は言語拡張を表す更に別のリンクされたノードを備え、前記言語拡張は前記コンピュータプログラミング言語に対する拡張であり、前記実装規則は、前記第１のデータ構造により表される前記第１のデータ構造の１つ以上の前記更に別のノードの前記代替可能性の関係への準拠を確立するために前記実装を使用するその後の処理操作中に、前記第１のデータ構造の１つ以上の前記更に別のノードに対応する前記第２のデータ構造に関連して強制される代替可能性の規則を定義し、前記インスタンスノードの前記特定の型は、前記言語拡張の前記更に別のノードのインスタンスノードである、請求項２９に記載のコード部分ハンドリングツール。
【請求項３１】
前記候補コード部分は前記言語拡張に起因する部分を含む、請求項３０に記載のコード部分ハンドリングツール。
【請求項３２】
前記少なくとも１つのプロセッサは、前記抽象化形式の前記候補コード部分をその対応する入力形式に変換するように動作可能である、請求項１３に記載のコード部分ハンドリングツール。
【請求項３３】
前記少なくとも１つのプロセッサは、前記実装に依存して前記候補コード部分を操作するように動作可能であり、かつ、前記少なくとも１つのプロセッサは、前記候補コード部分で実行された前記操作の前後に前記変換を実行するように動作可能である、請求項３２に記載のコード部分ハンドリングツール。
【請求項３４】

前記候補コード部分は、前記コンピュータプログラミング言語で表現されたコード部分であり、前記少なくとも１つのプロセッサは、オブジェクトコードとして前記操作の前後に前記候補コード部分を出力するように動作可能である、請求項３２に記載のコード部分ハンドリングツール。

【請求項３５】

前記少なくとも１つのプロセッサは、コンパイラとして構成される、請求項３４に記載のコード部分ハンドリングツール。

【請求項３６】

コード部分で動作するコード部分ハンドリング方法であって、

前記コード部分は第１のデータ構造の実装を生成するためにコンピュータプログラミングツールによって生成された実装の第２のデータ構造のインスタンスであり、前記第１のデータ構造はコンピュータプログラミング言語の少なくとも一部を表し、前記言語は構文規則の組を満たす複数の構文的要素と複数のリンクされたノードを備える前記第１のデータ構造を備え、そのようなノードはルートノード、前記ルートノードに直接リンクされた複数の第１階層のノード、および１つ以上の前記ノードを介して前記ルートノードに間接的にリンクされた複数の下層ノードを備え、前記ノードは前記言語の構文的要素と、前記ノード間の実装およびインターフェースではなく代替可能性の継承の経路を表す前記ノード間のリンクのパターンを表し、前記ツールは、

前記第１のデータ構造またはその記述を記憶する操作可能なメモリと、

前記第１のデータ構造から、前記第１のデータ構造に対応する第２のデータ構造またはその記述を生成するように構成された少なくとも１つのプロセッサと、を備え、

前記第２のデータ構造は、前記第２のデータ構造の内、前記ルートノードに直接リンクされたルートノード以外の前記第２のデータ構造の全てのノードを有する前記第１のデータ構造のノードに対応するノードを備え、

前記第２のデータ構造の前記ノード間の前記リンクは前記ノード間の実装およびインターフェースではなく代替可能性の継承の経路を表し、

前記プロセッサはさらに、前記１つの第１のデータ構造のリンクパターンに基づいて、前記第１のデータ構造により表される前記代替可能性の継承への準拠を確立するための前記第２のデータ構造を使用するその後の処理操作中に、前記実装を利用するその後の処理操作中に前記第２のデータ構造のノードに関連して強制される前記第１のデータ構造の代替可能性の規則を定義する実装規則を生成するように操作可能であり、

前記第２のデータ構造と前記実装規則は、前記第１のデータ構造の前記１つの実装を形成し、前記コード部分は前記コンピュータプログラミング言語で表現され、

前記方法は前記実装に依存して候補コード部分で動作することを含む、コード部分ハンドリング方法。

【請求項３７】

コンピューティングデバイスで実行されたときに、前記コンピューティングデバイスにコード部分で動作するコード部分ハンドリング方法を実行させるコード部分ハンドリング・コンピュータプログラムを記憶した記憶媒体であって、前記コード部分は第１のデータ構造の実装を生成するためのコンピュータプログラミングツールによって生成された実行の第２のデータ構造のインスタンスであり、前記第１のデータ構造はコンピュータプログラミング言語の少なくとも一部を表し、前記言語は構文規則の組を満たす複数の構文的要素と複数のリンクされたノードを備える前記第１のデータ構造を備え、そのようなノードはルートノード、前記ルートノードに直接リンクされた複数の第１階層のノード、および１つ以上の前記ノードを介して前記ルートノードに間接的にリンクされた複数の下層ノードを備え、前記ノードは前記言語の構文的要素と、前記ノード間の実装およびインターフェースではなく代替可能性の継承の経路を表す前記ノード間のリンクのパターンを表し、前記ツールは、

前記第１のデータ構造またはその記述を記憶する操作可能なメモリと、

前記第１のデータ構造から、前記第１のデータ構造に対応する第２のデータ構造または

その記述を生成するように構成された少なくとも１つのプロセッサと、を備え、
前記第２のデータ構造は、前記第２のデータ構造の内、前記ルートノードに直接リンクされたルートノード以外の前記第２のデータ構造の全てのノードを有する前記第１のデータ構造のノードに対応するノードを備え、
前記第２のデータ構造の前記ノード間の前記リンクは前記ノード間の代替可能性の継承の経路、実装およびインターフェースを表し、
前記プロセッサはさらに、前記１つの第１のデータ構造のリンクパターンに基づいて、前記第１のデータ構造により表される前記代替可能性の継承への準拠を確立するための前記第２のデータ構造を使用するその後の処理操作中に、前記実装を利用するその後の処理操作中に前記第２のデータ構造のノードに関連して強制される前記第１のデータ構造の代替可能性の規則を定義する実装規則を生成するように操作可能であり、
前記第２のデータ構造と前記実装規則は、前記第１のデータ構造の前記１つの実装を形成し、前記コード部分は前記コンピュータプログラミング言語で表現され、
前記方法は、前記実装に依存して候補コード部分で動作することを含む、記憶媒体。
【請求項３８】
コンピュータプログラミングツールによってコンピュータプログラミング言語を拡張する方法であって、
前記コンピュータプログラミングツールが備えるプロセッサが、前記言語の少なくとも一部を表わす第１のデータ構造またはその記述を取得することと、
前記プロセッサが、言語拡張を表わす更に別のリンクされたノードを含むように、前記第１のデータ構造またはその記述を適応させることと、
前記プロセッサが、前記適応された第１のデータ構造の実装を生成することと、を含み、
前記コンピュータプログラミングツールは第１のデータ構造の実装を生成するためのものであり、前記第１のデータ構造は前記コンピュータプログラミング言語の少なくとも一部を表し、前記言語は構文規則の組を満たす複数の構文的要素と複数のリンクされたノードを備える前記第１のデータ構造を備え、そのようなノードはルートノード、前記ルートノードに直接リンクされた複数の第１階層のノード、および１つ以上の前記ノードを介して前記ルートノードに間接的にリンクされた複数の下層ノードを備え、前記ノードは前記言語の構文的要素と、前記ノード間の実装およびインターフェースではなく代替可能性の継承の経路を表す前記ノード間のリンクのパターンを表し、
前記ツールは、
前記第１のデータ構造と当該第１のデータ構造の一つを記憶する操作可能なメモリと、
前記第１のデータ構造から、前記第１のデータ構造に対応する第２のデータ構造またはその記述を生成するように構成された少なくとも１つの前記プロセッサと、を備え、
前記第２のデータ構造は、前記第２のデータ構造の内、前記ルートノードに直接リンクされたルートノード以外の前記第２のデータ構造の全てのノードを有する前記第１のデータ構造のノードに対応するノードを備え、
前記第２のデータ構造の前記ノード間の前記リンクは前記ノード間の代替可能性の継承の経路、実装およびインターフェースを表し、
前記プロセッサはさらに、前記１つの第１のデータ構造のリンクパターンに基づいて、前記第１のデータ構造により表される前記代替可能性の継承への準拠を確立するための前記第２のデータ構造を使用するその後の処理操作中に、前記実装を利用するその後の処理操作中に前記第２のデータ構造のノードに関連して強制される前記第１のデータ構造の代替可能性の規則を定義する実装規則を生成するように操作可能であり、
前記第２のデータ構造と前記実装規則は、前記第１のデータ構造の前記１つの実装を形成し、
前記第１のデータ構造はさらに、言語拡張を表す更に別のリンクされたノードを備え、前記言語拡張は前記コンピュータプログラミング言語に対する拡張であり、
前記実装規則は、前記第１のデータ構造により表される前記第１のデータ構造の１つ以

上の前記更に別のノードの前記代替可能性の関係への準拠を確立するために前記実装を使用するその後の処理操作中に、前記第１のデータ構造の１つ以上の前記更に別のノードに対応する前記第２のデータ構造に関連して強制される代替可能性の規則を定義する、方法。

【請求項３９】

コンピュータプログラミングツールによってコンピュータプログラムを生成または適応させる方法であって、

前記コンピュータプログラミングツールが備えるプロセッサが、候補コード部分としてコンピュータプログラミング言語の少なくとも一部で表現された候補コンピュータプログラムをコード部分ハンドリングツールに入力することと、

前記プロセッサが、前記候補コード部分で動作するようにコード部分ハンドリングツールを使用することと、

前記プロセッサが、前記動作の結果、コンピュータプログラムを出力するように前記コード部分ハンドリングツールを使用することと、を含み、前記出力されるコンピュータプログラムは生成または適応されたコンピュータプログラムであり、

前記コード部分ハンドリングツールはコード部分で動作するためのものであり、前記コード部分は第１のデータ構造の実装を生成するためのコンピュータプログラミングツールによって生成された実行の第２のデータ構造のインスタンスであり、前記第１のデータ構造は前記コンピュータプログラミング言語の少なくとも一部を表し、前記言語は構文規則の組を満たす複数の構文的要素と複数のリンクされたノードを備える前記第１のデータ構造を備え、そのようなノードはルートノード、前記ルートノードに直接リンクされた複数の第１階層のノード、および１つ以上の前記ノードを介して前記ルートノードに間接的にリンクされた複数の下層ノードを備え前記ノードは前記言語の構文的要素と、前記ノード間の実装およびインターフェースではなく代替可能性の継承の経路を表す前記ノード間のリンクのパターンを表し、前記コンピュータプログラミングツールは、

前記第１のデータ構造またはその記述を記憶する操作可能なメモリと、

前記第１のデータ構造から、前記第１のデータ構造に対応する第２のデータ構造またはその記述を生成するように構成された少なくとも１つの前記プロセッサと、を備え、

前記第２のデータ構造は、前記第２のデータ構造の内前記ルートノードに直接リンクされたルートノード以外の前記第２のデータ構造の全てのノードを有する前記第１のデータ構造のノードに対応するノードを備え、

前記第２のデータ構造の前記ノード間の前記リンクは前記ノード間の代替可能性の継承の経路、実装およびインターフェースを表し、

前記プロセッサはさらに、前記１つの第１のデータ構造のリンクパターンに基づいて、前記第１のデータ構造により表される前記代替可能性の継承への準拠を確立するための前記第２のデータ構造を使用するその後の処理操作中に、前記実装を利用するその後の処理操作中に前記第２のデータ構造のノードに関連して強制される前記第１のデータ構造の代替可能性の規則を定義する実装規則を生成するように操作可能であり、

前記第２のデータ構造と前記実装規則は、前記第１のデータ構造の前記１つの実装を形成し、前記コード部分は前記コンピュータプログラミング言語で表現され、前記コード部分ハンドリングツールは、

前記実装を記憶するためのメモリと、

前記実装に応じて候補コード部分で動作するように構成された少なくとも１つのプロセッサとを備え、前記候補コード部分は前記第２のデータ構造のノードに対応するインスタンスノードを備える、方法。

【請求項４０】

コンピュータまたはネットワークコンピュータで実装される、請求項１に記載のコンピュータプログラミングツール。

【請求項４１】

メタプログラミングを実行するように動作可能である、請求項１に記載のコンピュータ

プログラミングツール。
【請求項４２】
コンピュータまたはネットワークコンピュータで実装される、請求項１２に記載のコード部分ハンドリングツール。
【請求項４３】
メタプログラミングを実行するように動作可能である、請求項１２に記載のコード部分ハンドリングツール。
【請求項４４】
前記不均一なツリー構造は抽象構文ツリーである、請求項２に記載のコンピュータプログラミングツール。
【発明の詳細な説明】
【技術分野】
【０００１】
本発明は、システムを表現およびハンドリングするための技術に関し、より詳細には、不均一ツリーおよび／またはグラフ構造として表現可能なシステムを表現およびハンドリングするための技術に関する。このようなシステムは、例えば、例えばコンピュータプログラミング言語などの言語、またはその一部であってもよい。
【０００２】
また、本発明は、このようなシステムを表わすデータ構造の実装を生成するための技術、更にこのような実装を使用してシステムのインスタンスをハンドリングするための技術にも関する。システムがコンピュータプログラミング言語である例示的な文脈では、このようなインスタンスは、その言語で表現されたコード部分であってもよい。
【０００３】
このようなコード部分は、相互に関連するシンボルの組を含む構造化表現であると考えることができる。本発明を実施する技術は、コード部分に関して解析および／または操作を実行するために使用することができる。この点に関して、本発明は、パースおよびコンパイルの技術と、メタプログラミング技術とに関する。
【背景技術】
【０００４】
言語は、本発明の実施形態を適用可能な種類のシステムの一例である。言語はコードの一形態であり、コンピュータプログラミング言語は、今日の技術システムで多用されているコードの一例である。以下に、主としてコンピュータプログラムに関して本発明を説明するが、本発明を、他の種類のコード（ならびに言語以外のシステム）にも等しく適用できることはいうまでもない。
【０００５】
一般に、言語とは、任意のシンボルのシステムであり、これらのシンボルをどのように操作するかを定義する規則であると考えることができる。すなわち、言語は単なるシンボルの組にとどまらない。言語には、シンボルを操作するために使用される文法、すなわち規則の体系も含まれる。シンボルの組は、表現または通信に使用することができるが、シンボル間に明確かつ規則的な関係がないため、実際には、シンボルの組それ自体は、比較的意味を有さない。また、言語は文法も有するため、シンボルを操作して、シンボル間の明確かつ規則的な関係を表現することができる。プログラミング言語は、マシン（特にコンピュータ）の挙動を制御するために使用することができる人工言語である。プログラミング言語は、人間の言語のように、構造を決定するために構文的規則を使用して、意味を決定するために意味論的な規則を使用して定義される。
【０００６】
コードの評価は、時間と手間がかかる作業である。このような評価では、例えば、コードを理解し、コードを使用して作業を実行し、コードを操作して、何らかの方法でコードを変更する。
【０００７】

コードの評価が行なわれる作業の一例として、メタプログラミングがある。最も幅広い意味において、メタプログラミングとは、（例えば）既存のプログラムコードの解析、操作、再構成、改善、または単純化であると考えることができる。また、メタプログラミングとは、他のプログラム（または、それ自身）を、そのデータとして記述または操作するか、あるいは実行時に行なわれる作業の一部をコンパイル時に行なうコンピュータプログラムの記述であるとも考えることができる。多くの場合、メタプログラミングは、プログラマが、全てのコードを手動で記述するのと同じ時間内に、多くの作業を行なえるようにする。メタプログラミングでは、一般に、プログラミング命令を含む文字列表現の動的実行が行なわれる。つまり、プログラムが、プログラムを記述することができる。

【０００８】

メタプログラミング手順では、一般に、プログラムコードを、有用かつ有益な方法で、閲覧または提示できるようにすることが行なわれる。例えばメタプログラミングのためのコンピュータプログラミングコードの評価では、一般に、パースの作業が行なわれる。パース（別名「構文解析」）とは、（一般に、コードの一部から抽出される）トークンのシーケンスを解析して、対象の言語の所定の形式文法に関して、その文法的構造を決定するプロセスである。パースは、入力テキストを、後の処理に適しており、入力コードの暗黙的な階層構造を捉えているデータ構造（通常はツリー）に変換する。字句解析は、入力文字のシーケンスから（すなわち入力コードから）トークンを生成し、このトークンがパーサによって処理されて、例えば、対象のインスタンス（すなわちコード部分）に対するパースツリーまたは抽象構文ツリーなどのデータ構造が生成される。

【０００９】

パーサの最も一般的な使用は、コンパイラのコンポーネントとしての使用である。これは、コンピュータプログラミング言語のソースコードをパースして、何らかの形式の内部表現を作成する。例として、図１は、特定のコンピュータプログラミング言語で記述された入力コードをパースする単純化した例を示す概略図である。この例では、コンピュータプログラミング言語は、文法の２つのレベル、すなわち、語彙的レベルと構文的レベルを有する。

【００１０】

図１からわかるように、パースの第１段階では、トークン生成または字句解析が行なわれ、これにより、入力コードの入力文字ストリームが、正規表現の文法によって定義される、意味のあるシンボルに分割される。図１の例では、入力コード「ａ＋ｂ＊ｃ」が検査されて、それぞれが代数式の文脈で意味のあるシンボルである、＋、ｂ、＊、およびｃの各トークンに分割される。パーサは、文字＊と＋は新しいトークンの開始を指定し、「ａ＋」のような意味がないトークンは生成しないことを知らせる規則を有している。次の段階は、構文的なパース、すなわち構文解析であり、トークンが許容できる表現を形成していることをチェックする。この段階の結果、例えば図１に示すようなデータ構造（抽象構文ツリー）が生成されうる。最終段階（図１に不図示）は意味論的なパースすなわち解析であり、検証したばかりの表現の意味を考え出し、適切なアクションをとることが行なわれうる。コンパイラの場合、入力ソースコードからオブジェクトコードの生成が行なわれうる。

【００１１】

抽象構文ツリーとは、ツリー構造に倣ったデータ構造であり、ノードの組がリンクされている。抽象構文ツリーは、コード部分の固有の（inherent）論理構造を表わすために一般に使用される。このような論理構造は、究極的には対象の言語の文法に基づくため、「固有である」と考えられる。例えば、図１の抽象構文ツリーは、図１の入力コードストリームの固有の論理構造を表わすと考えることができるが、この構造は、対象の言語の文法に定義される優先順位の規則のため、固有でしかない。このため、図１においては、対象の言語の文法により、乗算記号＊が加算記号＋よりも優先されることが指示されるとされる。このため、この文法に基づいて、「固有の」論理構造は、オペランドトークン「ｂ」および「ｃ」を、演算子トークン「＊」に関連付け、同様に、オペランドトークン「ａ」

