Vol.2024-SE-216 No.23
2024/3/5

情報流解析におけるIDEを前提とした機密度アノテーション

桑原 寛明[1]

概要：本稿では，IDE の利用を前提として，情報流解析における機密度を記述および管理する手法を提案する．型検査に基づく情報流解析では，プログラム中の変数や関数の返り値の型として機密度を指定する必要があるが，実際に利用されているプログラミング言語にはそのための構文が存在しない．構文を拡張すれば対応できるが，その場合は既存の言語処理系や開発環境をそのまま利用することが難しくなる．現在のソフトウェア開発では IDE を利用してプログラムを作成することが一般的であるため，構文を拡張することなく機密度を扱う手法を，IDE の機能を活用して実現する．本稿では，機密度を Inlay Hints によって表示するプロトタイプツールを VSCode の拡張機能として実装し，機密度の指定，表示，保存の方法を検討する．

1. はじめに

プログラムが処理する機密情報がプログラムの外部に流出しないことを静的に検査する手法として，型検査に基づく情報流解析が提案されている [1], [2], [3], [4]．型検査に基づく情報流解析では，データの機密度を型として利用し，型付け可能なプログラムが非干渉性を満たすように型システムを構築する．非干渉性は，機密度の低いデータが機密度の高いデータに直接および間接的に依存しないことを表し，機密データ自体に加え機密データを推測できる情報も流出しないという意味でよい性質である．

型検査に基づく情報流解析では，プログラム中の変数や関数の返り値の型として機密度を指定する必要がある．そのため，これまでの情報流解析のための型システムの大半は，機密度が構文要素として含まれているプログラミング言語を対象に構築されている．しかし，実際に活用されているプログラミング言語は構文に機密度を含んでいない．そのような言語で開発されたプログラムに対して情報流解析を適用するためには，変数や関数の返り値の機密度を指定するための仕組みを用意する必要がある．

この問題に対するひとつのアプローチとして，情報流解析のための Java アノテーションが提案されている [5], [6]．Java には，一部の構文要素に対して追加情報を注釈するアノテーションと呼ばれる仕組みが言語仕様として存在する．情報流解析に必要な機密度を注釈するためのアノテーションを定義し，対象プログラムのコンパイル時にアノテーションに基づいて情報流解析を行う検査器を実装することで，Java プログラムについては構文を拡張することなく情報流解析が実現されている．一方，Java アノテーションのように特定の構文要素に対して追加情報を記述する仕組みを持たないプログラミング言語も一般的である．このような言語のソースコードにおいて変数などの機密度を記述する手法については課題として残されている．機密度を記述できるように構文を拡張すれば実現は可能であるが，言語処理系や IDE（統合開発環境）に対する影響が大きいため，構文の拡張はできる限り避けたい．

本稿では，IDE の利用を前提として，プログラミング言語仕様を拡張することなく機密度を付与し情報流解析を実現する手法を提案する．現在のソフトウェア開発では IDE を利用してプログラムを作成することが一般的であり，かつ情報流解析の検査器は IDE の拡張機能（あるいはプラグイン）として提供されるため，IDE の利用を前提とすることは妥当であると考えられる．本稿では，プロトタイプツールを VSCode の拡張機能として開発し，機密度の付与，表示，保存の方法を検討する．

2. 型検査に基づく情報流解析の概略

情報流解析ではデータの機密度に着目し，機密度の低いデータが機密度の高いデータに直接あるいは間接的に依存する時に不正な情報流が存在するとみなす．型検査に基づく情報流解析では，データの機密度を型として利用し，型付け可能なプログラムが非干渉性を満たすように型システムを構築する．検査器は型システムに基づいて型検査することで，不正な情報流を検出できる．

以下では，[7] に従って型検査に基づく情報流解析の概略

[1] 南山大学 理工学部
Faculty of Science and Technology, Nanzan University

$$\eta ::= L \mid H$$
$$P ::= \overline{F}$$
$$F ::= \eta\ f(\overline{\eta\ x})\ B$$
$$B ::= \{\overline{\eta\ x}\ \overline{S;}\}$$
$$S ::= x = e \mid x = f(\overline{e}) \mid \text{if } (e)\ B \text{ else } B$$
$$e ::= x \mid \text{true} \mid \text{false} \mid e == e$$

