Checker

学生番号 09B20083 吉崎　響

2022 年 12 月 16 日

1　外部仕様

Pascal 風言語で入力されたプログラムが構文的に正しいか、また意味的に正しいか検査するプログラムである。構文的に正しいというのは、Paser と同様の定義である。意味的に正しくないとは資料 p.25　意味的な誤りに従う。構文エラーがあった場合は、標準出力に”Syntax error: line XX” XX は最初に発見したエラーの行番号。意味的なエラーがあった場合は、標準出力に”Semantic error: line　XX” XX は最初に発見したエラーの行番号。エラーがない場合は、標準出力に”OK”を出力

2　設計方針

2.1　全体

意味解析を行うにあたって課題 2 の paster で作成した構文木を利用する。構文木は EBNF をベースに作成している。(1 つの単語　変数名等につき 1 つのノード　”if”などの意味のない終端記号はカット) 構文木を利用することによって複数回に渡ってプログラムを探索できるので、処理別に機能を分けることができる (詳しくは考察に記述)。今回は意味解析の機能を 1. 記号表を作成する 2. 意味的エラーをチェックするの 2 つの機能に分割した。解析方法は作成された木のノードを上から順に深さ優先探索で進めていく。命令は子供のノードに行く前、子供のノードから帰ってきた際の 2 つのタイミングで行える。

2.2　記号表作成

2.2.1　記号表構造

Cheker を機能 1 と 2 に分けたため、機能 2 で記号表を活用するため記号表を静的に利用できるよう工夫する必要がある。記号表はどのプログラム (メイン、その他の副プログラムなど) のものであるかわかるように作成すると機能 2 でチェックがしやすくなる。

例　　参照されている変数が宣言されていないエラーを検出する場合

副プログラム A が変数を利用する際には、副プログラム A とメインのどちらか一方ではその変数は宣言されていなければならないのだが、変数表をプログラム別に分けていると、どの記号表をチェックすればいいかわかりやすくなる。

作成した記号表は　変数名 型　配列型かどうか の 3 つの情報から成るまた ● 重複して宣言されている変数があるエラーは記号表作成段階で実装している。記号表を追加する際、現在追加すべき表にすでにその記号があればエラーとしている。

2.2.2　記号表追加

図 1 のメインプログラムや副プログラム A などの記号表自体の名前は手続き名を付ける。メインを除く新たな記号表の作成は副プログラム頭部のノードで行う。理由は副プログラム宣言頭部の下のノードは手続き名のノードがあること、そして副プログラム頭部ノードに帰ってきた後、変数宣言ノードを訪れるからである。

ここで記号表を作れば、他の副プログラムの変数が記号表に入ることはない。なぜなら副プログラム宣言なしにほかのプログラムの変数宣言は行われないからである。メイン文の新たな記号表の作成はブロックノードで、副プログラムの記号表の作成は副プログラム宣言頭部で行う、ブロックノード後に変数宣言にたどり着き、変数宣言ノードやその子供を処理した後に副プログラム宣言群となるためメイン文の記号表が最初に作成される。こうすることで記号表に記号を追加する際はどの記号表に追加する必要があるのか考慮しなくても最新の記号表に追加すれば良くなる。現在の記号表に記号を追加するには変数宣言の並びのノードで探索順序を　型ノード、変数名の並びのノードの順で探索する。型を特定し、変数名の並びノードで子供のノードを参照し変数名を獲得し、記号表に型とともに登録する。型を先に探索することで変数を登録する際に型がわかっている状態になり登録処理が簡単になる。また変数名のノードではなく変数名の並びノードで子供のノードを参照して変数の名前を取ってくるのは、変数名ノードは式や変数のノードの子供にも含まれているため、変数宣言ではなく式の場合でも表に格納してしまう。そこで、変数宣言でしか登場しない変数名の並びノードで登録を行うことで宣言された記号のみを獲得している。

2.3　意味的エラーをチェック

2.3.1　式の評価方法

多くのエラー項目では式が意味的に正しいか、式の結果的な型は何であるかの 2 つが必須である。まず式が意味的に正しいかどうかとは式に含まれる非演算子同士や演算子と非演算子の型が同じであるかということである。そこで式の型をチェックするための関数を用意する。関数では最初に入力された型を式の型とし、以降異なる型が入力されるとエラーを返す。また関係演算子があった場合は、関係演算子があったという情報を保持する。式の型を求められると、関係演算子がない場合は最初に入力された型を返し、あった場合は”boolean”を返す。別の式の評価を行う際は関数を初期化する必要がある。関数の初期化は式ノードを訪れた際に行う。ただし、それだけでは、連続で式が呼び出されると、式の情報を失ってしまう。
例 if i >(j+1)
この場合、関係演算子の後が式とみなされ、関数が初期化され、結果式の型が integer になってしまう。そこで履歴保存機能を追加した。因子と添え字の 2 つノードは式ノードよりも上の位置にも下の位置にも来るくることがある。つまり、式の途中で式を呼び出す。そこでその 2 つのノードの場合には子供のノードに行く前に履歴保存を行い、帰ってきた際にロードすることでこの問題を解決した。保存できる履歴の数に制限はない。この関数を利用してチェックを行う。式に出てくる要素は not と変数と定数と加法演算子、乗法演算子、関係演算子である。つまり、式ノードの下のノードにこれらが出てきた場合に型を渡せばいい。演算子の場合は加法演算子は”or”であれば”boolean”それ以外なら”integer”また乗法演算子は”and”であれば”boolean”それ以外であれば”interger”を返す。関係演算子が来た場合は、その後に必ず単純式が来るが、この単純式は前の単純式と型が同じでなければならないため、関数の初期化は行わずチェックを続行することで式全体の評価ができる。

