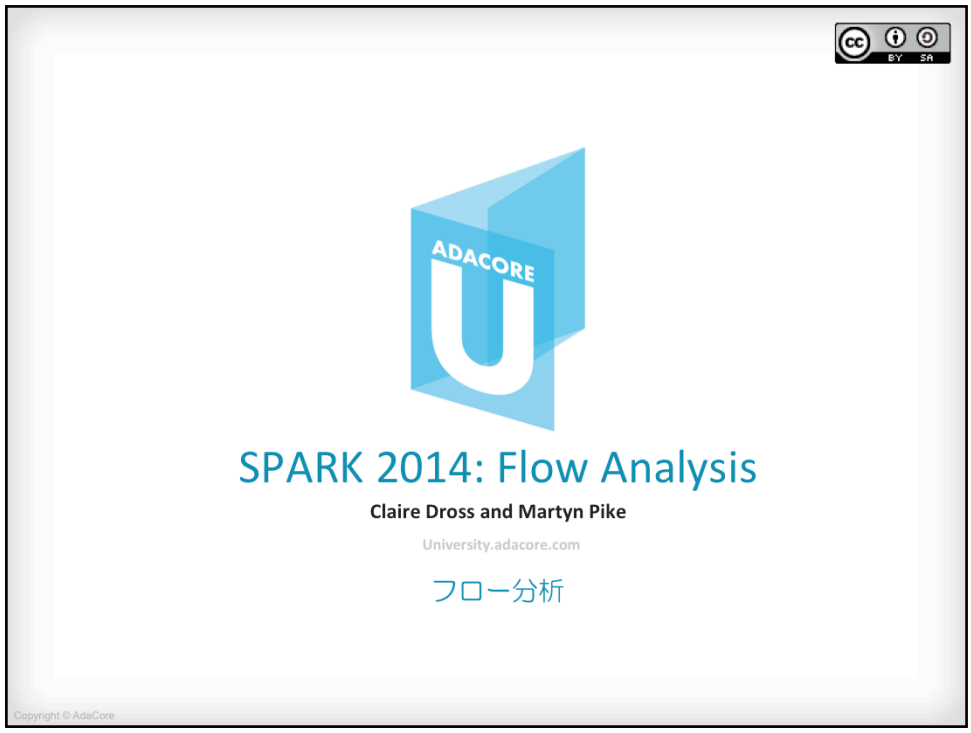
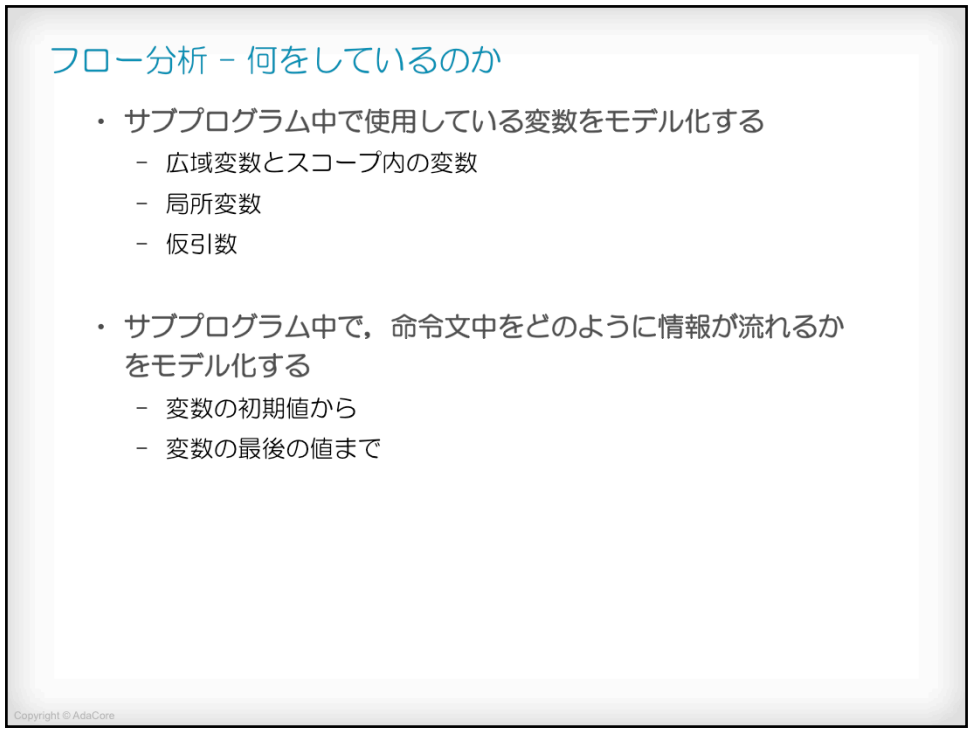


この講義では，SPARK2014 に対応した GNATprove ツールが提供する静的フロー分析能力について，説明します．



フロー分析は最初に変数を扱います

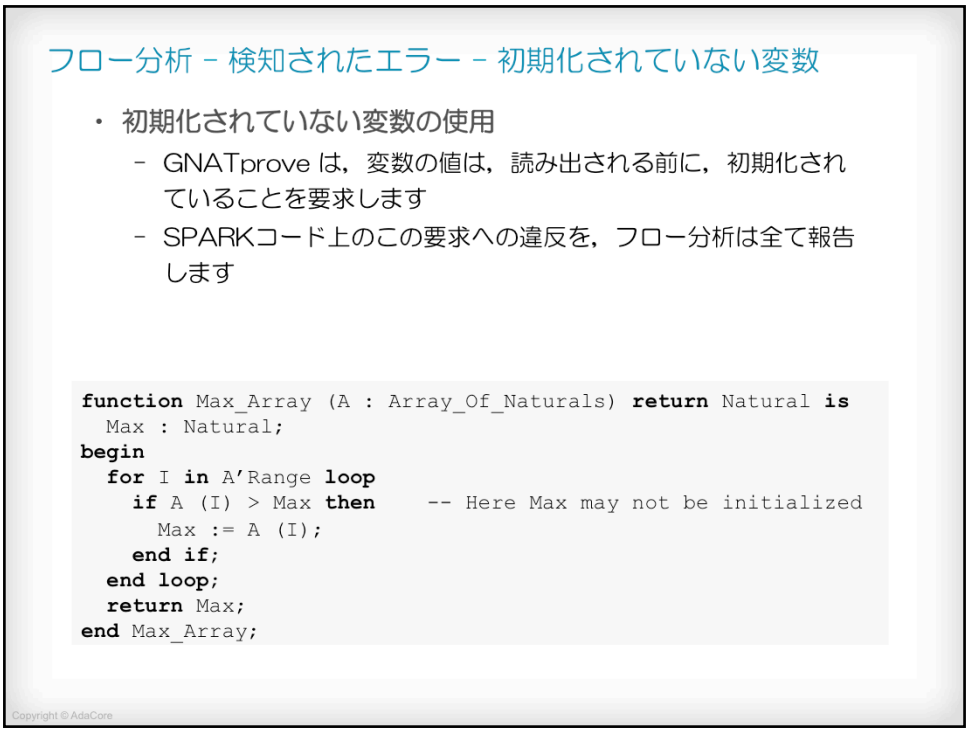
サブプログラムが実行中に，変数の間をどのように情報が流れるかをモデル化します．このとき，変数の最後の値を初期値に紐付けます．

また，ライブラリレベルで宣言されている広域変数も考慮に入れます．サブプログラム内部で定義されている局所変数や，仮引数（formal parameter）も同様です．

入れ子になったサブプログラムに対して，内包ユニットに対して局所的に宣言された変数を，「スコープ変数」と呼ぶ．スコープ変数は，入れ子のサブプログラムの観点からは，実質上広域変数と見なすことができます．

フロー分析は，通常高速です．コンパイル時間とほぼ同じ時間で終了します．

SPARK の要求に対する違反と同様に，種々のを検知することができます．式中に，別名化と副作用がないことの検査を含みます．フロー分析は確実な検査を行うので (sound)，もし，対象となっているエラーが発見されなければ，実際にも存在しないと考えることができます．

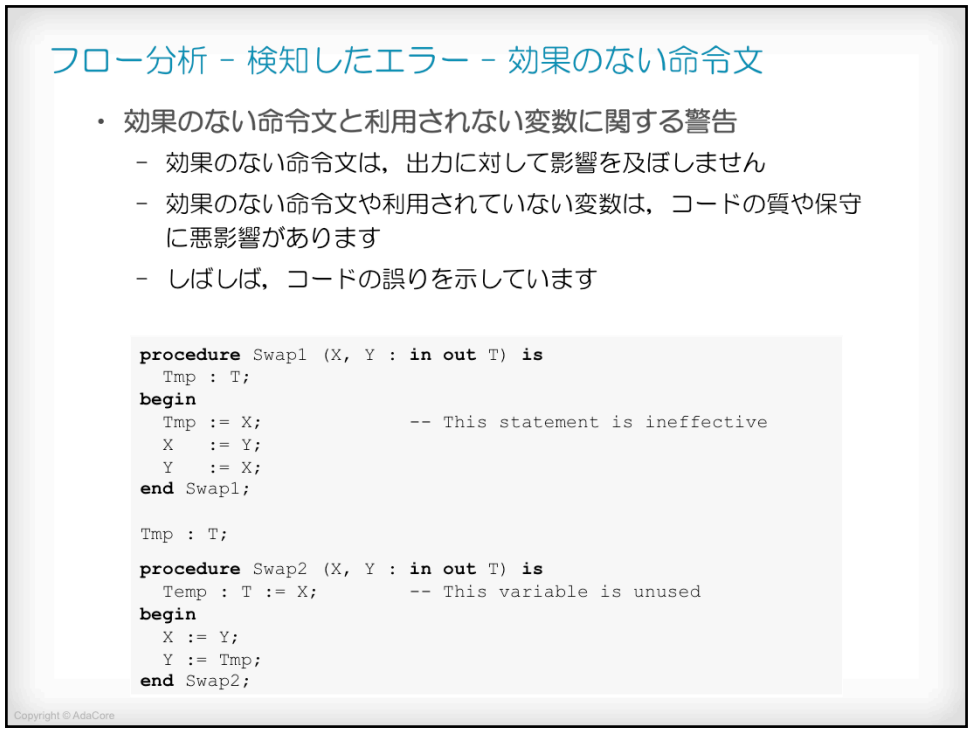


フロー分析で検知できる全てのエラークラスを示します．初期化されていない変数（ほとんどの場合エラーとなります）を読むことから始めます．

（初期化されていない変数は）非決定論的であり，型システムを壊すことがあります．変数の値が，副型の範囲外にあるかもしれないからです．こういった理由から，GNATprove は，全ての変数は初期化されるべきとしています．

フロー分析は，SPARK コードがこの要求を満足していることを保証します．例えば，このスライド中の Max_Array 関数中で，ループに入る前に，Max の値を初期化することを忘れてはいけません．

