実験3SW 座学2

馬谷 誠二

2016/04/14

内容

1. 疑似クォート

2. LR構文解析

3. よい抽象構文木

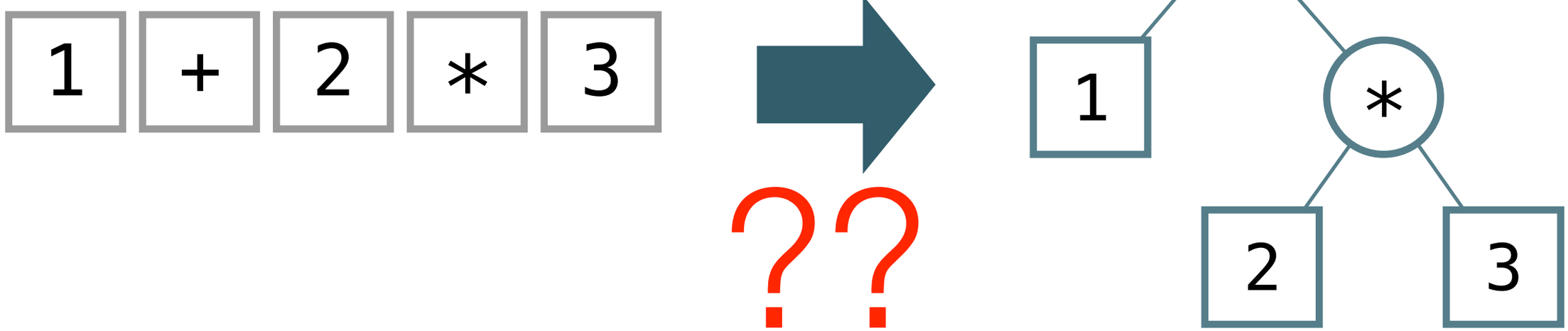
4. gen.rkt に関する注意

疑似クォート（`）

- データ・データ構造をつくるコードが読みやすくなる
  - 一種のテンプレート
    - Tiny Cソースコードやデータフロー解析用配布コードでふんだんに使用
- くわしくは実験資料の付録Aを参照

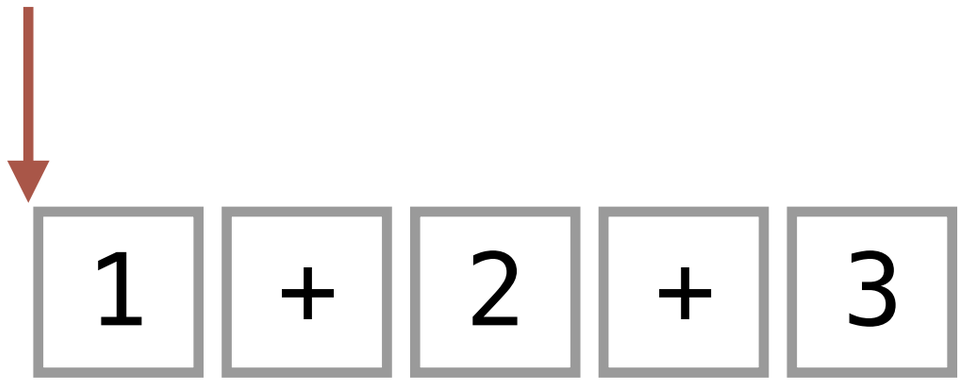
LR構文解析

```
A ::= M          { $1 }
    | A + M      { `(+ ,$1 ,$3) }
M ::= N          { $1 }
    | M * N      { `(* ,$1 ,$3) }
```



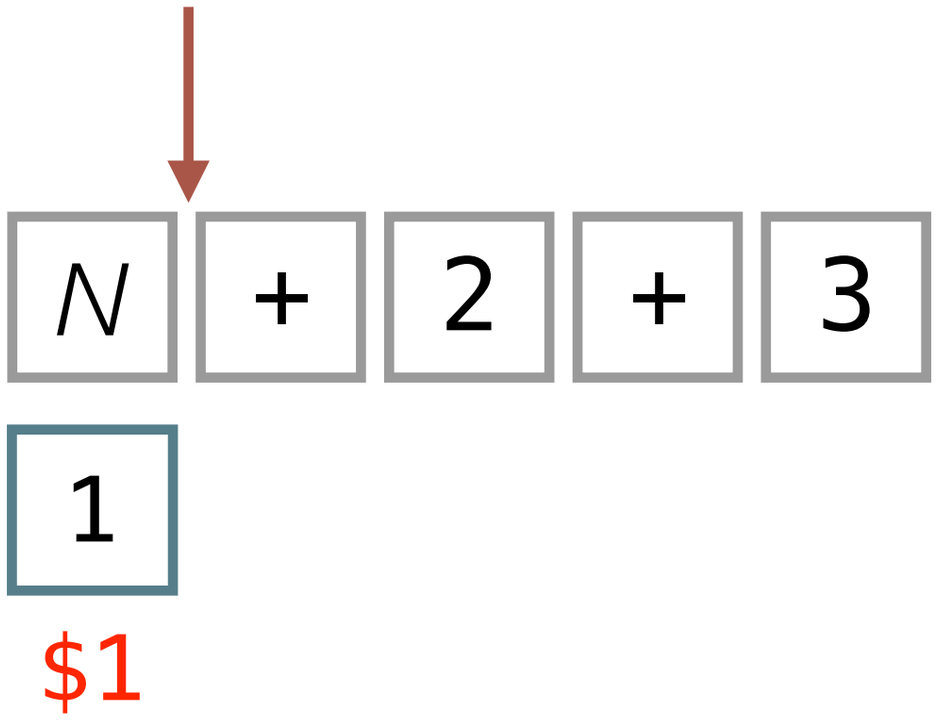
復習：LR(0)構文解析

```
A ::= N          { $1 }
    | A + N      { `(+ ,$1 ,$3) }
```



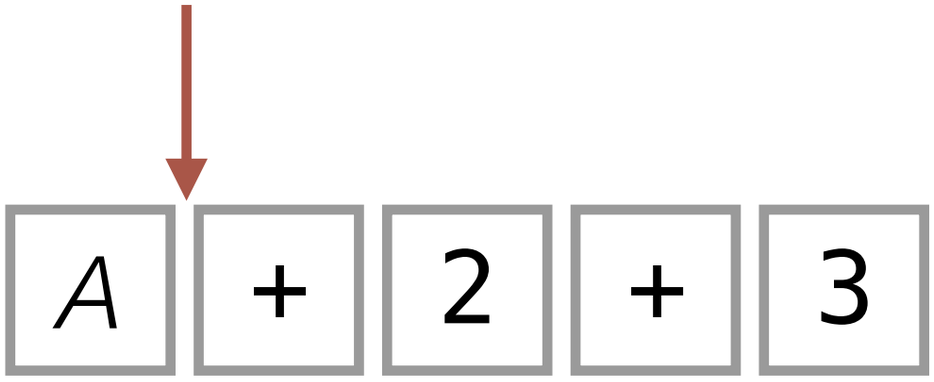
復習：LR(0)構文解析

```
A ::= N              { $1 }
    | A + N          { `(+ ,$1 ,$3) }
```



