

コンパイラ理論 8 LR構文解析

櫻井彰人

上昇型構文解析 bottom up parsing

- 上昇型構文解析は、下降型構文解析より一般的
 - そして同様に効率的
 - 下降型構文解析のアイデアに立脚
 - 実務上、推奨される手法
- LR parsing と呼ばれる
 - L はトークンを左から右へ left to right 読むことを示す
 - R は、パーサが最右導出 rightmost derivation 木を作ることを意味する

Based on Prof. Necula's lecture notes

2

一つの簡単な例

- LR 構文解析は左括りだし left-factoring は必要ないし、左再帰の文法も扱える

- 次の文法を考えよう:

$$E \rightarrow E + (E) \mid \text{int}$$

- 対象とする入力列: `int + (int) + (int)`

Based on Prof. Necula's lecture notes

3

基本アイデア

- LR 構文解析は、生成規則を逆に適用することによって入力列を開始記号に *reduce* する:

`str` ← 終端記号からなる入力列

repeat

- `str` 中で、 $A \rightarrow \beta$ が生成規則であるような β を見つけよ (i.e., $\text{str} = \alpha \beta \gamma$)

- `str` 中の β を A で置換えよ (i.e., str は $\alpha A \gamma$ となる)

until `str` = S (開始記号)

Based on Prof. Necula's lecture notes

4

LR構文解析動作原理 (1)

`int + (int) + (int)` $E \rightarrow E + (E) \mid \text{int}$

`int + (int) + (int)`

Based on Prof. Necula's lecture notes

5

LR構文解析動作原理 (2)

`int + (int) + (int)` $E \rightarrow E + (E) \mid \text{int}$
`E + (int) + (int)`

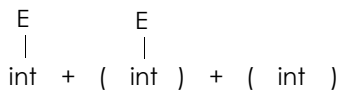
E
|
int + (int) + (int)

Based on Prof. Necula's lecture notes

6

LR構文解析動作原理 (3)

int + (int) + (int) $E \rightarrow E + (E) \mid \text{int}$
 E + (int) + (int)
 E + (E) + (int)

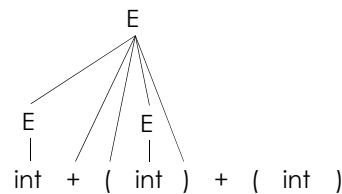


Based on Prof. Necula's lecture notes

7

LR構文解析動作原理 (4)

int + (int) + (int) $E \rightarrow E + (E) \mid \text{int}$
 E + (int) + (int)
 E + (E) + (int)
 E + (int)

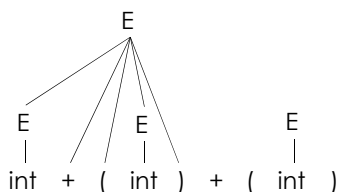


Based on Prof. Necula's lecture notes

8

LR構文解析動作原理 (5)

int + (int) + (int) $E \rightarrow E + (E) \mid \text{int}$
 E + (int) + (int)
 E + (E) + (int)
 E + (int)
 E + (E)



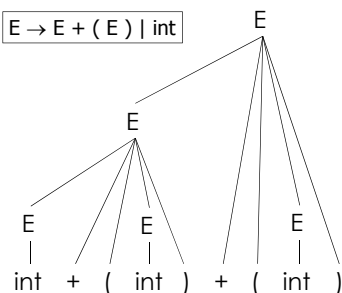
Based on Prof. Necula's lecture notes

9

LR構文解析動作原理 (6)

int + (int) + (int) $E \rightarrow E + (E) \mid \text{int}$
 E + (int) + (int)
 E + (E) + (int)
 E + (int)
 E + (E)
 E

最右導出 rightmost derivation を逆順に



Based on Prof. Necula's lecture notes

10

重要事項 #1

LR構文解析の重要事項 #1:

LRパーサは最右導出 rightmost derivation を逆順にたどる

Based on Prof. Necula's lecture notes

11

還元 reduction が発生する場所

重要事項 #1 からの帰結:

- $\alpha\beta\gamma$ をLR構文解析の途中の列とする
- 次の還元 reduction は $A \rightarrow \beta$ によると仮定する
- このとき γ は終端記号の列となる!

なぜ? 簡単. $\alpha A \gamma \rightarrow \alpha\beta\gamma$ は最右導出の1ステップであるから (Aは最右の非終端記号)

Based on Prof. Necula's lecture notes

12

記法

- 考え: 文字列を2つの部分列に分解
 - 右部分列(終端記号の列) パーサが未だ見ていない部分
 - 左部分列は、終端記号と非終端記号が混在
- 分割点を | でマークする
 - この | は列の一部ではない
- 初期には、入力全体が未処理: $|x_1x_2 \dots x_n$

Based on Prof. Necula's lecture notes

13

Shift-Reduce 構文解析

- LR構文解析には2つの基本動作がある:

Shift

Reduce

Based on Prof. Necula's lecture notes

14

Shift

- Shift:* 記号 | を場所 1 個だけ右へ移す
- 1 終端記号を左部分列に移動することに対応

$$E + (| int) \Rightarrow E + (int |)$$

Based on Prof. Necula's lecture notes

15

Reduce

- Reduce:* 左部分列の最右端に、生成規則を逆に適用する
- もし $E \rightarrow E + (E)$ が生成規則なら、

$$E + (E + (E) |) \Rightarrow E + (E |)$$

Based on Prof. Necula's lecture notes

16

Shift-Reduce 例

次に続く動作

| int + (int) + (int)\$ shift

int + (int) + (int)
↑

Shift-Reduce 例

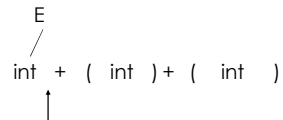
次に続く動作

| int + (int) + (int)\$ shift
int | + (int) + (int)\$ red. $E \rightarrow int$

int + (int) + (int)
↑

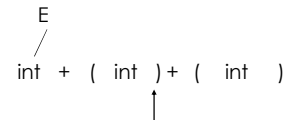
Shift-Reduce 例

次に続く動作
 | int + (int) + (int)\$ shift
 int | + (int) + (int)\$ red. E → int
 E | + (int) + (int)\$ shift 3 回



