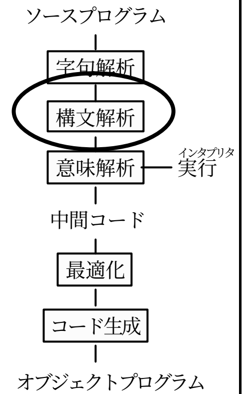
第２回（平成１５年度９月９日）

top-down parserの作り方
tiny Cの概要とデータ構造

筑波大学　佐藤

言語処理系の基本構成

- **字句解析**(lexical analysis): 文字列を言語の要素（トークン、token）の列に分解する。
- **構文解析**(syntax analysis): token列を意味を反映した構造に変換。この構造は、しばしば、木構造で表現されるので、抽象構文木（abstract syntax tree）と呼ばれる。ここまでの言語を認識する部分を言語のparserと呼ぶ。
- **意味解析**(semantics analysis): 構文木の意味を解析する。インタプリターでは、ここで意味を解析し、それに対応した動作を行う。コンパイラでは、この段階で内部的なコード、中間コードに変換する。



課題１

- 掛け算、割り算の優先度を入れたインタープリターを作りなさい。tokenの種類に*や/に対応した演算子が増えることになる。入力として、

  12*3 + 3*4 - 10

  をいれて、正しく実行できることを確認しなさい。

- さらに、括弧をいれた式が正しく処理できるよう拡張せよ。tokenの種類に括弧に対応するものが増えることになる。入力として、

  12*(3+13) – 10

  をいれて、正しく実行できることを確認しなさい。<u>できたプログラムを提出すること。</u>

---

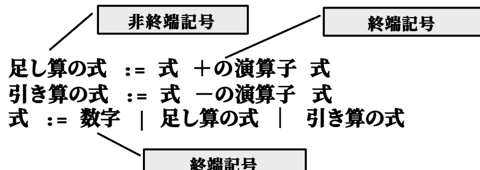
top-down parser

- これから、読み込むべきものを仮定して、それに合致するかを調べていく構文解析法
  - 構文規則の上位から下位に向かって、parserを構成する。
- 構文規則から、直感的なプログラムを構成する。
- 人間にわかりやすい。人手でparserを書く場合に有効な方法。

構文規則のおさらい

- BNFによる数式の構文規則
- 構文規則の最終的な要素＝終端記号（terminal）
- ほかの構文規則によって定義される記号

  ＝　非終端記号(non-terminal)



構文規則の定義

- **定義**
  1. 構文規則は，非終端記号<T>に対して、<T> := e（eは構文規則）で、表現される。これは、非終端記号<T>は、構文規則eによって置き換えられることを意味する。
  2. eは空でもよい。
  3. eは、非終端記号、終端記号、もしくはe1 | e2、e1 e2、e1*のいずれか。e1 | e2は、e1もしくはe2であることを意味し、e1 e2 はe1の次にe2が現れることを意味し、e1*は、e1の0個以上の繰り返しを意味する。

- (...)は、構文規則のまとまりを示す
- e1|e2|e3は、((e1|e2)|e3) と同じ
- e1 e2 e3 は（(e1 e2) e3）と同じである。
- 正規表現と似ている

構文規則と文脈

- 構文規則の左辺が、一つの終端記号<T>だけという文法を考える 。これはすなわち、どのような場合でも

  <T> := e

  の規則を使って、右辺に置き換えることができることを意味し、このような文法を<u>文脈自由文法</u>とよぶ
- e1 e2の間に構文要素<T>が現れたときだけ、置き換えることができるというような文法が考えられるが、このような文法を<u>文脈依存文法</u>と呼ぶ

  e1 <T> e2 := ...

top-down parser

- これから、読み込むべきものを仮定して、それに合致するかを調べていく構文解析法
  - 構文規則の上位から下位に向かって、parserを構成する。
  - <u>再帰的下向き構文解析(recursive decent parsing)</u>
- <A> := a <B> c という構文規則に対し<A>のための構文解析関数readAは以下のように作ることができる

<A> := a <B> b

```
readA()
{
      aを読み込む；
      readB(); /* Bを読み込むための関数を呼ぶ*/
      bを読み込む；
}
```


top down parserの構成法

- 非終端記号<T>に対して、<T>を読む関数readTという関数を作る。
- tokenを先読みしておく。
- <T>:= a bならば、aに対する処理、bに対する処理を書く。

  readT() { aに対する処理； bに対する処理; }
- <T>:= a | bならば、先読みしているtokenを使って、aかbかを選択

  readT() { if(a or b) aに対する処理 else bに対する処理; }
- aが、終端記号ならば、先読みしているtokenがaであるかを判定する。

数式の例

```
<expression> := <expression> <expr_op> <term>
<term> := <term> <term_op> <factor>
<factor> := number | '(' <expression> ')'
<expr_op> := '+' | '-'
<term_op> := '*'
```

```
<expression> := <expression> <expr_op> <term>

readExpr()
{
   readExpr();
   readExprOp();
   readTerm();
}
```

readExpr()が永遠にとまらない！？

左再帰性の問題

左再帰性の問題

- 次の規則を持つ場合にうまくいかない（εは空）
  - 最終的に <T> := <T> ε になるもの

```
readExpr() {
   readTerm();
   while(readExprOp() is OK)
         readTerm();
}
```

  - 以下のように書き換えたのと同等

```
<expresssion> := <term> <expression1>
<expression1> := <expr_op> <term> <expression1> | ε
```


左再帰性の問題

- <T> := b (c| ε)
  - (c| ε)は、cを読むかなにもしないか
- <T>の次に何がくるかによって、cを読むかどうかが決まるので、top-down parser では、処理ができないことになってしまう