と複合オペランドトークン「ｂ＊ｃ」とを、演算子トークン「＋」に関連付ける。

【００１２】

ノードは、値または条件を含んでも、あるいは別個のデータ構造または自身のツリーを表わしてもよい。ツリー内の各ノードは、ツリーの下に（慣例上、この種のツリーは下向きに成長する）０以上の子ノードを持つ。子を持っているノードは、その子の「親ノード」と呼ばれる。ツリー内の最上位のノードは「ルートノード」と呼ばれる。ルートノードは、最上位のノードであり、一般に親を有さず、通常は、ツリーに対する操作の始点となるノードである（しかし、当然、一部のアルゴリズムは、ツリーの他のノード（例えばリーフノード）から開始することもある）。ルートノードを始点としてノード間のリンクを辿ることにより、ツリーの他の全ノードに到達することができる。このようなリンクは、一般に「エッジ」と呼ばれる。ツリーはルートノードを１つだけ有するが、ツリー内の他のノードは、そのノードにおいて、サブツリーのルートノードであるとみなすことができる。ツリーの最下位レベルのノードは、「リーフノード（または終端ノード）」あるいは単に「終端」と呼ばれる。これらは、最下位のレベルにあるため、子を持たない。内部ノード（またはインナーノード、または分岐ノードまたは非終端ノード、あるいは単に「非終端」）は、子ノードを持ち、このためリーフノードでないツリーの任意のノード（ルートノード以外）である。サブツリーは、それ自体を完全なツリーとしてみなすことができるツリーデータ構造の一部である。

【００１３】

抽象構文ツリー（ＡＳＴ）は、有限ラベル付き有向ツリーとして定義され、（図１の場合）その内部ノードは演算子によってラベル付けされており、リーフノードは演算子のオペランドを表わす。このため、リーフは、ｎｕｌｌの演算子であり、変数または定数のみを表わす。ＡＳＴは、一般に、パーサにおいて、解析ツリーとデータ構造間の中間構造として使用され、このうちの後者は、多くの場合、コンピュータプログラムのコンパイラまたはインタプリタの内部表現として使用される一方、最適化され、ここからコード生成が実行される。ＡＳＴは、プログラムの意味論に影響しない構文規則のノードおよびエッジを省略することにより、パースツリーとは異なる。例えば、オペランドのグループ分けがツリー構造から明示的であるため、グループ化括弧は、通常、ＡＳＴからは省略される。このことも、図１を検討することにより認められよう。図１において、入力コードストリームは、例えば「ａ＋（ｂ＊ｃ）」でもよく、この場合、グループ化括弧により、優先順位の規則が明確となる。しかし、このコードストリームに対する抽象構文ツリーは、図１に図示したものと同じとなり、意図されたグループ化（または優先順位の規則）は、ツリー構造から明白である。

【００１４】

いうまでもなく、本発明の文脈のツリーは、論理構造またはデータ構造の例である。一般に、このようなツリーのルートノード以外の各ノードは、多くても１つの親ノードを有する。しかし、本発明の文脈において、一部のノードが複数の親を有しており、構造がツリーではなく有向非巡回グラフ（ＤＡＧ）のようになる「ツリー」が使用されることが明らかとなろう。したがって、主に、抽象構文ツリーの形式を取る論理構造およびデータ構造に関して本発明を説明するが、いうまでもなく、従来のツリー（例えばグラフ）以外の構造も意図される。ツリーは、グラフの特殊な形式であると考えることができる。グラフ理論では、ツリーは、連結非巡回グラフである。ＤＡＧは、ツリーを一般化したものであると考えることができ、特定のサブツリーがツリーの複数の異なる部分によって共有されてもよい。多くの同じサブツリーを有するツリーでは、これにより、構造を記憶するための空間要件を大きく低減することができる。

【００１５】

図１を参照して上で説明したデータ構造（例えば、抽象構文ツリーおよび／またはグラフを表わす）は、対象のシステムのインスタンスを表わすため、「インスタンス」データ構造であると考えられることが強調される。図１の入力コードは、このコードが表現される言語のインスタンスである。言い換えると、その言語で記述される表現である。

【００１６】
本発明のシステム（例えば言語）は、それ自体、データ構造によって表現されうる。このような「システム」データ構造は、抽象構文ツリーおよび／またはグラフの形式をとりうる。例えば、言語のためのこのような「システム」データ構造は、言語の要素の編制およびその文法規則を表わし、その言語での表現は、そのデータ構造のインスタンス（すなわち「インスタンス」データ構造）である。
【００１７】
一般に、システムのインスタンスをハンドリングするためのツールは、このようなハンドリングを行うために、システムの実装を使用する。例えば、このようなツールは、システムのシステムインスタンスをハンドリングするために、システムを表わす「システム」データ構造の実装を使用しうる。コンピュータプログラミング言語の文脈では、このツールは、パーサまたはコンパイラであり、その言語で記述されるコード部分をハンドリングするために、その言語を表わす抽象構文ツリーの実装を使用しうる。例えば、ツールは、「システム」データ構造の実装に基づいて、コード部分を表わす「インスタンス」データ構造（それ自体が抽象構文ツリーでもよい）を生成しうる。
【００１８】
コンピュータプログラム（コンピュータプログラミング言語のインスタンスであるコード部分）に着目すると、コンピュータプログラムを表わすために使用できる多数の異なる種類のＡＳＴが存在する。最も一般的な種類のＡＳＴは、不均一ツリー構造であり、関連するコンストラクト固有の挙動を有することができるコンパクトな表現を提供するため、ツリー内の各コンストラクトタイプが、特定のデータ構造によって表わされている。この種の「インスタンス」構造は、手動で記述したコードによって実装できるが、より一般的には、ソフトウェアツール（例えば、ＡＳＴコードジェネレータ）によって自動的に生成される。このツールは、（言語の場合、システムの実装を表わす）所望の論理構造の明解な記述を、その入力として取り（特定の候補コード部分もツールへの入力として与えられる）、候補コード部分を実装するために、「システム」論理構造に基づいて「インスタンス」データ構造を生成可能にするための手段を、その出力として生成する。この文脈では、「実装」との文言は、対象の候補コード部分を実装するために、「システム」論理構造に基づいて、「インスタンス」データ構造を生成可能にする動作を指すと考えることができる。生成されたデータ構造を使用するためのインタフェースも提供されうる。
【発明の概要】
【発明が解決しようとする課題】
【００１９】
「システム」データ構造を実装し、この実装を使用して、対象のシステムのインスタンスを表わす「インスタンス」データ構造を生成およびハンドリングするための既存のツールおよび技術は、多くの問題を有することがわかっている。より詳細には、コンピュータプログラミングの分野における既存の実装は柔軟性が低く、メタプログラミング等の技術に問題を引き起こすことがわかっている。これらの既存のツールおよび技術は、使用が複雑であり、プログラマの多くの時間と労力を必要とすることがわかっている。このような問題の原因となる、これらの既存のツールおよび技術の技術的特徴については、本明細書で後述するが、このような問題の一部または全てを解決することが望ましい。
【００２０】
システム（例えばコンピュータプログラミング言語）のインスタンス（例えばコード部分）を柔軟かつ効率的にハンドリング（操作、評価、解析、変更、変換など）可能にするシステムの実装を提供することが望ましい。
【課題を解決するための手段】
【００２１】
本発明の第１の態様の実施形態によれば、複数のリンクされた構造要素を有し、所定のシステムの意味のある要素および該意味のある要素間の相互関連性の親の集合を表わす第１のデータ構造の実装を生成するための実装ツールであって、前記第１のデータ構造また

はその記述を受け取るように動作可能な第１のデータ構造入力手段と、受け取った前記第１のデータ構造の実装を生成するように動作可能な処理手段と、を有し、前記実装は、前記第１のデータ構造に対応し、前記相互関連性の親の集合の部分集合によって定義される第２のデータ構造またはその記述と、前記実装を利用するその後の処理操作中に前記相互関連性の親の集合を強制させることを可能にする実装規則と、を有する実装ツールが提供される。

【００２２】

第２のデータ構造が、相互関係性の親の集合の部分集合によって定義されるため、第２のデータ構造は、第１のデータ構造よりも関連する制約または規則が少なくなる。このため、第２のデータ構造のインスタンスは、関係するデータ構造に従って、第１のデータ構造のインスタンスよりも柔軟に操作することができる。実装規則によって、例えば、第２のインスタンスのデータ構造をハンドリングする場合、相互関係性の親の集合を適用させることが可能である。第２のデータ構造に基づいて、実装規則の使用を、インスタンスの操作（ハンドリングの一種）から分離できる。このため、第１のデータ構造が非常に複雑であり、インスタンスのハンドリングに関する問題（後で更に詳細に説明する）を引き起こす場合には、複雑さがある程度軽減され、付随する実装規則において（別個に）表現された第２のデータ構造によって、第１のデータ構造を実装することが有利となる。実装は、コードで表現され、例えばコンピュータプログラミング言語で表現されうる。第１のデータ構造（と当然その記述）は、理想化された構造である（あるいはこれを記述する）、すなわち理想的または簡略化された形でシステムを表わすと考えることができる。リンクされた構造要素は、このようなデータ構造内のノードおよびリンクであると考えることができる。相互関係性の親の集合は、システムの固有の論理構造（例えば、システムが言語である場合は言語の文法）を定義すると考えることができる。

【００２３】

好ましくは、前記構造要素は、前記システムのそれぞれの意味のある要素を表わす。

【００２４】

前記システムは、抽象構文ツリーまたはグラフなどの不均一な抽象構造によって表わすことができるどのようなシステムでもよい。前記システムは言語または言語の一部でもよく、前記言語はコンピュータプログラミング言語でもよい。好ましくは、前記第１のデータ構造は、少なくとも部分的に抽象構文ツリーなどの不均一なツリー構造であり、前記構造要素の一部または全ては、前記ツリー構造のリンクされたノードである。前記第１のデータ構造は、少なくとも部分的に有向非巡回グラフ構造であり、前記構造要素の一部または全ては前記グラフ構造のリンクされたノードであってもよい。

【００２５】

言語を前記システムとみなすと、前記システムは、構文規則の組を満たす複数の構文的要素を有してもよい。前記ノードは前記構文的要素を表わし、前記リンクは前記構文規則を表わしてもよい。前記システムは複数のトークンから構成され、各前記構文的要素は、前記トークンの群あるいは前記トークンの所定の組合せを表わしてもよい。例えば、コンピュータプログラミング言語の場合、このような１つの構文的要素は特定の命令の集合を表わし、別の構文的要素はこれらの命令の部分集合を表わしてもよい。

【００２６】

前記第１のデータ構造の前記構造要素は、ルート構造要素と、前記ルート構造要素に直接リンクされた複数の第１層構造要素と、１つ以上の他の前記構造要素を介して前記ルート構造要素に間接的にリンクされた複数の下層構造要素と、を有してもよい。

【００２７】

前記第１のデータ構造の前記構造要素間の前記リンクは、該構造要素の間の実装およびインタフェースではなく代替可能性の継承の経路を表わしてもよい。また、前記リンクが、特定の用途に応じて他の関係を表わしてもよい。

【００２８】

オブジェクト指向プログラミングでは、「継承」とは、既に定義済みのクラスを使用し

て、抽象構文ツリーなどのデータ構造内で新しいクラス（そのインスタンスは「オブジェクト」と呼ばれる）を作成するための方法である。このようなクラスは、第１のデータ構造と第２のデータ構造内のノードまたは構造要素であると考えることができる。新しいクラスは、派生クラス（または継承クラス）とも呼ばれ、基底クラス（または祖先クラス）とも呼ばれる既存のクラスの属性および挙動を引き継ぐ（すなわち継承する）。このような継承は、既存のコードを一切あるいはほとんど変更することなく、再利用するのを助けることを目的としている。

【００２９】

継承の利点は、インタフェースが十分に似ているモジュール(クラスまたはノード）が多くのコードを共有し、プログラムの複雑さを緩和することにある。このため、継承は、「ポリモルフィズム」と呼ばれる別のビュー、二重を有し、共有された制御コードによって制御される多くのコードを記述する。

【００３０】

この背景に鑑み、第１のノード（継承ノード）が、第２のノード（祖先ノード、一般には第１のノードの親ノードである）から代替可能性を継承する場合、継承ノードは、祖先ノードの一種または部分集合であるとみなすことができ、第２のノードのインスタンスの代わりに、第１のノードのインスタンスを有効に用いることができる。インタフェースの継承は、祖先ノードによって継承ノードのインタフェースに露出されるインタフェースの交配（mating
up）であると考えることができる。実装の継承とは、継承ノードのメソッドによる、祖先ノードによって露出される１つ以上のメソッドのオーバーライド（置換）、あるいは、祖先ノードによって露出されるメソッドへの、継承ノード内に表現される新しいメソッドの追加であると考えることができる。

【００３１】

前記第１のデータ構造の前記構造要素間のリンクが、該構造要素の間の実装およびインタフェースではなく代替可能性の継承の経路を表わすことは有利である。代替可能性の継承のみを考慮することにより、他のノードに対するノードの可能な置換のみを考慮するだけで済む。より詳細には、実装およびインタフェースを無視することによって、実装およびインタフェースの多重継承に伴って発生しうる問題を回避または黙認することができる。

【００３２】

好ましくは、前記第２のデータ構造は、前記第１のデータ構造の前記構造要素にそれぞれ対応する構造要素を有する。例えば、前記第２のデータ構造は、前記第１のデータ構造の上述のノードに対応するノードを有してもよい。

【００３３】

好ましくは、前記第２のデータ構造の前記構造要素は、ルート構造要素を有し、前記第２のデータ構造の前記他の構造要素は、前記ルート構造要素に直接リンクされている。更に、好ましくは、前記第２のデータ構造の前記構造要素間の前記リンクは、該構造要素の間の代替可能性、実装およびインタフェースの継承の経路を表わす。

【００３４】

前記第２のデータ構造の前記構造要素間のリンクが、前記構造要素の間の代替可能性、実装およびインタフェースの継承の経路を有利に表わすことは有利である。後に行なわれる処理は、この第２のデータ構造（実装の一部を形成する）に基づき、このため、第２のデータ構造のインスタンスを正しく実装するために、完全な継承（すなわち代替可能性、実装およびインタフェースの継承）の経路を使用して、インスタンスノードが作成される。第２のデータ構造のルート構造要素以外の構造要素がルート構造要素から直接継承するように構成することによって、多重継承の問題が発生しない。特に、リンクを介して完全な継承が実装されたとしても、第１のデータ構造内の対応するリンクのネットワークが多重の継承問題を発生させる場合であっても、上記が当てはまる。

【００３５】

好ましくは、前記第１のデータ構造によって表わされる代替可能性関係への準拠を確立するために、前記実装規則は、前記実装を利用するその後の処理操作中に前記第２のデータ構造の構造要素に関連して強制される前記代替可能性の規則を定義する。第１のデータ構造によって表される代替可能性関係は、システムの忠実な表現のために中心的であるが、このような関係が、第２のデータ構造自体からは明白でなくてもよい。例えば、第２のデータ構造のルート構造要素以外の構造要素が、ルート構造要素から直接継承するが、異なるリンクのネットワーク（代替可能性を表わす）が第１のデータ構造に存在する場合に、上記が当てはまる。したがって、後続の処理（例えば、第２のデータ構造のインスタンスをハンドリングする場合）が、システムに忠実に（または正しく）なるように、第１のデータ構造によって表される代替可能性関係との遵守を確立（適用またはチェック）するために、このような実装規則を使用することが有利である。

【００３６】

任意選択で、前記第１のデータ構造（あるいは当然その記述も）は、システム拡張を表わす更に別のリンクされた構造要素を有し、前記システム拡張は、前記システムに対する拡張でもよい。前記第１のデータ構造の前記更に別の構造要素は、前記システム拡張のそれぞれの意味のある要素を表わしてもよい。なお、前記第１のデータ構造の特徴は、その記述内に表わされるかまたは定義されてもよい。

【００３７】

前記第１のデータ構造の前記更に別の構造要素の１つ以上は、前記第１のデータ構造の前記他の構造要素の一部もしくは全部に対して代替可能として定義されてもよい。いうまでもなく、第１のデータ構造内のリンクが実装およびインタフェースの継承を表わす場合には、このような更に別の構造要素に関して多重継承の問題が発生する可能性がある。しかし、前述したように、この問題は第２のデータ構造内では取り除くことができる。より詳細には、第１のデータ構造のこのような追加の構造要素に関連する代替可能性関係は、実装規則によって、第２のデータ構造の対応する構造要素に関して表されてもよい。

【００３８】

前記システム拡張は、前記システムが言語である場合、前記言語に対する拡張であると考えることができる。前記拡張は、別の言語の一部または全体でもよい。例えば、このようにして、１つのコンピュータプログラミング言語（例えばＣ＋＋）が、別のこのような言語（例えばＪａｖａ（登録商標））の所望の一部と、前記ツールを使用して実装される複合言語（システムとシステム拡張の組み合わせ）とで補足されうる。

【００３９】

本発明の第２の態様の実施形態によれば、複数のリンクされた構造要素を有し、所定のシステムの意味のある要素および該意味のある要素間の相互関連性の親の集合を表わす第１のデータ構造の実装を生成するための実装方法であって、前記第１のデータ構造またはその記述を受け取るステップと、前記受け取った第１のデータ構造の実装を生成するステップと、を有し、前記実装は、前記第１のデータ構造に対応し、前記相互関連性の親の集合の部分集合によって定義される第２のデータ構造またはその記述と、前記実装を利用するその後の処理操作中に前記相互関連性の親の集合を強制させることを可能にする実装規則と、を有する方法が提供される。

【００４０】

本発明の第３の態様の実施形態によれば、コンピューティングデバイスで実行されたときに、前記コンピューティングデバイスを上述の本発明の第１の態様に係るツールにさせる実装コンピュータプログラムが提供される。

【００４１】

本発明の第４の態様の実施形態によれば、コンピューティングデバイスで実行されたときに、前記コンピューティングデバイスに上述の本発明の第２の態様に係る方法を実行させる実装コンピュータプログラムが提供される。

【００４２】

本発明の第５の態様の実施形態によれば、上述の本発明の第１の態様に係る実装ツール

によって生成される前記実装の前記第２のデータ構造のインスタンスに動作するためのインスタンスハンドリングツールであって、前記実装を記憶する手段と、前記実装に依存して前記候補インスタンスに動作するための手段と、を有するインスタンスハンドリングツールが提供される。

【００４３】

前記候補インスタンスは、前記第２のデータ構造の前記構造要素に対応するインスタンス構造要素を有してもよい。すなわち、前記候補インスタンスの前記インスタンス構造要素は、前記実装によって実装（構築、作成、生成、管理、またはハンドリング）されてもよい。

【００４４】

前記インスタンスハンドリングツールは、前記第２のデータ構造の前記候補インスタンスを、そのインスタンス構造要素およびその間のリンクが明示的に表現されていない入力形式で受け取るための手段と、前記受け取った候補インスタンスを、前記インスタンス構造要素およびこれらの間のリンクが明示的に表現されている抽象化形式に変換するための変換手段と、を更に有してもよい。

【００４５】

前記システムがコンピュータプログラミング言語であり、前記候補インスタンスが、前記コンピュータプログラミング言語で表現されたコード部分である場合、前記入力形式は、前記コード部分をテキストで表わしたものであると考えることができ、前記抽象化形式は、前記コード部分（またはその記述）を抽象構文ツリーおよび／またはグラフで表わしたものと考えることができる。

【００４６】

前記インスタンスハンドリングツールは、前記候補インスタンスの視覚的表現を生成するように動作可能な可視化手段を更に有してもよい。このような視覚的表現は、ツールのユーザが候補インスタンスを理解するため、特に当該ユーザがインスタンスをハンドリングするために、ツールと対話できるようにするための効率的な方法となりうる。このようなハンドリングには、前記候補インスタンスの操作が含まれてもよく、このような操作を視覚的に監視することが望ましいことがある。前記インスタンスハンドリングツールは、前記視覚的表現を前記ユーザに表示するための表示手段を有してもよい。好ましくは、前記可視化手段は、前記候補インスタンスの視覚的表現を前記抽象化形式で生成するように動作可能である。