IF 文中のガード条件中で読み出される値は，初期化されていない可能性があります．フロー分析は，このエラーを検知し，報告します．



役に立たないコードは，デッドコード（到達しないコード）とは違い，実行されます．多くの場合，変数の値を変更することもあります．

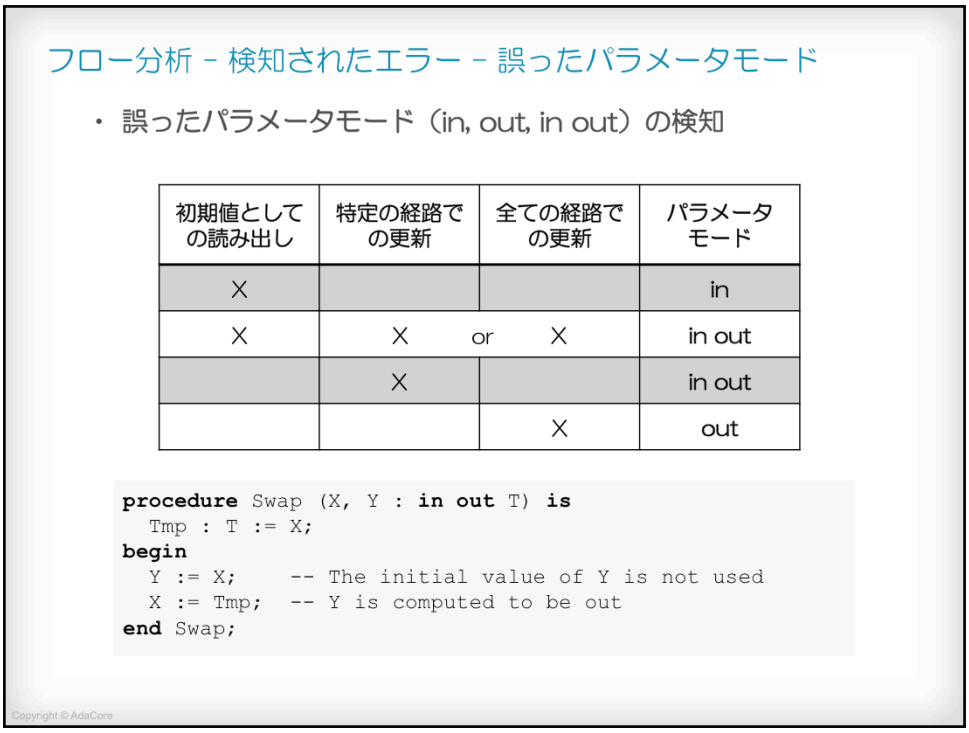
パラメータ，広域変数，或いは関数の結果といったサブプログラムの出力に影響を及ぼしません．

利用されていない変数と同様に，効果のないコードを，避けるべきです．コードを読むのが難しくなり，保守が困難になるからです．

更に，プログラム中でエラーが生じている場合があり，それを見つけ出すのが難しい場合があります．

サブプログラム Swap1 と Swap2 の場合がそうです．ここでは，パラメータ X と Y の値が正しく交換されていません．

それ自身は，誤りでなくとも，フロー分析では，効果のない命令文と利用されていない変数がある，と警告します．



パラメータモードは，コンパイラの振る舞いに影響を与えます．また，サブプログラムの利用方法を文書化する時に重要です．

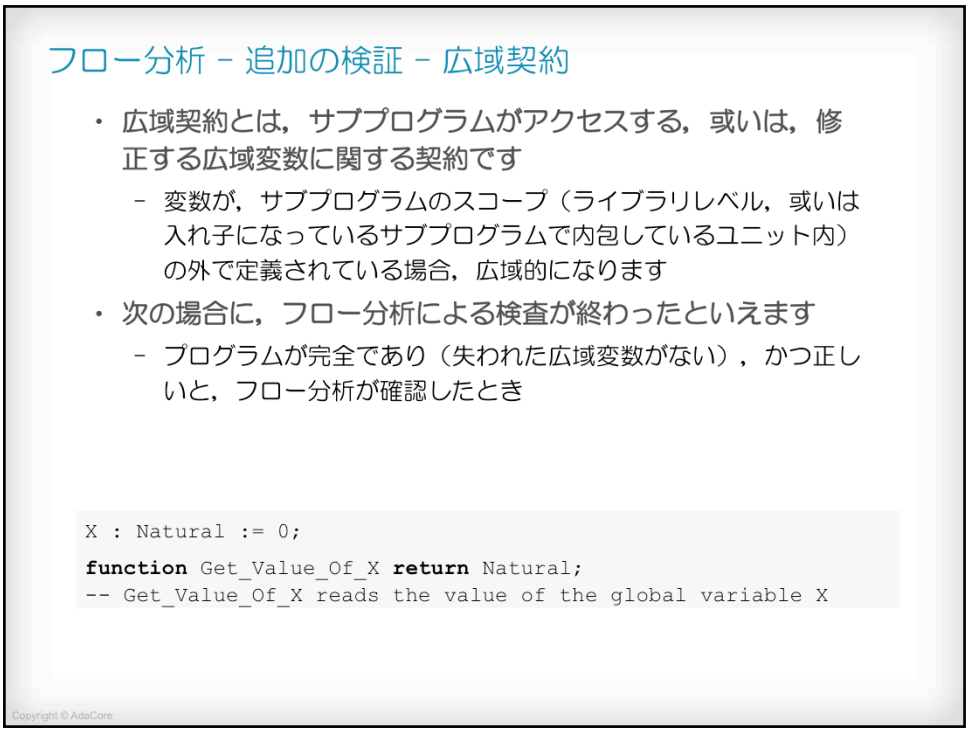
フロー分析は，特定のパラメータモードが，サブプログラムのボディ部における（実際の）利用とどう関係しているかを，常に検査しています．

更に正確には次の様にいえます．直接に，或いはサブプログラム呼び出しを通して，”in“パラメータが更新されることはない，ということを検査します．

同様に，”out”パラメータの初期値は，サブプログラム中で呼び出されることはない，ということも検査します．サブプログラムのエントリーで，（値を）コピーされることはないからです．

最後に，“in out” パラメータが更新されない，その初期値がサブプログラム中で使用されない時に，フロー分析は警告を出します．誤りの兆候かもしれないからです．スライド中の Swap を呼び出すサブプログラムがこの例になります．

値の読み出しがなく，全経路での更新もないパラメータは，SPARKでは，”in out“ として宣言されるべきであることに注意して下さい（表中三番目）．最終的な値が，その初期値に依存するかもしれないからです．



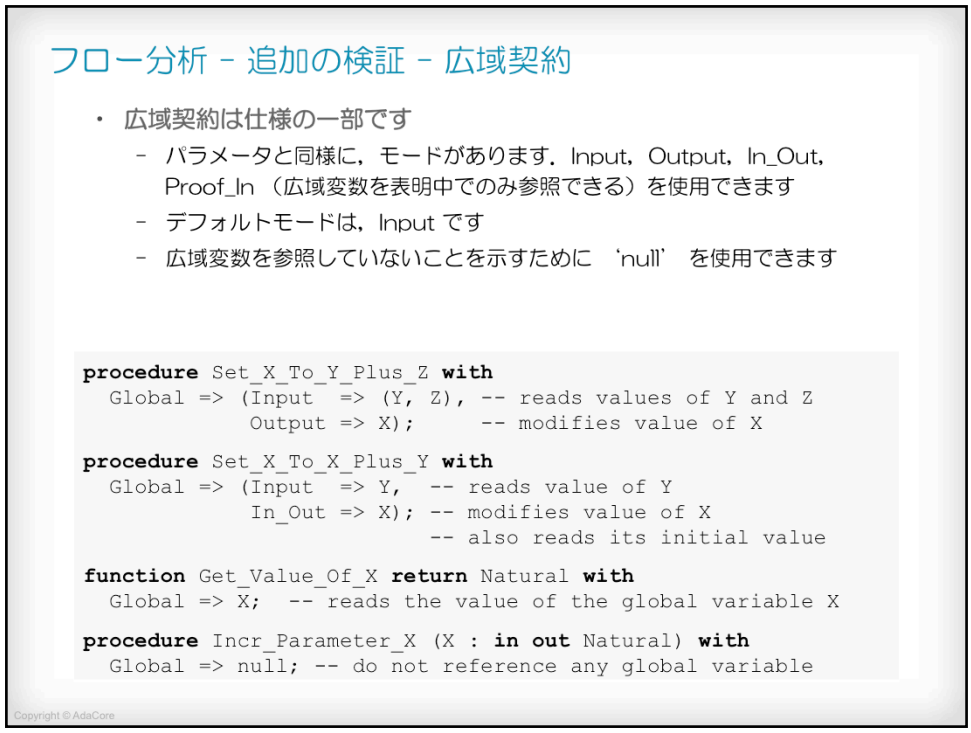
これまで，開発者が注釈を追加する必要がない検証について見てきました．フロー分析は，ユーザが記述したフロー注釈もチェックします．

SPARK では，サブプログラムがアクセスし更新する広域的・特定のスコープを持った変数を指定することが可能です．

これは，Global と名付けられた契約と同様に Ada 2012 を用いて行われます．

Global 契約が，あるサブプログラムに対して与えられたとき，フロー分析は，それが正当であり完全であると検査します．即ち，契約中で述べられている変数のみがアクセスされ，更新されます．直接的な場合もありますし，サブプログラムにより行われる場合もあります．

例えば，関数 Get_Value_Of_X が，広域変数 X の値を読み出し，他の広域変数にはアクセスしないと記述したいとします．



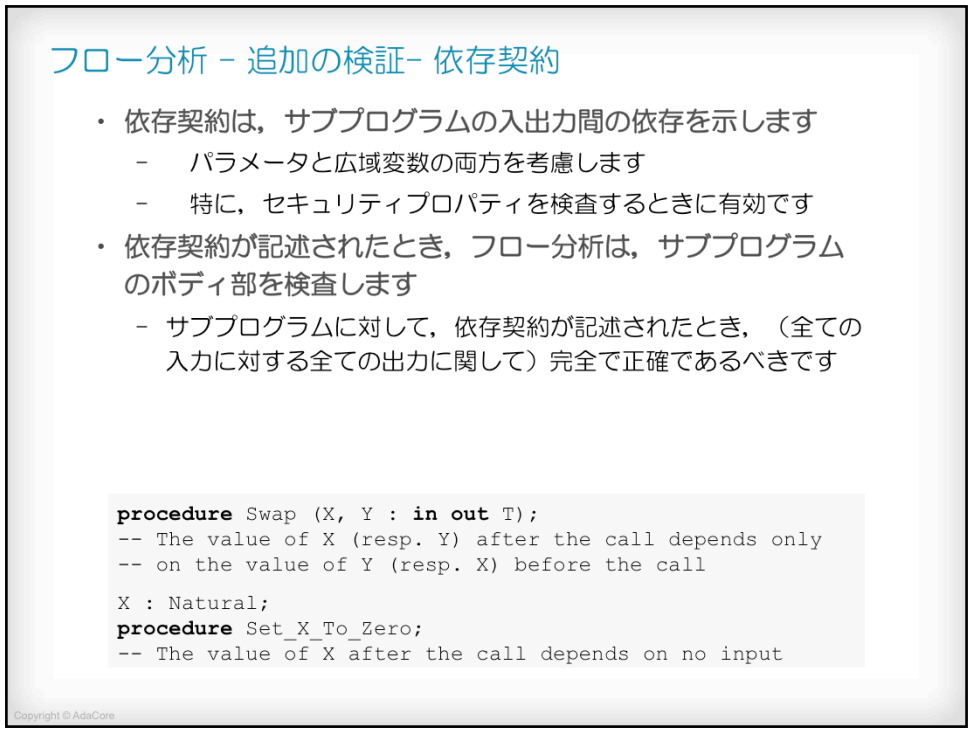
広域契約は，サブプログラム仕様の一部として提供されます．さらに，サブプログラムのユーザに有益な情報を提供します．

Global アスペクトに対して記述される値は，モードに従ってグループ化され，広域変数名の集約型風のリストとなります．

例えば，手続き Set_X_To_Y_Plus_Z は，Y と Z の値を読むので，Input とし，X を更新するので，Outputとしています．Set_X_To_X_Plus_Yでは，X の初期値を読み，値を更新するので，Xのモードは In_Out になります．

