

# コンパイラ構成

構文解析, 文脈自由文法

情報工学系  
権藤克彦



# 文脈自由文法

# 構文解析の概要 (1/3)

- プログラミング言語 = 構文(syntax)
  - + 制約(constraint)
  - + 意味(semantics)
- 構文 = プログラム中のトークン列の並べ方の規則
  - 通常、文脈自由文法 (or 部分集合 LL(1) や LALR(1)) で定義
  - 通常、文脈自由文法は BNF 記法や EBNF 記法で表現
- 例：式の構文の定義例

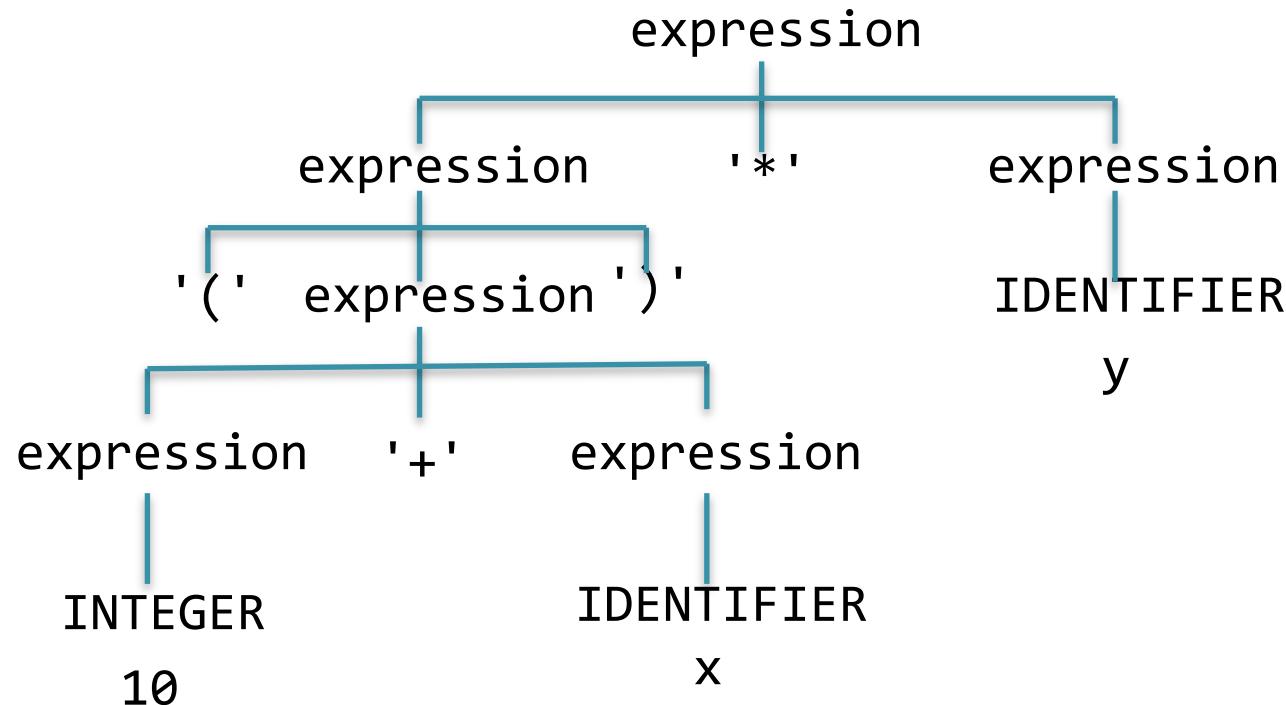
```
expression
: expression "+" expression
| expression "*" expression
| "(" expression ")"
| IDENTIFIER
| INTEGER
; 
```

式は、  
式と '+' と式を並べたもの  
または、式と '\*' と式の並び  
または、式をカッコで囲んだもの  
または識別子  
または整数定数



# 構文解析の概要 (2/3)

- ・ 構文解析=トークン列から構文木(syntax tree)への変換
- ・ 例： $(10+x)*y$  の構文解析の結果





# 構文解析の概要 (3/3)

- $(10+x)*y$  が式を確認するステップ（導出）

**expression**

⇒ **expression \* expression**

⇒ ( **expression** ) \* **expression**

⇒ ( **expression** + **expression** ) \* **expression**

⇒ ( INTEGER + **expression** ) \* **expression**

⇒ ( INTEGER + IDENTIFIER ) \* **expression**

⇒ ( INTEGER + IDENTIFIER ) \* IDENTIFIER

( 10                + x                ) \* y



# 文脈自由文法 (context-free grammar; CFG)

- 文脈自由文法は生成規則から文を再帰的に導出する
  - 文脈自由文法は生成規則 (production rules) で定義
- 生成規則の形式:  $X_0 \rightarrow X_1 X_2 \cdots X_n$ 
  - 左辺の  $X_0$  は非終端記号. ちょうど 1 つ
  - 右辺の  $X_i$  ( $1 \leq i \leq n$ ) は非終端記号か終端記号.
  - 右辺は 0 個以上. つまり右辺は空でも良い ( $X_0 \rightarrow \varepsilon$ )
- 用語
  - 終端記号 (terminal symbols)=トークン. 単語. 展開しない.
  - 非終端記号 (nonterminal symbols)=トークン以外の記号. 展開する
    - 例: 「式」 「文」 「名詞」 「動詞」などの抽象的な文法要素
  - 開始記号 (start symbols)=プログラム全体を表す非終端記号
    - 例: program, translation\_unit (C 言語の場合)
  - 記号=非終端記号 + 終端記号



# 生成規則

- 生成規則の例：加算式

$\text{expression} \rightarrow \text{expression} \ " + " \ \text{expression}$

- 読み方 「式とプラス記号と式を並べたものは式である」  
「式は式とプラス記号と式に展開して良い」

- 同じ左辺を持つ生成規則があって良い

$\text{expression} \rightarrow \text{expression} \ " + " \ \text{expression}$

$\text{expression} \rightarrow \text{INTEGER}$



# 直接導出 (direct derivation)

- Aを非終端記号.  $\alpha, \beta, \gamma$ を任意の文法記号列とする
- 生成規則  $A \rightarrow \gamma$ がある時
  - 直接導出 =  $\alpha A \beta$ から  $\alpha \gamma \beta$ への展開.  $\alpha A \beta \Rightarrow \alpha \gamma \beta$ と表記
  - 直接還元 =  $\alpha \gamma \beta$ から  $\alpha A \beta$ に展開を戻す. つまり直接導出の逆
- 直接導出の例
  - 次の生成規則がある時  
 $\text{expression} \rightarrow \text{expression} \ " + " \ \text{expression}$
  - 以下は上の生成規則を用いた直接導出  
 $( \text{expression} ) + \text{expression}$   
 $\Rightarrow ( \text{expression} + \text{expression} ) + \text{expression}$
- 実は p. 5も直接導出の例



# 導出 (derivation)

- 導出 = **0回以上** の直接導出の繰り返し
- 導出の記法：
  - $\alpha \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_n \Rightarrow \beta$  の時 (ただし  $\Rightarrow$  の数は 0 個以上)  
 $\alpha$  から  $\beta$  への導出を  $\alpha \Rightarrow^* \beta$  と表記
- 導出の例：
  - expression  $\Rightarrow^*$  ( INTEGER + IDENTIFIER ) \* IDENTIFIER
  - expression  $\Rightarrow^*$  expression 全く展開しないのも導出
- **1回以上** の直接導出で  $\alpha$  が  $\beta$  になる時,  $\alpha \Rightarrow^+ \beta$  と表記
- 文脈自由文法 G が生成する言語 L(G)：
  - G の開始記号から導出可能な終端記号列 (= 文) の集合
  - $L(G) = \{\omega \mid S \Rightarrow^+ \omega\}$ , ただし S は開始記号,  $\omega$  は終端記号列



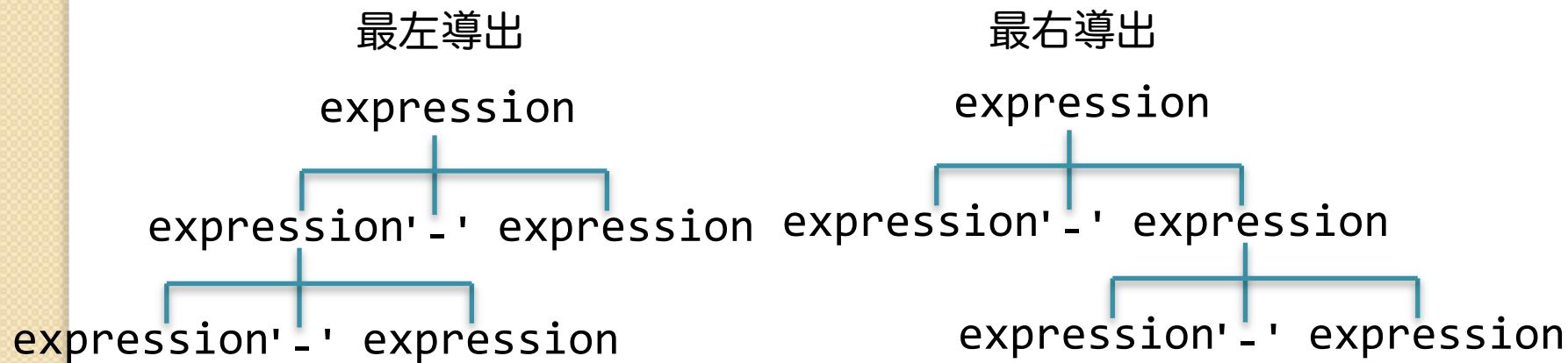
# 構文木 (syntax tree)

- 構文木=文法の開始記号からの導出の木構造表現。
  - 導出木ともいう
  - 木の根は開始記号, 内部ノードは非終端記号, 葉は終端記号
  - あるノードAの子ノードが $B_1, \dots, B_n$ の時
    - $A \rightarrow B_1, \dots, B_n$  という生成規則が存在し
    - Aから $B_1, \dots, B_n$ を直接導出したことを示す
- 構文木の例：
  - [p.4](#) の木は [p.5](#)の導出の構文木
- 抽象構文木 (abstract syntax tree; AST)
  - 終端記号や字句情報を落とした構文木. 構文木とほぼ同義.
    - 字句情報=コメントや空白類等の情報



# 最左/最右導出 (leftmost/rightmost derivation)

- 最左導出 = 最も左の非終端記号を展開する導出
  - 例 :  $\text{expression} \Rightarrow \text{expression} - \text{expression}$   
 $\Rightarrow \text{expression} - \text{expression} - \text{expression}$
- 最右導出 = 最も右の非終端記号を展開する導出
  - 例 :  $\text{expression} \Rightarrow \text{expression} - \text{expression}$   
 $\Rightarrow \text{expression} - \text{expression} - \text{expression}$



# あいまいな文 (ambiguous sentence)

- あいまいな文 = 複数の異なる構文木が存在する文
  - どの非終端記号を展開するかで構文木が変化 (例 : [p.11](#))
- あいまいな文法 = あいまいな文が存在する文法
- 対処法：
  - 文法を書き替える (文法が分かりにくくなる可能性あり)
  - 優先順位や結合性を別に指定する (例 : GNU Bison) .

```
%right '='
%left "||"
%left "&&"
%left "==" 
%left '<' 
%left '+' '-'
%left '*' '/'
```

- 下に行くほど高い優先度
- + と - は同じ優先度
- right は右結合, left は左結合  
(後述)



# ぶらぶらelse (dangling else)

- 以下の if文とif-else文を含む文法はあいまい
  - statement → "if" "(" expression ")" statement
  - statement → "if" "(" expression ")" statement  
"else" statement
- 「if ( $e_1$ ) if ( $e_2$ )  $s_1$  else  $s_2$ 」はあいまいな文
  - elseをどっちのifに対応付けるかで2通り構文木が存在
  - if ( $e_1$ ) **if ( $e_2$ )  $s_1$  else  $s_2$**  C言語では常にこちらと解釈

```
graph TD; A["if (e1)"] --> B["if (e2) s1 else s2"]; B --> C["if文"]; A --> D["if (e2)"]; D --> E["s1 else s2"]; E --> F["if文"]
```



# 結合性 (associativity)

- 結合性 = 同じ（優先度の）演算子が複数ある時の計算順序を示す性質
  - 左結合 (left associative). 例 :  $1-2-3$  は  $(1-2)-3$  と同じ
  - 右結合 (right associative). 例 :  $x=y=10$  は  $x=(y=10)$  と同じ
  - 非結合 (non-associative). 例 :  $a < b < c$  は許されない
    - 注 : C言語では  $<$  は左結合.  $3 < 2 < 1$  は  $(3 < 2) < 1$  と同じ.  $(3 < 2)$  の結果は 0 (false) なので,  $(3 < 2) < 1 = 0 < 1 = 1$  となる.  $0 < x \&\& x < 10$  と,  $0 < x < 10$  の意味は異なるので, 通常は  $0 < x < 10$  は使わない.
- GNU Bisonでは以下の記法で指定可能
  - %left            '-'      左結合
  - %right          '='      右結合
  - %nonassoc    '<'      非結合
  - GNU Bisonでは, 文法の変更であいまいさを除去せず, この記法を使った方が, 通常, 文法が簡潔で分かりやすい.

# 優先順位 (precedence)

- 優先順位=異なる演算子が複数ある時の計算順序
  - 例：通常， \*は+より優先順位が高い。 $1+2*3$ は $1+(2*3)$ と同じ
- 異なる演算子を同じ優先順位とすることもある
  - 例：通常， +と-は同じ優先順位。 $1+2-3$ は $(1+2)-3$ と同じ。

左結合  
なので



# C言語の演算子の優先順位と結合性

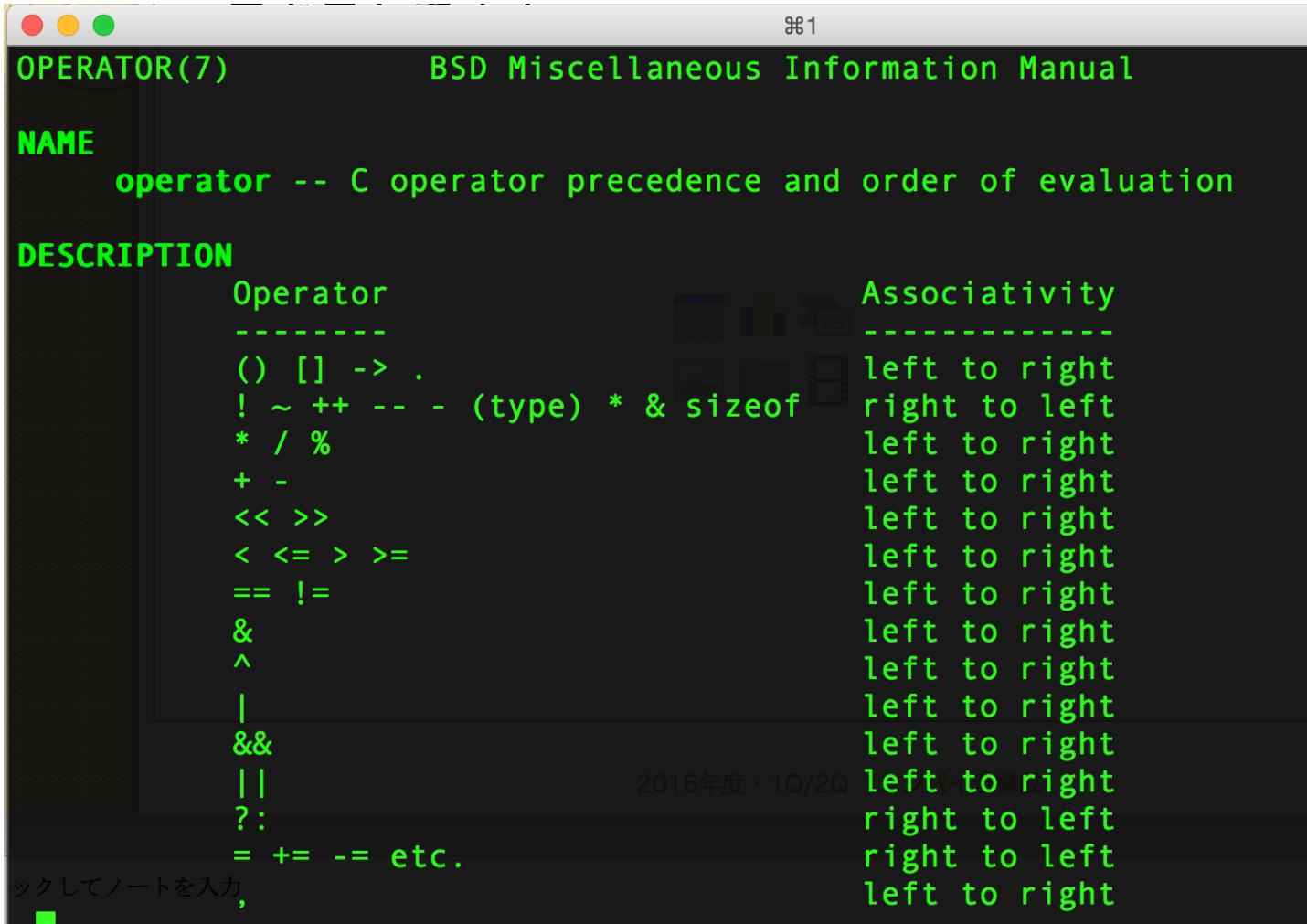
- 上の演算子が下より優先順位が高い
  - 単項演算子 (+, -, \*, &) は2項演算子の場合より高優先度

演算子	結合性
() [] -> . 後置++ 後置--	左結合
! ~ + - * & sizeof 前置++ 前置--	右結合
(型)	左結合
* / %	左結合
+ -	左結合
<< >>	左結合
< <= > >=	左結合
== !=	左結合
&	左結合
^	左結合
	左結合
&&	左結合
	左結合
?:	右結合
= += -= *= /= %= &= ^=  = <<= >>=	右結合
,	左結合



# man operator

オンラインマニュアルで確認可能  
p.16と微妙に違う…



Operator	Associativity
( ) [ ] -> .	left to right
! ~ ++ -- - (type) * & sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
	left to right
? :	right to left
= += -= etc.	right to left
	left to right

ツクしてノートを入力、

# BNF：バッカス・ナウア記法

- 生成規則（構文規則）を記述する記法
  - p.6の記法とほぼ同じ。→の代わりに : や ::= を使う
  - 同じ左辺を持つ生成規則を | でまとめる（「または」と読む）

```
expression
: expression "+" expression
| expression "*" expression
| "(" expression ")"
| IDENTIFIER
| INTEGER
;
```

