プログラミング言語論2014 # 2 —
構文と意味/言語処理系

久野　靖 *

2014.4.17

1　構文と意味

1.1　BNF による構文定義

プログラミング言語を定義する際には通常、その構文を BNF(Backus Naur Form、ないし Backus Normal Form) ないし類似の記法を用いて定義します。[1] たとえば、次のような具合です (ちなみに、ϵ は「何もない」空列を意味しています)。

```
文　::=　代入文　|　IF 文　|　WHILE 文　|　{　文列　}
文列　::=　文　|　文列　文
代入文　::=　変数　=　式 ;
IF 文　::=　if(　式　)　文　ELSE 部
ELSE 部　::=　ϵ　|　else　文
WHILE 文　::=　while(　式　)　文
式　::=　因子 | 式 演算子 因子
演算子 ::= + | - | * | / | > | < | …
因子 ::= 定数 | 変数 | ( 式 )
```

ちなみに、BNF は「::=」の左辺の**記号** (symbol) を右辺の並びで定義する、という形のもので、加えて | で「または」を表します (| は無くても同じ左辺の定義を複数並べれば同等の記述ができますが、この方が見やすいので)。左辺に現れるような記号は、プログラム上の概念を表すものであり、**非端記号** (nonterminal symbol) と呼びます。これに対し、「`if`」とか「`(`」のようにプログラムのコードに直接現れる記号を**端記号** (terminal symbol) と呼びます。「変数」も端記号です (変数がどんな名前かとかを文法で定義し始めると面倒なので名前をまとめて変数ということにするため)。

1.2　構文定義の役割

さて、このような構文定義はどのような役割を果たしているのでしょう。まず思い付くのは、この構文定義に合致するようなコードだけが、その言語の正しいプログラムとして受け入れられる可能性がある、という点です。[2] つまり「形が合っているかどうか」は構文定義に照らして定まります。誰もが「syntax error」で悩まされた経験をお持ちだと思います。

*経営システム科学専攻

[1] BNF で定義するというのは、文脈自由文法を与えるというのと同じことですが、文脈自由文法の説明を始めると長くなるので、ここではその説明は省略します。なお、FORTRAN や COBOL は BNF の発明以前にできた言語なので、そのような構文定義を持ちません。だから今見るとヘンな言語だなと思います。

[2] もちろん、形が合っていてもさらに別の面で間違っていて受け入れられない可能性はあります。しかし形が合っていなければ、それだけで拒否されるわけです。

しかしもちろん、プログラムの目的は「計算を記述する」ことであり、そのとき「構文に合った記述だけが許される」ということは単に形を限定する、という以上の意味があります。それはつまり「ある形の構文に合致する記述は、その構文に対応する意味を持つ」ということです。

具体的にはたとえば、次の記述があったとします。

```
if(x > y) max = x; else max = y;
```

これは if 文にあてはまるので、まず条件部の式を評価し、そしてそれが真なら then 部の文を、そうでなければ (かつ else 部があれば)else 部の文を実行する、という「意味」を持ち、そのように実行される (処理系がコンパイラであればそのような動作をするように翻訳される)、ということになります。

このように、構文に基づいて動作ないし意味が決まることを**構文主導** (syntax-directed) と呼び、Alogol-60 以後、今日の多くの言語ではこれが普通になっています。それに限らない変わったものが言語としてより良い特性を持つ、という可能性も無くはないのですが、現在のところ、この考え方が人間にとって素直で分かりやすい言語につながっている、ということは (実証的に) 否めないように思えます。

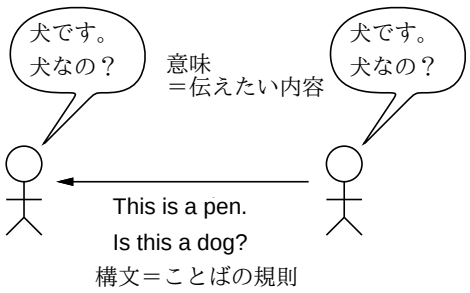

図 1: 構文と意味の対応づけ

実はこのことは、自然言語においてもある程度あてはまります。自然言語の文法でも、ある特定の形 (たとえば英語の BE 動詞の文なら、動詞と主語の入れ替え) によって特定の意味 (疑問) を表す、というふうに、構文と意味が対応しているのはよくあることです。

ただし、日本語みたいにあまり語順などに重みのない言語では、むしろ特定のフレーズの有無「例: ～なんですか?」だけで疑問の有無が決まったりするかも知れません。しかしそのような言語では、必然的にボキャブラリ (とりわけ動詞や形容詞や副詞) の役割が大きくなります。このような「違い」はプログラミング言語の設計にもある程度あり得ると考えます (そのような言語の例はまた別の回に)。

1.3 構文木と言語処理系の構成

前節では、言語の実行について、構文定義との合致を確認して、その上で構文に対応した意味動作を行う、というふうに説明しました。実際にそれを行うには、具体的なプログラムのコードが構文定義と「どのように」合致しているかをきちんと表現する必要があります。そのために使うのが**構文木** (syntax tree) です。

木構造 (tree) というのは、コンピュータサイエンスではあるノードの下に 0 個以上のノードが枝分かれしているようなデータ構造を言います。そして構文木とは、それぞれのノードが文法規則のいずれかの左辺であり、そこから枝分かれした各枝の先がその規則の右辺になっているものを言います。図 2 に先の BNF の「IF 文」に対応するコード「`if(x > y) max = x; else max = y;`」に対応する構文木を示します。

なお、ここで示したように、文法に忠実に構成した構文木には「文→代入文」のような「1 本枝」のノードが多数できるのが普通です。実際にはこの後の処理のためには、このような余分な枝は無

