

プログラミングII

<http://bit.ly/Prog3i>

言語処理系 (1)

前期 第10週

2019/6/25

本日は・・・

- ▶ 計算機による言語解析
- ▶ lex, yaccによる言語解析プログラムの実装
- ▶ 文脈自由文法
- ▶ lex, yaccの仕組み
- ▶ 簡単な電卓の例

本日は・・・

- ▶ 計算機による言語解析
- ▶ lex, yaccによる言語解析プログラムの実装
- ▶ 文脈自由文法
- ▶ lex, yaccの仕組み
- ▶ 簡単な電卓の例

言語の種類

▶自然言語

人間が普段情報伝達に用いる言語

特徴：曖昧性を含む

▶人工言語

人間が何かしらの目的のために作り出した言語（例：C, Javaなど）

特徴：曖昧性がない

計算機による言語解析の手順

1. 字句解析

- ・入力言語を「字句（トークン）」並びに分割する処理
- ・字句：受理する文字列を正規表現で定義

2. 構文解析

- ・字句解析した言語から構文木を構築
- ・構文木構築：文脈自由文法（CFG）で規則を定義

3. 意味解析

- ・構文解析した結果に曖昧性がある場合に
 もっともらしい意味を推定
- ・人工言語でほぼ不要、自然言語で必要な処理

字句解析の例 (Cの場合)

```
int x = 0;
```

↓ 字句解析

int	x	=	0	;
-----	---	---	---	---

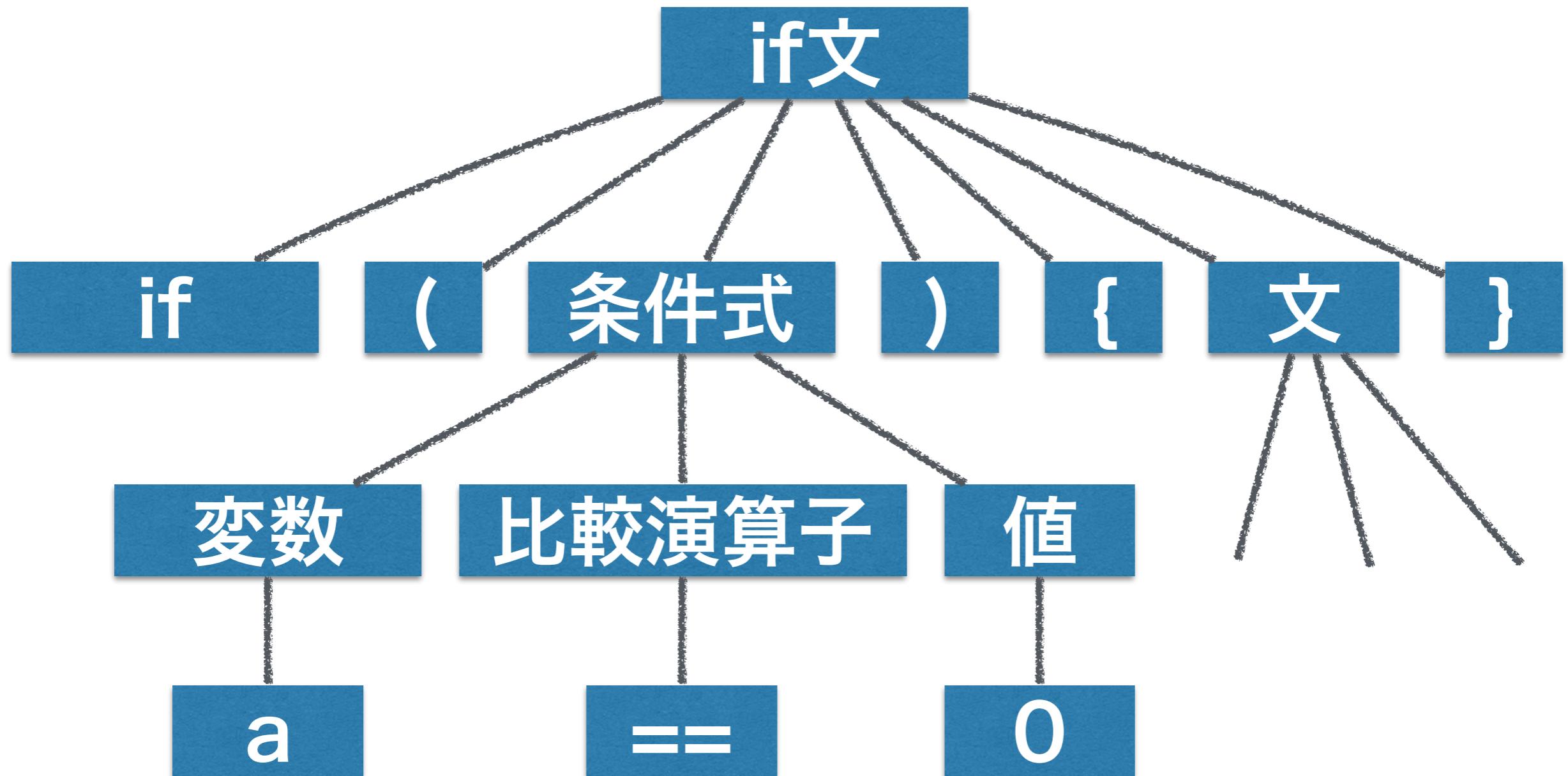
```
if (i == 0) { printf("OK\n"); }
```

↓ 字句解析

if	(i	==	0)	{	...
----	---	---	----	---	---	---	-----

構文解析の例 (Cの場合)

構文は木構造で解析される



コンパイラの役割

- ▶ (多くの場合) 文脈自由文法で定義された規則をもとに字句解析、構文解析を行う
- ▶ 構文木が出来れば (=受理されれば) 、より低水準の言語に翻訳する
- ▶ 構文木が出来なければ、エラーを出す

本日は・・・

- ▶ 計算機による言語解析
- ▶ lex, yaccによる言語解析プログラムの実装
- ▶ 文脈自由文法
- ▶ lex, yaccの仕組み
- ▶ 簡単な電卓の例

lexとyacc