# あいまい性の除去 (1/6)

- 例 : p.3の文法から、あいまい性を除去した文法

```
additive-expression
: multiplicative-expression
| additive-expression "+" multiplicative-expression
;

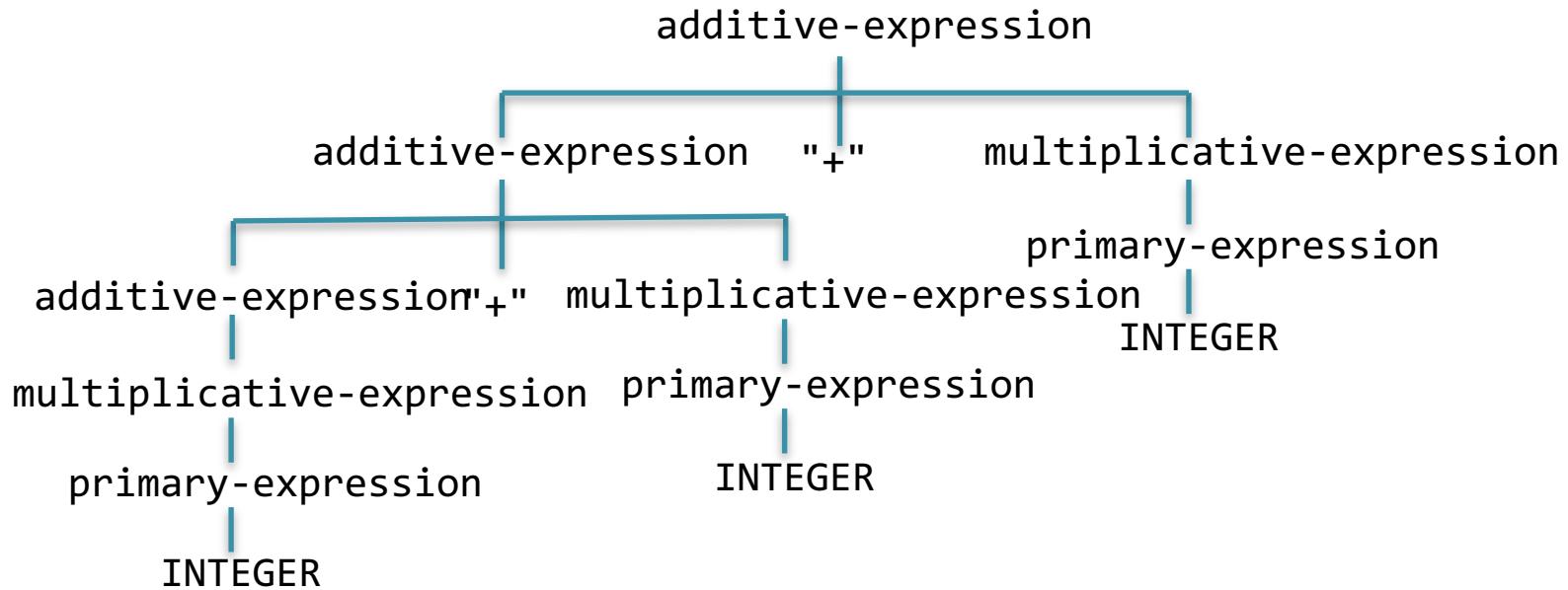
multiplicative-expression
: primary-expression
| multiplicative-expression "*" primary-expression
;

primary-expression
: "(" additive-expression ")"
| IDENTIFIER
| INTEGER
;
```



# あいまい性の除去 (2/6)

- 例： $1+2+3$ の構文木.  $(1+2)+3$ に相当

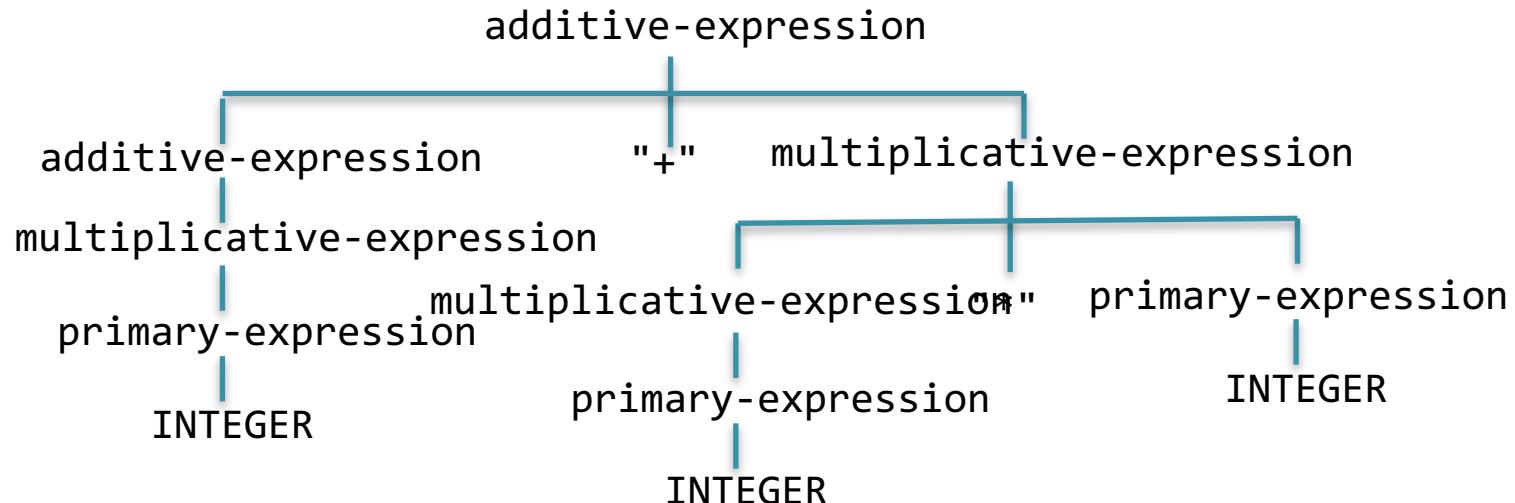


- ポイント
  - 式を加算式 (additive-expression) と  
加算式では無い式 (multiplicative-expression) に分けた.
  - additive-expressionを左再帰にして、左結合を実現.



# あいまい性の除去 (3/6)

- 例： $1+2*3$ の構文木。 $1+(2*3)$ に相当



- ポイント
  - `additive-expression`から`multiplicative-expression`は導出可能。逆はカッコを経由しないと導出不可能→優先順位の実現。
  - K&R「プログラミング言語C」巻末の文法が複雑なのは、この手法で、文法中に優先順位と結合性を実現しているから



# K&R本 卷末の文法（一部）

```
equality-expression:  
    relational-expression  
    equality-expression == relational-expression  
    equality-expression != relational-expression  
  
relational-expression:  
    shift-expression  
    relational-expression < shift-expression  
    relational-expression > shift-expression  
    relational-expression <= shift-expression  
    relational-expression >= shift-expression  
  
shift-expression:  
    additive-expression  
    shift-expression << additive-expression  
    shift-expression >> additive-expression  
  
additive-expression:  
    multiplicative-expression  
    additive-expression + multiplicative-expression  
    additive-expression - multiplicative-expression  
  
multiplicative-expression:  
    cast-expression  
    multiplicative-expression * cast-expression  
    multiplicative-expression / cast-expression  
    multiplicative-expression % cast-expression  
  
cast-expression:
```

出典：プログラミング言語C 第2版 ANSI規格準拠 (1989)  
B.W. カーニハン (著), D.M. リッチャー (著), 石田 晴久 (翻訳)  
ISBN-10: 4320026926, <http://www.amazon.co.jp/dp/4320026926/>

# あいまい性の除去 (4/6)

- （文法の書き替えではなく）規則で除去
  - ツールが自動的に変換
- 例：GNU Bisonによる定義。

```
%left '+' 左結合.  
%right '*' 右結合.+より優先順位が高い.  
%%  
expression 以下は p.2 の文法と同じ.  
| expression '+' expression  
| expression '*' expression  
| '(' expression ')'   
| IDENTIFIER  
| INTEGER  
;
```



# EBNF：拡張バッカス・ナウア記法

- 正規表現の記法を取り入れ、簡潔に表現
  - $(\alpha)^*$   $\alpha$ の0回以上の繰り返し
  - $(\alpha)^+$   $\alpha$ の1回以上の繰り返し
  - $[\alpha]$   $\alpha$ または空 ( $\varepsilon$ ) ,  $(\alpha|\varepsilon)$ と同じ
  - $(\alpha)$  グループ化
- 記述力はBNF（文脈自由文法）と同じ
  - 以下の書き換えでEBNFをBNFに書き替え可能だから
    - $A: \alpha (\beta)^* r; \Rightarrow A: \alpha B r; B: \beta B | \varepsilon;$
    - $A: \alpha (\beta)^+ r; \Rightarrow A: \alpha \beta B r; B: \beta B | \varepsilon;$
    - $A: \alpha [\beta] r; \Rightarrow A: \alpha B r; B: \beta | \varepsilon;$
    - $A: \alpha (\beta) r; \Rightarrow A: \alpha B r; B: \beta;$ 
      - ただし、上のBは新たに導入した非終端記号
      - 例えば、 $A: B (C|D) E;$  は、 $A: BZE; Z: C|D;$  に書き替える

# ぶらぶらelseの対処

- 右辺の共通する左部分をくくり出す

statement:

```
: "if" "(" expression ")" statement
| "if" "(" expression ")" statement "else" statement
;
```

statement:

```
→ : "if" "(" expression ")" statement [ "else" statement ]
;
```

- 最初のstatement処理終了時に、次のトークンが  
**else**なら、if-else文として処理する
  - 先読み記号が必要（後述）
- GNU BisonではそのままでOK
  - shift-reduce衝突（後述）の際、デフォルトではshiftが選択されるから

# 共通部分のくくりだし

- 一般的なくくりだし方法

A:  $\alpha \beta_1 | \cdots | \alpha \beta_m | r_1 | \cdots | r_n;$



A:  $\alpha B | r_1 | \cdots | r_n;$   
B:  $\beta_1 | \cdots | \beta_m;$

- ただし,  $r_1 \cdots r_n$  は  $\alpha$  で始まらない記号列. B は新たな非終端記号.
- 右辺の選択子中に共通の左部分が無くなるまで, 繰り返す.



# 文脈自由文法は正規表現より強力

- 例：文脈自由文法ではカッコの対応が取れた式を定義可能
  - 文脈自由文法では, expression: "(" expression ")" ;
  - 正規表現では表現できない. "("\* expression ")"\* ではダメ.
    - 正規表現は任意の大きさの数を数えられない←有限状態だから
- なぜ字句解析に正規表現を使うのか
  - たいてい字句解析には正規表現で十分
  - 正規表現の方が字句解析プログラムの効率が良い
  - (コメントはどこでも出現可能 → 文脈自由文法で書きにくい)

# 文脈自由文法は変数の二重定義をチェックできない

- 文脈自由文法は非終端記号を文脈（前後関係）と独立に展開する
  - 例：int x; int x; を導出可能
  - xが定義済みかどうか関係無く、2番目のint x;を導出できるから
- 普通、以下は構文解析ではチェックしない（意味解析でチェック；後述）
  - 変数や関数の二重定義が無いこと
  - 未定義の変数や関数を使ってないこと
  - 変数や関数の、定義と参照の型が合致すること
- 普通、以下は意味解析でもチェックしない（できない）
  - プログラムが停止すること
  - プログラムが意図通りの計算結果を出力すること



## 例題：XC言語



# 簡易言語XCの概要

- C言語の真のサブセット
  - XCコードはCコンパイラで処理可能
- コンパクトで理解しやすい構文規則
- 本質的な言語機能を保持
  - 型 : int, long, char, void, 関数型, ポインタ型.
  - 制御構造 : if-else, while, goto, return
  - 関数呼出し : 引数, ライブラリ関数呼び出し可.



# 簡易言語XCの文法 (1/2)

translation\_unit

: ( typeSpecifier declarator ( ";" | compoundStatement ) )\* ;

typeSpecifier

: "void" | "char" | "int" | "long";

declarator

: IDENTIFIER | "\*" declarator | "(" declarator ")"

| declarator "(" [ parameterDeclaration  
                  ( "," parameterDeclaration )\* ] ")";

parameterDeclaration

: typeSpecifier declarator ;

statement

: [ expression ] ";"        // 式文(expression statement)

| IDENTIFIER ":" statement

| compoundStatement

| "if" "(" expression ")" statement [ "else" statement ]

| "while" "(" expression ")" statement

| "goto" IDENTIFIER ";"

| "return" [ expression ] ";" ;



# 簡易言語XCの文法 (2/2)

compound\_statement

: "{" (type\_specifier declarator ";")\* ( statement )\* "}" ;

expression

: IDENTIFIER | INTEGER | CHARACTER | STRING  
| unary\_operator expression  
| expression binary\_operator expression  
| expression "(" [ argument\_expression\_list ] ")"  
| "(" expression ")" ;

unary\_operator

: "&" | "\*" | "+" | "-" | "!" ;

binary\_operator

: "=" | "||" | "&&" | "==" | "<" | "+" | "-" | "\*" | "/" ;

argument\_expression\_list

: expression ( "," expression )\* ;



# 構文の用語

用語	意味
translation unit	翻訳単位（ファイルのこと）
declaration	宣言（宣言全体。例：int *(*fp)();）
declarator	宣言子（int *(*fp)(); のうち、*(*fp)()の部分）
type specifier	型指定子（int *(*fp)(); のうち、intの部分）
compound statement	ブロック文、複合文（ブレース {} で囲まれる部分）
parameter	仮引数（formal parameter とも言う）
argument	実引数（actual argument とも言う）
statement	文
expression	式
expression statement	式文（x=10; や printf ("\n");）
identifier	識別子（名前のこと）
binary operator	二項演算子（例：4-3の「-」）
unary operator	単項演算子（例：-3 の「-」）

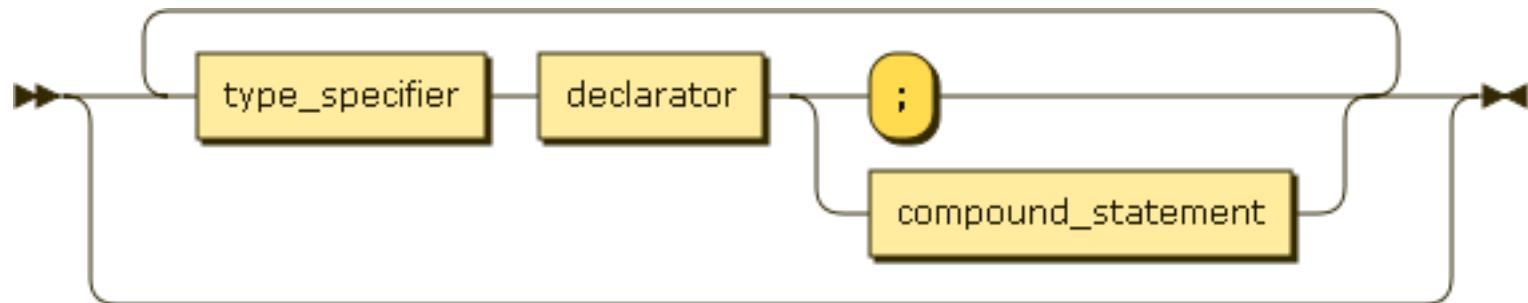
# 式文

- 式文 = 式にセミコロンをつけると文になる
  - C言語で「文はセミコロンで終わる」という説明は間違い。  
if文やブロック文などはセミコロンで終わらない
- 本質的に副作用(side effect)のある式につかう
  - 例 : `x = 10; printf ("\n");`
- 副作用が無くても文法的には合法（でも意味は無い）
  - 例 : `x + 10;`
  - `x+10`という式の評価結果（計算した値）は単に捨てられる

