課題名　：　情報科学演習 D　課題 3 レポート

氏名　：　ブアマニー　タンピモン
学籍番号　：　09B20065
メール　：　u946641i@ecs.osaka-u.ac.jp

提出年月日　：　2022 年 12 月 15 日

1　システムの仕様

この課題はコンパイラの三つ目のフェーズ、意味解析器 (Checker) を作成する。概念としては変数表と関数表を作成し、毎回それらを呼び出すと定義するかを確認し、意味的に誤りがあるかどうかを確認する。意味解析器のメソッドは「Checker.run(String)」で、構文解析とともに意味解析を行う。String は字句解析した ts ファイルのファイル名となる。ファイル名から入力ファイルをみつけて確認し、構文解析で間違いがあるなら見つけた行 X を「Syntax error: line X」で標準エラーに出力し、意味解析で間違いがあるなら見つけた行 X を「Semantic error: line X」で標準エラーに出力する。ファイルが存在しない場合は「File not found」を標準エラーに出力する。但し、問題がない場合はそのまま「OK」を標準出力する。

2　課題達成の方針と設計

課題 3 は課題 2 の構文解析器から意味解析を追加するプログラムである。課題 2 のプログラムの構成は課題 1 に切り出されたトークンの列を ENBF 式で構文を確認する。非終端記号が他の非終端記号、または終端記号から作られるため、非終端記号が関数に作成し、それぞれ中にある構成を確認する。あるはず記号がない場合は throw で例外 SyntaxError を投げる。今回は意味解析でエラーが見つかると例外 SemanticError を投げる。

まず、変数の管理を説明する。変数は有効範囲ごとに変数表を作られ、まだ有効ならスタックに保存される。図 1 で見れる通り、オレンジ色はプログラムヘッダーで定義される変数なので先にスタックに入れ、プログラム終了まで削除されない。また、動的な管理で他のブロックに入る直前に新しい変数表が作成される。そのため、他のブロックと同じ名前でも二重定義にならない。検索するや型検査するときは STACK の POP 順に全部の表を確認するため、複数の同名変数の場合は最新で有効な記号表で検査させ、手続き内にプログラムヘッダーに定義された変数を利用可能である。また、手続きの範囲に出ると、手続き内に定義された変数表が消される。型検査について、Pascal には'int','char','bool','array/(型)' という 4 つの種類でわける。基本なルールとしては条件文は bool でないといけないし、関係演算は左辺と右辺が同じ型でないといけないし、関係演算の結果は bool である。加法演算や乗法演算はすべて被演算子が同じ型でないといけない。変数は変数表より型がわかる。それの管理が確認できるように非終端記号表が式の型を返して、条件があってるかどうかを確認する。

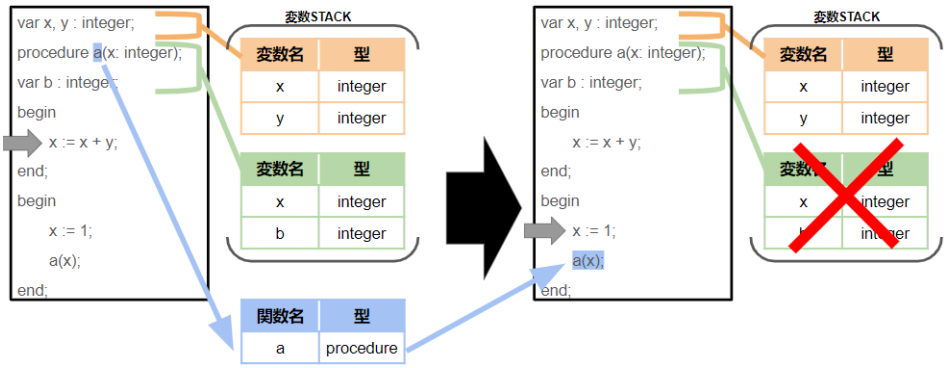

図 1: 課題 3 の仕組み

3　実装プログラム

意味解析器 (Checker) は構文または意味のエラーが見つかると、「SyntaxError,SemanticError」という例外を投げて、エラーの行列を出力する。まず、この課題は課題 2 の続きであり、構文解析器を作成終了と前提する。そのため、意味解析を作成するためには構文解析器のコードの追加としてになる。この課題の問題は大きく 3 つの部分に分けられる。それは「変数表と関数表の作成」「変数表での処理」「関数表での処理」である。