図 1 対象言語の構文

$$\frac{\vdash F_i \quad i \in \{1, \ldots, n\}}{\vdash F_1\ \ldots\ F_n}\ [\text{PROGRAM}]$$

$$\frac{\overline{x : \eta_x}, result : \eta_r \vdash B : \eta_B}{\vdash \eta_r\ f(\overline{\eta_x\ x})\ B}\ [\text{FDEC}]$$

$$\frac{\Delta, \overline{x : \eta_x} \vdash S_i : \eta_i \quad \eta \sqsubseteq \eta_i \quad i \in \{1, \ldots, n\}}{\Delta \vdash \{\overline{\eta_x\ x;}\ S_1; \ldots S_n;\} : \eta}\ [\text{BLOCK}]$$

$$\frac{\Delta \vdash e : \eta_e \quad \eta_x = \Delta(x) \quad \eta_e \sqsubseteq \eta_x \quad \eta \sqsubseteq \eta_x}{\Delta \vdash x = e : \eta}\ [\text{ASSIGN}]$$

$$\frac{\begin{array}{l}\Delta \vdash e_i : \eta_{e_i} \quad \eta_{e_i} \sqsubseteq \eta_i \quad i \in \{1, \ldots, n\} \\ y_1 : \eta_1, \ldots, y_n : \eta_n \to \eta_r = ftype(f) \\ \eta_x = \Delta(x) \quad \eta_r \sqsubseteq \eta_x \quad \eta \sqsubseteq \eta_x\end{array}}{\Delta \vdash x = f(e_1, \ldots, e_n) : \eta}\ [\text{CALL}]$$

$$\frac{\begin{array}{l}\Delta \vdash e : \eta_e \quad \Delta \vdash B_t : \eta_t \quad \Delta \vdash B_f : \eta_f \\ \eta_e \sqsubseteq \eta \quad \eta \sqsubseteq \eta_t \quad \eta \sqsubseteq \eta_f\end{array}}{\Delta \vdash \text{if } (e)\ B_t \text{ else } B_f : \eta}\ [\text{IF}]$$

$$\frac{}{\Delta \vdash x : \Delta(x)}\ [\text{VAR}] \quad \frac{c \in \{\text{true,false}\}}{\Delta \vdash c : L}\ [\text{CONST}]$$

$$\frac{\Delta \vdash e_1 : \eta_1 \quad \Delta \vdash e_2 : \eta_2 \quad \eta_1 \sqsubseteq \eta \quad \eta_2 \sqsubseteq \eta}{\Delta \vdash e_1 == e_2 : \eta}\ [\text{COMP}]$$

図 2 型付け規則

を述べる．なお，[7] では簡単な手続き型言語が対象であるが，オブジェクト指向言語が対象であっても本質的には同様である [2], [5], [6].

対象言語の構文を図 1 に示す．η はデータの機密度を表し，束 $(\mathcal{H}, \sqsubseteq)$ の元であるとする．ここでは簡単のために $L \sqsubseteq H$ かつ $H \not\sqsubseteq L$ を満たす機密度束 $(\{L, H\}, \sqsubseteq)$ を仮定する．プログラム P は関数定義の並びである．$\overline{A}$ は長さ 0 以上の有限リストを表す．F は関数定義であり，f が関数名を表す．B はブロック，S は文，e は式である．ブロックの先頭でローカル変数を宣言できる．関数の返り値は予約変数 *result* への代入によって設定する．変数や関数の返り値の型として機密度を記述する．整数型などのデータ型の記述はないが，整合していることを前提とする．

型付け規則を図 2 に示す．PROGRAM 規則がプログラム全体，FDEC 規則が関数定義，BLOCK 規則がブロック，ASSIGN 規則，CALL 規則，IF 規則が文，残りが式の型付け規則である．図中の *result* は関数の返り値を表す変数，*ftype* は関数のシグネチャを取得する関数である．プログラムを構成するすべての関数定義が型付け可能であればプログラムは型付け可能である．引数と返り値に関する型環境の下で関数本体が型付け可能であれば関数定義は型付け可能である．プログラムと関数定義は型付けできるか否かが重要であり，プログラムと関数定義の具体的な型は定義しない．ブロックあるいは文の型判定式 $\Delta \vdash S : \eta$ は，型環境 Δ のもとで S の実行によって変更される変数の機密度が η 以上であることを表す．式の型判定式 $\Delta \vdash e : \eta$ は，型環境 Δ のもとで式 e の機密度が η 以下であることを表す．本稿では詳細を省略するが，対象言語の意味論を定義し，型システムが非干渉性に対して健全であることを証明できる．

```
1 H h;
2 L l;
3 l = h;
4 if (h) { l = 1; } else { l = 0; }
```

図 3 プログラム例

例えば，変数 h は秘密のデータが格納されるので機密度を H，変数 l は外部に出力されるデータが格納されるので機密度を L とする．この時，図 3 の 3 行目の代入文は，秘密のデータを公開用の変数に代入しているため拒絶したい．図 2 の ASSIGN 規則によると，右辺 h の機密度は左辺 l の機密度以下でなければならないが，$H \not\sqsubseteq L$ であるため成り立たず，この代入文は拒絶される．

