第0章 「オートマトン」の復習

0.1 この授業の目的

- プログラミング言語の"文法"を形式的に定める。
    - 理論 … 「オートマトン」（第２Q）
    - 実践 … 「コンパイラ」（後期）

> プログラミング言語の仕様は主に構文・文法 (syntax) に関する部分と意味 (semantics) に関する部分に分けられる。
>
> 構文 … プログラムのかたち・構造に関すること
> どのような文字列がプログラム（あるいは、文・式）であるのか、ないのか？
> 例えば、"`while { c++; }`" や "`1 + return n;`" は C のプログラム（の一部）ではない。
> 正規表現や BNF などで記述する。
>
> 意味 … プログラムの実行結果に関すること、例えば、
> "`int i, n = 0; for (i = 0; i < 10; i++) n += i;`" を実行したあと、 `n` は 45 になる。
> 表示的意味論・操作的意味論などの手法がある。詳しくは大学院の「プログラミング言語論」で扱う。

0.2 この授業を受講すべき理由

- オートマトンの直接の応用である字句解析・構文解析は、プログラミング言語処理系やデータマイニングだけではなく、広い範囲のアプリケーションプログラムで必要になる。

> （何らかの文法を持つ）テキストの入力 → 構造を持つデータ →（処理） → 別形式の出力

また、字句解析・構文解析は計算機科学の古典であり、叡智の結集である。コンパイラーやインタプリターなどのプログラミング言語処理系は大規模な記号処理プログラムの身近な例である。

> コンパイラーはソースプログラム（人間にとって理解しやすい形、C 言語なら 〜`.c`）をオブジェクトプログラム（CPU が直接理解できる形、Windows なら 〜`.exe`）に翻訳するプログラムである。
>
> 一方、インタプリターはソースプログラムを翻訳しながら実行する。

- 情報系の技術者にとってコンパイラーやインタプリターは日常使用する道具であり、ブラックボックスのままにしておくわけにいけない。例え

ば、エラーメッセージの意味を理解する必要がある。

- 字句解析に使用する正規表現や、構文解析に使用する文脈自由文法の限界を知る必要がある。さらにはコンピューターに解くことができない問題があることを認識する必要がある。

0.3 この授業で学ぶこと

問題:「"12+34*56" のように式を表す文字列を受け取って、その値を計算するプログラム `calc` を作成せよ。」

例: `calc("12+34*56")` ⇒ 1916
`calc("x*(y+78)")` ⇒ ？（x と y に依存）

問題':「"12+34*56" のように式を表す文字列を受け取って、その値を計算する機械語を生成するプログラム `compile` を作成せよ。」

字句解析

まず、どうやって単語に切り分けるのか ??? というところから始まる。

- `"12+34*56"` ⇒ "12", "+", "34", "*", "56"
- "`for(ans=0; ans<max; ans++)` …"
  ⇓
  "`for`", "`(`", "`ans`", "`=`", "`0`", "`;`", "`ans`, "`<`", "`max`", "`;`", "`ans`, "`++`", "`)`", ...

ソースプログラム（や自然言語の文）を単語に切り分ける処理を字句解析という。

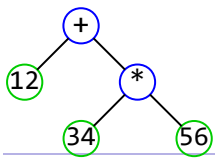
字句解析のキーワード

字句 (lexeme), トークン (token), 正規表現 (regular expression), 有限オートマトン (finite automaton), 字句解析器生成系 (lexer generator), `lex`, `flex`, （正規言語の）反復補題 (pumping lemma),

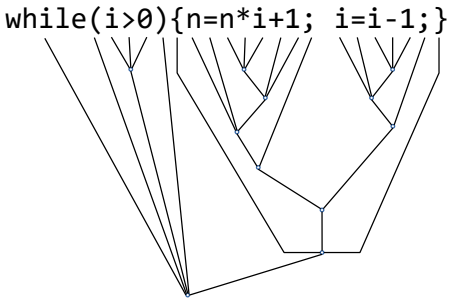
構文解析

- 切り分けた単語を、どうやってまとまった単位（例えば、文や式）にまとめて正しい順序で計算するのか？
- そもそも、式や文って何なのか？

プログラムの構造を木（構文木）などの形に表現することを構文解析という。

- `12+34*56` ⇒ 

- `while (i > 0) { n = n * i + 1; i = i - 1; }` ⇒



構文解析のキーワード

文脈自由文法 (context-free grammar), バッカス・ナウア記法 (Backus-Naur Form, BNF), プッシュダウン・オートマトン (pushdown automaton), （文脈自由言語の）反復補題, 再帰下降構文解析 (recursice descent parsing), LR 構文解析 (LR parsing), 構文解析器生成系 (parser generator), `yacc`, `bison`,