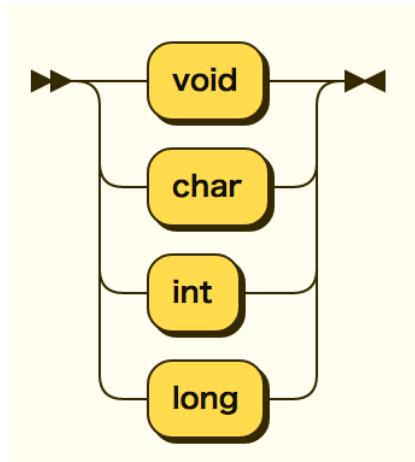


# レイルロード・ダイアグラム (1/7)

- translation\_unit



- type\_specifier

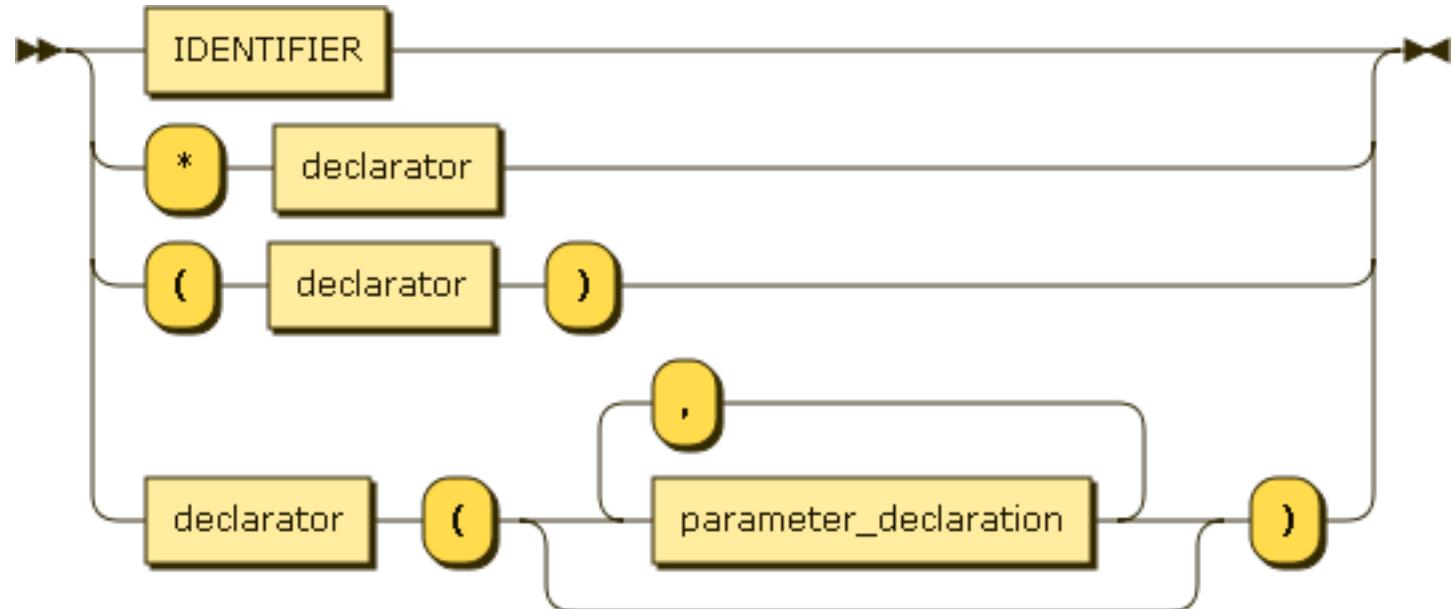


generated with <http://bottlecaps.de/rr/ui>



# レイルロード・ダイアグラム (2/7)

- declarator



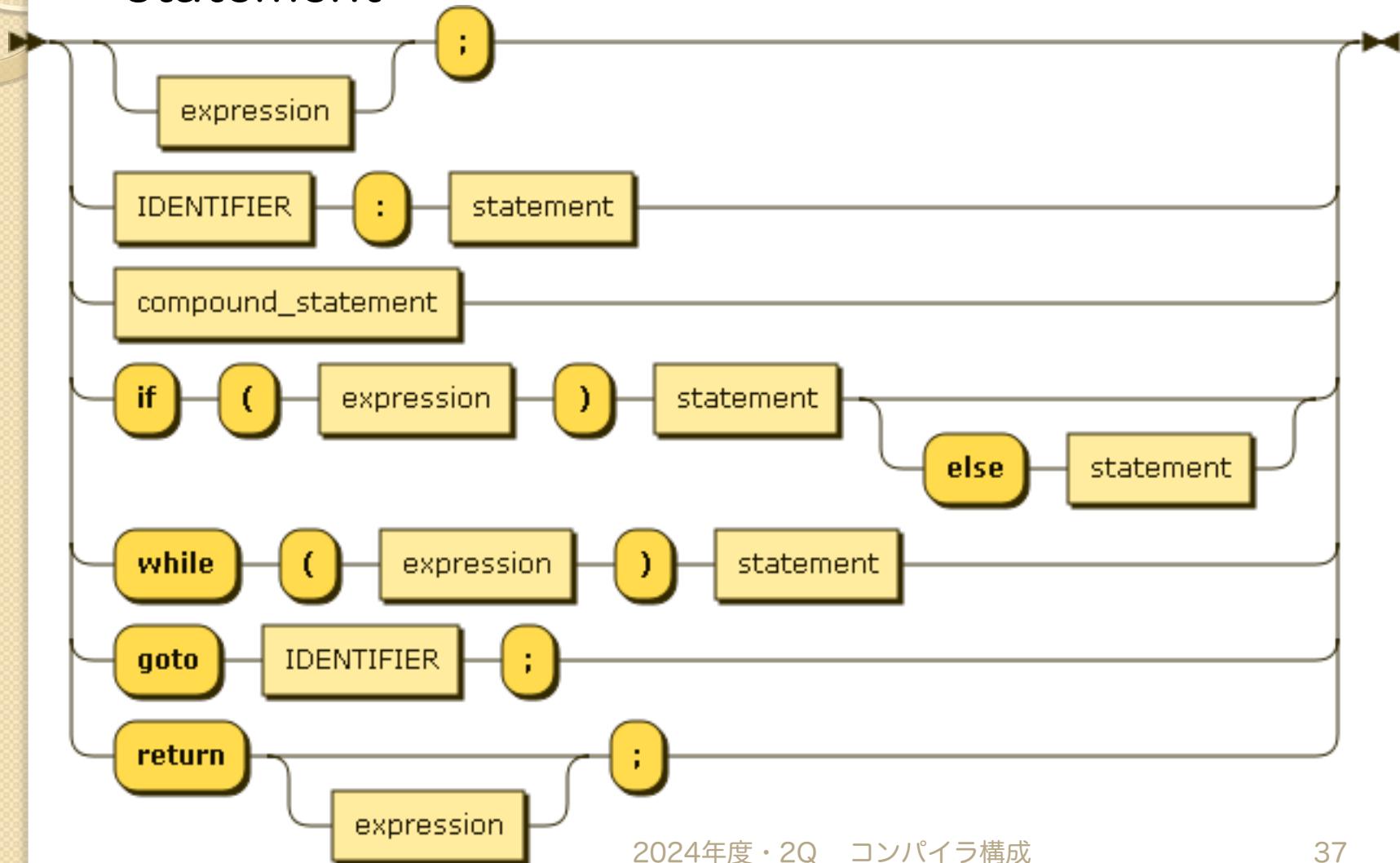
- parameter\_declaration





# レイルロード・ダイアグラム (3/7)

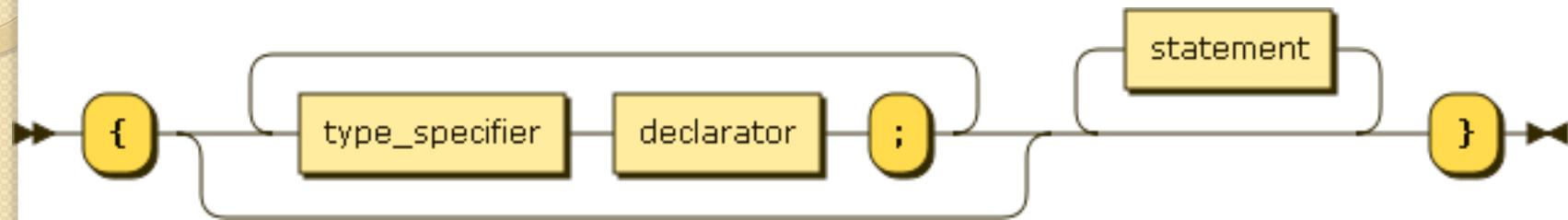
- statement





# レイルロード・ダイアグラム (4/7)

- compound\_statement



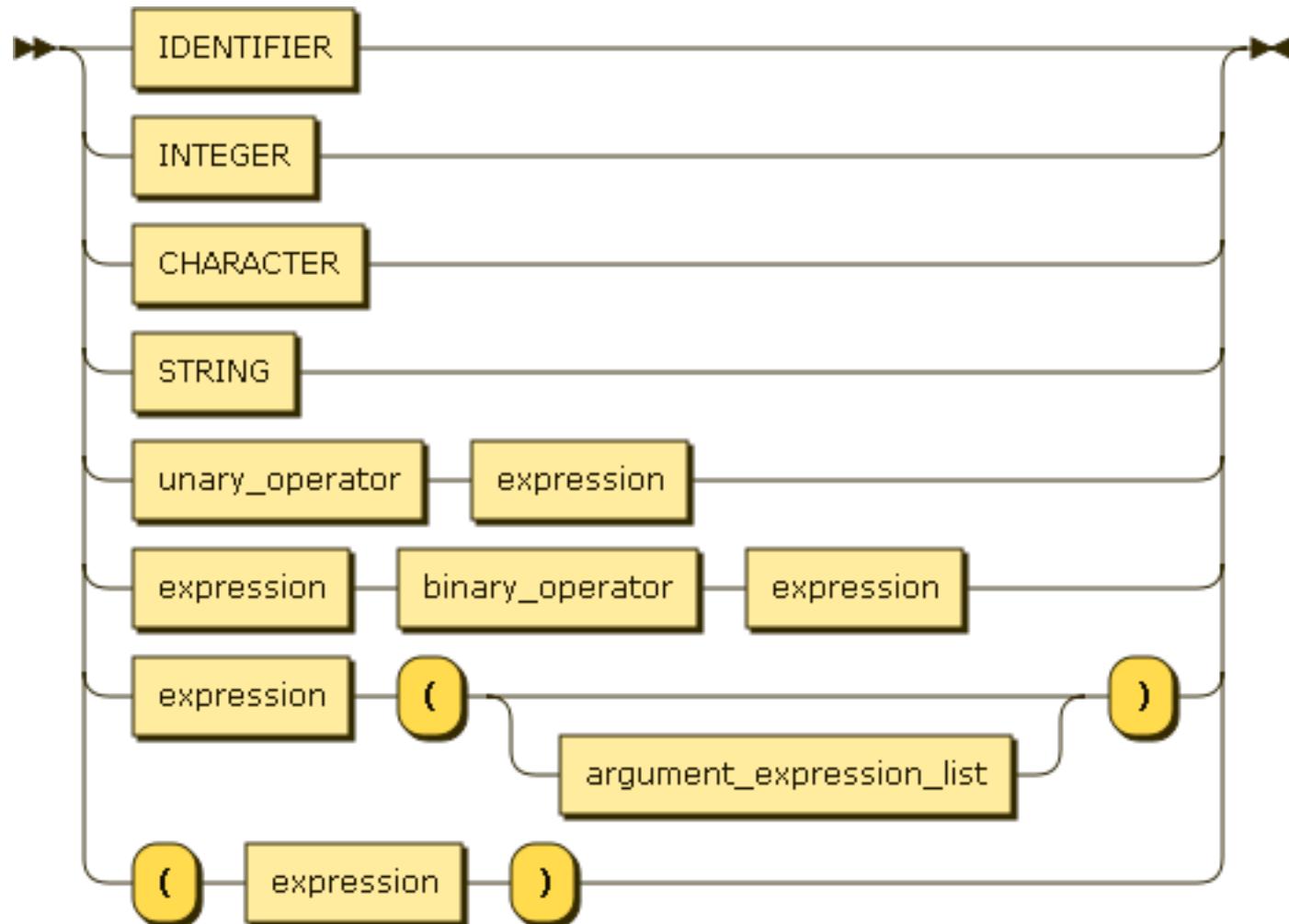
- 注：C言語ではC99以降、文の後に宣言が来てもOK

```
#include <stdio.h>
int main (void)
{
    printf ("hello\n");
    int x = 99;
    printf ("%d\n", x);
}
```



# レイルロード・ダイアグラム (5/7)

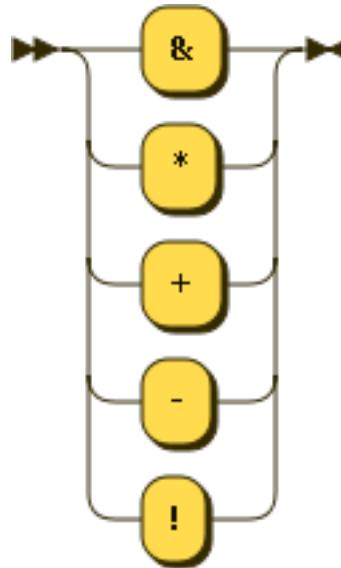
- expression



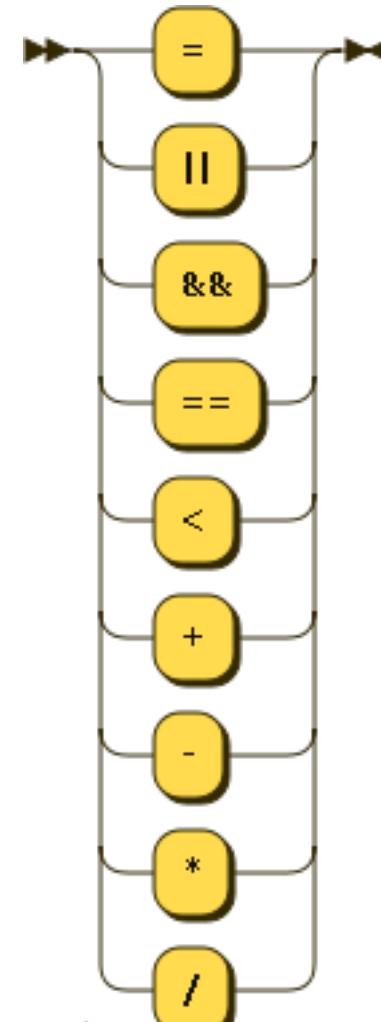


# レイルロード・ダイアグラム (6/7)

- unary\_operator



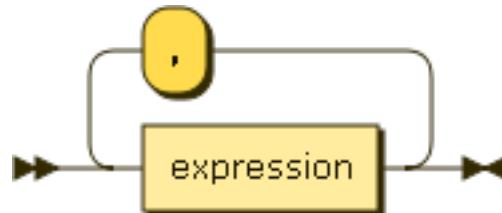
- binary\_operator





# レイルロード・ダイアグラム (7/7)

- argument\_expression\_list





# XCコード例：ポインタ（1）

- ポインタのポインタのポインタ…も使える
  - レイルロード・ダイアグラムで確認せよ

```
int printf ();
int main ()
{
    int i;
    int *p1;
    int **p2;
    int ***p3;
    i = 999;
    p1 = &i;
    p2 = &p1;
    p3 = &p2;
    printf ("%d\n", ***p3);
}
```



# XCコード例：ポインタ (2)

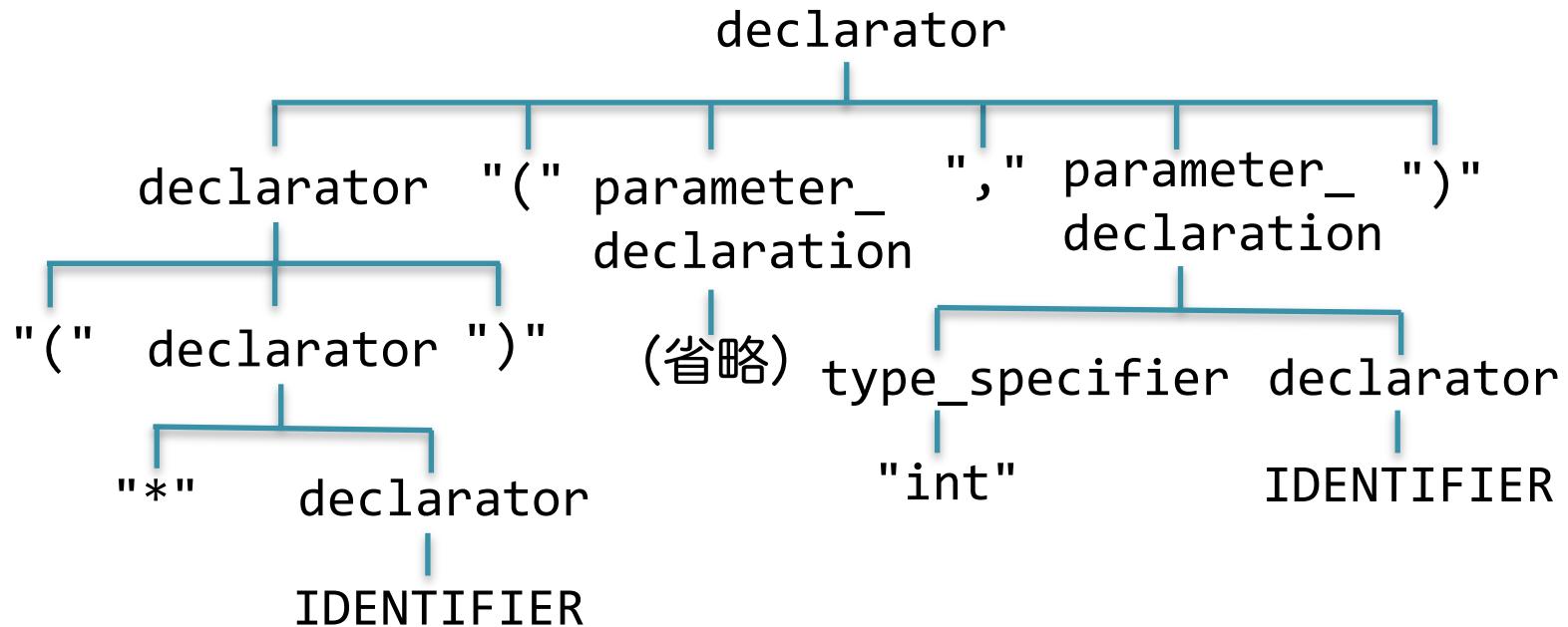
- 関数ポインタも使える
  - レイルロード・ダイアグラムで確認せよ

```
int printf ();
int add (int a, int b)
{
    return a + b;
}
int main ()
{
    int (*fp)(int a, int b);
    fp = add;
    printf ("%d\n", fp (10, 20));
}
```

- C言語では `int (*fp)(int, int);` でもOK.  
XCでは変数名必須. これはXCの文法簡単化のため.