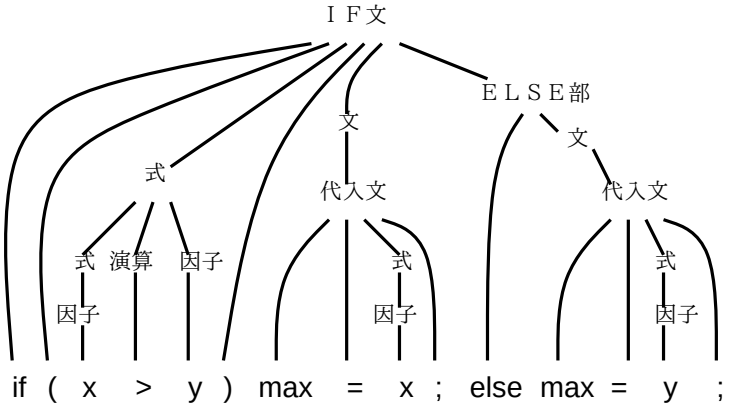

図 2: 構文木の例

い方がよいので、必要な枝だけに整理したデータ構造を作ることが一般的です。これを**抽象構文木** (abstract syntax tree) ないし AST と呼びます。

構文主導な言語処理系を基本に忠実に構成するなら、まずソースコードを解析して、そのコードに対応する構文木 (抽象構文木) を構築します。この処理のことを**構文解析** (parsing) と呼びます。次の段階として、構文木の上で意味的なチェックをおこないます。なぜそうするかというと、上述のように構文木の特定の形は特定の意味に対応しているので、それぞれの形ごとにそれが表す意味がちゃんと実行できるかをチェックするのが自然だからです。これを**意味解析** (semantic analysis) と呼びます。その後、この構文木を調べながら直接動作を実行する場合は、その処理系を**解釈実行系** (interpreter) と呼びます。一方、この構文木をもとに、実行しやすい何らかの形を生成する場合は**変換系** (translatro) と呼び、その中でも変換先が機械語ないしそれに近いものである場合には**コンパイラ** (comipler) と呼ぶわけです。

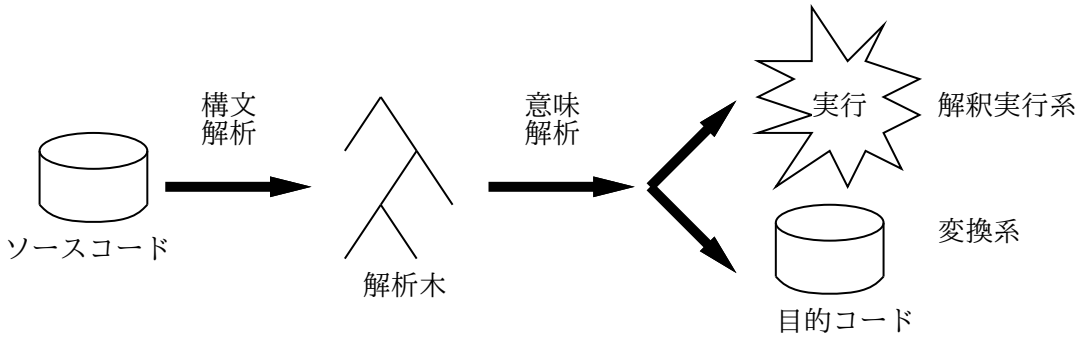

図 3: 言語処理系の構成

演習 2-1 先の文法に合致する適当なコード辺を紙に書き、その上方に構文木を描いてみよ。慣れたら、次のことをおこなえ。

a. 「`3 + x * 2`」と「`3 * x + 2`」に対する構文木を描いてみよ。これから何が分かるか述べよ。

b. 文法のうち式の定義を「式 ::= 因子 | 因子 演算子 式」に取り替えて a. をやってみよ。これから何が分かるか述べよ。

c. 「`if(x > 1) if(x < 5) y = 1; else y = 0;`」に対して 2 通りの構文木が作れることを示せ。

いずれも、できればここに現れている問題を解決するにはどう文法を直せばよいかまで検討できるとなおよい。

2 Rubyによる抽象構文木の実装

2.1 式木

抽象構文木の分かりやすい例として、演算式に対応するものを取り上げましょう。式に対応する抽象構文木のことは、**式木** (expression tree) と呼ばれることもあります。たとえば「`x + 1`」という式は、+という (加算の) ノードの左右に x(変数ノード) と 1(定数ノード) がぶら下がった式木で表現できます (図 4 左)。抽象構文木なので、「式」「項」などのノードは現れません。

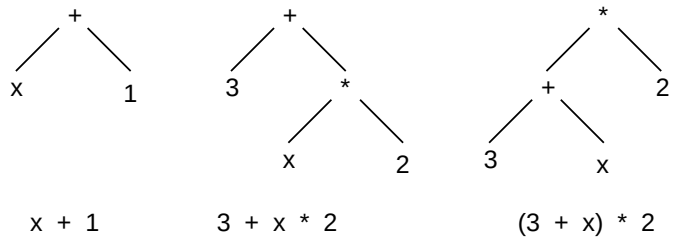

図 4: 演算式の抽象構文木 (式木)

もう少し複雑な「`3 + x * 2`」「`(3 + x) * 2`」を考えると、前者は x に 2 を掛けて 3 を足し、後者は x と 3 を足してから 2 を掛けることになります。これらの演算順序の違いも、式木では素直に表せます (図 4 中・右)。

つまり、通常の数式では「乗除算は加減算より優先」「かっこ内は先に計算」などの規則がありますが、これらは「1 列に」式を表す時に必要な解釈の規則であって、式木になった状態では演算順序は木の構造によって表されています。