※ 灰色の項目は主に第 2 Q「オートマトン」で扱う。

0.4 コンパイラーの構成

コンパイラーはいくつかの フェーズ にわけることができる

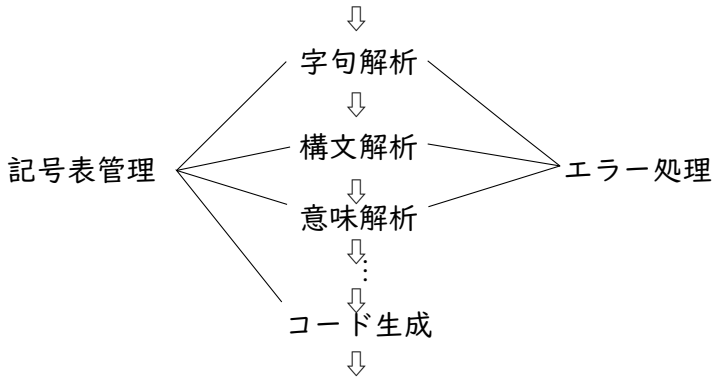
1. 字句解析 (lexical analysis)
2. 構文解析 (syntax analysis, parsing)
3. 意味解析 （ 静的解析 ）(semantic analysis, static analysis)
   型検査 など、字句解析・構文解析では見つけられない間違いを発見する。
   - 誤り検知（引数の数や型など）
   - オーバーローディング（多重定義）の解決
     例えば + は `double` の演算, `int` の演算?
   - 自動型変換の挿入
     `2 * 3.14` … `int` → `double` の変換を挿入する必要がある。
4. 中間語生成
   中間語は理想的なコンピューターの機械語と考えることができる。「理想的」とは例えば「レジスターが無限にある」などである。
5. コード最適化 (code optimization)
   生成したコードを、効率のよい形へ変換する。

   > **注意:** 「最適」という表現を使うが、本当に「最適」にするのはそもそも無理である。

6. コード生成
   レジスター割り付け (register allocation, レジスターをどのように使うかを決める) を行い、機械語を生成する。

意味解析までを フロントエンド 、中間語生成以降を バックエンド と呼ぶことがある。

0.5 全体図



0.6 記号表

記号表 (symbol table) とは、識別子 (identifier, ソースプログラム中に出現する名前)に関する情報の表である。 型、ソースプログラム中の場所、アドレス、スコープなどさまざまな情報を含む。

0.7 バッカス・ナウアー記法

BNF (Backus=Naur Form, バッカス・ナウアー記法) は言語の文法を記述するための形式である。まず例で説明する。

例 1

$$
\begin{aligned}
expr &\rightarrow expr \text{ “+” } num \\
&\quad \mid expr \text{ “−” } num \\
&\quad \mid num \\
num &\rightarrow \text{“0”} \mid \text{“1”} \mid \text{“2”} \mid \text{“3”} \mid \text{“4”} \mid \text{“5”} \mid \text{“6”} \mid \text{“7”} \mid \text{“8”} \mid \text{“9”}
\end{aligned}
$$

記号の読み方は以下の通りである。

$\rightarrow$	左辺は右辺の形からなる。左辺を右辺の形で書き換える。
$\mid$	または

ようするに、BNF は書き換え規則の集まりである。BNF の場合、左辺は 一つの記号 に限られ、右辺は記号列（空の場合も含む）である。この規則で $expr$ から書き換えられるものを $expr$ （すなわち *expr*ession — 式）とみなす。再帰的である（右辺の記号列の中に、左辺の記号が現れても良い）ところが、正規表現とのだいじな違いである。 BNF で記述される文法を 文脈自由文法 (context-free grammar) と言う。文脈自由文法で生成される言語を文脈自由言語 (context-free language) という。

