情報科学科演習 D
課題 2

基礎工学部情報科学科ソフトウェア科学コース
学籍番号: 09B20001

イグレシウスエドゥアルド

2022 年 11 月 18 日

1　システムの仕様

1. 構文解析器は Parser.run(String) から実行される。
2. プログラムの入力としては構文をチェックするためのデータパス（ts ファイル）である。
3. 与えられたデータパスがない場合は”File not found”というエラーメッセージを出力して、終了する。
4. 与えられたデータパスを見つけた場合、
   （a）構文が正しい場合、標準出力に”OK”を出力する。
   （b）構文のエラーがある場合、そのエラーの見つけた行の番号と共に、”Syntax error : line { LineNumber } ”というエラーメッセージを標準エラーに出力する。
   （c）与えられたファイル内に複数のエラーがある場合、その最初のエラーのみを出力する。

2　プログラムの方針・設計

本課題のプログラムの設計は大体図 1 のようになる。

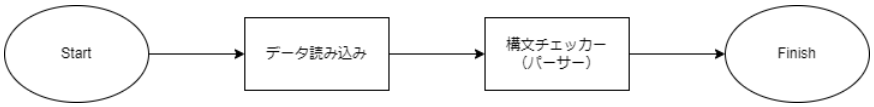

図 1　プログラムの流れ

2.1　入力ファイルの解釈

本課題の入力としては ts ファイルなので、最初に、その ts ファイルを解釈して、実際に扱えるデータの配列に格納する。ts ファイルの各行は解釈された pas ファイルのトークンの pas ファイル上のトークン名、字句解析器のトークン名、トークン ID、と行番号を示すので、各行に対してその四つの情報を持っている新しいオブジェクト（Token）を定義する。

ts ファイルから一文字ずつ読み込み、”\t”の場合、次の要素に移して、”\n”の場合、次のトークンに移す（図 2 参考)。4 つの情報をもらったら、一つのトークンとして格納して、リストとして扱う。

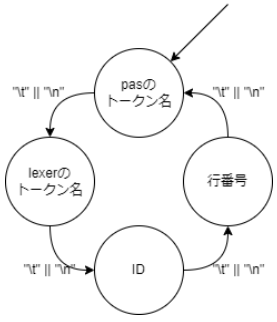

図 2　ワード区切る方法

2.2　パーサー

トークンのリストが得られた上で、本課題（パーサー）を作るために、基本的に LL(1) 構文解析を使う（LL(2) が一箇所に使う)。構文ルールは指導書の 15 ページに従う（何個か使わないこともある)。

また、2.1 章で得られたトークンのリストを一トークンずつ読み込み、構文をチェックする。基本的に、次の構文はトークンであったら、その時にトークンをチェックするが、次の構文はトークンではない場合、つまり、他の構文に移る場合、その次の構文のルールで次のトークンをチェックする。他の構文をすべてチェックして、正しければ、また、元の構文に戻り、残りの構文をチェックする。

ただし、選択肢 (”|”又は”{}”又は”[]”) がある場合、その一トークンを先読みする必要がある。各々の選択肢のファースト集合を求める。本プログラムではそのファースト集合は手動でつまり、自分で勝手に定義するという形になる。いずれのファースト集合に含んだら、その構文要素に移るが、どの選択肢でも含んでない場合、プログラムを終了してエラーを出力する。

また、選択肢のファースト集合が同じである場合、左括り出しをする。例えば、

```
文 = 基本文 |  "if" 式 "then" 複合文 "else" 複合文
| "if" 式 "then" 複合文 "else" 複合文 | "while" 式 "do" 複合文
```

の場合は以下のように左括り出しをして、LL(1) 構文解析をする。

```
文 = 基本文 |  if 文 | while 文
if 文 = "if" 式 "then" 複合文 ["else" 複合文]
while 文 = "while" 式 "do" 複合文
```