2.2 抽象構文木のオブジェクト表現/動的分配

抽象構文木を Ruby のオブジェクトとして表現してみましょう。多くのノードは (上の「+」のように) 左と右の子を持つので、2 つの子を持つノードを全ノードの土台となるクラス `Node` として用意します。

```
class Node
  def initialize(l=nil, r=nil)
   @left = l; @right = r; @op = '?'
  end
  def to_s()
    return '(' + @left.to_s + @op.to_s + @right.to_s + ')'
  end
  def getleft() return @left end
  def getright() return @right end
  def getop() return @op end
end
```

Ruby では `initialize` がインスタンス生成時に自動的に呼び出されて初期設定をおこなうメソッドとなっていて、ここでインスタンス変数を初期化しています。`@left`、`@right` が左と右の子を保持す

るインスタンス変数であり、あと表示用の**演算子** (operator) — +、/などの演算記号のこと — を入れるインスタンス変数@opも持っています。**アクセサ** getleft、getright、getopは各インスタンス変数の内容を参照したい時に備えて用意してあります。

次に「+」のノードを用意することにして、Nodeを土台にしてサブクラスAddを定義します (Rubyでは「< 親クラス」という指定があるとそのクラスが指定した親クラスのサブクラスとなります):

```
class Add < Node
  def initialize(l, r) super; @op = '+' end
  def exec() return @left.exec + @right.exec end
end
```

このサブクラスではメソッド initialize と exec を定義していて、前者は親クラスにあるメソッドなのでオーバライド (差し換え) になります。initializeの最初にある **super** というのは、親クラスの同名のメソッドを (同じ引数で) 呼び出すという意味になり、これにより@left、@rightの初期化が正しく行えます。もし引数を変更したい場合は、(...)で独自の引数を指定することもできます。@opについては'+'を自分で入れています。

もう1つのメソッド execは、このノードの「処理を実行」するメソッドで、ノードの種類ごとに対応する動作を記述します。式のノードでは、実行とは「式の値を計算する」ことに相当します。ですから、足し算を行うわけですが、何を足し算するのでしょうか? それは、左と右の子をそれぞれ「計算」して、それらの結果を足し算するわけです。

足し算しか無いと実行してもつまらないので、引き算、掛け算、割り算のノードを、同様に用意します。違うのは演算子と計算の演算だけです (当然ですが)。

```
class Sub < Node
  def initialize(l, r) super; @op = '-' end
  def exec() return @left.exec - @right.exec end
end
class Mul < Node
  def initialize(l, r) super; @op = '*' end
  def exec() return @left.exec * @right.exec end
end
class Div < Node
  def initialize(l, r) super; @op = '/' end
  def exec() return @left.exec / @right.exec end
end
```

さて、「(x + 1)」を表現するためには、変数と定数 (リテラル、即値) のノードも必要です。これらの定義を示しましょう。

```
$vars = { 'x' => 5 }
class Var < Node
  def initialize(v) super; @op = '$' end
  def exec() return $vars[@left] end
end
class Lit < Node
  def initialize(v) super; @op = '#' end
  def exec() return @left end
end
```

変数は値を書き換える必要があるので、$varsというハッシュ表に名前と値をペアで保持させるようにします。ここではとりあえず、xという変数に初期値5を入れてあります。そして、変数のノード

Var では@left に変数の名前を格納しておき、exec は$vars からその名前に対応する値を取り出して返すようにしています。一方、定数のノード Lit はもっと簡単で、@left にその定数の値を入れておき、exec ではその値を返すだけです。これら 2 つのノードについては、@right は使っていません (Node の中では使わないインスタンス変数は nil に初期設定されます)。

全体として、@left.exec のようなメソッド呼び出しは、@left に現在何が入っているかに応じて、そのオブジェクトのメソッドを呼び出すことに注意してください。たとえば、@left が Add オブジェクトなら足し算、Var オブジェクトなら変数値の参照が行われて、その結果が返されます。

このように、「実行時に実際に使われているものに応じてメソッドが選択される」機能のことを**動的束縛** (dynamic binding) ないし**多態** (polymorphism) と言い、オブジェクト指向言語に不可欠な機能です。このおかげで、式の中身が何であっても「実行する」とだけ言えばその内容に応じて動作してくれるわけです。

さて、ではこれらのコードを動かして見ましょう。Ruby では irb という **REPL**(read-eval-print loop) インタフェースがあるので、これを使えば 1 つずつ例ながら動かすことができます。プログラム本体は修正したりするので、たとえば nodes1.rb というファイルに打ち込んで保存しておき、これを load により読み込ませんます。

```
% irb  ←コマンド行から irb を起動
irb> load "nodes1.rb"
  => true
irb> e = Add.new(Var.new('x'), Lit.new(1))
  => ...
irb> puts e
  => ((x$)+(1#))
irb> e.exec
  => 6
irb> f = Add.new(Lit.new(3), Mul.new(Var.new('x'), Lit.new(2)))
  => ...
irb> puts f
  => ((3#)+((x$)*(2#))
irb> f.exec
  => 13
```

確かに計算できていることが分かる。なお、変数や定数の見た目がなんだかごちゃごちゃしているが、これを直すにはクラス Var、Lit とも「def to_s() return @left.to_s end」を追加して to_s を差し替えればよい。