▶ lex

字句規則を正規表現で与えると、**字句解析**を行うプログラムを自動生成するツール

▶ yacc

文脈自由文法による規則を与えると **構文解析**を行うプログラムを自動生成するツール

まずは・・・

lexとyaccを用いて、
簡単な電卓を字句解析・構文解析する
プログラムを生成して動かしてみます。

字句解析プログラムの作成

calc.lex

```
%%
"+" return ADD;
"-" return SUB;
"*" return MUL;
"/" return DIV;
 "(" return LP;
 ")" return RP;
"\n" return NL;

[0-9]+ {
    yylval = atoi(yytext);
    return NUMBER;
}
%%
```

詳しくは後程説明します

構文解析プログラムの作成

calc.yacc

```
%token NL LP RP NUMBER
%token ADD SUB MUL DIV
%%
list : 
    | list expr NL { printf("%d\n", $2); }
    ;
expr : expr ADD expr { $$ = $1 + $3; }
      | expr SUB expr { $$ = $1 - $3; }
      | LP expr RP   { $$ = $2; }
      | NUMBER        { $$ = $1; }
    ;
%%
#include "lex.yy.c"
```

詳しくは後程説明します

プログラム生成から実行までの流れ

1. yaccの実行 (y.tab.cが生成される)

```
$ yacc calc.yacc
```

2. lexの実行 (lex.yy.cが生成される)

```
$ flex calc.lex
```

3. Cコンパイラ実行 (警告が出るが今回は無視)

```
$ cc y.tab.c -ly -lfl
```

4. 実行して動作を確かめる

```
$ ./a.out
```

本日は・・・

- ▶ 計算機による言語解析
- ▶ lex, yaccによる言語解析プログラムの実装
- ▶ 文脈自由文法
- ▶ lex, yaccの仕組み
- ▶ 簡単な電卓の例

