POPL Meeting
(2006/4/19)

米澤研 M2　佐藤秀明

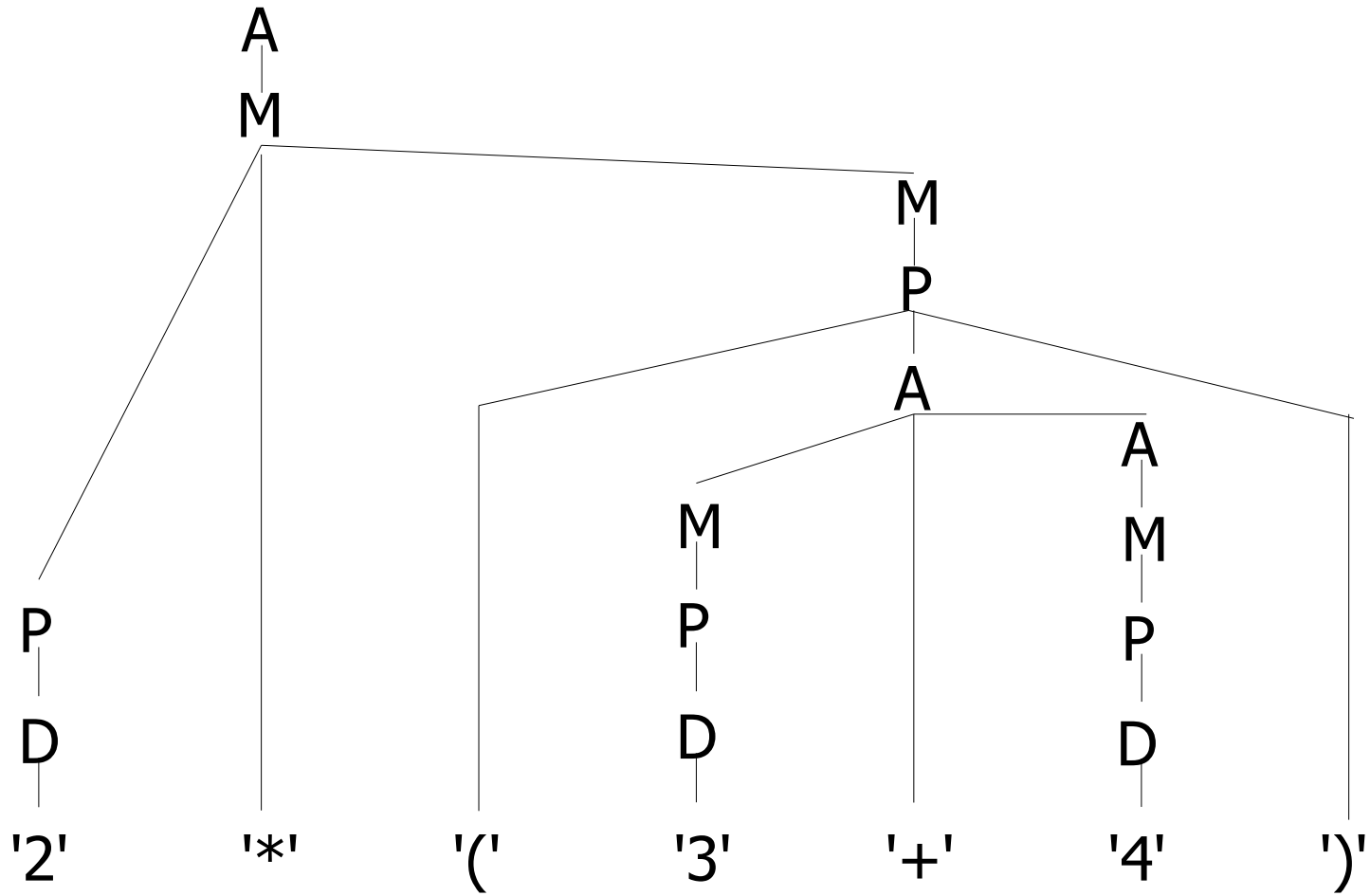
概要

- Packrat Parsing
  - シンプル
  - 線形時間アルゴリズム

構文解析

- 字句解析後のトークン列を構文木に変換
  - 文脈自由文法 (CFG) でルールを記述することが多い

A ← M '+' A | M
M ← P '*' M | P
P ← '(' A ')' | D
D ← '0' | ... | '9'