2.3.2　各エラー項目実装方針

- 参照されている変数が宣言されていない.

純変数ノードまたは添え字付き変数ノードで子供のノードから変数の名前を獲得し変数が使用されているプログラムの記号表とメイン文の記号表 (宣言されているのがメインの場合メインの記号表のみ) を参照し、記号表に登録されていなければエラーを出す。この処理を純変数または添え字付き変数のノードで行うのは変数ノードは変数宣言では登場しないからである

- 代入文の左辺に配列型の変数名が指定されている.

代入文ノードの下で出てくる純変数または添え字付き変数ノードは型のチェックが行われており、純変数のノードで子供のノードから変数名を獲得する。獲得した変数が記号表で array であった場合にエラーを出す。同様に添え字付き変数のノードでは獲得した変数が記号表で array ではなかった場合エラーを出す。

- 添字の型が integer でない.

添え字の下のノードは式ノードである。よって添え字のノードに戻ってきたときに式をチェックする関数で型

を確認し、integer でなければエラーとする。
• if 文や while 文の式の型が boolean でない.
if と while の下のノードは式である。よって if と while のノードに戻ってきた際に式をチェックする関数で型を確認し、boolean でなければエラーとする。
• 演算子と被演算子の間で型の不整合が発生している.
式のチェックの際にエラーを検出できる。
• 代入文の左辺と右辺の式の間で型の不整合が発生している.
代入文の下のノードは変数と式である。下のノードを参照することで変数の型を獲得し、式のノードから帰ってきた際に式の型と変数の型を比較する。一致しなければエラーを出す。
• 手続き名が，副プログラム宣言において宣言されていない.
手続き呼び出し文で子供のノードを参照することで手続き名を獲得する。手続き名と一致する名前の**記号表**がなければエラーを出す。

3 実装プログラム

3.1 全体

Visitor モデルを参考にし木構造を利用した解析を行った。作成した木構造のノード class は、ノードの名前、ノードの単語 (終端記号)、ノードの行、子供のノードが入った arraylist children から成る。子供はノード.children.get(0) のように呼び出せるので、葉まで繋がる木構造になっている。Class で分類ではなくノードの名前で分類を行っている点が Visitor モデルとは異なる。ノード class は、accept 関数を持つ。accept 関数は Visit クラスを引数とし、Visit に自身を引数として渡すことで Visit がノードの情報を獲得できるようになっている。Visior に木構造の頂上 (今回なら program のノード) を渡すと、Visitor ではノードに対する処理を行った後、ノード.children.get(0).accept(this) によって子供のノード accept 関数を呼び出し、子供のノードを訪れる仕組みになっている。ノードの種類によって処理を変更できるように、swich 文を利用し各ノードの名前で分類して個別によりを行う。

例

```
swich(ノードの名前) {
  case プログラム :
       プログラムノードを訪れた際の処理
       for(int i=0;i<methodtree.children.size();i++) {
  methodtree.children.get(i).accept(this);
}
        プログラムノードへ戻ってきた際の処理
   break;
  case ブロック :
        同様
  break;
 }
```

作成したクラス Visitor.make_tabel.visit と Visit.check_visit がそれぞれ記号作成とエラ―チェックを行うメソッドである。2 つを Visit の継承クラスとして別々に定義することで処理を分断している。

3.2 記号表 make_table_visit

記号表は build クラスで管理している。記号表作成や参照で利用する build クラスの変数とメソッドは以下のものである。

Class table (String,int,int)	型名、arrayどうか(0か１)、単語の行番号
HashMap<String,HashMap<String,table>> variable_list	記号表
String nowprogram	現在操作している記号表の名前
String nowtype	現在追加している記号の型名
int array	現在追加している記号が配列型か標準型か
next_table(プログラム名)	プログラム名の新しい記号表作成
add_name_table(変数名　行)	記号表に変数を追加
standard_type_name(型名)	nowtypeを型名に変更
array_type_name()	arrayを立てる

図１　使用メソッド

add_name_table(String int) は重複した記号を追加しようとすると、多重定義でエラーを出す。追加するマップに対して containsKey を行うことで重複を確認する。meke_tabele_visit では設計方針の記号表追加に従って各ノードに対応する case 文で build クラスのメトッドを呼び出し実現した。方針に詳細な動作が書いてあるためここではメソッドの説明のみとする。