【００４７】

前記インスタンスハンドリングツールは、前記実装に依存して前記候補インスタンスを操作するように動作可能な操作手段を更に有しうる。好ましくは、前記操作手段は、前記候補インスタンスを前記抽象化形式で操作するように動作可能である。

【００４８】

好ましくは、前記操作手段は、前記実装と照らし合わせてこのような操作を検証し、前記第２のデータ構造および／または前記実装規則に違反する操作を禁止するように動作可能である。好ましくは、前記操作手段は、前記第２のデータ構造および前記実装規則を遵守する操作を許可するように動作可能である。このようにして、前記インスタンスハンドリングツールは、前記候補インスタンスの操作を、前記システムに遵守させる（またはシステムに対して正しい状態にする）ことを可能にしうる。例えば、対象のコンピュータプログラミング言語（言語拡張の有無を問わず）に遵守させつつ、候補コード部分が（上で説明した柔軟な方法で）操作されうる。

【００４９】

このような操作は、前記候補インスタンスを増やすおよび／または前記候補インスタンスを減らすことを含んでもよい。例えば、前記操作は、前記候補インスタンスに新しいインスタンス構造要素を追加すること、および／または前記候補インスタンスからインスタンス構造要素を削除すること、および／または前記候補インスタンスのインスタンス構造要素を、他のこのようなインスタンス構造要素によって置き換えることを含んでもよい。

このような操作は、前記候補インスタンスの特定のインスタンス構造要素を何らかの方法でアノテートすることまたは強調することを含んでもよい。コンピュータプログラミング言語で表現される候補コード部分の場合、このことはプログラマに有用となりうる。例えば、プログラマにとって関心のあるコード部分の特定の箇所（例えばエラーを含んでいる部分や、最適化できる部分）を強調することが可能でもよい。

【００５０】

このような操作は、前記候補インスタンスの一部または全体に所定のプロセスを実行することを含んでもよい。例えば、インスタンスハンドリングツールによってアクセス可能なコンピュータプログラムなどの所定のプロセスが、アクションの組内に定義されてもよい。前記インスタンスハンドリングツールは、このようなプロセスを任意裁量で複数有してもよく、前記候補インスタンス自体に依存して、またはツールのユーザからの入力に依存して、これらのプロセスを選択的に使用してもよい。

【００５１】

前記所定のプロセスは、所定の目的のために前記候補インスタンスを最適化するように養成された最適化プロセスであってもよい。例えば、候補インスタンスが、コンピュータプログラミング言語で表現されたコード部分である場合、最適化プロセスは、特定のタイプのプロセッサまたはプロセッサの組み合わせで実行されるコード部分を最適化するように構成されうる。別の例として、最適化プロセスは、コード部分に対して得られるオブジェクトコードがかなり縮小し、これにより処理の時間とパワーが節約されるように、コード部分によって特定のプロセス（例えばマトリックス計算）が実行されるやり方を簡略化するように構成されうる。

【００５２】

前記操作手段は、前記候補インスタンスの前記インスタンス構造要素に依存してこのような操作を実行するように動作可能であってもよい。すなわち、このような操作が、候補インスタンス内の、特定の型のインスタンス構造要素が存在するいくつかの位置で実行されうる。前記操作手段は、インスタンス構造要素の特定の型を識別し、前記識別されたインスタンス構造要素に依存してこのような操作を実行するように動作可能であってもよい。

【００５３】

例えば、前記操作手段は、システム拡張の前記更に別の構造要素に対応する前記候補インスタンスのインスタンス構造要素を識別するように構成されうる。（上記したように）この更に別の構造要素が、他の任意の構造要素に対して代替可能に定義される場合、かなりの利点を実現することができる。例えば、コンピュータプログラミング言語で表現されたコード部分の場合、このような更に別の構造要素の普遍的な代替可能性のため、プログラマが、代替可能性の規則に違反することなく、コード部分のどこにでも、（その更に別の構造要素のインスタンスである）エントリを配置できるようになる。この場合、プログラマは、例えば、最適化プロセス（操作）を実行すべき位置を、コード部分内で、自由かつ柔軟に指示することができる。

【００５４】

当然、いうまでもなく、前記操作手段は、候補インスタンスの特定箇所、あるいは候補インスタンス内のパターンを特定し、その位置において前記操作を実行するように動作可能でもよい。

【００５５】

上記に基づくと、いうまでもなく、前記候補インスタンスは、前記システム拡張に由来する部分を含んでもよい。

【００５６】

前記インスタンスハンドリングツールの前記変換手段は第１の変換手段であると考えることができ、前記インスタンスハンドリングツールは、前記抽象化形式の前記候補インスタンスをその対応する入力形式に変換するように動作可能な第２の変換手段を更に有してもよい。前記第２の変換手段は、このような操作が前記候補インスタンスに実行される前

にあるいはその後に、このような変換を実行するように動作可能であってもよい。前記インスタンスハンドリングツールは、その所期の目的に応じて、前記第１の変換手段でなく前記第２の変換手段を有しても、あるいはこの逆であってもよい。前記インスタンスハンドリングツールは、このような操作の前か後に、前記候補インスタンスをオブジェクトコードとして出力するように動作可能な手段を有してもよい。

【００５７】

前記インスタンスハンドリングツールは、パーサまたはコンパイラであっても、このようなパーサまたはコンパイラを含む任意のツールであってもよい。

【００５８】

本発明の第６の態様の実施形態によれば、上述の本発明の第１の態様に係る実装ツールによって生成される前記実装の前記第２のデータ構造のインスタンスに動作するためのインスタンスハンドリング方法であって、前記実装に依存して前記候補インスタンスに動作するステップを有するインスタンスハンドリング方法が提供される。

【００５９】

本発明の第７の態様の実施形態によれば、コンピューティングデバイスで実行されたときに、前記コンピューティングデバイスを上述の本発明の第５の態様に係るインスタンスハンドリングツールにさせるインスタンスハンドリングコンピュータプログラムが提供される。

【００６０】

本発明の第８の態様の実施形態によれば、コンピューティングデバイスで実行されたときに、前記コンピューティングデバイスに上述の本発明の第６の態様に係る方法を実行させるインスタンスハンドリングコンピュータプログラムが提供される。

【００６１】

本発明の第９の態様の実施形態によれば、システムを拡張する方法であって、前記システムを表わす第１のデータ構造またはその記述を取得するステップと、システム拡張を表わす更に別のリンクされた構造要素を含むように、前記第１のデータ構造またはその前記記述を適応させるステップと、前記適応された第１のデータ構造の実装を生成するために、本発明の上述の第１の態様に係る実装ツールを使用するステップと、を有する方法が提供される。このように、効率的に、かつアドホックで、このようなシステムを拡張することが可能である。例えば、このようにして、あるコンピュータプログラミング言語を、別のコンピュータプログラミング言語の特徴を有するように拡張することができる。

【００６２】

本発明の第１０の態様の実施形態によれば、本発明の上述の第１の態様に係る実装ツールおよび／または本発明の上述の第５の態様に係るインスタンスハンドリングツールを有するパーサまたはコンパイラが提供される。

【００６３】

本発明の第１１の態様の実施形態によれば、本発明の上述の第１の態様に係る実装ツールおよび／または本発明の上述の第５の態様に係るインスタンスハンドリングツールによって生成または適合されるコンピュータプログラムが提供される。

【００６４】

本発明の第１２の態様の実施形態によれば、コンピュータプログラムを生成あるいは適応させる方法であって、候補コンピュータプログラムを、前記上述した第５の態様に係るインスタンスハンドリングツールに入力させ、対象前記のシステムは、前記候補コンピュータプログラムを表現しているコンピュータプログラミング言語であるステップと、前記候補インスタンスに動作するために前記インスタンスハンドリングツールを使用するステップと、このような操作から得られるコンピュータプログラムを出力するために前記インスタンスハンドリングツールを使用するステップと、を有する方法が提供される。前記生成または適合されるコンピュータプログラムは、このような方法の前記直接の生成物である。

【００６５】

本発明の第１３の態様の実施形態によれば、本発明の上述の第１の態様に係る実装ツールおよび／または本発明の上述の第５の態様に係るインスタンスハンドリングツールとして機能するように構成されたコンピュータまたはコンピュータのネットワークが提供される。

【００６６】

本発明の第１４の態様の実施形態によれば、本発明の上述の第１の態様に係る実装ツールによって生成されるような実装が提供される。このような実装は、コンピュータプログラムの形であってもよい。

【００６７】

本発明の第１５の態様の実施形態によれば、上述の本発明の態様のいずれかによるコンピュータプログラムを記憶しているコンピュータ可読記憶媒体が提供される。

【００６８】

装置（ツール）の態様の特徴は方法の態様およびコンピュータプログラムの態様にも、本発明の他の態様にも等しく適用することができ、この逆も当てはまる。

【００６９】

本発明は、上述の本発明の態様に係るそれぞれのツール、方法およびコンピュータプログラムを含むメタプログラミングツール、方法およびコンピュータプログラムに拡張される。

【図面の簡単な説明】

【００７０】

【図１】特定のコンピュータプログラミング言語で記述された入力コードをパースする単純化した例を示す概略図である。

【図２】言語の文法の例の抜粋である。

【図３】図２に基づくＡＳＴノード構造の例である。

【図４】図２に基づくＡＳＴノード構造の別の例である。

【図５】図２に基づくＡＳＴノード構造の別の例である。

【図６】構文拡張ブロックが挿入されているＡＳＴノード構造の例である。

【図７】図３から所望の設計を記述するための１つの方法（第１のデータ構造記述）を示す。

【図８】複数型参照の使用を示す。

【図９】本第１のデータ構造記述スキームにおけるワイルドカード擬似継承の表現を示す。

【図１０】図３の所期の論理構造に適用された実装（第２のデータ構造）の例である。

【図１１】生成されるアクセサ関数（Ｃ＋＋の場合）の例である。

【図１２】生成されるアクセサ関数（Ｃ＋＋の場合）の別の例である。

【図１３】これらの反復子モデル関数を使用するＣ＋＋ループの例である。

【図１４】ｖｉｓｉｔ（）関数の使用の例を示す。

【図１５】ｅｎｕｍｅｒａｔｅ（）関数の使用の例を示す。

【図１６】コード変換ツールの一部としての使用における、本システムを実施するメタコンパイラの単純化した使用例の概略図である。

【図１７】コード変換ツールの一部としての使用における、本システムを実施するメタコンパイラの簡略使用の例の別の概略図である。

【図１８】コード変換ツールの一部としての使用における、本システムを実施するメタコンパイラの簡略使用の例の別の概略図である。

【図１９】コード理解ツールの一部としての使用における、本システムを実施するメタコンパイラの単純化した使用例の概略図である。

【図２０】コードリファクタリングツールの一部としての使用における、本システムを実施するメタコンパイラの単純化した使用例の概略図である。

【図２１】ビルド最適化ツールの一部としての使用における、本システムを実施するメタコンパイラの単純化した使用例の概略図である。

【図２２】コードインスツルメンテーションツールの一部としての使用における、本システムを実施するメタコンパイラの単純化した使用例の概略図である。
【図２３】コード変換ツールの一部としての使用における、本システムを実施するメタコンパイラの簡略使用の例の別の概略図である。
【発明を実施するための形態】
【００７１】
例示のために、添付の図面を参照する。
【００７２】
本発明の実施形態は、（その拡張の有無を問わず）コンピュータプログラミング言語（の一部または全て）を表わす論理構造の記述（第１のデータ構造の記述）を、その言語で記述されたコード部分をハンドリングするために有用な、（対応する第２のデータ構造と、実装規則の組とを含む）実装に変換するために設計されたソフトウェアツールに関する。改変されたオブジェクト指向プログラミングモデルが提供され、このモデルは、「システム」ＡＳＴの既存のオブジェクト指向モデルよりも有用な論理構造を実装する定義を生成し、既存のＡＳＴコードジェネレータよりも高い表現上の柔軟性を与える。本発明の好ましい実施形態は、このようなシステムのより明解かつ柔軟な表現を提供し、メタプログラミングシステムを作成するうえでのより優れた基盤を提供するものである。本発明の別の態様は、このモデルに構築された、完全なＡＳＴコード生成ソリューションを構築するための他の方法に関する。
【００７３】
既存のツールの考察
本発明の実施形態の理解をより深めるために、まず、既存のツールについて説明する（その１つとして、ＴｒｅｅＣＣ（http://www.southern-storm.com.au/treecc.html）が知られている）。明らかとなるように、論理構造記述とデータ構造設計とが本発明の重要な態様を形成している。比較のため、これらは、このような既存のツールにおいて使用されている「システム」ＡＳＴ生成の方法にある程度対応している。このため、既存のツールのＡＳＴ生成の方法の不具合を理解するために、これらについて（ＴｒｅｅＣＣにおいて使用されているものに着目して）考察する。
【００７４】
これらの既存のツールは、論理構造（「システム」ＡＳＴ構造）を実装するために有用なデータ構造を生成することができるが、得られる実装は、一般に、標的プログラミング言語の標的モデルに大きく依存するが、このようなモデルはＡＳＴ構造の実装時に使用するのに必ずしも最も適したモデルであるとは限らない。
【００７５】
この点に関して、図２の言語文法の（単純化した）抜粋の例を参照する。Ｓｔａｔｅｍｅｎｔ、ＤｅｃｌａｒａｔｉｏｎおよびＩｎｉｔｉａｌｉｚｅｒは、非終端（非終端ノード）であり、それぞれが取ることができる、さまざまな形に更に分解される。単純化したこの抜粋において、Ｓｔａｔｅｍｅｎｔは、ｉｆ－ｔｈｅｎ－ｅｌｓｅ文の２つの形のうちのいずれか一方であるか、またはＥｘｐｒｅｓｓｉｏｎである。同様に、Ｉｎｉｔｉａｌｉｚｅｒは、ＩｄｅｎｔｉｆｉｅｒまたはＥｘｐｒｅｓｓｉｏｎのいずれかである。
【００７６】
これは、言語の文法でかなり一般的に出現する。ここでは、Ｅｘｐｒｅｓｓｉｏｎは、Ｓｔａｔｅｍｅｎｔが予想される場所と、Ｉｎｉｔｉａｌｉｚｅｒが予想される場所に出現しうる。
【００７７】
ＴｒｅｅＣＣツールにより、ＡＳＴの設計者が、一種の継承によって、非終端のために予想される異なる変種を記述できるようにし、ＡＳＴが実装される言語（ここでは「実装言語」と呼ぶ）における継承を介してこれを実装する。
【００７８】
本発明の文脈における「継承」とは、既に定義済みの親ノード（祖先ノード）から子ノ

ード（継承ノード）を形成するやり方を指すものと考えることができる。子ノードは、派生ノード（または派生クラス）とも呼ばれ、基底ノード（または祖先ノード）とも呼ばれる親ノード（既存のクラス）の属性および挙動を引き継ぐ（すなわち継承する）。継承は、一般に、既存のコードを一切あるいはほとんど変更することなく、再利用するのを助けることを目的としている。継承の利点は、インタフェースが十分に似ているノードが多くのコードを共有し、必要なコードの複雑さを緩和することにある。継承によって、子ノードが、既に親ノードに存在するコードを再利用することが可能となり（これは一般に「実装継承」と呼ばれる）、この目的でインタフェースを一致させることは「インタフェース継承」として知られている。

【００７９】

ＴｒｅｅＣＣの例に戻り、図２の文法について考えると、継承の使用により、一般に、図３に示すＡＳＴノード構造が指定されるようになる。図３に示すやり型でＡＳＴ構造を指定する大きな利点の１つは、型チェックを使用できるようになる点にある。Ｉｎｉｔｉａｌｉｚｅｒとその派生型（ＩｄｅｎｔｉｆｉｅｒおよびＥｘｐｒｅｓｓｉｏｎ）の関係が明示的に指定されているため、Ｉｎｉｔｉａｌｉｚｅｒが予想される場所で、ＩｄｅｎｔｉｆｉｅｒおよびＥｘｐｒｅｓｓｉｏｎ以外が使用されないことを、候補コード部分の任意の実装が保証することができる。実装言語が静的に型付けされる場合には、型チェックの大半が、コンパイラによってコンパイル時に提供され、動的に型付けされる場合には、コードジェネレータが、使用するオブジェクトの型が、ＡＳＴ設計者の指定と一致することをチェックするコードを生成することができる。また、同じ機能は、ＡＳＴノードからの他のＡＳＴノードへの参照（例えば、ＶａｒｉａｂｌｅＤｅｃｌａｒａｔｉｏｎノードのＴｙｐｅフィールドによって参照されるＴｙｐｅＮａｍｅノードなど）にも該当する。

【００８０】

しかし、図３に示すＥｘｐｒｅｓｓｉｏｎノードは、ＴｒｅｅＣＣでは表現することができない。ＳｔａｔｅｍｅｎｔとＩｎｉｔｉａｌｉｚｅｒの両方からＥｘｐｒｅｓｓｉｏｎを派生させることは「多重継承」と呼ばれ、ＴｒｅｅＣＣはこれを許していない。これは、ＴｒｅｅＣＣの側の自由裁量の判断ではなく、実装言語のなかには多重継承をサポートしていないものもあり、サポートしている言語では、多重継承は、多くの場合、潜在的な名前の衝突とダイヤモンド形の継承ツリーのため問題となるコンストラクトである。

【００８１】

このため、ＴｒｅｅＣＣで、この文法の例を指定するために、図４に示すように、Ｅｘｐｒｅｓｓｉｏｎが２つの別個のノード型（例えばＳｔａｔｅｍｅｎｔＥｘｐｒｅｓｓｉｏｎとＩｎｉｔｉａｌｉｚｅｒＥｘｐｒｅｓｓｉｏｎ）に分割され、これにより多重継承が不要となる。

【００８２】

これは有効な解決策であるが、ＡＳＴ設計者の側に多くの負担を強い、表現を処理するためにコードを追加するときに、問題が発生する場合もある。２つの無関係な表現型が存在するため、実装言語によっては、それぞれ別個の処理コードの記述が必要となることがある。特に、ＡＳＴコードジェネレータの第一の目的が、労力を軽減し、ヒューマンエラーの可能性を最小化することにある場合には、これでは理想からはほど遠い。

【００８３】

他の公知のＡＳＴコードジェネレータは、ＴｒｅｅＣＣとは僅かに異なる手法を使用する。継承によって関連付けられた複数のノードを介した非終端の内部に異なる変種を指定する代わりに、これらが、親ノード記述自体の内部に直接表現される。これらは、一般に、真のノードへの参照を有する単純なコンストラクトとして表現されるが、単純なノードを直接表現してもよい。このため、これらの他の公知のツールによって、文法の例を図５に示すように表現する必要がある。

【００８４】

図５の構造は、図３と図４の構造よりもさらに複雑であり、生成されるコード中での実

装のされ方によっては、同じプログラムを表現するために必要な実行時オブジェクトの数が増えてしまうことがある。また、ＡＳＴ記述言語での図５の構造の指定は、特にＡＳＴ設計者が可能な限り無駄を最小化したい場合には、図３と図４の構造よりも更に複雑である。この構造は、仕様から継承の概念を事実上なくすことによって、多重継承の問題を解決するが、ＴｒｅｅＣＣを使用する場合に必要な解決策のように、その結果は理想的でない。