復習：LR(0)構文解析

```
A ::= N          { $1 }
    | A + N      { `(+ ,$1 ,$3) }
```

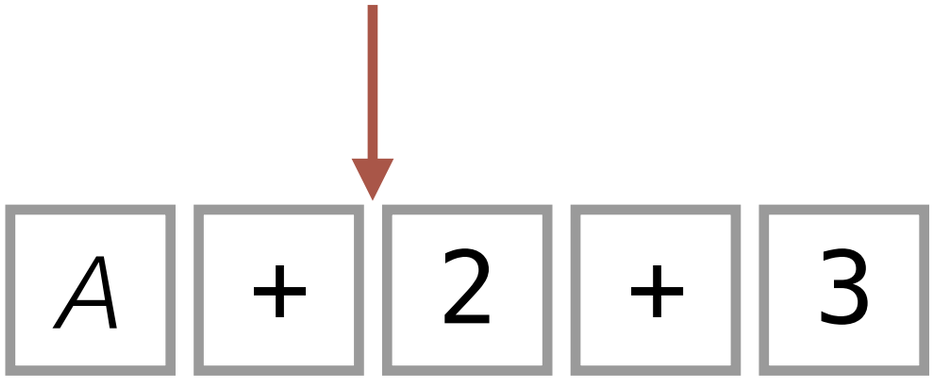


T

最後の*A*の抽象構文木*T*

復習：LR(0)構文解析

```
A ::= N          { $1 }
    | A + N      { `(+ ,$1 ,$3) }
```

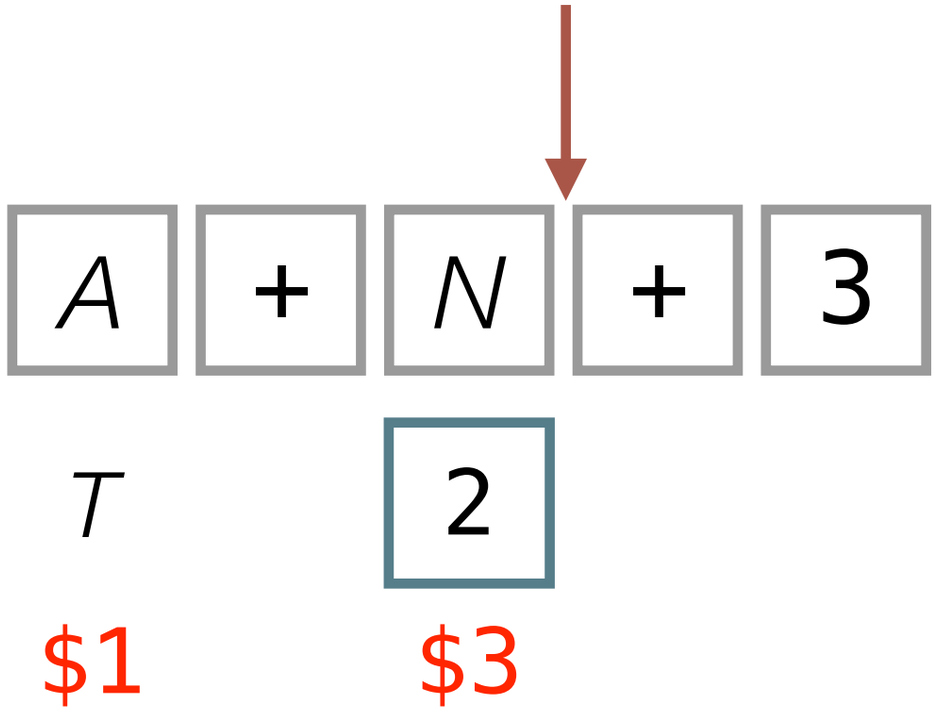


T

最後の*A*の抽象構文木*T*

復習：LR(0)構文解析

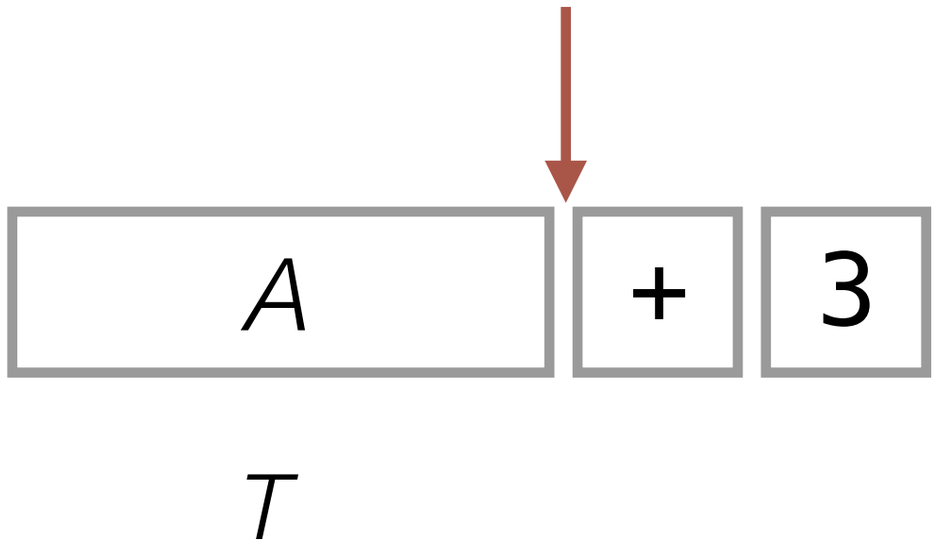
```
A ::= N          { $1 }
    | A + N      { `(+ ,$1 ,$3) }
```