演習 2-2 上の例題を打ち込み、式の内容や変数の値を変えて、正しく計算できることを確認せよ。納得したら、以下のようなノードのクラスを追加し、動作を確認せよ。かっこ内は子ノードの数。to_s での表現方法は適当に決めてよい。

a. 出力 (1)。このノードを exec すると、子ノードの内容を puts により出力し、値としてはその出力した値をそのまま返す。
b. 入力 (0)。このノードを exec すると、プロンプトを出してから「gets.to_i」で整数を読み込み、それを結果として返す。
c. 連続 (2)。このノードを exec すると、左の子ノード、右の子ノードの順に exec する。結果としては、右の子ノードを exec した結果を返すものとします。
d. ループ (2)。このノードを exec すると、左の子ノードを exec して値を求め、その回数だけ右の子ノードを繰り返し exec する。結果としては、最後に右の子ノードを exec した結果を返すものとします。

2.3 制御構造の実現

もっと違った動作のノードとして、「代入」「連接」「ループ」「何もしない」を用意しました。代入は、左辺が変数であるものとして、その@left に変数文字列が入っているはずですから、$vars のその変数名のエントリに右辺の値を格納します:

```
class Assign < Node
  def initialize(l, r) super; @op = '=' end
  def exec()
    v = @right.exec; $vars[@left.getleft] = v; return v
  end
end
```

連接は、左を実行して、それから右を実行し、その結果を返すだけです。:

```
class Seq < Node
  def initialize(l, r) super; @op = ';' end
  def exec() @left.exec; return @right.exec end
end
```

ループは左を実行した結果の回数だけ右を繰り返し実行します。最後に実行した値を返すため変数 v に毎回値を保存しています (0 回実行の場合は 0 が返ります):

```
class Loop < Node
  def initialize(l, r) super; @op = 'L' end
  def exec()
    v=0; @left.exec.times do v=@right.exec end; return v
  end
end
```

ここまでは内部の計算でしたが、入出力も無いとプログラミング言語らしくないので入れます。Read は指定した変数に数値を読み込みます。Print は指定した式 (抽象構文木) を出力します。

```
class Read < Node
  def initialize(l) super; @op = '?' end
  def exec() print "? "; $vars[@left] = gets.to_i end
end
class Print < Node
  def initialize(l) super; @op = '!' end
  def exec() puts @left.exec end
end
```

最後に「何もしない」というのは後で使うものですが、to_s もオーバライドしてただの「X」を返すようにしてあります。:

```
class Noop < Node
  def initialize() end
  def exec() return 0 end
  def to_s() return 'X' end
end
```

ではちょっと試してみましょう。以下は、n に入力し、x に 1 を入れ、n の回数だけ繰り返し、x = x * n と n = n - 1 を実行し、最後に x の値を全体の結果とする (つまり 5 の階乗を計算する) コードの抽象構文木を組み立て、印刷して、実行します:

```
def test1
  e =
    Seq.new(
      Read.new('n'),
      Seq.new(
        Assign.new(Var.new('x'), Lit.new(1)),
        Seq.new(
          Loop.new(
            Var.new('n'),
            Seq.new(
              Assign.new(Var.new('x'), Mul.new(Var.new('x'),
                         Var.new('n'))),
              Assign.new(Var.new('n'), Sub.new(Var.new('n'),
                         Lit.new(1))))),
          Print.new(Var.new('x')))))
  puts(e)
  e.exec
end
```

では実行してみましょう:

```
irb> test1
((n?);(((x$)=(1#));(((n$)L(((x$)=((x$)*(n$)));
((n$)=((n$)-(1#)))));((x$)!))))
? 6
720
  => nil
```

確かに、入力を聞いてきて、6 を入れるとちゃんと 6!=720 が出力されます。

このように、プログラムの構造を内部的に表現し、それを「実行」することで、「プログラムの記述どおりの動作を行うプログラム」が作れます。一般に「プログラムの記述どおりの動作を行うプログラム」のことを**インタプリタ** (interpreter) ないし解釈実行系と呼びます。

ここで示したような抽象構文木を直接解釈するタイプのインタプリタは作りやすく、それなりに使われています (ただし遅いという弱点があります)。たとえば Ruby のバージョン 1.8.x までの実行系はこのタイプのインタプリタを用いています。

演習 2-3 ノードの種別として次のようなものを増やしてみよ。

a. 大小比較の演算子。ここでは値を全部整数としているので、たとえば「`x < y`」は条件の成否に応じて 1 または 0 を結果として持つ、ということにするとよい。
b. while 文。条件部分は「0 が `false`、それ以外はすべて `true`」として扱うものとする。
c. if 文。とりあえず then 部だけでよいが、頑張って else 部も書けるようにしたければそうしてもよい。
d. 絶対値、2 数の最大、最小などの演算子。
e. その他、目新しい/面白い機能を持った文や演算。

3 字句解析と構文解析

3.1 字句解析器

ここまでは、Ruby の構文を使って直接オブジェクト群を組み合わせ、抽象構文木を組み立ててきました。しかしもちろん、普段我々がプログラムを作る時は、普通にコードを書いて、言語処理系がそれを解析して抽象構文木を組み立てています。以下ではその簡易版を作ってみましょう。

プログラミング言語処理系の入口となるのは、入力を「名前」「定数」「記号」などに切り分ける**字句解析器** (lexical analyzer) と呼ばれる部分です。ここでは簡単のため「プログラムは 1 行だけ、すべての名前、定数、記号は 1 文字だけ、余分な空白などは一切あってはいけない」という非常に制限された字句解析用クラスを作りました:

```
class Lexer
  def initialize(s) @str = s + '$'; @pos = 0 end
  def peek() return @str[@pos..@pos] end
  def fwd() if @pos<@str.length-1 then @pos=@pos+1 end end
  def to_s() return @str[0..@pos-1]+'!'+ @str[@pos..-1] end
end
```

initalize は解析用の文字列を@str、解析位置を@pos に設定します。「$」は終わりの印の文字として使うもので、プログラム中には存在しないものとします。peek は「現在位置の文字を返す」メソッド、fwd は「現在位置を (終わりでない場合に)1 つ進める」メソッドです。最後に to_s はエラー表示用に「どこを読んでいるか」が分かりやすいように表現した文字列を返すようにしました。