同様に，図 3 の 4 行目の if 文は，実行後の l の値から h の値を予測できるため拒絶したい．IF 規則によると，then ブロックの機密度と else ブロックの機密度はともに条件式 h の機密度以上でなければならない．ここで，ブロックの機密度はそのブロックの実行中に値が代入される変数の機密度の交わりであり，この例では then ブロックも else ブロックも機密度は l の機密度 L である．しかし，$H \not\sqsubseteq L$ であるため，この if 文は拒絶される．

3. 機密度アノテーション

型検査に基づく情報流解析を適用するためには，変数や関数の返り値の機密度が指定されている必要がある．例えば，図 1 の言語では，変数宣言中の変数に対する型として機密度を記述することで変数の機密度を指定する．このように特定の構文要素に対する機密度の記述を機密度アノテーションと呼ぶ．

図 1 の言語では，機密度アノテーションは構文の一部に含まれているためソースコード中に直接記述され，ソースコードの一部として表示される．しかし，実際に活用されているプログラミング言語にはそのような構文が存在しな

いため，機密度を注釈する手段と，注釈された機密度を表示する手段を用意する必要がある．Java については，Java アノテーションによる機密度アノテーション [5] を利用することでソースコードの一部として機密度を注釈できるが，Java アノテーションに相当する仕組みを持たないプログラミング言語も多い．

機密度アノテーションは以下を満たすことが望ましい．

機密度の指定対象を一意に特定できる　機密度を注釈する際にはその対象の構文要素（例えば，どの変数宣言中の変数であるか）が，機密度を表示する際にはその機密度が指定された構文要素が一意に定まる必要がある．図 1 のような言語であれば，対象の構文要素は構文規則から一意に定まる．しかし，機密度をソースコードの外部から与える場合は構文規則のようなルールが存在しないため，注釈する際にも表示する際にも対象の構文要素が特定されるようにする必要がある．

ソースコードと一体化している　現在のソフトウェア開発では git などのバージョン管理システムを利用することが一般的になっている．機密度は，構文に含まれていないという意味ではソースコードの一部ではない．しかし，変数や関数の返り値の型の一部であるという意味ではソースコードの一部とみなすべきであり，共同開発時における機密度の共有等を実現するためにも，ソースコードと一緒にバージョン管理の対象とすることが望ましい．加えて，バージョン管理のミスを防ぐために開発者が機密度とソースコードを一体の存在として認識できることが望ましい．

これらをおおよそ満たす手段としてコメントの利用が挙げられる．例えば，変数宣言中の変数など機密度を指定したい構文要素の付近に，指定したい機密度をコメントとして記述する方法である．コメントはソースコード中に埋め込まれるため，自動的にソースコードと一体化する．コメントはソースコード中の任意の位置に記述できるが，対象の構文要素の直前の行に記述するなど，事前に定められた対象との相対的な位置関係を守るように記述すれば，機密度の指定対象を一意に特定することも可能である．ただし，構文規則ではないため記述位置を強制することは不可能であり，機密度と対象が一意に対応付けられることは保証されない．

本研究では，ソフトウェア開発における IDE の利用が一般的になっている状況を踏まえ，機密度の注釈をソースコードのコメントによるのではない方法で，IDE による支援を得て実現する方法を探る．

4. 統合開発環境

4.1 ユーザインタフェース

IDE（統合開発環境）は，ソフトウェア開発において利用される各種ツールが統合された開発環境であり，特にソースコードの作成や修正，テストの実行に活用される．IDE の主な機能には，コーディング支援，ビルド，デバッグ支援，構成管理，バージョン管理，チーム開発支援などがある．画面構成としては，ソースコードを編集するためのエディタが中央に据えられ，プロジェクト配下のファイル，ソースコードのエラー箇所，ビルド結果などを表示する複数のビューがエディタの周囲に配置されていることが多い．

図 4　CodeLens の例

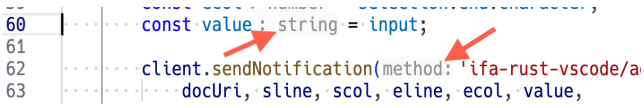

図 5　Inlay Hints の例

エディタには編集中のソースコードに加え，そのソースコードに関連する何らかの情報が表示されることが多い[*1]．行番号や，構文エラーなどビルドに失敗する原因の行に対するマーク，あるいは原因箇所に対する下線などがその一例である．さらに，最近の IDE の中には，エディタに表示されたソースコードの行間や文字間に追加情報を表示する CodeLens や Inlay Hints と呼ばれる機能を提供するものもある．これらの機能により，ソースコードに記述されていない情報を IDE 上でソースコードと混在させて開発者に提示できる環境が整えられている．

