コンパイラ理論 6 LL構文解析

櫻井彰人

このスライドの内容

- 下降型構文解析を詳しく
- 再帰下降型（recursive descent）
- LL(1)

生成の例

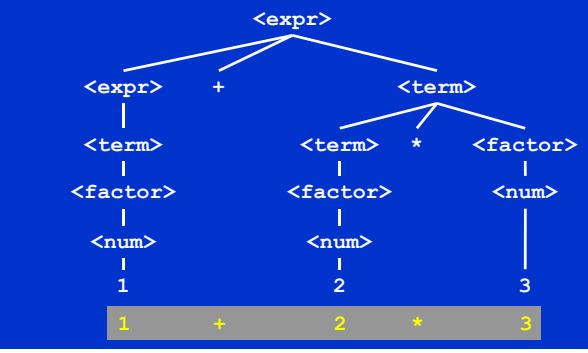
```
<sentence>     ::= <noun-phrase><verb-phrase>
<noun-phrase>  ::= <cmplx-noun>
                 | <cmplx-noun><prep-phrase>
<verb-phrase>  ::= <cmplx-verb>
                 | <cmplx-verb><prep-phrase>
<prep-phrase>  ::= <prep><cmplx-noun>
<cmplx-noun>   ::= <article><noun>
<cmplx-verb>   ::= <verb> | <verb><noun-phrase>
<article>      ::= a | the
<noun>         ::= boy | girl | flower
<verb>         ::= touches | likes | sees
<prep>         ::= with
```

```
<sentence> ::= <noun-phrase><verb-phrase>
           ::= <cmplx-noun><verb-phrase>
           ::= <article><noun><verb-phrase>
           ::= a <noun><verb-phrase>
           ::= a boy <cmplx-verb>
           ::= a boy <verb>
           ::= a boy sees
```

構文木の例

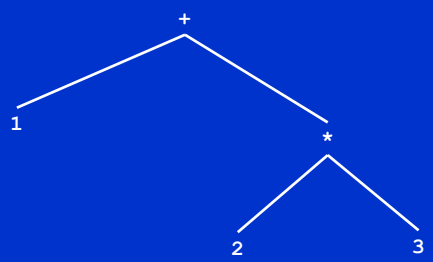
```
<expr> ::= <expr> + <term> | <term>
<term> ::= <term> * <factor> | <factor>
<factor> ::= '(' <expr ')' | <num>
<num> ::= 0 | 1 | 2 | 3 | 4 |
          5 | 6 | 7 | 8 | 9
```

1 + 2 * 3



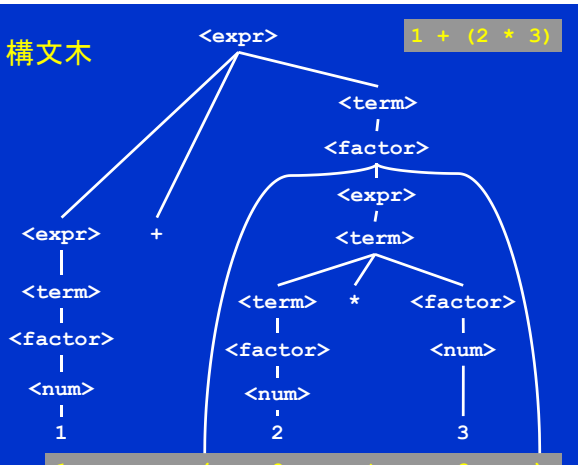
抽象構文木

1 + 2 * 3



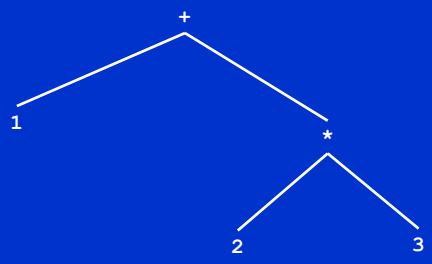
構文木

1 + (2 * 3)



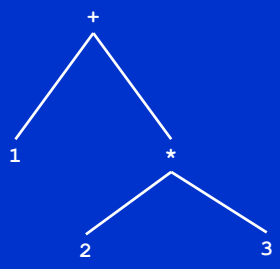
抽象構文木

```
1 + (2 * 3)
```



探索

```
1 + 2 * 3
```

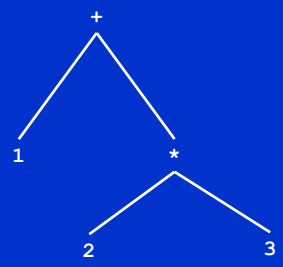



```
Preorder
In order
Post order

Breadth-first
Depth-first
```

探索

```
1 + 2 * 3
```




```
Preorder:    + 1 * 2 3
In order:    1 + 2 * 3
Post order: 1 2 3 * +
```