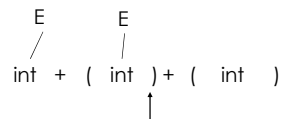
Shift-Reduce 例

次に続く動作
 | int + (int) + (int)\$ shift
 int | + (int) + (int)\$ red. E → int
 E | + (int) + (int)\$ shift 3 回
 E + (int |) + (int)\$ red. E → int



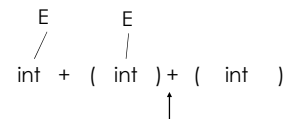
Shift-Reduce 例

次に続く動作
 | int + (int) + (int)\$ shift
 int | + (int) + (int)\$ red. E → int
 E | + (int) + (int)\$ shift 3 回
 E + (int |) + (int)\$ red. E → int
 E + (E |) + (int)\$ shift
 E + (E |) + (int)\$ shift



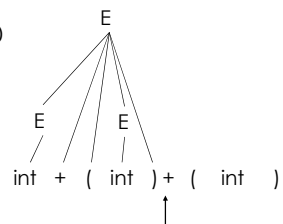
Shift-Reduce 例

次に続く動作
 | int + (int) + (int)\$ shift
 int | + (int) + (int)\$ red. E → int
 E | + (int) + (int)\$ shift 3 回
 E + (int |) + (int)\$ red. E → int
 E + (E |) + (int)\$ shift
 E + (E |) + (int)\$ shift
 E + (E |) + (int)\$ red. E → E + (E)



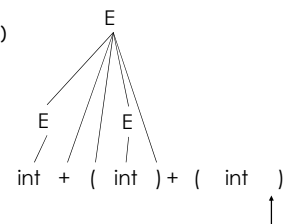
Shift-Reduce 例

次に続く動作
 | int + (int) + (int)\$ shift
 int | + (int) + (int)\$ red. E → int
 E | + (int) + (int)\$ shift 3 回
 E + (int |) + (int)\$ red. E → int
 E + (E |) + (int)\$ shift
 E + (E |) + (int)\$ red. E → E + (E)
 E | + (int)\$ shift 3 回



Shift-Reduce 例

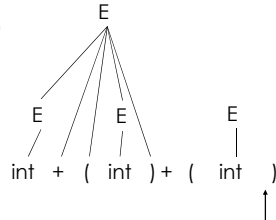
次に続く動作
 | int + (int) + (int)\$ shift
 int | + (int) + (int)\$ red. E → int
 E | + (int) + (int)\$ shift 3 回
 E + (int |) + (int)\$ red. E → int
 E + (E |) + (int)\$ shift
 E + (E |) + (int)\$ red. E → E + (E)
 E | + (int)\$ shift 3 回
 E + (int |)\$ red. E → int



Shift-Reduce 例

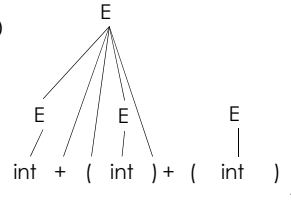
次に続く動作

int + (int) + (int)\$	shift
int + (int) + (int)\$	red. E → int
E + (int) + (int)\$	shift 3 回
E + (int) + (int)\$	red. E → int
E + (E) + (int)\$	shift
E + (E) + (int)\$	red. E → E + (E)
E + (int)\$	shift 3 回
E + (int)\$	red. E → int
E + (E)\$	shift
E + (E) \$	red. E → E + (E)



Shift-Reduce 例

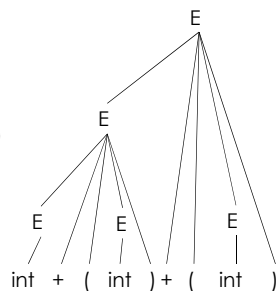
int + (int) + (int)\$	shift
int + (int) + (int)\$	red. E → int
E + (int) + (int)\$	shift 3 回
E + (int) + (int)\$	red. E → int
E + (E) + (int)\$	shift
E + (E) + (int)\$	red. E → E + (E)
E + (int)\$	shift 3 回
E + (int)\$	red. E → int
E + (E)\$	shift
E + (E) \$	red. E → E + (E)



Shift-Reduce 例

次に続く動作

int + (int) + (int)\$	shift
int + (int) + (int)\$	red. E → int
E + (int) + (int)\$	shift 3 回
E + (int) + (int)\$	red. E → int
E + (E) + (int)\$	shift
E + (E) + (int)\$	red. E → E + (E)
E + (int)\$	shift 3 回
E + (int)\$	red. E → int
E + (E)\$	shift
E + (E) \$	red. E → E + (E)
E \$	受理



スタック

- 左部分列はスタックとして実装できる
 - スタックのトップは先ほどの 1
- Shift は 1 終端記号をスタックに積むこと
- Reduce は 0 個以上の記号 (生成規則の右辺) をスタックからポップし 1 個の非終端記号 (生成規則の左辺) をスタックに積む

Based on Prof. Necula's lecture notes

28

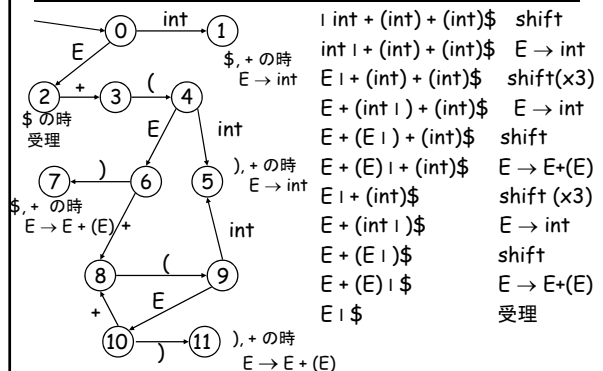
主要課題: いつ Shift, いつ Reduce?