動かしてみましょう:

```
irb> sc = Lexer.new('x=x+1')
=> #<Lexer:0x810b460 @str="x=x+1$", @pos=0>
irb> sc.peek
=> "x"
irb> sc.fwd
=> 1
irb> sc.peek
=> "="
irb> sc.fwd
=> 2
irb> sc.fwd
=> 3
irb> sc.fwd
=> 4
irb> sc.peek
=> "1"
irb> sc.fwd
=> 5
irb> sc.peek
=> "$"
irb> puts(sc)
x=x+1!$
=> nil
```

最後のは、「x=x+1 というプログラムを読み終わってファイル終端のところにいる」ということを表しているわけです。

演習 2-4 Lexer のインタフェース (メソッド peek、fwd を使うこと) は変えずに、普通のプログラミング言語のように複数文字から成る名前や定数 (整数だけでよい) を扱い、空白や改行は無視するような字句解析クラスを作り、後に出てくる構文解析器と組み合わせて動作させてみよ。[3]

[3] 文字列を渡す代わりにファイルからプログラムを読むようにできると、なおよいでしょう。

3.2 おもちゃ言語の構文定義

構文が決まらないと解析器が作れないので、「おもちゃ言語」の構文を定義します。

$$
\begin{aligned}
&prog ::= stat \mid stat\ \texttt{';'}\ prog \\
&stat ::= \texttt{'\{'}\ prog\ \texttt{'\}'} \mid \texttt{'L'}\ expr\ stat \mid \texttt{?}identidire \mid \texttt{!}expr \mid expr \\
&expr ::= fact \mid fact\ \texttt{'+'}\ expr \mid fact\ \texttt{'-'}\ expr \\
&\quad \mid fact\ \texttt{'*'}\ expr \mid fact\ \texttt{'/'}\ expr \mid fact\ \texttt{'='}\ expr \\
&fact ::= identfier \mid number \mid \texttt{'('}\ expr\ \texttt{')'}
\end{aligned}
$$

前述のように、全部 1 文字なので入力命令は「?変数」、出力命令は「!式」、回数指定のループは「L 回数 文により表しています。

また、この文法では**識別子** (identifier) — 名前のこと —と**数値** (number) が定義されていませんが、これらの構造は端記号として字句解析器の側で決めるのが普通です。ここでは上記の字句解析器を使うので、それぞれ英小文字 1 文字、数字 1 文字に対応させます。

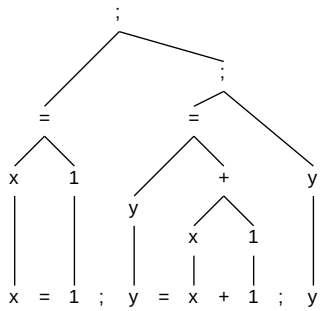

図 5: おもちゃ言語の抽象構文木

図 5 に「`x = 1; L 5 x = x * 2; x`」という簡単なプログラム (2 を 5 回掛ける、つまり 2 の 5 乗を計算するものです) に対する抽象構文木を示します。上掲の文法との対応関係を確認してみてください。

演習 2-5 次のプログラムに対する抽象構文木を描け。

```
a.  x=y=z=1
b.  x=2*3+4*5
c.  n=5;x=1;L(n){x=x*n;n=n-1};x
```

演習 2-6 自分が (演習 2 などで) 導入した独自の機能の構文を決めてそれを含むように「おもちゃ言語」の BNF を拡張せよ。さらに、その構文を持つプログラム例を選び、抽象構文木を描け。

3.3 再帰下降型構文解析器

字句解析を下請けに使ってプログラムを解析し、抽象構文木を組み立てるプログラムのことを**構文解析器** (syntax analyzer) または**パーザ** (parser) と呼びます。ここでは、**再帰下降解析器** (recursive descent parser) と呼ばれる方式の解析器を作ってみます。再帰下降解析器は、再帰手続きの動作がプログラムの構文木をたどる動作と類似していることを利用した構文解析方式です。その作り方は次のとおりです:

- 各構文記号の定義ごとに 1 つずつ手続きを用意する。(手続きの引数は字句解析器で、返値はその手続きに対応する抽象構文木となります。)

- 各手続きは、その構文定義の右辺にある構文記号に対応する手続きを 1 つずつ呼び、返されてきた木を束ねて自分の構文木を組み立てる。(呼ぶものが 1 つしか無い場合は、返された木をそのまま返すこともあります。)
- 右辺に現れるもののうち個別の文字 (記号、英字、数字) については、次の文字がその文字であることを確認して入力を先に進める。
- 右辺が | によって複数の可能性から選択になっている場合は、どの選択肢へ進むかも入力に基づいて決定する。

特に重要なのは「入力に基づいて決定する」というところで、そのような選択がうまく行えない文法はこの方法では解析できなません。幸い、先にに出てきた文法はこの方法で解析できるようになっています。というかもちろん私がそのように設計したのですが。

図 6 に、おもちゃ言語の簡単なプログラムを再帰下降解析によって解析している様子を図示します。一番先頭の *prog* はどの規則でも先頭が *stat* なので *stat* を呼び出します。*stat* は次の文字が「{」でも「L でもないので *expr* を呼び出し、*expr* はどの規則でも最初は *fact* なので *fact* を呼び出し、*fact* の中では次の文字が'x' なのでこれを変数として認識します。このように、構文木の構造と再帰手続き群の呼び出し関係がきっちり対応していることが見て取れます。