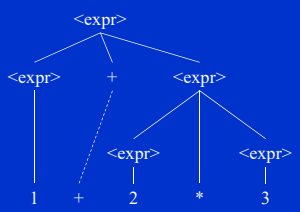
構文木

- 表現するもの
  - 構文
  - 意味
- 式であれば、演算順序が表現されていないといけない
  - すなわち、文法に組み込まれていないといけない
- 補足：抽象構文木では、構文のうち、木構造から推定できるものは捨象している

曖昧な文法

```
<expr> ::= <expr> + <expr>   |
           <expr> * <expr>   |
           '(' <expr> ')'    |
           0 | 1 | 2 | 3 | 4 |
           5 | 6 | 7 | 8 | 9
```

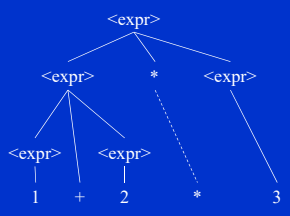
```
<expr> ::= <expr> + <expr>   |
           <expr> * <expr>   |
           '(' <expr> ')'    |
           0 | 1 | 2 | 3 | 4 |
           5 | 6 | 7 | 8 | 9
```



問題なし?

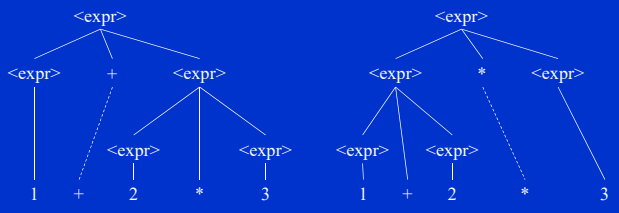
```
<expr> ::= <expr> + <expr>   |
           <expr> * <expr>   |
           '(' <expr> ')'    |
           0 | 1 | 2 | 3 | 4 |
           5 | 6 | 7 | 8 | 9
```

問題あり!



9


```
<expr> ::= <expr> + <expr>   |
           <expr> * <expr>   |
           '(' <expr> ')'    |
           0 | 1 | 2 | 3 | 4 |
           5 | 6 | 7 | 8 | 9
```

7

曖昧 (ambiguous) !

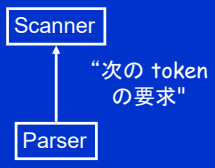
9

パーサ動作例で用いる規則

<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR <MAIN-BODY>
<PARAMS> → NULL
<PARAMS> → VAR <VAR-LIST>
<VARLIST> → , <VAR> <VARLIST>
<VARLIST> → NULL
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT>
  CURLYCLOSE
<DECL-STMT> → <TYPE><VAR><VAR-LIST>;
<ASSIGN-STMT> → <VAR> = <EXPR>;
<EXPR> → <VAR>
<EXPR> → <VAR><OP><EXPR>
<OP> → +
<OP> → -
<TYPE> → INT
<TYPE> → FLOAT

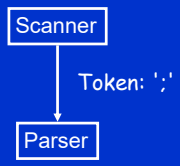
パーサ動作例

```
main() {
  int a,b;
  a = b;
}
```



パーサ動作例

```
main() {
  int a,b;
  a = b;
}
```



<C-PROG> → MAIN OPENPAR <PARAMETERS> CLOSEPAR <MAIN-BODY>
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
<DECL-STMT> → <TYPE><VAR><VAR-LIST>;
<VARLIST> → , <VAR> <VARLIST>
<VARLIST> → NULL

他になすべきこと?

- 意味動作 semantic actions
- 意味検査 semantic checks
- 記号表 symbol tables

記号表

- int a,b;
- 変数 a と b の型宣言
  - 現在のスコープ（有効範囲）内で
  - integer 型である
- これによって a と b が使用可能となる

記号表		
識別名	型	スコープ
a	int	"main"
b	int	"main"

意味動作

- 意味動作の例
- 起動の仕方
- 起動するとどうなるか?

意味動作の代表例

- 宣言された変数を記号表に書く
- 記号表で変数名を探す
- 変数の対応をとる（スコープに関する規則等）
- 型のチェック（整合性）
- 意味的な文脈の維持（型等）

```
a + b + c ⇒ t1 = a + b
            t2 = t1 + c
```

意味的な文脈

意味動作の起動

- 文法中に意味動作記号を書き込む
- 意味動作は、パーサにより、構文解析の途中で適宜呼び出され実行される
- 意味動作は、計算を行なったり、記憶したり、値を返したりすることができる
- スタックが使える
- 記号表を用いて、型チェック等ができる

意味動作の例

```
<decl-stmt> → <type>#put-type<var-list>#do-decl
<var-list>  → <var>, <var-list>#add-decl
<var-list>  → <var>
<var>       → ID#proc-decl
```

#put-type 意味スタックに「型」をつむ
#proc-decl 変数の宣言レコードを作る
#add-decl 宣言のチェーン（decl-chain）を作る
#do-decl 意味スタック上のチェインを逆方向にたどり、それぞれの変数を記号表に入れる

#do-decl
⇒

Name	Type	Scope
id1	1	3
id2	1	3
id3	1	3

意味動作

- 記号表への書き込みと読み出し以外に何を？
- 二種類
  - 型チェック (束縛, 型整合性, スコープ, etc.)
  - 通訳translation (中間変数値を生成, 意味文脈を保存すべく、その値を伝播させる).