- 左部分列 (スタック) に基づき決定
 - しかし、左部分列の長さは不定 (いくらでも長い可能性あり)、簡単な計算方法はないだろうか?
- アイデア: 決定性有限状態オートマトン (DFA) を用いて、shift や reduce をするタイミングを決定
 - これのできる範囲 (の文法) を考えようという意味。
 - この DFA への入力は、スタック上の値 (底からトップへ)
 - DFA の受理言語は、終端記号と非終端記号からなる
- この DFA をスタック上の値に対して実行させ、最後の状態 X と 1 の後のトークン tok を調べ
 - もし X にラベル tok をもつ状態遷移があるなら shift
 - もし X がラベル "tok" であれば $A \rightarrow \beta$ をもつなら reduce

つまり、この DFA は枝にも (終了) 状態にもラベルがついている²⁹

LR(1) 構文解析 例

フロンティアが与えられたと仮定し、それを左から読んで、次にすべき動作を決定するオートマトンを作成する。入力文字で遷移し、到った終了状態で、次の動作を指定する。



DFA の表現

- 例の DFA を2次元の表で表現
 - 順位文法(や字句解析)と同様
- 行が DFA の状態に対応
- 列が終端記号と非終端記号とに対応
- 普通は、この列を2つに分割:
 - 終端記号の列: action 表
 - 非終端記号の列: goto 表

Based on Prof. Necula's lecture notes

31

DFA の表現. 例

- 例に使用している DFA の一部:

状態遷移が、終端記号では s, 非終端記号では g となっているのは、実際の運用状態、すなわち、「先頭からscanする」のではない状況を想定しているからである。前スライドで示した、先頭からscanする方法では、どちらにおいても、shift しなければならぬ。しかし、実際の運用状態(先頭からのscan)はせず、「reduce」によって得られた最後の非終端記号をstackにpushした直後から再開する場合には、当該非終端記号が既にstackにあるので、shiftする必要がないというわけである。

	int	+	()	\$	E
...						
3						
4	s5					g6
5	rE → int					rE → int
6	s8					s7
7	rE → E+(E)					rE → E+(E)
...						

Based on Prof. Necula's lecture notes

32

LR 構文解析アルゴリズム

- shift 動作や reduce 動作のあと、DFA をスタックに再適用する
 - しかし、これは無駄が多い。同じ作業が繰返されるから
- スタック上の各要素に、DFA をスタックの底からそこまで適用した結果の状態を記憶させよう
- LR パーサは、スタックを次の形で管理する
 - $\langle \text{sym}_1, \text{state}_1 \rangle \dots \langle \text{sym}_n, \text{state}_n \rangle$
 - state_k は、DFA を $\text{sym}_1 \dots \text{sym}_{k-1}$ に適用した結果の状態
 - sym_k は reduce した式の左辺(または先読み記号)

Based on Prof. Necula's lecture notes

33

LR 構文解析アルゴリズム

	int	+	()	\$	E
...						
3						
4	s5					g6
5	rE → int					rE → int
6	s8					s7
7	rE → E+(E)					rE → E+(E)
...						

I = w\$ を(初期)入力列
 j = 0
 DFA state 0 を開始状態
 stack = $\langle \text{dummy}, 0 \rangle$

repeat

case action[top_state(stack), I[j]] of

shift k: push (I[j++], k)

reduce X → α:

pop |α| 組

push (X, Goto[top_state(stack), X])

accept: 正常終了

error: 終了しエラーの報告

Based on Prof. Necula's lecture notes

34

LR 構文解析アルゴリズム

↑

state_n) < , 6 >

< sym_n, ... > < ') 4 >

state₁) < E, 4 >

< sym₁, ... > < '(' 3 >

↓

< sym₁, state₁ > < '(' 3 >

... < E, 4 >

< sym_n, state_n > < ') 6 >

↑

	int	+	()	\$	E
...						
3						
4	s5					g6
5	rE → int					rE → int
6	s8					s7
7	rE → E+(E)					rE → E+(E)
...						

Based on Prof. Necula's lecture notes

36

LR 構文解析 注釈

- LLより多くの文法が解析できる
- プログラミング言語は、通常、LR で記述できる
- 単純な表で表現できる
- 表を作成するためのツールがある
- では、その方法は?

Based on Prof. Necula's lecture notes

36

LRパーサの自動生成

概要

- LR構文解析の復習
- 構文解析用 DFA の作成
- パーサ作成ツール

Based on Prof. Necula's lecture notes

38

LR構文解析のアイデア (復習)

- 上昇型構文解析は、入力列を開始記号に書き換える
- パーサの状態は、次の形で表現できる
 - $\alpha | \gamma$
 - α はスタックで、終端記号と非終端記号からなる
 - γ は終端記号の列。パーサはまだ見ていない部分
- 初期状態: $| x_1 x_2 \dots x_n$

Based on Prof. Necula's lecture notes

39

Shift 動作と Reduce 動作 (復習)

- CFGの例: $E \rightarrow \text{int} \mid E + (E)$
- 上昇型構文解析には2つの動作がある:
 - **Shift** は終端記号を入力列からスタックへプッシュ
 $E + (\text{int}) \Rightarrow E + (\text{int} |)$
 - **Reduce** は 0 個以上の記号 (生成規則の右辺) をスタックからポップし 1 個の非終端記号 (生成規則の左辺) をスタックに積む
 $E + (E + (E) |) \Rightarrow E + (E |)$

Based on Prof. Necula's lecture notes

40

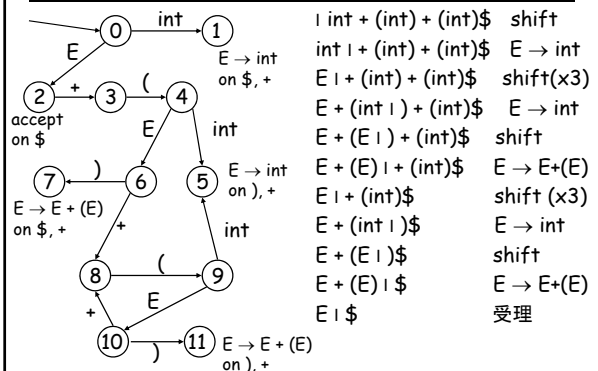
主要課題: いつ Shift, いつ Reduce?