3.1　二分木上の作成

変数表と関数表が速く探索するため新しいクラス Binary Tree（二分木）を作成する。このクラスはノードの根で表示する。このノードは４つのデータが保存する。それはノード名、ノードの型、左のノードと右のノードである。Binary Tree をはじめに定義する時は根が null で、ノードがないが、1 個目の情報をノードとして入れるとそれが二分木の根となる。それで次々と来るノードがノード名を木にあるノードと確認し、木にあるノードよりアルファベット順が前ならそのノードの左に行って、アルファベット順が後なら右に行く。二分木のメソッドが大きく 2 つあり、search と insert であり、search は引数としてノード名を入力し、木に存在するならそのノードの型を出力するが、存在しないなら””を出力する。insert は新しいノードを追加することで、ノード名とノードの型が引数として入力される。insert のやり方はまず二重定義を防ぐために search を行って、同じノード名がある場合は木に追加できないため、false を出力する。しかし木に追加できるなら追加し、true を出力する。その二分木により 2 つのオブジェクト変数表 tree と関数表 procedure を Checker.java に作成する。

3.2　変数表での処理

変数の概念はまず Pascal の言語に変数の定義の部分に変数表に入れる。変数の利用部分から情報を収集し、有効範囲ごとに変数表に変数がなければ未定義なのでエラーを発生する。また変数と変数表の型を確認し、整合性がなければエラーを発生する。それぞれの処理は以下に説明する。

3.2.1　変数表の作成

変数表は変数宣言に定義される。Pascal にはプログラムの変数と手続きの変数があり、プログラムの関数はどこでも使えるが、スコープルールにより手続きの変数宣言がその手続き以外には使えない。そのため、allvariable という binarytree を集める stack を作成し、手続きの複合文が終わると範囲内の変数を消す。そのためプログラムの変数と関数の変数が別の表に保存されて、同じ名でも二重定義ではない。検索するときはスタックの pop してから表ごとに検索するため、同じ名前でたくさんあるばあいでも一番中のスコープに保存される。例は図 2 のようにオレンジ色はプログラムの関数で、緑色の関数の変数と別々で保存される。手続きの複合文の変数確認は先に緑色の表を確認して、見つからなかったれオレンジ色の表を確認する。そのため変数 x が緑色に定義される変数を使う。手続きの複合文が終わると緑色の表が消される。

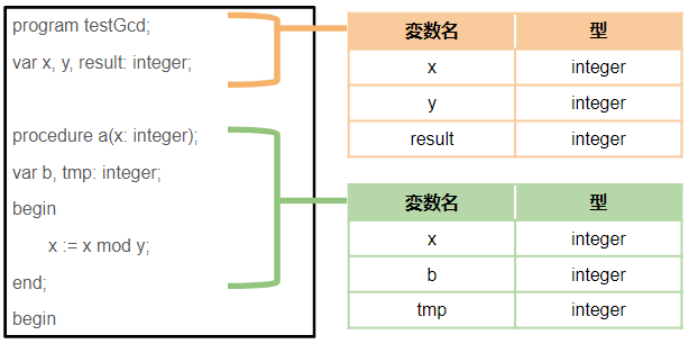

図 2: 変数表の stack の仕組み

変数表は複数あって、allvariable という slack に保存する。図 3 には stack を処理する場所を説明する。stack の中には外層の変数表であり、最新変数表が二分木 tree に保存される。まず、プログラムの変数は変数宣言に定義するため、ブロック関数で、変数宣言に入る直前に「1 NEW」に 2 分木 tree を初期化する。それで変数宣言にプログラム変数が入れられて、変数宣言の関数に出ると「2 PUSH」でその変数表が allvariable に追加させる。それで副プログラム宣言群に入る。副プログラム宣言群の中に関数が定義するたびに引数としての変数と一時変数があり、すべて同じ関数に入れる。まず仮パラメータ関数に変数を処理するため副プログラム頭部の関数に「3 NEW」で新しい変数表を作るために初期化する。また仮パラメータに入ってでるとすぐに「4