3.3 チェック check_visit

bulidクラス	
check_array(プログラム名、変数名、行番号)	記号表を参照し変数が宣言されているか、型が正しいかチェックする(純変数用)
check_pure(プログラム名,変数名,行番号)	記号表を参照し変数が宣言されているか、型が正しいかチェックする(純変数用)
String return_type(プログラム名,変数名)	記号表を参照し変数の型を返す(配列型か標準型かまでは返さない)
check_procedure(手続き名,行番号)	手続き名が利用された記号表があるかチェック
Simple_formalクラス	
int relational	関係演算子の有無を表す
String ideal	式の現在の型を表す
ArrayList<String> preideal	idealの履歴保存用
ArrayList<int> relational	Relationalの履歴保存用
check_type(型名、行番号)	idealと引数の型が一致するかチェックする
ideal_check(型名、行番号)	式の先頭の場合(ideal=null)型名をidealへ代入 それ以外はcheck_type()を呼ぶ

図２　使用メソッド

Simple_formalクラス	
find_reletional()	relationalを立てる
clear_all()	ideal,relationalなどの初期化
String print_formal()	式の型を返す関数　relational=1でboolean それ以外ならidealを返す
inout_formal()	readlnにおける関数評価の停止(エラーを出さない)
sava_informal()	idealとrelationalを履歴に保存
reload_informal()	preidealの最後に追加された値がnullであった場合、idealは不変　relationalを履歴から取り出す　それ以外では relationalが1の場合は“boolean”をそれ以外ではidealをcheck_typeに出す。その後、idealとrelationalを履歴から呼び出し戻す。

[H]

図３　使用メソッド

check_type() は加法演算子と乗法演算子と ideal_check メソッドで呼び出される。ideal_check( ) は式の先頭である可能性がある、符号、変数、定数、not の 4 つのノードで呼ばれる

4 警告追加

waring があるとテストケースが通らないため新たなクラス waring_visit で実装しており課題に影響はない.

4.1 実装方法

Waring の追加を行った。検出項目は以下の 3 つである。

- グローバル変数でプログラム中のどのにも使用されていないものがある
- 仮パラメーターで副プログラム中で使用されていないものがある
- メイン文の式でまだ値が代入されていない変数を使っている

実装にあたり、記号表の変数の情報に、代入の有無、参照の有無、仮パラメーターであるかどうかの 3 つの情報を追加した。代入の有無は左辺ノードで使用された場合に代入を記録する。参照の有無では参照は必ず変数ノードを訪れるため変数ノードで参照されたことを記録する。また変数ノードで登場する変数は宣言時以外のすべて変数である。よって登場した変数が代入されていないメイン文の変数であればエラーを出す。プログラム中で使用されていな変数はメイン文の変数の場合、代入がなければ使われていないと判断する (代入なしで参照した場合は先に警告が出るため)。仮パラメーターで使用されていない変数は参照がない変数で判断する。

4.2 実行結果

一部抜粋して結果を載せる　仮パラメーターを使用していないテストケースがないため、新たに作成した
normal08-10.ts: ”Waring :There are no using i line 2”
semerr02.ts:”Waring :using unassignment variable line 30”
Waring :There are no using fn3 line 4

5 考察工夫 感想

Visitor モデルの利点は、処理を分割できる点である。木を作成していない場合、構文解析と同時進行になるため、処理の分割を行うためには構文解析と同じ処理を行うメソッドを作ったりと処理が大変でありコードが複雑になる。しかし木構造を作成していると、引数として木を参照するだけでよいため、意味解析だけのコードになる。この特徴から処理をクラス別に分割することができる。課題 3 では機能を 2 つしか分割していないがより複雑な処理が求められる課題 4 ではより多くの機能を分割できる。処理を分割することはエラーの原因がどの機能にあるのかわかりやすくなりデバックがやり易くなること、別の機能と独立にコーディングできるため、エラーはその機能単体の問題に落とし込めるなどデザインパターンとして機能の分割は優れていると考えられる。機能分割のデメリットは今回の場合、記号表を動的に利用できない点である。分割をしない場合、副プログラムの終わりが来た場合、その副プログラムの記号表を POP できる。これによって、変数をチェックする際には常に今ある変数表のみをチェックすればよいので記号表の管理、参照が楽になるというメリットがある。今回の課題はデザインパターンについて知る機会ができた。機能の分割のメリットをとても実感できた。また木構造に関して工夫すればよかったという点が多かった。例えば、変数名ノードにおいて変数宣言からやってきたノードであるか、変数からきたノードであるかを区別すれば、子供のノードを参照しなくても区別ができた。木構造を作成する前に、checker の挙動を理解していれば、最適な木構造が作れた。次回からは用途に合うよう工夫した木構造を作ることを意識したい。