- 左部分列 (スタック) に基づき決定
 - しかし、左部分列の長さは不定 (いくらでも長い可能性あり)、簡単な計算方法はないだろうか?
- アイデア: 決定性有限状態オートマトン (DFA) を用いて、shift や reduce をするタイミングを決定
 - これのできる範囲 (の文法) を考えようという意味。
 - この DFA への入力は、スタック上の値 (底からトップへ)
 - DFA の受理言語は、終端記号と非終端記号からなる
- この DFA をスタック上の値に対して実行させ、最後の状態 X と $|$ の後のトークン tok を調べ
 - もし X にラベル tok をもつ状態遷移があるなら **shift**
 - もし X にラベル "tok がある時 $A \rightarrow \beta$ " の遷移があるなら **reduce**

Based on Prof. Necula's lecture notes

41

LR(1) 構文解析. 例



復習終了

主要課題: 例の DFA をどう作るか?

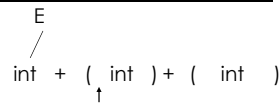
- スタックはパーサの現在の状態を表現している
 - 今探している非終端記号 (reduce の結果となるもの)
 - 今探している (生成規則の) 右辺 (reduce の対象)
 - その右辺の途中までの内容 (その途中結果)

Based on Prof. Necula's lecture notes

44

解析中の文脈(context)とは

- 途中状態の例:



- この時スタックは $E + (\text{int}) + (\text{int})$
- 文脈:
 - 今探している規則は $E \rightarrow E + (\bullet E)$
 - これまでに目撃したのは、右辺のうち $E + ($
 - 同時にこれも探している $E \rightarrow \bullet \text{int}$ または $E \rightarrow \bullet E + (E)$
 - まだ、右辺のどれも目撃していない
- 従って、DFA の一個の状態で、複数個の文脈を表現する

Based on Prof. Necula's lecture notes

45

LR(1) アイテム

- LR(1) アイテム は次の形をしたペア:

$$X \rightarrow \alpha \bullet \beta, a$$

- $X \rightarrow \alpha \beta$ は構文規則
- a は終端記号 (先読み記号 lookahead terminal)
- LR(1) は一個先読みすることを意味する
- $[X \rightarrow \alpha \bullet \beta, a]$ はパーサの文脈を表すことになる
 - 今探しているのは、 X とそれに続く a であって、
 - スタックのトップには α が既にある
 - 従って、 βa から導出される先頭文字列 prefix を探す必要がある

Based on Prof. Necula's lecture notes

46

注

- 記号 \uparrow は、もともと、スタック(左部分列)と入力列中未処理の部分との境目として使用した
 - $\alpha \uparrow \gamma$, ただし α はスタックであり γ は残りの終端記号列
 - これは、処理する文字列からみた、パーサの状態
 - パーサの心積もりからいうと、探索途中の生成規則とそのどこまで見つけたかという場所の集まり(複数個ある)。これを表すのが、LR(1) アイテムの左側である
- LR(1) アイテムにおいて、 \bullet は生成規則右辺の(ありうる)先頭部分 prefix をマークするのに使用:
 - $X \rightarrow \alpha \bullet \beta, a$
 - ここで β は非終端記号を含むこともある
- どちらの場合もスタックは左側にある(とみなしている)

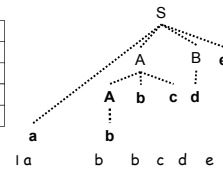
Based on Prof. Necula's lecture notes

47

注: 文脈

生成規則
$S \rightarrow aABe$
$A \rightarrow Abc$
$A \rightarrow b$
$B \rightarrow d$

文脈
$S \rightarrow \bullet aABe$



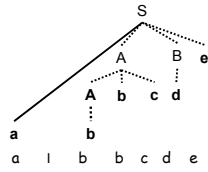
Based on Prof. Necula's lecture notes

48

注: 文脈

生成規則
$S \rightarrow aABe$
$A \rightarrow Abc$
$A \rightarrow b$
$B \rightarrow d$

文脈
$S \rightarrow a \bullet ABe$
$A \rightarrow \bullet Abc$
$A \rightarrow \bullet b$



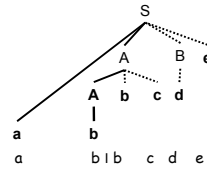
Based on Prof. Necula's lecture notes

49

注: 文脈

生成規則
$S \rightarrow aABe$
$A \rightarrow Abc$
$A \rightarrow b$
$B \rightarrow d$

文脈
$S \rightarrow a \bullet ABe$
$A \rightarrow \bullet Abc$



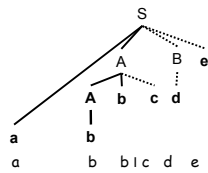
Based on Prof. Necula's lecture notes

50

注: 文脈

生成規則
$S \rightarrow aABe$
$A \rightarrow Abc$
$A \rightarrow b$
$B \rightarrow d$

文脈
$S \rightarrow a \bullet ABe$
$A \rightarrow \bullet Ab \cdot c$



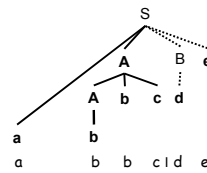
Based on Prof. Necula's lecture notes

51

注: 文脈

生成規則
$S \rightarrow aABe$
$A \rightarrow Abc$
$A \rightarrow b$
$B \rightarrow d$

文脈
$S \rightarrow a \bullet ABe$
$B \rightarrow \bullet d$



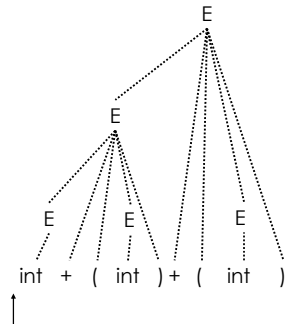
Based on Prof. Necula's lecture notes

52

文脈(必要な先読みをサポートしている)