パラメータの場合と同様に，モードが指定されていなければ，デフォルト値は，Input になります．Get_Value_Of_X における宣言の場合が相当します．

最後に，Incr_Parametr_X のように，サブプログラムが，広域変数を参照しないのであれば，広域契約の値は，‘null’ と指定されるべきです．

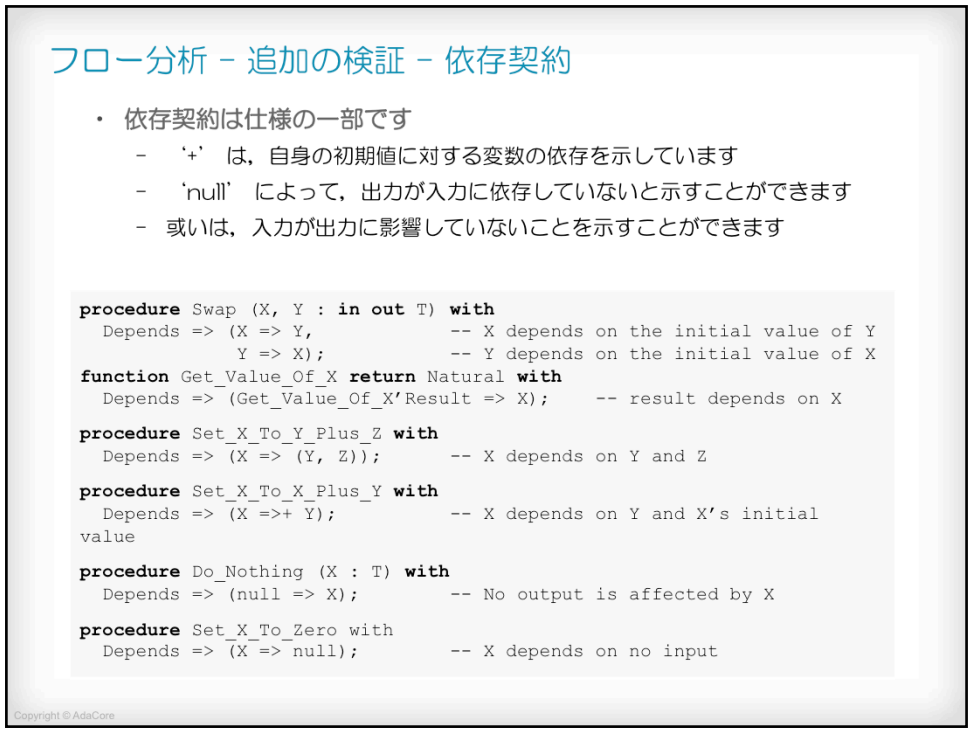


また，サブプログラムがその出力と入力間に依存があることを記述するために，ユーザは依存契約を指定することができます．

ここで，広域変数を考慮し，またパラメータと関数の結果についても考慮します．

依存契約がサブプログラムに対して与えられるとき，フロー分析は，それが正当で完全であるかを検査します．すなわち，各サブプログラム出力が，全ての入力に関係しているかです．

例えば，あるユーザは次のようなことを確認できます： Swap からの戻りにおいて，各パラメータが他のパラメータの初期値に依存している．手続き Set_X_To_Zero の戻りにおいて，Xの値は，広域変数に依存していないといったことです．



広域契約のように，依存契約は，アスペクトを用いてサブプログラム宣言上に記述されます．

その値は，プログラムの出力と入力間の一つ以上の依存関係です．各関係は，矢印で区切られた二つの変数名の並びとして表現します．

矢印の左側は，その最終的な値が，右側の変数の初期値に依存しています．

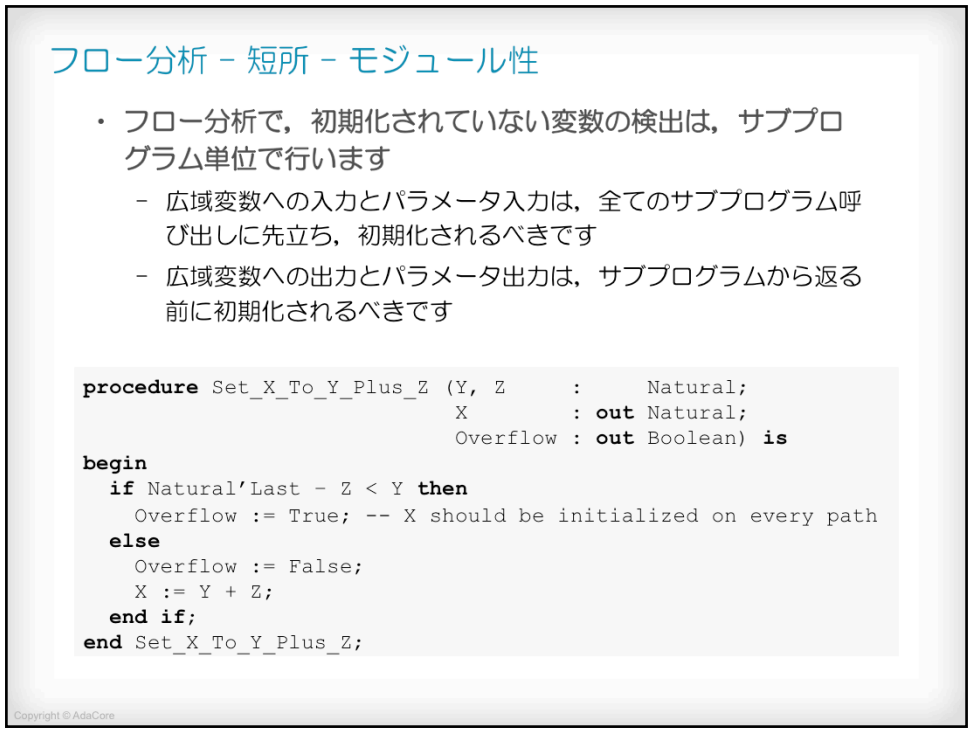
例えば，Swapの各パラメータの最終的な値は，他のパラメータの初期値にのみ依存しています．もし，サブプログラムが関数であれば，その結果は出力の並びに含まれるべきです．Get_Value_Of_Xをみてください．

変数の最終的な値が，それ自身の初期値に依存することがしばしばあります．このときは，‘+’文字を用いて，より簡潔に記述することができます．Set_X_To_X_Plus_Y にその例があります．矢印の左側に一つ以上の変数がある場合は，‘+’は，各変数がそれ自身に依存していることを示しています．全てが互いに依存しているわけではない，ということに注意して下さい．

入力が，どの出力の最終値を計算するためにも用いられていない場合があります．これは，nullを依存関係の左側に記述することで，表現できます．この例（4番目）では，Do_Nothing 手続きのように記述します．

そのような依存関係は一つであり，サブプログラムの使用していない全ての入力の並びは，最後に記述するということに注意する必要があります．また，未使用のパラメータに関してフロー分析は警告を行わないということにも注意が必要です．

最後の例です．null は，また依存関係の右側に記述される場合があります．出力は，入力に依存してい



フロー分析は，確実に分析します．つまり，もし，分析した SPARK コード上にメッセージの出力がなければ，ツールが検知可能なエラーはコード中に存在していないということを意味しています．

一方で，実際にはエラーがなくても，フロー分析がメッセージを出力する場合があります．

最初のケースは，モジュール性に関係しており，最も良く生じます．

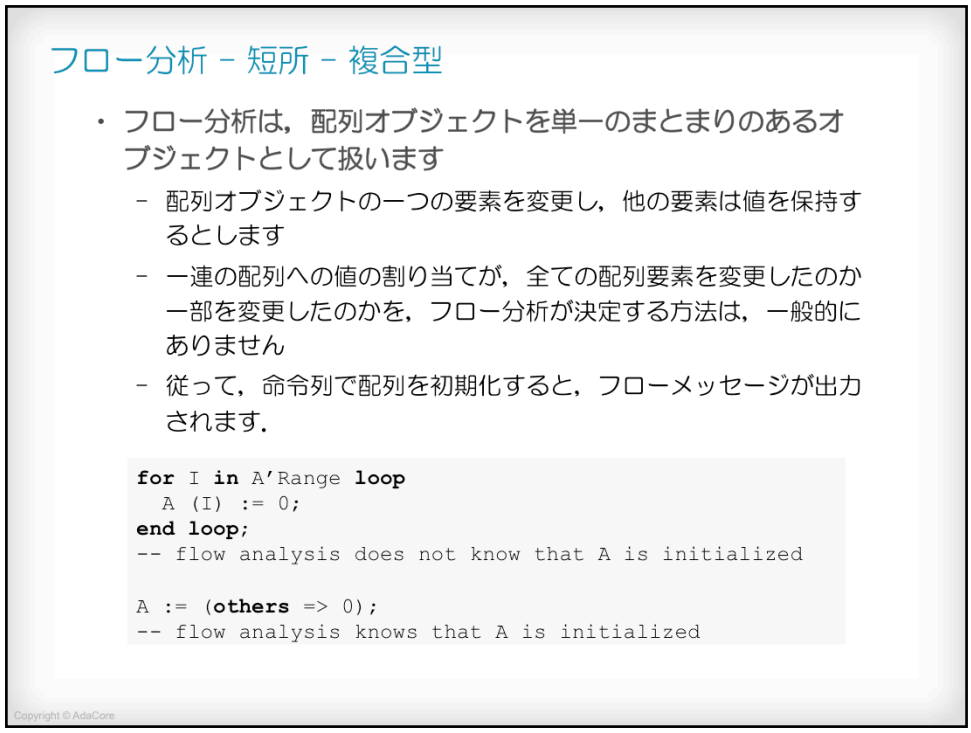
大きなプロジェクトで，効率性を向上させるために，一般に，サブプログラム単位で，検証を行います．

これは，特に，初期化されていない変数を検知した場合です．

この検知が，モジュール単位で行われた場合，サブプログラムにエントリーしたときに入力が初期化されていることと，サブプログラム呼び出しのあとで，出力が初期化されていることを，フロー分析は，前提としています．

従って，サブプログラムが呼び出されるたびに，広域変数と入力パラメータが初期化されていることを，フロー分析は検査します．また，サブプログラムから返ってくるたびに，広域変数とパラメータ出力が初期化されていることを検査しています．

このことは，Set_X_To_Y_Plus_Z のような完全に正しいサブプログラム上において，メッセージが発行されることになります．パラメータ X は，Overflow が偽の場合にのみ設定されるからです．
これは単純には次のコトを意味します．フロー分析は，初期化されていない変数の値を読むことはないということを，検証することができないということです．