計算機による言語解析の手順

～人工言語の場合～

1. 字句解析

- ・入力言語を「字句（トークン）」並びに分割する処理
- ・字句：受理する文字列を正規表現で定義

2. 構文解析

- ・字句解析した言語から構文木を構築
- ・構文木構築：文脈自由文法 (CFG) で規則を定義

3. 意味解析

- ・構文解析した結果に曖昧性がある場合に
 もっともらしい意味を推定
- ・人工言語でほぼ不要、自然言語で必要な処理

文脈自由文法 (CFG)

- ▶ Context-Free Grammar
- ▶ 受理される文の構文規則を定義したもの
(生成規則という)
- ▶ 「○○はxxである」という形式で規則を記述

【例】英文法（一部）のCFGによる生成規則

- ❖ <文> → <叙述文> | <命令文>
- ❖ <叙述文> → <主語> <動詞> | <主語> <動詞> <名詞>
- ❖ <命令文> → <動詞> | <動詞> <名詞>
- ❖ <主語> → I | You
- ❖ <動詞> → play | buy | run
- ❖ <名詞> → an apple | a pineapple

終端記号と非終端記号

▶ 終端記号 (Terminal Symbol)

これ以上展開されない文字列

【例】an apple, a pineappleは終端記号

<名詞> → an apple | a pineapple

▶ 非終端記号 (Non-terminal Symbol)

さらに展開される文字列

【例】<叙述文><主語><動詞>は非終端記号

<叙述文> → <主語> <動詞> | ...