$E \rightarrow int$
$E \rightarrow E + (E)$

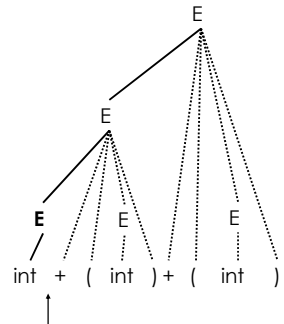
$E \rightarrow \bullet int$
 $E \rightarrow \bullet E + (E)$



文脈(必要な先読みをサポートしている)

$E \rightarrow int$
$E \rightarrow E + (E)$

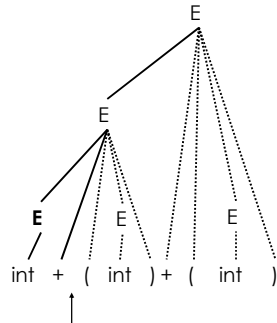
$E \rightarrow \bullet E + (E)$
 $E \rightarrow E \bullet + (E)$



文脈(必要な先読みをサポートしている)

$E \rightarrow \text{int}$
 $E \rightarrow E + (E)$

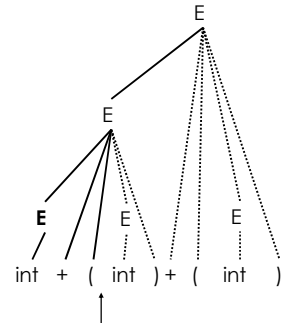
$E \rightarrow \bullet E + (E)$
 $E \rightarrow E \bullet + (E)$



文脈(必要な先読みをサポートしている)

$E \rightarrow \text{int}$
 $E \rightarrow E + (E)$

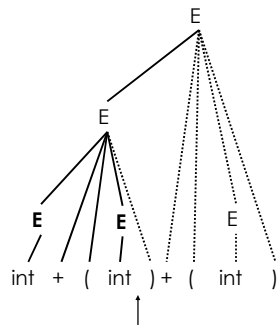
$E \rightarrow \bullet E + (E)$
 $E \rightarrow E \bullet + (E)$
 $E \rightarrow \bullet \text{int}$
 $E \rightarrow \bullet E + (E)$



文脈(必要な先読みをサポートしている)

$E \rightarrow \text{int}$
 $E \rightarrow E + (E)$

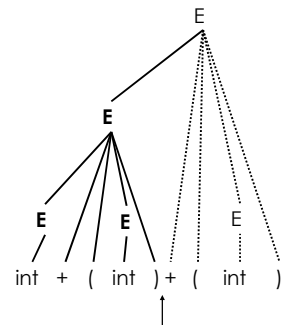
$E \rightarrow \bullet E + (E)$
 $E \rightarrow E + (E \bullet)$



文脈(必要な先読みをサポートしている)

$E \rightarrow \text{int}$
 $E \rightarrow E + (E)$

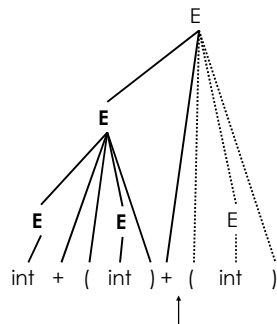
$E \rightarrow \bullet E + (E)$
 $E \rightarrow E \bullet + (E)$



文脈(必要な先読みをサポートしている)

$E \rightarrow \text{int}$
 $E \rightarrow E + (E)$

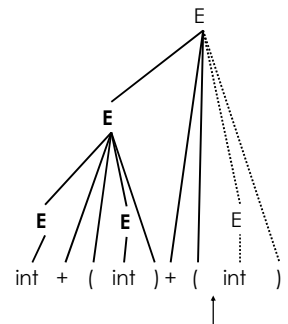
$E \rightarrow \bullet E + (E)$
 $E \rightarrow E + \bullet (E)$



文脈(必要な先読みをサポートしている)

$E \rightarrow \text{int}$
 $E \rightarrow E + (E)$

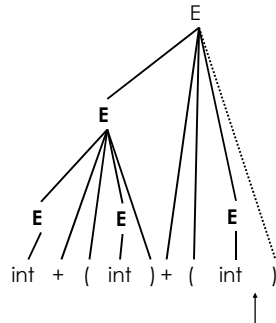
$E \rightarrow \bullet E + (E)$
 $E \rightarrow E + (\bullet E)$
 $E \rightarrow \bullet \text{int}$
 $E \rightarrow \bullet E + (E)$



文脈(必要な先読みをサポートしている)

$E \rightarrow \text{int}$
 $E \rightarrow E + (E)$

$E \rightarrow \bullet E + (E)$
 $E \rightarrow E + (E \bullet)$



約束事

- 新しい開始記号 S とそれに伴い新しい生成規則 $S \rightarrow E$ を付け加える
 - ただし E はこれまで使っていた開始記号
 - E に唯一つの生成規則しかなければ、必要なし
- 初期の文脈は:
 - $S \rightarrow \bullet E, \$$
 - 探すものは $E\$$ から導出される記号列で S にいたるもの
 - スタックは空

Based on Prof. Necula's lecture notes

62

LR(1) アイテム (続)

- この文脈
 - $E \rightarrow E + \bullet (E), +$
 - では、もし $($ が続いたら shift を行ない、次の文脈になる
 - $E \rightarrow E + (\bullet E), +$
- この文脈
 - $E \rightarrow E + (E) \bullet, +$
 - では、 reduce が、規則 $E \rightarrow E + (E)$ を用いてできる
 - ただし、それは $+$ が続いているときに限られる

Based on Prof. Necula's lecture notes

63

LR(1) アイテム (続)

- このアイテムの表す文脈で考えよう
 - $E \rightarrow E + (\bullet E), +$
- つぎの文字列は $E) +$ から導出されたもの
- E には2つの生成規則がある
 - $E \rightarrow \text{int}$ と $E \rightarrow E + (E)$
- これを記述するのに文脈を拡張して2個のアイテムを含むようにする:
 - $E \rightarrow \bullet \text{int},)$
 - $E \rightarrow \bullet E + (E),)$