誤った警告を出す他のあり得る原因は，フロー分析中で複合型を扱う方法です．

最初に特に配列について見てみます．

フロー分析において，配列オブジェクトを，単一のオブジェクトとして扱います．

このことは，配列要素の一つを更新することが，配列オブジェクト全体の更新として扱われることを意味します．

明らかに，アクセスする広域変数と依存性に関して，推論が不正確になります．

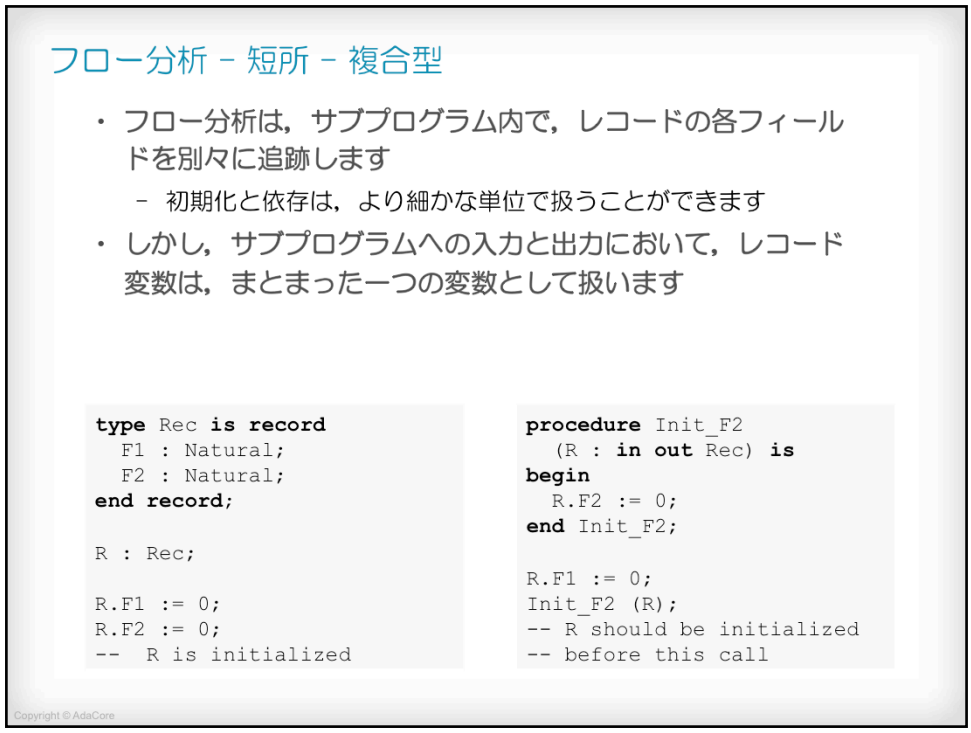
しかし，初期化されていない変数の値の読み出しを検知することにも，また影響があります．

実際に，オブジェクト全体が初期化されたかどうかを，フロー分析は，しばしば，決定することができません．きわめて単純なケースにおいてもです．

例えば，次の場合を考えます．無制約配列型 A の全ての要素をゼロに初期化した後，それでも，配列が初期化されていないとフローメッセージが云う場面に出会います．

この問題を解決するためには，ユーザは配列集成式（aggregate）を使用するか，もしそれが難しければ，他の手段でオブジェクトの初期化を検証する必要があります．

（訳注1）複合型（Composite Object）は，配列型とレコード型を指す．構造型という訳語が与えられる場合もあるが（JIS X），レコード型と混同しやすいので，複合型とする



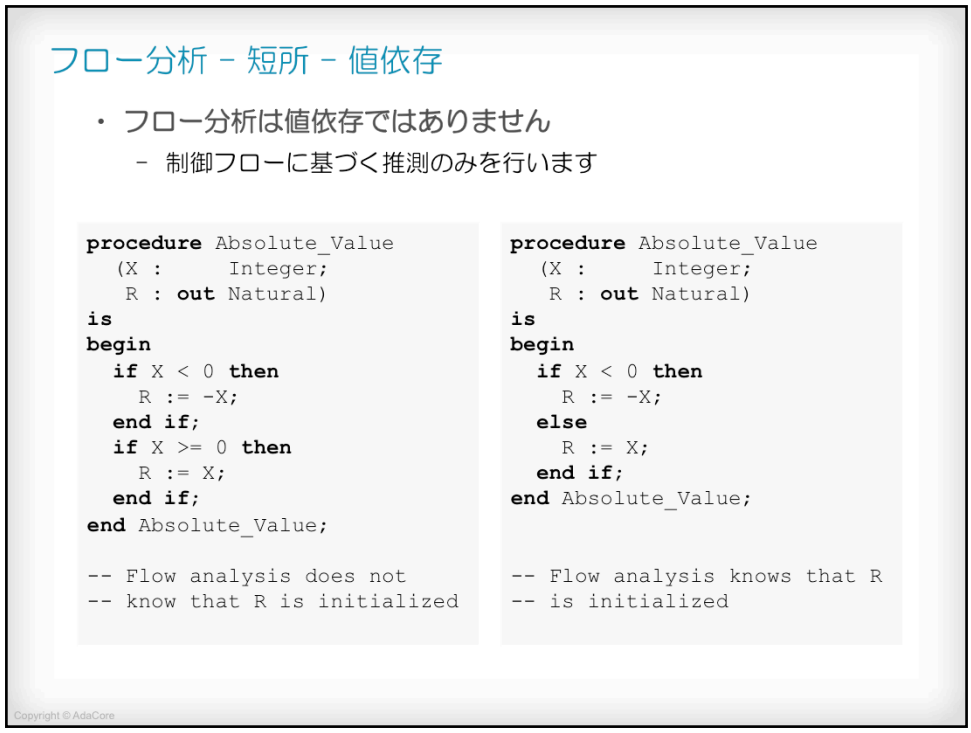
フロー分析は，レコードオブジェクトに対して，より正確な分析を行います．単一のサブプログラム内で，各コンポーネントの値をそれぞれ追跡できるという意味においてです．

その結果として，レコードオブジェクトは，そのコンポーネントへの連続する割り当てによって初期化されたとき，フロー分析は，全てのオブジェクトが初期化されたと確認することができます．

レコードオブジェクトが，サブプログラムの入力・出力で用いられるときは，引き続き一体のオブジェクトとして扱われることに注意して下さい．

例えば，レコードオブジェクトの幾つかのコンポーネントのみを初期化する手続き呼び出しがあると，フロー分析は次のように警告します．初期化されるべきコンポーネントがサブプログラムのエントリで初期化されていない．これは，Init_F2と同様です．

（訳注）レコード型は，様々な型から構成されるもの（c.f. C言語の構造体）
レコードオブジェクトは，名前付きコンポーネントからなる複合体である（Ada参照マニュアル 3.8）



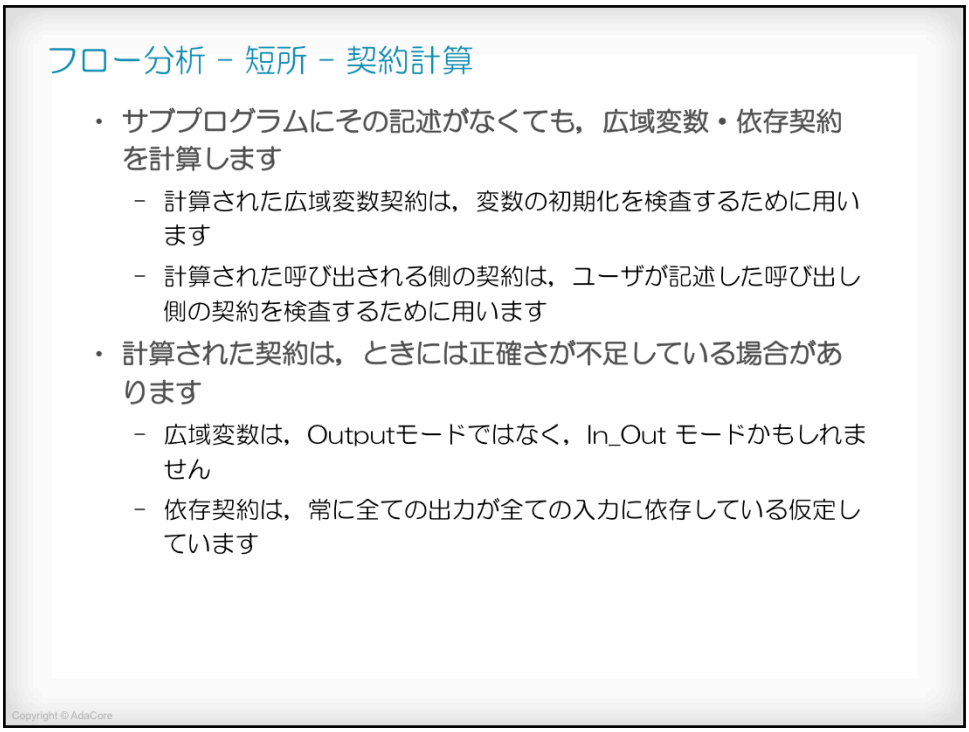
また，フロー依存は値依存ではない，ということには余り意味がありません．フロー分析は，式の値について，決して推測を行わないからです．

結果として，もしサブプログラム中のあるパスが式の値によって実行不可能であるとき，フロー分析は，そのパスが適切であるとみなすし，その点に関する不要なメッセージを出力しません．

Absolute_Value の左側の例において，二つの条件式で，（構造上）ともに偽となるパスがあると，フロー分析は　R　は初期化されていないと考えます．

フロー分析は，式の値を考えることはありません．従って，そういった場合が起こらない，とは考えません．

この問題をさけるために，制御フローを明示的に記述することが望ましくなります．
Absolute_Valueの右側のバージョンをみてください．



最後に，フロー契約の計算における不正確さから，望まないフローメッセージが出力される場合があることについて考えます．

どうしてフロー分析は，契約を計算するのでしょうか．最初の頃に説明したように，広域契約と依存契約の記述は，選択可能です．

しかし，GNATprove は，分析のためにある程度，各契約を必要としています．

例えば，サブプログラムによってアクセスされる広域変数の組を知ることは，初期化されていない変数の使用を検知するために必要です．

あるサブプログラム中の依存契約に関して云えば，ユーザが記述したサブプログラムの呼び出し側の依存契約を検査するために必要です．

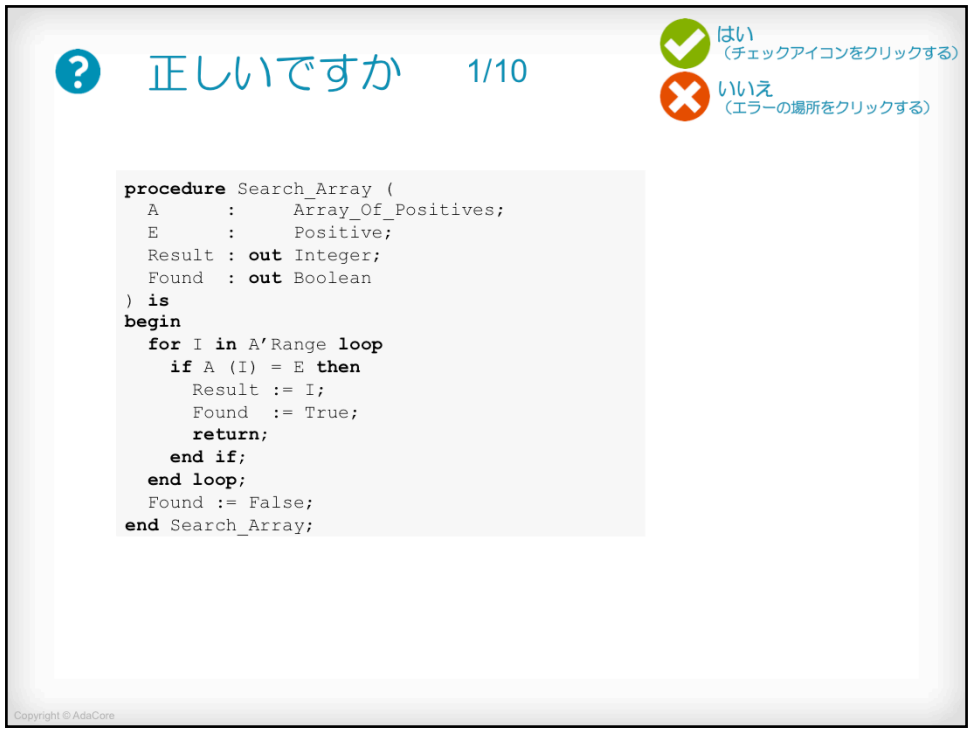
サブプログラムに関する各フロー契約は，そのボディ部内部で呼ばれる全てのサブプログラムのフロー契約に依存しています．この計算は，たちどころに計算時間を消費します．