文脈自由文法の表現方法

▶ 展開

<記号1> → <記号2> 記号3

▶ 記号の連続

<記号1> <記号2> 記号3 <記号4>

▶ 論理和

<記号1> | <記号2>

文脈自由文法の受理

▶ 受理される文

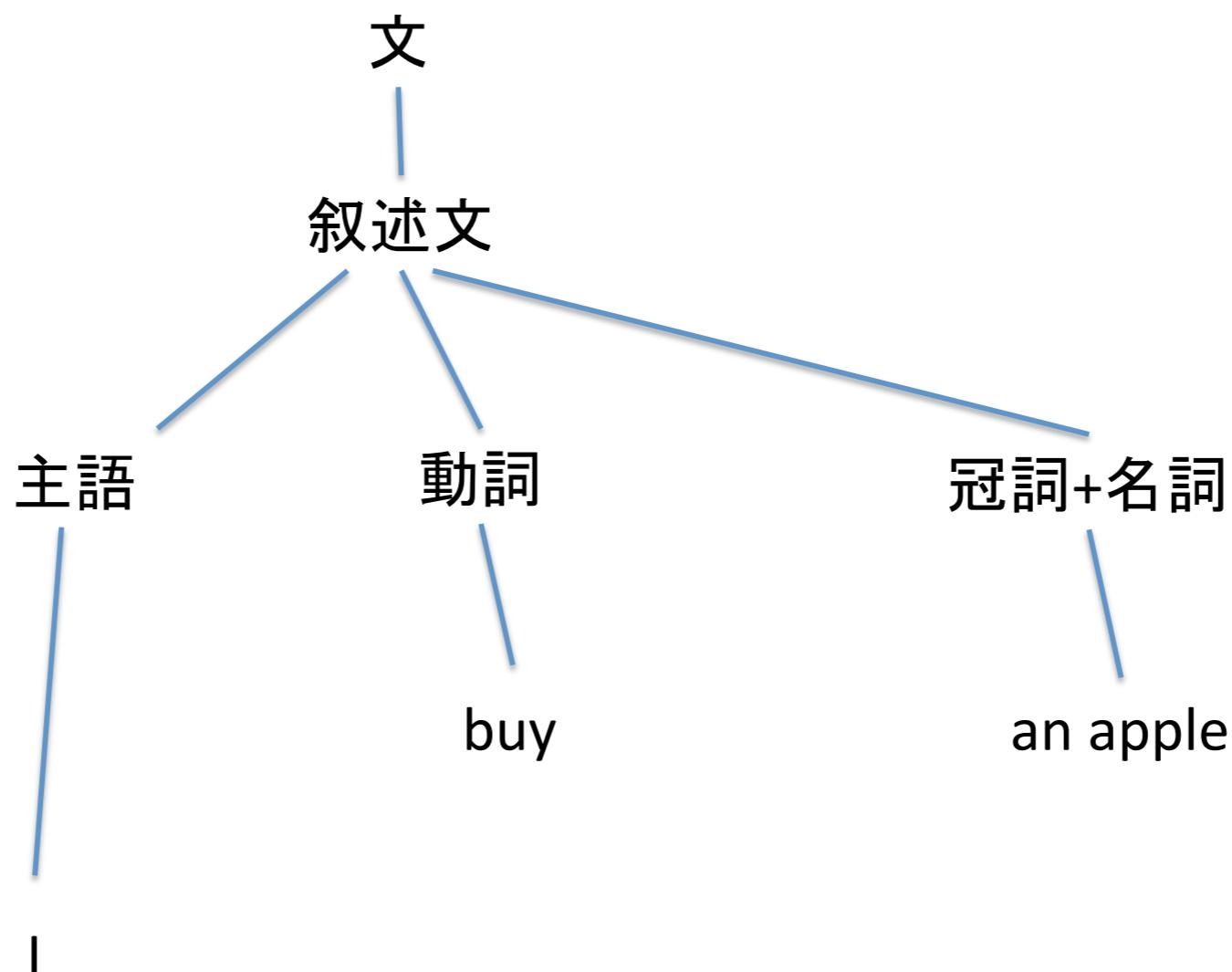
→ 規則を展開していき文が生成されれば受理

▶ 却下される文

→ 規則をどのように展開しても文が生成なければ却下

受理される文の例

文から順に展開していくと「I buy an apple」が生成されるので受理（この木を構文木と呼ぶ）



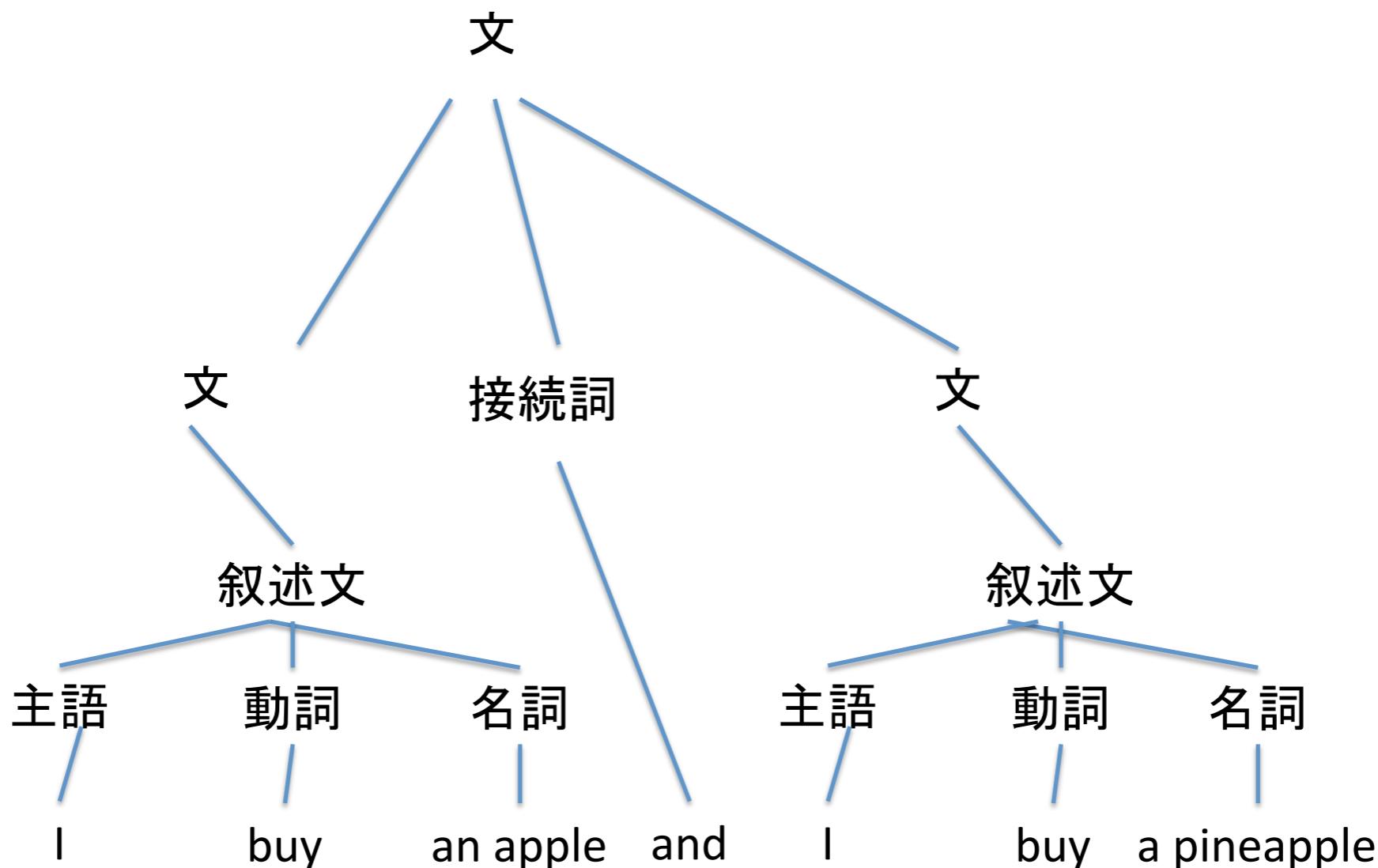
再帰するCFGの例

【例】接続詞 (and/or) も含むより複雑な英文法
(赤字部分を追加)

- ❖ <文> → <叙述文> | <命令文> | <文> <接続詞> <文>
- ❖ <接続詞> → and | or
- ❖ <叙述文> → <主語> <動詞> | <主語> <動詞> <名詞>
- ❖ <命令文> → <動詞> | <動詞> <名詞>
- ❖ <主語> → I | You
- ❖ <動詞> → play | buy | run
- ❖ <名詞> → an apple | a pineapple

受理される文の例

「I buy an apple and I buy a pineapple」 がCFGにより
生成されたので受理された



バッカスナウア記法 (BNF)

- ▶ Backus-Naur Form
- ▶ CFGをプログラム上で表現するのによく使われる記法
- ▶ 【規則1】 規則の右辺と左辺は「 ::= 」でつなぐ
- ▶ 【規則2】 非終端記号は<>で囲む
