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 とも呼ばれる
  - 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

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 → E + (E) | int

- 対象とする入力列: int + (int) + (int)

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 Ã 終端記号からなる入力列

repeat

- str 中で、A → β が生成規則であるようなβ を見つけよ (i.e., str = α β γ)
- str 中の β を A で置換えよ(i.e., str は α A γ となる)

until str = S

LR構文解析動作原理 (1)

int + (int) + (int)

E → E + (E) | int

int + (int) + (int)

LR構文解析動作原理 (2)

int + (int) + (int)

E + (int) + (int)

E → E + (E) | int

E

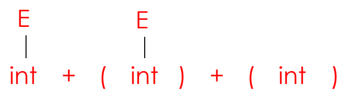
|

int + (int) + (int)

LR構文解析動作原理 (3)

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)

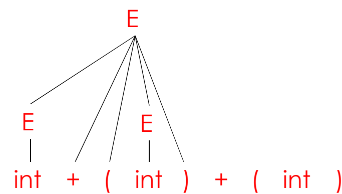
E → E + (E) | int



LR構文解析動作原理 (4)

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)

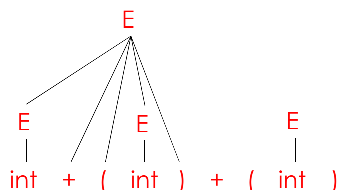
E → E + (E) | int



LR構文解析動作原理 (5)

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)
E + (E)

E → E + (E) | int

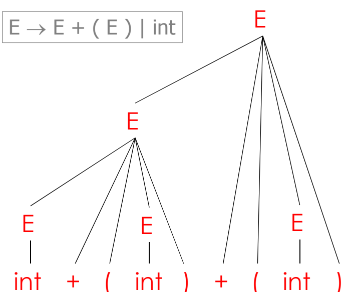


LR構文解析動作原理 (6)

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)
E + (E)
E

E → E + (E) | int

最右導出 rightmost derivation を逆順に



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

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）

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 I でマークする
  - この I は列の一部ではない
- 初期には, 入力全体が未処理: $I x_1 x_2 \ldots x_n$

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

Shift

Shift: 記号 I 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 + (I int) ⇒ E + (int I)

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 → E + (E) が生成規則なら,

E + (E + (E) I) ⇒ E +(E I)

Shift-Reduce 例

I int + (int) + (int)$ shift

int + (int) + (int)
↑

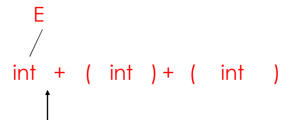
Shift-Reduce 例

I int + (int) + (int)$ shift
int I + (int) + (int)$ red. E → int

int + (int) + (int)
↑

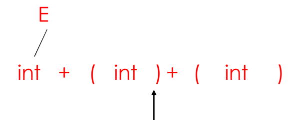
Shift-Reduce 例

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



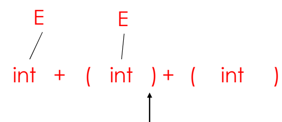
Shift-Reduce 例

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



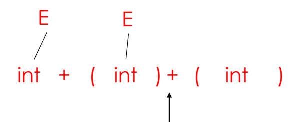
Shift-Reduce 例

| 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



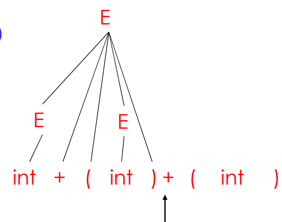
Shift-Reduce 例

| 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)