【００８５】

公知の既存のＡＳＴコードジェネレータは、全てこのような問題を有している。より理想的な解決策は、図２に示す構造を許する一方で、多重継承によって発生する問題をなくすものである。柔軟性と操作が容易であることが重要な要素となるメタプログラミング環境でＡＳＴが使用される場合、既存の解決策によって更なる問題が発生する。

【００８６】

例えば、メタプログラミング時に、多くの場合、元の「システム」ＡＳＴ設計者ではなく、プログラマが指定するコンストラクトに関する特定のプログラム（コード部分）に対して、「インスタンス」ＡＳＴを装飾することが望ましい。これにより、プログラマは、多くの独創的な方法でプログラムの表現に追加の意味を付加でき、このことは、実現可能なメタプログラミング環境を提供するうえで重要な要素である。

【００８７】

このような柔軟性を与える１つのやり方には、属性を使用するものがあり、属性とは、ＡＳＴに追加の意味を付加するために、プログラマが既存のノードに追加することができるカスタムメイドの値である。属性は、有用であるが、既存の「システム」ノードにしか付加することができず、特定の作業に使用するためには、この作業のために「システム」ツリーに完全に一体化した特定のノード型を有する場合と比べ扱いにくい。

【００８８】

この例として、所望のメタプログラミングコンストラクトが挙げられ、これを、本発明の実施形態の以下の説明においては、「構文拡張ブロック（ＳＥＢ）」と呼ぶ。ＳＥＢは、本質的に、一般に、スクリプト言語を使用して表現され、サブツリーの上の「インスタンス」ＡＳＴに直接付加して、これに動作するためのコード片である。ＳＥＢは、ＡＳＴに属性として付加することもできるが、このように別個のノードとして表現するほうがより簡潔かつ効率的である。

【００８９】

しかし、ＡＳＴの場合によってはどの場所にもＳＥＢを付加することが望ましため、型安全性に関する問題を発生させかねない。この状況が図６に図示されており、ＳＥＢノードが、Ｅｘｐｒｅｓｓｉｏｎ＃１とＥｘｐｒｅｓｓｉｏｎ＃２との間の「インスタンス」ＡＳＴに挿入されており、Ｅｘｐｒｅｓｓｉｏｎ＃１のＬＨＳ参照は、型Ｅｘｐｒｅｓｓｉｏｎのノードを参照することを予想している。ＳＥＢノードが、Ｅｘｐｒｅｓｓｉｏｎに関連しない（すなわち、Ｅｘｐｒｅｓｓｉｏｎから派生されていない）場合には、静的に型付けされた言語が提供する型確認システムはこれを許さない。

【００９０】

この種類のコンストラクトに関する問題は、これを型安全に許す唯一の方法は、ＳＥＢが「システム」ＡＳＴ設計のあらゆる他のノード型から継承するのを許すことである。しかし、既存のシステムに深刻な問題を発生させかねない多重継承の濫用である。

【００９１】

メタプログラミングの既存の解決策の使用に関する（および一般的な）別の問題として、これらが、いわゆる「ビジターパターン」に基づいてツリーウォーキングインタフェースを提供するという点が挙げられる。ビジターパターンとは、提供されたコード片が、複雑なデータ構造（「インスタンス」ＡＳＴまたは論理構造など）を、一般に各ノードを１回だけ訪問してウォークし、訪問するノードごとにユーザ提供の関数を呼び出す方法である。このメカニズムによって、ユーザコードが、その基礎をなす構造を知らなくても、あるいは、有効なウォークを実行するために従う必要のある困難な規則を扱わなくても複雑

なデータ構造に動作することができる。

【００９２】

このメカニズムは有用な方法であるが、いくつかの不具合がある。ユーザの観点からの最も顕著な不具合は、ビジターハンドリングコードに対して、訪問するそれぞれのノードに対して呼び出す関数（「コールバック」と呼ばれることが多い）を渡される必要がある点にある。一部の言語では、この呼び出しは直接的である（例えば「クロージャー」を有する言語や、ＬＩＳＰやＲｕｂｙなどの似ている言語）が、他の言語（例えばＣ＋＋やＪａｖａ（登録商標））では、かなり面倒で複雑なプロセス（一般に、関数オブジェクトが関与する）となり、このような言語では一般に、ビジターの使用を周囲のコードに組み込むことが不可能である。

【００９３】

別の大きな不具合（周囲のコードにビジターを組み込むのが困難であることに一部関連している）は、ウォークの制御である。ビジターパターンでは、ウォークの制御は、ビジターコードに完全に託されている。つまり、ビジターコードは通常、終了するまで、特定のデータ構造内の全ノードを訪問する。ユーザが何らかの方法でウォークプロセスを変更するために呼び出すことができる追加の終了／制御関数が、ビジターメカニズムに設けられてもよいものの、変更は、一般にコールバック関数の内部で行なう必要がある。しかし、コールバック関数は、訪問コールの呼び出しの周囲のコードから離れているため、このような判断を行なうのに十分なコンテキスト情報を有さない。この場合も、この問題は、クロージャーサポート（またはこれに相当するもの）を有する言語では、解決できる些細な問題であるが、このようなサポートのない言語では、通常、関数オブジェクトに余分の情報を渡す必要があるが、この作業は、プログラマにとって無駄であり、多くの追加の作業を発生させかねない。

【００９４】

既存の解決策の使用に関する別の問題として、プログラムまたはコード部分の「インスタンス」ＡＳＴ表現を操作する（プログラムのパース時のこのような表現の構築を含む）際に、サブツリーを再使用して、構造内の複数の場所からサブツリーを参照できることが、時として便利である点にある。更に、全てのこのような参照が、対象のサブツリーを「所有」しているようにこれを行なうことが、時として便利である。

【００９５】

この一例が、あいまいさを有する複雑な言語をパースするための一般化ＬＲ（ＧＬＲ）パースメカニズムを使用する場合に発生し、入力内のあいまいな要素により、パーサが、事実上、その要素の実際の意味を異なって理解する２つのパーサに分割されることがある。これが発生すると、２つのパーサは、その時点までに既に構築済みの「インスタンス」ＡＳＴサブツリーを渡される必要がある。各ノードが親を１つだけ有する標準的なツリー構造によって論理構造が表現される場合、サブツリーを深くコピーする必要がある（サブツリー内の全ノードのコピーを作成する）が、この作業は、サブツリーが大きい場合には非常に高コストとなる。より効率的な解決策は、浅いコピーを許して、これにより、サブツリーが、所有権のセマンティクスを共有している複数の親ノードによって参照されるようにする（すなわち、どの親もサブツリーを参照していないときにのみサブツリーが破壊される）ことである。

【００９６】

これも、一部の実装言語では、ＡＳＴコードジェネレータからの支援を受けずに行うことができ、実装言語がガーベジコレクションをサポートしている場合、複数ノードが特定のノードを参照し、この特定のノードは、どのノードからも参照されていない場合にのみ破壊される。しかし、ガーベジコレクションのない実装言語（例えばＣ＋＋）では、この機能を提供するために追加の手順を実行する必要がある。

【００９７】

関連する問題は、明示的な親の参照の概念である。単純な「インスタンス」ツリー構造では、親ノードは子ノードを参照するが、子ノードは自身の親ノードを参照しない。この

ことにより構造を多少単純化するものの、ツリーを操作するために要するプログラマの労力が増えることがある。

【００９８】

例えば、ビジターパターンによる「インスタンス」ＡＳＴの訪問中に、コールバック関数が、現在のノードを削除する（当該ノードのみ、あるいは子ノードも含めて）ことを決定した場合、親ノードから子ノードへの参照を削除するために、親ノードを参照する必要がある。この親の参照は、子ノードからは取得できないため、このような操作は不可能であるか、あるいは、ビジターメカニズムが、ビジターが論理構造をウォークする際にトラッキングする親の参照を、コールバック関数が取得可能にする機能を提供することに依存する。親のないツリーを使用する場合は、他の可能な解決策も存在するが、このような解決策は一般に、プログラマに余分な労力を強い、他のさまざまな妥協を伴う。

【００９９】

公知のＡＳＴコードジェネレータは、一般に、親リンクのないツリー構造を使用し、ビジターメカニズムを介して親リンクを取得するための機能を提供しそうにないため、これらの解決策によるツリー操作（このため、メタプログラミング）は、時として扱いにくいことがある。

【０１００】

概要

本発明者は、前述のシステムは、対象のプログラミング言語（システム）の論理構造（ＡＳＴ）の記述と、データ構造の形での当該システムの実装との間に比較的直接的な変換を使用することを認識している。この結果、このようなシステムは、柔軟性と簡潔性を欠いている傾向がある。このため、これらの解決策のユーザは、柔軟性を改善するために、あるレベルの型安全性を犠牲にすることを強いられている。

【０１０１】

本発明の実施形態は、言語を記述し、人間が言語等の構造について考えるやり方に自然にフィットする（第１のデータ構造の）簡潔かつ柔軟な記述フォーマットから、コンピュータプログラミング言語の表現を記述、表現および生成するために、（第２のデータ構造またはその記述と、実装規則とを有する）実装を生成するための方法を提供する。ここではコンピュータプログラミング言語に着目するが、本発明は、他の種類の言語、ならびに不均一ツリー／グラフ構造によって表現できる他の種類のシステムにも適用することができることはいうまでもない。

【０１０２】

特に、本発明の実施形態により、データ構造（「システム」ＡＳＴ）のノードを、（インタフェースまたは実装ではなく、代替可能性の）「シミュレートされた」多重継承によって互いに関連付け、複数型参照によって相互に参照できるようにし、簡潔、明解、柔軟に生成されたプログラミングインターフェース内で動作する型安全な構造と操作機能を提供する。

【０１０３】

本発明の実施形態は、柔軟性、明解性および型安全性を同時に最大化することを目的とする。いうまでもなく、型安全性を保つやり型で、ＡＳＴ内のあらゆる場所に配置することができる、非常に柔軟なノードの表現方法が提供され、この重要な追加により、メタプログラミングなどのアプリケーションがより実際的となる。

【０１０４】

また、特定の場合にプログラミングインターフェースを複雑にし、クロージャーのためのサポートのない実装言語で使用するには扱いにくいビジターパターンベースのメカニズムによってではなく、ユーザ制御の反復プロセスによって使用可能な、ツリーウォーキングメカニズムも提供される。

【０１０５】

また、ここに開示の表現方法により、ＡＳＴを、各ノードが複数の他のノードによって「所有」され、各ノードが、そのノードを所有する全てのノード（その親）をトラック可

能な有向非巡回グラフ（ＤＡＧ）に似た形で表現可能となる。これにより、ＡＳＴ構造を効率化できる機会と、前述の純粋なツリーベースの解決策では使用不可能な操作とが提供される。

【０１０６】

詳細な説明

本実施形態は、コンピュータプログラミング言語の場合、好ましくはシステムの実装を生成するソフトウェアツールの形で提供される。このような実施形態は、関連する言語を表現し、ノード間のリンクが、実装およびインタフェースではなく代替可能性の継承を表わす第１のデータ構造（「システム」ＡＳＴ）の記述を入力として取る。このような実施形態は、この入力から、第２のデータ構造（「システム」ＡＳＴ）と実装規則とを有するシステムの実装を生成する。第２のデータ構造では、ルートノード以外のノードは、ルートノードから直接継承する。実装規則は、第２のデータ構造のインスタンス（コード部分）と共に使用するために、第１のデータ構造において固有の代替可能性関係を特定する。また、本実施形態は、このように生成された実装を使用して、第２のデータ構造のインスタンス（コード部分）をハンドリングするソフトウェアツール（例えばパーサまたはコンパイラ）の形でも提供される。

【０１０７】

なお、更なる説明のために、プログラミング言語の３つのカテゴリについて本明細書で触れる。各カテゴリは、以下に示す実施形態において異なる関連性／重要性を有する。この３つのカテゴリは、ベース言語（ＢＬ）、変換言語（ＴＬ）、およびフレームワーク実装言語（ＦＩＬ）である。

【０１０８】

ベース（ＢＬ）は、ハンドリング（評価／解析／変換）するプログラム（コード）を表現する言語である。Ｃ＋＋言語は、ＢＬの可能な一例であるが、このようなハンドリング（解析、変換、および拡張機能）の恩恵を受けるどのような言語を使用してもよい。このため、本発明の文脈で、候補コード部分はＢＬで表現されることはいうまでもない。

【０１０９】

変換言語（ＴＬ）は、メタレベルプログラム変換を表現する（組み込む等）言語である。メタレベルプログラム変換について本明細書中で後述するが、現段階において、どのような所望のメタプログラミング動作を表わしてもよいと考えることができる。選択する実際のＴＬは、それが表現する機能ほどは重要ではない。ＴＬは、実際には１つの言語に限定されず、異なる言語の組み合わせでもよく、ＢＬとして使用される言語も含まれてもよい。ＴＬの主要な特徴は、メタレベルプログラムがＴＬで表現されるという点にある。しかし、ＢＬの特定の場合には、ＴＬに異なる言語を使用することは、表現性、ひいては生産力の点で有利である。

【０１１０】

フレームワーク実装言語（ＦＩＬ）は、本発明を実施するツールを主として記述する言語である。このようなツールは、ＦＩＬで表現された生成済みのライブラリおよびインタフェースを含むか、あるいはこれらにアクセスしてもよい。ＦＩＬも、例えばＣ＋＋であってもよい。ＦＩＬの重要性は、ＢＬで表現された候補コード部分（例えばプログラム）に動作（ハンドリング）し、第一にＢＬの実装を生成するためのツールまたはツールアドオンの生成に関する。

【０１１１】

ツールを、「特殊な」環境（例えばＪａｖａ（登録商標）または他の仮想マシン）で動作するように開発することもできると考えられるが、パフォーマンスを考えると、ほとんどの場合、Ｃ＋＋がＦＩＬの適切な選択肢となる。ＢＬとＦＩＬの間に差がなくてもよい。主として、本発明の実施形態において実施または使用されるさまざまな異なる機能、ツール、およびプロセスの実装に使用される言語を指す際の混乱を避けるために、これらの言語が本明細書中で別の名前で呼ばれる。

【０１１２】

第１のデータ構造記述

第１のデータ構造記述（システムＡＳＴ記述）は、ＢＬのインスタンス（すなわち候補コード部分）を最終的にハンドリングできるようにするためのＢＬの論理構造の記であると考えることができる。本発明の実施形態において利用されるこのような（ＡＳＴ）記述の特性により、このような論理構造を、「代替可能性の模倣多重継承」と「複数型参照」の各特徴を提供する高水準の記述によって記述できるようになる。これらの機能のそれぞれについては以下で詳述する。次に、これらの特徴が、ＦＩＬで利用可能な何らかの機能を使用して、第１のデータ構造（ＢＬを実装するために効力がある）の実装に使用される実装（第２のデータ構造および実装規則）にエンコードされる。

【０１１３】

ここで使用される第１のデータ構造記述（「システム」ＡＳＴ記述）を理解するために、図７は、図３の所望の設計を記載するための１つのやり方を示す。

【０１１４】

論理構造ノードは、ルート、抽象およびノードの３つの型のいずれかである。（上で述べたように）ルートノードは１つだけ存在し、究極的には他の全てのノードがここから派生しなければならない。抽象ノードは、決して作成できないノードであり、他のノードの継承元としてのみ存在するものであり、主として変種の形をリストする文法の非終端部分と似たものである。ノード型のノードは、ノードの内容を記述するボディを含むことができ、現実の具象ノードである。記述中に、各型のノードは、ノード宣言の最初に、関連するキーワード（ｒｏｏｔ、ａｂｓｔｒａｃｔまたはｎｏｄｅ）を記載することによって記述される。

【０１１５】

図７の例に示すように、Ｒｏｏｔという１つのルートノードが存在し、ＮｏｄｅからＳｔａｔｅｍｅｎｔ、ＤｅｃｌａｒａｔｉｏｎおよびＩｎｉｔｉａｌｉｚｅｒという３つの抽象ノードが「疑似継承」している。この疑似継承（以下で説明する）は、＜演算子によって指定される。これらの抽象ノードは、図３の同じ名前の３つのノードに対応する。

【０１１６】

この後、４つの具象ノード（すなわちノード型のノード）が存在し、これらは図３の同じ名前の残りのノードに相当する。これらのノードは、ＩｆＴｈｅｎＥｌｓｅ、ＶａｒｉａｂｌｅＤｅｃｌａｒａｔｉｏｎ、ＩｄｅｎｔｉｆｉｅｒおよびＥｘｐｒｅｓｓｉｏｎと呼ばれる。Ｉｄｅｎｔｉｆｉｅｒ以外のこれらの具象ノードのそれぞれは、図７の記述中に、括弧｛と｝で囲まれたボディを有する。具象ノードは一般にこのようなボディを有するが、これは必須でない。

【０１１７】

ここで特に興味があるのが、ノードＥｘｐｒｅｓｓｉｏｎに指定された疑似継承関係（下記で説明する）である。＜演算子の後に、コンマで区切って２つのノード型のリストが記述されている。これは、Ｅｘｐｒｅｓｓｉｏｎが、ＳｔａｔｅｍｅｎｔとＩｎｉｔｉａｌｉｚｅｒの両方から疑似継承することを示す。これは、ある程度、上で説明したＴｒｅｅＣＣツールでは表現できない多重継承関係に相当する。これが、ＴｒｅｅＣＣに関して上で説明した多重継承関係に、ある程度しか相当しない理由は、以下のとおりである。第１のデータ構造記述の本方法の文脈では、別のノードから疑似継承されるとして記述されるノードは、親の型のノードを予想する参照によって使用されうる。すなわち、この疑似継承関係は、単に、どの型のノードを、どの他の型のノードに代えて使用できるかを示すに過ぎない（ここでは、これを「代替可能性」と称する）。これは、上述の「代替可能性の模倣多重継承」の特徴が意図する意味である。より詳細には、この関係は、実装およびインタフェースの継承は示さない。

【０１１８】

図７の具象ノードの内部には、２つのタイプのフィールド宣言がある。＞＞演算子によって示される第１のタイプは、子参照であり、フィールドが現在のノードから別のノードを参照することと、そのフィールドに有効な参照が置かれた場合には、現在のノードがそ

の他のノードを「所有する」こととを示す。＞＞演算子の左の項は、フィールドの名前であり、＞＞演算子の右の項は、このフィールドによって参照可能なノードの型である。

【０１１９】

：演算子によって示される第２のタイプのフィールド宣言は、データフィールドであり、ノードが特定の型のデータ部分を含むことを示す。子参照と同様に、フィールドの名前が：演算子の左に、フィールドの型が：演算子の右に記述される。

【０１２０】

ルートノードの使用についてみると、この第１のデータ構造（ＡＳＴ）記述は、図３に示した所望の構造と厳密に一致している。このやり方は、この構造を表現するには最も自然なやり方であるが、図８に示すように、（多重疑似継承の代わりに）「複数型参照」を使用することによっても、ほぼ同等の構造を記述することも可能である。

【０１２１】

図８の記述は、図７の記述と構造的に異なるが、究極的には、使用時にほぼ完全に同等な実装を得ることができる。図７と容易に比較できるように、図８においては異なる箇所には下線を付している。