- ▶ 【規則3】 論理和は「|」で書く

<文> → <叙述文> | <命令文> | <文> <接続詞> <文>



BNFで書くと…

<文> ::= <叙述文> | <命令文> | <文> <接続詞> <文>

CFGの応用先

- ▶ プログラミング言語の作成
- ▶ 言語仕様策定 (C/Javaなどメジャーな言語の仕様はBNFで記述される)
- ▶ コンパイラの構築
- ▶ 人間の使う言語（自然言語）を計算機で扱う際の基盤技術

本日は・・・

- ▶ 計算機による言語解析
- ▶ lex, yaccによる言語解析プログラムの実装
- ▶ 文脈自由文法
- ▶ lex, yaccの仕組み
- ▶ 簡単な電卓の例

電卓の文脈自由文法

加減乗除とカッコの計算を受理する

リスト ::= (空)

式 ::= 式 + 式 | リスト 式 改行

 | 式 - 式

 | 式 * 式 | 式 / 式

 | (式)

 | 数値

字句解析プログラムの作成

パターンとアクションの組合せで記述する

「calc.lex」の中身

```
%{  
    "+"    return (ADDOP) ;  
    "-"    return (SUBOP) ;  
    "*"    return (MULOP) ;  
    "/"    return (DIVOP) ;  
    "("    return (LP) ;  
    ")"    return (RP) ;  
    $n    return (NL) ;  
    [0-9]+ { yyval = atoi(yytext) ; return (NUMBER) ; }  
    .  
%}
```