CodeLens を利用すると追加情報や何らかの動作を起動するリンクをソースコードの行間や行末に表示できる．例えば，メソッド定義の直前の行間にそのメソッドの参照箇所の数と各箇所へジャンプするリンクを表示する，カーソル位置の行末にその行の git blame 情報を表示する，といった用途に利用されている．図 4 は TypeScript プログラムに対して VSCode が提供する CodeLens の例であり，クラスや変数の宣言に対してその参照箇所の数を表示している．“18 references” などのテキストをクリックすると参照箇所の一覧とソースコードが表示される．

Inlay Hints は文字間に追加情報を表示する機能であり，典型的な用途として，メソッド呼び出しにおける実引数の前方に対応する仮引数名を表示する，式の後方にその式の型を表示する，などが挙げられる．図 5 は TypeScript プログラムに対して VSCode が提供する Inlay Hints の例であり，変数に対する型とメソッドの仮引数名が表示されている．

[*1] 具体的な表示内容や表示方法は IDE の種類や設定に依存する．

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "textDocument/definition",
  "params": {
    "textDocument": {
      "uri": "file:///..."
    },
    "position": {
      "line": 2,
      "character": 6
    }
  }
}
```

図 6 LSP リクエストの JSON の例

4.2 言語サーバプロトコル

言語サーバプロトコル (Language Server Protocol, LSP) は，エディタと言語サーバ間の通信を標準化するプロトコルである．言語サーバは，コード補完や定義への移動などプログラミングの支援に有用であるがプログラミング言語毎に実現する必要がある機能を提供するプログラムである．エディタは LSP に従って言語サーバから必要な情報を受信することで，エディタ自体はプログラミング言語のことを知らなくても開発者に支援を提供できる．

LSP が規定するエディタと言語サーバ間の通信は，エディタから言語サーバへのリクエストに対して言語サーバからエディタへレスポンスが返る双方向通信と，レスポンスが返らない通知を送信する単方向通信がそれぞれの向きに 2 種類の，計 3 種類がある．リクエストと通知のいずれも様々な種類のものがあり，それぞれ固有の名前（メソッドと呼ばれる）を持つ．双方向通信の例として，エディタ上で「定義への移動」を実行する場合には，エディタがソースコードの URI とカーソルの位置情報をパラメータとして `textDocument/definition` メソッドのリクエストを言語サーバに送信すると，言語サーバは移動先となる定義の位置情報をレスポンスとして返し，エディタはレスポンスの位置情報に基づいて表示を更新する．単方向通信の例として，エディタ上でソースコードが変更された場合には，その変更内容が `textDocument/didChange` メソッドの通知によって言語サーバに送信される．これにより，エディタ上で編集中のソースコードと言語サーバ側で把握しているソースコードを同期できる．

LSP の構造は HTTP の構造に類似している．ヘッダには Content-Lenght および Content-Type の 2 つのフィールドがあり，本体は図 6 のような JSON-RPC 形式となっている．`method` がリクエストや通知の名前，`params` がパラメータ名と値の組を持つオブジェクトである．`id` はリクエストとレスポンスを対応付けるための識別子である．`method` と `params` は LSP で定義されているが，独自に定義することもできる．

CodeLens や Inlay Hints の表示にも LSP が利用される．Inlay Hints の場合，ソースコードの表示に関係して Inlay Hints が必要になると，エディタはソースコードの URI と範囲情報をパラメータとして `textDocument/inlayHint` メソッドのリクエストを言語サーバに送信し，レスポンスとして返された Inlay Hints を表示する．レスポンスの Inlay Hints にはヒント文字列や位置情報が含まれており，これらに基づいてエディタ上に表示される．Inlay Hints の具体的な表示方法はエディタが理解しているため，言語サーバではレスポンスとして適切な Inlay Hints を生成するのみでよい．

5. 設計と実装

IDE を用いた機密度アノテーションを実現するプロトタイプツールを実装した．

5.1 対象の言語と IDE

プロトタイプツールでは対象のプログラミング言語として Rust を想定している．Rust プログラムを対象とする情報流解析の研究 [8] を行っていること，Rust には Java アノテーションのような仕組みが存在しないことが理由である．しかし，現在の実装は言語に依存しない範囲に留まっており，いくつかの制約が存在する．詳細については後述する．

対象の IDE として VSCode を選択し，プロトタイプツールを VSCode の拡張機能として実装する．Inlay Hints と LSP が実装されており，プラグインが開発可能であれば他の IDE でもよい．

5.2 実現の方針

機密度アノテーションの実現にあたり，ユーザインタフェースに関連して

- 機密度の注釈対象の指定方法
- 注釈する機密度の記述方法
- 注釈された機密度の表示方法

の 3 点，および機密度を表す文字列と注釈対象の位置情報を永続化する方法を決める必要がある．以下，機密度を表す文字列と注釈対象の位置情報をまとめて機密度アノテーション情報と呼ぶ．

注釈対象の指定と機密度の記述については，エディタ上で選択された範囲内の文字列を機密度の注釈対象とし，コンテキストメニューから機密度注釈の項目を選択して表示されるダイアログに機密度を入力することとする．この時，注釈対象として適切な文字列が選択され，機密度として適切な文字列が入力されるとする．機密度の表示については，Inlay Hints により注釈対象の範囲の直後に表示することとする．Inlay Hints を利用することで注釈対象のすぐ近くに表示できる．さらに，LSP を利用すること

で Inlay Hints の表示に関する IDE に固有な実装が不要となり，Inlay Hints を生成する言語サーバのみ実装すればよい．

機密度を Inlay Hints により表示するため，入力された機密度アノテーション情報は言語サーバに送信し，言語サーバ側で管理する．IDE の終了時に言語サーバも終了するが，機密度はソースコードの一部とみなすべきであるため，言語サーバの終了に伴って管理している機密度アノテーション情報が消えないようにする必要がある．ファイルに保存する方法が簡便であり，機密度をソースコードと一体的に扱うことが望ましいことから，ソースコードが記述されたファイルの末尾に，ソースコードのコメントとして，そのファイル内に存在する構文要素に対する機密度アノテーション情報を記録することとする．

5.3 ユーザインタフェースの実装

ユーザインタフェースの実装は，ユーザが機密度を入力するためのダイアログの表示と，ダイアログを表示するコマンドの定義の 2 つに大きくわけられる．利用できる機密度は決まっており，それらを文字列として入力できればよいため，VSCode で用意されている `QuickPick` を利用する．`QuickPick` は選択肢付きの文字列入力ダイアログである．ダイアログを表示し機密度が入力あるいは選択された後に機密度アノテーション情報を言語サーバに送信する処理をコマンドとして名前を付けて登録し，さらにコマンドとコマンドを起動するトリガーの対応を登録する．トリガーにコンテキストメニューを指定すればコンテキストメニューの適切なメニューから，キーコンビネーションを割り当てれば適切なキーボード操作により，ダイアログを表示して機密度を表す文字列の入力を受け付けることができる．

なお，ユーザインタフェースのうち Inlay Hints による機密度の表示に関しては，言語サーバから表示すべき機密度を Inlay Hints として提供するのみでよい．言語サーバに対する Inlay Hints のリクエストとレスポンスとして返された Inlay Hints の表示は，VSCode が適切に行うため実装の必要はない．

5.4 言語サーバの実装

言語サーバでは大きく 3 つの機能を実装する必要がある．1 つ目は，ダイアログから入力された機密度を受信する機能である．2 つ目は，機密度の注釈対象の位置情報をソースコードの編集に伴って適切に更新する機能である．3 つ目は，`textDocument/inlayHint` メソッドのリクエストに対して適切な機密度を Inlay Hints として返す機能である．

5.4.1 入力された機密度の通知と受信

言語サーバから機密度を Inlay Hints として返すために，IDE 上で入力された機密度アノテーション情報を言語サーバに送信しておく必要がある．しかし，LSP にそのような仕様は存在しないため，LSP を拡張して IDE から言語サーバに機密度アノテーション情報を送るための通知を定義する．