それゆえ，フロー分析は，効率化のためにこの計算の正確さに対するトレードオフがあります．

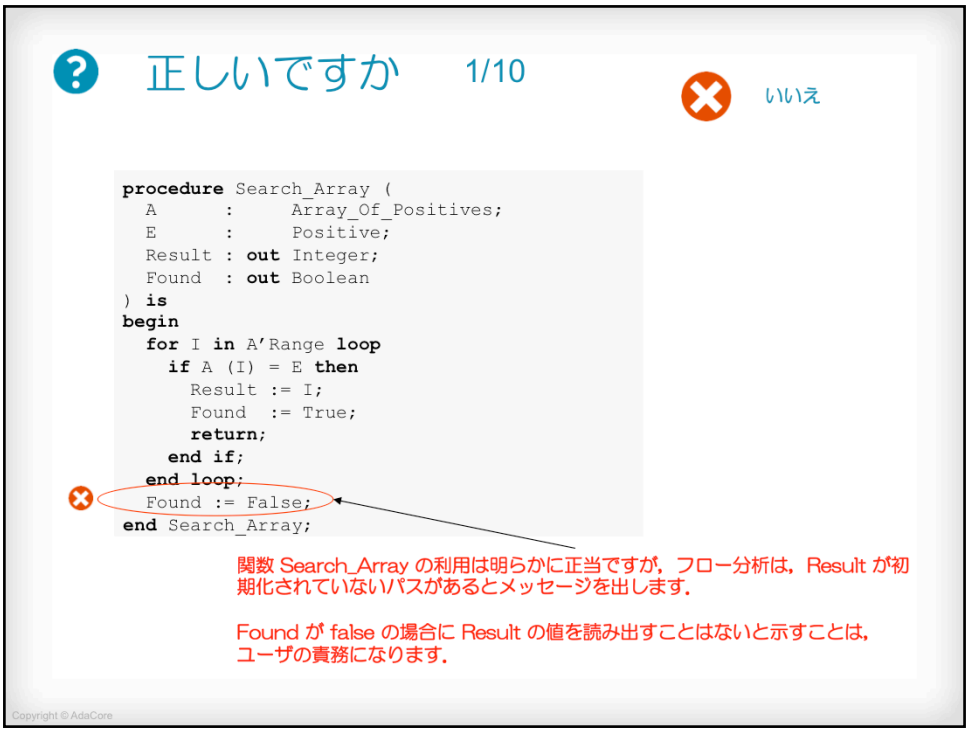
特に依存契約が相当します．フロー分析は最悪の場合を想定するからです．それは，各サブプログラムの出力は，全てのサブプログラムの入力に依存すると仮定することです．

この問題を解くために，正確さが十分に得られないならば，手動で契約を追加すれば十分です．広域契約は大きなプロジェクトにおいてフロー分析の時間短縮につながる良いアイデアであることを心にとめ

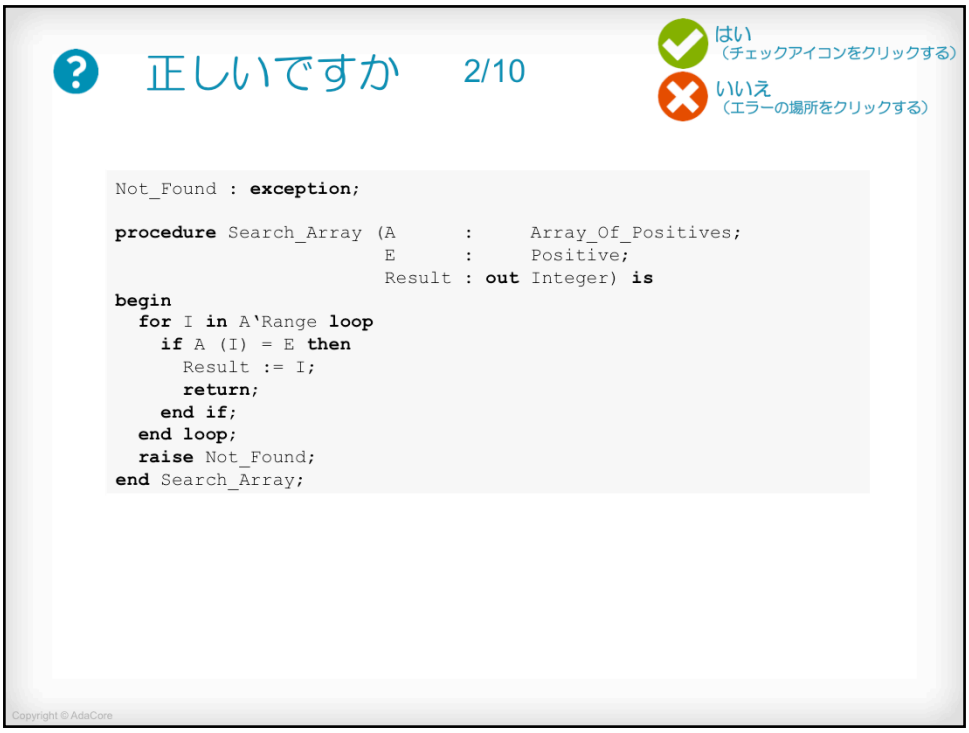

Quiz
Copyright © AdaCore



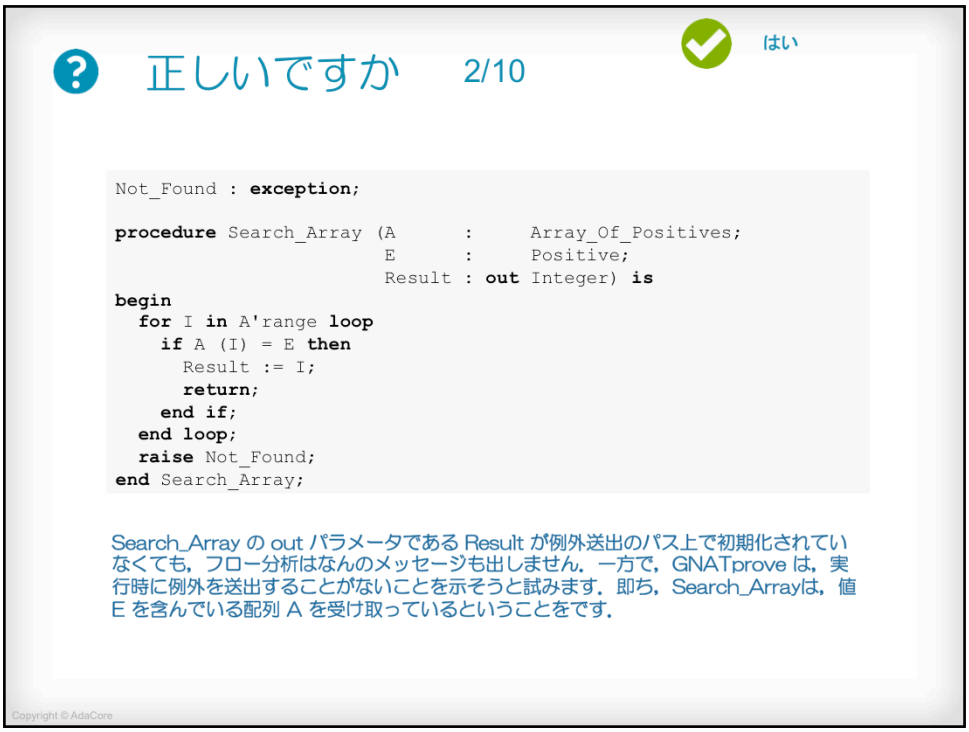
手続き Search_Array は，配列 A の中に，特定の値 E を探します．もし，その要素が見つかれば，Result に保持します．そうでなければ，false をセットします．



関数 Search_Array の利用は明らかに正当ですが，フロー分析は，Result が初期化されていないパスがあるとメッセージを出します．このプログラムが正しくても，フローメッセージを無視できない場合があります．実際，Resultが初期化されていないときに決して読み出されないということを，フロー分析は保証できないことを意味します．そしてそのことは，GNATprove によるより詳細な分析の仮定となります．それゆえ，Found が偽であるときに Result を初期化するか（フロー分析のメッセージはなくなります），他の手段によってこの仮定を検証する必要があります．



前のスライドにあるフローメッセージを避けるために，Search_Arrayは，配列 A 中で値 E が見つからないという例外を送出します．

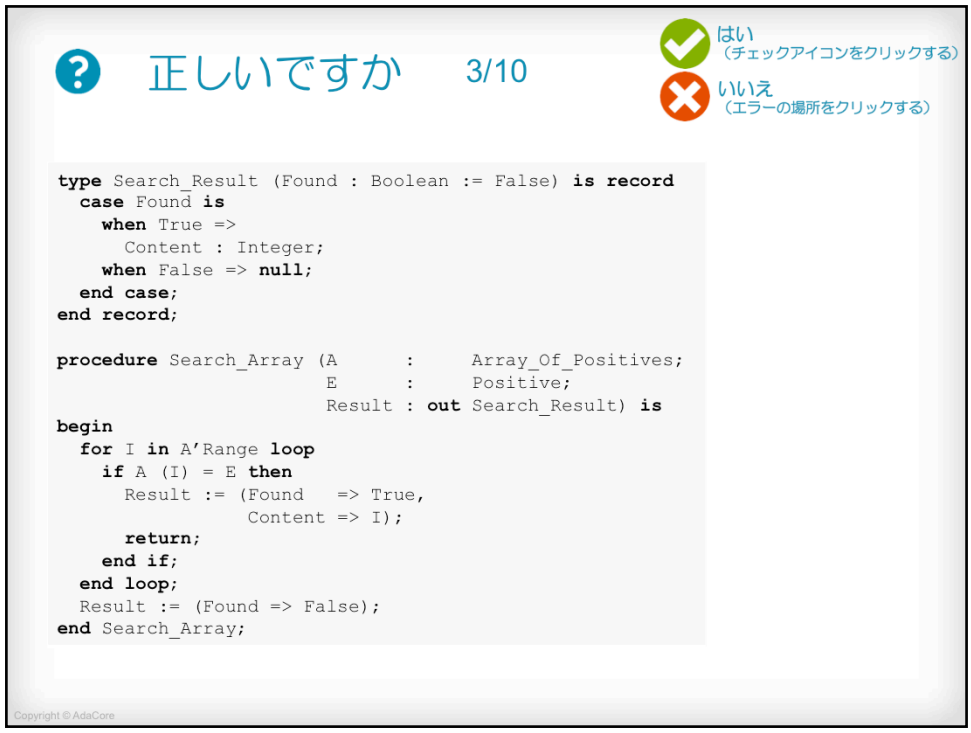


フロー分析は，ここではメッセージを発行しようとはしません．SPARK コード中では，Result が初期化されるまま読み出されることはないということが確かだからです．Result は，値 E が配列 A に存在しないときに，初期化されていないにも関わらずである．どうしてでしょうか?