# (\*fp)(int a, int b) の構文木



# \*と引数カッコの優先度

- `int (*foo)();`
  - `foo`は、`int`を返す関数へのポインタ
- `int *(foo());`
  - `int *foo();`と同じ（引数のカッコは、`*`よりも優先度高い）
  - `foo`は、`int`へのポインタを返す関数



# XCコード例：バブルソート

```
void bubble_sort (int *data, int size)
{
    int i; int j;
    i = size - 1;
    while (0 < i) {
        j = 0;
        while (j < i) {
            if (*(data + (j+1)) < *(data + j))
                swap (data + j, data + (j + 1));
            j = j + 1;
        }
        i = i - 1;
    }
}
```

# XCでの配列の扱い

- `data[j]` ではなく `*(data + j)` と書く
  - C言語の式中で、`data[j]` と `*(data+j)` は同じ意味
  - 「ポインタ+整数」はポインタ演算 (pointer arithmetic)
- 配列確保は `int data[10];` ではなく、  
`data = malloc (4 * 10);` とする

```
#include <stdio.h>
int main (void)
{
    int a[] = {10, 20, 30};
    printf ("%d, %d, %d, %d\n",
            a[2], *(a+2), *(2+a), 2[a]); // 全部同じ
}
```

```
% gcc -g foo.c
% ./a.out
30, 30, 30, 30
```



# ポインタ演算

- `int *p; int i;` のとき,  $p + i$  の意味は,
  - $p$ が指している先のオブジェクト $i$ 個分,  
 $p$ 中のアドレスを増やした値.
  - つまり 「 $p$ 中のアドレス +  $i * \text{sizeof } (*p)$ 」 .
  - C言語でも同じ.

```
int printf ();
void *malloc ();
int main (void)
{
    int *p;
    p = malloc (100);
    printf ("%p, %p\n", p, p + 3);
}
```

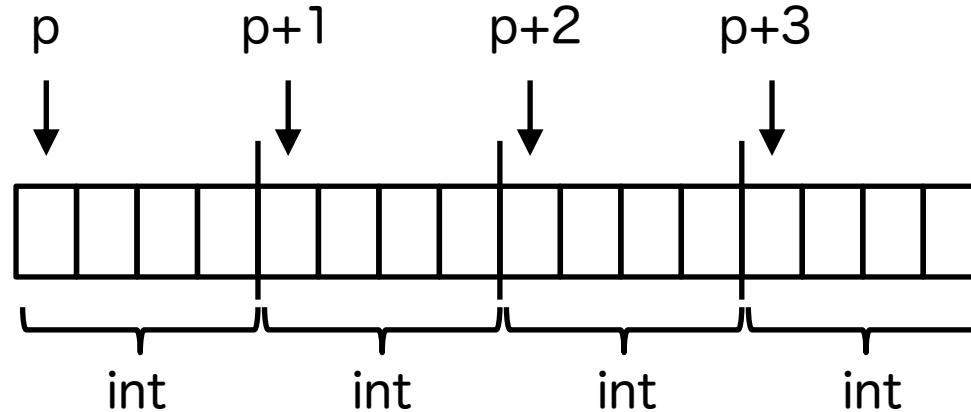
12増えてるのがポイント.  
 $\text{sizeof(int)}=4$  だったので,  
 $0x7f9949c04cf0 + (3 * 4)$   
を計算.

```
% ./a.out
0x7f9949c04cf0, 0x7f9949c04cf0
```



# ポインタ演算：図解

- `int *p; sizeof(int)==4バイト` を仮定
- `p+3`は「`int`を3個分、ポインタをずらした場所」





# XCで書けないこと (1/2)

- 「char, int, long, void, ポインタ型, 関数型」  
以外の型
  - int x; int \*p; int (\*fp)();
  - ✗ unsigned, const, static, typedef など
  - ✗ 配列, 構造体, 共用体, 列挙型 (enum)
- 1つの変数宣言で複数の変数宣言
- 変数宣言時の初期化
  - int x; int y; int z; z = 10;
  - ✗ int x, y; int z = 10;



# XCで書けないこと (2/2)

- 一部の制御文

- ○ if, if-else, while, goto, return
  - × for, do-while, switch, break, continue

- 一部の演算子

- ×  $a \leq 0$  ( $a < 0 \ || \ a == 0$  で代替)
  - ×  $a > 0$  ( $0 < a$  で代替)



# printf の呼出し

注：ARMだとうまくいかない

<https://qiita.com/hotpepsi/items/bd1f496411a2df74b704>

<https://qiita.com/qnighy/items/be04cfe57f8874121e76>

- 「int printf ();」を使用前に宣言しておく。
- int printf (const char \*, ...); とは宣言できない。
  - XCが可変長引数やconstを未サポートのため。
- gccのこの警告は無視してよい。
  - XCのプログラムはgccでもコンパイル可能

```
foo.c:1:5: warning: incompatible redeclaration of  
library function 'printf'  
int printf ();  
^
```

- 引数を持つプロトタイプ宣言をさぼると「既定の実引数拡張」が起こる。
  - 例：char は int に、 float は double に暗黙に変換。
  - 可変長引数でも「既定の実引数拡張」が起こるので問題なし。

既定の実引数拡張 default argument promotion



# #include <stdio.h> って？

- printfのプロトタイプ宣言をすれば、#include <stdio.h> は無くてもコンパイル&実行可能

```
int printf (const char *, ...);
int main (void)
{
    printf ("hello\n");
}
```

- 標準ヘッダファイルには
  - プロトタイプ宣言や定数の定義（NULLやEOF）が入ってる
  - ライブラリ関数（例：printf）の実体は入っていない
    - 実体は、例えば /usr/lib/libc.a にある（静的ライブラリの場合）



## 構文木の構築



# 構文木のデータ表現

- JSONとしてのデータ表現
- 構造体としてのデータ表現

# 可変長引数を持つ関数の定義 (1/2)

- stdarg.hと、マクロ va\_start, va\_end, va\_argを使う
  - 固定長の引数は最低1個必要 (C23からは不要)

```
#include <stdio.h>
#include <stdarg.h>
int sum (int num, ... ) // numは可変長引数の個数
{
    va_list ap;           // 可変長引数リスト
    int i, n, total = 0;
    va_start (ap, num); // 可変長引数リストの初期化
    for (i = 0; i < num; i++) {
        n = va_arg (ap, int); // 引数を1つ取り出す
        printf ("%d, %d\n", i, n);
        total += n;
    }
    va_end (ap);          // 可変長引数リストの終了
    return total;
}
```

va\_arg.c

# 可変長引数を持つ関数の定義 (2/2)

- 呼び出し例と実行例

```
int main (void)
{
    printf ("total=%d\n", sum (0));
    printf ("total=%d\n", sum (3, 10, 20, 30));
    printf ("total=%d\n", sum (5, 10, 20, 30, 40,
50));
}
```

```
% ./a.out
total=0
0, 10
1, 20
2, 30
3, 40
2, 30
total=60
```

```
0, 10
1, 20
2, 30
3, 40
4, 50
total=150
```

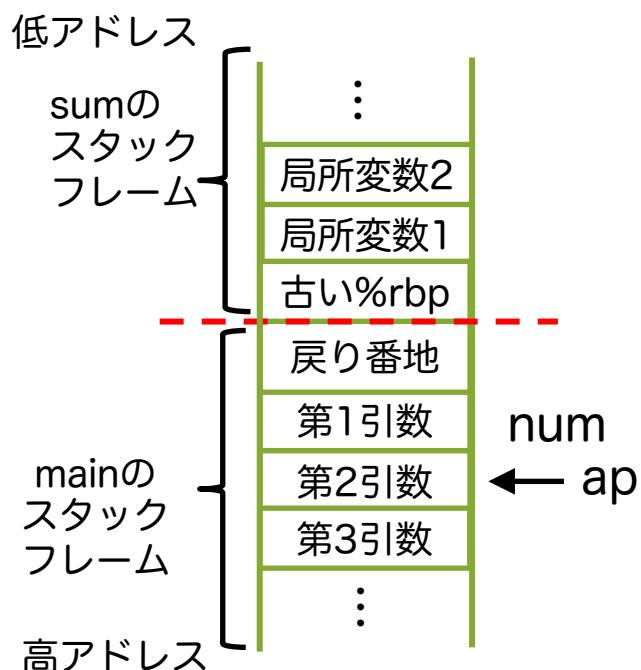


注：x86-64の第1～6引数はレジスタ渡しなのでこの図とは異なる。  
また引数間のパディングも考慮する必要あり。

# va\_start, va\_arg の図解

```
va_list ap;  
va_start (ap, num); // 可変長引数リストの初期化  
n = va_arg (ap, int); // 引数を1つ取り出す
```

例えば仮にスタックレイアウトが  
こうなっている場合



- **va\_start (ap, num);**
  - ap = (char\*)&num + sizeof (num)  
// apは可変長引数の先頭を指す
  - コンパイラは可変長引数の先頭と引数numのオフセットを知っている
- n = **va\_arg (ap, int);**
  - num = \*(int \*)ap;  
// アドレスapからintを取り出しnumに代入
  - ap += sizeof (int);  
// 次の可変長引数を指すようにapを増加



注：\*\*child の気持ちは \*child[]. でも\*child[]と書くと文法エラー。  
foo[]と書けるのは仮引数，初期化付きの配列宣言，extern宣言のみ

# 構文木のプログラム表現（構造体）

```
struct AST {  
    char          *ast_type;      // 生成規則を区別  
    struct AST   *parent;        // 親へのバックポインタ  
    int           nth;           // 自分が何番目の兄弟か  
    int           num_child;     // 子ノードの数  
    struct AST   **child;        // 子ノードポインタの配列  
    char          *lexeme;        // 葉ノード用（整数，文字，  
    文字列の定数，識別子）  
};
```

AST.h

- ast\_typeはenum型でも可。
- childとlexemeは、unionにしても良い。同時には使わないから  
union {  
 struct AST \*\*child;  
 char \*lexeme;  
} u;



メモリ効率はこちらの方が良い

# 余談：enum型→文字列の変換法

- 1. 配列を準備する

```
#include <stdio.h>
enum Junken { ROCK, PAPER, SCISSORS, };
char *JunkenStr [] = { "ROCK", "PAPER", "SCISSORS", };
int main (void)
{
    printf ("%s=%d\n", JunkenStr [PAPER], PAPER);
}
```

- 2. C前処理系の#演算子を使う (関数マクロ中のみ)

```
#include <stdio.h>
#define STR(a) (#a)
enum Junken { ROCK, PAPER, SCISSORS, };
int main (void)
{
    printf ("%s=%d\n", STR(PAPER), PAPER);
}
```



# 構文木のプログラム表現 (1/4)

```
#include <stdio.h>
#include <stdarg.h> // for 可変長リスト
#include <stdlib.h> // for malloc
#include <string.h> // for strcmp
#include "AST.h"

static struct AST*
create_AST (char *ast_type, int num_child, ...)
{
    va_list ap;
    struct AST *ast;
    ast = malloc (sizeof (struct AST));
    ast->parent = NULL;
    ast->nth      = -1;                                malloc がNULLを返した際の
    ast->ast_type = ast_type;                          エラー処理は省略
    ast->num_child = num_child;
    ast->lexeme = NULL; // 内部ノードなので、ここは空
    va_start (ap, num_child);
```



# 構文木のプログラム表現 (2/4)

```
if (num_child == 0) {
    ast->child = NULL;
} else {
    int i;
    ast->child = malloc (sizeof(struct AST *) * num_child);
    for (i = 0; i < num_child; i++) {
        struct AST *child = va_arg (ap, struct AST *);
        ast->child [i] = child;
        if (child != NULL) {
            child->parent = ast;
            child->nth    = i;
        }
    }
    va_end (ap);
    return ast;
}
```

# 構文木のプログラム表現 (3/4)

```
static struct AST*
create_leaf (char *ast_type, char *lexeme)
{
    struct AST *ast;
    ast = malloc (sizeof (struct AST));
    ast->parent      = NULL;
    ast->nth         = -1;
    ast->ast_type   = ast_type;
    ast->num_child  = 0;
    ast->child       = NULL; // 葉ノードなので、子どもは無し
    ast->lexeme     = lexeme;
    return ast;
}
```

# 構文木のプログラム表現 (4/4)

```
int main (void)
{
    struct AST *ast1, *ast2, *ast3, *ast4, *ast5, *ast6;
    // (10+x)*y の構文木を作る
    ast1 = create_leaf ("AST_int", "10");
    ast2 = create_leaf ("AST_id", "x");
    ast3 = create_AST ("AST_add", 2, ast1, ast2);
    ast4 = create_AST ("AST_paren", 1, ast3);
    ast5 = create_leaf ("AST_id", "y");
    ast6 = create_AST ("AST_mult", 2, ast4, ast5);
    return 0;
}
```

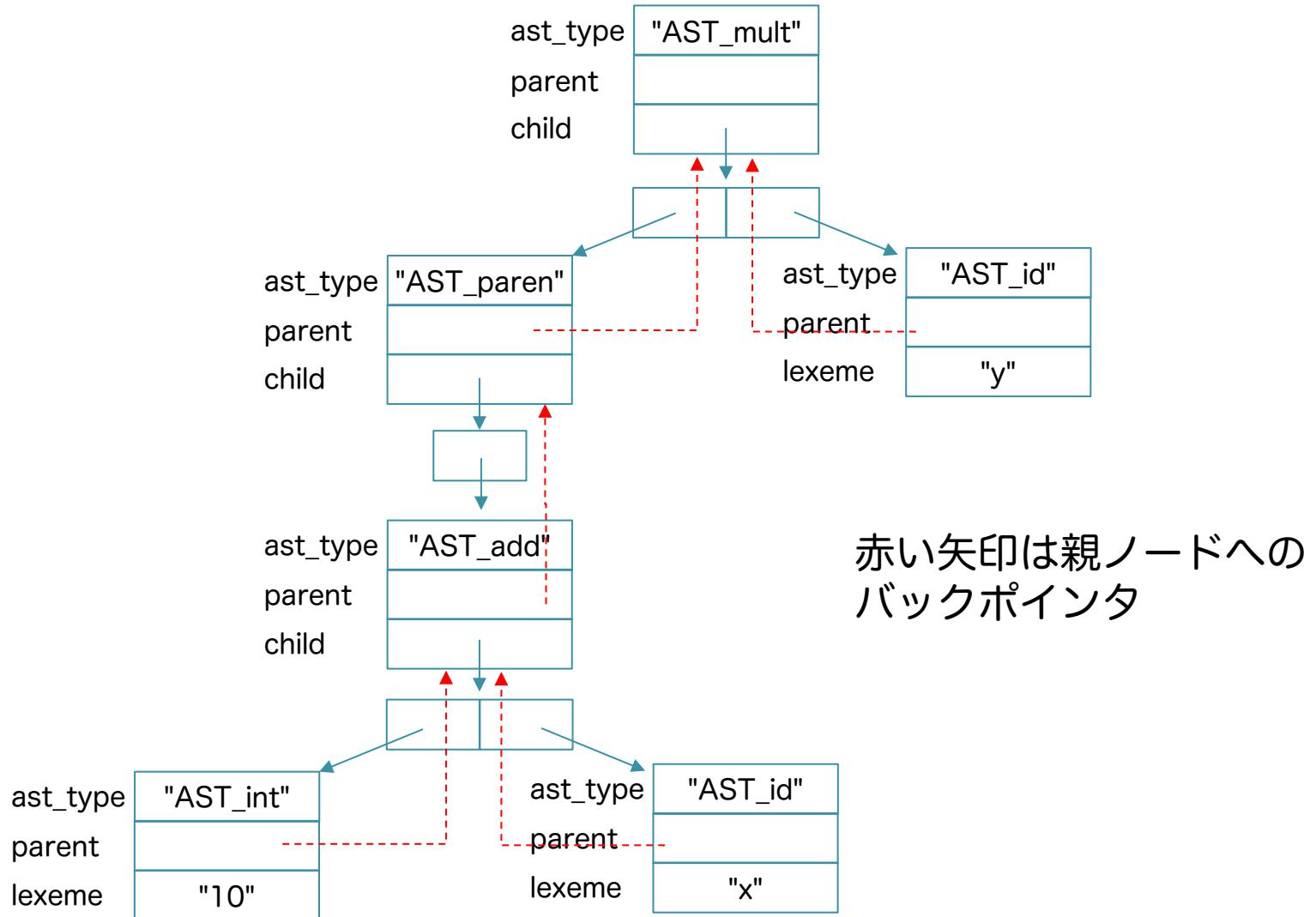
可変長引数ではなく、struct \*AST型の配列を使う方法もあり



# 構文木のプログラム表現（実行例）

```
% gcc -g AST.c
% lldb ./a.out    (lldbからの表示はいろいろ省略)
(lldb) b -l 60    60行目にブレークポイントを設定
(lldb) r          実行開始
      59         ast6 = create_AST ("AST_mult", 2, ast4, ast5);
-> 60         return 0;
      61     }
(lldb) p *ast6->child[0] 構造体の中身を表示
(AST) $0 = { ast_type = "AST_paren", parent =
0x0000000100104f10, nth = 0, num_child = 1, child =
0x0000000100104eb0, lexeme = 0x0000000000000000 }
(lldb) p *ast6->child[0]->child[0]->child[1]
(AST) $1 = { ast_type = "AST_id", parent =
0x0000000100104e40, nth = 1, num_child = 0, child =
0x0000000000000000, lexeme = "x" }
(lldb) q
%
```

# 構文木のプログラム表現（構文木）





# 余談：struct は typedef すべきか

- 好みだけど、struct は typedef しない方が好き
  - typedef すれば、いちいち struct を書かずにする
  - でも、struct と書いてある方が「これは構造体？」と悩まずにする

```
typedef struct AST AST; // 名前空間が別なので同名OK
int main (void)
{
    AST          *ast = create_AST ("foo", 0);
    struct AST *ast = create_AST ("foo", 0);
}
```

- 同じ理由で、enumやunionも typedef しない方が好き

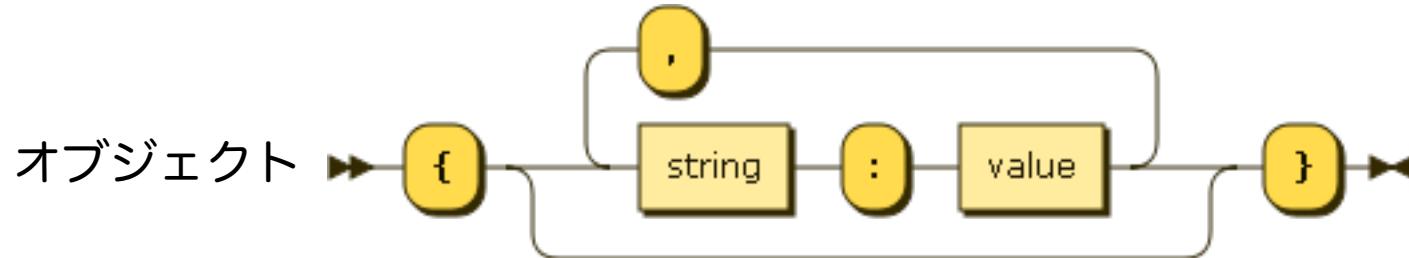


# JSON

- JSON=JavaScript Object Notation
- 構造的なデータを表現するテキスト記法
  - 類：XML, LispのS式, CSV
- 単純で、読み込みが容易
  - JavaScriptではevalするだけで読める



# JSONの文法



値 ➡ string

number

object

array

true

false

null

配列 ➡ [ value ]

注：文字列で次の文字は要エスケープ  
\" \\\\" \\/



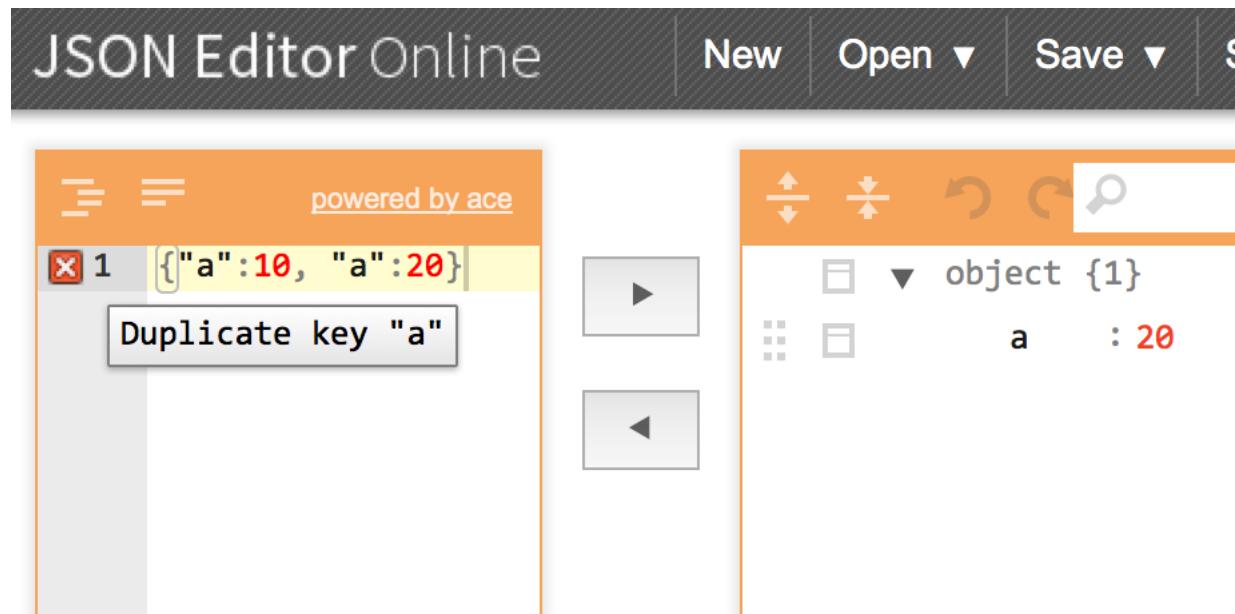
# JSON (例)