【０１２２】

図８において、Ｅｘｐｒｅｓｓｉｏｎは、ＳｔａｔｅｍｅｎｔとＩｎｉｔｉａｌｉｚｅｒから派生せず、ノードの２つにおいて、子参照の３つに追加の項が追加されている。ＩｆＴｈｅｎＥｌｓｅノードのｔｈｅｎＣｌａｕｓｅフィールドとｅｌｓｅＣｌａｕｓｅフィールドは、参照可能なノード型のリストを指定しており、Ｓｔａｔｅｍｅｎｔノードのほかに、Ｅｘｐｒｅｓｓｉｏｎノードが指定されている。同様の変更は、ＶａｒｉａｂｌｅＤｅｃｌａｒａｔｉｏｎノードのｉｎｉｔフィールドにも行われている。子参照フィールドに複数のノード型が指定されている場合、リスト内のどのノード型も、そのフィールドによって参照される有効な型であることを意味すると解釈される。

【０１２３】

このため、図８の例では、ノードＥｘｐｒｅｓｓｉｏｎが、ＳｔａｔｅｍｅｎｔおよびＩｎｉｔｉａｌｉｚｅｒに代替可能であることを示すために、ノードＥｘｐｒｅｓｓｉｏｎに関する多重疑似継承の使用が削除されており、代わりに、その代替可能性が子参照フィールドに直接エンコードされている。いずれの場合も、Ｅｘｐｒｅｓｓｉｏｎノードは、ＳｔａｔｅｍｅｎｔノードまたはＩｎｉｔｉａｌｉｚｅｒノードを使用できる全ての子参照フィールドにおいて使用することができる。

【０１２４】

なお、有用性の視点から、図７の多重疑似継承のアプローチは、図８のアプローチ（複数型参照アプローチが過度に繰り返される）よりも、この例には簡潔かつ明解である。しかし、複数型参照はより特殊であるため、多重疑似継承が、望ましいよりも一般的な状態では、極めて便利となりうる。実際、多くの言語文法は、特定の場合、より特殊な変種の形態を使用しており、このため、記述中で同様の表現を使用することができることにより、ＢＬの文法の構造に一致させることが望ましい場合に、設計者がこれを行うことが可能となる。

【０１２５】

上で述べたように、本記述スキームの文脈における疑似継承とは、大部分のオブジェクト指向言語が提供する継承とは多少異なる特定のタイプの継承を意味する。多くの言語では、「継承」とは、主として、派生クラス（例えば子ノード）が、親クラス（例えば親ノード）からのインタフェースおよび実装を取り込むことと、（プログラム言語によっては）親クラスを使用可能な場所ではどこでも、派生クラスを使用することができることと、派生クラスの関数が親の関数の代わりに呼び出されるように派生クラスが親クラスの関数をオーバーライドすることができること（「ポリモルフィズム」と呼ばれる）とを意味する。しかし、本発明の文脈では、第１のデータ構造記述スキームで指定される疑似継承は、親クラスを使用できる場所で派生クラスを使用できること（ここでは「代替可能性」と呼ぶ）を意味するに過ぎない。派生クラスは、親クラスから何らのインタフェースまたは

実装機能を継承せず、この関係によってポリモルフィックな関数呼び出しに対応する能力も継承しない。実際、ＢＬの実装として第２のデータ構造および実装規則を定義する実装コードの生成に関して下で説明するように、ＢＬで表現されたコード部分をハンドリングで使用するために、第２のデータ構造の派生クラスは、第２のデータ構造の親ノードのクラスから実際に継承していない。この点は、第２のデータ構造を、本発明の模倣多重継承モデルをサポート可能にする大きな要因である。

【０１２６】

この唯一の例外はルートノードであり、第２のデータ構造（後述するように）においては、他の全ノードがルートノードから直接継承する。第２のデータ構造におけるルートノードからの直接の継承は、従来の完全な継承（すなわち、代替可能性、実装およびインタフェースの継承）である。

【０１２７】

本論理構造（ＡＳＴ）記述スキームにおける別の重要な要素は、「所有された」参照と「所有されない」参照である。図７と図８の例は、所有された参照（「子参照」と呼ばれる）を排他的に使用する。名前が示すように、子（所有されている）参照は、構造中でノードが、自身の子ノードをどのように参照するかを指定するために使用される。子ノード参照の存在は、実質的に全体的な論理構造を形成する。例えば、図７と図８の例のＥｘｐｒｅｓｓｉｏｎノードは、ｌｈｓとｒｈｓの２つの子ノード参照を含む。表現（すなわちインスタンス）が実行時に生成される場合、これらの参照が使用されて表現ツリー（「インスタンス」データ構造）が構築され、このツリーでは、任意の複雑さの表現が形成されるまで、各Ｅｘｐｒｅｓｓｉｏｎノードは、２つの子Ｅｘｐｒｅｓｓｉｏｎノードに参照することができ、これが再帰的に行なわれる。この表現ツリー内で、各Ｅｘｐｒｅｓｓｉｏｎノードは、自身が参照する２つの子ノードを所有する。「所有」とは、親ノードが子ノードの寿命を制御することを意味し、親ノードが削除されるか、またはその子参照フィールドが消去されると、子ノードも通常は破棄される。なお、複数の親を持つことにより多少これが複雑となる。複数の親を持つことについては、下で更に説明する。

【０１２８】

所有されない参照は、所有された参照と同様であるが、違いは、親ノードが参照されているノードの寿命を制御しないという点である。実際、これは真の親子関係でさえないため、このような参照は「リンク参照」と呼ばれる。リンク参照については、親ノードは、参照されたノードの寿命を制御せず、親ノードが存在する限り、参照されたノードが存在し続けることを保証することができない。参照されたノードは別の親ノードによって所有されるが、その別の親ノードの寿命が親ノードの寿命よりも短いことがあり、その子ノードの寿命も同様である。

【０１２９】

リンク参照は、子参照と同様にこの第１のデータ構造（ＡＳＴ）記述スキーム中で指定されるが、＞＞演算子の代わりに＞演算子を使用する。リンク参照は特に、ＡＳＴ内のシンボル使用点を、シンボル宣言点にリンクするのに有用であるが、他の多くの用途にも使用することができる。

【０１３０】

また、子参照とリンク参照間の違いは、論理構造（ＡＳＴ）をウォークおよび操作する場合には重要である。論理構造をウォークする際には、主として、リンク参照は真の子ノードを表さないために無視され、その真の親を反復することによりウォークされる。論理構造を操作する際には、リンク参照は、このような参照で真の子ノードでないため、ノードに対する入力親接続のリスト（incoming
parent connections to a node）に現れない。

【０１３１】

本第１のデータ構造記述スキームの別の重要な要素は、ワイルドカード疑似継承の使用に関する。このスキームは、模倣多重継承（すなわち多重疑似継承）の任意の使用を可能にするため、あるノードを、他の全てのノードから疑似継承するように作成でき、このこ

とは、子／リンク参照において、このノードを、他の全ノードのいずれかの代わりに使用できることを意味する。この機能は、このような便利な機能であるため、図９に示すように、この論理構造記述スキームで特別の形式を有する。

【０１３２】

図９に示すように、ワイルドカード疑似継承を使用するノードの構文は、他のどのノードと同じであるが、特定のノードのリストの代わりに、疑似継承元を示す＊記号を有する。このようにノードが指定される場合、記述において、疑似継承リストが他の全ノードを含むと解釈される。このようなノードはＳＥＢでもよい。

【０１３３】

実装コードの生成

本発明の実施形態の原理を実現する、すなわち、特定の言語（ＢＬ）で記述された候補コード部分を効率的にハンドリング（理解／解析／操作／強化されるなど）するために、候補コード部分（「インスタンス」ＡＳＴ）の表現が生成される。このような「インスタンス」ＡＳＴを生成するために、上記の第１のデータ構造記述の実装が使用され、以下の説明は、この実装（より詳細には、第２のデータ構造および実装規則を含む）に関する。

【０１３４】

このため、本発明は、少なくとも一部には、第１のデータ構造記述からのこのような実装（第２のデータ構造および実装規則）の生成に関する。本発明の文脈では、このような第２のデータ構造は、それぞれ第１のデータ構造（論理構造）のノードに対応するノードを有するとみなすことができる。この意味では、この２つの構造は、僅かに異なるが、互いに同等である（すなわち、いずれもＢＬを表わす）とみなすことができる。

【０１３５】

このような実装の生成により、この実装を表現する、すなわち、第２のデータ構造と実装規則とを表現する実装コードが得られる。本実施形態では、このような実装コードはＦＩＬで表わされ、このため、そのＦＩＬに大きく依存している。ここで想定する実装言語（ＦＩＬ）は、Ｃ＋＋であるが、例えばＪａｖａ（登録商標）等の他の静的型付けオブジェクト指向言語などの、他の言語が使用されてもよいことはいうまでもない。本生成の特定の態様は、どのような実装言語にも適用することができる。

【０１３６】

本実装コード生成の１つの重要な機能は、第１のデータ構造記述で表現された「模倣」多重継承、すなわち多重疑似継承が、第２のデータ構造（および実装規則）を定義するために、実装コードに変換される方法にある。上で説明したように、上述の第１のデータ構造記述スキームで表現された「継承」は、代替可能性のみをモデルする疑似継承である。このため、論理構造ノード間のこのような疑似継承は、実装言語（ＦＩＬ）の「通常の」継承メカニズムを使用して、第２のデータ構造の対応するノード間のリンクにより、これらのノード間の通常の継承として直接実装できない。この理由は、大部分の言語は、疑似継承によって暗示される継承の概念とは更に別の意味を付加しており、これが、多重継承に関する問題を引き起こすためである。更に、多くの実装言語（ＦＩＬ）は、通常の多重継承をサポートしていない。

【０１３７】

ＢＬの実装に使用される、第２のデータ構造を定義する本実装コードによって提供されるこの重要な機能は、ルートノード以外の第２のデータ構造のノードが、ルートデータ構造ノードから直接継承することにある。実際、（ルートノード以外の）全ノードが、通常の意味（すなわち、代替可能性、実装およびインタフェース）でルートノードから継承し、互いに継承しない。これには単一の継承のみが必要となり、第２のデータ構造のノードはインタフェースまたは実装を互いに継承しないことを意味する。この実装第２のデータ構造を、図３の目的の論理構造に適用した例を、図１０（本実施例の第２のデータ構造を表わす）に示す。

【０１３８】

図１０に示すように、（ルートノード以外の）全ノードがルートノード（ノード）から

継承する。（第１のデータ構造記述では）ＳｔａｔｅｍｅｎｔとＩｎｉｔｉａｌｉｚｅｒから疑似継承されていたＥｘｐｒｅｓｓｉｏｎも、（実装の第２のデータ構造では）Ｎｏｄｅからのみ継承する。（どの種類の）全ての多重継承が、実装から除去されている、むしろ、第２のデータ構造である実装の一部から除去されている。しかし、実装が、ＢＬと、その関連する論理構造とについて正しくなるためには、第１のデータ構造記述で表現された多重疑似継承関係（すなわち代替可能性）を、何らかの形で表わすことが必要となる。

【０１３９】

また、図１０に示すように、ＩｆＴｈｅｎＥｌｓｅ、ＶａｒｉａｂｌｅＤｅｃｌａｒａｔｉｏｎ、およびＥｘｐｒｅｓｓｉｏｎの子参照フィールドが全て、型Ｎｏｄｅ（ルートデータ構造ノード型）のデータ構造ノードを参照している。（第１のデータ構造から第２のデータ構造への）この変更が必要な理由は、実装言語（ＦＩＬ）の型システムが、（ほとんどの場合）ＳｔａｔｅｍｅｎｔまたはＩｎｉｔｉａｌｉｚｅｒ、あるいは他の関連する任意のノードについて、これらが関連付けられていない（例えば、Ｅｘｐｒｅｓｓｉｏｎが、第２のデータ構造のＳｔａｔｅｍｅｎｔから派生していない）ため、Ｅｘｐｒｅｓｓｉｏｎの代替可能性を許容しないためである。

【０１４０】

図１０の第２のデータ構造は、大半の実装言語（ＦＩＬ）によって直接サポートされ有利であるが、例えば、図７と図８に示すような、第１のデータ構造記述によって記述された、ＢＬを表わす元の目的の論理構造（第１のデータ構造）の特定の重要な要素を失っている。このような要素には、対応する第２のデータ構造のノード間のリンクのパターンに存在しない、例えば（疑似継承経路、すなわち代替可能性の継承を表わす）第１のデータ構造のさまざまなノード間のリンクのパターンが含まれる。下で説明するように、実装がＢＬに対して正しくなるように、これらの要素が、下に説明するように、実装規則によって実装に再導入される。

【０１４１】

本実装方法は、元の論理構造記述で指定されている型規則（すなわち代替可能性）を反映する型チェックメカニズム（実装規則）によって、これらの要素を再導入する。これは、好ましくは、第１のデータ構造（ＡＳＴ）記述に指定されている、模倣多重継承または多重疑似継承（図７）と複数型参照（図８）規則とに従って、ツリー内の全ての参照フィールドに対して、互換する型のノードの参照が割り当てられることを保証する動的な型チェッカーによって実行される。

【０１４２】

型チェックのプロセスは、後で更に詳細に説明するが、この段階では、システムの大きな利点について概要を説明する。

【０１４３】

このような型チェックの目的は、型安全性を保つことにある。生成される実装は、ＢＬで表現されたコード部分に対する「インスタンス」ＡＳＴを構築するために使用されうる。本システムの型チェックは、対象のコード部分（候補コード部分）に対する任意の操作または「インスタンス」ＡＳＴの作成中および／またはその後に実行されうる。このようにして、各段階で、「インスタンス」ＡＳＴが、ＢＬに対して有効である（すなわち、第１のデータ構造記述で表現され、実装規則によって実装される代替可能性の規則に違反していない）ことを保証することができる。しかし、第１のデータ構造は、代替可能性の多重継承の点で極めて複雑であり、いくつかのノードは、第１のデータ構造の他の任意のノードに対して代替可能であると定義されていることがある。実装は、実装規則によって維持される代替可能性のこれらの規則を許容する一方で、「インスタンス」ＡＳＴが作成される第２のデータ構造において１つの継承のみが使用されるため、多重継承の問題が発生しないことを保証する。実際、（ワイルドカード疑似継承を有する）ＳＥＢは、第１のデータ構造記述中に比較的容易に作成できる（図９を参照）が、第２のデータ構造は、極めて簡潔であり、単一継承関係のみを有する。実装規則のために代替可能性関係の複雑さが

残されるが、これを同様に簡単に表現することができる。したがって、ＳＥＢは他の任意の型のノードに対して代替可能であると定義されるため、「インスタンス」ＡＳＴ（候補コード部分を表わす）は、どの位置においても、いかなる数のインスタンスＳＥＢを有することができる。

【０１４４】

好適な型チェッカーによって使用される型チェック規則は、一般に以下の通りである。任意の所定の参照フィールドの有効なターゲットとしてリストされている全てのノード型に対して、第１のデータ構造記述におけるこれらの型、ならびに疑似継承および／または複数型参照によって表現された、これらから派生される任意の型は、参照フィールドによって参照される有効なノード型である。例えば、例示の第１のデータ構造記述の参照フィールドが、ターゲット型としてＩｎｉｔｉａｌｉｚｅｒを指定している場合、「インスタンス」ＡＳＴ内の型Ｉｎｉｔｉａｌｉｚｅｒ、ＩｄｅｎｔｉｆｉｅｒおよびＥｘｐｒｅｓｓｉｏｎのノードに対する参照は、型チェックにパスするが、他の全てのノード型は、エラー（実装に依存するが、通常、例外処理メカニズムによってハンドリングされる）を発生させる。これらの規則をサポートする効率的な型チェックメカニズムの特定の実装は、実装の詳細事項であるが、例えば、ノード型ごとに、そのノード型から派生する全てのノード型を含む（ハッシュベースの）集合を作成することは、極めて効率的な解決案となる。

【０１４５】

動的な型チェック（すなわち、コンパイル時ではなく実行時）のために、アクセッサ関数を介してノード内のフィールドへのアクセスを提供してもよく、これにより、実装が、ノードが有効なことを保証するために参照に対する型チェックシステムを起動することが可能となる。例えば、Ｅｘｐｒｅｓｓｉｏｎノードの場合、生成される（Ｃ＋＋のために）アクセッサ関数は、図１１に示すようなものとなる。

【０１４６】

図１１に着目すると、最初の関数ペア（いずれもｏｐ（））は、データフィールドであり参照型ではないＯｐｅｒａｔｏｒＴｙｐｅフィールドへのアクセスを提供する（一方は読み出し用、もう一方は書き込み用）。アクセッサ関数は、主として、全フィールド型にわたり一貫したインタフェースを保つために、データフィールドに対して生成される。

【０１４７】

ｌｈｓ（）とｒｈｓ（）の関数ペアは、対応するＡＳＴ記述におけるｌｈｓとｒｈｓの２つの子参照へのアクセスを提供する。この２つの関数の対応する関数は、パラメータとしてＮｏｄｅＰｔｒをとり、関連するフィールドに新しい値をセットするために使用され、これらの関数の実装は、互換する型であることを保証するために入力（incoming）参照に対する型チェッカーを起動する。

【０１４８】

図１１に示すように、第１のデータ構造記述の参照はＣ＋＋実装規則のポインタに変換される。本方法において、これらは、「参照カウントスマートポインタ」と呼ばれる特別なタイプのポインタに変換される。このポインタにより、複数のオブジェクトが、別のオブジェクトの参照を保持しているオブジェクトの数をカウントすることによって、その別のオブジェクトの所有権を共有することができる。これらのスマートポインタの１つがオブジェクトに対する参照をとると、参照カウントをインクリメントし、オブジェクトへの参照を解放すると、参照カウントをデクリメントする。参照カウントがゼロになると、オブジェクトがどのポインタからも参照されておらず、破棄される。このメカニズムにより、第２のデータ構造の任意のインスタンスノードを、複数の親によって所有することが可能となり、グラフ構造（ＤＡＧ）に近づく。

【０１４９】

本方法では、子参照とリンク参照は、いずれもスマートポインタを使用するが、実際には、２つの異なるが、関連するスマートポインタ型を使用する。子参照は、上で説明したポインタである強いポインタを使用するが、リンク参照は弱いポインタを使用する。強／

弱スマートポインタ方式により、オブジェクトが、強いカウントと弱いカウントの２つの参照カウントを有することが必要となる。強いカウントは、オブジェクトを「生存状態」に保持するために使用されるが、強いカウントと弱いカウントの組み合わせは、カウント自体を「生存状態」に保持するために使用される。例えば、オブジェクトが１つの強いポインタと弱いポインタとによって参照される場合、強いカウントが１、弱いカウントが１となる。強いポインタがその参照を解放すると、強いカウントは０になり、オブジェクトが破棄されるが、総（弱＋強）カウントが１であるため、カウントが残る。オブジェクトがこの状態のときに弱いポインタが逆参照されると、破棄されたオブジェクトに対する参照ではなく、ＮＵＬＬが得られる。このメカニズムにより、弱いポインタを使用して、他のオブジェクトによって所有された（および破棄された）オブジェクトを安全に参照することができる。