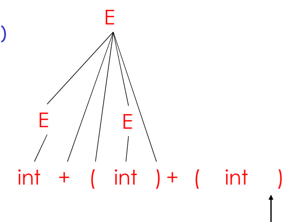
Shift-Reduce 例

| 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



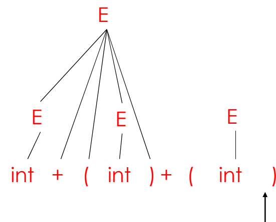
Shift-Reduce 例

| 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



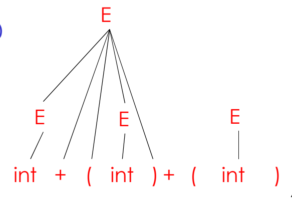
Shift-Reduce 例

- | 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



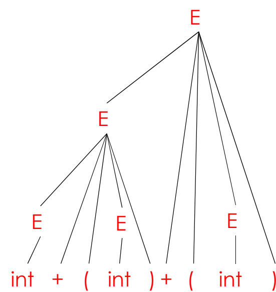
Shift-Reduce 例

- | 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

- | 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

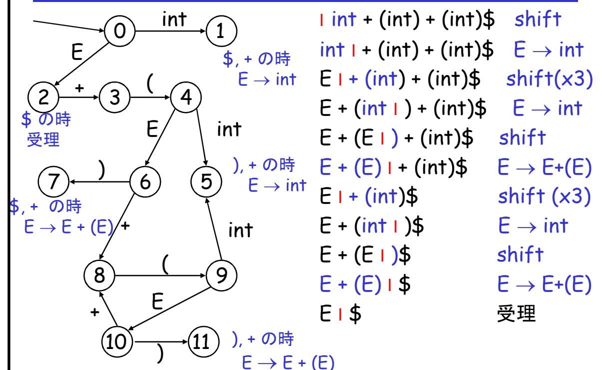
- 左部分列はスタックとして実装できる
  - スタックのトップは先ほどの |
- Shift は1終端記号をスタックに積むこと
- Reduce は 0 個以上の記号（生成規則の右辺）をスタックからポップし1個の非終端記号（生成規則の左辺）をスタックに積む

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
  - これでできる範囲（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 → β ” の遷移があるなら reduce

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。入力文字で継続し、到った終了状態で、次の動作を指定する。



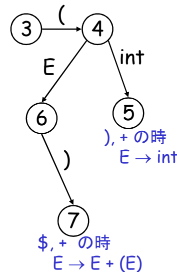
DFA　の表現

- 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、列を2つに分割:
  - 終端記号の列: action 表
  - 非終端記号の列: goto 表

DFA の表現．例

単なる状態遷移が、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」のではない状態を想定しているから。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 しなければならない。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」ときには、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。

- 例に使用している DFA の一部:



	int	+	(	)	$	E
…						
3			s4			
4	s5					g6
5		$r_{E \to int}$		$r_{E \to int}$		
6		s8		s7		
7		$r_{E \to E+(E)}$			$r_{E \to E+(E)}$	
…						

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

  $\langle sym_1, state_1 \rangle \ldots \langle sym_n, state_n \rangle$

  $state_k$ は、DFAを $sym_1 \ldots sym_k$ に適用した結果の状態

LR 構文解析アルゴリズム

```
I = w$ を（初期）入力列
j = 0
DFA state 0 を開始状態
stack = 〈 dummy, 0 〉
   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
```


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?

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

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

$\alpha \mid \gamma$

  - 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: $\mid x_1 x_2 \ldots x_n$

Shift 動作と Reduce 動作 (復習)

- CFGの例: E → int | 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 + (| int) ⇒ E + (int |)

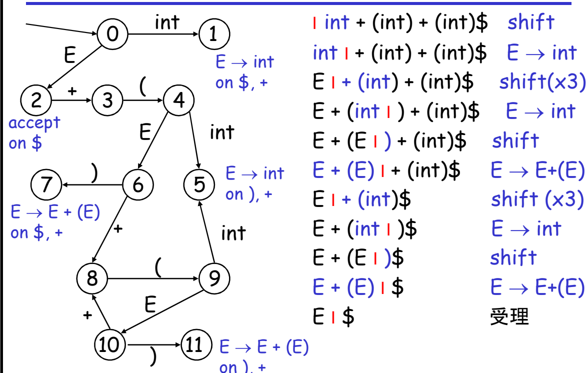
- Reduce は 0 個以上の記号(生成規則の右辺)をスタックからポップし1個の非終端記号(生成規則の左辺)をスタックに積む