- { "name": "Yamada", "age": 20 }
- { "array": [10, 20, 30] }
  - 配列も値
- { "array": [10, "hello", {"age": 20}, [30, 40] ] }
- 配列の中身は型が別でも良い
- { "object": {"age":20}}
  - オブジェクトも値（オブジェクトの入れ子可）
- NG: { "a": 10, "a": 20}
  - コロン（:）の左の文字列（キー）は重複不可



# JSON editor onlineが便利

- <http://jsoneditoronline.org/>
- インストール不要，構文チェック，木構造表示



# JSONでの構文木表現

- { "ノード名": [ 子ノード<sub>1</sub>, 子ノード<sub>2</sub>, …] } と表現
  - 配列ではなくオブジェクトを使うと、同名キー問題が面倒
- 例 : x+10のJSON表現

```
{"expression": [
    {"expression": [
        {"IDENTIFIER": "x"}],
    {"+": "+"},
    {"expression": [
        {"INTEGER": [10]}]
}]}
```

```
▼ object {1}
  ▼ expression [3]
    ▼ 0 {1}
      ▼ expression [1]
        ▼ 0 {1}
          IDENTIFIER : x
    ▼ 1 {1}
      +
      :
    ▼ 2 {1}
      ▼ expression [1]
        ▼ 0 {1}
          INTEGER : 10
```



# 再帰下降型構文解析 LL(1)



# 概要

- 再帰下降構文解析 (recursive descent parsing)
  - 文法の非終端記号ごとに解析関数 (parsing function) を作る
  - 文法と構文解析プログラムの対応が分かりやすい
  - LL(1)文法 = 「**バックトラック無し, 先読み記号1つ**」で再帰下降構文解析プログラムを作れる文法クラス
- LL(1)文法 (LL=Left-to-right leftmost derivation)
  - LL(1)の気持ち = 構文木を解析中に非終端記号Aと先読み記号aが与えられた時, **Aを展開する生成規則が一意に決まる**文法
  - 形式的には **first, follow, director**集合で定義. LR(1)でも使用
  - 次の文法はLL(1)では無い
    - 左再帰的な生成規則を含む文法. 例 : **exp: exp "+" exp**
    - あいまいな文法



# 例題の文法G1

- 次の文脈自由文法G1を考える
  - +, \*, (,), i は終端記号. E, E2, T, T2, F は非終端記号.  
E は開始記号

```
E: T E2;  
E2: "+" T E2 | ε;  
T: F T2;  
T2: "*" F T2 | ε;  
F: "(" E ")" | "i";
```

- EBNFでは以下と書ける

```
E: T ("+" T)*;  
T: F ("*" F)*;  
F: "(" E ")" | "i";
```

単純に

$E: E "+" E \mid E "*" E \mid "(" E ")" \mid "i";$   
と書くと、左再帰＆あいまいとなる。  
左の文法では

- ・左再帰を除去
- ・+より\*の優先度が高い

を満たす等価な文法に変換している

Eはexpression (式)

Tはterm (項)

Fはfactor (因子) の略

因子を "\*" で組み合わせると項になり、  
項を "+" で組み合わせると式になる



# G1の再帰下降型構文解析器 1 (1/3)

parse-G1.c

```
static char *ptr;

static void print_NT_begin (char *NT_name)
{
    printf ("{\\"%s\\":[\n", NT_name);
}
static void print_NT_end (void)
{
    printf ("]},\n");
}
static void print_token (void)
{
    printf ("{\\"%c\\": \\"%c\\"},\n", ptr [0], ptr [0]);
}
```

- ・構文木をJSON形式で出力するための関数群（ただし、末尾カンマ問題あり）
- ・簡単のため、構文解析する文字列はptrが指し、トークンはすべて1文字と仮定



# G1の再帰下降型構文解析器 2 (2/3)

```
static void parse_error (void) { // 構文エラー時に呼び出す
    fprintf (stderr, "parse error: %s\n", ptr); exit (1);
}
static char lookahead (int i) { // i番目の先読み記号
    return ptr [i - 1];
}
static char next_token (void) { // 先読み記号を返し, 次に進む
    return *ptr++;
}
// 引数cが先読み記号と同じなら, 表示して次に進む
static void consume_token (char c) {
    if (lookahead (1) == c) {
        print_token ();
        next_token ();
    } else {
        parse_error ();
    }
}
```

parse-G1.c



# G1の再帰下降型構文解析器 1 (3/3)

```
// 生成規則 E2: "+" T E2 | ε の解析関数
static void parse_E2 (void) {
    print_NT_begin ("E2");
    switch (lookahead (1)) {
        case '+':
            consume_token ('+');
            parse_T ();
            parse_E2 ();
            break;
        case '\0': // εの時、先読み記号は'\0'か'
            break; // 何もしない
        case ')':
            default: parse_error (); break;
    }
    print_NT_end ();
}
```

parse-G1.c

- ・他の関数 (parse\_E, parse\_E2, parse\_T, parse\_T2) は同様なので省略
- ・生成規則からの機械的な変換方法は後述

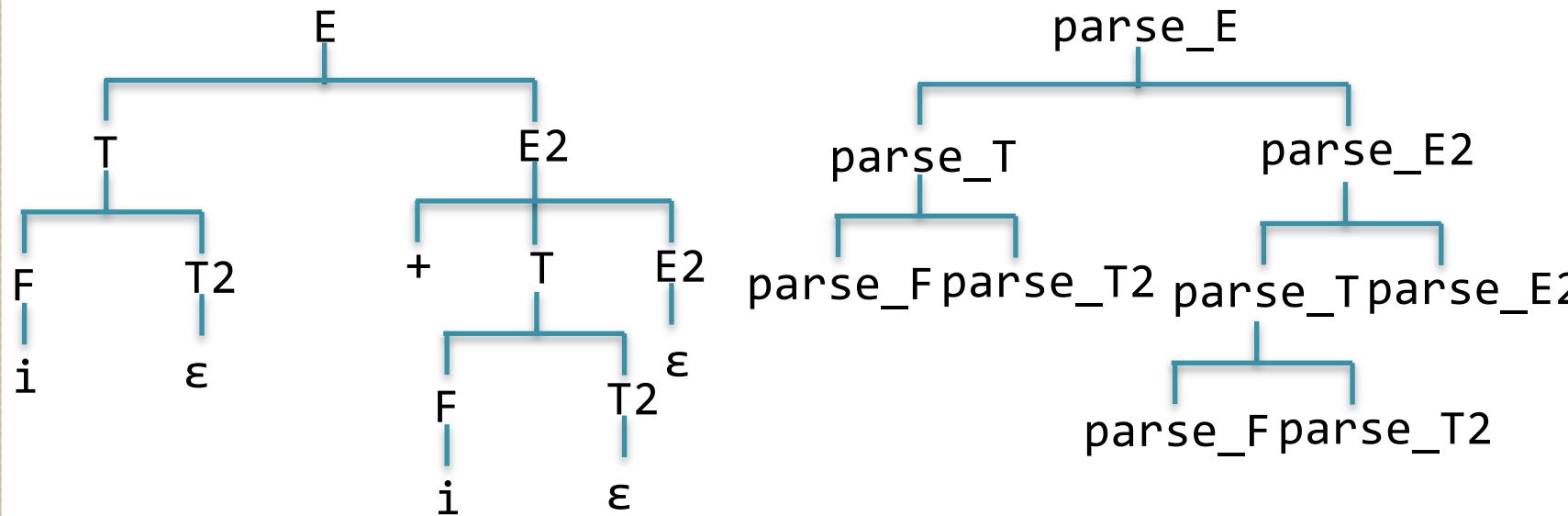


# parse-G1.c 実行例

```
% gcc -g parse-G1.c
% ./a.out "i+i" > in      i+iを構文解析
% ./rm-tail-commma.csh < in > out 末尾カンマ（後述）を除去
% cat out  構造が分かりやすいよう手動で整形
>{"E": [
    {"T": [
        {"F": [ { "i": "i" }]},
        {"T2": []}
    ]},
    {"E2": [
        { "+": "+" },
        { "T": [
            { "F": [ { "i": "i" }]},
            {"T2": []}
        ]},
        { "E2": []}
    ]}
]
```

# 構文木と解析関数のコールグラフは対応する

- $i + i$  の構文木と、 解析関数のコールグラフ
  - 非終端記号に注目すると、 両者は同じ構造
  - これが再帰下降型構文解析が理解しやすい理由



前ページのJSONと同じ構造



# 末尾カンマ問題 (1/2)

俗称：ケツカンマ問題

- JSONで以下の記述（末尾カンマ）は文法エラー
  - { "name": "taro", "age": 22, }
  - [  
  10,  
  20,  
  30 // この一行を消すと、前の行のカンマも要削除]  
]
- 末尾カンマが許されると嬉しい
  - 理由1：（特に行）削除時に、どこでも編集操作が同じ
  - 理由2：データ自動生成のプログラムが単純化
- 末尾カンマを許す言語も多い

```
// C言語の列挙型や初期化子
enum Junken { Gu, Choki, Pa, };
int a[] = {10, 20, 30, };
```



# 末尾カンマ問題 (2/2) 俗称：ケツカンマ問題

- ここでは末尾カンマを単純に出力し、後で消す作戦
- 以下はPerlで「]か}かEOFが（空白だけを挟んで）直後に来るカンマを削除」する一行野郎
  - -pe (sedのように) 1行ごとにスクリプト処理
  - -0 レコードセパレータを'\0'にする（行では無く）

```
% perl -0 -pe 's/,([\t\r\n]*[\]\}]])/\1/g; s/, [\t\r\n]*\z//' < in > out
```

\z は文字列の終端にマッチ



# G1の再帰下降型構文解析器2

parse-G1e.c

```
static void parse_E (void) { // 生成規則 E: T ("+" T)*;
    print_NT_begin ("E");
    switch (lookahead (1)) {
        case '(': case 'i': parse_T (); break;
        default:             parse_error (); break;
    }
    while (1) {
        switch (lookahead (1)) {
            case '+': consume_token ('+'); parse_T (); break;
            default:   goto loop_exit;
        }
    }
loop_exit:
    print_NT_end ();
}
```

- ・EBNFで、\* や + を使った場合は、while文やdo-while文に変換する



# G1の再帰下降型構文解析器3 (1/3)

```
static void show_AST (struct AST *ast, int depth)
{
    int i;
    // 空文字列をdepth幅で印字
    printf ("%*s%s\n", depth, "", ast->ast_type);
    for (i = 0; i < ast->num_child; i++) {
        if (ast->child [i] != NULL) {
            show_AST (ast->child [i], depth +1);
        }
    }
}
```

- struct AST構造体で構文木を作成

parse-G1s.c



# G1の再帰下降型構文解析器3 (2/3)

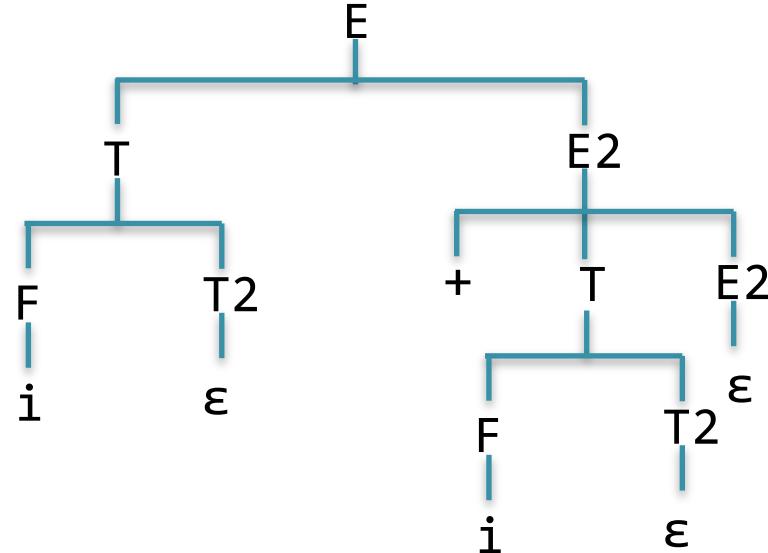
```
static struct AST* parse_E2 (void) { // E2: "+" T E2 | ε
    struct AST *ast, *ast1, *ast2;
    switch (lookahead (1)) {
        case '+':
            consume_token ('+');
            ast1 = parse_T ();
            ast2 = parse_E2 ();
            ast = create_AST ("AST_E2_+", 2, ast1, ast2);
            break;
        case '\0':
        case ')':
            ast = create_AST ("AST_E2_ε", 0); break;
        default: parse_error (); break;
    }
    return ast;
}
```

parse-G1s.c

- ・注：parse\_T()をcreate\_ASTの引数に書いてはダメ。  
C言語では実引数や式の評価順序は決まってない（未規定動作）から

# G1の再帰下降型構文解析器3 (3/3)

```
% ./a.out "i+i"
AST_E
AST_T
AST_F_i
AST_T2_ε
AST_E2_+
AST_T
AST_F_i
AST_T2_ε
AST_E2_ε
```



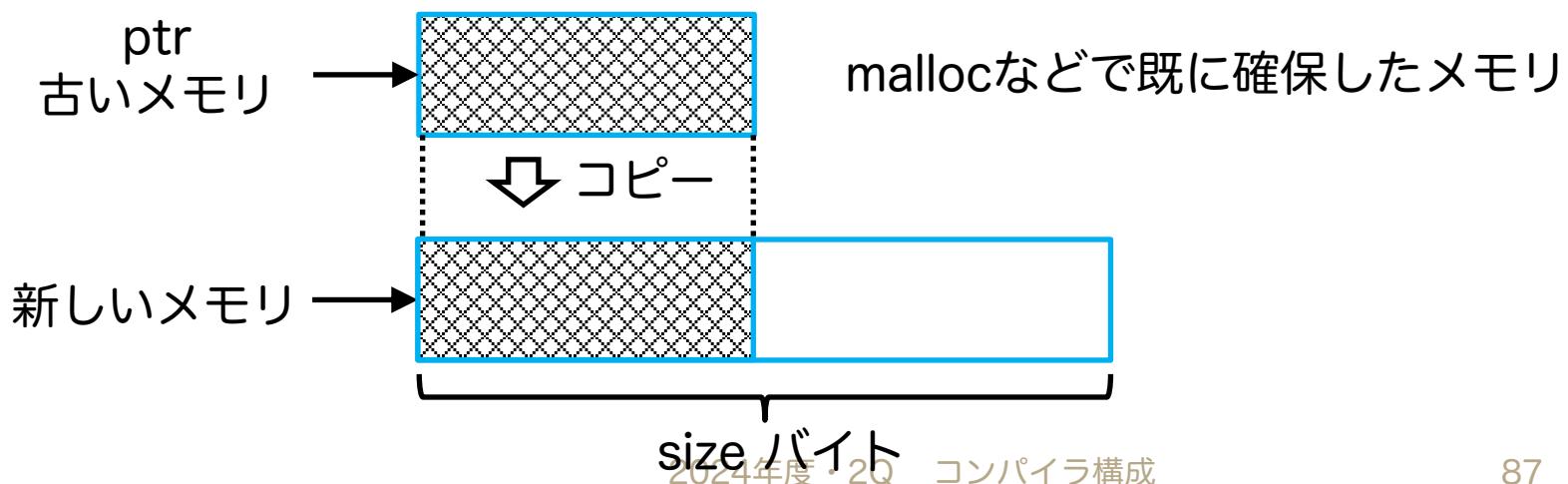


既存のアドレス

新たに確保したい  
サイズ (バイト)

# void \*realloc (void \*ptr, size\_t size)

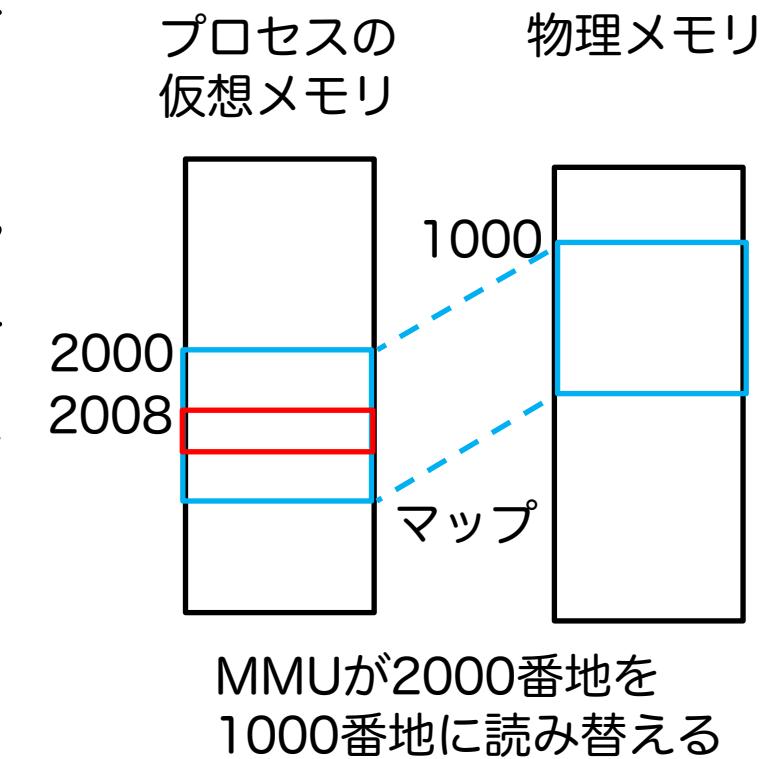
- mallocなどで確保したメモリを大きくする
  - すぐ後ろを延長できれば延長する. ptr をそのまま return
- 延長できなければ,
  1. 別の場所に size バイトのメモリを新たに確保.
  2. 古いメモリを新しいメモリにコピー
  3. 古いメモリを free で解放.
  4. 新しいメモリの先頭アドレスを return で返す.