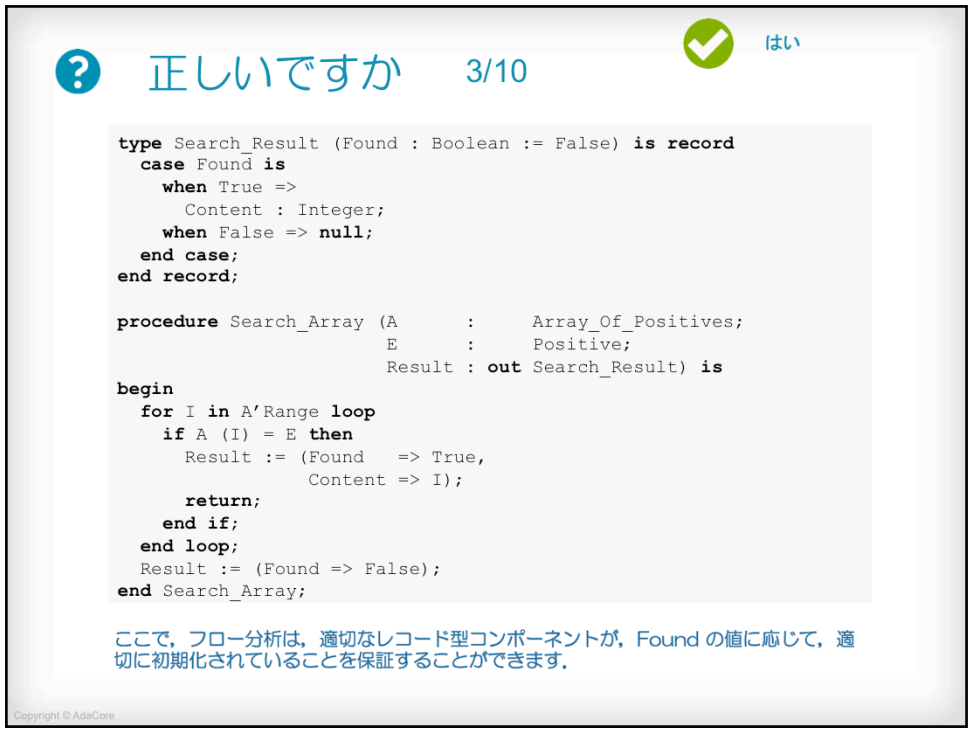
例外 Not_Found は，SPARKコード内部で決して捕らえられることはできないという事実からそうなります．

それゆえ，Resultが初期化されていないときに，決して読み出されないということを保証する責務は，ユーザの責務になります．もはや，ツールによってのみ明示的に示されることはないのです．それは，一般的な仮定のカテゴリに属します．また，このことはユーザガイドに示されています．

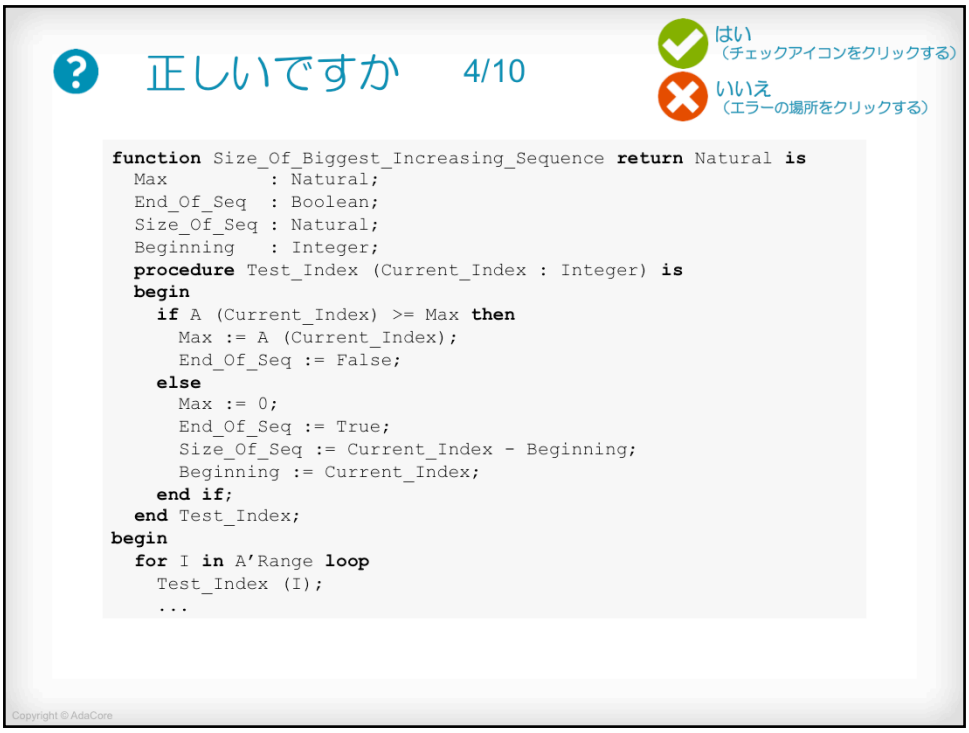
また，GNATproveツールは，SPARKコード中に実行時エラーがないことを保証するための一部としてこのプログラム中で，Not_Found 例外が送出されないことを保証しようと，試みることに注意して下さい．



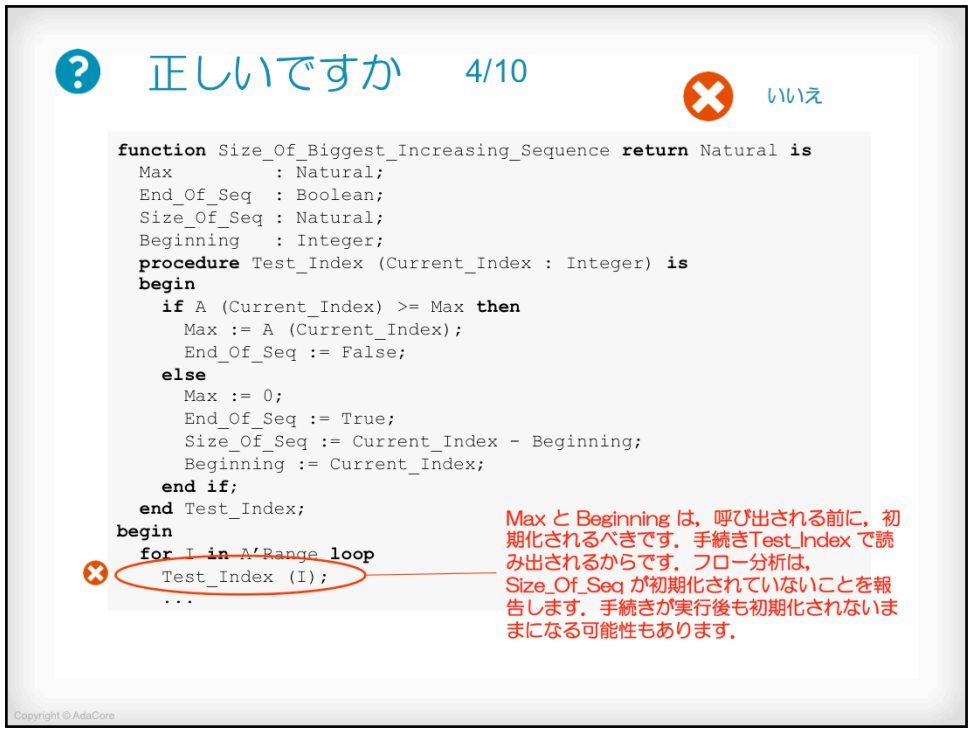
例外を送出する代わりに， Search_Array の結果に対して，区別子付きレコード（discriminant record）を用いる方法があります．ここで示したように，A 中で発見した E のインデックスは， E が実際に見つかったときのみ設定されます．



フローメッセージはここでは発行されません．初期化されていない変数は，Search_Array のボディ部で読み出されることはなく，全ての出力は，手続きから戻るときに初期化されているからです．

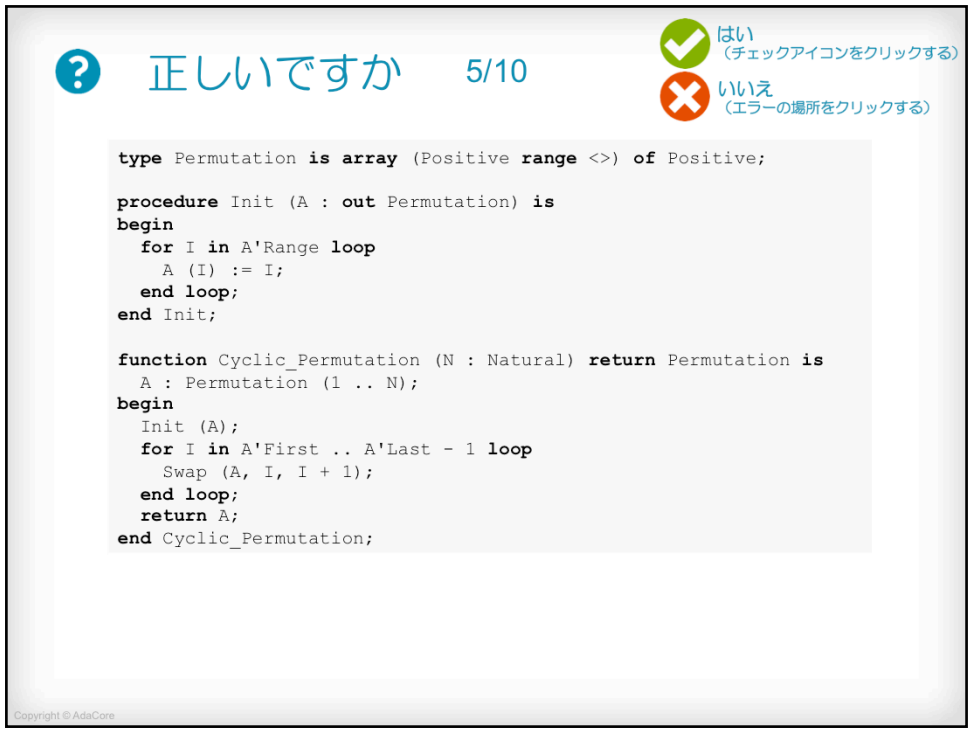


関数 Size_Of_Biggest_Increasing_Sequence は，広域配列 A のシーケンスを全て点検します．この配列は，最大の長さを計算するために，増加する要素を含んでいます．入れ子になっている手続き Test_Index は，A の全ての要素で繰り返し呼び出されます．もし，シーケンスが増加中であれば，Test_Index は，検査をします．その場合，これまでの最大値を更新します．そうでなければ，増加するシーケンスの最後を見つけたと云うことを示します．従って，このシーケンスのサイズを計算し，Size_Of_Seq にその値を保持することになります．

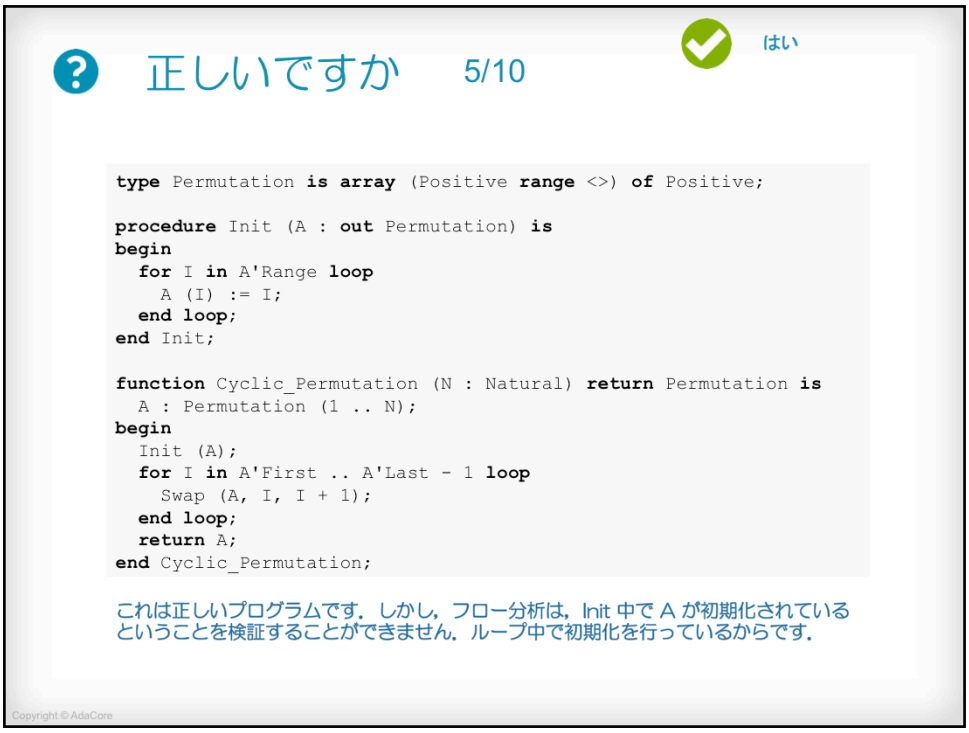


フロー分析は，MAX，Beginning, Size_Of_Seq は，呼び出し前に初期化されるべきであると Test_Index の呼び出し時にメッセージを発行します．