  E + (E + (E) |) ⇒ E +(E |)

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
  - これでできる範囲(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 → β" の遷移があるなら reduce

LR(1) 構文解析. 例



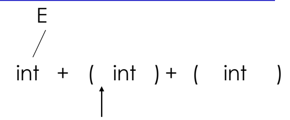
復習終了

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)

解析中の文脈(context)とは

- 途中状態の例:



  - この時スタックは E + (| int) + (int)
- 文脈:
  - 今探している規則は E → E + (• 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 → • int または E → • 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

LR(1) アイテム

- LR(1) アイテム は次の形をしたペア:

  X → α • β, a
  - X ! αβ 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 → α • β, 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

注

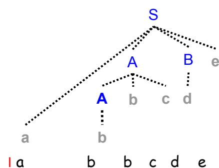
- 記号 | は、もともと、スタック(左部分列)と入力列中未処理の部分との境目として使用した
  - α | γ, ただし α はスタックであり γ は残りの終端記号列
  - これは、処理する文字列からみた、パーサの状態
  - パーサの心積もりからいうと、探索途中の生成規則とそのどこまで見つけたかという場所の集まり(複数個ある)。これを表すのが、LR(1) アイテムの左側である
- LR(1) アイテムにおいて、• は生成規則右辺の(ありうる)先頭部分 prefix をマークするのに使用:

  X → α • β, 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)

注: 文脈

生成規則
S → aABe
A → Abc
A → b
B → d

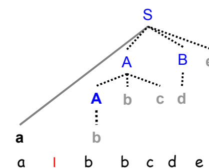
文脈
S → •aABe



注: 文脈

生成規則
S → aABe
A → Abc
A → b
B → d

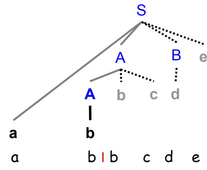
文脈
S → a•ABe
A → •Abc
A → •b



注：文脈

生成規則
S → aABe
A → Abc
A → b
B → d

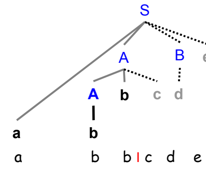
文脈
S → a•ABe
A → •Abc



注：文脈

生成規則
S → aABe
A → Abc
A → b
B → d

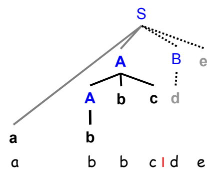
文脈
S → a•ABe
A → Ab•c



注：文脈

生成規則
S → aABe
A → Abc
A → b
B → d

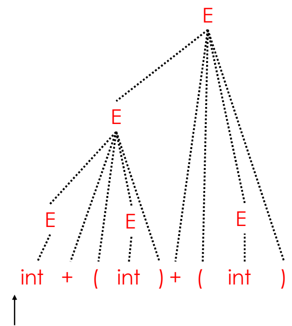
文脈
S → aA•Be
B → •d



文脈（必要な先読みをサポっている）

E → int
E → E + (E)

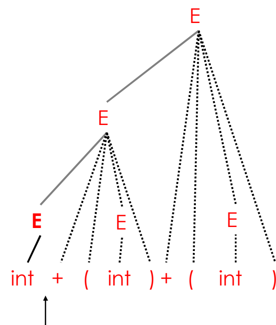
E → •int
E → •E + (E)



文脈（必要な先読みをサポっている）

E → int
E → E + (E)

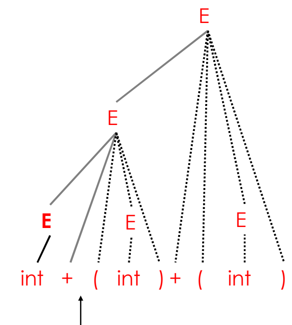
E → • E + (E)
E → E • + (E)