【０１５０】

このため、第１のデータ構造記述の参照フィールドからＦＩＬ（この例では、Ｃ＋＋）実装規則への変換は、かなり直接的であり、本例の主な違いは、これらが、Ｃ＋＋のｔｙｐｅｄｅｆ宣言を介してＮｏｄｅＰｔｒという名前のスマートポインタを使用して表わされるという点にある。ＮｏｄｅＰｔｒの名前は、ルートノード型の名前（Ｎｏｄｅ）の後にＰｔｒを付加して得られる。特定のノード型に対する全スマートポインタも同様に名付けられるが、（全ての参照がＮｏｄｅＰｔｒに変換されるため）第２のデータ構造では直接使用されない。

【０１５１】

記述固有の型規則を強制するために実装によって追加される型チェックの好適な形態は、動的なチェッキングメカニズムを使用するが、特定の実装言語（Ｃ＋＋を含む）が、実行時ではなく、コンパイル時に特定の種類の型エラーをプログラマに通知できるカスタムの静的な型チェック方式をサポートしてもよい。

【０１５２】

例えば、（少なくとも本例では）Ｅｘｐｒｅｓｓｉｏｎから派生するノードが存在しないため、本例のＥｘｐｒｅｓｓｉｏｎのｌｈｓフィールドとｒｈｓフィールドは、型Ｅｘｐｒｅｓｓｉｏｎのノードしか参照できない。この場合、ｌｈｓフィールドとｒｈｓフィールドを表わすか、またはｌｈｓ（）とｒｈｓ（）の関数ペアのためにＮｏｄｅＰｔｒを使用することは不要であり、代わりにＥｘｐｒｅｓｓｉｏｎＰｔｒを使用することができる。

【０１５３】

しかし、この例は実際には発生しうるが、かなり不自然な例である。より一般的なのは、ＩｆＴｈｅｎＥｌｓｅのｔｈｅｎＣｌａｕｓｅフィールドであり、このフィールドはＳｔａｔｅｍｅｎｔに対する参照を可能にし、（継承を介して）ＩｆＴｈｅｎＥｌｓｅおよびＥｘｐｒｅｓｓｉｏｎに対する参照を可能にする。ｔｈｅｎＣｌａｕｓｅは、複数の型のオブジェクトを参照することができるため、より特定的ではなくＮｏｄｅＰｔｒでなければならないが、アクセッサ関数はより特定的であってもよい。ｔｈｅｎＣｌａｕｓｅのためのアクセッサ関数の１つの可能な組が図１２に示される。

【０１５４】

図１２に示すように、読み出しアクセッサはＮｏｄｅＰｔｒを返し、この点は、ｌｈｓ（）とｒｈｓ（）の例とは一切変わらない。しかし、Ｃ＋＋関数オーバーロードを利用（Ｃ＋＋は好適なＦＩＬである）し、他の型ではなくこの３つの型の参照により（Ｃ＋＋コンパイラによってコンパイル時にチェックされうる）、書き込みアクセスを提供するための３つの書き込みアクセッサが存在する。同じ効果は、参照型の妥当性に対するコンパイル時アサートを含む１つのテンプレート化された関数により得ることもできる。全ての実装言語（ＦＩＬ）がこの効果を実現するために必要な機構を提供しているというわけではなく、提供していないＦＩＬは異なる方法でこのような機能を提供してもよい。しかし、静的な型チェックは必須でなく、可能な場所で追加の補助の層をプログラマに提供するに過ぎないことはいうまでもない。なお、Ｃ＋＋のオーバーロード解決では戻り型が使用さ

れないため、関数オーバーロードを介してではなく、静的型チェック読み出しアクセッサをＣ＋＋に提供することも可能である。

【０１５５】

一般に、第１のデータ構造記述（ＡＳＴ）のワイルドカード継承が存在するため、対応するデータ構造の定義を生成するために、実装コードジェネレータの特定の領域に特定のサポートが必要となる。ワイルドカード継承付きのノードが宣言された場合、そのノードは、単に、第１のデータ構造記述の既存の全ノードに対する派生型として追加される。実装コードジェネレータのほかのメカニズムは全て同じである。

【０１５６】

（ＳＥＢに関して）ワイルドカード継承を使用した場合は、型チェックシステムが何らかの形で脆弱化するように見えるが、そんなことはない。実際、静的な型チェックであっても僅かに影響を受けるに過ぎない。図１２のｔｈｅｎＣｌａｕｓｅの例のアクセッサ関数を例にとり、ワイルドカード継承を有するＡＳＴに、Ｗｉｌｄｃａｒｄというノードが追加されたと仮定すると、ＷｉｌｄｃａｒｄＰｔｒを受けるｔｈｅｎＣｌａｕｓｅのための追加の書き込みアクセッサ関数が生成されるだけである。当然、第２のデータ構造の全参照フィールドに対してこのような追加の関数が１つ存在するが、この場合でも、静的な型チェックにより、妥当なレベルの安全性がプログラマに対して提供されていることが明らかとなろう。

【０１５７】

本発明の実施形態において使用されるツリーウォーキングの１つの方法は、（既存のＡＳＴコードジェネレータによってほぼ排他的に使用される）ビジターパターンではなく、反復子モデルを使用する。ビジターパターンと反復子モデル間の根本的な違いは、ビジターパターンはウォークプロセスの制御を想定しているが、反復子モデルは、ユーザに制御を委譲する点にある。ビジターパターンアプローチを使用する場合、ユーザが訪問関数を呼び出し、訪問する全ノードに対して、ビジター関数によって呼び出されるコールバック関数を提供する。反復子の場合、ユーザは、反復子オブジェクトを作成して、反復子に現在のノードを問い合わせ、各反復を進めるように指示するループを記述する。

【０１５８】

反復子モデルアプローチの主要な利点は、クロージャーのサポートを提供しない実装言語（ＦＩＬ）（Ｃ＋＋およびＪａｖａ（登録商標）など）の場合であっても、ユーザが反復プロセスを制御し、反復の進行に伴うローカルなプログラム状態を参照することができることにある。クロージャーを提供する実装言語は、ビジターパターンアプローチでもこの特性を有するが、本システムは、このようなサポートのない言語でも、できる限り使用可能とすることを目指している。

【０１５９】

ＡＳＴの簡単な反復子モデルは、以下の関数を提供する。

ｍｏｒｅ（）－反復子が生成すべき更に別のノードを有する場合、真を返す。

ｎｏｄｅ（）－反復子が現在居るノードへの参照を返す。

ｎｅｘｔ（）－ウォーク時に、反復子がサポートする特定のウォーク順序の決定（最も一般的なのは前順ウォークである）に従って、反復子を次のノードに進める。

【０１６０】

これらの関数を使用するＣ＋＋ループの例を、図１３に示す。この例では、ｓｔａｒｔＮｏｄｅは、反復を開始する（反復は、構造内のどこからでも開始することができる）「インスタンス」ノードへの参照であり、ＴｒｅｅＩｔｅｒａｔｏｒは、生成されたクラスであり、反復子機能を提供する。各反復において、現在のノードへの参照がｎｏｄｅにコピーされ、これが、このノードに動作する以下の任意のコードによって使用されうる。

【０１６１】

不均一ツリーに反復子機能を提供することは、ビジターパターン機能を提供することとは多少異なる。この２つのメカニズムは微妙に異なるようにみえるが、反復子アプローチには、多少の労力が追加で必要となる。ビジターコードの生成時に、コードジェネレータ

は各ノードにｖｉｓｉｔ（）関数を生成してもよく、その例が図１４に示される。この例は、再帰を介して実装される前順ウォークを示す。まず、各ｖｉｓｉｔ（）関数は、現在のノードへの参照を渡して、ユーザコードによって提供されるコールバック関数を呼び出す。次に、ノード内の非ｎｕｌｌの各子参照フィールドについて再帰的にｖｉｓｉｔ（）を呼び出し、これにより、参照されたノード型に対して特定のｖｉｓｉｔ（）関数が呼び出され、これがツリーのリーフにおいて再帰が終了するまで反復される。

【０１６２】

しかし、ｖｉｓｉｔ（）関数が、コールバックを呼び出すのではなく、呼び出し元に譲る（yield to the caller）ことを可能にするメカニズムが必要となるため、このアプローチは、多くの実装言語の反復子方式に容易に変換することができない。この効果は、クロージャーによって模倣できるが、全ての実装言語がクロージャーのサポートを提供するというわけではないため、異なるメカニズムを使用することが望ましい。

【０１６３】

使用可能な反復メカニズムを提供する方法が多く考えられるが、一般に、各ノード型に対して参照フィールドの列挙メカニズムを提供するという同じ手法になる。この列挙メカニズムは、任意のノード型に対して、呼び出し元が、ノードからＮ番目の参照フィールドを要求可能にする。このようなメカニズムが提供される場合、１つのノードの反復においては、列挙子がｎｕｌｌ参照（または何らかの同等のエラーコード）を返すまで、単に、全参照フィールドが、０から上向きに要求される。

【０１６４】

列挙子メカニズムを実装する方法が数多く存在する。１つの可能な方法は、各ノード型に対して、スロット番号を、フィールド参照への参照に変換するｃａｓｅ文を有するｅｎｕｍｅｒａｔｅ（）関数を提供する方法であり、これが図１５に示される。しかし、本コードジェネレータツールは、データベースによる列挙方式を優先しており、この技術は使用しない。

【０１６５】

本反復メカニズムを拡張して、スタックを使用して、子ノードの再帰が終了したときに戻るために親ノードと参照フィールドとを記録することによって、「インスタンス」ツリー全体を反復してもよい。明示的な親参照がツリーにエンコードされる場合、スタックの使用を省略することができ、この結果、反復子が、更にメモリ効率が高いものとなる。反復子を更に拡張して、（ビジターメカニズムのように）異なるウォーク戦略をサポートすることも可能である。本反復子は、例えば前順ウォークと後順ウォークの両方のほか、前順ウォークと後順ウォークの組み合わせを提供してもよい。ユーザコードは、現在のノードの子が既にウォークされている場合に真を返すヘルパー関数のｉｓＰｏｓｔ（）によって、ウォークのどの部分（前または後）から現在のノードに到達したかを決定することができる。

【０１６６】

反復子アプローチの１つの利点は、ほとんどの実装言語について、反復子メカニズムの一番上にビジターメカニズムを実装することが可能であるが、この逆は一般に不可能な点が挙げられる。実際、このため、ユーザが、数行のコードで自身のビジターを作成できるため、本方法（好適な形において）は、ビジターサポートコードを生成しない。

【０１６７】

前述のように、ノード内に明示的な親参照を提供しないツリー構造は、操作が面倒なことがある。したがって、実装の本方法は、ノードが親参照を含むことを許容する。しかし、サブツリーの浅いコピーを実行できることが、特にＧＬＲ（一般化された左から右の右端の派生）パースメカニズムと組み合わせて使用された場合には、効率的な操作のためにも、ＡＳＴ表現の望ましい特質である。この２つの可能性は共存せず、ノードが、複数の親を有することができ、これらの親を直接参照できる場合、実際に、１つの親参照ではなく複数の親参照を必要とし、ノードごとに何らかのタイプのリストまたはアレイ構造が示唆されるが、これは、明示的な親リンクや複数の親を有さない表現よりも遙かに非効率と

なる可能性がある。更に、ノードが複数の親参照を有する場合、そのノードをその親の１つから分離する場合に、いずれの親参照を破棄するべきかという決定が必要となる。

【０１６８】

本ＡＳＴ表現は、これらの問題を二段階の解決案により扱う。第一に、大部分のノードが親を１つしか有さないため、この場合、１つの親を用いて参照を最適化することが可能である。第２に、複数の親を有するノードでは、１つの親参照を再利用して、そのノードに対する複数の親の参照を含む親リスト構造を参照することが可能である。

【０１６９】

第一に、複数の親の使用は、一般に、効率の問題（例えば、浅いコピーは、深いコピーよりも記憶領域と計算量に関して非常に低コストとなる）によるため、複数の親使用される場合にのみこの効率を犠牲にすることは、妥当なトレードオフである。このため、複数の親を有するノードが、追加の記憶領域を使用することと、分離操作を実行するために追加の計算を必要とすることは、一般に許容される。

【０１７０】

本ＡＳＴ実装へのこの追加の全体的な影響として、実装される構造は、ツリーよりも有向非巡回グラフ（ＤＡＧ）のようになるが、簡単なツリー構造の効率の多くが保たれる一方で、必要な場合にはグラフのような機能を提供するように構成されるという点が挙げられる。

【０１７１】

用途

以下に説明するように、本システム（単に本発明の一実施形態のみにすぎない「本システム」）が、ルートノードから直接継承するように第２のデータ構造（ルートノード以外の）の全ノードを構成し、元の第１のデータ構造記述に指定されている型規則（実装規則）を再導入するために型チェックメカニズムを実装することによって、記述された論理構造（第１のデータ構造）を実装できることにより、柔軟性および堅固性の点で多くの利点が得られる。

【０１７２】

本システムおよび本システムを採用したツールは、文法非依存の構文拡張を記述するための方法（上記では「構文拡張ブロック」またはＳＥＢと呼び、この機能を実現するために、いわゆるワイルドカード継承を使用する）を使用することができる。これは、最新のプログラム言語に必要な複雑な言語文法記述に干渉する必要なく、新しい言語機能のＢＬへの取り込みを容易にする。本システムは、コード生成技術を多用して、新しいプログラミング言語用のコード解析および変換システムを構築するためのプロセスを簡略化する。本システムは、論理構造記述スキーム（第１のデータ構造記述）と、構造に対する文法ベースの制約を提供する（すなわち型チェックが実装される）実装ジェネレータ（第２のデータ構造および実装コード）を採用する一方で、ＢＬの基礎的な言語文法からはどうしても得られないワイルドカード文法要素（構文拡張またはＳＥＢなど）について、制御された柔軟度を許容する。これは、上で詳述し、実装に組み込まれた疑似継承と複数型参照の技術（すなわち代替可能性）によって実現される。

【０１７３】

したがって、本システムは、メタプログラマにとって有用な、各種ツールおよび他のアプリケーションに利用できるフレームワークを提供すると考えることができる。このようなツール（この多くについて以下に説明する）を生成可能なＢＬ（ベース言語）で記述された候補コード部分に対する「システム」論理構造（ＡＳＴ）およびパーサ（例えばＧＬＲパーサ）を表現することが可能である。本システムの仕様（すなわち実装）を使用して、ＢＬで表現されたプログラム（候補コード部分）を処理するためのＦＩＬ（フレームワーク実装言語）においてメタコンパイラツールおよびライブラリを生成することが可能となる。次に、上述のメタコンパイラツールおよびライブラリによって、ＢＬで表現されたプログラムソースコード（候補コード部分）を、ＢＬに対する「インスタンス」ＡＳＴに変換することが可能となる。また、ＴＬ（変換言語）で表現されたメタレベルプログラム

を使用して、生成された「インスタンス」ＡＳＴを解析および任意選択で変換することも可能となる。更に、変更された「インスタンス」ＡＳＴ（抽象形式）を、ネイティブなテキスト形式（入力形式に相当するか、アプリケーションによっては他の何らかの同様の有用な形式）のＢＬのコードに逆変換することも可能となる。

【０１７４】

例として、図１６は、コード変換ツールとしての使用における、本システムを実施するメタコンパイラ１０の単純化した使用例の模式図であり、この図において、ＢＬはＣ＋＋である。したがって、メタコンパイラ１０は、変換エンジンを備える。

【０１７５】

図１６からわかるように、ＢＬで記述された候補コード部分１５が、ＴＬで記述されたメタ変換２０と共にメタコンパイラ１０への入力として提供される。メタコンパイラ１０は、候補コード部分１５に対して、（「インスタンス」ＡＳＴを定義する）ＡＳＴ記述コードを生成し、上で詳細に記載したように、本システムに従ってこのＡＳＴを実装するように動作可能である。したがって、入力メタ変換２０に従って（例えば、インスタンスＳＥＢの位置に基づいて）、候補コード部分１５に対してＡＳＴを操作することが可能である。

【０１７６】

型安全性を保つために、得られたＡＳＴがＢＬに関して有効であることを保証するために、ＡＳＴの操作中におよび／またはその後に、本システムの型チェックが実行されうる。

【０１７７】

図１６に点線で示すように、（このような操作の前、間および／または後に）候補コード部分１５に対するＡＳＴが、外部ツールによる検査または解析のため、および／またはユーザが目で見るために便利な任意の形式２５で表わされうることはいうまでもない。ＧｒａｐｈＶｉｚによる視覚化層は、例えば、検査およびデバッギングの目的のために提供されうる。

【０１７８】

操作後に、メタコンパイラ１０は、操作された「インスタンス」ＡＳＴを、操作された「インスタンス」ＡＳＴと同等であり、ＢＬで記述されている、変換されたコード部分３０に変換するように動作可能である。

【０１７９】

実際には、本システムは、２つの形式の入力（プログラムコードとメタレベル変換）を、一元的表現にマージし、プログラムコードとメタレベル変換プログラムの両方が、同じソースファイル内で共存できるようにする。これが、図１７に示されている。

【０１８０】

図１７は、図１６のようなコード変換ツールとしての使用における、本システムを実施するメタコンパイラ１０の単純化した使用例の模式図であり、この図において、ＢＬはＣ＋＋である。

【０１８１】

図１７からわかるように、図１６のように、ＢＬで記述された候補コード部分１５が、ＴＬで記述されたメタ変換２０と共にメタコンパイラ１０への入力として提供される。しかし、図１７では、この２つの入力が、１つの結合された入力３５として一緒に提供されている、すなわち、メタ変換２０が、候補コード部分１５に埋め込まれている。

【０１８２】

この拡張の埋め込みは、（上で詳述したように）「構文拡張」またはＳＥＢによって実現される。ＳＥＢの使用は、既存の言語構文内に言語拡張を埋め込むための文法非依存の技術である。この方式により、ＢＬで記述された既存のソースコードを、ＴＬで記述されたメタレベル変換に徐々に取り込むことが可能となり、その際、基本的な最小要件は、ＢＬ仕様から一切逸脱がない（with a base minimum requirement of zero deviations from the

BL specification）。変更されていないソースコードが、「メタ拡張」ソースコードと全く同様に扱われるため、このような方式は、プログラミングコミュニティ内での現実的な採用／理解にとって重要であることが考察される。

【０１８３】

本発明のシステムの更に別の用途が考察される。本発明の実施形態によって提供されるフレームワークは、一般的な用途を有するため、これらの用途がシステムの普遍的な柔軟性を示す一方で、多くの予想される使用例を提供するものである。本明細書に開示される使用例は、決して全てを網羅するものではない。

【０１８４】

本発明のシステムの１つの用途は、埋め込まれたコード変換（すなわち言語拡張）の使用を可能にすることにある。図１８は、図１６，１７のようなコード変換ツールの一部としての使用における、本システムを実施するメタコンパイラ１０の単純化した使用例の模式図であり、この図において、ＢＬはＣ＋＋である。埋め込まれたコード変換が使用されるため、図１８の使用例は、図１７の使用例と極めて似ている。

【０１８５】

図１８において、１つの結合された入力３５（すなわち、メタ変換２０が候補コード部分１５に埋め込まれている）が、メタコンパイラ１０によって実行される、ＴＬで表現された変換を含むＣ＋＋（ＢＬ）の変更版で表現されていると考えられる。メタコンパイラ１０は、次に、操作されたＡＳＴに相当する、ＢＬで記述された変換されたコード部分３０（ネイティブＣ＋＋出力）を生成する。次に、変換された出力が、従来のＣ＋＋コンパイラ４０によって処理され、オブジェクトコード４５が生成されうる。