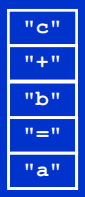
意味動作 (通訳translation)

- 対象: a = b + c + d;
- 文法:

```
<ASSGNSTMT> → <VAR> = <EXPR>#do-assign;
<EXPR>      → <VAR><EXPRTAIL>
<VAR>       → ID#process-id
<EXPRTAIL>  → <OP>#process-op<VAR>#do-infix<EXPRTAIL>
<EXPTAIL>   → NULL
```

意味動作の呼出し

- process-id: "c" の意味記述をスタックにつむ



パージング・構文解析

- パージング – 構文解析(意味解析も少し)

Top-down パージング	Bottom-up パージング
1. 根節 root node → 葉 leaves	1.葉 leaves →根節 root node
2. 抽象的範疇 → 具体的範疇	2.具体的範疇 →抽象的範疇
3. 文法規則を 左 → 右	3. 文法規則を 右 → 左
4. 手順は “予測” predictive parsing とも言う	4. 手順は “パターンマッチング”

再帰下降 recursive descent

- 基本アイデア: CFG は、一つの非終端記号に一つの関数を対応させると、(相互)再帰呼び出しをする関数の集合に写像することができる。
- 例えば、文法の一部である次の例では:

$$A \rightarrow bB \mid cC$$

```
tree  *A ()  {
  switch (nextToken()) {
    case TOK_b: {  tree *t  = B();     return
const(maketree(b), t); }
    case TOK_c: {  tree *t  = C();     return
const(maketree(c), t);  }
    default: ... error ...
  }
```

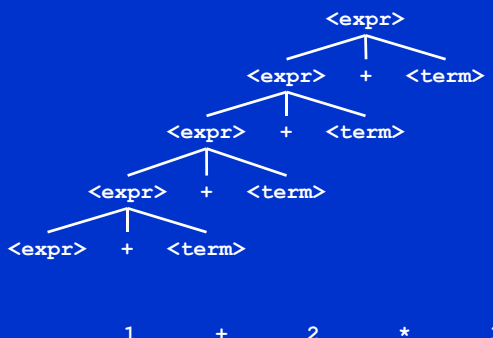
再帰下降型パージング 例

```
<expr> ::= <expr> + <term> | <term>
<term> ::= <term> * <factor> | <factor>
<factor> ::= '(' <expr> ')' |  num   |
                               ident
```

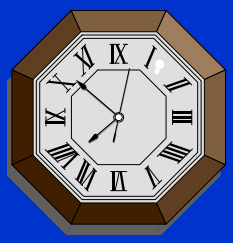
注: num と ident は終端記号と考えよう

本当のところは、字句解析の話のときに

次の予測



無限・・・・



多分、予測の仕方が悪い!?

採るべきであったのは <term>

```
<expr> ::= <expr> + <term> | <term>
<term> ::= <term> * <factor> | <factor>
<factor> ::= '(' <expr> ')' | num | ident
```

<expr>
<term>
<factor>
<ident>

選択の間違いに気づいたら、後戻りして、再試行...

1 + 2 * 3

採るべきであったのは <term>

```
<expr> ::= <expr> + <term> | <term>
<term> ::= <term> * <factor> | <factor>
<factor> ::= '(' <expr> ')' | num | ident
```

<expr>
<term>
<factor>
<num>

しかし、解析し残しがあるのは大問題

1 + 2 * 3

問題の所在

- 下記の文法は、下降型パージングに適した形ではない
- この文法には、左再帰 *left-recursive* の生成規則あり

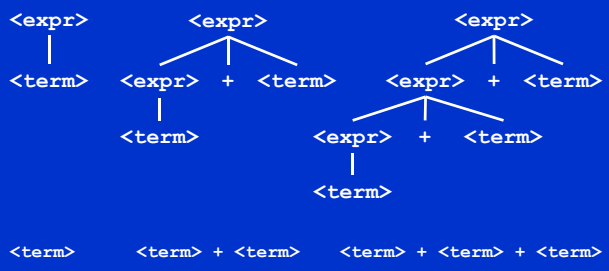
```
<expr> ::= <expr> + <term> | <term>
<term> ::= <term> * <factor> | <factor>
<factor> ::= '(' <expr> ')' |  num   |
                               ident
```