文脈（必要な先読みをサポっている）

E → int
E → E + (E)

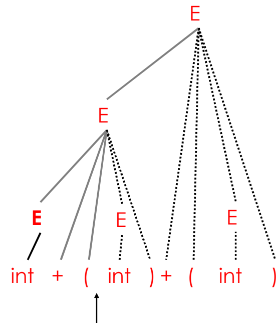
E → • E + (E)
E → E • + (E)



文脈（必要な先読みをサボっている）

E → int
E → E + (E)

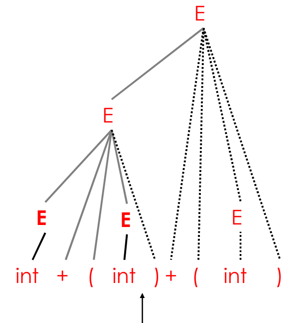
E → • E + (E)
E → E • + (E)
E → • int
E → • E + (E)



文脈（必要な先読みをサボっている）

E → int
E → E + (E)

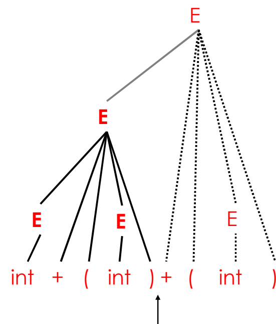
E → • E + (E)
E → E + (E •)



文脈（必要な先読みをサボっている）

E → int
E → E + (E)

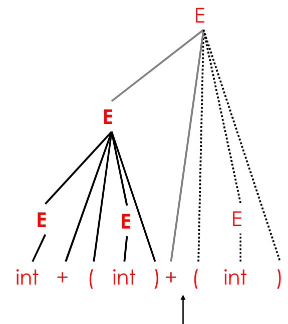
E → •E + (E)
E → E • + (E)



文脈（必要な先読みをサボっている）

E → int
E → E + (E)

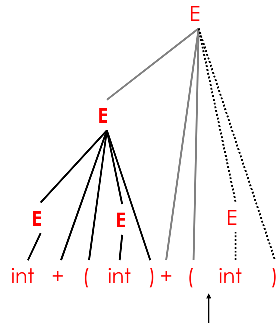
E → •E + (E)
E → E + • (E)



文脈（必要な先読みをサボっている）

E → int
E → E + (E)

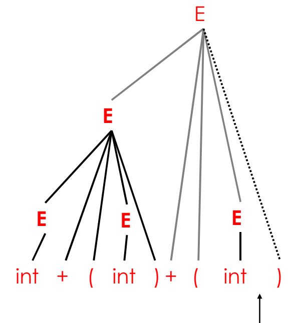
E → •E + (E)
E → E + (• E)
E → •int
E → •E + (E)



文脈（必要な先読みをサボっている）

E → int
E → E + (E)

E → •E + (E)
E → E + (E •)



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 ! 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 → • 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

LR(1) アイテム (続)

- この文脈

  E → E + • (E), +

  では、もし (が続くなら shift を行ない、次の文脈になる

  E → E + (• E), +
- この文脈

  E → E + (E) •, +

  では、reduce が、規則 E → 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

LR(1) アイテム (続)

- このアイテムの表す文脈で考えよう

  E → E + (• 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 → int と E → 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 → • int,)

  E → • E + (E) ,)

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 → α • Yβ, a]
      for 各生成規則 Y → γ
        for 各 b ∊ First(βa) /* First(βa) は空ではない */
          [Y → •γ, b] を Items に追加
  until Items が変わらない

構文解析 DFA の作成 (1)

- 開始文脈の作成: Closure({S → • E, $})

  S → • E, $
  E → • E+(E), $
  E → • int, $
  E → • E+(E), +
  E → • int, +
- 次のように簡略標記する:

  S → • E, $
  E → • E+(E), $/+
  E → • int, $/+

構文解析 DFA の作成 (2)