最後に、本課題ではループが必要な場合、for 文や while 文を使わずに、再帰関数を使っている。

3 実装プログラム

3.1 入力ファイルの読み込み

トークンオブジェクトを定義するために新しいクラスを定義する。その新しいオブジェクトは図 3 に示される。

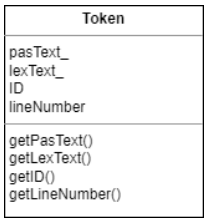

図 3　トークンのオブジェクト

図 3 を使い、また、図 2 の方法に従い、ts ファイルの一文字ずつ読み込み、pasText_, lexText_, ID, lineNumber という順番に四つの要素を逐次的に読み込む。その四つの要素が得られたら、その情報を持つ新しいトークンオブジェクトを宣言して、tokens というトークンリストに入れていく。また、今回の tokens オブジェクトはすべてのメソッドに使うので、run() メソッド内に定義するではなく、static 的にフィールド変数として扱う。

3.2 構文エラーを見つけた対策

構文エラーを見つけた場合、system.exit() を使って、プログラムを強制終了させるのは危険なものであるので、本プログラムでは、それを使わずに新しい例外クラスを定義する。例外クラスは図 4 に示される。

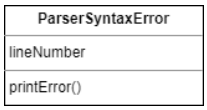

図 4　トークンのオブジェクト

図 4 に示されるオブジェクトを使い、構文エラーが見つかった時に、強制終了をせずに、現時点のトークンの行番号を持つ ParserSyntaxError のオブジェクトを throw する。そのエラーをキャッチするために、パーサーの最初のメソッドを行う前に、try-catch で囲まれる。また、新しい例外を宣言するので、パーサーの各々の関数が throws ParserSyntaxError というキーワードをつける。つまり、

```
run(){
     try{
        program()
    }catch(ParserSyntaXError e){
```

```
            e.printError()
        }
    }
    ...
    program() throws ParserSyntaxError{
        ...
    }
```

という風に設定して、構文に誤りがある場合、その例外クラスを throw する。

3.3 パーサー

指導書の 15 ページに従い、本プログラムではそれぞれの構文要素名をメソッド化して、本プログラムに使う構文要素名とメソッド名は表 1 に示される（本プログラムのメソッド名はデバッグのため、できるだけメソッド名に構文要素名と似ている名前をつけた)。

表 1　パーサーに使う構文要素名とそのメソッド名

構文要素名	メソッド名	構文要素名	メソッド名
プログラム	program()	複合文	complexStatement()
プログラム名	programName()	文の並び	sentenceSeq()*
ブロック	block()	文	sentence()*
変数宣言	varDec()*	if 文	ifStatement()*
変数宣言の並び	varDecSeq()*	while 文	whileStatement()
変数名の並び	varNameSeq()*	基本文	basicStatement()*
変数名	varName()	代入文	subStatement()
型	type()*	左辺	leftSide()
標準型	standardType()	変数	variable()*
配列型	arrayType()	純変数 + 添字付き変数	variable2()
添字の最小値	minIndex()	添字	index()
添字の最大値	maxIndex()	手続き呼出し文	procedureCallStatement()*
整数	integer()	式の並び	equationSeq()*
副プログラム宣言群	subDecGroup()*	式	equation()*
副プログラム宣言	subDec()	単純式	simpleEquation()*
副プログラム先頭	subDecHead()	単純式（後半）	simpleEquation2()*
手続き名	procedureName	項	clause()*
仮パラメータ	tempParameter()*	因子	factor()*
仮パラメータの並び	tempParameterSeq()*	入出力文	inOutStatement()*
仮パラメータ名の並び	tempParameterNameSeq()*	変数の並び	varSeq()*
仮パラメータ名	tempParameterName()		

また、トークンの取得は getToken() というメソッドを宣言して、得られたトークンリストの現時点のインデックスをインクリメントして、次のトークンを取得する。先読みの方は LL(int k) というメソッドを宣言して、インデックスの値を変更せずに、index を k で足したインデックスのオブジェクトを返す（図 5 参照)。ただし、足したインデックスはインデックスの最大値を超える可能性があるので、その時に、ParserSyntaxError を throw する。