再帰 recursion

- 再帰がうまく動くには
  - 終了条件のチェック
  - 繰り返し
- ダメなのは
  - 繰り返し
  - 終了条件のチェック
- 文法を右再帰 *right-recursive* にすればよい

右再帰にする

<expr> ::= <expr> + <term> | <term>



拡張BNF
Extended Backus-Naur Form

- EBNF は { } を用いて 0 回以上の繰り返しを表す

  <expr> ::= <term> { + <term> }

- 正規表現と同等のアイデア:

  Num ::= [0-9][0-9]*

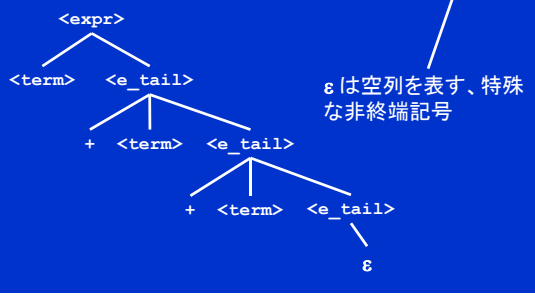
<expr> ::= <term> { + <term> }

head *tail*

BNF に戻ると

<expr> ::= <term><e_tail>

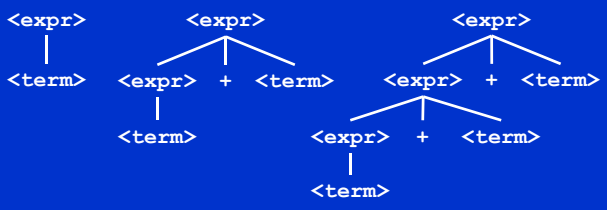
<e_tail> ::= + <term> <e_tail> | ε



ε は空列を表す、特殊な非終端記号

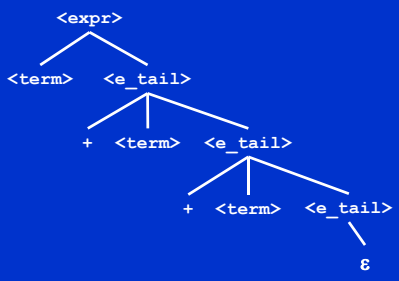
最左導出 leftmost derivation

注:最左導出のような顔をしている、というのが正しい表現



最右導出 rightmost derivation

注:最右導出のような顔をしている、というのが正しい表現



再び

<term> ::= <term> * <factor> | <factor>

- EBNF

  <term> ::= <factor> { * <factor> }

- BNF

  <term> ::= <factor><t_tail>

  <t_tail> ::= * <factor><t_tail> | ε

- これで、下降パージングができる!

文法例

```
<expr> ::= <term> <e_tail>
<e_tail> ::= + <term> <e_tail> | ε
<term> ::= <factor> <t_tail>
<t_tail> ::= * <factor> <t_tail> | ε
<factor> ::= '(' <expr> ')' | num | id
```

再帰下降型構文解析
Recursive Descent Parsing

- top-down パージングの代表
- 各非終端記号がある関数に対応
- (終端記号である)トークン token には何が?
- 字句解析 scanner が対応する
- 字句解析関数を呼ぶには:
  `gettok()`
  と仮定する

- token の型を定義する必要あり

ちょっとした小道具

```
enum {SUCCESS, FAILURE};

int Succeed(int arg)
{
  if(arg == SUCCESS)
    return 1;
  else if(arg == FAILURE)
    return 0;
  else
    /* ここはエラー処理 */
}
```

```
enum {PLUS, MULT, LPAREN, RPAREN, NUM, ID};
enum {SUCCESS, FAILURE};

int expr(void)
{
  if( Succeed(term()) && Succeed(e_tail()) )
      return SUCCESS;
  else
      return FAILURE;
}
```

```
<expr> ::= <term> <e_tail>
<e_tail> ::= + <term> <e_tail> | ε
<term> ::= <factor> <t_tail>
<t_tail> ::= * <factor> <t_tail> | ε
<factor> ::= '(' <expr> ')' | num | id
```

```
int e_tail(void)
{
  if( gettok() == PLUS &&
      Succeeds(term()) &&
      Succeeds(e_tail()) )
     return SUCCESS;
  else
     return SUCCESS; /* epsilon! がある故 */
}
```

```
<expr> ::= <term> <e_tail>
<e_tail> ::= + <term> <e_tail> | ε
<term> ::= <factor> <t_tail>
<t_tail> ::= * <factor> <t_tail> | ε
<factor> ::= '(' <expr> ')' | num | id
```

```
int term(void)
{
  if(Succeed(factor()) && Succeed(t_tail()))
      return SUCCESS;
  else
      return FAILURE;
}
```

```
<expr> ::= <term> <e_tail>
<e_tail> ::= + <term> <e_tail> | ε
<term> ::= <factor> <t_tail>
<t_tail> ::= * <factor> <t_tail> | ε
<factor> ::= '(' <expr> ')' | num | id
```

```
int t_tail(void)
{
  if( gettok() == MULT &&
      Succeeds(factor()) &&
      Succeeds(t_tail()) )
     return SUCCESS;
  else
     return SUCCESS; /* epsilon! がある故 */
}
```

```
<expr> ::= <term> <e_tail>
<e_tail> ::= + <term> <e_tail> | ε
<term> ::= <factor> <t_tail>
<t_tail> ::= * <factor> <t_tail> | ε
<factor> ::= '(' <expr> ')' | num | id
```

```
int factor(void)
{
  if( gettok() == LPAREN &&
      Succeeds(expr())  &&
      gettok() == RPAREN) )
      return SUCCESS;
  else if (gettok() == NUM)
      return SUCCESS;
  else if (gettok() == ID)
      return SUCCESS;
  else
      return FAILURE;
}
```