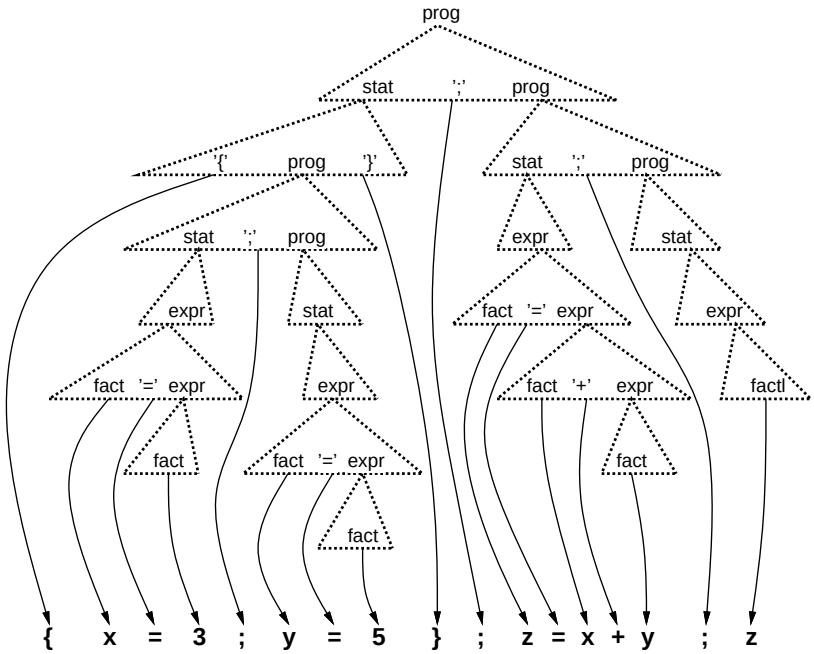

図 6: 再帰下降解析の呼び出し例

構文記号ごとに対応する手続きを見ていきましょう。まず *prog*:

```
def prog(sc)
  s = stat(sc); c = sc.peek
  if c == '$' || c == '}' then return s
  elsif c == ';' then sc.fwd; return Seq.new(s, prog(sc))
  else puts('STAT:' + sc.to_s); return Noop.new
  end
end
```

prog の右辺の先頭はいずれにせよ *stat* なので、まず stat を呼びます。次の文字が「$」か「}」ならこれで *prog* は終わりなので (なぜそうなのかは長くなるので省略します。興味ある人は参考文献を見

てください)、$prog ::= stat$ が適用されて $stat$ の木をそのまま返します。次の文字が「;」の場合は $prog ::= stat\ ';'\ prog$ なので、「;」は読み捨てて prog を再帰呼び出しし、その結果の木と先の $stat$ の木とを Seq ノードで連結したものを返します。どちらでもない場合は構文エラーなのでエラーメッセージを出力した後「何もしないノード」を返します (クラス None は実はそのために用意したのでした)。

次に $stat$:

```
def stat(sc)
  c = sc.peek
  if c == '{' then
    sc.fwd; p = prog(sc)
    if sc.peek != '}'
      puts('NO_}:' + sc.to_s); return Noop.new
    end
    sc.fwd; return p
  elsif c == 'L'
    sc.fwd; e = expr(sc); return Loop.new(e, stat(sc))
  elsif c == '?'
    sc.fwd; c = sc.peek; sc.fwd; return Read.new(c)
  elsif c == '!'
    sc.fwd; return Print.new(expr(sc));
  else
    return expr(sc)
  end
end
```

まず先頭が「{」である場合は $stat ::= '\{' prog '\}'$ に対応するので、まず 1 文字進め、prog を呼び、次が「}」であることを確認して (そうでない場合はエラー)、prog から返された木をそのまま返します。先頭が「L」の場合は $stat ::= 'L' expr\ stat$ に対応するので、1 文字進め、expr と prog を呼んでこれらを子にもつ Loop ノードを作って返します。入力や出力の場合も同様です。どれでもない場合は $stat ::= expr$ に対応するので、expr を呼んでその結果をそのまま返します。

次は $expr$ を見てみましょう:

```
def expr(sc)
  e = fact(sc); c = sc.peek
  if c == '+' then sc.fwd; return Add.new(e, expr(sc))
  elsif c == '-' then sc.fwd; return Sub.new(e, expr(sc))
  elsif c == '*' then sc.fwd; return Mul.new(e, expr(sc))
  elsif c == '/' then sc.fwd; return Div.new(e, expr(sc))
  elsif c == '=' then sc.fwd; return Assign.new(e, expr(sc))
  else return e
  end
end
```

どれを選ぶにせよまず最初は $fact$ なので fact を呼び、次に演算子のいずれかがあれば 1 文字進めて expr を呼び、これらを 2 つの子供として持つ適切な演算ノードを作って返します。これら以外の場合は $expr ::= fact$ に対応するので、fact の結果をそのまま返します。

最後は $fact$ です:

```
def fact(sc)
```

```
    c = sc.peek; sc.fwd
    if c >= 'a' && c <= 'z'
      return Var.new(c)
    elsif c >= '0' && c <= '9'
      return Lit.new(c.to_i)
    elsif c == '('
      e = expr(sc)
      if sc.peek != ')'
        puts('NO_):' + sc.to_s); return Noop.new
      end
      sc.fwd; return e
    else
      puts('FACTOR:' + sc.to_s); return Noop.new
    end
  end
```

次の文字が英小文字なら変数ノード、数字なら定数ノードを作って返します。「(」なら $fact ::='('expr')'$ なので expr を呼んでその結果を返しますが、ただし対応する「)」がなければエラーとします。これらのどれでもない場合はやはりエラーです。以上です。4 つの再帰手続きがあって分かりにくいかもしれませんが、要はそれぞれの手続きが構文記号に対応していて、構文規則にしたがって呼び出しを進めていく、ということですね。