```
{
  jsonrpc: "2.0",
  method: "ifa-rust-vscode/add",
  params: {
    docUri: "file://.../sample.rs",
    sline: 2,
    scol: 0,
    eline: 2,
    ecol: 3,
    value: "L",
  }
}
```

図 7 機密度を送信する LSP 通知の JSON の例

機密度アノテーション情報を言語サーバに送信するために，図 7 のような通知を定義する．メソッドは `ifa-rust-vscode/add` とする．パラメータは 6 つで，`docUri` は注釈対象を含むソースコードの URI，`sline,scol,eline,ecol` はそれぞれ注釈対象の範囲の先頭と末尾の行番号と列番号，`value` が機密度を表す文字列である．IDE 側で LSP のクライアント API である `sendNotification` を，`method` の文字列と `params` のオブジェクトを引数に呼び出せば，言語サーバに適切な通知が送信される．

言語サーバが独自に定義された通知を受信すると，LSP のサーバ API である `onNotification` によって通知のメソッドと対応付けて登録されたコールバック関数が呼び出される．言語サーバでは，ソースコードの URI をキー，同一ソースコード内の各注釈対象の位置情報と機密度の文字列の組の集合を値とするマップとして機密度アノテーション情報を管理し，通知とともに受信したパラメータをコールバック関数内でマップに登録する．

5.4.2 位置情報の更新

言語サーバでは，通知されたソースコードの URI，注釈対象の位置情報，機密度を表す文字列の 3 つ組を管理する．ここで，注釈対象の位置情報は先頭と末尾の行番号と列番号であるため，注釈対象よりも前方のソースコードが変更されると行番号や列番号が変わる可能性がある．つまり，ソースコードの変更に伴い必要に応じて各注釈対象の位置情報を更新する必要がある．

更新は IDE 上で行われたソースコードの変更が言語サーバに通知されたタイミングで行う．言語サーバがソースコード変更の通知を受信すると，LSP のサーバ API である `onDidChangeTextDocument` によって登録されたコールバック関数が呼び出されるため，コールバック関数内で更新処理を実行する．

ソースコード変更の通知では，パラメータとして変更されたソースコードの URI と，挿入の場合は挿入位置と挿入