ん?

```
<expr> ::= <term> <e_tail>
<e_tail> ::= + <term> <e_tail> | ε
<term> ::= <factor> <t_tail>
<t_tail> ::= * <factor> <t_tail> | ε
<factor> ::= '(' <expr> ')' | num | id
```

なぜ、まずいか？

```
/* Version 2 */
int factor(void)
{
  int TokType;
  if( (TokType = gettok()) == LPAREN &&
      Succeeds(expr()) &&
        gettok() == RPAREN )
    return SUCCESS;
  else if( TokType == NUM ||
           TokType == ID )
    return SUCCESS;
  else
    return FAILURE;
}
```

それでもバグは残っている!!!

問題の所在

- 成功すれば – 問題なし。この世の習い
- 失敗したときには、その失敗を知るまでの間に食べてしまったトークンを何らかの方法で回復しないといけない

i.e. 例えば、こんな感じのプログラムに対して:

```
if(something with factor)
      return Success;
else if(something else with factor)
      return Success
else if(another thing with factor)
      etc.
```

問題の所在

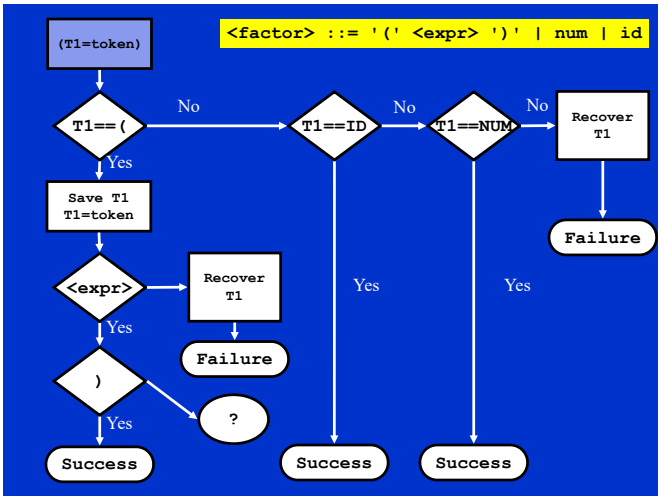
- 成功すれば – 問題なし。この世の習い
- 失敗したときには、その失敗を知るまでの間に食べてしまったトークンを何らかの方法で回復しないといけない
- 押し戻せばよい？例えば、unget で？

問題の所在

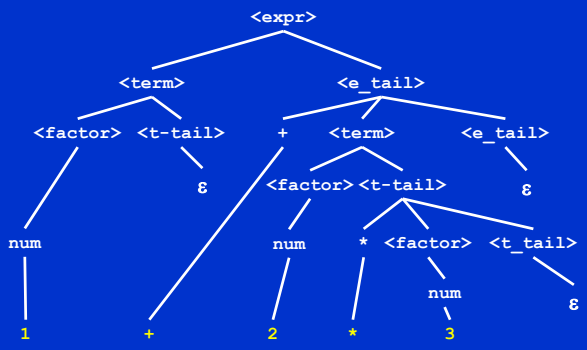
- 成功すれば – 問題なし。この世の習い
- 失敗したときには、その失敗を知るまでの間に食べてしまったトークンを何らかの方法で回復しないといけない
- 押し戻せばよい？例えば、unget で？
- そうでもない。消費してしまったのはトークンであって、文字ではないから
- では？