3.4 インタプリタの完成

これで全部役者が揃ったので、インタプリタドライバを用意しました。文字列をもとに、Lexer を作り、抽象構文木を作り、その木を一応表示してみて、最後に実行します。

```
  def toylang(s)
    sc = Lexer.new(s); tree = prog(sc); puts tree; tree.exec
  end
```

さっそくこれで、階乗計算プログラムを実行させてみます。

```
  irb> toylang '?n;x=1;Ln{x=x*n;n=n-1};!x'
  ((n?);(((x$)=(1#));(((n$)L(((x$)=((x$)*(n$)));((n$)=((n$)-(1#)))));
  ((x$)!))))
  ? 6
  720
    => nil
```

ここでは「おもちゃの言語」でしたが、実際の言語処理系も基本的にはこのような原理でできています。

なお、今回の処理系は構文さえ合っていれば意味がおかしいプログラムでも実行開始してしまいます。[4] 実際の処理系では構文解析の後に**意味解析器** (semantic analyzer) と呼ばれるフェーズがあり、意味的なチェックをおこなっておかしいところがあれば実行に進まないようにします。

演習 2-7 再帰下降構文解析器を (字句解析器とともに) 打ち込み、おもちゃ言語のプログラムが解析でき実行できることを確認せよ。動いたら、自分が (演習 2 などで) 導入した新しいの機能の構文にも対応させてみよ。

[4] たとえば「(x+1)=10」みたいなものでもエラーにならずに動作します。その結果がどうなるのかは考えてみてください。

演習 2-8 ここで示した構文ではすべての演算子の順位 (優先度) が同じであり、また演算子が**右結合** (right associative) である。[5] これを、乗除算が加減算に優先され、なおかつ左結合になるように手直ししてみよ。

4 トランスレータ

ここまで作ったら、解釈実行する代わりに対応する動作をする何らかのコードを出力すれば、トランスレータやコンパイラができます。ここでは例題として、AMD64 のアセンブリ言語を出力するようにしてみました (つまりコンパイラです)。全体構成として、グローバル変数を次のように用意します。

- `$vars[変数名]` — インタプリタでは変数の値を保持するのに使っていたが、今度は変数名に対して、それが`%rsp` のどれだけ先にあるか (ローカル変数のオフセット) が取り出せるように初期化する。このため、変数に対応するノードは `initialize` でその変数を`$vars` に登録しておく。
- `$t` — 「空いている場所」の先頭オフセットを格納。場所 (作業変数) が必要になったらこの値を使った後 4 増やす。
- `$f` — アセンブリ言語プログラムを出力するファイルのストリームを格納しておく (ここに対して `puts` で出力する)。

次に、ノードについて、メソッド `emit` でコードを出力することとし、式の (値を計算する) メソッドではここから式の結果を格納した変数のオフセットを返すことにします。

では、先と同様に四則演算のノードから始めます。まず、左辺と右辺を計算し、それぞれの結果のオフセットが `l` と `r` に入ります。そして、左辺の値を`%eax` に持って来て、右辺の値を演算し、できた結果を新しい作業変数に入れて、そのオフセットを返します。

```
class Add < Node
  def initialize(l, r) super; @op = '+' end
  def emit()
    l = @left.emit; r = @right.emit; $t += 4;
    $f.puts " movl #{l}(%rsp),%eax"
    $f.puts " addl #{r}(%rsp),%eax"
    $f.puts " movl %eax,#{$t}(%rsp)"
    return $t
  end
end
class Sub < Node
  def initialize(l, r) super; @op = '-' end
  def emit()
    l = @left.emit; r = @right.emit; $t += 4;
    $f.puts " movl #{l}(%rsp),%eax"
    $f.puts " subl #{r}(%rsp),%eax"
    $f.puts " movl %eax,#{$t}(%rsp)"
    return $t
  end
end
```

[5] 右結合とは、`x - y - 1` のように演算子が並んでいる時、`x - (y - 1)` のように右側の演算が優先されることを 言います。これに対し、普通の計算規則は**左結合** (left associative) であり、左側の演算が優先されます。

```
class Mul < Node
  def initialize(l, r) super; @op = '*' end
  def emit()
    l = @left.emit; r = @right.emit; $t += 4;
    $f.puts " movl #{l}(%rsp),%eax"
    $f.puts " imull #{r}(%rsp),%eax"
    $f.puts " movl %eax,#{$t}(%rsp)"
    return $t
  end
end
class Div < Node
  def initialize(l, r) super; @op = '/' end
  def emit()
    l = @left.emit; r = @right.emit; $t += 4;
    $f.puts " movl #{l}(%rsp),%eax"
    $f.puts " idivl #{r}(%rsp),%eax"
    $f.puts " movl %eax,#{$t}(%rsp)"
    return $t
  end
end
```

変数については、オフセットを返せばいいので、アセンブリ言語は出力せず、単に変数のオフセットを返します。定数は変数ではなくてオフセットが無いので、新たな作業変数を用意してそこに値を入れ、そのオフセットを返します。

```
class Var < Node
  def initialize(v) super; @op = '$'; $vars[v] = 1 end
  def emit() return $vars[@left] end
end
class Lit < Node
  def initialize(v) super; @op = '#' end
  def emit()
    $t += 4
    $f.puts " movl $#{@left},#{$t}(%rsp)"
    return $t
  end
end
```

