Parser

吉崎　響

2023 年 11 月 3 日

1　外部仕様

Pascal 風プログラムを字句解析した結果である ts ファイルに対して構文的な誤りが含まれるかをチェックするプログラムである。解析した結果、誤りがなければ標準出力に”OK”、誤りがあれば標準出力に”Syntax error :line ”+該当の行番号を出力する。関数は Paster.run(String) であり、解析する ts ファイルの指定は第一引数で指定する。出力するエラーとはすべてのエラーではなく ts ファイルを先頭から読み込み最初に誤りを含む行番号である。指定されたファイルが見つらない場合は、標準出力に”File not find”を出力する。

2　設計方針

2.1　解析

構文解析を主に LL(1) を利用して行った。また AST を作成しており、作成のために指導書 p.15 の EBNF をベースに EBNF の 1 つの名詞 (変数名や式など) に対して対応するメソッドをすべて作成し、メソッドは対応する名詞が左辺に含まれる式の右辺を解析する。メソッドが対応するメソッドを読み出し解析が進められる。解析内容は主に 3 つである。ただし、1 つの EBNF が 2 つあるいは 3 つの内容を同時に包含する場合もある。
1. 終端記号の確認
2. 非終端記号に対応するメソッドの呼び出し
3.First 集合によるメソッド判断
1. は終端記号が構文的に正しいか確認する。構文エラーは終端記号が一致しない場合にエラーとなる。(2,3) は基本的にエラーは出さない。
例: プログラム = ”program” プログラム名 ”;” ブロック 複合文 ”.”.
プログラムに対応するメソッドが終端記号”program”を確認する。一致すれば次の解析へ、一致しなければエラーが発見される。
2. は単純に対応するメソッドを呼び出す。このとき選択肢が 1 つ (対応するメソッドが 1 つに絞れる) であれば First 集合の確認は行わず進める
例: プログラム名 = 識別子.
この場合単に識別子に対応するメソッドを呼び出せばいい
3. は呼び出すべきメソッドの候補が同時に複数あった場合にどのメソッドを呼び出すべきか、First 集合の違いを利用して決定する。
例: 型 = 標準型 | 配列型.
標準型 = ”integer” | ”char” | ”boolean”
配列型 = ”array” ”[” 添字の最小値 ”..” 添字の最大値 ”]” ”of” 標準型.
型が標準型、配列型のいずれのメソッドを呼び出すべきであるかは、次に読み込むトークンで判断する。例の場合であれば、標準型の First 集合は (integer,char,boolean)、配列型は (array) である。2 つの First 集合には一致する要素がないので、それぞれの要素が来た場合に対応するメソッドを呼び出せばよい。指導書

p.15 の EBNF では基本文 = 代入文 | 手続き呼出し文 | 入出力文 | 複合文
において代入文と手続き呼び出し文の First 集合が一致している。このような場合は単純には解析できない。この場合おもに 2 つの手法がある。
１つは EBNF を変更する方法である。これについては今回使用していないので考察に記述する。2 つ目は、LL(2) を使用する方法である。
代入文と手続き文は First 集合では区別できないが、入出力文と複合文は区別できるため分類し、代入文または手続き文の場合にだけもう 1 つ先のトークンを見ることで区別する。代入文は”[”か”:=”手続き文は”(”か無しである。2 つの要素に重複がないため LL(2) で判断できる。

2.2 木

木が含む情報はノードの名前、終端記号があった場合その終端記号 (葉のみ)、下に延びる枝のリストを持つ。より抽象的な木を作成するため、意味を持たない終端記号などは取り除き、また意味が欠落しないように、対応する名詞 (変数の並びなど) すべてのノードを作成した。
例:”program” プログラム名 ”;” ブロック 複合文 ”.”.
の場合以下のような木になる。”program”のような特に役割のない終端記号は木に保存していない。(識別子のようにノードがただ１つ終端記号のみを持つ場合だけ保存している) プログラム名は識別子しか持たないことが決まっているが、ここでプログラム名というノードを作成していないと、なんの識別子であるかわからなくなってしまう。