```
[
  {
    "spos": {"line": 1, "character": 0},
    "epos": {"line": 1, "character": 3},
    "level": "L"
  }
]
```

図 8　記録時の JSON の例

されたテキスト，削除の場合は削除された範囲の先頭と末尾の位置および削除されたテキストが送信される．これらの位置情報とテキストの行数や長さを用いて，ソースコード上の変更箇所よりも後方に位置する注釈対象の位置情報を更新する．

5.4.3　Inlay Hints の生成

IDE 側で Inlay Hints が必要になると言語サーバに対して `textDocument/inlayHint` メソッドのリクエストが送信される．言語サーバがこのリクエストを受信すると API の `_Connection.languages.inlayHint.on` によって登録されたコールバック関数が呼び出されるため，コールバック関数内でレスポンスとして機密度アノテーションのための Inlay Hints を生成する．Inlay Hints の生成は，Inlay Hints として表示したい文字列と位置情報を指定して生成する API を呼び出せばよい．

5.5　永続化の実装

機密度アノテーション情報は言語サーバ内で管理されているが揮発的であるため，適当なタイミングでディスクなどに記録する必要がある．記録のタイミング，記録先，記録形式は複数の選択肢があるが，本稿のプロトタイプツールでは，IDE 上でソースコードを保存する時および閉じる時に，同じファイルの末尾にソースコードのコメントとして追記することで記録する．記録する内容は，保存するソースコード内の各機密度アノテーションに対応する注釈対象の開始位置と終了位置および機密度の文字列の組である．これらを図 8 のような JSON 形式で記録する．

記録は言語サーバ側で実行する．記録のタイミングがソースコードを保存する時と閉じる時であるので，API の `onDidSaveTextDocument` と `onDidCloseTextDocument` でコールバック関数を登録する．コールバック関数内で，言語サーバが管理している機密度アノテーション情報の中から対象ソースコード内に注釈された機密度アノテーションの情報のみを取り出し，JSON 化とコメント文字列化を行い，既存のコメントを上書きするようにファイルに出力する．対象ソースコードの URI はコールバック関数の引数として渡される．

記録のタイミングや記録先に依存するが，記録を IDE 側で実行できる場合もある．ただし，IDE は機密度アノテーション情報を保持していないため，言語サーバから取り寄せる必要がある．例えば，LSP を拡張して機密度アノテーション情報を取り寄せるためのリクエストとレスポンスを定義すればよい．

図 9　注釈対象の選択

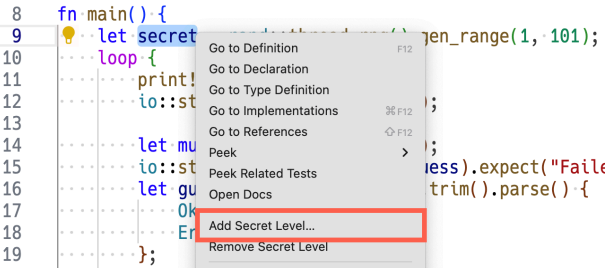

図 10　コンテキストメニュー

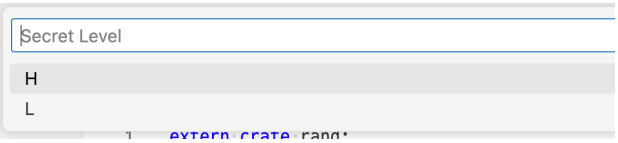

図 11　入力ダイアログ

機密度アノテーション情報の復元は IDE でソースコードを開く時に実行する．機密度アノテーション情報は言語サーバが管理するため，復元も言語サーバで行う．ソースコードを開くと API の `onDidOpenTextDocument` によって登録されたコールバック関数が呼び出されるため，コールバック関数内で該当ソースコードから機密度アノテーション情報の JSON を抽出して内部形式に変換する．

5.6　実行例

プロトタイプツールを利用してソースコードに機密度を注釈する例を示す．初めに，図 9 の変数 `secret` のように注釈対象とする変数全体を過不足のないようにエディタ上で選択する．選択状態のまま右クリックなどで図 10 にあるコンテキストメニューを表示し，機密度を追加する項目を選ぶ．図 11 のようなダイアログが開くので，適切な機密度を選択するか直接入力して確定する．機密度に `H` を選択すると，図 12 のように選択した機密度が Inlay Hints として注釈対象である変数 `secret` の直後に表示される．ファイルを保存すると，図 13 のように機密度アノテーション情報が JSON 形式でコメントとしてファイルの末尾に記録される．

6.　考察

6.1　コメントによる方法との比較

コメントは開発者が慣れ親しんでおり，ソースコードの一部としてキーボードから自然に入力できる点や，git な