Based on Prof. Necula's lecture notes

64

閉包 Closure 演算

- 文脈を拡張して新たなアイテムを含むようにする演算を閉包演算と呼ぶ

Closure(Items) =
 repeat
 for Items 中の各 $[X \rightarrow \alpha \bullet Y\beta, \alpha]$
 for 各生成規則 $Y \rightarrow \gamma$
 for 各 $b \in \text{First}(\beta\alpha)$ /* $\text{First}(\beta\alpha)$ は空ではない */
 $[Y \rightarrow \bullet \gamma, b]$ を Items に追加
 until Items が変わらない

Based on Prof. Necula's lecture notes

65

構文解析 DFA の作成 (1)

- 開始文脈の作成: $\text{Closure}(\{S \rightarrow \bullet E, \$\})$

$S \rightarrow \bullet E, \$$
 $E \rightarrow \bullet E + (E), \$$
 $E \rightarrow \bullet \text{int}, \$$
 $E \rightarrow \bullet E + (E), +$
 $E \rightarrow \bullet \text{int}, +$

- 次のように簡略標記する:

$S \rightarrow \bullet E, \$$
 $E \rightarrow \bullet E + (E), \$/+$
 $E \rightarrow \bullet \text{int}, \$/+$

Based on Prof. Necula's lecture notes

66

構文解析 DFA の作成 (2)

- DFA 状態は, LR(1) アイテムの"閉じた"集合
 - すなわち, 閉包 Closure を行なう (因みに, 閉じた= closed)
- 開始状態に含まれる: $[S \rightarrow \bullet E, \$]$
- $[X \rightarrow \alpha \bullet, b]$ をもつ状態につけるラベル: "bがあるとき, $X \rightarrow \alpha$ を用いて還元"
- 状態遷移は ...

Based on Prof. Necula's lecture notes

67

DFA 状態遷移

- "State" という状態に $[X \rightarrow a \bullet yb, b]$ があるとす。この時, 先の状態が "Transition(State, y)" であるラベル y という遷移を設ける
 - y は終端記号または非終端記号

Transition(State, y)

Items $\leftarrow \emptyset$

for 各 $[X \rightarrow \alpha \bullet \gamma\beta, b] \in \text{State}$

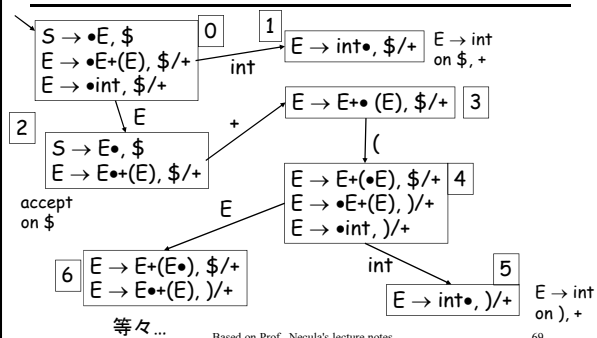
$[X \rightarrow \alpha\gamma \bullet \beta, b]$ を Items に追加

return Closure(Items)

Based on Prof. Necula's lecture notes

68

構文解析 DFA の作成. 例



Based on Prof. Necula's lecture notes

69

LR 構文解析表. 注意

- 構文解析表 (i.e. the DFA) は, CFG が与えられれば自動的に作ることができる
- とはいっても, パーサ生成器を使うには, そのメカニズム知っている必要がある
 - E.g., アイテムを用いてエラーリポートがあつたりする
- どんなエラーがありうるか?

Based on Prof. Necula's lecture notes

70

シフト還元衝突 Shift/Reduce Conflicts

- DFA 状態に次の2つが含まれているとしよう $[X \rightarrow \alpha \bullet a\beta, b]$ と $[Y \rightarrow \gamma \bullet, a]$
- このとき, 入力 "a" に対して次のどちらもありうる
 - Shift して状態 $[X \rightarrow \alpha a \bullet \beta, b]$ になる, または
 - Reduce を $Y \rightarrow \gamma$ で行なう
- これを, シフト還元衝突 shift-reduce conflict と呼ぶ

Based on Prof. Necula's lecture notes

71

Shift/Reduce Conflicts

- よくあるのは, 文法の曖昧性
- 古典的な例: the dangling else
 - $S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \mid \text{OTHER}$
- このとき, DFA 状態で次のものをもつものがある
 - $[S \rightarrow \text{if } E \text{ then } S \bullet, \text{ else}]$
 - $[S \rightarrow \text{if } E \text{ then } S \bullet \text{ else } S, x]$
- もし else が次にくると shift も reduce もありうる
- default (bison, CUP, 等) は shift

Based on Prof. Necula's lecture notes

72

もう一つ Shift/Reduce Conflicts

- 次の曖昧な文法を考えよう
 $E \rightarrow E + E \mid E * E \mid \text{int}$
- 次のものをもつ状態がある
 $[E \rightarrow E * \bullet E, +]$ $[E \rightarrow E * E \bullet, +]$
 $[E \rightarrow \bullet E + E, +] \Rightarrow^E [E \rightarrow E \bullet + E, +]$
...
- 入力に + があると, shift/reduce があろうる
- ここでは reduce すべし(* の結合は + より強い)
- 回避策: 優先順位(* と +)の宣言

Based on Prof. Necula's lecture notes

73

更に Shift/Reduce Conflicts

- bison では順位と結合性 associativity を宣言:
%left +
%left *
- 規則の優先順位 = 最後の終端記号の優先順位
- override 可能。マニュアルをみるのが一番。
- shift/reduce 衝突の解消に shift を用いるのは:
- 規則にも終端記号にも優先順位が宣言されていない
- 入力記号の優先順位は規則のそれより高い
- 優先順位が同じで、右結合性 right associative