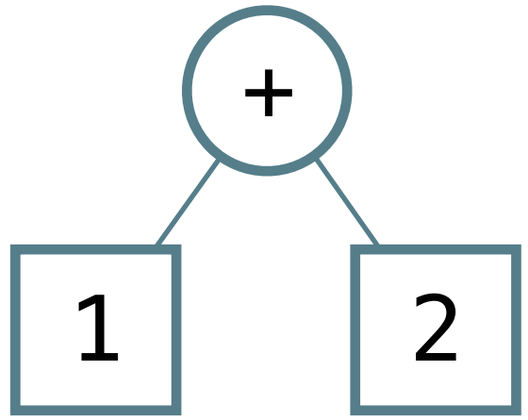
最後のAの抽象構文木T

復習：LR(0)構文解析

```
A ::= N          { $1 }
    | A + N      { `(+ ,$1 ,$3) }
```

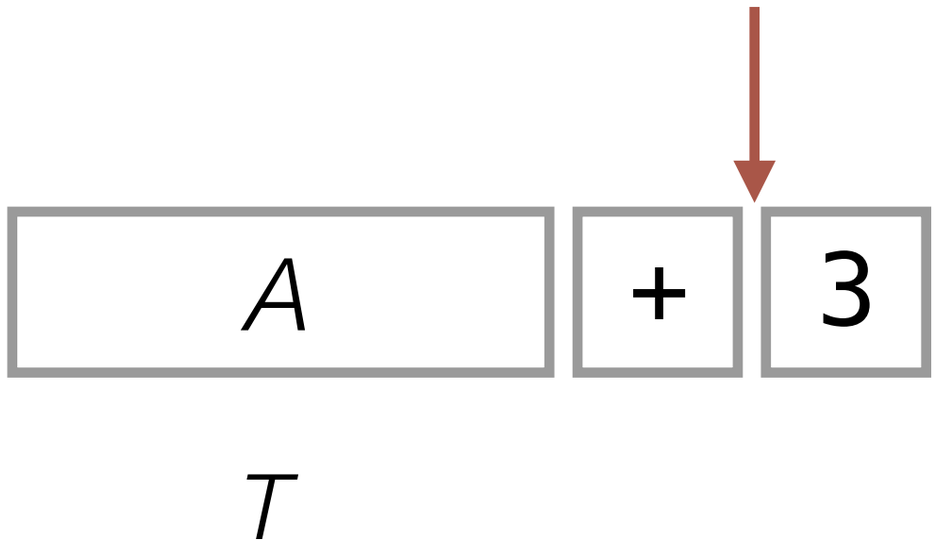


最後のAの抽象構文木T

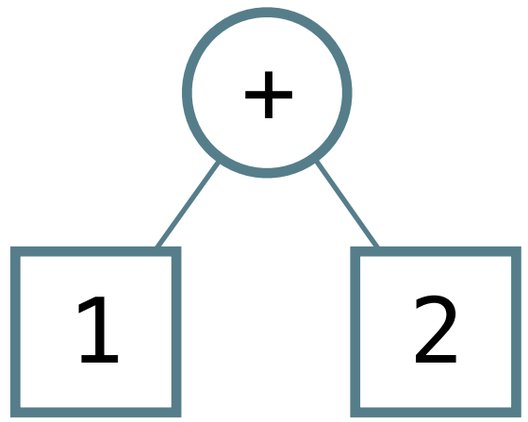


復習：LR(0)構文解析

```
A ::= N          { $1 }
    | A + N      { `(+ ,$1 ,$3) }
```

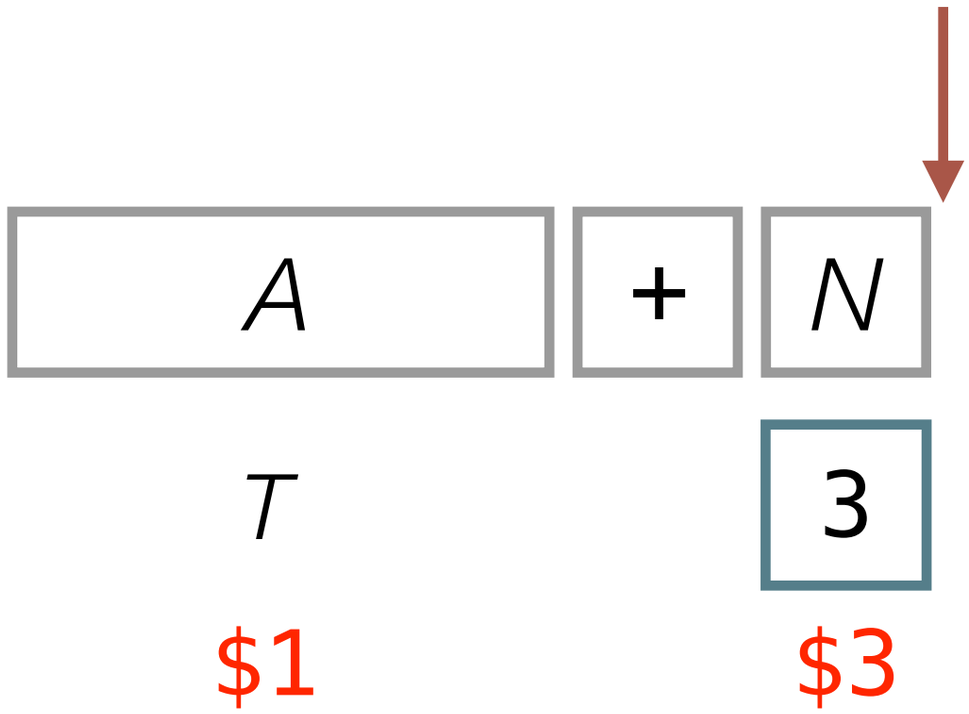


最後のAの抽象構文木T

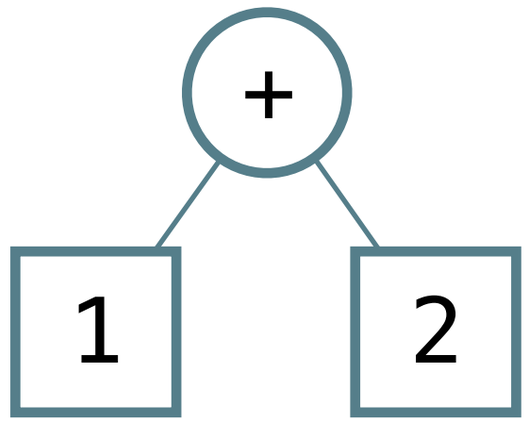


復習：LR(0)構文解析

```
A ::= N              { $1 }
    | A + N          { `(+ ,$1 ,$3) }
```