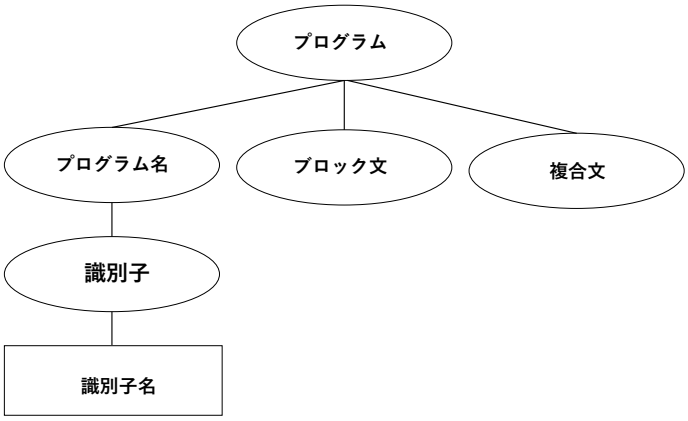

図 1: AST

3 実装プログラム

3.1 ts ファイル

ファイルを 1 行ずつ読みながら解析するのではなく、ファイル全体を 1 度に保存した。このほうがコードが簡潔になるためである。またファイルの情報をそのまま配列に保存するのではなく、要素ごとに分割したクラスを用意し、その class の ArrayList という形で ts ファイル保存する。要素は実際のトークン、トークン id、行番号から成る。字句解析器上でのトークン名は構文解析では必要ないので省略した
例: program SPROGRAM 17 1 の場合
クラス名.code=”program”, クラス名.id=17, クラス名.line=1
また ts ファイルの操作も 1 つの class のみで完結するようになっており、ts ファイルに関する ArrayList を 1 行進める　クラス名.next(),　同じ行の情報を返す　クラス名.same(),　1 つ前の情報を返す　クラス名.back() を用意し、この 3 つの関数だけで ts ファイルの操作を行う。

3.2 エラー出力

エラー出力用の class を用意した。エラー class では初めてエラーが出た際の行数とその単語を保存する。エラーが初めてかどうかはフラグで管理している。エラーがでた単語を用意したのはデバックの際にどのトークンでエラーが出たかわかるようすることでデバックが簡単になるためである。

3.3 木構造

木構造を作成するために tree というクラスを定義した。tree はノードの名前 (プログラム名であれば programname) と終端記号用の変数 (終端記号でなければ null) そして下のノードを保存するために tree 型の ArrayList children を持っている。すべてのメソッドは tree 型で定義されており、メソッドの最後では必ず自身を return することになっている。自身を return するとは、自分のノードの名前、終端記号があれば終端記号をなければ null、そして自身の下のノードに関する tree 型の ArrayList を返す。ArrayList には呼び出したメソッドが return した tree が保存される。こうすることで、構文解析ではメソッドの呼び出し元が親になるため、子供が終了するまでメソッドは終わらない。つまり、図のように木構造でいう葉の部分から順にメソッドが終了し、すべて子供が return を行ってから親メソッドが終了するので木構造が成立する。