解

- 押し返すわけにはいかない
  - 字句解析器で工夫すれば可能
- 変数やスタックを用いればよい
  - 必要個数の上限が分かっていれば、変数で
  - そうでなければスタックで
- 結局のところは、どこかで「戻しいれ」を行なうことになる
- ただし、先読みすべき個数が分かっていて、その個数あれば次にあるべき非終端記号が高々一個に決まる場合、「戻し入れ」なしにパースが可能となる



結果



これでよいか？

- 勿論！何の問題もない。
- しかし、プログラムを一々書くのは面倒！
  - 同じことの繰り返しが多い。改善の方法がありそう。

LL(1) 構文解析

- スタックを明示的に用いたアルゴリズム
  - 再帰型ではなく、繰り返し型
- 開始記号をスタックに積む
- スタック上の非終端記号を、構文規則に従い、書き換える（スタック上で）。右辺の記号列左端がトップにくる。
- 目的は、入力列の先頭部分と一致するように、スタック上にものをおくこと
- スタックトップが入力列（トークン）先頭と一致すれば、両方を消去する。
- もし、たまたま、スタックと入力列が同時に空となれば、パースは成功

LL(1) の動作例

- 構文規則

  S → (S) S | ε

- これは、対応のとれた括弧列を生成する
  - S
  - (S) S
  - ((S) S) S
  - ((S) S) (S) S
  - (( )) ( )

問題

- 再帰降下型のときと同様に、構文規則は左再帰があっては困る
- 構文規則は、また、曖昧であっては困る。すなわち、ある入力列が構文解析可能であれば、その方法は唯一である必要がある。
- この簡単な例においてさえ、次に展開する規則の選択には、選択肢がある。どの選択をすればよいか?
  - どういうときにどのような選択をすればよいかを表にしておく。構文解析表 parse table という。
  - それが可能(ただし1トークン先読みを許す)な文法を LL(1) という

構文解析表

- 構文解析表 parse table は、終端記号(入力列の先頭にある)と現在の状態(次に展開する非終端記号)から、次にどの構文規則を次に選択するかを指示するように作る
- ところで, もし規則: $A \rightarrow \alpha \mid \beta$ があったとき、$A \rightarrow \alpha$ を用いるときと $A \rightarrow \beta$ を用いるときとをどう区別したらよいだろうか?
- これは、α と β の ***First*** 集合を定めることにより解決する。***First*** 集合: その記号列から導出される終端記号列の先頭終端記号の集合
  - α と β の ***First*** 集合が排他的であればよい。入力列先頭がどちらに属するかで、次に適用する規則が決定できる

構文解析表

- 更に、次のような規則は、いつ、適用すべきかを知る必要がある(この規則には、***First*** 集合というアイデアが適用できないため、別の考え方が必要)

  $A \rightarrow \varepsilon$

- これは、非終端記号 A の後に(これは上記の規則では決まらない。A を右辺に持つ規則から決まる)どんな終端記号がくるかを知ることにより、解決できる。
- 終端記号の、こうした集合を A の ***Follow*** 集合という。
  - 次の終端記号が A の***Follow*** 集合に含まれれば、$A \rightarrow \varepsilon$ を適用してよいことになる

構文解析表

- LL(1) の構文解析表は、終端記号(トークン)に対応する列と、非終端記号に対応する行とからなる
  - 入力列は、終端記号(トークン)の列であるとする
- スタック先頭は、終端記号(トークン)か非終端記号
- 構文解析表の値(entry)は, その場合に適用する構文規則である

簡単な例

非終端記号 \ トークン	(	)	$
S	S → (S) S	S → ε	S → ε

First 集合と *Follow* 集合

- LL(1) 構文解析表の作成はアルゴリズム的に可能であり、従って、自動化できる
- 実際の言語の文法に対して、これを手で行なうことはつらい！
- 構文表を作るには、文法で用いる記号に対して、まず、*First* 集合と *Follow* 集合とを作る必要がある
- 文法例

```
exp → exp addop term | term
addop → + | -
term → term mulop factor | factor
mulop → *
factor → ( exp ) | number
```

文法の変換

- この文法は、左再帰のため、まず、右再帰の文法に変換する必要がある

```
exp → term exp'
exp' → addop term exp' | ε
addop → + | -
term → factor term'
term' → mulop factor term' | ε
mulop → *
factor → ( exp ) | number
```

書き直すと、、、

```
exp → term exp'
exp' → addop term exp'
exp' → ε
addop → +
addop → -
term → factor term'
term' → mulop factor term'
term' → ε
mulop → *
factor → ( exp )
factor → number
```

何をすべきであったか?