Based on Prof. Necula's lecture notes

74

優先順位を用いて S/R Conflicts をとく

- 例の例にもどろう:
 $[E \rightarrow E * \bullet E, +]$ $[E \rightarrow E * E \bullet, +]$
 $[E \rightarrow \bullet E + E, +] \Rightarrow^E [E \rightarrow E \bullet + E, +]$
...
- ここでは reduce を選ぶ. なぜなら, 優先順位は、規則 $E \rightarrow E * E$ の方が終端記号 + より高い

Based on Prof. Necula's lecture notes

75

優先順位を用いて S/R Conflicts をとく

- 例によって例の文法
 $E \rightarrow E + E \mid E * E \mid \text{int}$
- 次の状態もある
 $[E \rightarrow E + \bullet E, +]$ $[E \rightarrow E + E \bullet, +]$
 $[E \rightarrow \bullet E + E, +] \Rightarrow^E [E \rightarrow E \bullet + E, +]$
...
- 入力 + に対して, shift/reduce ともにあろうる
- reduce を選ぶ. というのも $E \rightarrow E + E$ と + は同じ優先順位をもち + が左結合的 left-associative であるから

Based on Prof. Necula's lecture notes

76

優先順位を用いて S/R Conflicts をとく

- dangling else 問題に戻ろう
 $[S \rightarrow \text{if } E \text{ then } S \bullet, \text{ else}]$
 $[S \rightarrow \text{if } E \text{ then } S \bullet \text{ else } S, x]$
- これは、優先順位を else の方が then より高いと宣言することで解消できる
- または、default に任せて shift 動作としてもよい
- でも、これは、パーサのハッキングに見える
- 優先順位の過剰使用は避けた方がよい. さもないと、予期しなかった構文木が得られることになる

Based on Prof. Necula's lecture notes

77

Reduce/Reduce Conflicts

- かりに DFA 状態に次の二つがあったとする
 $[X \rightarrow \alpha \bullet, a]$ and $[Y \rightarrow \beta \bullet, a]$
- そうすると、入力に "a" があるとどちらの規則を用いて還元してよいかかわらなくなる
- これは reduce/reduce conflict と呼ばれる

Based on Prof. Necula's lecture notes

78

Reduce/Reduce Conflicts

- 通常は、文法の曖昧性による
- 例: 識別子の列
 $S \rightarrow \varepsilon \mid id \mid id S$
- id に対して、2個の構文木がある
 $S \rightarrow id$
 $S \rightarrow id S \rightarrow id$
- パーサはどんな混乱をきたすのだろうか?

Based on Prof. Necula's lecture notes

79

さらに Reduce/Reduce Conflicts

- 次の状態を考えよう
 $[S' \rightarrow \bullet S, \$]$ $[S \rightarrow id \bullet S, \$]$
 $[S \rightarrow \bullet, \$] \xrightarrow{id} [S \rightarrow \bullet, \$]$
 $[S \rightarrow \bullet id, \$]$ $[S \rightarrow \bullet id, \$]$
 $[S \rightarrow \bullet id S, \$]$ $[S \rightarrow \bullet id S, \$]$
- 入力 $\$$ が現れると reduce/reduce conflict
 $S' \rightarrow S \rightarrow id$
 $S' \rightarrow S \rightarrow id S \rightarrow id$
- 文法を書き換えるべき: $S \rightarrow \varepsilon \mid id S$

Based on Prof. Necula's lecture notes

80

パーサ作成ツールの利用

- パーサ作成ツール parser generators を用いると CFG から 構文解析 DFA が作れる
 - 衝突 conflicts は、優先順位の宣言と default を使って解消する
 - 構文解析アルゴリズムは、どの文法でも同じ (そして、ライブラリとして提供されている)
- しかし、殆どの作成ツールは DFA を作るのに、ここで述べた方法は用いていない
 - というのも LR(1) 解析 DFA は単純な言語に対しても何 1000 の状態を作ってしまうから

Based on Prof. Necula's lecture notes

81

LR(1) 構文解析表は大きい

- けれども多くの状態は、実質、同じ, e.g.
 $E \rightarrow int \bullet, \$/+$ $E \rightarrow int$ on $\$, +$ と $E \rightarrow int \bullet,)/+$ $E \rightarrow int$ on $) , +$
- アイデア: 先読みトークンだけが違う DFA 状態はマージしてしまう
 - 同じ核 core をもつ状態ということにする
- 上記の例では
 $E \rightarrow int \bullet, \$/+)$ $E \rightarrow int$ on $\$, +,)$

Based on Prof. Necula's lecture notes

82

LR アイテム集合のコア core

- 定義: LRアイテム集合のコア core とは、アイテムの第一要素の集合
 - すなわち、先読み終端記号は省くということ
- 例: この
 $\{ [X \rightarrow \alpha \bullet \beta, b], [Y \rightarrow \gamma \bullet \delta, d] \}$
 コアはこれ
 $\{ X \rightarrow \alpha \bullet \beta, Y \rightarrow \gamma \bullet \delta \}$

Based on Prof. Necula's lecture notes

83

LALR 状態

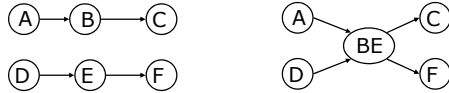
- 例えば、次の LR(1) 状態を考える
 $\{ [X \rightarrow \alpha \bullet, a], [Y \rightarrow \beta \bullet, c] \}$
 $\{ [X \rightarrow \alpha \bullet, b], [Y \rightarrow \beta \bullet, d] \}$
- 同じコアを持つのでマージできる
- マージした状態には次のものがある:
 $\{ [X \rightarrow \alpha \bullet, a/b], [Y \rightarrow \beta \bullet, c/d] \}$
- これらは LALR(1) 状態と呼ばれる
 - もととなった語: LookAhead LR
 - たいてい、LALR(1) の状態数は、LR(1) の 10分の1