記述の始め
と終わり

識別情報を
返す

識別情報と
数値を返す
(yaccとの
組合せ)

具体的な数値を構文解析プログラムに渡すには、
yyvalというグローバル変数に値を代入

構文解析プログラムの作成

「calc.yacc」の中身

```
%token NL LP RP NUMBER
%token ADD SUB MUL DIV
%%
list : /* 空でも良い */
      | list expr NL  { printf("%d\n", $2); }
      ;
expr : expr ADD expr { $$ = $1 + $3; }
      | expr SUB expr { $$ = $1 - $3; }
      | LP expr RP    { $$ = $2; }
      | NUMBER        { $$ = $1; }
      ;
%%
#include "lex.yy.c"
```

構文解析プログラムの作成

「calc.yacc」の中身

```
%token NL LP RP NUMBER  
%token ADD SUB MUL DIV  
%%
```

lexで定義したトークンを列挙

```
list : /* 空でも良い */  
      | list expr NL { printf("%d\n", $2); }  
      ;  
  
expr : expr ADD expr { $$ = $1 + $3; }  
      | expr SUB expr { $$ = $1 - $3; }  
      | LP expr RP { $$ = $2; }  
      | NUMBER { $$ = $1; }  
      ;  
%%
```

CFGの規則を
BNFで記述

どのような計算を
するのかを記述

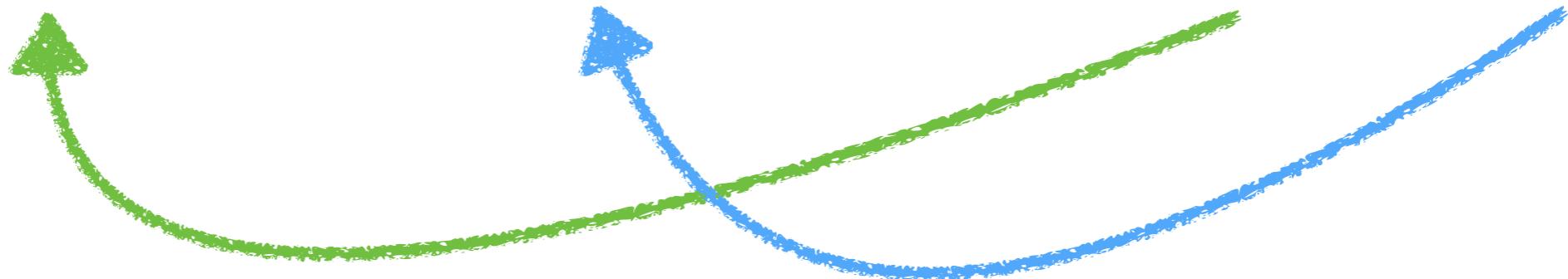
```
#include "lex.yy.c"
```

lexで作った字句定義を読み込む

構文解析プログラムの作成

\$\$は「:」の左辺の要素を表す

expr : expr ADD expr { \$\$ = \$1 + \$3; }



「:」の右辺の要素は、先頭から順番に
\$1, \$2, \$3, \$4 … と表す

本日は・・・

- ▶ 計算機による言語解析
- ▶ lex, yaccによる言語解析プログラムの実装
- ▶ 文脈自由文法
- ▶ lex, yaccの仕組み
- ▶ 簡単な電卓の例

【課題の準備】

演習室で作業する前に、以下のコマンドを入れるだけで準備が完了する

```
$ mygitclone 「自分のGitHubユーザ名」  
$ cd prog3i-(ユーザ名)  
$ ./myconf
```

※本体をシャットダウンするまでは、
上記「mygitclone」と「myconf」の設定は有効です

【課題の準備】

以下の流れで、課題のプログラムを作るためのフォルダを準備しましょう。

1. 端末を起動して、以下のコマンドを実行して**前期第10週のフォルダ**を作る

```
$ cd prog3i-(ユーザ名)          ←既に移動しているなら不要)
$ mkdir week110
$ cd week110
```