- 構文規則の右辺の列から導出される終端記号列の先頭に来る終端記号（トークン）の集合を求めること
- **A → B… であれば、first(A)=first(B) となるので（そうでない場合については、次の次のスライドに）、先の例では、次の方程式をとけばよいことになる**

```
first(exp  ) = first(term)
first(exp' ) = first(addop)∪{ε}
first(addop) = {+,-}
first(term ) = first(factor)
first(term') = first(mulop)∪{ε}
first(mulop) = {*}
first(factor)= {( , number}
```

```
exp → term exp'
exp' → addop term exp'
exp' → ε
addop → +
addop → -
term → factor term'
term' → mulop factor term'
term' → ε
mulop → *
factor → ( exp )
factor → number
```

どうやって？

- 反復法！
  - 方程式の等号を代入記号と読替える。
  - 初期値をいずれも空集合として、収束するまで繰り返す
  - 収束するか？ 収束する！
    - 各集合とも小さくなることはない。繰り返しの過程で、ある終端記号がある集合に追加されることがあっても、減らされることはない
    - けれども、天井がある。高々有限集合だから。

```
first(exp  ) ← first(term)
first(exp' ) ← first(addop)∪{ε}
first(addop) ← {+,-}
first(term ) ← first(factor)
first(term') ← first(mulop)∪{ε}
first(mulop) ← {*}
first(factor)← {( , number}
```

注意事項

- **A → B… であれば、first(A)=first(B) となるかというと、そうでもない**
  - **first(A)⊇first(B) であることは確かであるが**
- **B→…→ε となる可能性があるからである**
- **A → BCD のとき、 B→…→ε とも C→…→ε ともなりうるならば、**
  first(A)=first(B)∪first(C)∪first(D)
- **今回の例ではこうした事態は考えなくてよい。空列になる非終端記号は、 exp' と term' だけで、いずれも、右辺の先頭には来ていないから**

構文規則	
exp → term exp'	first(exp) = { (, number }
exp' → addop term exp'	first(exp') = { +, -, ε }
exp' → ε	first(exp') = { +, -, ε }
addop → +	first(addop) = { +, - }
addop → -	first(addop) = { +, - }
term → factor term'	first(term) = { (, number }
term' → mulop factor term'	first(term') = { *, ε }
term' → ε	first(term') = { *, ε }
mulop → *	first(mulop) = { * }
factor → (exp)	first(factor) = { (, number }
factor → number	first(factor) = { (, number }

Follow 集合

- First 集合は、(当該非終端記号が左辺に現れる構文規則の)右辺の記号列から導出される終端記号列の先頭に現れる終端記号の集合であった
- 非終端記号 A の Follow 集合は、A の後に続く終端記号の集合であり、構文規則 $A \rightarrow \varepsilon$ が適用可能かどうかを判断するのに用いられる
- すなわち、スタックのトップが A であり、入力列の先頭が A の follow 集合に含まれるなら、$A \rightarrow \varepsilon$ を適用しようということになる

Follow 集合

- 非終端記号 A が与えられたとき, 終端記号($を含む)の集合 Follow(A) は次のように定義される
  1. A が開始記号であれば, $ ∈ Follow(A)
  2. もし構文規則 $B \rightarrow \alpha A \gamma$ があれば, First(γ) - {ε} ⊆ Follow(A)
  3. もし構文規則 $B \rightarrow \alpha A \gamma$ があり, かつ, ε ∈ First(γ) であれば, Follow(B) ⊆ Follow(A)
     - 注: このことから、構文規則が $B \rightarrow \alpha A$ の形をしていれば、Follow(B) ⊆ Follow(A) といえる

同じ問題を再び

文法

```
exp → term exp'
exp' → addop term exp'
exp' → ε
addop → +
addop → -
term → factor term'
term' → mulop factor term'
term' → ε
mulop → *
factor → ( exp )
factor → number
```

文法

```
exp → term exp'
exp' → addop term exp'
exp' → ε
addop → +
addop → -
term → factor term'
term' → mulop factor term'
term' → ε
mulop → *
factor → ( exp )
factor → number
```

赤で書いた構文規則は非終端記号を右辺に持たないため、follow 集合には影響を与えない

考慮する規則のみ

```
exp → term exp'
exp' → addop term exp'
term → factor term'
term' → mulop factor term'
factor → ( exp )
```

番号を振ろう

(1) exp → term exp'
(2) exp' → addop term exp'
(3) term → factor term'
(4) term' → mulop factor term'
(5) factor → (exp)

まとめると

Follow(term) += First(exp')
Follow(exp') += Follow(exp)
Follow(term) += Follow(exp)
Follow(addop) += First(term)
Follow(term) += First(exp')
Follow(term) += Follow(exp')
Follow(factor) += First(term')
Follow(factor) += Follow(term)
Follow(term') += Follow(term)
Follow(mulop) += First(factor)
Follow(factor) += First(term')
Follow(factor) += Follow(term')
Follow(exp) += {) }

Follow(exp) = {}
Follow(exp') = {}
Follow(addop) = {}
Follow(term) = {}
Follow(term') = {}
Follow(mulop) = {}
Follow(factor) = {}