【０１８６】

本発明のシステムの別の用途は、コードの理解を可能にすることにある。コードの理解には、ＡＳＴの解析と、抽出された意味の最終表現が必要となる。図１９は、コード理解ツールの一部としての使用における、本システムを実装するメタコンパイラ１０の単純化した使用例の概略図である。図１６のように、メタコンパイラ１０は、候補コード部分１５に対して、ＡＳＴ記述コードを生成し、上で詳細に記載したように、本システムに従ってこのＡＳＴを実装するように動作可能である。コードの理解のために、メタコンパイラ１０を使用して、この目的のためにＡＳＴに動作することができるコード理解ツール５０が更に提供される。コード理解プロセスの後、生成された最終表現５５は、どのような形式もとることができるが、構文強調表示、文脈依存型ヘルプ、およびリフォーマット／ショートカットの提案が含まれうる。

【０１８７】

本発明のシステムの別の用途は、コードリファクタリングにある。コードリファクタリングは、コード理解の態様に加えて、プログラム表現を変換し、得られた変更を、比較的広いスケールで元の入力に自動的に再統合する手段を必要とする。図２０は、コードリファクタリングツールの一部としての使用における、本システムを実装するメタコンパイラ１０の単純化した使用例の概略図である。図１６のように、メタコンパイラ１０は、候補コード部分１５に対して、ＡＳＴ記述コードを生成し、上で詳細に記載したように、本システムに従ってこのＡＳＴを実装するように動作可能である。

【０１８８】

コードリファクタリングのために、柔軟なＡＳＴ表現と本システムによって提供される操作の利点とが利用される。リファクタリングツール６０は、メタコンパイラ１０で生成されるＡＳＴを使用して、ユーザインタフェース６５のために表現を提供するように動作可能である。図１９のようなコード理解機能が利用され、（図１６，１７に示すように）「インスタンス」ＡＳＴが操作されて、このようなコードリファクタリングが実行されうる。従って、このような操作後に、メタコンパイラ１０は、操作されたＡＳＴを、操作されたＡＳＴに相当する変換されたコード部分３０に変換するように動作可能である。正確なＡＳＴソース座標とＣ＋＋（ＢＬ）ソースコードジェネレータがこの機能の基礎を提供する。このように、元の候補コード部分１５がリファクタリングされる。

【０１８９】
次に、変換された出力が、従来のＣ＋＋コンパイラ４０によって処理され、オブジェクトコード４５が生成されうる。説明を簡単にするために、候補コード部分１５と変換されたコード部分３０とが、図２０では同じボックス内に示されている。
【０１９０】
本発明のシステムの別の用途は、ビルド最適化にある。ビルド最適化は、所定のファイルの集合に対して、（実行する作業と、このため、これに要する時間の点で）コンパイルまたはビルドのコストが大きく削減されるように、既存のコードをソースコードレベルでリファクタリング可能にする。これは、抽象プログラムの意味論ではなく、物理的なプロジェクト構造に着目したコードリファクタリング（図２０）の一種であるが、この２つの要素を、有用に組み合わせもてよい。
【０１９１】
図２１は、ビルド最適化ツールの一部としての使用における、本システムを実装するメタコンパイラ１０の単純化した使用例の概略図である。いうまでもなく、リファクタリングツール６０とユーザインタフェース６５が、ビルド最適化ツール７０に変わっている点を除き、図２１は図２０と同様である。ＡＳＴベースのソースファイル関係情報は、プロジェクトソースファイルを、極めて微細な粒度レベル（サブファイルレベル）で安全に再編成するための手段を提供する。
【０１９２】
本発明のシステムの別の用途は、コードインスツルメンテーションツールとしての用途である。コードインスツルメンテーションにより、既存のコードを、コードを変更または拡張しつつ自動的に生成可能となり、その際、入力ソースコード内への変換またはアノテーションの埋め込みの有無は問わない。インスツルメンテーションの例としては、デバッギング、パフォーマンスプロファイリング、アンチハッキング技術がある。この技術は、コード変換の要素と、ある程度はリファクタリングおよびソースフォーマッティングとに基づいている。本システムの柔軟なコード生成されたＡＳＴは、堅牢で自動化されたインスツルメンテーションのために可能なＡＳＴ検査、解析および変更を行う。
【０１９３】
例として、図２２は、インスツルメンテーションツールの一部しての使用における、本システムを実施するメタコンパイラ１０の単純化した使用例の模式図であり、この図において、ＢＬはＣ＋＋である。図２２から分かるように、ＢＬで記述された候補コード部分１５が、メタコンパイラ１０への入力として提供される。メタコンパイラ１０は、候補コード部分１５に対して「インスタンス」ＡＳＴ記述コードを生成し、上で詳細に記載したように、本システムに従ってこのＡＳＴを実装するように動作可能である。したがって、前述のように、入力メタ変換２０に従って、候補コード部分１５に対してＡＳＴを操作することが可能である。
【０１９４】
インスツルメンテーションシステム７５は図２２に示すように提供され、メタコンパイラ１０で生成されたＡＳＴと対話して、コードインスツルメンテーションプロセスの一環として、ＡＳＴを操作するように動作可能である。操作後に、メタコンパイラ１０は、操作されたＡＳＴを、（図１６に示すように）操作されたＡＳＴに相当する変換されたコード部分３０に変換するように動作可能である。次に、変換された出力が、従来のＣ＋＋コンパイラ４０によって処理され、オブジェクトコード４５が生成されうる。
【０１９５】
理解されるように、上記の使用例の多くが、何らかの形の「コード変換」に関連しており、したがって、このようなコード変換を、図２３を参照して更に詳しく考察する。
【０１９６】
図２３は、コード変換ツールの一部としての使用における、本システムを実装するメタコンパイラ１０の単純化した使用例の概略図である。図２３は、ある程度は図１７，１８を拡張したものであり、メタコンパイラ１０の内部コンポーネントに、参照のために符号

と名前を付している。したがって、図１７，１８を参照して上で説明した図２３の要素には、同じ名前を付している。

【０１９７】

必要な内部コンポーネントは多少用途に依存しているため、このようなコンポーネントの全てが各使用例に必要とは限らない。説明のために、コード変換の例は、図２３に示す全コンポーネントを利用するとする。

【０１９８】

ＢＬ（本例ではＣ＋＋）で表現され、場合によっては、ＴＬで表現されたメタプログラムインスタンス２０（この場合は、拡張言語構文および任意のスクリプト言語の組み合わせ）によって拡張された入力プログラム（候補コード部分）１５が、入力プログラムの忠実性を保持するＡＳＴ表現に変換される。これが、レキサー／パーサ要素８０によってハンドリングされる。レキサー／パーサ要素８０は、他のシステムフレームワークツールによって生成されるコードでもよい。レキサーとパーサは、機能に関して独立しているコンポーネントでもよいが、好ましくは、ＢＬのめの１つの統合された文法ファイル（第１のデータ構造記述を含む）からユニットとして一緒に生成される。次に、「インスタンス」ＡＳＴ表現（変換前）８５が、（上記の使用例で例示したように）多く異なるソースから入力を取ることができる変換エンジン９０によって変更されうる。この特定の場合、入力はＴＬで表現されたメタプログラムコードであり、ＡＳＴにおいても別個に表わされる。

【０１９９】

次に、変更されたＡＳＴ（変換後または操作後）９５が、従来のＢＬコンパイラ４０または他の適切な開発ツールに処理させるために、（この場合、Ｃ＋＋構文ジェネレータ要素１００によって）ＢＬ形式で送信されうる。ＡＳＴは、記憶のため、検査／視覚化または解析のために他の任意の形式で送信されてもよい。一般に、ＡＳＴは、メタプログラムのデバッギングのためにグラフ視覚化形式として送信される。

【０２００】

例えば図２０の例と類似する本システムの１つの可能な使用例として、特定のプロセッサ構成によって処理されるように記述されたコードを、異なるプロセッサ構成によって効率的に処理されるようにリファクタリングする場合がある。例えば、シングルプロセッサコアによって処理されるように記述されたコードを、例えば並列処理機能を利用するために、デュアルプロセッサコアで効率的に処理するために適合させることができる。本システムは、このようなリファクタリングを実行するための効率的な方法を提供する。

【０２０１】

上記の態様のいずれにおいても、各種特徴を、ハードウェアで実装しても、１つ以上のプロセッサで実行されるソフトウェアモジュールとして実装してもよい。ある態様の特徴を、別の態様に適用することができる。

【０２０２】

また、本発明は、ここに記載する方法を実装するためのコンピュータプログラムまたはコンピュータプログラム製品と、ここに記載する方法を実装するためのプログラムが記憶されているコンピュータ可読記憶媒体を提供する。本発明を実施するコンピュータプログラムは、コンピュータ可読媒体に記憶されても、例えば、インターネットのウェブサイトから提供されるダウンロード可能なデータ信号などの信号の形式をとっても、他のどのような形式をとってもよい。

【０２０３】

本明細書に記載したように、本明細書に記載の方法はコンピュータまたはコンピュータのネットワークによって実行されうる。当業者が当然理解できるように、コンピュータまたはコンピュータのネットワークは、１つ以上のプロセッサ、入力データ、中間データ、および出力データを記憶するためのメモリおよび／またはストレージと、入出力部品（例えばインタフェースカード、有線または無線通信部品、キーボードおよび／またはディスプレイデバイス）とを有しうる。１つ以上のプロセッサは、処理（例えば、本明細書に記載の方法の処理）を実行し、メモリまたはストレージへのデータの記憶および取得を命令

し、ディスプレイへの視覚表示の生成を命令するように動作可能とするために、一般に命令によって構成される。

【０２０４】

本発明を、下記に記載する付記（statement）によって記載する。この付記は、本願の特許請求の範囲には含まれない。本願の特許請求の範囲は第３７ページから始まる。

（付記１）

構文規則の組を満たす複数の構文的要素を有するコンピュータプログラミング言語の少なくとも一部を表わす第１のデータ構造の実装を生成するための実装ツールであって、前記第１のデータ構造は、複数のリンクされたノードを有し、前記ノードはルートノードと、前記ルートノードに直接リンクされた複数の第１階層ノードと、１つ以上の他の前記ノードを介して前記ルートノードに間接的にリンクされた複数の下層ノードと、を有し、前記ノードは、前記言語のこのような構文的要素と、このような構文規則を表現する前記ノードの間のリンクとを表わし、前記ツールは、

前記第１のデータ構造またはその記述を受け取るように動作可能な第１のデータ構造入力手段と、

前記第１のデータ構造の実装を生成するように動作可能な処理手段と、を有し、前記実装は、

前記第１のデータ構造に対応する第２のデータ構造またはその記述であって、前記第１のデータ構造の前記ノードに対応するノードを有し、そのルートノード以外のノードが、前記第２のデータ構造の前記ルートノードに直接リンクされている第２のデータ構造と、

前記第１のデータ構造によって表現された前記構文規則への準拠を確立するために、前記実装を利用するその後の処理操作中に、前記第２のデータ構造のノードに関連して強制される前記第１のデータ構造の前記構文規則を定義する実装規則と、を有する実装ツール。

【０２０５】

（付記２）

前記第１のデータ構造は、少なくとも部分的に抽象構文ツリーなどの不均一なツリー構造である付記１に記載の実装ツール。

【０２０６】

（付記３）

前記第１のデータ構造は、少なくとも部分的に有向非巡回グラフ構造である付記１または２に記載の実装ツール。

【０２０７】

（付記４）

前記言語は多数のトークンから構成され、前記言語の各前記構文的要素は、前記トークンの群あるいは前記トークンの所定の組合せを表わす上記付記のいずれか１項に記載の実装ツール。

【０２０８】

（付記５）

前記第１のデータ構造の前記ノード間の前記リンクは、該ノードの間の実装およびインタフェースではなく代替可能性の継承の経路を表わす上記付記のいずれか１項に記載の実装ツール。

【０２０９】

（付記６）

前記第２のデータ構造の前記ノード間の前記リンクは、該ノードの間の代替可能性、実装およびインタフェースの継承の経路を表わす上記付記のいずれか１項に記載の実装ツール。

【０２１０】

（付記７）

前記第１のデータ構造によって表わされる代替可能性関係への準拠を確立するために、

前記実装規則は、前記実装を利用するその後の処理操作中に前記第２のデータ構造のノードに関連して強制される代替可能性の規則を定義する上記付記のいずれか１項に記載の実装ツール。

【０２１１】

（付記８）

前記第１のデータ構造は、言語拡張を表わす更に別のリンクされたノードを有し、前記言語拡張は前記コンピュータプログラミング言語に対する拡張である上記付記のいずれか１項に記載の実装ツール。

【０２１２】

（付記９）

前記第１のデータ構造の前記更に別のノードは、前記言語拡張のそれぞれの構文的要素を表現する付記８に記載の実装ツール。

（付記１０）

【０２１３】

前記受け取った第１のデータ構造の前記更に別のノードの１つ以上は、前記第１のデータ構造の他の前記ノードの一部もしくは全部に対して代替可能として定義される付記８または９に記載の実装ツール。

【０２１４】

（付記１１）

前記実装規則は、前記第１のデータ構造によって表わされる前記第１のデータ構造の前記１つ以上の更に別のノードの前記代替可能性関係への準拠を確立するために、前記実装を利用するその後の処理操作中に、前記受け取った第１のデータ構造の前記１つ以上の更に別のノードに対応する前記第２のデータ構造のノードに関連して強制される代替可能性の規則を定義する付記８～１０のいずれか１項に記載の実装ツール。

【０２１５】

（付記１２）

前記言語拡張は、前記コンピュータプログラミング言語以外の言語の一部または全体である付記８～１１のいずれか１項に記載の実装ツール。

【０２１６】

（付記１３）

構文規則の組を満たす複数の構文的要素を有するコンピュータプログラミング言語の少なくとも一部を表わす第１のデータ構造の実装を生成するための方法であって、前記第１のデータ構造は、複数のリンクされたノードを有し、前記ノードはルートノードと、前記ルートノードに直接リンクされた複数の第１階層ノードと、１つ以上の他の前記ノードを介して前記ルートノードに間接的にリンクされた複数の下層ノードと、を有し、前記ノードは、前記言語のこのような構文的要素と、このような構文規則を表現する前記ノードの間のリンクとを表わし、前記方法は、

前記第１のデータ構造またはその記述を受け取るステップと、

前記第１のデータ構造の実装を生成するステップと、を有し、前記実装は、

前記第１のデータ構造に対応する第２のデータ構造またはその記述であって、前記第１のデータ構造の前記ノードに対応するノードを有し、そのルートノード以外のノードが、前記第２のデータ構造の前記ルートノードに直接リンクされている第２のデータ構造と、

前記第１のデータ構造によって表現された前記構文規則への準拠を確立するために、前記実装を利用するその後の処理操作中に、前記第２のデータ構造のノードに関連して強制される前記第１のデータ構造の前記構文規則を定義する実装規則と、を有する実装方法。

【０２１７】

（付記１４）

コンピューティングデバイスで実行されたときに、前記コンピューティングデバイスに付記１３に記載の方法を実行させる実装コンピュータプログラム。

【０２１８】

（付記１５）
付記１～１２のいずれか１項に記載の実装ツールによって生成される前記実装の前記第２のデータ構造のインスタンスに動作するためのインスタンスハンドリングツールであって、
前記実装を記憶する手段と、
前記実装に依存して前記候補インスタンスに動作するための手段と、を有し、前記候補インスタンスは、前記第２のデータ構造のノードに対応するインスタンスノードを含むインスタンスハンドリングツール。
【０２１９】
（付記１６）
前記第２のデータ構造の前記候補インスタンスを、そのインスタンスノードおよびその間のリンクが明示的に表現されていない入力形式で受け取るための手段と、
前記受け取った候補インスタンスを、前記インスタンスノードおよびその間のリンクが明示的に表現されている抽象化形式に変換するための変換手段と、を更に有する付記１５に記載のインスタンスハンドリングツール。
【０２２０】
（付記１７）
前記候補インスタンスは、前記コンピュータプログラミング言語内で表現されたコード部分であり、
前記入力形式は、前記コード部分のテキスト版であり、
前記抽象化形式は、前記コード部分の抽象構文ツリーまたはグラフ版である付記１６に記載のインスタンスハンドリングツール。
【０２２１】
（付記１８）
前記候補インスタンスの視覚的表現を生成するように動作可能な可視化手段を更に有する付記１５～１７のいずれか１項に記載のインスタンスハンドリングツール。
【０２２２】
（付記１９）
少なくとも付記１６に付加して解釈される場合に、前記可視化手段は、前記候補インスタンスの視覚的表現を前記抽象化形式で生成するように動作可能である付記１８に記載のインスタンスハンドリングツール。
【０２２３】
（付記２０）
前記実装に依存して前記候補インスタンスを操作するように動作可能な操作手段を更に有する付記１５～１９のいずれか１項に記載のインスタンスハンドリングツール。
【０２２４】
（付記２１）
少なくとも付記１６に付加して解釈される場合に、前記操作手段は、前記候補インスタンスを前記抽象化形式で操作するように動作可能である付記２０に記載のインスタンスハンドリングツール。
【０２２５】
（付記２２）
前記操作手段は、前記実装と照らし合わせてこのような操作を検証し、前記第２のデータ構造および／または前記実装規則に違反する操作を禁止するように動作可能である付記２０または２１に記載のインスタンスハンドリングツール。
【０２２６】
（付記２３）
前記操作手段は、前記第２のデータ構造および前記実装規則を遵守する操作を許可するように動作可能である付記２２に記載のインスタンスハンドリングツール。
【０２２７】

（付記２４）
このような操作は、前記候補インスタンスを増やすおよび／または前記候補インスタンスを減らすことを含む付記２０～２３のいずれか１項に記載のインスタンスハンドリングツール。
【０２２８】
（付記２５）
このような操作は、前記候補インスタンスに新しいインスタンスノードを追加することを含む付記２０～２４のいずれか１項に記載のインスタンスハンドリングツール。
【０２２９】
（付記２６）
このような操作は、前記候補インスタンスからインスタンスノードを削除することを含む付記２４または２５に記載のインスタンスハンドリングツール。
【０２３０】
（付記２７）
このような操作は、前記候補インスタンスの特定のインスタンスノードをアノテートすることを含む付記２４～２６のいずれか１項に記載のインスタンスハンドリングツール。
【０２３１】
（付記２８）
このような操作は、前記候補インスタンスの一部または全体に所定のプロセスを実行することを含む付記２０～２７のいずれか１項に記載のインスタンスハンドリングツール。
【０２３２】
（付記２９）
前記所定のプロセスは、前記インスタンスハンドリングツールによってアクセス可能な、コンピュータプログラムなどのアクションの組内に定義される付記２８に記載のインスタンスハンドリングツール。
【０２３３】
（付記３０）
前記所定のプロセスは、所定の目的のために前記候補インスタンスを最適化するように動作可能な最適化プロセスである付記２８または２９に記載のインスタンスハンドリングツール。
【０２３４】
（付記３１）
前記操作手段は、前記候補インスタンスの前記インスタンスノードに依存してこのような操作を実行するように動作可能である付記２０～３０のいずれか１項に記載のインスタンスハンドリングツール。
【０２３５】
（付記３２）
前記操作手段は、インスタンスノードの特定の型を識別し、前記識別されたインスタンスノードに依存してこのような操作を実行するように動作可能である付記３１に記載のインスタンスハンドリングツール。
【０２３６】
（付記３３）
前記実装および前記第２のデータ構造は、付記８～１２のいずれか１項に記載の実装ツールによって生成される前記実装および前記第２のデータ構造であり、
インスタンスノードの前記特定の型は、前記言語拡張の前記更に別のノードのインスタンスノードである付記３２に記載のインスタンスハンドリングツール。
【０２３７】
（付記３４）
前記実装および前記第２のデータ構造は、付記８～１２のいずれか１項に記載の実装ツールによって生成される前記実装および前記第２のデータ構造であり、