Based on Prof. Necula's lecture notes

84

LALR(1) DFA

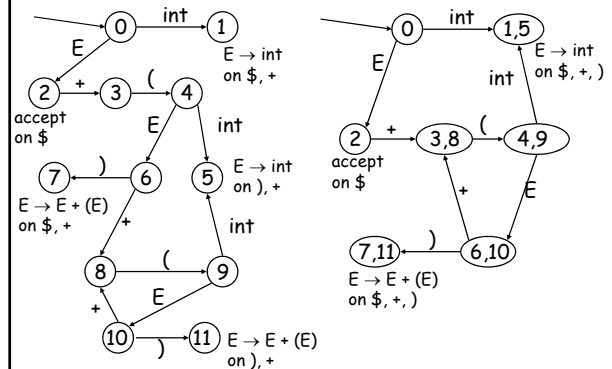
- 全ての状態が異なるコアをもつまで繰返す
 - 同じコアをもつ二つの異なる状態を選ぶ
 - 二つの状態をマージして、アイテムを全部もつ新しい状態を作る
 - (前の状態へのエッジを)新しい状態へのエッジとする
 - (前の状態からのエッジを)新しい状態からのエッジとする



Based on Prof. Necula's lecture notes

85

LR(1) から LALR(1) への変換. 例.



LALR パーサの conflicts

- 例えば, 次の LR(1) 状態を考える
 - $\{[X \rightarrow \alpha \bullet, a], [Y \rightarrow \beta \bullet, b]\}$
 - $\{[X \rightarrow \alpha \bullet, b], [Y \rightarrow \beta \bullet, a]\}$
- マージした LALR(1) 状態は
 - $\{[X \rightarrow \alpha \bullet, a/b], [Y \rightarrow \beta \bullet, a/b]\}$
- 新しく reduce/reduce conflict が発生
- しかし, 実応用では稀なケースである

Based on Prof. Necula's lecture notes

87

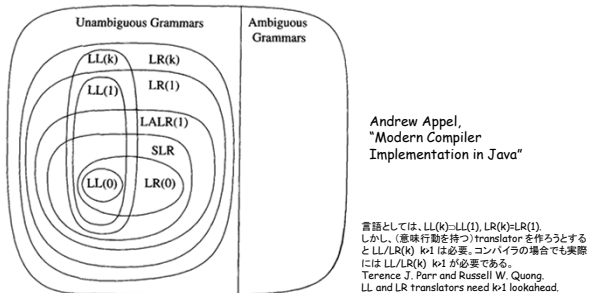
LALR vs. LR 解析

- LALR 言語は自然 natural ではない
 - 効率化を目指した, LR 言語へのハッキング
- まともな reasonable プログラム言語には, たいいてい, LALR(1) 文法がある
- LALR(1) は, 今では, プログラミング言語とパーサ作成ツールにとっては, 標準となっている

Based on Prof. Necula's lecture notes

88

文法クラスの階層



Based on Prof. Necula's lecture notes

89

注: 構文解析

- 構文解析
 - 理論的基盤: 文脈自由文法 context-free grammars
 - 単純なパーサ: LL(1)
 - より強力なパーサ: LR(1)
 - 効率重視のハッキング: LALR(1)
 - LALR(1) パーサ作成ツール parser generators
- 次は意味解析

Based on Prof. Necula's lecture notes

90

補遺: LR 構文解析

風変わりな Reduce/Reduce Conflicts
LALR にまつわる
(Bison マニュアルより)

風変わりな Reduce/Reduce Conflicts

- 次の文法を考える

$$\begin{aligned} S &\rightarrow PR, & NL &\rightarrow N \mid N, NL \\ P &\rightarrow T \mid NL : T & R &\rightarrow T \mid N : T \\ N &\rightarrow id & T &\rightarrow id \end{aligned}$$
- P - パラメータ仕様 parameters specification
- R - 結果の仕様 result specification
- N - パラメータまたは結果の識別子
- T - 型名
- NL - 識別名のリスト

Based on Prof. Necula's lecture notes

92

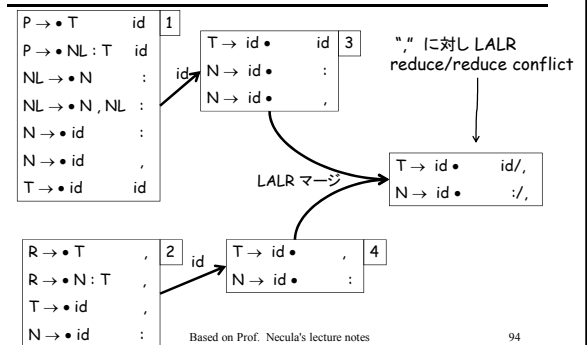
LALR(1) Reduce/Reduce Conflicts

- P において id は
 - N, もし, が続く
 - T, もし id が続く
- R において id は
 - N, もし : が続く
 - T, もし, が続く
- これは LR(1) 文法.
- しかし, LALR(1) ではない. なぜ?
 - 理由は少々分かりにくい

Based on Prof. Necula's lecture notes

93

LR(1) 状態の一部



Based on Prof. Necula's lecture notes

94

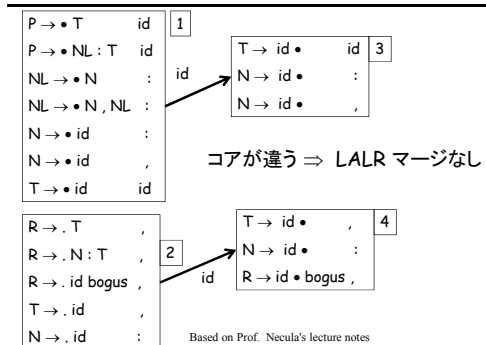
何が起きたか?

- 二つの相異なる状態が、ともに同じコアを持つために、混同されてしまった
- 修理: ダミーの生成規則を追加して、この2個の状態を区別する
- E.g., 下記を追加
 - $R \rightarrow id \text{ bogus}$
 - bogus は字句解析 lexer では使用しない終端記号
 - この生成規則は、構文解析中に使用されない!
 - しかし、R と P とを分離するのに役立つ

Based on Prof. Necula's lecture notes

95

修理後の LR(1) 状態



Based on Prof. Necula's lecture notes

96