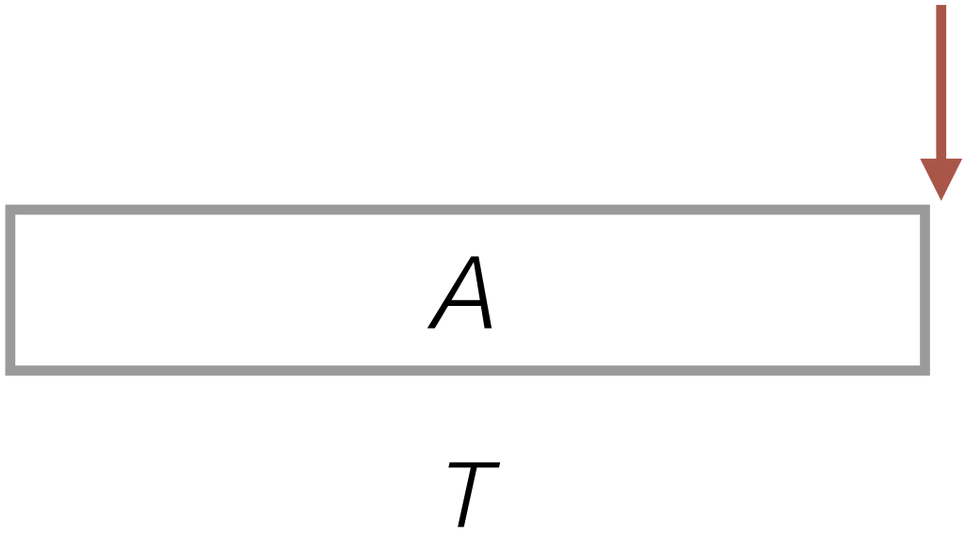


最後のAの抽象構文木T

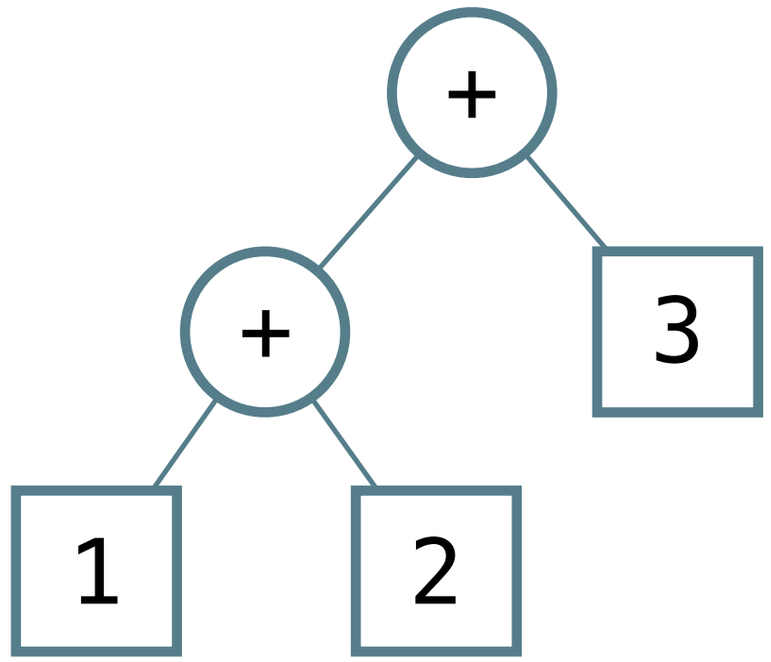


復習：LR(0)構文解析

```
A ::= N          { $1 }
    | A + N      { `(+ ,$1 ,$3) }
```

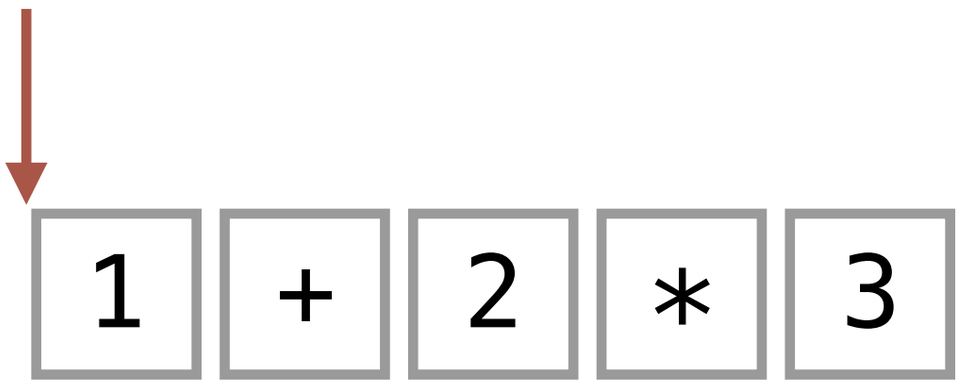


最後のAの抽象構文木T



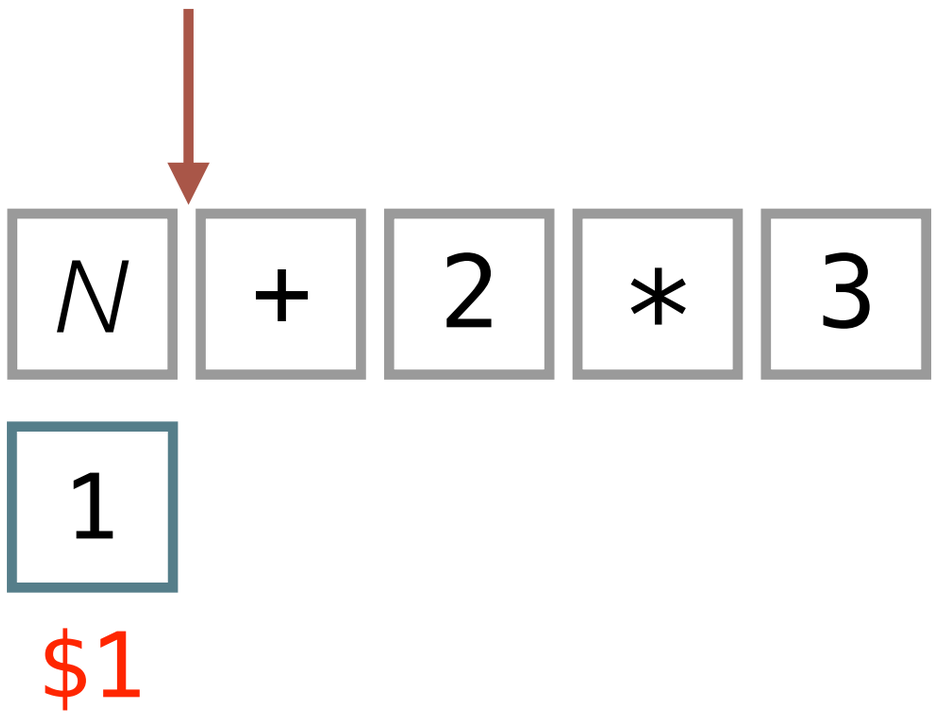
LR(0)構文解析による試行

```
A ::= M          { $1 }
    | A + M      { `(+ ,$1 ,$3) }
M ::= N          { $1 }
    | M * N      { `(* ,$1 ,$3) }
```



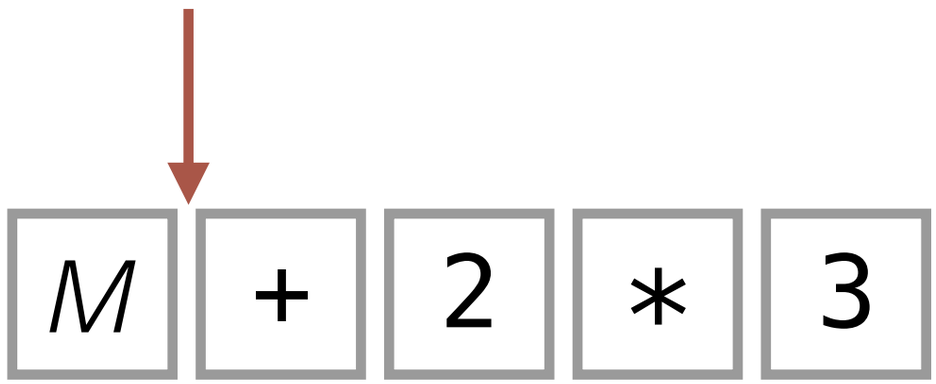
LR(0)構文解析による試行

```
A ::= M          { $1 }
    | A + M      { `(+ ,$1 ,$3) }
M ::= N          { $1 }
    | M * N      { `(* ,$1 ,$3) }
```



LR(0)構文解析による試行

```
A ::= M          { $1 }
    | A + M      { `(+ ,$1 ,$3) }
M ::= N          { $1 }
    | M * N      { `(* ,$1 ,$3) }
```



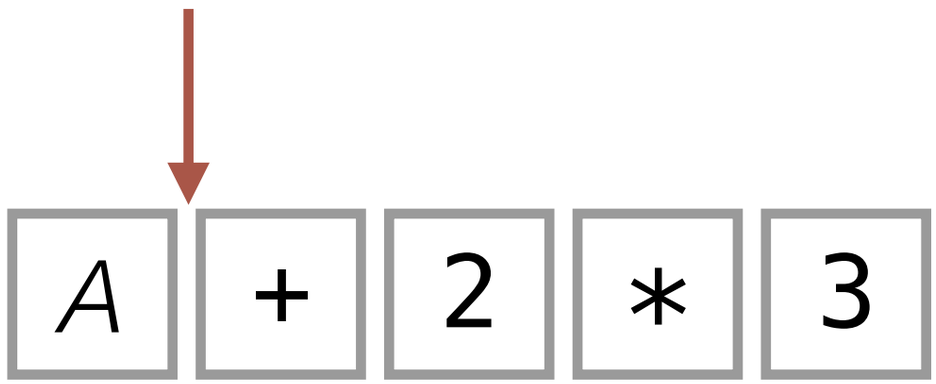
T

$1

最後の*M*の抽象構文木*T*

LR(0)構文解析による試行

```
A ::= M          { $1 }
    | A + M      { `(+ ,$1 ,$3) }
M ::= N          { $1 }
    | M * N      { `(* ,$1 ,$3) }
```



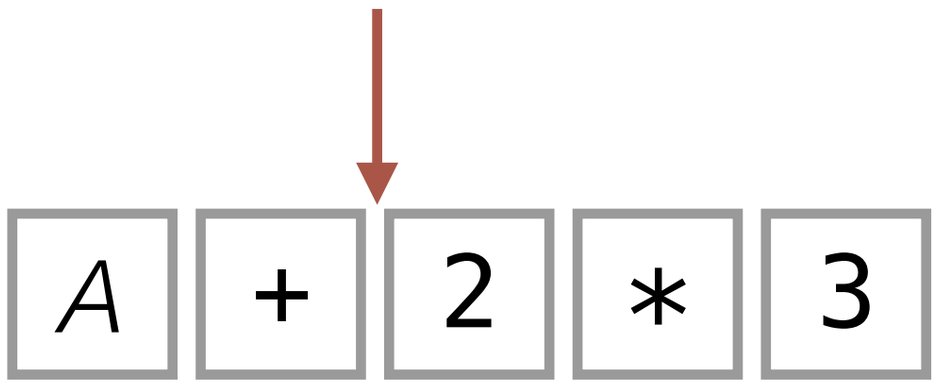
S

最後のAの抽象構文木S

最後のMの抽象構文木T

LR(0)構文解析による試行

```
A ::= M          { $1 }
    | A + M      { `(+ ,$1 ,$3) }
M ::= N          { $1 }
    | M * N      { `(* ,$1 ,$3) }
```



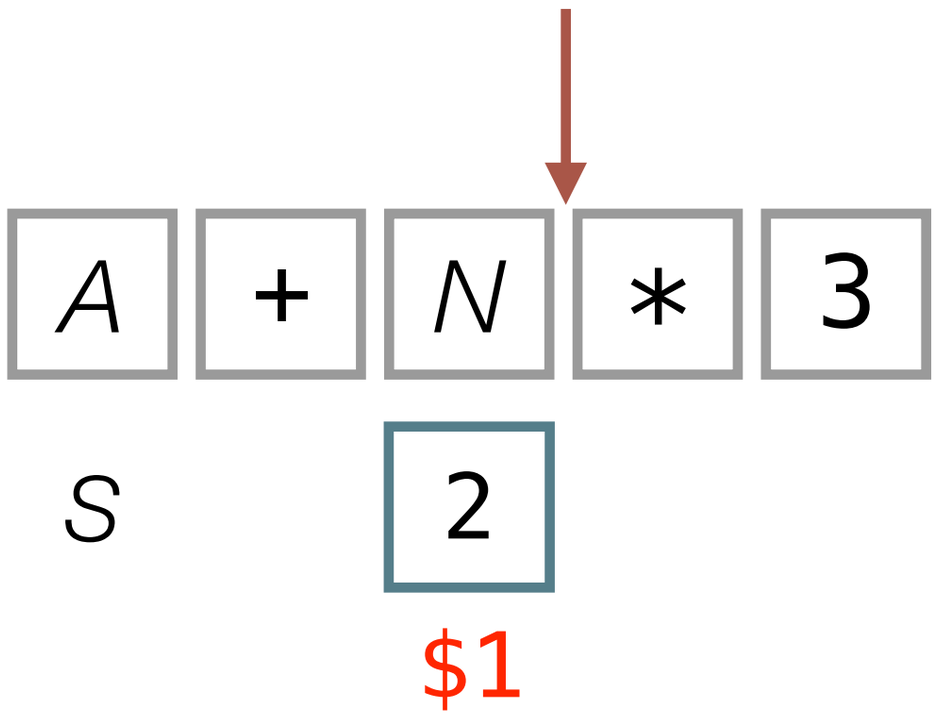
S

最後のAの抽象構文木S

最後のMの抽象構文木T

LR(0)構文解析による試行

```
A ::= M          { $1 }
    | A + M      { `(+ ,$1 ,$3) }
M ::= N          { $1 }
    | M * N      { `(* ,$1 ,$3) }
```