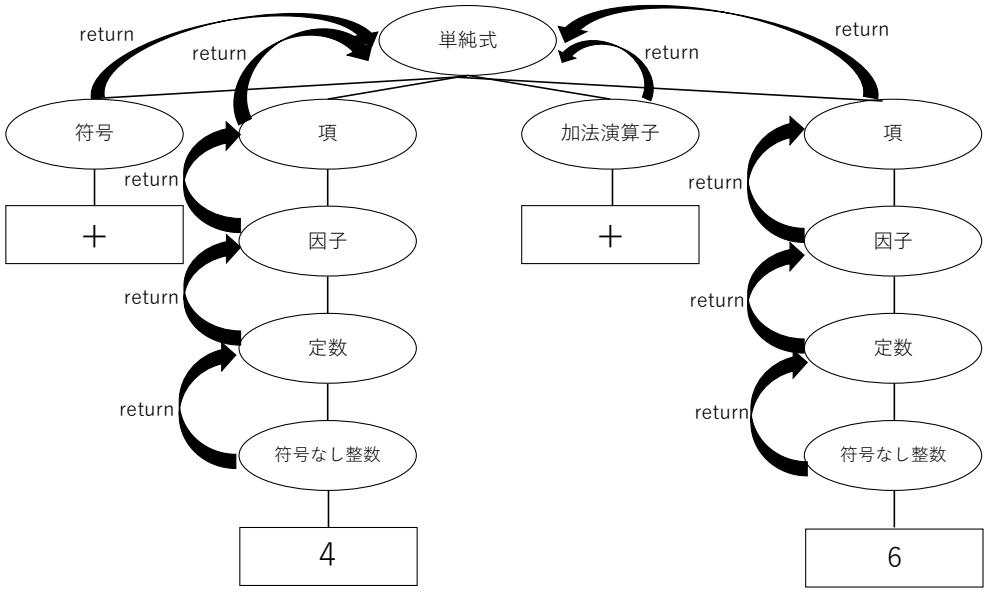

図 2: return による木構造

3.4 解析

入出力メソッドの概要を下に記す

```
1 入出力メソッド(){
2               Array List<tree> children;
3               if(トークンがreadlnである){
4                       next()/*次のトークンへ*/
5                       if(トークンが”(”){
6                               next()
7                               children.add(変数の並びメソッド)
8                               if(トークンが”)”ではない) {
9                                       next()
10                                      error処理
11                              }
12                      }
13              }
14              if(トークンがwriteである){
15                                      同様
16              }
17          return new tree(“入出力”,null,children);
18 }
```

図 3: 入出力メソッド

右辺に非終端記号の候補が複数ある場合は First 集合で決定する (今回は終端記号で判断している) 3 行目のように readln が来た場合、後には 0 回か 1 回の”(” 変数の並び ”)”が来る 0 回か 1 回の場合はその部分の処理を if 文で囲みこむ。そうすることで”(”なければ処理をスルーできるので 0 回の場合でも対応できるまた 1 回以上の繰り返しであれば do-while 文で対応できる
”(”があればその後”変数の並び”が来るので変数の並びに対応するメソッドを呼び出し結果を ArrayList に tree として登録する。8 行目では”)”がなければエラーを出す。このように構文エラーは終端記号のチェックで判明する。最後に tree を return して入出力メソッドは終わる。
各メソッドはこのように作成されている。EBNF に対応するように各名詞のメソッドを作成することで program のメソッドを呼び出しすだけで解析は実現する。代入文と手続き文の区別に関しては LL(2) 解析を行っている。これは back() を利用して実現しており、こうすることで適切なメソッドが呼び出せるのでノードの名前が EBNF に沿ったものになる。

4 考察工夫

終端記号ですべてのエラーを出せばよいということに途中から気づいたため、一部 First 集合によるエラー検出を含んでしまった。この助長を削除するとプログラムの挙動が追いやすくなったり、コードが簡潔になる。今回の EBNF では LL(2) を使うことで文法の書き換えが必要なかった。左再帰性は含まれていなかったためである。LL(2) を行わない場合は EBNF を変更する必要がある。書き換えとしては以下のようなものがある。
基本文 = 代入手続き呼出し文 | 入出力文 | 複合文
代入手続き呼び出し文＝識別子 ([” [”添え字”] ”] ”:=” 式 | [” (”式の並び”) ”])
このように書き換えることで LL(1) で解析可能になる。今回 LL(2) を利用した理由としては、上記のような EBNF では代入文も手続き文も先頭は識別子であるため、LL(1) で解釈すると識別子というノードのみ作成される。しかし同じ識別子でも変数名であるか手続き名であるかという点で意味が変わってくる。意味が判明しないと意味解析で木が機能しなくなる可能性がある (そうでなければ EBNF 上で区別されない)、

一方 LL(2) であれば、識別子として解析する前に、代入文か手続き文か判明するため、なんの識別子であるか判別できる (例: 手続き文であれば識別子の上に手続き名とうノードができる)。EBNF を大規模かつ丁寧に改変すれば意味を保存しつつ LL(1) で解析することも可能であろうが、複雑でわかりにくくなるため今回は避けた。木構造を作成するためには各名詞に対してメソッドではなくクラスを定義することでも作成できた。(例: class programname) しかし、ArrayList を選んだのは、EBNF は 1 回以上の繰り返し (回数は不明) を含んでおり、クラスのみでは対応できないため、ArrayList を一部使わなくてはならなくなる。すると統一感がなくなりわかりにくくなるためである。