

## コンパイラ演習資料 LL(1)構文解析器 担当：佐々木晃

### □演習問題 A8

問題番号: A8

課題名：LL(1)構文解析器の作成

**問題 1.** 構文解析の様子を図示する演習。(呼び出し木(call tree)の作成)

(1) 前回配布資料のスライド 2 1 ページのプログラム 5. 1 で実装された構文解析器に次の入力 (トークン列) が与えられた時の、構文解析の様子を示せ。(a + b\*c の例は、スライド 2 1 の右側にある。)

(A) (a + b)\*c (B) a + b\*c + d

(2) スライド 2 3 ページのプログラム 5. 2 は文法 G3' に基づいて実装した構文解析器であるが、これに次の入力 (トークン列) が与えられた時の、構文解析の様子を示せ。(下記注意 2 を参照)

(A) a + b\*c (B) a + b\*c\*d + e

**問題 2.** プログラム課題

前回配布資料のスライド P42. 問題 B(2)。構文解析器については java でプログラムせよ。終端記号 i は、 $i \rightarrow \text{LETTER} \{\text{LETTER} \mid \text{DIGIT}\}$

なるトークンとする。字句解析ルーチンは配布する。(b2.zip)

また、問題 B(1)についてはサンプル配布。(ex8.zip)

**注意 1:** [+ ]などは文法の一部ではなく、構文解析の途中で行う「アクション」であることに注意すること。

(p 2 0 の左側を参照) 例えば、"+ T [+]" は、

- 1, トークン+ を読み、
- 2, 非終端記号 T の部分を構文解析した後に
- 3, + を出力 (表示) する。

という意味である。[+] は上記 3 の部分である。

**注意 2 :**

$A ::= B \{CD\}$

は、 $A ::= B \mid BCD \mid BCDCD \mid BCDCDCD \mid \dots$

のことである。CD が 2 回繰り返される場合は、A からの呼び出し木は、次のようになる。

```
A --- B
   | - C
   | - D
   | - C
   | - D
```

## □問題 2 用の字句解析プログラム (b2.zip)

行読み取り器 CharReader.java

→ nextChar()メソッドで、次の読み取り文字を返す。

字句解析器

→ トークンの種類の整理 TokenClass.java

TokenClass.NOT / AND / OR / IDENT / LPAREN / RPAREN / EOF

→それぞれ、 not, and, or, 識別子, (,) 入力終了 の 7 種類を表す。

→ 字句解析器本体 B1Tokenizer.java

nextToken()メソッドで次のトークンを読み込む。返り値は、トークンの種類 (上記のどれか)。

currentString()メソッドは、直前にトークンを読み込んだ際に切り出された文字列を返す。

Main メソッドの例を参照。

```
public static void main(String[] args) {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    B2Tokenizer tokenizer = new B2Tokenizer(new CharReader(br));
    TokenClass tc;
    while((tc= tokenizer.nextToken())!= TokenClass.EOF){
        //EOF(最後のトークン) まで、1 トークンずつ取り出して処理する。
        System.out.print("tc = " + tc + " : ");
        System.out.println("tokenizer.currentString() = " + tokenizer.currentString());
    }
}
```

上記の入力例

not and abc ( hoge )

出力例

```
line = not and abc ( hoge )
tc = NOT : tokenizer.currentString() = not
tc = AND : tokenizer.currentString() = and
tc = IDENT : tokenizer.currentString() = abc
tc = LPAREN : tokenizer.currentString() = (
tc = IDENT : tokenizer.currentString() = hoge
tc = RPAREN : tokenizer.currentString() = )
```

## □サンプルプログラム。

それぞれのパーサーのクラスにメインメソッドがあるので、使いかたはそれを参考にする。

### ○問題 B(1)のサンプル /ex8.zip

パーサーは以下の2種類(×2 ... WithAction については後述)を用意した。

#### 1. B1Parser1.java, B1Parser1WithAction.java

→左再帰の除去を行った文法をもとにした構文解析器

#### 2. B1Parser2.java, B1Parser2WithAction.java

再帰を繰り返し(正規右辺文法、拡張BNF)に変換したものをもとにした構文解析器

行読み取り器 CharReader.java

→ nextChar()メソッドで、次の読み取り文字を返す。

字句解析器(上記2×2種類で共通)

→ トークンの種類の整理 TokenClass.java

TokenClass.BEGIN / END / SEMI / EOF

→ 字句解析器本体 B1Tokenizer.java

nextToken()メソッドで次のトークンを読み込む。返り値は、トークンの種類(上記のどれか)。

currentString()メソッドは、直前にトークンを読み込んだ際に切り出された文字列を返す。

Main メソッドの例を参照。

### ○サンプルその2

スライド p 21 プログラム 5. 1 (後置記法への変換) を Java で動かせるようにしたもの。

### exParser.zip

パーサー本体は、ExParser.java

## 問題 B(1)の説明(B1Parser.java)

```
S → begin L end | i
L → L ; S | S
```

上記文法は、L に対する生成規則で左再帰があるため、次のように文法の書き換えを行う。

```
S → begin L end | i
L → S L'
L' → ; S L' | ε
```

終端記号  $i$  は、identifier(名前)とする。すなわち  $i = \text{LETTER} \{\text{LETTER} \mid \text{DIGIT}\}$

$\text{VN} = \{S, L, L'\}$

$\text{VT} = \{\text{"begin"}, \text{"end"}, \text{"i"}, \text{";"}\}$

入力テキストの例(1): test.txt (テキストファイル)

```
begin
  stm1 ;
  begin
    stm2 ;
    stm3
  end ;
  stm4
end
```

入力テキストの例(2): test1.txt (内側の end のあとに;がないので parse error)

```
begin
  begin stm1 end
  stm2
end
```

問題 B(1)の文法にアクションを加えた例 B(1)'を挙げる。

```
S → begin [{} L end [ {} ] | i [ i ]
L → L ; [.] S | S
```

入力テキスト(1)の例の場合

{ stm1. { stm2. stm3 }. stm4 }

と出力される。(サンプルは B1Parser1WithAction.java および B1Parser2WithAction.java)