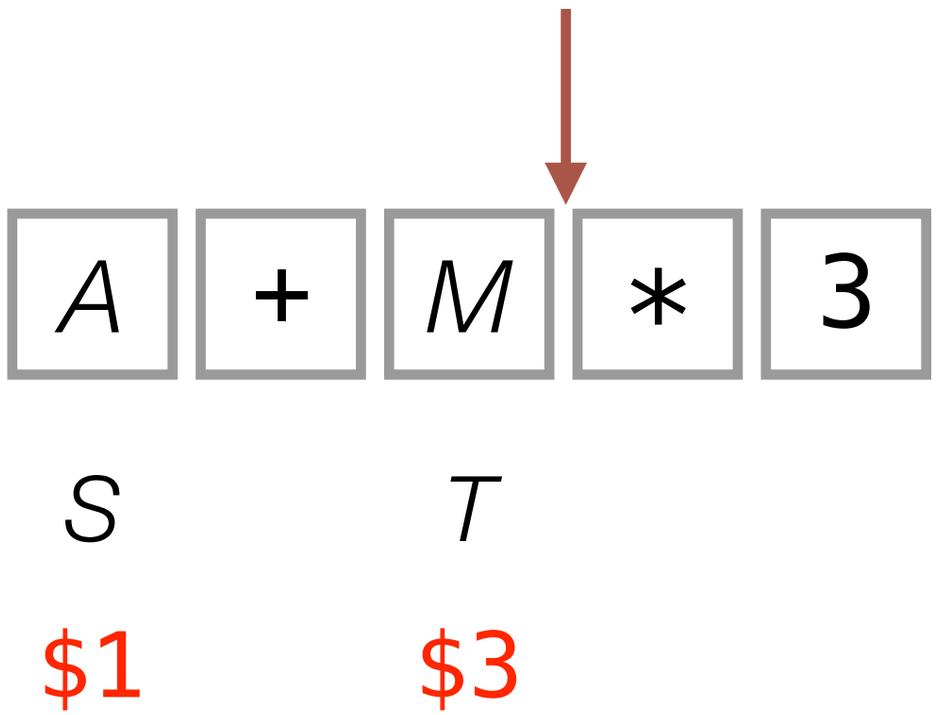


最後のAの抽象構文木S

最後のMの抽象構文木T

LR(0)構文解析による試行

```
A ::= M          { $1 }
    | A + M      { `(+ ,$1 ,$3) }
M ::= N          { $1 }
    | M * N      { `(* ,$1 ,$3) }
```



最後のAの抽象構文木S

1

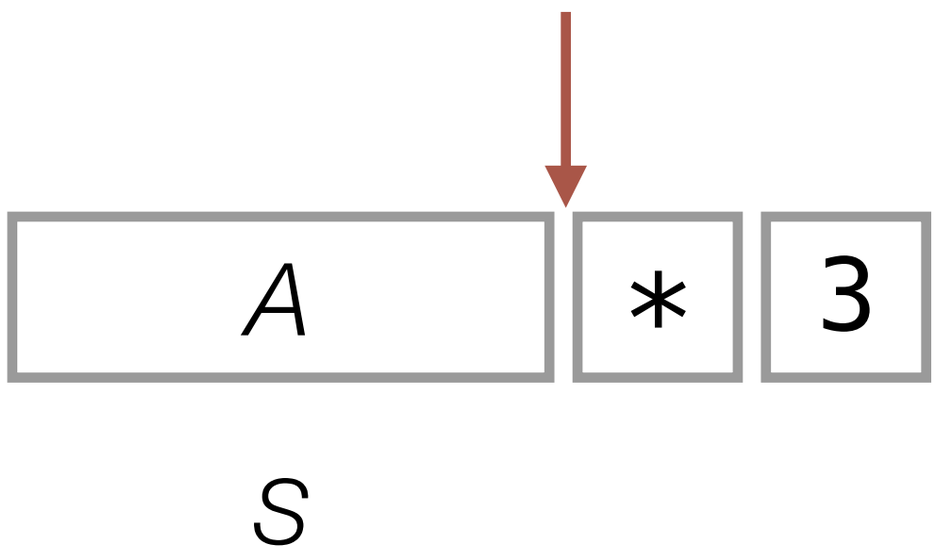
最後のMの抽象構文木T

2

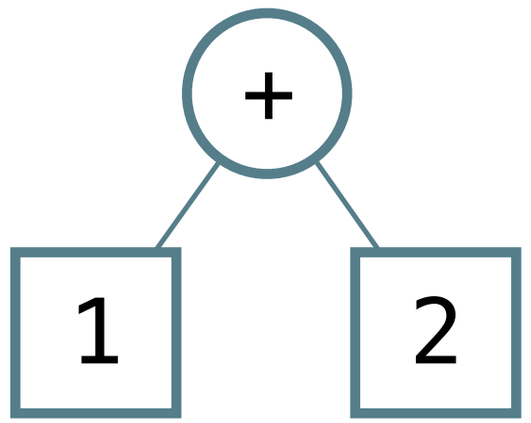
LR(0)構文解析による試行

$A*$を含むような
規則はどこにもない！

```
A ::= M          { $1 }
    | A + M      { `(+ ,$1 ,$3) }
M ::= N          { $1 }
    | M * N      { `(* ,$1 ,$3) }
```



最後のAの抽象構文木S



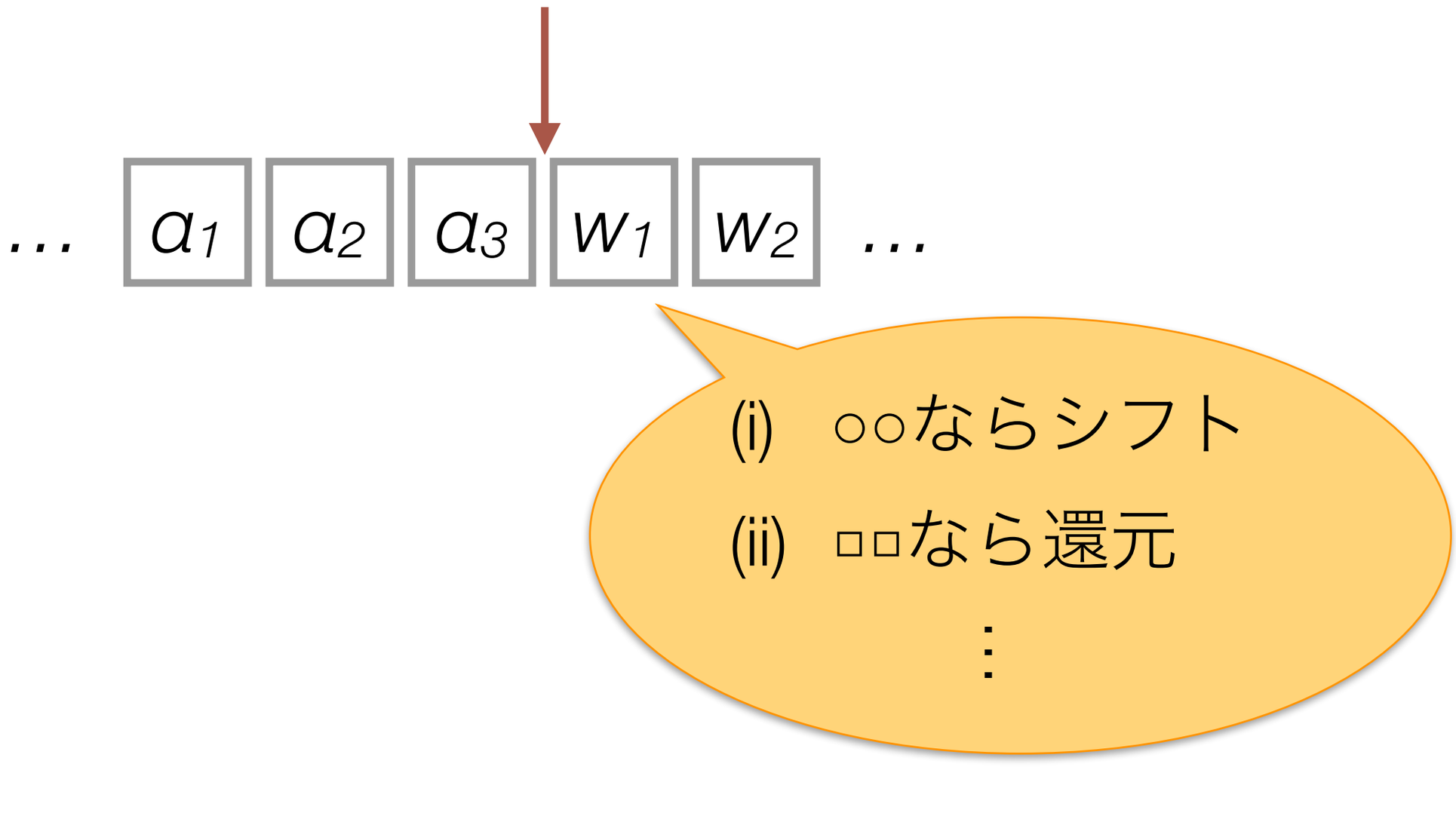
LR(0)構文解析による試行

```
A ::= M          { $1 }
    | A + M      { `(+ ,$1 ,$3) }
M ::= N          { $1 }
    | M * N      { `(* ,$1 ,$3) }
```