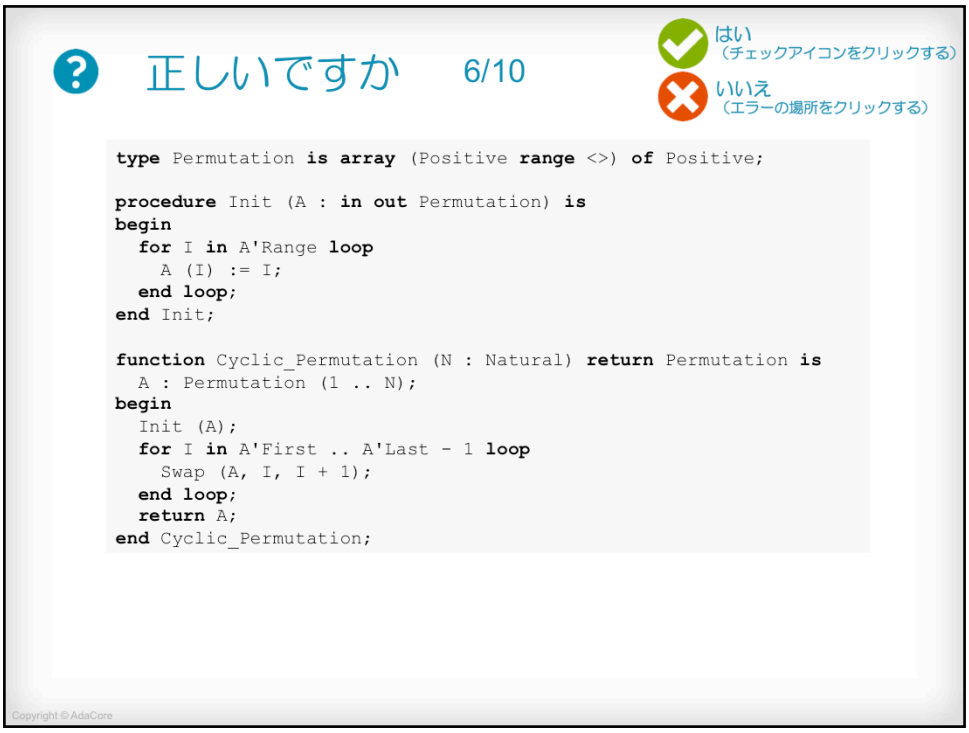
実際，MAX と Beginning は，Test_index で呼ばれるときに初期値を必要とします．Size_Of_Seq についていえば，End_Of_Seq が真であるときに，その値を読むだけであれば，そしてそれが設計であれば，問題はないでしょう．フロー分析は，モジュールの内側で，その初期化に関して，単純に検証することはできません．



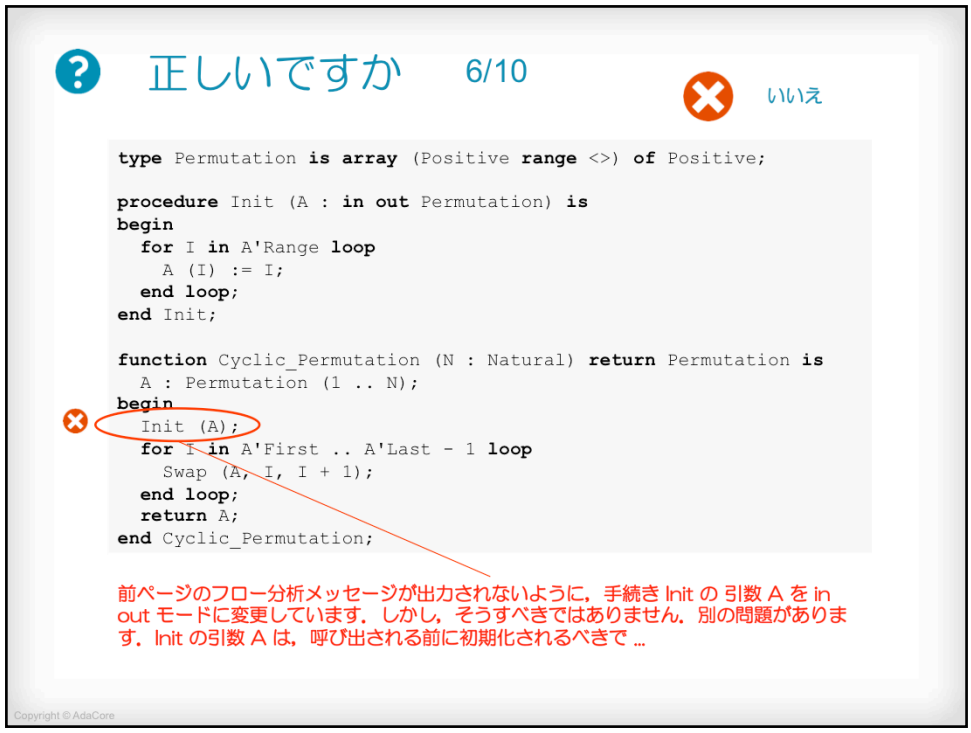
Permutation（入れ替え)は，配列でモデル化されています．I番目のインデックスの要素は，Permutation I 番目の要素の位置です．手続き Init は，Permutation I番目の要素を，I番の位置におきます．Cyclic_Permuation は，Init を呼び出し，巡回置換（cyclic permutation）が完成するまで，要素を入れ替えます．



これは正しいプログラムです．フロー分析はメッセージを出力します．なぜならば，ループ中で全ての要素が初期化されているかどうかを確認できないからです．これは誤った警告で，安全に無視することができます．

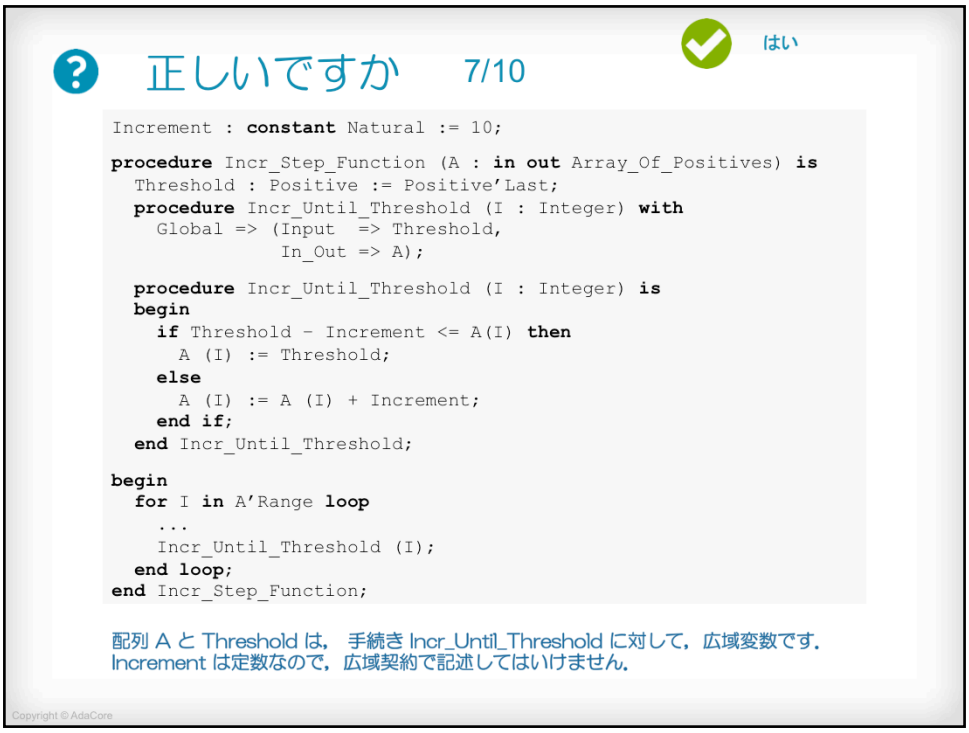


このプログラムは，先のプログラムと同じですが，次の点が異なっています：配列への割り当てにおいてフロー警告がでないようにしている．A のモードを in out に変更している．

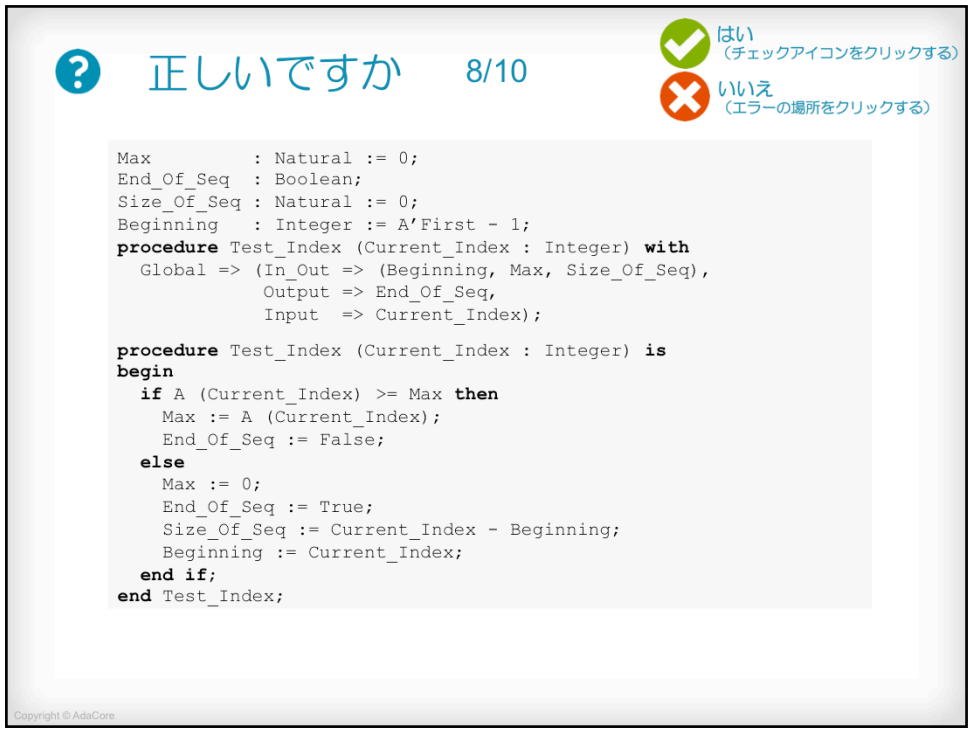


誤った警告がでないように out から in out にパラメータモードを変更することは，良くないアイデアです．Init の仕様を分からなくするだけではなく，手続き呼び出し毎に，A が初期化されていないというメッセージに出会うことになります...