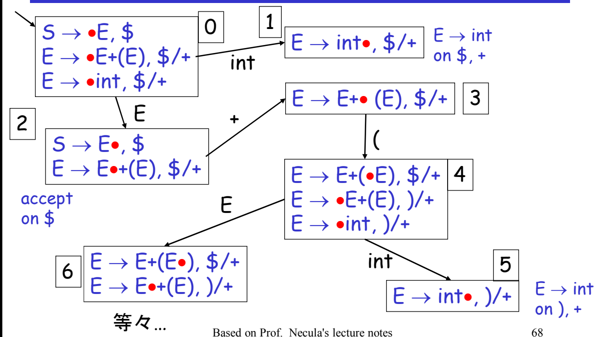
- DFA 状態は, LR(1) アイテムの"閉じた"集合
  - すなわち、閉包 Closure を行なう
    (因みに、閉じた= closed)
- 開始状態に含まれる: [S → • E, $]
- [X → α •, b] をもつ状態につけるラベル: "bがあるとき, X → α を用いて還元"
- 状態遷移は ...

DFA 状態遷移

- "State" という状態に $[X \to a \bullet yb, b]$ があるとする. 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 ← ∅
  for 各 $[X \to \alpha \bullet y\beta, b] \in$ State
    $[X \to \alpha y \bullet \beta, b]$ を Items に追加
  return Closure(Items)

構文解析 DFA の作成. 例



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ったりする
- どんなエラーがありうるか?

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
  $[X \to \alpha \bullet a\beta, b]$ と $[Y \to \gamma \bullet, a]$
- 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 $[X \to \alpha a \bullet \beta, b]$ になる, またはを
  - Reduce を $Y \to \gamma$ で行なう
- これを, シフト還元衝突 shift-reduce conflict と呼ぶ

Shift/Reduce Conflicts

- よくあるのは、文法の曖昧性
- 古典的な例: the dangling else
  S → if E then S | if E then S else S | OTHER
- このとき、DFA 状態で次のものをもつものがある
  [S → if E then S•, else]
  [S → if E then S• 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

もう一つ Shift/Reduce Conflicts

- 次の曖昧な文法を考えよう
  E → E + E | E * E | int
- 次のものをもつ状態がある
  [E → E * • E, +]   [E → E * E •, +]
  [E → • E + E, +] $\Rightarrow^E$ [E → E • + E, +]
  …   …
- 入力に + があると, shift/reduce がありうる
  - 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

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

優先順位を用いて S/R Conflicts をとく

- 例の例にもどろう:

[E → E * • E, +] [E →E * E •, +]
[E → • E + E, +] ⇒E [E →E • + E, +]
… …

- ここでは reduce を選ぶ. なぜなら, 優先順位は、規則 E → E * E の方が終端記号 + より高い

優先順位を用いて S/R Conflicts をとく

- 例によって例の文法

E → E + E | E * E | int

- 次の状態もある

[E → E + • E, +] [E → E + E •, +]
[E → • E + E, +] ⇒E [E → E • + E, +]
… …

- 入力 + に対して, shift/reduce ともにありうる
  - reduce を選ぶ. というのも E → 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

優先順位を用いて S/R Conflicts をとく

- dangling else 問題に戻ろう

[S → if E then S •, else]
[S → if E then S • 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

Reduce/Reduce Conflicts

- かりに DFA 状態に次の二つがあったとする

[X → α•, a] and [Y → β•, 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がわからなくなる
- これは reduce/reduce conflict と呼ばれる

Reduce/Reduce Conflicts

- 通常は、文法の曖昧性による
- 例: 識別子の列

S → ε | id | id S

- id に対して、2個の構文木がある

S → id
S → id S → id

- パーサはどんな混乱をきたすのだろうか?