> ついでに英語も覚えよう。

expression	式
number	数

ちなみに、文脈依存文法 (context-sensitive grammar) とは書き換え規則が、次のような形をしている文法である。

$$\alpha A \beta \to \alpha \gamma \beta$$

ここで A は一つの記号で、α, β, γ は記号**列**である。

文脈依存文法は文脈自由文法よりも強い表現力を持つが、プログラミング言語の文法は、通常、文脈自由文法の範囲に収まってしまうので、構文解析で文脈依存文法が必要となることは、ほとんどない。

「"1 + 2 + 3" は $expr$ である。」（「$expr$ は "1 + 2 + 3" を 導出する (derive)。」ともいう）ことを示すには、次のような列を示せばよい。

$$\begin{aligned} expr &\Rightarrow expr + num \Rightarrow expr + 3 \\ &\Rightarrow expr + num + 3 \Rightarrow expr + 2 + 3 \\ &\Rightarrow num + 2 + 3 \Rightarrow 1 + 2 + 3 \end{aligned}$$

このような $\Rightarrow$ の列（導出列）が存在することを、まとめて「$expr \overset{*}{\Rightarrow} 1 + 2 + 3$」と書く。

ここで記号の意味は次の通りである。

$\Rightarrow$	$\to$ を記号列（の一部）に適用すること
$\overset{*}{\Rightarrow}$	$\Rightarrow$ を 0 回以上適用すること

Q 0.7.1 $expr$ は "7 + 8 - 9" を導出することを示せ。

0.8 用語

終端記号 (terminal)
: 実際にプログラム中に現れる記号、これ以上書き換えできない記号
: BNF では引用符に囲んで表す（が、明らかなら省略する）
: さっきの例では 0, 1, +, - など

非終端記号 (non-terminal)
: $\to$ の左辺に現れる記号、書き換えできる記号
: 文法上の分類（式・文など）を表す。 さっきの例では、$expr$ と num

ε（イプシロン）
: 空の記号列を表す。

以上の 3 つを合わせて 構文記号 という。一方、「$\to$」と「|」はメタ記号（meta symbol）という。

「メタ」は「高次の」「超」という意味の接頭辞である。

生成規則 (production rule)
: 「非終端記号 $\to$ 構文記号の列」のかたちのこと

開始記号 (start symbol)
: 「プログラム」など、BNF で主に表現したい非終端記号のこと
: （特に断らない限り、BNF の一番上に書かれた非終端記号が開始記号に

なる。）

習慣として、非終端記号は*斜体 (italic)* や 𝒶𝒷𝒸で、終端記号は**太字 (bold)**や活字体 abc あるいは引用符囲み（" ～ "）で書き分けることが多い。

例 2

$$
\begin{aligned}
expr &\to term \mid expr + term \mid expr - term \\
term &\to factor \mid term * factor \mid term \,/\, factor \\
factor &\to const \mid var \mid (\ expr\ ) \\
const &\to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
var &\to \mathtt{x} \mid \mathtt{y} \mid \mathtt{z} \mid \mathtt{w}
\end{aligned}
$$

ついでに英語も覚えよう。

term	項
factor	因子
constant	定数
variable	変数

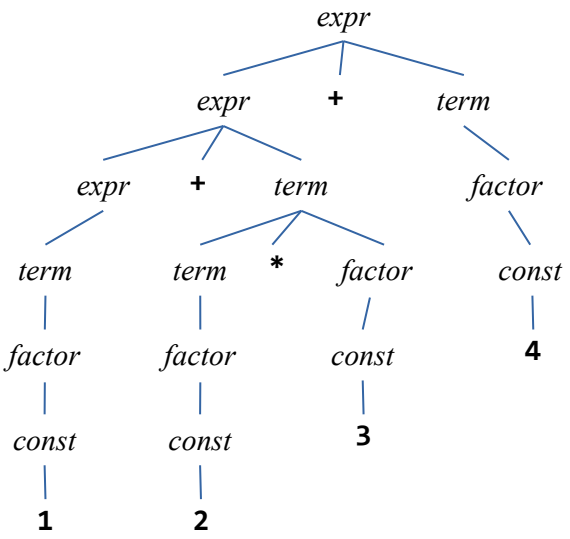
「`1 + 2 * 3 + 4`」は $expr$ である。

$$
\begin{aligned}
expr &\Rightarrow expr + term \Rightarrow expr + factor \Rightarrow expr + const \\
&\Rightarrow expr + 4 \Rightarrow expr + term + 4 \Rightarrow \ldots
\end{aligned}
$$