# 「メモリを確保」って何？

- OSは物理メモリをすべて好きに使える
- その一部をプロセスの仮想メモリにマップする
  - 「この番地にメモリを割り当てたから使っていいよ」
- プロセスはmallocでメモリを要求されると
  - 自分が持ってるメモリ（ヒープ）の空き場所を探して、その先頭アドレスとサイズを管理テーブルにメモして、その先頭番地を返す
    - 空き場所がなかったら、OSにメモリを要求。
    - 管理テーブルも自分のメモリ中に存在
  - 例：malloc(8)すると、青枠の中から赤枠を選び、2008番地を返す





astは不变なので (ast->childは変わる) returnする必要なかった

# G1の再帰下降型構文解析器4 (1/3)

```
static struct AST* add_AST (struct AST *ast, int num_child, ...)  
{  
    va_list ap; int i, start = ast->num_child;  
    ast->num_child += num_child;  
    assert (num_child > 0);  
    ast->child = realloc (ast->child,  
                          sizeof(struct AST *) * ast->num_child);  
    va_start (ap, num_child);  
    for (i = start; i < ast->num_child; i++) {  
        struct AST *child = va_arg (ap, struct AST *);  
        ast->child [i] = child;  
        if (child != NULL) {  
            child->parent = ast; child->nth = i;  
        }  
    }  
    va_end (ap);  
    return ast;  
}
```

parse-G1se.c

- parse-G1s.c の EBNF版. reallocでASTを追加可能にした



# G1の再帰下降型構文解析器4 (2/3)

```
static struct AST* parse_E (void) { // E: T ("+" T)*;
    struct AST *ast, *ast1;
    switch (lookahead (1)) {
        case '(': case 'i':
            ast1 = parse_T ();
            ast = create_AST ("AST_E", 1, ast1); break;
        default: parse_error (); break;
    }
    while (1) {
        switch (lookahead (1)) {
            case '+':
                consume_token ('+');
                ast1 = parse_T ();
                ast = add_AST (ast, 1, ast1); break;
            default: goto loop_exit;
        }
    }
loop_exit:
    return ast;
}
```

parse-G1se.c



# G1の再帰下降型構文解析器4 (3/3)

```
% ./a.out "i*i*i"  
AST_E  
AST_T  
AST_F_i  
AST_F_i  
AST_F_i
```

```
% ./a.out "(i+i+i)*i*i"  
AST_E  
AST_T  
AST_F_()  
AST_E  
AST_T  
AST_F_i  
AST_T  
AST_F_i  
AST_T  
AST_F_i  
AST_F_i  
AST_F_i
```

# 再帰下降構文解析器への変換 (1/2)

- A:  $\alpha_1|\alpha_2|\cdots|\alpha_n$  の変換

```
struct AST *parse_A (void)
{
    struct AST *ast, *ast1, ...;
    switch (lookahead (1)) {
        case director(A, α1):
            α1の変換結果
            ast = create_AST ("AST_A_α1", ast1, ...); break;
            ...
        case director(A, αn):
            αnの変換結果
            ast = create_AST ("AST_A_αn", ast1, ...); break;
        default: parse_error (); break;
    }
    return ast;
}
```

# 再帰下降構文解析器への変換 (2/2)

- $\text{director}(A, \alpha) = \{c_1, \dots, c_n\}$  の時 (後述) ,  
**case director(A, α):** の変換

```
case 'c1': ...: case 'cn':
```

- $\beta_1 \beta_2 \cdots \beta_k$  の変換

```
β1の変換結果 β2の変換結果 ... βkの変換結果
```

- 非終端記号 A の変換

```
asti = parse_A();
```

- 終端記号 'a' の変換

```
consume_token ('a');
```



# 左再帰はうまくいかない (1/2)

- 例：次の文法を変換したプログラムは無限再帰する
  - $E: E + i \mid i$        $E$ は非終端記号,  $i$ は終端記号

```
static struct AST* parse_E (void) { // E: E "+" "i" | "i"
    struct AST *ast, *ast1;
    switch (lookahead (1)) {
        case 'i':
            ast1 = parse_E (); // parse_E が直接再帰
            consume_token ('+');
            consume_token ('i');
            ast = create_AST ("AST_E_1", 1, ast1); break;
        case 'i': // 上のcaseと同じエントリがある
            consume_token ('i');
            ast = create_AST ("AST_E_2", 0); break;
        default: parse_error (); break;
    }
    return ast;
}
```



# 左再帰はうまくいかない (2/2)

- 再帰下降型構文解析で、左再帰があるとまずい
  - もちろん、caseのエントリが重複するのもまずい
- 次ページの手法で左再帰の無い等価な文法に変換可能
  - 左再帰を右再帰に変換



# 左再帰の除去 (1/2)

- 左再帰の無い等価な文法に変換できる。
  - ただし、一般的に読みにくくなる。
- 例： $A: A a \mid b;$  を変換すると以下.  $ba^*$  を生成。
  - $A: b A2;$   
 $A2: a A2 \mid \epsilon;$
- 例：次の文法を変換すると G1 (p.75) になる
  - $E: E + T \mid T;$   
 $T: T * F \mid F;$   
 $F: ( E ) \mid i;$
  - 例えば、 $E: E+ T \mid T;$  は以下になる
    - $E: T E2;$   
 $E2: + T E2 \mid \epsilon$



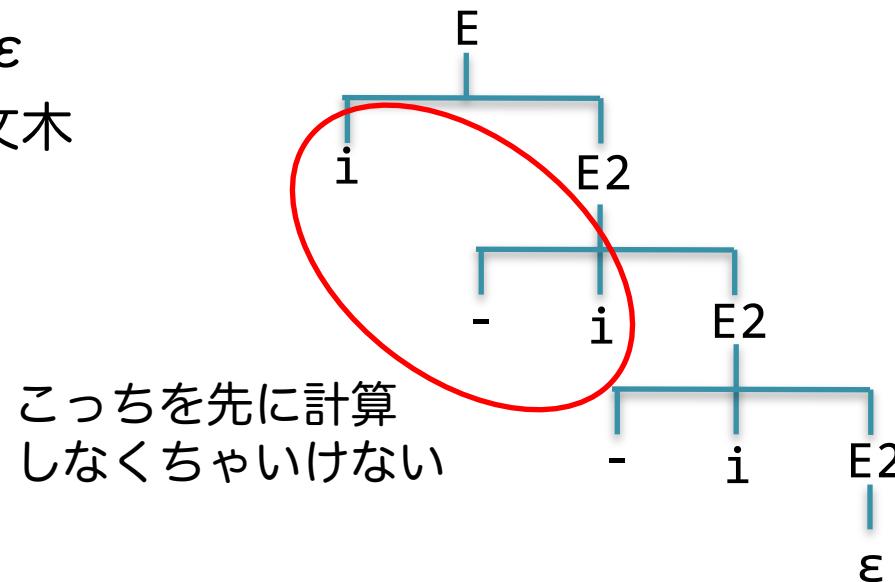
# 左再帰の除去 (2/2)

- 直接左再帰の除去の一般的な方法：
  - $A : A \alpha \mid \beta$ ;  $\beta$  は  $A$  で始まらない記号列
  - $A$  は  $\beta(\alpha)^*$  を導出する
  - ということは以下に変換すれば良い  
 $A : \beta B$ ;  
 $B : \alpha B \mid \varepsilon$ ;  $B$  は新たに導入した非終端記号
- 上の方法は以下の形にも拡張可能
  - $A : A \alpha_1 \mid A \alpha_2 \mid \cdots \mid A \alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$ ;  
 $\beta_i$  は  $A$  で始まらない記号列



# 右再帰の注意

- 構文木をそのまま単純に解釈すると右結合になる
- 例えば、引き算（-）は左結合なので要注意
- 例：
  - $E: i \ E2;$   
 $E2: - \ i \ E2 \mid \epsilon$
  - $i - i - i$  の構文木





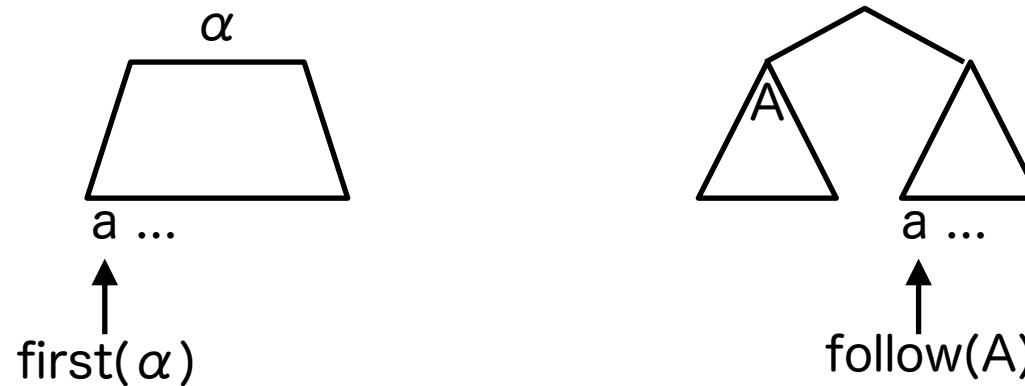
# first, follow, director集合

- 準備：
  - $\alpha, \beta, \gamma$  は0個以上の記号列.  $N$  は非終端記号の集合,  $A \in N$ ,  $S$  は開始記号,  $T$  は終端記号の集合,  $\$$  は入力の右端を示す特別な記号
- first ( $\alpha$ )
  - 気持ち： $\alpha$  の最初に出現する終端記号の集合
  - 定義： $\text{first}(\alpha) = \{ a | a \in T, \alpha \Rightarrow^* a\beta \} + \{ \epsilon | \alpha \Rightarrow^* \epsilon \}$
- follow (A)
  - 気持ち：Aの直後に続いて出現する終端記号の集合
  - 定義： $\text{follow}(A) = \{ a | a \in T + \{\$\}, S\$ \Rightarrow^* \beta A a \gamma \}$
- director (A,  $\alpha$ )
  - 気持ち：Aを $\alpha$ に展開すべきかを指示する終端記号の集合
  - 定義： $\text{director}(A, \alpha) = \begin{cases} \text{first}(\alpha) - \{ \epsilon \} + \text{follow}(A) & (\alpha \Rightarrow^* \epsilon \text{ の場合}) \\ \text{first}(\alpha) & (\text{それ以外}) \end{cases}$

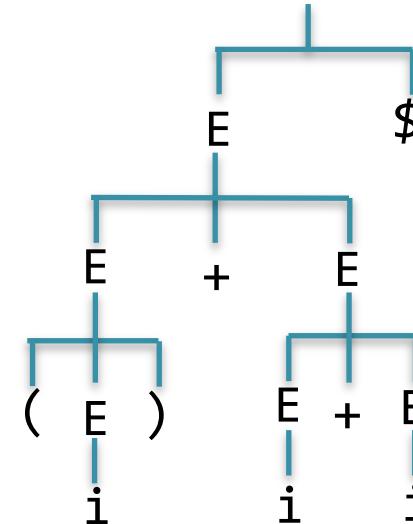


# first/follow/director 図解 (1/2)

- 図解： $a$ を全部集めると,  $\text{first}(\alpha)$ と  $\text{follow}(A)$ になる

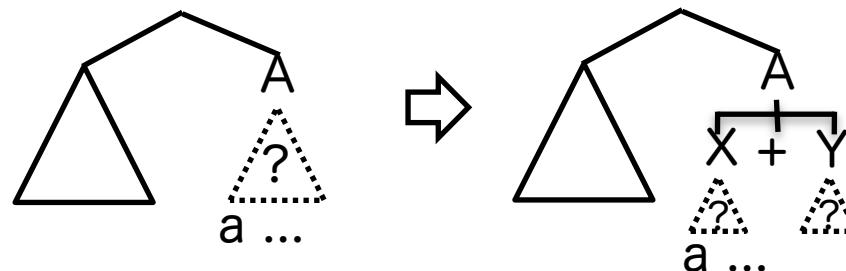


- 例： $E: E + E \mid ( E ) \mid i ;$ 
  - $\text{first}(E) = \{ (, i \}$
  - $\text{first}((E)) = \{ ( \}$
  - $\text{first}(E+E) = \{ (, i \}$
  - $\text{follow}(E) = \{ +, ), \$ \}$

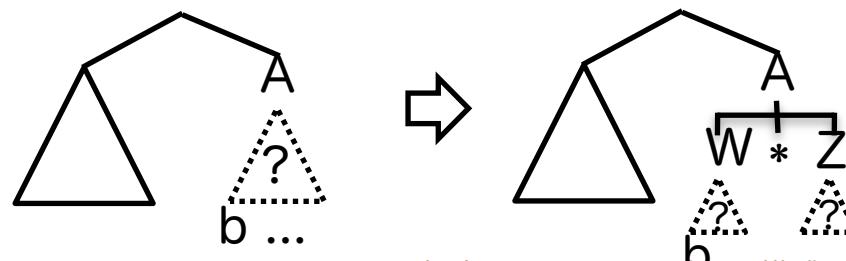


# first/follow/director 図解 (2/2)

- 図解：director集合は「非終端記号Aを何に展開するか」を決める
  - 生成規則  $A: X + Y \mid W^* Z;$  があるとする
  - $\text{director}(A, X+Y)=\{a\}$ ,  $\text{director}(A, W^*Z)=\{b\}$ とする
  - 次の文字がaなら,  $a \in \text{director}(A, X+Y)$ なので, AをX+Yに展開



- 次の文字がbなら,  $b \in \text{director}(A, W^*Z)$ なので, AをW^\*Zに展開





# first集合の計算方法 (1/2)

- 終端記号の場合
  - $\text{foreach } X \in T \text{ do first}(X) = \{X\}$
- 非終端記号の場合
  - $\text{foreach } X \in N \text{ do first}(X) = \emptyset$   
 $\text{foreach 生成規則 } X: \varepsilon \text{ do first}(X) = \{ \varepsilon \}$   
 $\text{repeat}$   
 $\text{foreach 生成規則 } X: X_1 X_2 \dots X_n \text{ do}$   
 $\text{first}(X) += \text{first}(X_1) - \{ \varepsilon \}$   
 $\text{for } k = 1 \text{ to } n-1 \text{ do}$   
 $\quad \text{if } \wedge_{1 \leq i \leq k} \varepsilon \in \text{first}(X_i) \text{ then first}(X) += \text{first}(X_{k+1}) - \{ \varepsilon \}$   
 $\quad \text{if } \wedge_{1 \leq i \leq n} \varepsilon \in \text{first}(X_i) \text{ then first}(X) += \{ \varepsilon \}$   
 $\text{until どのfirst集合にも変更が無かった}$



# first集合の計算方法 (2/2)

- 記号列の場合
  - $\text{first}(\varepsilon) = \{\varepsilon\}$
  - $\text{first}(X\beta) = \begin{cases} (\text{first}(X)-\{\varepsilon\}) + \text{first}(\beta) & (X \Rightarrow^* \varepsilon \text{ の場合}) \\ \text{first}(X) & (\text{それ以外}) \end{cases}$
- $\varepsilon \in \text{first}(\alpha)$  は  $\alpha \Rightarrow^* \varepsilon$  を覚えておくためのマーク
- コツ：次の生成規則から始めると、計算が速く収束
  - 終端記号が右辺の左端にある生成規則
  - 開始記号から遠い生成規則
- コツ： $\text{first}(X)$  の計算は **左辺が X の生成規則に注目する**



# first集合の例

- 以下の文法を考える

$E: T \ E2;$

$E2: + \ T \ E2 \mid \varepsilon;$

$T: F \ T2;$

$T2: * \ F \ T2 \mid \varepsilon;$

$F: ( \ E ) \mid i;$

- 上の文法に対するfirst集合は以下の通り

$$\text{first}(F) = \{(, i\} \quad \text{first}(+) = \{+\}$$

$$\text{first}(T2) = \{*, \varepsilon\} \quad \text{first}(*) = \{*\}$$

$$\text{first}(T) = \{(, i\} \quad \text{first}(()) = \{()\}$$

$$\text{first}(E2) = \{+, \varepsilon\} \quad \text{first}(()) = \{()\}$$

$$\text{first}(E) = \{(, i\} \quad \text{first}(i) = \{i\}$$

# follow集合の計算方法

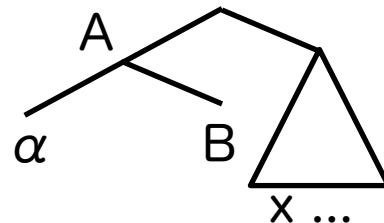
- follow(A)の計算方法
  - foreach  $X \in N$  do  $\text{follow}(X) = \emptyset$   
 $\text{follow}(S) = \{\$\}$   
repeat
    - foreach 生成規則  $A: \alpha B \beta$  do  
 $\text{follow}(B) += \text{first}(\beta) - \{ \epsilon \}$
    - foreach 生成規則  $A: \alpha B$  or  $(A: \alpha B \beta, \epsilon \in \text{first}(\beta))$  do  
 $\text{follow}(B) += \text{follow}(A)$until どのfollow集合にも変更がなかった
- コツ：開始記号から始めると、計算は速く収束
- コツ：follow(X)の計算は右辺にXがある生成規則に注目する



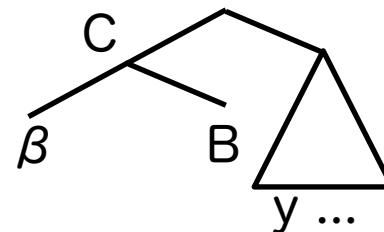
A:  $\alpha B$

$\text{follow}(B) += \text{follow}(A)$

- Aの後に来るものは、必ずBの後ろにも来る



- Bの後に来るものは、必ずしもAの後ろには来ない
  - C:  $\beta B$  という別の規則があるとすると…





# follow集合の例

- 以下の文法を考える (first集合も併記)

$E: T \ E2 ;$

$E2: + \ T \ E2 \mid \varepsilon ;$

$T: F \ T2 ;$

$T2: * \ F \ T2 \mid \varepsilon ;$

$F: ( \ E ) \mid i ;$

$\text{first}(E) = \{ (, i\}$

$\text{first}(E2) = \{ +, \varepsilon \}$

$\text{first}(T) = \{ (, i\}$

$\text{first}(T2) = \{ *, \varepsilon \}$

$\text{first}(F) = \{ (, i\}$

- 上の文法に対するfollow集合は以下の通り

$\text{follow}(E) = \{ \$, ) \}$

$\text{follow}(E2) = \text{follow}(E) = \{ \$, ) \}$

$\text{follow}(T) = (\text{first}(E2)-\{ \varepsilon \})+\text{follow}(E2)+\text{follow}(E)=\{ +, \$, ) \}$

$\text{follow}(T2) = \text{follow}(T)+\text{follow}(T2)=\{ +, \$, ) \}$

$\text{follow}(F) = (\text{first}(T2)-\{ \varepsilon \})+\text{follow}(T2)+\text{follow}(T)=\{ *, +, \$, ) \}$



# director集合の例

- 以下の文法を考える (first/follow集合も併記)