PUSH」に stack に入れる。副プログラム宣言関数の中に、変数発言は一時変数を定義するため「5 POP」で仮パラメータを入れられた 2 分木を出し、変数宣言してから「6 PUSH」を入れる。また複合文の直前に「7 POP」で変数表を出して、変数型確認できるようにする。副プログラムが宣言するたびに新しい変数表が「3 NEW」作られ「6 PUSH」に slack に入れられ、使われる直前に「7 POP」に抜き出す。最後に、プログラムの関数が使われる直前に「8 POP」で変数表が slack から抜き出し、複合文に変数と変数の型を確認する。

```
プログラム        =  "program" プログラム名 ";" ブロック 8 POP 複合文 ".".
ブロック          =  1 NEW 変数宣言 2 PUSH 副プログラム宣言群.
副プログラム宣言  =  副プログラム頭部 5 POP 変数宣言 6 PUSH 7 POP 複合文.
副プログラム頭部  =  "procedure" 手続き名 3 NEW 仮パラメータ 4 PUSH ";".
```

図 3: 変数表の stack の処理

3.2.2 変数表での定義確認と変数型確認

プログラムであろか副プログラムであろか変数の型確認は複合文にある。上の変数表管理に確認する関数は二分木 tree または allvariable スラックにある。まず、毎回複合文に変数名が出るとき、searchallscope 関数で二分木 tree にある変数表だけでなく、stack に残る変数表も変数が定義されるかを確認する。存在しない場合は未定義であるためエラーを出す。また、被演算の型検査しないといけないため、変数や式を表す関数に、返し値が void から型の String に変更し、現在の型を表示する。式の型検査が必要な場面が図 4 のようである。左の関数「文、代入文、添え字付き関数」は中に型検査しないといけない。右の関数「左辺、変数、添え字、式、単純式、項、因子」は中に型検査しないといけないし、それで返し値が今の型を出す。

まず、左の関数を説明する。「文関数」に、if と while の直後の式が必ず boolean ではないといけない。それは式関数の返し値を確認し、boolean ではない場合は例外を投げる。「代入文」は中に左辺と式の関数を呼び出し、それらは値を代入するため型が同じでないといけない。異なる場合は例外を投げる。

左の関数には、基礎として「定数関数」は false または true の文字列なら”bool”、整数なら”int”、文字列なら”char”を返す。単純式は加法演算される項を確認し、すべて同じ型ではない場合エラーを出す。しかし、同じの場合はその型を返す。「項関数」も乗法演算されるすべての因子が同じ型である確認し、同じではない場合はエラーを出し、同じの場合はその型を出す。「添え字付き変数の関数」は、行列変数のトークンの管理するため、添え字が数字 int でないとエラーを出す。また、「添え字付き変数の関数」は行列が持っている要素 1 個を表すため、”array/int”なら int を返すなど、行列にある要素の型を返す。「純関数」は変数により、それらの型 int,char,boolean または array を返す。最後に、その他の関数は中に呼ぶ関数の返し値をそのままリターンする。

```
文          = 基本文 |                                  左辺          = 変数.
              "if" 式 "then" 複合文 "else" 複合文 |     変数          = 純変数 | 添字付き変数.
              "if" 式 "then" 複合文 |                   添字付き変数  = 変数名 "[" 添字 "]".
              "while" 式 "do" 複合文.                   添字          = 式.
代入文      = 左辺 ":=" 式.                             式            = 単純式 [ 関係演算子 単純式 ].
                                                        単純式        = [ 符号 ] 項 { 加法演算子 項 }.
                                                        項            = 因子 { 乗法演算子 因子 }.
                                                        因子          = 変数 | 定数 | "(" 式 ")" | "not" 因子
                                                        定数          = 符号なし整数 | 文字列 | "false" | "true".
```