```
8  fn main() {
9      let secret:H = rand::thread_rng().g
10     loop {
11         print!("> ");
```

図 12 Inlay Hints による機密度の表示

```
34  }
35
36  /*@ifa-rust-vscode
37  [{"spos":{"line":8,"character":8},"epos":{"line":8,"character":13},"level":"H"}]
38  */
```

図 13 記録された機密度アノテーション情報

どのバージョン管理ツールを利用すれば変更を容易に把握できる点が利点である．問題点は，コメントと対象の構文要素を一意に対応付けるためのルールが必要であり，そのルールを強制するにはそのための処理系が別途必要となる点である．本稿のように，対象の構文要素を開発者が IDE 上でグラフィカルに選択すれば，この問題は解決される．

プロトタイプツールでは，入力された機密度の表示に Inlay Hints を利用したが，コメントとしてソースコード中に挿入する方法も考えられる．しかし，対象の構文要素に対して適切なコメントの挿入位置がソースコードの書かれ方に依存することと，コメントの挿入によって対象の構文要素の位置がずれて管理が難しくなることから断念した．LSP を利用すれば Inlay Hints の表示に関しては何も実装する必要がなく，移植も容易になることは Inlay Hints を利用することの有利な点である．

定性的な評価ではなくプロトタイプツールを試用した著者の印象に過ぎないが，注釈対象の選択はコメントの記述位置の選択と同程度の手間であり，機密度は直接入力も候補からの選択もできる点は便利である．ダイアログの表示にキーボードショートカットを割り当てればキーボードのみで機密度の入力まで行えるため，使用感はコメントによる方法と大きく変わらないという印象である．

6.2 言語処理系の必要性

本稿の実装では，対象ソースコードの字句解析，構文解析，意味解析は行っておらず，プレーンテキストとして行番号と列番号のみを用いている．対象のプログラミング言語に依存しない実装ではあるが，このことにより主に 4 つの制約がある．1 つ目の制約は，注釈対象をエディタ上で選択しなければならないことである．これは字句解析を行っていないからであり，注釈対象として適切な字句の特定をユーザに任せている．カーソル位置から字句を適切に特定できれば，ユーザは注釈対象にカーソルを移動するのみで十分であり，エディタ上で選択する操作は不要となる．

2 つ目の制約は，注釈対象に選択された構文要素が注釈対象として適切な構文要素であるか確認できないことである．適切な構文要素のひとつは変数宣言中に出現する変数であるが，プロトタイプツールでは宣言される変数を表す文字列全体を過不足なく選択する責任をユーザに負わせている．字句の特定のみでは不十分であり，ツールが構文を理解している必要がある．

3 つ目の制約は，変数に対して機密度が注釈されている状態で，その変数の先頭あるいは末尾に隣接するように文字列が挿入された場合に，注釈対象の領域を拡張するか否か決定できないことである．例えば，注釈対象の変数が `abc` である時，`c` の直後に `d` が挿入された場合は注釈対象を `abcd` に変更すべきであるが，`+`の場合は注釈対象は変更しない．適切に対応するためには対象のプログラミング言語における字句の知識が必要である．

4 つ目の制約は，注釈した箇所にしか機密度を表示できないことである．例えば，注釈対象が変数宣言中の変数である時に，その変数の参照箇所でも機密度を表示するためには，意味解析を行って変数の宣言と参照を追跡する必要がある．

なお，情報流解析の検査器の実装には構文解析が必須であるので，最後の制約以外は検査器の実装の副作用として将来的には解消が見込まれる．

7. 関連研究

既存のプログラミング言語を拡張して情報流解析に対応させた言語が開発されている．例えば，Java を拡張した JFlow[9] や Jif[10]，Paragon[11]，Rust のサブセットを拡張した FlowRust[8]，OCaml のサブセットを拡張した Flow Caml[12] などの言語がある．言語仕様を拡張する場合，既存の言語処理系や開発環境をそのまま利用することは困難である．Jif 向けの IDE[13] などが提案されているように，何らかの対応が求められる．

HLIO[14] は Haskell プログラムを対象とする情報流解析のためのライブラリである．Haskell の言語仕様の範囲内で実現されているが，Haskell の強力な言語機能が前提であり，他のプログラミング言語への展開は難しい．

情報流解析のための Java アノテーション [5] では，機密度の指定や機密度束の定義のためのアノテーションが定義されている．アノテーションを利用することで，Java の言語仕様の範囲内で機密度の記述が可能であり，既存の言語処理系や開発環境による支援を受けられる．

本稿で検討した機密度を付与する手法は，より一般的には付箋を用いてソースコードに追加情報を上乗せする手法 [15] の変種であり，それを情報流解析のために応用したと考えることできる．Inlay Hints はソースコードの文字間に表示されることから，長くても 20 文字程度しか記入できないごく小さな付箋に相当し，機密度の表示には適している．

8. おわりに

本稿では，情報流解析における機密度をソースコードに