前記候補インスタンスは、前記言語拡張に由来する部分を含む付記１５～３３のいずれか１項に記載のインスタンスハンドリングツール。

【０２３８】

（付記３５）

少なくとも付記１６に付加して解釈される場合に、前記変換手段は第１の変換手段であり、前記インスタンスハンドリングツールは、前記抽象化形式の前記候補インスタンスをその対応する入力形式に変換するように動作可能な第２の変換手段を更に有する付記１５～３４のいずれか１項に記載のインスタンスハンドリングツール。

【０２３９】

（付記３６）

少なくとも付記２０に付加して解釈される場合に、前記第２の変換手段は、このような操作が前記候補インスタンスに実行される前にあるいはその後に、このような変換を実行するように動作可能である付記３５に記載のインスタンスハンドリングツール。

【０２４０】

（付記３７）

少なくとも付記２０に付加して解釈される場合に、

前記候補インスタンスは、前記コンピュータプログラミング言語内で表現されたコード部分であり、

前記インスタンスハンドリングツールは、このような操作の前か後に、前記候補インスタンスをオブジェクトコードとして出力するように動作可能な手段を有する付記１５～３６のいずれか１項に記載のインスタンスハンドリングツール。

【０２４１】

（付記３８）

パーサまたはコンパイラである付記３７に記載のインスタンスハンドリングツール。

【０２４２】

（付記３９）

付記１～１２のいずれかの１項に記載の実装ツールによって生成される前記実装の前記第２のデータ構造のインスタンスに動作するためのインスタンスハンドリング方法であって、前記方法は、前記実装に依存して前記候補インスタンスに動作するステップを有し、前記候補インスタンスは、前記第２のデータ構造のノードに対応するインスタンスノードを含む方法。

【０２４３】

（付記４０）

コンピューティングデバイスで実行されたときに、前記コンピューティングデバイスに付記３９に記載の方法を実行させるインスタンスハンドリングコンピュータプログラム。

【０２４４】

（付記４１）

コンピュータプログラミング言語を拡張する方法であって、

前記言語の少なくとも一部を表わす第１のデータ構造またはその記述を取得するステップと、

言語拡張を表わす更に別のリンクされたノードを含むように、前記第１のデータ構造またはその前記記述を適応させるステップと、

前記適応された第１のデータ構造の実装を生成するために、付記８～１２のいずれか１項に記載の実装ツールを使用するステップと、を有する方法。

【０２４５】

（付記４２）

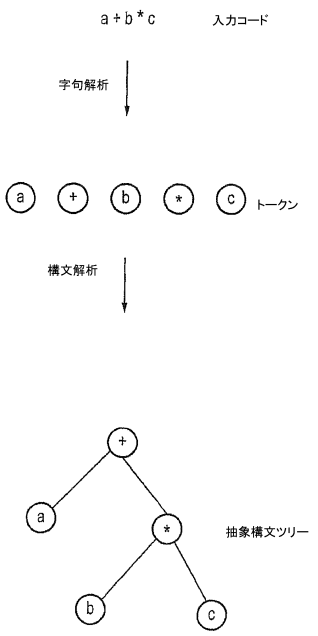
付記１～１２のいずれか１項に記載の実装ツールおよび／または付記１５～３８のいずれか１項に記載のインスタンスハンドリングツールを有するパーサまたはコンパイラ。

【０２４６】

（付記４３）

コンピュータプログラムを生成あるいは適応させる方法であって、
前記コンピュータプログラミング言語の少なくとも前記一部で表現された候補コンピュータプログラムを、前記候補インスタンスとして、付記１５～３８のいずれか１項に記載のインスタンスハンドリングツールに入力するステップと、
前記候補インスタンスに動作するために前記インスタンスハンドリングツールを使用するステップと、
このような操作から得られるコンピュータプログラムを出力するために前記インスタンスハンドリングツールを使用するステップと、を有し、前記出力されるコンピュータプログラムは、このように生成または適応されたコンピュータプログラムである方法。
【０２４７】
（付記４４）
付記１～１２のいずれか１項に記載の実装ツールおよび／または付記１５～３８のいずれか１項に記載のインスタンスハンドリングツールとして機能するように構成されたコンピュータまたはコンピュータのネットワーク。
【０２４８】
（付記４５）
付記１～１２のいずれか１項に記載の実装ツールおよび／または付記１５～３８のいずれか１項に記載のインスタンスハンドリングツールを有するメタプログラミングツール。

【図１】



【図２】

```
Statement :=
     "if" Expression "then" Statement "else" Statement
   | Expression

Declaration :=
    TypeName Identifier "=" Initializer


Initializer :=
     Identifier
   | Expression

Expression :=
     Expression "+" Expression
   | Expression "*" Expression
```

FIG. 2

【図３】

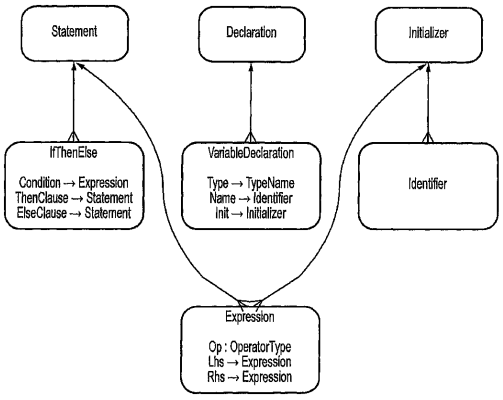


FIG. 3

【図４】

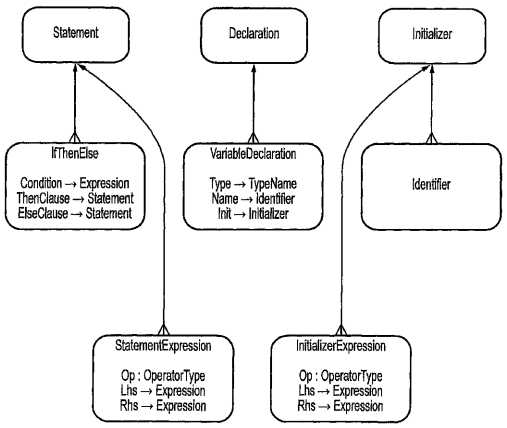


FIG. 4

【図５】

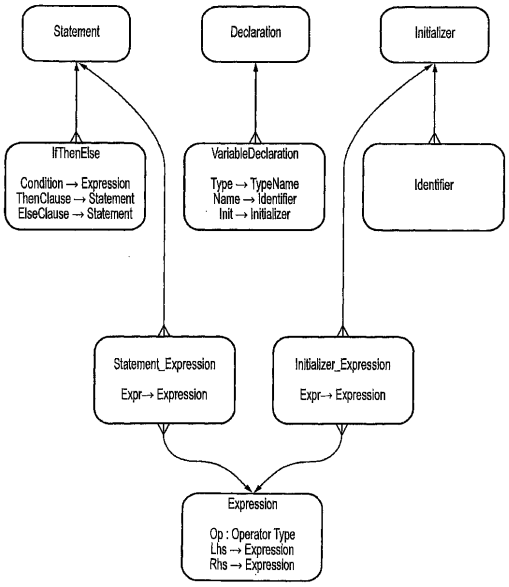


FIG. 5

【図６】

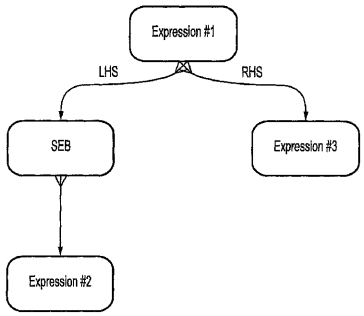


FIG. 6

【図７】

```
root Node;

abstract Statement < Node;
abstract Declaration < Node;
abstract Initializer < Node;

node IfThenElse < Statement
{
  @condition >> Expression;
  @thenClause >> Statement;
  @elseClause >> Statement;
}

node VariableDeclaration < Declaration
{
  @type >> TypeName;
  @name >> Identifier;
  @init >> Initializer;
}

node Identifier < Initializer;

node Expression < Statement , Initializer
{
  @op : OperatorType;
  @lhs >> Expression;
  @rhs >> Expression;
}
```

FIG. 7

【図８】

```
root Node;

abstract Statement < Node;
abstract Declaration < Node;
abstract Initializer < Node;

node IfThenElse < Statement
{
  @condition >> Expression;
  @thenClause >> Statemént. Expression;
  @elseClause >> Statement. Expression;
}

node VariableDeclaration < Declaration
{
  @type >> TypeName;
  @name >> Identifier;
  @init >> Initializer. Expression;
}

node Identifier < Node;

node Expression < Node
{
  @op : OperatorType;
  @lhs >> Expression;
  @rhs >> Expression;
}
```

FIG. 8

【図９】

```
node wildcard < *
{
}
```

FIG. 9

【図１０】

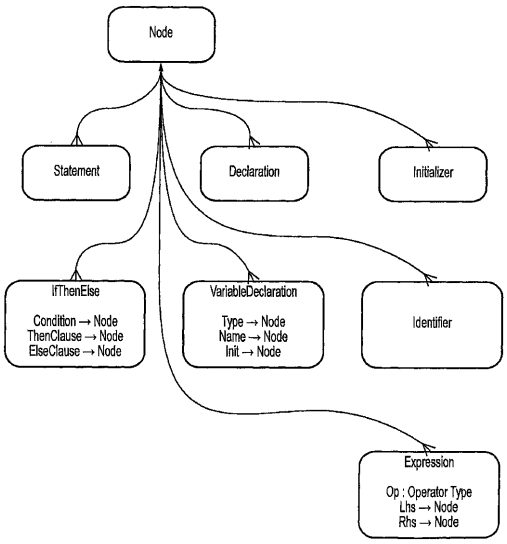


FIG. 10

【図１１】

```
OperatorType op();
void op(OperatorType val);

NodePtr lhs();
void lhs(NodePtr n);

NodePtr rhs();
void rhs(NodePtr n);
```

FIG. 11

【図１２】

```
NodePtr thenClause();
Void thenClause(StatementPtr n);
Void thenClause(IfThenElsePtr n);
Void thenClause(ExpressionPtr n);
```

FIG. 12

【図１３】

```
for(TreeIterator iter = startNode;
iter.more();iter.next())
{
  NodePtr node = iter.node();

  //Use 'node' to operate on the current node.
}
```

FIG. 13

【図１４】

```
void IfThenElse::visit(Callback callback)
{
  callback(this);

  if(condition) visit(condition);
  if(thenClause) visit(thenClause);
  if(elseClause) visit(elseClause);
}
```

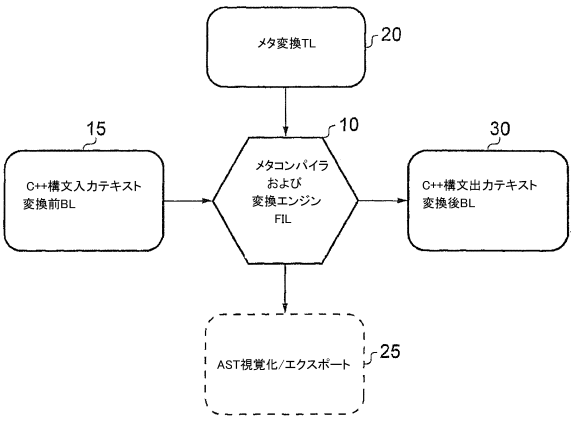
FIG. 14

【図１５】

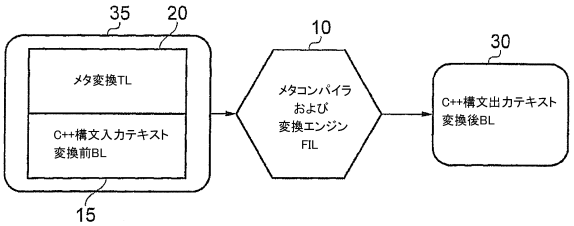
```
NodePtr * IfThenElse::enumerate(int slot)
{
  switch(slot)
  {
  case 0:
    return &condition;
  case 1:
    return &thenClause;
  case 2:
    return &elseClause;
  default:
    return NULL;
  }
}
```

FIG. 15

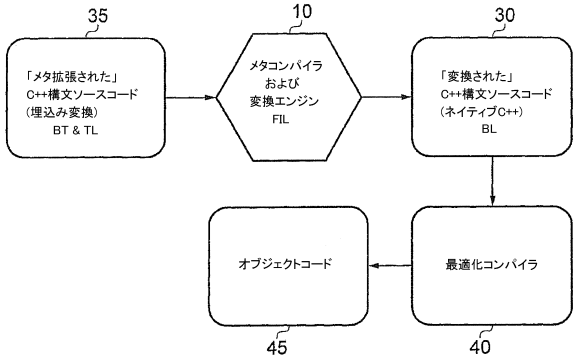
【図１６】



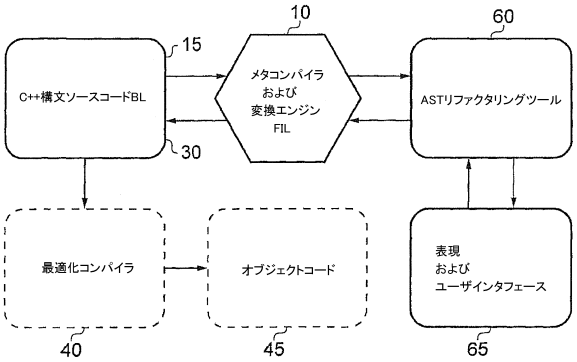
【図１７】



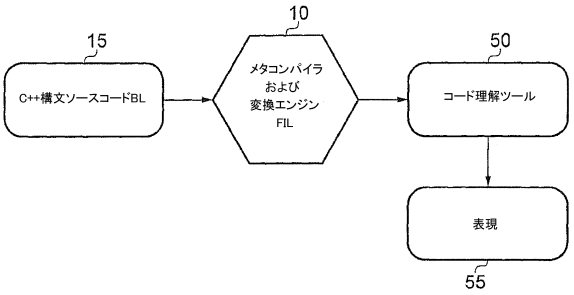
【図１８】



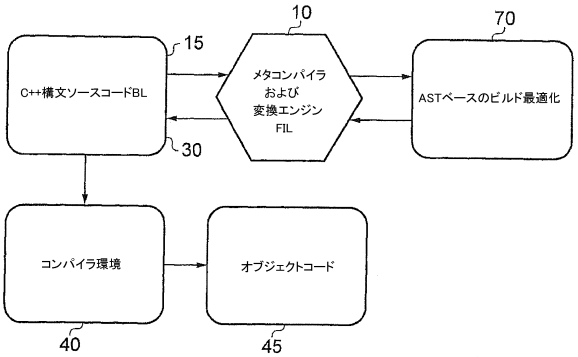
【図２０】



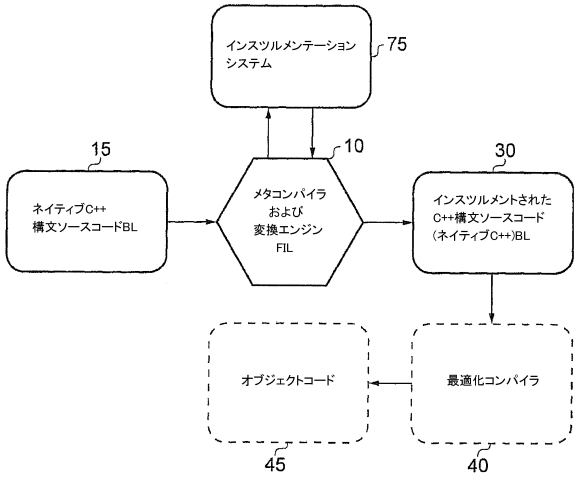
【図１９】



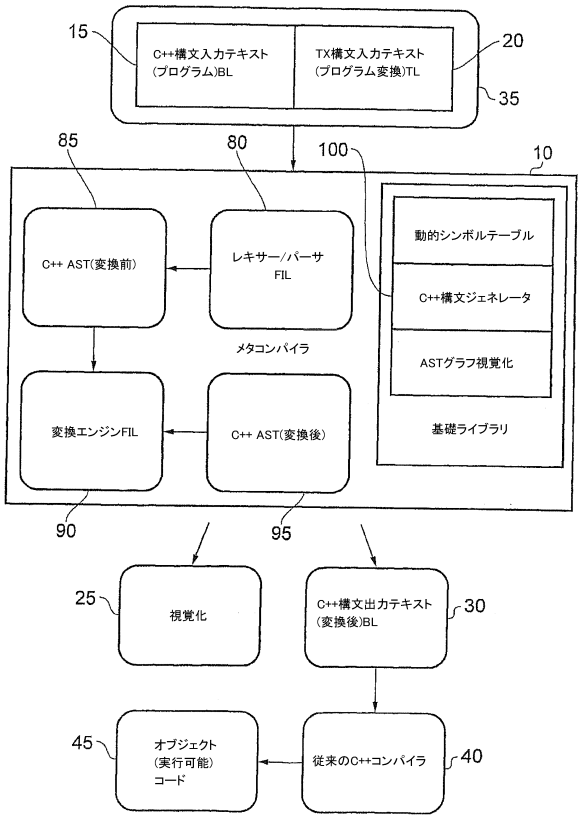
【図２１】



【図２２】



【図２３】



フロントページの続き

(72)発明者　スチュワート，ニール
イギリス国，グラスゴー　ジー１　２アールエヌ，１４１　ウエスト　ナイル　ストリート，カレドニアン　スイート，セント　アンドリュー　ハウス，スラム　ゲームス　リミテッド

審査官　坂庭　剛史

(56)参考文献　特開平０５－１７３７７２（ＪＰ，Ａ）
小藤哲彦、河野健二、竹内郁雄，¬＜＞＜∪∪（ｎｏｔａｖａＣＣ）：オブジェクト指向抽象構文木を生成するコンパイラ・コンパイラ，情報処理学会論文誌，日本，社団法人情報処理学会，２００３年１０月１５日，第４４巻，Ｎｏ．ＳＩＧ１３（ＰＲＯ１８），ｐｐ．８４～９９
Rhys Weatherley, Treecc: An Aspect-Oriented Approach to Writing Compilers, Free Software Magazine［ONLINE］，２００２年，ＵＲＬ，http://www.gnu.org/projects/dotgnu/treecc_essay.html
中田育男、渡邊 坦，連載 ２１世紀のコンパイラ道しるべ ２：ＨＩＲの説明と簡単な言語のフロントエンド，情報処理，日本，社団法人情報処理学会，２００６年　5月１５日，第４７巻，第5号，ｐｐ．５２６～５３９

(58)調査した分野(Int.Cl.，ＤＢ名)
Ｇ０６Ｆ　　９／４５