E: T E2 ; first(E) = { (, i} follow(E) = { \$, ) }  
E2: + T E2 | ε ; first(E2) = { +, ε } follow(E2) = { \$, ) }  
T: F T2; first(T) = { (, i} follow(T) = { +, \$, ) }  
T2: \* F T2 | ε ; first(T2) = { \*, ε } follow(T2) = { +, \$, ) }  
F: ( E ) | i ; first(F) = { (, i} follow(F) = { \*, +, \$, ) }

- 上の文法に対するdirector集合は以下の通り

director(E, T E2) = first(T E2) = first(T) = { (, i }  
director(E2, + T E2) = first(+ T E2) = first(+) = { + }  
director(E2, ε) = (first(ε) - { ε }) + follow(E2) = { \$, ) }  
director(T, F T2) = first(F T2) = first(F) = { (, i }  
director(T2, \* F T2) = first(\* F T2) = first(\*) = { \* }  
director(T2, ε) = (first(ε) - { ε }) + follow(T2) = { +, \$, ) }  
director(F, ( E )) = first(( E )) = first(()) = { () }  
director(F, i) = first(i) = { i }

# LL(1)文法

- LL(1)文法の定義
  - ある文脈自由文法の、左辺は同じだが右辺が異なる任意の生成規則、 $A:\alpha$ と $A:\beta$ に対して $\text{director}(A, \alpha) \cap \text{director}(A, \beta) = \emptyset$ が成り立つ時、その文法はLL(1)文法である
- LL(1)文法の気持ち
  - 同じ $A$ の $\text{director}$ 集合に重複が無い。つまり先読み記号 $a$ に対して、 $a \in \text{director}(A, \alpha)$ となる生成規則 $A:\alpha$ は高々1つのみ
  - ということは、先読み記号を見れば、 $A$ をどの生成規則で展開すべきかが一意に決まる
- LL(1)文法の例：前ページの文法はLL(1)文法

# 練習問題：LL(1)かどうかを判定せよ

1.  $S: a B d;$

$B: b C;$

$C: c \mid \epsilon;$

2.  $S: a B c;$

$B: b C;$

$C: c \mid \epsilon;$

3.  $S: A B a;$

$A: a \mid \epsilon;$

$B: b \mid \epsilon;$

4.  $S: A c B a;$

$A: a \mid B \mid \epsilon;$

$B: b \mid \epsilon;$

5. 単純な数式

$S: E;$

$E: E + E$

$\mid E * E$

$\mid ( E )$

$\mid i ;$

6. 5を右再帰に変形

$S: E;$

$E: T + E$

$\mid T * E$

$\mid T ;$

$T: ( E )$

$\mid i ;$

7. 6の共通要素(T)を  
くくりだし

$S: E;$

$E: T E2;$

$E2: + T E2$

$\mid * T E2$

$\mid \epsilon ;$

$T: ( E ) \mid i ;$

8. 7の演算子(+, \*)  
に優先順位をつけると, p.104の文  
法になる



# 練習問題の答 (1/3)

## 1. LL(1)文法である

- $\text{director}(C, c) = \text{first}(c) = \{c\}$
- $\text{director}(C, \epsilon) = \text{follow}(C) = \{d\}$

	first	follow
S	a	\$
B	b	d
C	c, $\epsilon$	d

## 2. LL(1)文法ではない

- $\text{director}(C, c) = \text{first}(c) = \{c\}$
- $\text{director}(C, \epsilon) = \text{follow}(C) = \{c\}$

	first	follow
S	a	\$
B	b	c
C	c, $\epsilon$	c

## 3. LL(1)文法ではない

- $\text{director}(A, a) = \text{first}(a) = \{a\}$
- $\text{director}(A, \epsilon) = \text{follow}(A) = \{b, a\}$
- $\text{director}(B, b) = \text{first}(b) = \{b\}$
- $\text{director}(B, \epsilon) = \text{follow}(B) = \{a\}$

	first	follow
S	a, b	\$
A	a, $\epsilon$	b, a
B	b, $\epsilon$	a



# 練習問題の答 (2/3)

## 4. LL(1)文法ではない

- $\text{director}(A, a)=\{a\}$
- $\text{director}(A, B)=(\text{first}(B)-\{\varepsilon\})+\text{follow}(A)=\{b, c\}$
- $\text{director}(A, \varepsilon)=\text{follow}(A)=\{c\}$
- $\text{director}(B, b)=\{b\}$
- $\text{director}(B, \varepsilon)=\text{follow}(B)=\{a, c\}$

	first	follow
S	a, b, c	\$
A	a, b, $\varepsilon$	c
B	b, $\varepsilon$	a, c

## 5. LL(1)文法ではない

- $\text{director}(E, E+E)=\text{first}(E+E)=\{(, i\}$
- $\text{director}(E, E^*E)=\text{first}(E^*E)=\{(, i\}$
- $\text{director}(E, (E))=\text{first}((E))=\{()\}$
- $\text{director}(E, i)=\{i\}$

	first	follow
S	(, i	\$
E	(, i	\$, +, *, )



# 練習問題の答 (3/3)

## 6. LL(1)文法ではない

- $\text{director}(E, T+E) = \text{first}(T+E) = \{(, i\}$
- $\text{director}(E, T^*E) = \text{first}(T^*E) = \{(, i\}$
- $\text{director}(E, T) = \text{first}(T) = \{(, i\}$
- $\text{director}(E, (E)) = \text{first}((E)) = \{\{\}$
- $\text{director}(E, i) = \{i\}$

## 7. LL(1)文法である

- $\text{director}(E2, +T E2) = \{+\}$
- $\text{director}(E2, *T E2) = \{*\}$
- $\text{director}(E2, \varepsilon) = \text{follow}(E2) = \{\$, )\}$
- $\text{director}(T, (E)) = \{\{\}$
- $\text{director}(T, i) = \{i\}$

	first	follow
S	(, i	\$
E	(, i	\$, )
T	(, i	\$, +, *, )

	first	follow
S	(, i	\$
E	(, i	\$, )
E2	+, *, $\varepsilon$	\$, )
T	(, i	\$, +, *, )



# LR構文解析



# LR構文解析 (LR parsing)

- 上向きに構文解析する。最右導出を逆順に見つけることに相当。
- 構文解析表とスタックを使って、シフト(shift)と還元(reduce)という動作を繰り返して構文解析する
- LR(1)文法 (LR(1) grammar)
  - 先読み記号を1つ使い、1パスで決定的に上向き構文解析できる
  - LR(1)文法はLL(1)文法よりも言語クラスが広い
  - LR(1)を少し制限したLALR(1)やSLR(1)もある。
    - LALR(1)は GNU Bison や Yacc が受け付けるクラス
  - 構文解析表の作成は機械的に出来るが、人が作るには少し面倒。また表のサイズも大きい → 生成系 (GNU Bison) 向き



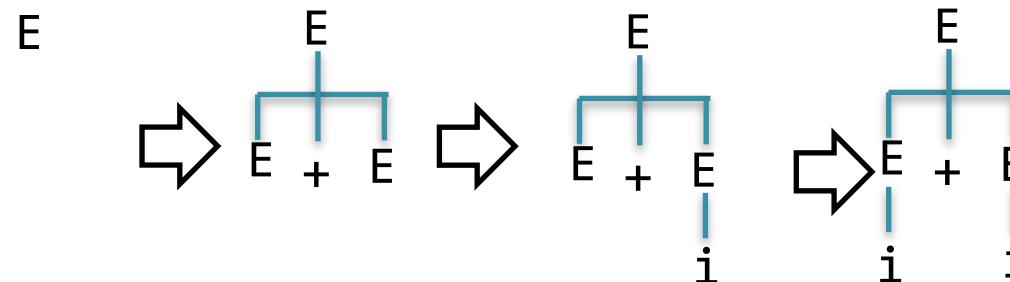
# 最右導出と上向き構文解析

- $E: E + E \mid i$ ; のみの文法を考える
- $i + i$  の最右導出 (rightmost derivation)
  - $E \Rightarrow E + E \Rightarrow E + i \Rightarrow i + i$
- $i + i$  の上向き構文解析 (最左還元 leftmost reduction)
  - $i + i \rightarrow E + i \rightarrow E + E \rightarrow E$
  - 上の最右導出の逆順
- 注目点 (●) を明示した最左還元
  - $\bullet i + i \rightarrow i \bullet + i \rightarrow E \bullet + i \rightarrow E + \bullet i \rightarrow E + i \bullet$   
 $\rightarrow E + E \bullet$
- シフトと還元
  - シフト=注目点を一つ右にずらすこと
  - 還元=注目点の左側の記号列を生成規則の左辺に置換すること

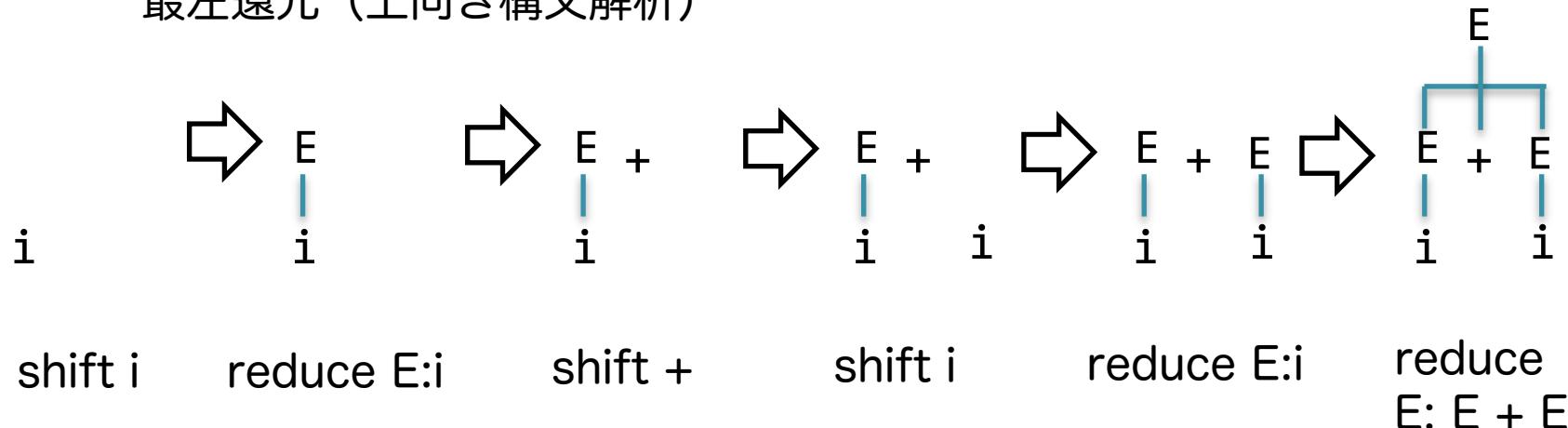


# 最右導出と最左還元（図）

最右導出



最左還元（上向き構文解析）



# LR構文解析器の概要

- 構文解析表とドライバ
  - ドライバは構文解析表と入力文字列を入力として構文解析を行う
  - SLR(1), LALR(1), LR(1)で, ドライバは共通, 構文解析表は異なる
- スタック: ドライバの中間状態を格納.
  - 注目点●の左側に相当
- 主なドライバの動作
  - シフト=先読み記号 (残り入力の先頭文字) を読んで, スタックにプッシュする
  - 還元=生成規則  $X: \alpha$  の還元を行う. つまり, スタックトップの記号列  $\alpha$  をポップして,  $X$ をプッシュする



# 例題の文法G2

1.  $R: E;$
2.  $E: E + T;$
3.  $E: T;$
4.  $T: T * F;$
5.  $T: F;$
6.  $F: ( E ) ;$
7.  $F: i$

# 構文解析表の例 (G2のSLR(1)構文解析表)

状態	action							goto		
	i	+	*	(	)	\$	E	T	F	
0	s5			s4			1	2	3	
1		s6				acc				
2		r2	s7		r2	r2				
3		r4	r4		r4	r4				
4	s5			s4			8	2	3	
5		r6	r6		r6	r6				
6	s5			s4				9	3	
7	s5			s4					10	
8		s6			s11					
9		r1	s7		r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

- 読み方： s5はシフトして状態5をプッシュ, r2は生成規則2で還元, accは受理(accept0), 空白は構文エラー
- 状態0は初期状態



# LR構文解析アルゴリズム（ドライバ）

- 入力：入力記号列 $\omega$ , 文法GのLR構文解析表
- 出力： $\omega \in L(G)$ なら $\omega$ の最左還元,  
 $\omega \notin L(G)$ なら構文エラー
- アルゴリズム
  - スタック = (0), 残り入力 = ( $\omega \$$ ) // 0は初期状態  
do forever  
    state = top(スタック); 先読み記号=残り入力[0];  
    switch (action (状態, 先読み記号))  
        case shift n: 先読み記号を残り入力から取り出しだす  
            スタックにpush. 状態nをスタックにpush. break;  
        case reduce n: 生成規則nを $A : \beta$ とする.  $\beta$ と付随する  
            状態をpop. Aとgoto(top(スタック), A)をスタックに  
            push. break;  
        case accept: 構文解析は成功して停止.  
        case error: 構文解析は失敗して停止（またはエラー回復）



# LR構文解析の過程 (文法G2, 入力(i+i)\*i)

	スタック	残り入力	action	説明
1	0	(i+i)*i\$	s4	
2	0(4	i+i)*i\$	s5	
3	0(4i5	+i)*i\$	r6	(6) F: i; goto (4, F)=3
4	0(4F3	+i)*i\$	r4	(4) T: F; goto (4, T)=2
5	0(4T2	+i)*i\$	r2	(2) E: T; goto (4, E)=8
6	0(4E8	+i)*i\$	s6	
7	0(4E8+6	i)*i\$	s5	
8	0(4E8+6i5	)*i\$	r6	(6) F: i; goto (6, F)=3
9	0(4E8+6F3	)*i\$	r4	(4) T: F; goto (6, T)=9
10	0(4E8+6T9	)*i\$	r1	(1) E: E + T; goto (4, E)=8
11	0(4E8	)*i\$	s11	
12	0(4E8)11	*i\$	r5	(5) F: ( E ); goto (0, F)=3
13	0F3	*i\$	r4	(4) T: F; goto (0, T)=2
14	0T2	*i\$	s7	
15	0T2*7	i\$	s5	
16	0T2*7i5	\$	r6	(6) F: i; goto (7, F)=10
17	0T2*7F10	\$	r3	(3) T: T * F; goto (0, T)=2
18	0T2	\$	r2	(2) E: T; goto (0, E)=1
19	0E1	\$	acc	

# LR(0)項

- LR(0)項
  - 生成規則の右辺に注目点●を1つ入れたもの
  - 0は先読み記号の数.
- LR(0)項の例
  - 生成規則  $E: E + E;$  の LR(0)項は次の4つ  
[  $E: \bullet E + E;$  ], [  $E: E \bullet + E;$  ], [  $E: E + \bullet E;$  ],  
[  $E: E + E \bullet;$  ],
  - 生成規則  $E: \varepsilon;$  のLR(0)項は次の1つ  
[  $E: \bullet;$  ]



# LR(0)項の集合の閉包演算

- LR(0)項の集合Iの閉包closure(I)の計算アルゴリズム
  - closure(I)=I  
repeat
    - foreach 生成規則  $B: \gamma$  do
    - if  $[A: \alpha \bullet B \beta] \in \text{closure}(I)$  then
    - $\text{closure}(I) += \{[B: \bullet \gamma]\}$
  - until  $\text{closure}(I)$ に変更が無かった
- 閉包closure(I)の気持ち
  - 閉包はLR構文解析器の状態に対応
  - 注目点が $[E: E + \bullet T;]$ にある=注目点が $[T: \bullet T * F;]$ にある
    - Tの左に●がある→Tを左辺に持つ生成規則の右辺の最左に●を置く
  - 閉包は注目点に関して同じ状態をすべて集めたもの

# LR(0)項の集合の閉包演算（例）

- 文法G2のclosure ( $\{[E:E+\bullet T;]\}$ )= $\{[E:E+\bullet T;], [T:\bullet T^*F;], [T:\bullet F;], [F: \bullet (E)];, [F:\bullet i;]\}$
- 文法G2のclosure ( $\{[E:\bullet E+T;]\}$ )= $\{[E:\bullet E+T;], [E:\bullet T;], [T:\bullet T^*F;], [T:\bullet F;], [F:\bullet (E)];, [F:\bullet i]\}$
- 文法G2のclosure ( $\{[F:(\bullet E);]\}$ )= $\{[F:(\bullet E)];, [E:\bullet E+T;], [E:\bullet T;], [T:\bullet T^*F;], [T:\bullet F;], [F:\bullet (E)];, [F:\bullet i]\}$



# LR(0)項の集合のgoto演算

- $\text{goto}(I, X)$ 
  - $\text{goto}(I, X) = \text{closure}(\{[A: \alpha X \bullet \beta ;] \mid [A: \alpha \bullet X \beta ;] \in I\})$
  - ただし、 $I$ はLR(0)項の集合、 $X$ は記号
- G2での例
  - $\text{goto}(\{[E:T \bullet ;], [T:T \bullet *F;]\}, *)$   
= closure ( $\{[T:T^* \bullet F;]\}$ )  
=  $\{[T:T^* \bullet F;], [F:\bullet(E);], [F:\bullet i;]\}$
- $\text{goto}(I, X)$ の気持ち
  - $I$ の注目点を一つ右の記号にずらしてから閉包を計算
  - 状態 $I$ で記号 $X$ をシフトした時の状態 $I'$ を計算



# goto演算の例題

- 文法G2で以下を計算せよ
  - $I_0 = \text{closure}(\{\text{R: } \bullet E; \})$
  - $I_1 = \text{goto}(I_0, E)$
  - $I_2 = \text{goto}(I_0, T)$
  - $I_3 = \text{goto}(I_0, F)$
  - $I_4 = \text{goto}(I_0, ()$
  - $I_5 = \text{goto}(I_0, i)$
  - $I_6 = \text{goto}(I_1, +)$
  - $I_7 = \text{goto}(I_2, *)$
  - $I_8 = \text{goto}(I_4, E)$
- 答は p.123 参照

# 正準LR(0)集成 (canonical LR(0) collection)

- 入力：拡大文法  $G'$ 
  - 拡大文法 = 文法  $G$  に規則  $S': S;$  を加えた文法。ただし、 $S$  は文法  $G$  の開始記号、 $S'$  は  $G$  に無い記号
- 出力：正準LR(0)集成 (=C)
- 計算方法
  - $C = \{\text{closure}(\{[S']: \bullet S;\})\};$   
repeat
    - foreach LR(0)項集合  $I \in C$  と文法記号  $X$  do
    - if  $\text{goto}(I, X) \neq \emptyset$
    - then  $C += \{\text{goto}(I, X)\};$
  - until  $C$  に新たなLR(0)項集合が加えられなかった