代入は左辺の変数のオフセットをまずもとめ、右辺の計算をしてから、その結果を左辺に移せばいいわけです。順次実行は左と右のコードを生成するだけ。

```
class Assign < Node
  def initialize(l, r) super; @op = '='; $vars[l.to_s] = 1 end
  def emit()
    l = $vars[@left.getleft]; r = @right.emit
    $f.puts " movl #{$t}(%rsp),%eax"
    $f.puts " movl %eax,#{l}(%rsp)"
    return l
  end
```

```
end
class Seq < Node
  def initialize(l, r) super; @op = ';' end
  def emit() @left.emit; @right.emit end
end
```

ループはラベルを 1 つ生成し、カウント用の作業変数を割り当ててから始めます。この変数に繰り返し回数を入れ、ラベルの後で本体部分を生成し、最後にカウントを減らしてまだ 0 より大きければラベルへ戻ります。`@right.emit` を実行すると中でいろいろ処理をするので、最後に使うラベル番号やカウンタのオフセットをローカル変数に覚えておくことが肝心です。

```
class Loop < Node
  def initialize(l, r) super; @op = 'L' end
  def emit()
    l = @left.emit; $t += 4; c = $t; $label += 1; b = $label
    $f.puts " movl #{l}(%rsp),%eax"
    $f.puts " movl %eax,#{c}(%rsp)"
    $f.puts ".L#{b}:"
    r = @right.emit
    $f.puts " decl #{c}(%rsp)"
    $f.puts " jg .L#{b}"
    return r
  end
end
```

入出力は printf/scanf を呼ぶためにごちゃごちゃしていますが、実際には C などで同じことをするコードと同等のものを機械的に生成しているだけです。Noop は本当に「何も出力しません」。

```
class Read < Node
  def initialize(l) super; @op = '?'; $vars[l] = 1 end
  def emit()
    l = $vars[@left]
    $f.puts " movl $.LC0,%edi"
    $f.puts " movl $0,%eax"
    $f.puts " call printf"
    $f.puts " leaq #{l}(%rsp),%rsi"
    $f.puts " movl $.LC1,%edi"
    $f.puts " movl $0,%eax"
    $f.puts " call scanf"
  end
end
class Print < Node
  def initialize(l) super; @op = '!' end
  def emit()
    l = @left.emit
    $f.puts " movl #{l}(%rsp),%esi"
    $f.puts " movl $.LC2,%edi"
    $f.puts " movl $0,%eax"
    $f.puts " call printf"
```

```
    end
  end
  class Noop < Node
    def initialize() end
    def exec() return 0 end
    def emit() end
  end
```

最後にコンパイラ本体ですが、まず構文木を生成したあと、作業変数オフセットの最小値やラベル番号を初期設定し、続いて$vars に入っている各変数のオフセットを割り当てます (12 以降 4 飛び)。そのあと、ファイルを出力モードで準備し、ファイルの冒頭部分を出力し (スタックフレームのサイズは適当に 64 取るので、あふれたら死にます)、本体を出力し、終わったら出口部分を出力します。末尾に scanf/printf 用の文字列を出力して終わりです。

```
  def toycomp(s)
    sc = Lexer.new(s); tree = prog(sc); puts tree
    $t = 12; $label = 0
    $vars.keys.each do |s| $vars[s] = $t; $t += 4 end
    p $vars
    $f = open("t.s", "w")
    $f.puts " .globl main"
    $f.puts "main:"
    $f.puts " pushq %rbp"
    $f.puts " movq %rsp,%rbp"
    $f.puts " subq $64,%rsp"
    tree.emit
    $f.puts " addq $64,%rsp"
    $f.puts " popq %rbp"
    $f.puts " ret"
    $f.puts ".LC0: .string \"? \""
    $f.puts ".LC1: .string \"%d\""
    $f.puts ".LC2: .string \"%d\\n\""
    $f.close
  end
```

では実際に動かしてみましょう。

```
  irb> toycomp '?n;x=0;Ln{!(x=x+1)}'
  ((n?);(((x$)=(0#));((n$)L(((x$)=((x$)+(1#)))!))))
  {"n"=>12, "x"=>16, "(x$)"=>20}
    => nil
  irb>
```

これでファイル t.s ができています。内容は次の通り。

```
   .globl main
  main:
   pushq %rbp
   movq %rsp,%rbp
   subq $64,%rsp
   movl $.LC0,%edi
   movl $0,%eax
```

```
 call printf
 leaq 12(%rsp),%rsi
 movl $.LC1,%edi
 movl $0,%eax
 call scanf
 movl $0,28(%rsp)
 movl 28(%rsp),%eax
 movl %eax,16(%rsp)
 movl 12(%rsp),%eax
 movl %eax,32(%rsp)
.L1:
 movl $1,36(%rsp)
 movl 16(%rsp),%eax
 addl 36(%rsp),%eax
 movl %eax,40(%rsp)
 movl 40(%rsp),%eax
 movl %eax,16(%rsp)
 movl 16(%rsp),%esi
 movl $.LC2,%edi
 movl $0,%eax
 call printf
 decl 32(%rsp)
 jg .L1
 addq $64,%rsp
 popq %rbp
 ret
.LC0: .string "? "
.LC1: .string "%d"
.LC2: .string "%d\n"
```

そして実行のようす。

```
% gcc t.s
% ./a.out
? 3
1
2
3
%
```

ちゃんと動いているようです。

演習 2-9 トランスレータを動かしてさまざなプログラムで試してみよ。その後、次のことを行え。

a. C で書いたプログラムと比べてどれくらい遅いか実測に基づいて検討せよ。

b. 遅いのを何とかする方法を考案せよ。

c. 1 つの方法として、式の `emit` で結果を変数に入れてオフセットを返すようになっているが、結果を`%eax` に残す方がずっとスマートである。全体をこのように手直しして動かし、どれくらい改良されたか評価せよ。