さらに Reduce/Reduce Conflicts

- 次の状態を考えよう
  - [S' → • S, $]
  - [S → •, $]
  - [S → • id, $]
  - [S → • id S, $]

  $\Rightarrow^{id}$

  - [S → id •, $]
  - [S → id • S, $]
  - [S → •, $]
  - [S → • id, $]
  - [S → • id S, $]
- 入力 $ が現れると reduce/reduce conflict
  - S' → S → id
  - S' → S → id S → id
- 文法を書き換えるべき: S → ε | id S

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

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.

  [1] E → int•, $/+ (E → int on $, +) と [5] E → int•,)/+ (E → 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

  [1'] E → int•, $/+/) (E → int on $, +,))

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 → α•β, b], [Y → γ•δ, d]}

  コアはこれ

  {X → α•β, Y → γ•δ}

LALR 状態

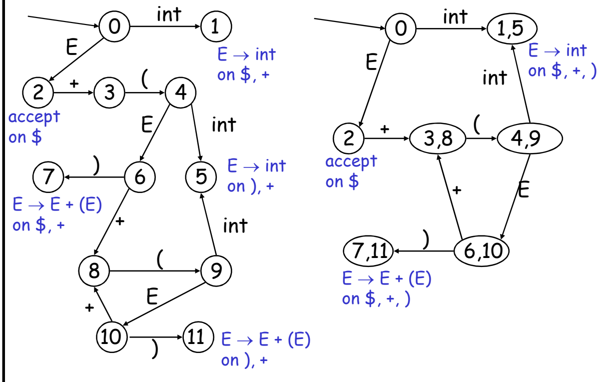
- 例えば、次の LR(1) 状態を考える
  - {[X → α•, a], [Y → β•, c]}
  - {[X → α•, b], [Y → β•, d]}
- 同じコアを持つのでマージできる
- マージした状態には次のものがある:
  - {[X → α•, a/b], [Y → β•, c/d]}
- これらは LALR(1) 状態と呼ばれる
  - もととなった語: LookAhead LR
  - たいてい, LALR(1) の状態数は、LR(1) の 10分の1

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

A → B → C
D → E → F

LR(1) から LALR(1) への変換. 例.



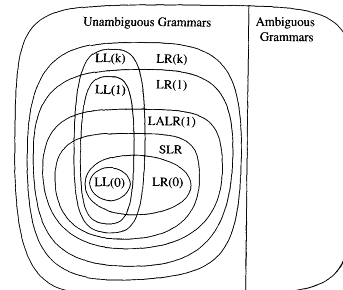
LALR パーサの conflicts

- 例えば, 次の LR(1) 状態を考える
  - {[X → α•, a], [Y → β•, b]}
  - {[X → α•, b], [Y → β•, a]}
- マージした LALR(1) 状態は
  - {[X → α•, a/b], [Y → β•, 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

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

文法クラスの階層



Andrew Appel,
"Modern Compiler Implementation in Java"

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)
  - より協力なパーサ: 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

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)

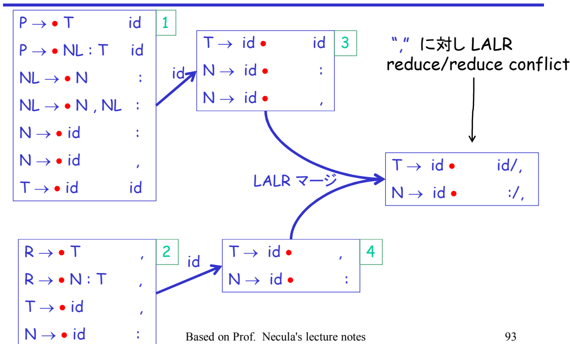
風変わりな Strange Reduce/Reduce Conflicts

- 次の文法を考える
  S → P R ,　　NL → N | N , NL
  P → T | NL : T　　R → T | N : T
  N → id　　T → id
- 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

LALR(1) Reduce/Reduce Conflicts

- P において id は
  - N , もし , か : 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

LR(1) 状態の一部



何が起こったか?

- 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 → id 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

修理後の LR(1) 状態