これに気づければ立派！

LALR(1)構文解析

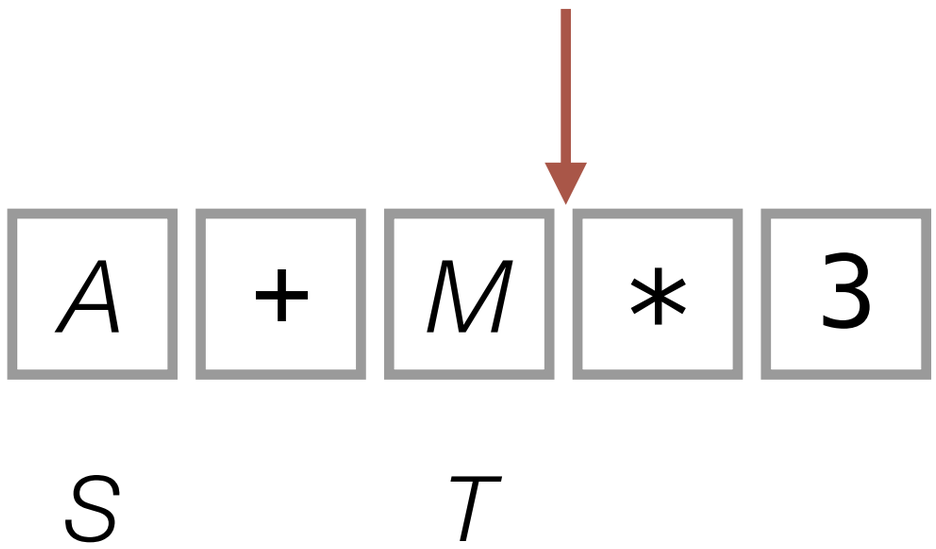
- yaccが実際に基づいている手法
- 1個先読みしたトークンに従ってどうするかを決める



先程の続き

(i) ∗ならシフト
(ii) +, $なら還元
(iii) Nならエラー

```
A ::= M          { $1 }
    | A + M      { `(+ ,$1 ,$3) }
M ::= N          { $1 }
    | M ∗ N      { `(∗ ,$1 ,$3) }
```



最後のAの抽象構文木S

1

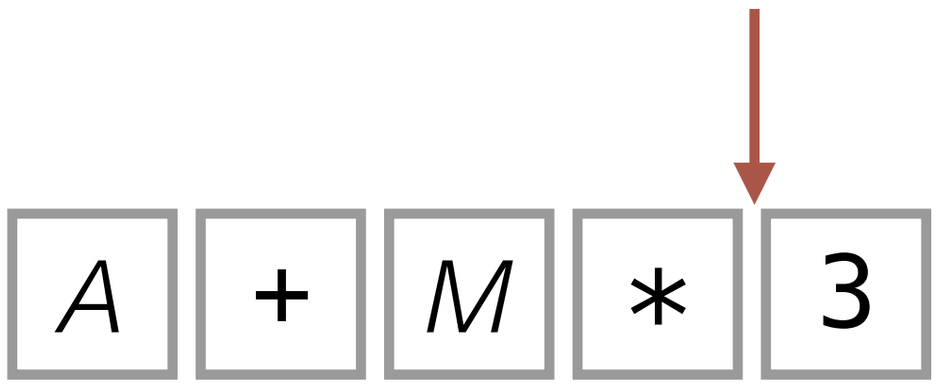
最後のMの抽象構文木T

2

先程の続き

(i) ∗ならシフト
(ii) +, $なら還元
(iii) Nならエラー

```
A ::= M          { $1 }
    | A + M      { `(+ ,$1 ,$3) }
M ::= N          { $1 }
    | M ∗ N      { `(∗ ,$1 ,$3) }
```



最後のAの抽象構文木S

1

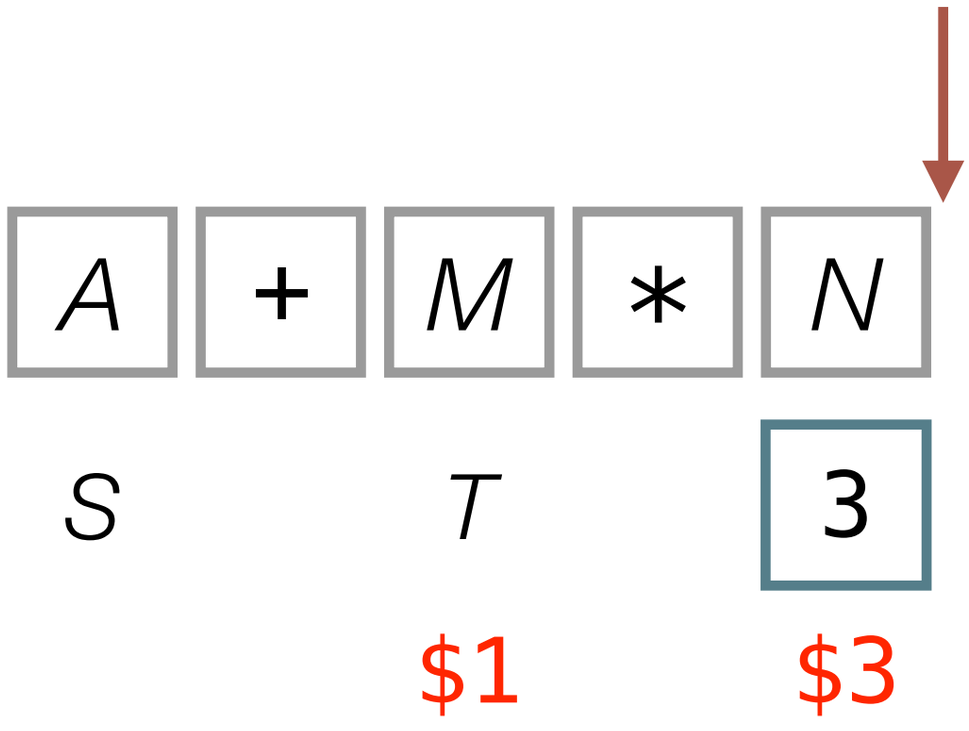
最後のMの抽象構文木T

2

先程の続き

(i) ∗ならシフト
(ii) +, $なら還元
(iii) Nならエラー

```
A ::= M          { $1 }
    | A + M      { `(+ ,$1 ,$3) }
M ::= N          { $1 }
    | M ∗ N      { `(∗ ,$1 ,$3) }
```



最後のAの抽象構文木S

1

最後のMの抽象構文木T

2

先程の続き

(i) ∗ならシフト
(ii) +, $なら還元
(iii) Nならエラー

```
A ::= M                { $1 }
    | A + M            { `(+ ,$1 ,$3) }
M ::= N                { $1 }
    | M ∗ N            { `(∗ ,$1 ,$3) }
```