注釈するための，IDE の利用を前提とした手法を提案した．エディタ上で選択した構文要素に対して，ダイアログから機密度を入力し，Inlay Hints によって機密度を表示するプロトタイプツールを実装した．

今後の課題として，構文解析を導入して機密度の注釈対象が適切であるか確認できるようにすること，任意の機密度束に対応すること，Inlay Hints とダイアログ以外のユーザインタフェースを検討することなどが挙げられる．

謝辞 本研究の一部は 2023 年度南山大学パッヘ研究奨励金 I-A-2 の助成による．

参考文献

[1] Banerjee, A. and Naumann, D. A.: Secure Information Flow and Pointer Confinement in a Java-like Language, *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pp. 253–267 (2002).
[2] 黒川 翔，桑原寛明，山本晋一郎，坂部俊樹，酒井正彦，草刈圭一朗，西田直樹：例外処理付きオブジェクト指向プログラムにおける情報流の安全性解析のための型システム，電子情報通信学会論文誌 D， Vol. J91-D, No. 3, pp. 757–770 (2008).
[3] Sabelfeld, A. and Myers, A. C.: Language-Based Information-Flow Security, *IEEE Journal on Selected Areas in Communications*, Vol. 21, No. 1, pp. 5–19 (2003).
[4] Volpano, D., Smith, G. and Irvine, C.: A Sound Type System for Secure Flow Analysis, *Journal of Computer Security*, Vol. 4, No. 2, pp. 167–187 (1996).
[5] 吉田真也，桑原寛明，國枝義敏：情報流解析のための Java アノテーション，コンピュータ ソフトウェア， Vol. 34, No. 4, pp. 47–53 (2017).
[6] 桑原寛明，國枝義敏：機密度パラメータ付き情報流解析のための型検査アルゴリズムと Java アノテーション，ソフトウェア工学の基礎 XXVI(FOSE2019)，pp. 109–114 (2019).
[7] 桑原寛明：型検査に基づく手続き型言語向け情報流解析における型エラースライシング，コンピュータソフトウェア， Vol. 27, No. 4, pp. 221–227 (2010).
[8] 長谷川健太，桑原寛明，國枝義敏：Rust プログラムの情報流解析のための型システム，信学技報，SS2019-53, Vol. 119, No. 451, pp. 73–78 (2020).
[9] Myers, A. C.: JFlow: Practical Mostly-static Information Flow Control, *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pp. 228–241 (1999).
[10] Myers, A. C., Zheng, L., Zdancewic, S., Chong, S. and Nystrom, N.: Jif 3.0: Java information flow, Jif projet (online), available from ⟨`http://www.cs.cornell.edu/jif`⟩ (accessed 2024/01/25).
[11] Broberg, N., van Delft, B. and Sands, D.: Paragon for Practical Programming with Information-Flow Control, *Programming Languages and Systems*, pp. 217–232 (2013).
[12] Pottier, F. and Simonet, V.: Information Flow Inference for ML, *ACM Trans. Program. Lang. Syst.*, Vol. 25, No. 1, pp. 117–158 (2003).
[13] Hicks, B., King, D. and McDaniel, P.: Jifclipse: Development Tools for Security-typed Languages, *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, PLAS '07, pp. 1–10 (2007).
[14] Buiras, P., Vytiniotis, D. and Russo, A.: HLIO: Mixing Static and Dynamic Typing for Information-flow Control in Haskell, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pp. 289–301 (2015).
[15] 沢田洋平，大久保弘崇，粕谷英人，山本晋一郎：付箋によるコミュニケーション機能を備えたソフトウェアブラウザ，信学技報，Vol. 103, No. 189(SS2003-8), pp. 13–18 (2003).