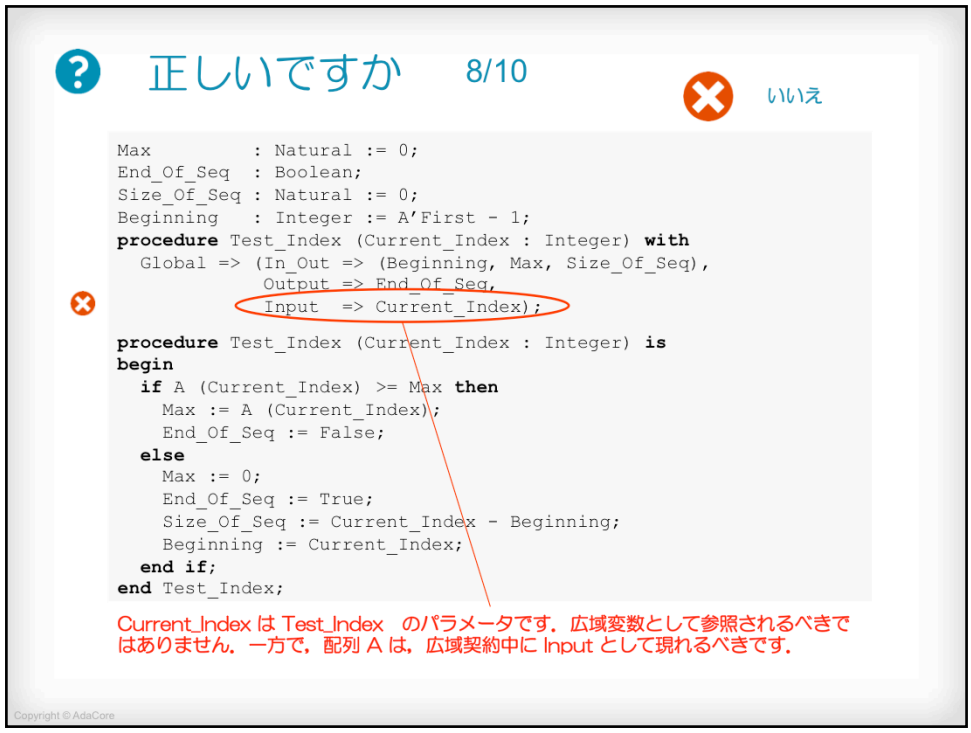
Incr_Step_Function 手続きは，引数として，配列 A をとります．Increment の値を用いて，繰り返しによって全ての配列 A の要素を増加させています．各インデックスごとに，increment を加えたあとで閾値を超えないかを検査しているだけです．広域契約を，Incr_Until_Threshold に対して記述しています．



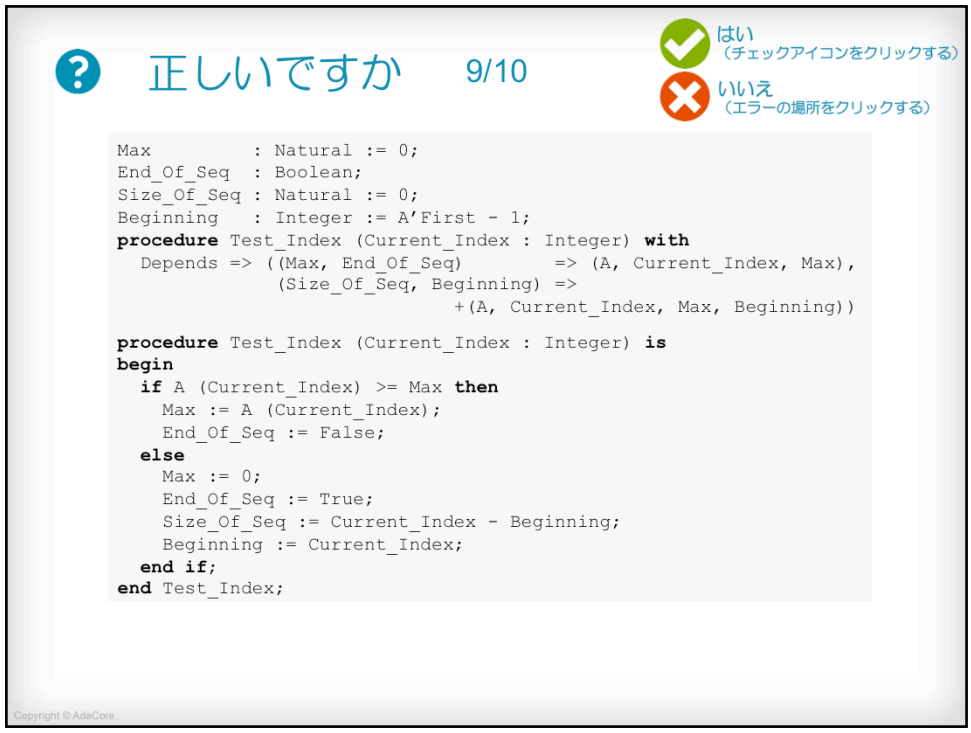
ここでは全てがうまくいっている．特に，広域契約は，正しい．手続き中で読み出されますが，更新されることはない Threshold と，読み出しと更新がある配列 A に関して広域契約があります．A は，内包しているユニットのパラメータであるという事実は，広域契約内部で利用できないということではありません．実際に Incr_Until_Threshold の広域契約です．Incrementは，静的な定数であり，ここでは記述していないことに，注意して下さい．



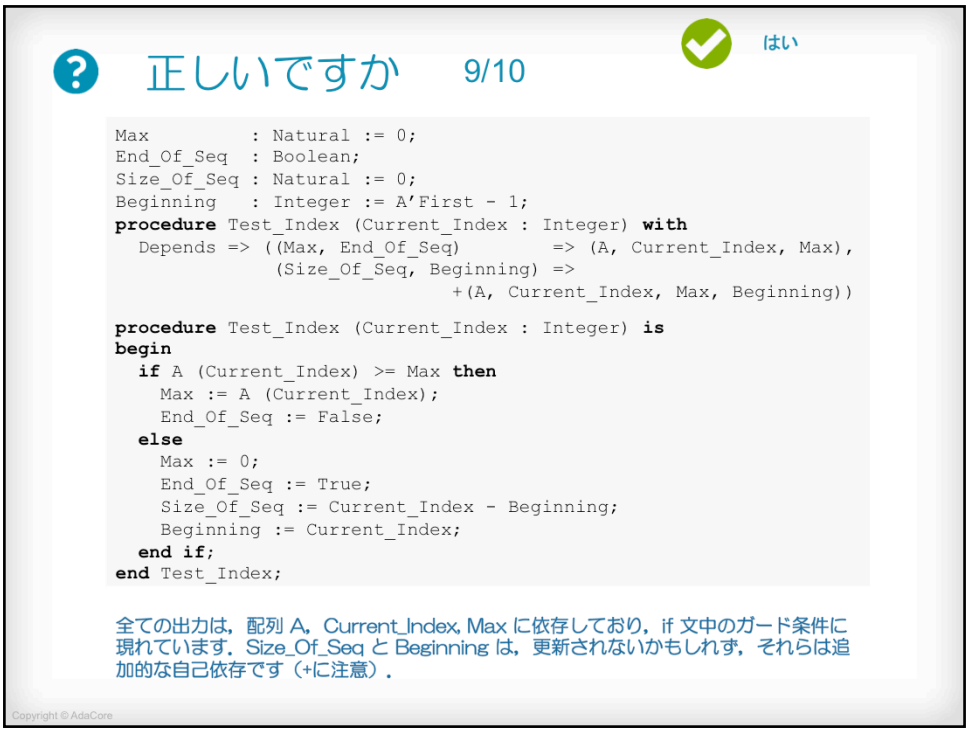
問題4の手続き Test_Index に戻ります．初期化の問題を修正し，Test_Index の広域契約に着目します．これは正しいでしょうか．



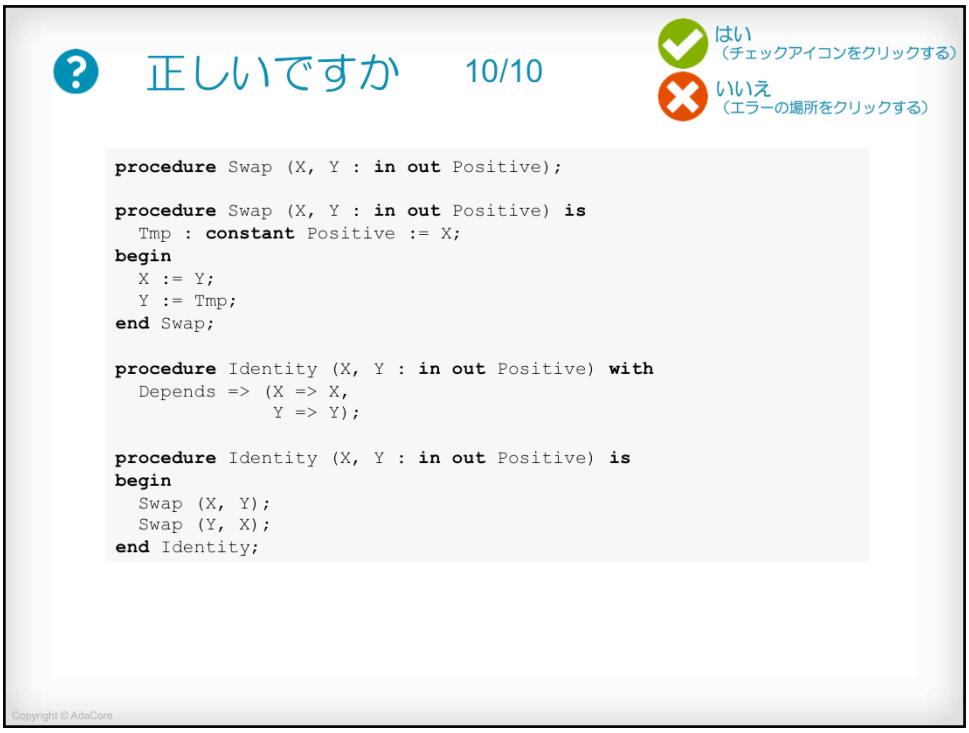
Current_Index は Test_Index のパラメータです．広域変数として参照されるべきではありません．また，もし，配列 A が定数でないならば，広域契約中に Input として現れるべきです．



ここでは，手続き Test_Index の広域契約を依存契約に変更しています．一般的に，依存契約から推定できる（アクセスされる）広域変数は，双方（広域契約と依存契約）記述する必要がないことに注意して下さい．



依存の幾つか，例えば Beginning に依存している Size_Of_Seq は，サブプログラム中での直接的な割り当て式によって分かります．制御フローは，出力全ての最終的な値，条件文中で読み出される変数，すなわち A，　Curent_Index，MAX　に影響するので，全ての依存関係に現れます．最後に，Size_Of_Eq と Beginning の自身への依存は，サブプログラムの実行によって修正されることがないかもしれないという事実から来ています．



サブプログラム Identity は，そのパラメータの値を 2 度交換します．その依存契約は次のようになっています：X の最終値は，X 自身の初期値に依存している．Yも同様である．

このコードは正しい．しかしフロー分析は，Identity の依存契約を検証することができません．実際，Swap にはユーザによる依存契約がありません．結果として，フロー分析は，次のように仮定します：Swap の全ての出力 X, Y は，全ての入力，即ち X, Y の初期値に依存する．この問題を解決するためには，Swap に関して，より正確な依存契約を手動で記述すれば十分です．


ADACORE
university.adacore.com
Copyright © AdaCore