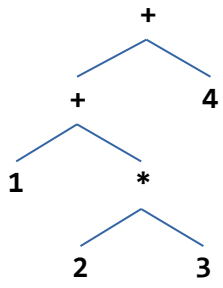
問 0.8.1 上記の導出列の続きを書け。

$$
\begin{aligned}
& expr + term * factor + 4 \Rightarrow expr + term * const + 4 \\
\Rightarrow\ & expr + term * 3 + 4 \Rightarrow expr + factor * 3 + 4 \\
\Rightarrow\ & expr + const * 3 + 4 \Rightarrow expr + 2 * 3 + 4 \Rightarrow term + 2 * 3 + 4 \\
\Rightarrow\ & factor + 2 * 3 + 4 \Rightarrow const + 2 * 3 + 4 \Rightarrow 1 + 2 * 3 + 4
\end{aligned}
$$

解析木 (parse tree)	導出列を木のかたちにあらわしたもの。 本質的でない書き換え順の違いを無視できる。
構文木 (syntax tree)	解析木を圧縮したもの 構文木の節は演算子などである。（解析木の節は非終端記号である。）



解析木



構文木

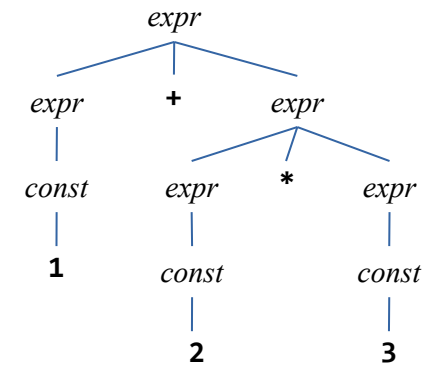
注: 例 2 の BNF は、「*」が「+」より、**結合力が強い**ことなどを表現できている。

> **問 0.8.2** 以下の記号列に対する解析木を書け。
>
> 1. `1 + 2 * 3 / 4`（例 2 の BNF で *expr* から）
> 2. `while (i > 0) { n = n * i; i = i - 1; }`（教科書 p.14 の BNF で Statement から）

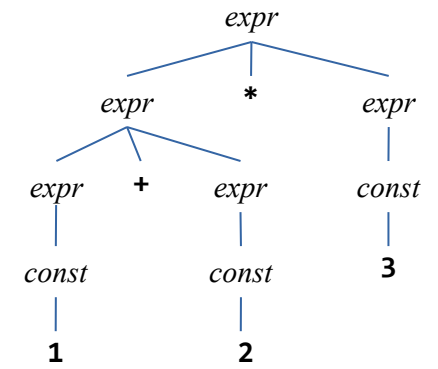
例 3

$$expr \rightarrow const \mid expr + expr \mid expr * expr$$
$$const \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

「`1 + 2 * 3`」の解析木は?



解析木 (1)



解析木 (2)

このような構文規則は 曖昧 (ambiguous) である、という

曖昧 (あいまい、ambiguous)	一つの記号列に対して、解析木が何通りもあること

曖昧な文法が役に立たない、というわけではなく、BNF に加えて 結合性 と 優先順位 を指定して、曖昧でなくすことができる（こともある）。

例

「3 - 2 - 1」、「1 + 2 * 3」など

左結合
: 同じ優先順位の演算子は左を優先する。ほとんどのプログラミング言語の「+」,「-」演算子などは左結合である。

右結合
: 同じ優先順位の演算子は右を優先する。例えば C 言語の「=」演算子などは右結合である。（「x = y = 0」は「x = (y = 0)」である。）

非結合
: 同じ優先順位の演算子が隣りあうとエラーになる。プログラミング言語によっては「<」,「>」演算子などは非結合である。その場合、「0 < x < 10」は構文エラーになる。（C 言語の「<」,「>」は左結合である。つまり、「0 < x < 10」は「(0 < x) < 10」の意味になって常に真になる。）