図 4: 変数表での変数型確認

3.3 関数表の処理

関数表は procedure と呼ばれる二分木である。図 5 は関数表に関わる構文である。まず、関数表を追加する場所としては副プログラムを呼ぶ時で、副プログラム頭部に「手続き名」が識別子と確認できたら関数表 procedure

に入れる。識別子ではない場合はエラーを出す。毎回入れる前に探索が行って、既に procedure に存在する場合は二重定義であるためエラーをだす。それで、その副プログラムが手続き呼び出しの部分に呼ばれるため、「基本文の関数」に次のトークンが手続き呼び出し文で確認できる場合、次のトークンが手続き名であるためそれを procedure 二分木に探査して、””がでる場合は未定義で副プログラムとして入ってないためエラーを出す。

```
副プログラム頭部    = "procedure" 手続き名 仮パラメータ ";".
基本文              = 代入文 | 手続き呼出し文 | 入出力文 | 複合文.
手続き呼出し文      = 手続き名 [ "(" 式の並び ")" ].
```

図 5: 関数表の定義確認

4　考察と工夫点

4.1　2 分木の作成

記号表は線形検索、ハッシュ検索、2 分検索など様々な検索方法があり、それぞれの検索により処理時間が異なる。線形検索は検索時間が O(n) であるため、検索には最適ではない。また、ハッシュ検索はよく分散できればもっとも速く O(1) で検索できるが、ハッシュ関数が必要し、最低なケースで変数名がすべて同じハッシュ値を持つ場合はまた O(n) になるため、安定な検索方法ではない。2 分検索は毎回新しいノードを入れるときソート時間 O(log n) がいるが、探索時間も O(log n) であるため探索が多いコンパイラには役に立つ。また、偏差がある変数が来ても平衡二分探索木で作れば検索時間がいつも O(log n) なので、安定な検索方法である。

4.2　searchallscope メソッド

スコープルールにより、それぞれの変数が定義される場所によって有効範囲が異なる。そのため、同じスコープの変数が同じ型変数表に保存され、各ブロックの入る直前に変数表が定義され、ブロック内に定義される変数がその変数表にいれる。ブロックに出ると無効範囲になるためその変数表が消される。しかし、ブロック内の利用変数はブロック内に定義される変数に限らずため、ブロック内の定義変数が優先される。そのため、searchallscope メソッドは全部の有効変数を確認し、見つけたらその変数の型を出し、見つからなければ””をだす関数であり、その関数は一時変数として stack が定義され、最新の変数表を tree に POP して検索し、見つからない場合は一時 stack にいれる。それで allvariable の stack からまた変数表を POP し、検索し続ける。そのため、見つかるまで変数表が最新から最初の順に確認され、POP されても一時の stack に保存されるため検索し終わるとまた allvariable の stack に入れられる。

5　感想

このプログラムを書いた時に、言語 A の第 5 資料を確認すると、課題 2 に作った関数が void 関数で前提されて、void から型を返すとオススメされた。しかし、私の課題 2 の関数がすべてトークン番号を返し、関数が void 関数ではないことを気づいた。それは課題 2 のプログラムがある時に、ちゃんと資料を読んでないことである。そのため、時間かかって課題 2 を訂正し、currenttoken が field method に入れて、全てが void 関数に変更した訂正前にそのまま進めばいいではないかと何度も悩んだが、。先生がオススメ方法は課題 2 からもっとやりやすいので訂正終わるとプログラミングするのがとても楽になった。これからは（日本語が多い場所の）資料を見逃し癖をちゃんと注意して、ちゃんと読むように練習したい。また、プログラムの構成よく、キレイに書くことの大切さを感じた。すべてのプログラムが同じ構造で書くと間違いがあるときはデバッグしやすいので、これからキレイなコードを書けるようにたくさん練習したい。