最後のAの抽象構文木S

1

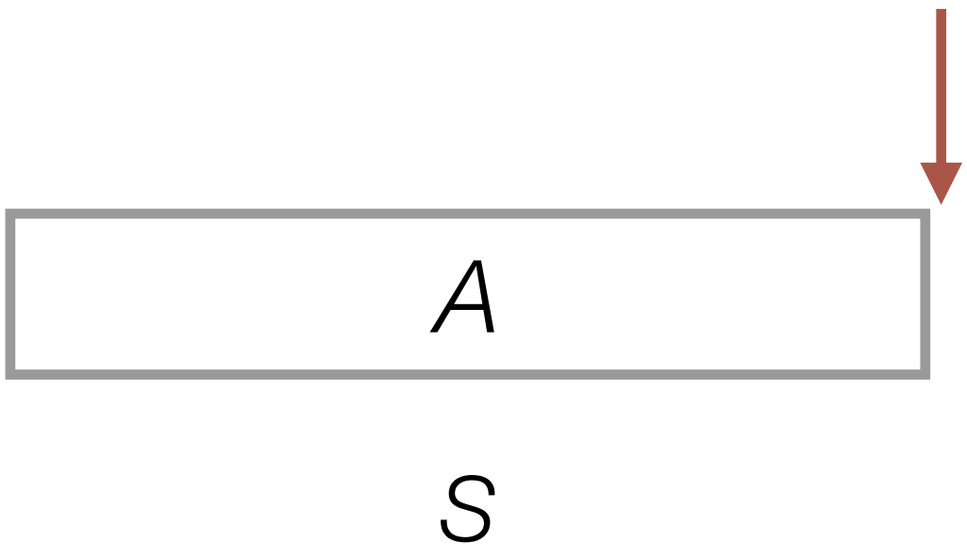
最後のMの抽象構文木T

(∗ 2 3)

先程の続き

(i) ∗ならシフト
(ii) +, $なら還元
(iii) Nならエラー

```
A ::= M          { $1 }
    | A + M      { `(+ ,$1 ,$3) }
M ::= N          { $1 }
    | M ∗ N      { `(∗ ,$1 ,$3) }
```



最後のAの抽象構文木S

(+ 1 (∗ 2 3))

最後のMの抽象構文木T

おまけ：Small Cの言語クラス ⊆ LALR(1)？

- 実は違う
  - 同じ先読みトークンに対し，シフトと還元が衝突
- parser.rktの

```
;;(debug "small-c-parser.tbl")
(suppress)
```

を

```
(debug "small-c-parser.tbl")
;;(suppress)
```

に書き換えて，生成されたファイルの中を「conflict」で検索すると要因が見つかります.