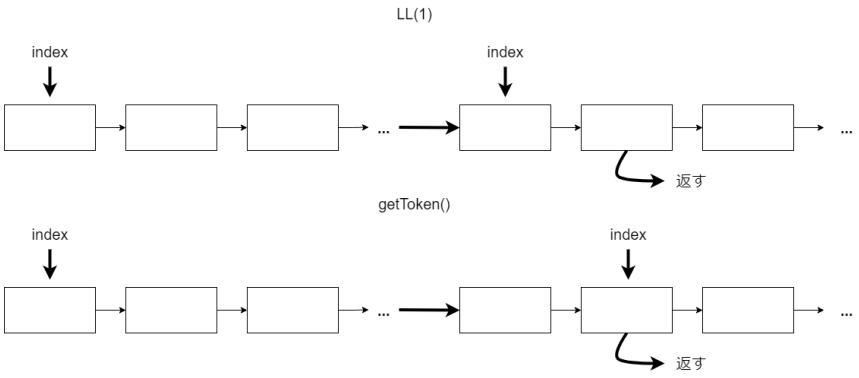

図 5　getToken() と LL(1) の違い

トークンの読み込みの流れは以下のようになる。

- 次の構文はトークンの場合、次のトークンを取得して、同じかどうかをチェックするが、
- 次の構文は構文要素名の場合、その構文要素名に適応するメソッドを呼び出す。
- 次の構文は選択肢がある、つまり、先読み (LL(k)) が必要な場合は各々のファースト集合を求めて、2 章に示された方法に従う。
  - ”|”の場合は各々のファースト集合にチェックして、どの選択肢に含まないとしたら、エラーを throw する。
  - ”[]”の場合は次のトークンをチェックして”[]”の中に含む場合、”[]”の中の構文をチェックする。含まない場合はメソッドが終了する。
  - ”{}”の場合は次のトークンをチェックして、”{}”の中に含む場合はメソッドを再帰的に呼び出す。含まない場合は終了する。

実際に先読みが必要な構文要素名は表 1 に示される星（*) をつけたメソッドである。

本課題では while 文や for 文を使わないので、ループがある時に、再帰関数を使っているが、ある場合ではメソッドを再帰的に行うことができない。例えば、単純式（simpleEquation()）の場合は最初に符号をチェックするが、”{}”の中は符号をチェックしないので、再帰的に行うために、”{}”の中は新しい simpleEquation2() で行う。

4 考察・工夫点

try-catch 文について考察したい。try-catch の try 文の中同じ try-catch 文を使おうとしたら、中の try 文に catch 文でキャッチされたエラーが起こるとしたら、どうなるかを考察する。例えば、

```
run(){
    try {
        test();
    }catch(ParserSyntaxError e) {
        System.out.println("Hello Outer Catch");
    }
}
test() throws ParserSyntaxError{
    try {
        System.out.println("Hello Inner Try");
        throw new ParserSyntaxError("1");
    }catch(ParserSyntaxError e) {
        System.out.println("Hello Inner Catch");
    }
}
```

と設定すれば、以下の出力値になる。

```
Hello Inner Try
Hello Inner Catch
```

つまり、一番奥の try-catch の catch のみが実行されて、外側の catch 文が実行されてないことになる。以上の事実を用いて、LL(k) の構文エラーが実行できるかなと考えた。実行方法は指導書の 15 ページに沿って、選択（”|”や”[]”や”{}”がある場合）try-catch 文を使い、どこまでコードに行ったか関係なく、最初に、パーサーの構文エラーがある場合、その catch 文に戻るので、try-catch 文を始める前に、現在のインデックス値を一時的に格納して、catch 文に移るとしたら、そのインデックスに戻す。ただし、”|”の場合にのみ、すべての選択を確認したとしたら、構文エラーを出す。こちらの考え方では、利点として、指導書の構文に全く沿うことができて、特にファースト集合を求めなくても良いが、構文が発生した時にどの行にそのエラーが行ったか分からなくなるので、デバッグには困難になる。

5 感想

本課題はプログラムがの行数が長すぎて、最初に、プログラムを作り始めるときに、方針などは作っていないので、最初に、プログラムを完成するときに、ほとんどの単体テストに落ちて、そのバグを解決するために時間がかかってしまった。今度よく課題を理解して、方針をよく作ってからプログラムを実際に作りたいと思う。