【練習10-1】

先に示した手順で、「加算と減算のみをする」簡単な電卓のプログラムを実行してみましょう。

[実行結果]

```
$ ./a.out  
1+2          (←数式を入力すると)  
3            (←計算結果が表示される)  
  
5-1  
4  
1+(7-3)  
5
```

【課題10-1】

電卓の例に「かけ算と割り算」の機能を追加して下さい。

- ▶ calc.lexには、かけ算と割り算の演算子は既に字句定義されている
- ▶ calc.yaccに、かけ算と割り算の構文を追加する
(足し算と引き算の構文を参考に作れる)

【課題の提出】

以下の流れで、作ったCプログラムをGitHubにプッシュして、Webサイトで確認してみましょう。

1. 端末内で、以下のコマンドで課題を提出

```
$ git add -A  
$ git commit -m "課題10-1提出"  
$ git push origin master
```

2. 自分のリポジトリを開いて、提出したファイルがプッシュされているか確認する

[https://github.com/nit-ibaraki-prog3i/prog3i-\(ユーザ名\)](https://github.com/nit-ibaraki-prog3i/prog3i-(ユーザ名))

【課題10-2】

課題10-1で作成した電卓に対して、「余りを求める」演算子「%」を追加して、余りの計算もできるようにして下さい。

- ▶ calc.lexに、余りを求める演算子「%」と識別子MODの字句定義を追記する
- ▶ calc.yaccに、余りを求める構文を追加する
(課題10-1と同様に作れる)

【課題の提出】

以下の流れで、作ったCプログラムをGitHubにプッシュして、Webサイトで確認してみましょう。

1. 端末内で、以下のコマンドで課題を提出

```
$ git add -A  
$ git commit -m "課題10-2提出"  
$ git push origin master
```

2. 自分のリポジトリを開いて、提出したファイルがプッシュされているか確認する

[https://github.com/nit-ibaraki-prog3i/prog3i-\(ユーザ名\)](https://github.com/nit-ibaraki-prog3i/prog3i-(ユーザ名))

【課題10-3】

課題10-2で作成した電卓に対して、「べき乗を求める」演算子「^」を追加して、べき乗の計算もできるようにして下さい。

- ▶ calc.lexに、べき乗を求める演算子「^」と識別子POWの字句定義を追記する
- ▶ calc.yaccに、べき乗を求める構文を追加する
(処理には標準ライブラリ関数powを使える)
- ▶ calc.yaccに、「math.h」をインクルードする行を追加する
- ▶ コンパイル時に「-Im」を付ける

【課題の提出】

以下の流れで、作ったCプログラムをGitHubにプッシュして、Webサイトで確認してみましょう。

1. 端末内で、以下のコマンドで課題を提出

```
$ git add -A  
$ git commit -m "課題10-3提出"  
$ git push origin master
```

2. 自分のリポジトリを開いて、提出したファイルがプッシュされているか確認する

[https://github.com/nit-ibaraki-prog3i/prog3i-\(ユーザ名\)](https://github.com/nit-ibaraki-prog3i/prog3i-(ユーザ名))

【課題10-4】

前置記法と呼れる数式の書き方があり、これは以下の
ように演算子を数字よりも前に置きます。

+ 2 3 ←これは「 $2 + 3$ 」と同じ意味

* (+ 2 3) (- 3 2) ←これは「 $(2+3)*(3-2)$ 」と同じ意味

課題10-3のプログラムをもとに、前置記法で入力で
きる電卓を作成して下さい。

- ▶ calc.yaccで記述した構文を、演算子が先にくるように変更する
- ▶ BNFの規則に対応した処理で、\$1, \$3も適した表記に変更する

【課題の提出】

以下の流れで、作ったCプログラムをGitHubにプッシュして、Webサイトで確認してみましょう。

1. 端末内で、以下のコマンドで課題を提出

```
$ git add -A  
$ git commit -m "課題10-4提出"  
$ git push origin master
```

2. 自分のリポジトリを開いて、提出したファイルがプッシュされているか確認する

[https://github.com/nit-ibaraki-prog3i/prog3i-\(ユーザ名\)](https://github.com/nit-ibaraki-prog3i/prog3i-(ユーザ名))