Follows 集合の計算に移る

Follow(term) += First(exp')
Follow(exp') += Follow(exp)
Follow(term) += Follow(exp)
Follow(addop) += First(term)
Follow(term) += First(exp')
Follow(term) += Follow(exp')
Follow(factor) += First(term')
Follow(factor) += Follow(term)
Follow(term') += Follow(term)
Follow(mulop) += First(factor)
Follow(factor) += First(term')
Follow(factor) += Follow(term')
Follow(exp) += {) }

Follow(exp) = { $) }
Follow(exp') = { $) }
Follow(addop) = { (number }
Follow(term) = { + - $) }
Follow(term') = {+ - $) }
Follow(mulop) = { (number }
Follow(factor) = { * + - $) }

No change!

結果

First(exp) = { (number }
First(exp') = { + - ε }
First(addop) = { + - }
First(term) = { (number }
First(term') = { * ε }
First(mulop) = { * }
First(factor) { (number }

Follow(exp) = { $) }
Follow(exp') = { $) }
Follow(addop) = { (number }
Follow(term) = { + - $) }
Follow(term') = {+ - $) }
Follow(mulop) = { (number }
Follow(factor) = { * + - $) }

構文解析表

再掲

- LL(1) の構文解析表は、終端記号（トークン）に対応する列と、非終端記号に対応する行とからなる
  - 入力列は、終端記号（トークン）の列であるとする
- スタック先頭は、終端記号（トークン）か非終端記号
- 構文解析表の値（entry）は, その場合に適用する構文規則である

構文解析表の作成

- 以下の2ステップを、各日終端記号 A と構文規則 A → α について繰り返せ
  1. First(α) 中のトークン ***a*** について，表 Table[A][***a***] に A → α を追加.
  2. もし ε ∈ First(α) ならば，Follow(A) (トークンか $) の各 ***a*** について,表 Table[A][***a***] に A → α を追加.
- 例
- 非終端記号: exp
- 構文規則: exp → term exp'
- First(term) = { (number }
- 従って、“ exp → term exp‘ ” を Table[exp][(] と Table[exp][number] に追加

表[N][T]	(	number	)	+	-	*	$
exp	exp→ term exp'	exp→ term exp'					
exp'			exp' → ε	exp' → addop term exp'	exp' → addop term exp'		exp' → ε
addop				addop → +	addop → -		
term	term → factor term'	term → factor term'					
term'			term' → ε	term' → ε	term' → ε	term' → mulop factor term'	term' → ε
mulop						mulop→ *	
factor	factor → (exp)	factor → number					

構文解析表の作成

- 以下の2ステップを、各日終端記号 A と構文規則 A → α について繰り返せ
  1. First(α) 中のトークン ***a*** について，表 Table[A][***a***] に A → α を追加.
  2. もし ε ∈ First(α) ならば，Follow(A) (トークンか $) の各 ***a*** について,表 Table[A][***a***] に A → α を追加.
- 例
- 非終端記号: term'
- 構文規則: **`term' → mulop factor term'`**
- First(term') = { * ε }
- 従って、“ term‘ → mulop factor term’ ” を 表[term‘][*] に追加

表[N][T]	(	number	)	+	-	*	$
exp	exp→ term exp'	exp→ term exp'					
exp'			exp' → ε	exp' → addop term exp'	exp' → addop term exp'		exp' → ε
addop				addop → +	addop → -		
term	term → factor term'	term → factor term'					
term'			term' → ε	term' → ε	term' → ε	term' → mulop factor term'	term' → ε
mulop						mulop→ *	
factor	factor → (exp)	factor → number					

構文解析表の作成

- 以下の2ステップを、各日終端記号 A と構文規則 A → α について繰り返せ
  1. First(α) 中のトークン ***a*** について，表 Table[A][***a***] に A → α を追加.
  2. もし ε ∈ First(α) ならば，Follow(A) (トークンか $) の各 ***a*** について,表 Table[A][***a***] に A → α を追加.
- 例
- 非終端記号: term'
- 構文規則: **`term' → ε`**
- Follow(term') = {+ - $) }
- 従って、“ term‘ → **ε** ” を 表[term’][+], 表[term’][-], 表[term’][$] および 表[term‘][)] に追加

表[N][T]	(	number	)	+	-	*	$
exp	exp→ term exp'	exp→ term exp'					
exp'			exp' → ε	exp' → addop term exp'	exp' → addop term exp'		exp' → ε
addop				addop → +	addop → -		
term	term → factor term'	term → factor term'					
term'			term' → ε	term' → ε	term' → ε	term' → mulop factor term'	term' → ε
mulop						mulop→ *	
factor	factor → (exp)	factor → number					