# 正準LR(0)集成（例）（1/2）

- 文法G2の正準LR(0)集成は

$$C = \{ I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9, I_{10}, I_{11} \}$$

- $I_0 \sim I_{11}$  の計算過程は以下。[]は省略。

- $I_0 = \text{closure}(\{R:\bullet E\}) = \{R:\bullet E, E:\bullet E+T, E:\bullet T, T:\bullet T^*F, T:\bullet F, F:\bullet(E), F:\bullet i\}$
- $I_1 = \text{goto}(I_0, E) = \text{closure}(\{R:E\bullet, E:E\bullet+T\}) = \{R:E\bullet, E:E\bullet+T\}$
- $I_2 = \text{goto}(I_0, T) = \text{closure}(\{E:T\bullet, T:T\bullet^*F\}) = \{E:T\bullet, T:T\bullet^*F\}$
- $I_3 = \text{goto}(I_0, F) = \text{closure}(\{T:F\bullet\}) = \{T:F\bullet\}$
- $I_4 = \text{goto}(I_0, ()) = \text{closure}(\{F:(\bullet E)\}) = \{F:(\bullet E), E:\bullet E+T, E:\bullet T, T:\bullet T^*F, T:\bullet F, F:\bullet(E), F:\bullet i\}$
- $I_5 = \text{goto}(I_0, i) = \text{closure}(\{F:i\bullet\}) = \{F:i\bullet\}$
- $I_6 = \text{goto}(I_1, +) = \text{closure}(\{E:E+\bullet T\}) = \{E:E+\bullet T, T:\bullet T^*F, T:\bullet F, F:\bullet(E), F:\bullet i\}$
- $I_7 = \text{goto}(I_2, *) = \text{closure}(\{T:T^*\bullet F\}) = \{T:T^*\bullet F, F:\bullet(E), F:\bullet i\}$
- $I_8 = \text{goto}(I_4, E) = \text{closure}(\{F:(E\bullet), E:E\bullet+T\}) = \{F:(E\bullet), E:E\bullet+T\}$
- $\text{goto}(I_4, T) = \text{closure}(\{E:T\bullet, T:T\bullet^*F\}) = I_2$
- $\text{goto}(I_4, F) = \text{closure}(\{T:F\bullet\}) = I_3$



# 正準LR(0)集成（例）（1/2）

- $\text{goto}(I4, ()) = \text{closure}(\{F:(\bullet E)\}) = I4$
- $\text{goto}(I4, i) = \text{closure}(\{F:i\bullet\}) = I5$
- $I9 = \text{goto}(I6, T) = \text{closure}(\{E:E+T\bullet, T:T\bullet^*F\}) = \{E:E+T\bullet, T:T\bullet^*F\}$
- $\text{goto}(I6, F) = \text{closure}(\{T:F\bullet\}) = I3$
- $\text{goto}(I6, ()) = \text{closure}(\{F:(\bullet E)\}) = I4$
- $\text{goto}(I6, i) = \text{closure}(\{F:i\bullet\}) = I5$
- $I10 = \text{goto}(I7, F) = \text{closure}(\{T:T^*F\bullet\}) = \{T:T^*F\bullet\}$
- $\text{goto}(I7, ()) = \text{closure}(\{F:(\bullet E)\}) = I4$
- $\text{goto}(I7, i) = \text{closure}(\{F:i\bullet\}) = I5$
- $I11 = \text{goto}(I8, ()) = \text{closure}(\{F:(E)\bullet\}) = \{(E)\bullet\}$
- $\text{goto}(I8, +) = \text{closure}(\{E:E+\bullet T\}) = I6$
- $\text{goto}(I9, *) = \text{closure}(\{T:T^*\bullet F\}) = I7$



# SLR(1)構文解析表の作成

- 入力：拡大文法 $G'$ と各非終端記号 $A$ の $\text{follow}(A)$
- 出力： $G'$ のSLR(1)構文解析表
- 計算方法
  1.  $G'$ の正準LR(0)集成 $C=\{I_0, I_1, \dots, I_n\}$ を作る。 $I_i$ を状態*i*とする。  
[ $S'$ : ● $S$ ]を含む項集合を初期状態とする。
  2. 次の通り表を埋める ( $a$ は終端記号,  $A$ は非終端記号)

( $i, a$ )のエントリ	条件
shift $j$	$[A: \alpha \bullet a \beta] \in I_i \wedge I_j = \text{goto}(I_i, a)$
reduce $A: \alpha$	$[A: \alpha \bullet] (A \neq S') \in I_i \wedge a \in \text{follow}(A)$
goto $j$	$I_j = \text{goto}(I_i, A)$
accept	$[S': S \bullet] \in I_i \wedge a = \$$
error	それ以外

3. 表のエントリに衝突（重複）があれば失敗。なければ成功



# SLR(1)構文解析表の作成（例）

- 文法G2のSLR(1)構文解析表の作成（一部）
  - $F: \bullet(E) \in I_0 \wedge I_4 = \text{goto } (I_0, ())$ なので、 $(0, ())$ のエントリは shift 4 (s4)
  - $E:T \bullet \in I_2 \wedge \text{follow}(E)=\{+, (), \$\}$ なので、 $(2, +), (2, (), (2, \$))$ のエントリは reduce E:T (r2)
  - $I_1 = \text{goto } (I_0, E)$ なので、 $(0, E)$ のエントリは goto 1(1)
  - $R:E \bullet \in I_1$ なので、 $(1, \$)$ のエントリは accept (acc)
- 全部を作成すると p.120の表になる
- p.120の表には重複が無い→文法G2はSLR(1)文法

# 衝突 (conflict)

- シフト / 還元衝突 (shift/reduce conflict)
  - LR構文解析表の1つのエントリに、シフトと還元がある衝突
- 還元 / 還元衝突 (reduce/reduce conflict)
  - LR構文解析表の1つのエントリに、複数の還元がある衝突



# 練習問題：SLR(1)かどうかを判定せよ

## 1. G3 (右再帰)

- R: E;  
E: T + E | T;  
T: F \* T | F;  
F: i;

## 2. G4 (LL(1))

- R: S;  
S: A a A b  
| B b B a;  
A:  $\epsilon$ ;  
B:  $\epsilon$ ;

## 3. G5 (ぶらぶらif)

- R: S;  
S: if E then S else S  
| if E then S  
|  $\epsilon$  ;  
E: true;



# 解答(G3)

SLR(1)である

$I_0 = \text{closure}(R: \bullet E)$

$= \{R: \bullet E, E: \bullet T+E, E: \bullet T, T: \bullet F*T, T: \bullet F, F: \bullet i\}$

$I_1 = \text{goto}(I_0, E) = \text{closure}(\{R: E \bullet\}) = \{R: E \bullet\}$

$I_2 = \text{goto}(I_0, T) = \text{closure}(\{E: T \bullet + E, E: T \bullet\})$   
 $= \{E: T \bullet + E, E: T \bullet\}$

$I_3 = \text{goto}(I_0, F) = \text{closure}(\{T: F \bullet * T, T: F \bullet\})$   
 $= \{T: F \bullet * T, T: F \bullet\}$

$I_4 = \text{goto}(I_0, i) = \text{closure}(\{F: i \bullet\}) = \{F: i \bullet\}$

$I_5 = \text{goto}(I_2, +) = \text{closure}(\{E: T + \bullet E\})$   
 $= \{E: T + \bullet E, E: \bullet T + E, E: \bullet T, T: \bullet F * T, T: \bullet F,$   
 $F: \bullet i\}$

$I_6 = \text{goto}(I_3, *) = \text{closure}(\{T: F * \bullet T\})$   
 $= \{T: F * \bullet T, T: \bullet F * T, T: \bullet F, F: \bullet i\}$

$I_7 = \text{goto}(I_5, E) = \text{closure}(\{E: T + E \bullet\}) = \{E: T + E \bullet\}$

$\text{goto}(I_5, T) = \text{closure}(\{E: T \bullet + E, E: T \bullet\}) = I_2$

$\text{goto}(I_5, F) = \text{closure}(\{T: F \bullet * T, T: F \bullet\}) = I_3$

$\text{goto}(I_5, i) = \text{closure}(\{F: i \bullet\}) = I_4$

$I_8 = \text{goto}(I_6, T) = \text{closure}(\{T: F * T \bullet\}) = \{T: F * T \bullet\}$

$\text{goto}(I_6, F) = \text{closure}(\{T: F \bullet * T, T: F \bullet\}) = I_3$

$\text{goto}(I_6, i) = \text{closure}(\{F: i \bullet\}) = I_4$

	+ * i \$	E T F
-+-----+-----+		
0	s4	1 2 3
1	acc	R: E
2  s5	r2	E: T
3  r4 s6	r4	T: F
4  r5 r5	r5	F: i
5	s4	7 2 3
6	s4	8 3
7	r1	E: T+E
8  r3	r3	T: F*T

SLR構文解析表に重複が無いので文法G3はSLR(1)。



# 解答(G4)

SLR(1)ではない

$I_0 = \text{closure}(\{R: \bullet S\})$

$= \{R: \bullet S, S: \bullet AaAb, S: \bullet BbBa, A: \bullet, B: \bullet\}$

$I_1 = \text{goto}(I_0, S) = \text{closure}(\{R: S \bullet\}) = \{R: S \bullet\}$

$I_2 = \text{goto}(I_0, A) = \text{closure}(\{S: A \bullet aAb\}) = \{S: A \bullet aAb\}$

$I_3 = \text{goto}(I_0, B) = \text{closure}(\{S: B \bullet bBa\}) = \{S: B \bullet bBa\}$

$I_4 = \text{goto}(I_2, a) = \text{closure}(\{S: Aa \bullet Ab\})$

$= \{S: Aa \bullet Ab, A: \bullet\}$

$I_5 = \text{goto}(I_3, b) = \text{closure}(\{S: Bb \bullet Ba\}) = \{S: Bb \bullet Ba, B: \bullet\}$

$I_6 = \text{goto}(I_4, A) = \text{closure}(\{S: AaA \bullet b\}) = \{S: AaA \bullet b\}$

$I_7 = \text{goto}(I_5, B) = \text{closure}(\{S: BbB \bullet a\}) = \{S: BbB \bullet a\}$

$I_8 = \text{goto}(I_6, b) = \text{closure}(\{S: AaAb \bullet\}) = \{S: AaAb \bullet\}$

$I_9 = \text{goto}(I_7, a) = \text{closure}(\{S: BbBa \bullet\}) = \{S: BbBa \bullet\}$

	a	b	\$	S	A	B
0	r3/r4	r3/r4		1 2 3	$A: \epsilon$	$B: \epsilon$
1			acc			
2	s4					
3		s5				
4	r3	r3		6	$A: \epsilon$	
5	r4	r4		7	$B: \epsilon$	
6		s8				
7	s9					
8			r1			$S: AaAb$
9			r2			$S: BbBa$

SLR構文解析表に重複があるので、文法G4はSLR(1)でない。重複はエントリ(0, a)と(0, b)にあり、 $A: \epsilon$ の還元と $B: \epsilon$ の還元が衝突している。つまり還元/還元衝突がある。



# 解答(G5)

SLR(1)ではない

$I0 = \text{closure}(\{R : \bullet S\})$

$\{R:\bullet S, S:\bullet AaAb, A:\bullet, B:\bullet\}$	1			acc	R:S
1=goto(I0,S)=closure( $\{R:S\bullet\}$ )= $\{R: S\bullet\}$	2		s4		3
2=goto(I0,A)=closure( $\{S:A\bullet aAb\}$ )= $\{S:A\bullet aAb\}$	3	s5			
3=goto(I0,B)=closure( $\{S:B\bullet bBa\}$ )= $\{S:B\bullet bBa\}$	4	r4			E:true
4=goto(I2,a)=closure( $\{S:Aa\bullet Ab\}$ )	5   s2	r3	r3   6	S: $\epsilon$	
= $\{S:Aa\bullet Ab, A:\bullet\}$	6	s7/r2	r2	S:if E then S	
5=goto(I3,b)=closure( $\{S:Bb\bullet Ba\}$ )	7   s2	r3	r3   8	S: $\epsilon$	
= $\{S:Bb\bullet Ba, B:\bullet\}$	8	r1	r1	S:if E then S	

I6=goto(I4,A)=closure({S:AaA●b})={S:AaA●b} else S

I7=goto(I5,B)=closure({S:BbB●a})={S:BbB●a}

I8=goto(I6,b)=closure({S:AaAb●})={S:AaAb●}SLR構文解析表に重複があるので、文法G5はSLR(1)ではない。重複はエントリ(6, else)にあり、elseのシフトとS;if E then Sの還元が衝突している。つまりシフト/還元衝突がある。



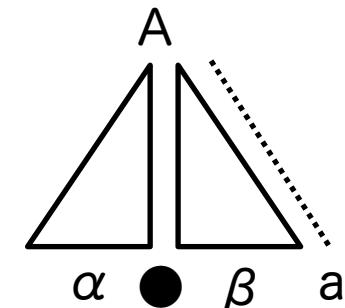
# SLR(1), LR(1), LALR(1)

- SLR(1)の文法クラスはやや小さい。
  - SLR(1)が使うLR(0)項は文脈を見ずに, follow集合で還元エントリを決めるから
  - LR(0)項の例 : [E: E + ● T]
- LR(1)の文法クラスは大きいが構文解析表が大きく効率が悪い
  - LR(1)が使うLR(1)項は左文脈で区別して項（の先読み記号）を作るから
  - LR(1)項の例 : [E: E + ● T, +]
- LALR(1)の文法クラスはSLR(1)とLR(1)の中間
  - LALR(1)はLR(1)項を使うので, follow集合だけのSLR(1)よりも構文解析は正確
  - LALR(1)の状態数はSLR(1)と同じ→効率はLR(1)より良い
    - LALR(1)は同じ第1要素を持つLR(1)項を同一視して構文解析表を圧縮



# LR(1)項

- LR(1)項
  - LR(1)項はLR(0)項に先読み記号(lookahead)  $a$ を追加したもの
  - LR(1)項の形式 :  $[A: \alpha \bullet \beta, a]$ 
    - ただし,  $a$ は終端記号で, 生成規則  $A: \alpha \beta$  の文脈の先読み記号
    - $a \in \text{follow}(A)$ だが,  $A: \alpha \beta$  という文脈に特定している点に注意
- LR(1)項の例 :  $[T: \bullet T^* F, *]$   $[T: \bullet T^* F, +]$ 
  - 略記法 : 上の2つをまとめて,  $[T: \bullet T^* F, */+]$





# LR(1)項の閉包演算

- 計算方法

- $\text{closure}(I) = I$

- repeat

- foreach 生成規則  $B: \gamma$  と終端記号  $b$  do

- if  $[A: \alpha \bullet B \beta, a] \in \text{closure}(I) \wedge b \in \text{first}(\beta a)$  then

- $\text{closure}(I) += \{ [B: \bullet \gamma, b] \}$

- until  $\text{closure}(I)$ に変更が無かった



# LR(1)項集合のgoto演算

- $\text{goto}(I, X)$ の計算方法
  - $\text{goto}(I, X) = \text{closure}(\{[A: \alpha X \bullet \beta, a] \mid [A: \alpha \bullet X \beta, a] \in I\})$
  - ただし、 $I$ はLR(1)項の集合。 $X$ は文法記号
  - 本質的にLR(0)項の集合のgoto演算と同じ
- 文法G2のLR(1)項集合の閉包とgoto演算の例（一部）
  - $I_0 = \text{closure}(\{[R: \bullet E, \$]\}) = \{[R: \bullet E, \$], [E: \bullet E + T, \$/+], [E: \bullet T, \$/+], [T: \bullet T^* F, \$/*/+], [T: \bullet F, \$/*/+], [F: \bullet (E), \$/*/+], [F: \bullet i, \$/*/+]\}$
  - $I_2 = \text{goto}(I_0, T) = \text{closure}(\{[E: T \bullet, \$/+], [T: T \bullet^* F, \$/*/+]\}) = \{[E: T \bullet, \$/+], [T: T \bullet^* F, \$/*/+]\}$
  - $I_4 = \text{goto}(I_0, ()) = \text{closure}(\{[F: (\bullet E), \$/*/+]\}) = \{[F: (\bullet E), \$/*/+], [E: \bullet E + T, )/+], [E: \bullet T, )/+], [T: \bullet T^* F, )/+/*], [T: \bullet F, )/+/*], [F: \bullet (E), )/+/*], [F: \bullet i, )/+/*]\}$
  - $I_{12} = \text{goto}(I_4, T) = \text{closure}(\{[E: T \bullet, )/+], [T: T \bullet^* F, )/+/*]\}) = \{[E: T \bullet, )/+], [T: T \bullet^* F, )/+/*]\}$

# 正準LR(1)集成 (canonical LR(1) collection)

- 入力：拡大文法  $G'$
- 出力：正準LR(1)集成 (=C)
- 計算方法
  - $C = \{\text{closure}(\{[S']: \bullet S; , \$]\})\};$   
repeat  
    foreach LR(1)項集合  $l \in C$  と文法記号  $X$  do  
        if  $\text{goto}(l, X) \neq \emptyset$   
            then  $C += \{\text{goto}(l, X)\};$   
until  $C$ に新たなLR(1)項集合が加えられなかった
  - 本質的に正準LR(0)集成の作成と同じ



# LR(1)構文解析表の作成

- 次の2点以外はSLR(1)と同じ
  - 正準LR(0)集成ではなく、正準LR(1)集成を作る
  - LR(1)項  $[A: \alpha \bullet, a] \in I_i$  (ただし  $A \neq S'$ ) がある時、エントリを  $\{ (i, x) \mid x \in \text{follow}(A) \}$  ではなく、 $(i, a)$  に対してのみ、reduce  $A: \alpha;$  とする



# LR(1)項集合の合併

- LR(1)項集合の合併=先読み記号を取ると同じになる  
LR(1)項集合同士をまとめて1つにすること。  
先読み記号は単純に和集合をとる。
- LALR(1)ではLR(1)項集合を合併して構文解析表を圧縮する。
- 例：p.136の2つのLR(1)項集合の合併は以下となる
  - $I_2 = \{[E:T\bullet, \$/+], [T:T\bullet^*F, \$/*/+]\}$
  - $I_{12} = \{[E:T\bullet, )/+], [T:T\bullet^*F, )/+/*]\}$
  - $\text{merge}(I_2, I_{12}) = \{[E:T\bullet, \$//+], [T:T\bullet^*F, \$//+/*]\}$
- 一般に $I$ と $I'$ が合併可能なら、 $\text{goto}(I, X)$ と $\text{goto}(I', X)$ も合併可能で、それは $\text{goto}(\text{merge}(I, I'), X)$ と同じ
  - LALR(1)の $\text{goto}$ 関係は、元のLR(1)の $\text{goto}$ 関係を重ねて作れる