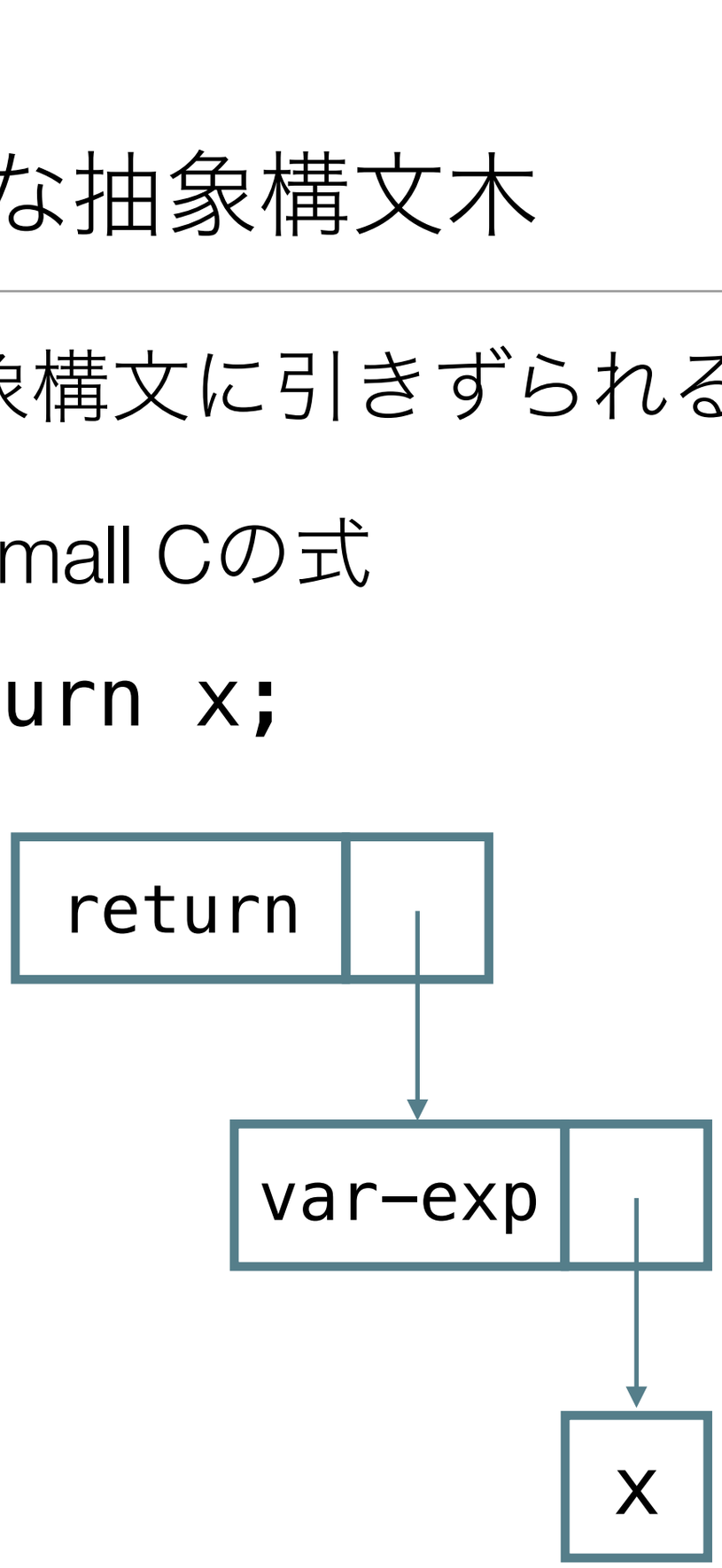
冗長な抽象構文木

- 具象構文に引きずられるとつくりがち

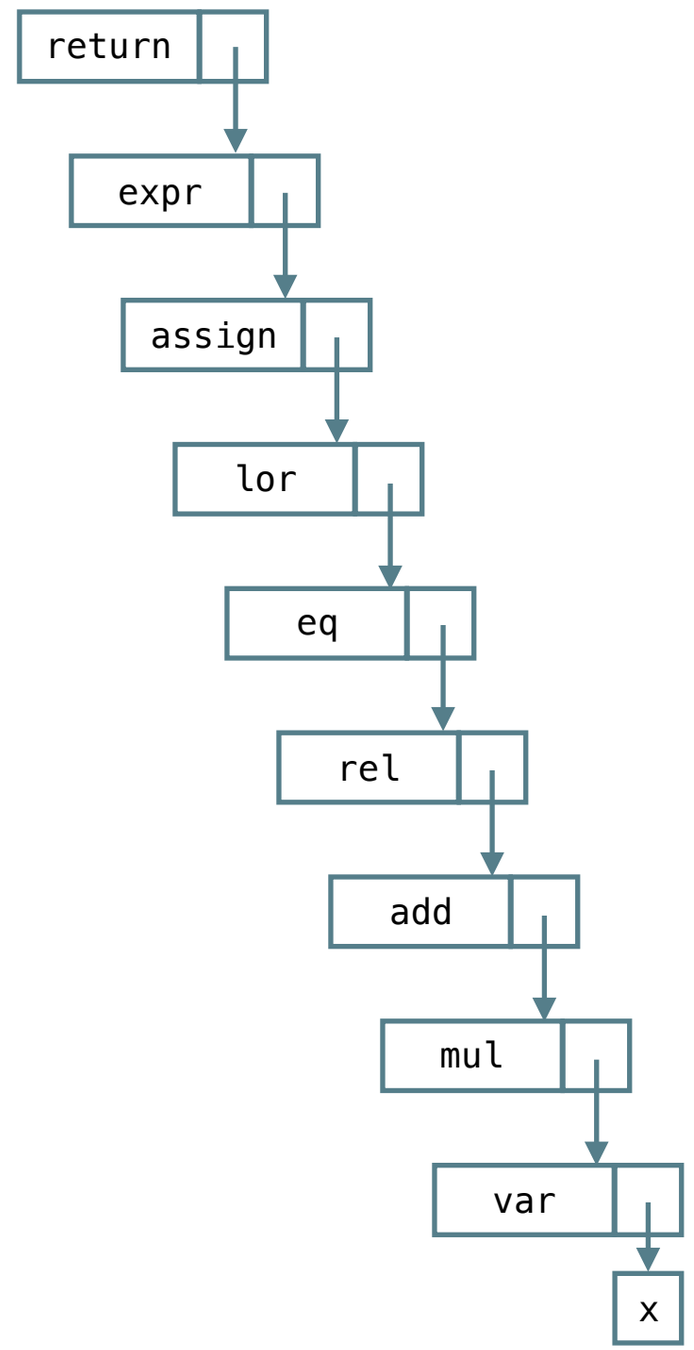
(例) Small Cの式

```
return x;
```

Good



Bad



よい抽象構文木

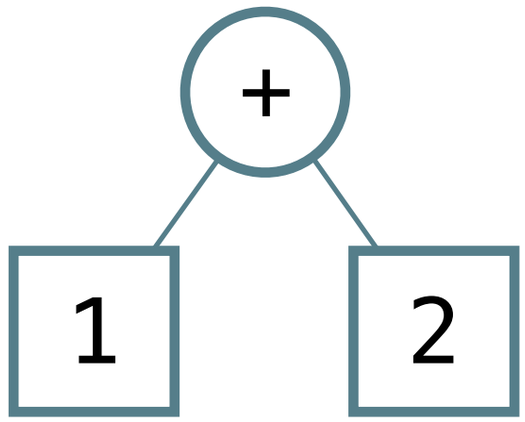
- つくるにはアクション中にプログラムが必要
  - 任意のRacket式を書けることを忘れていませんか？

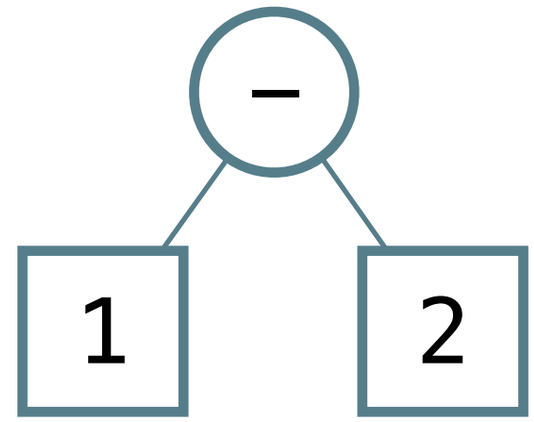
```
(grammar
 ...
 (statement ((ID = expression SEMI)
             (stx:assign-stmt $1 $3 $1-start-pos))
            ((* expression = expression SEMI)
             (stx:massign-stmt $2 $4 $1-start-pos))
            ...)
 ...
 (add-expr ((mult-expr) $1)
           ((add-expr + mult-expr)
            (stx:aop-exp '+ $1 $3 $2-start-pos))
           ((add-expr - mult-expr)
            (stx:aop-exp '- $1 $3 $2-start-pos)))
 ...)
```

例：加減算

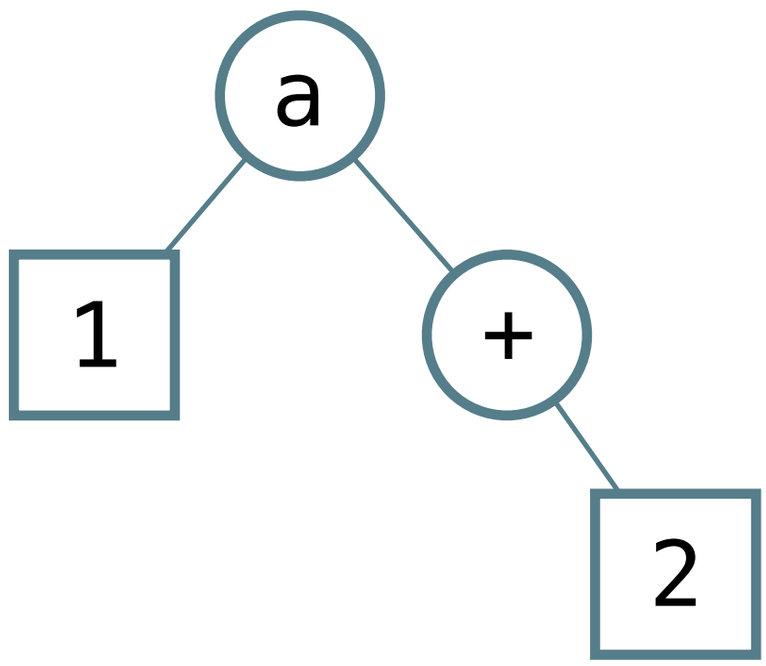
$$A ::= N \mid N\,R \qquad R ::= + N \mid - N$$

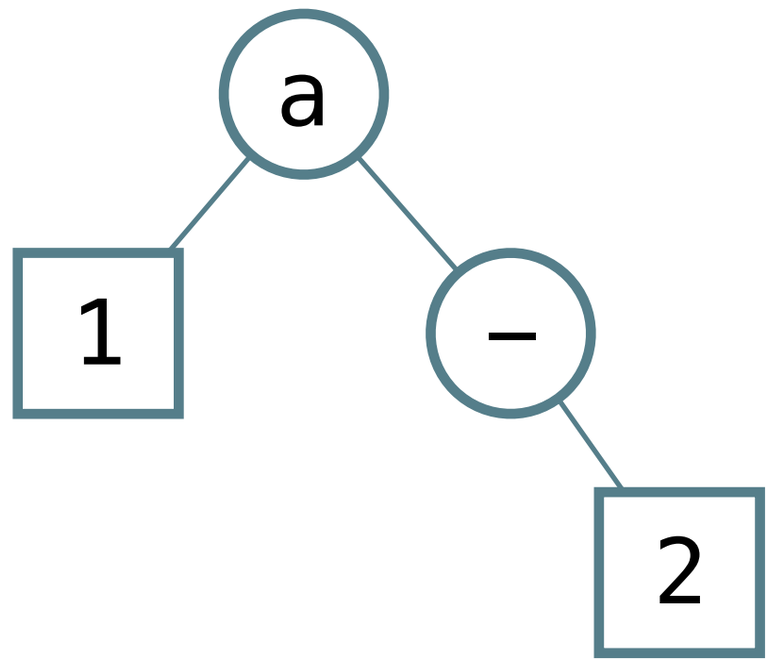
Good





Bad





解答

```
(struct aop-exp (op left right))
...
  (grammar
   (A ((N)    $1)
      ((N R) (let ((op    (car $2))
                   (right (cdr $2)))
               (aop-exp op $1 right))))
   (R ((+ N) (cons '+ $2))
      ((- N) (cons '- $2)))
   (N ((NUM) $1)))
...
```

関数プロトタイプ宣言（初回座学の最後にも同じ解説あり）

```
int *f(int a, int *b);
```

具象構文木

```
(function-prototype
  (type-specifier 'int)
  (function-declarator
    '* 'f
    (parameter-type-list
      (list
        (parameter-declaration
          (type-specifier 'int)
          (parameter-declarator 'a))
        (parameter-declaration
          (type-specifier 'int)
          (parameter-declarator '* 'b))))
    )))
```

よい抽象構文木

```
(function-prototype
  '(* int)
  'f
  (list
    (cons 'a 'int)
    (cons 'b '(* int))))
```

変数宣言

```
int x, *y, z[8];
```

具象構文木

```
(declaration
  (type-specifier 'int)
  (declarator-list
    (list
      (declarator
        (direct-declarator 'x))
      (declarator
        '* (direct-declarator 'y))
      (declarator
        (direct-declarator
          (array-declarator 'z 8))))))
```

よい抽象構文木

```
(declaration
  (list
    (cons 'x 'int)
    (cons 'y '(* int))
    (cons 'z '(array
                int 8))))
```

gen.rkt に関する細かな注意

- あまりよろしくないコードが含まれています

```
...
(define b          'b)    ; b    label
...
(define j          'j)    ; j    target
...
```

- 想定されている使用法

```
(emit j ($ t0))
```

- 想定されていない使用法

```
(let ((j 0)) (emit j ($ t0)))
```

- 回避のしかた

```
(let ((j 0)) (emit 'j ($ t0)))
```