構文解析法 (1): Descent Parsing

- ルールをトップダウンに適用
- 失敗したら backtrack して次候補を適用
  - 各候補間に優先順位を導入
    - 非決定性を排除
  - 問題：戻って同じ計算を複数回行う可能性
- 改良：n 文字分を先読み
  - 失敗 /backtrack しないように予測しつつ進む
  - 先読みで対処できるような文法の構築は面倒

Descent Parsing の例 (1/2)

- 返り値の定義
  - v: 返す構文の意味 (int など)
  - String: 読み残した文字列
  - NoParse: パーズ失敗

```
data Result v = Parsed v String
              | NoParse
```

- コードサンプルは次頁

Descent Parsing の例 (2/2)

- 単純に再帰呼び出し

A ← M '+' A | M
M ← P '*' M | P
P ← '(' A ')' | D
D ← '0' | ... | '9'

```
--Parse an additive-precedence expression
pAdditive :: String -> Result Int
pAdditive s = alt1 where
    -- Additive <- Multitive '+' Additive
    alt1 = case pMultitive s of
                Parsed vleft s' ->
                  case s' of
                    ('+':s'') ->
                      case pAdditive s'' of
                        Parsed vright s''' ->
                          Parsed (vleft + vright) s'''
                        _ -> alt2
                    _ -> alt2
                _ -> alt2
    -- Additive <- Multitive
    alt2 = case pMultitive s of
                Parsed v s' -> Parsed v s'
                NoParse -> NoParse
```

構文解析法 (2): Tabular Parsing

- トークン列の後ろから動的計画法でパーズ
  - 同じ計算を繰り返さない→線形時間に
    - 動的計画法： 小問題を合成して再帰的に大問題を解く
- 問題 1: 本当は不要な小問題も計算
  - トップダウンでないと本当に必要な小問題か分からない
- 問題 2: 各問題間の依存関係が複雑
  - 小問題を解いていく順序が重要

Tabular Parsing の例

- 表を順に埋めていく
  - (n, Cm): m-1 まで読んだ結果が n という意味を持つ

column	C1	C2	C3	C4	C5	C6	C7	C8
pAdditive				(7, C7)	X	(4, C7)	X	X
pMultitive			↑	(3, C5)	X	(4, C7)	X	X
pPrimary		←	?	(3, C5)	X	(4, C7)	X	X
pDecimal			X	(3, C5)	X	(4, C7)	X	X
input	'2'	'*'	'('	'3'	'+'	'4'	')'	(end)

  - '?' にあてはまるのは (7, C8)
    - '(' と (7, C7) と ')' を合成

Packrat Parser

- descent parsing with backtrack + tabular parsing
  - トップダウンに探索
    - 必要な小問題を必要になった順番で解く
  - 一度解いた小問題の解は表に記録しておく
    - 線形時間は保たれる
- 遅延評価 (call-by-need) で簡単に実装可能
  - 必要になったら解く
  - 同じ計算を繰り返さない

Packrat Parsing の例 (1/3)

- 返り値の再定義

```
data Result v = Parsed v Derivs
              | NoParse

data Derivs = Derivs {
                dvAdditive :: Result Int,
                dvMultitive :: Result Int,
                dvPrimary :: Result Int,
                dvDecimal :: Result Int,
                dvChar :: Result Char}
```

  - Derivs: 表の１つの列を表現

Packrat Parsing の例 (2/3)

- 再帰呼び出しの
  代わりに表を探索
  - 既に解があれば再利用
  - なければ評価開始
- 表をもらって
  表の読み残しを返す
  - cf. string をもらって
    string の読み残しを返す

```
A ← M '+' A | M
M ← P '*' M | P
P ← '(' A ')' | D
D ← '0' | ... | '9'
```

```
--Parse an additive-precedence expression
pAdditive :: Derivs -> Result Int
pAdditive d = alt1 where
  -- Additive <-Multitive '+' Additive
  alt1 = case dvMultitive d of
           Parsed vleft d' ->
             case dvChar d' of
               Parsed '+' d'' ->
                 case dvAdditive d'' of
                   Parsed vright d''' ->
                     Parsed (vleft + vright) d'''
                   _ -> alt2
               _ -> alt2
           _ -> alt2
  -- Additive <- Multitive
  alt2 = dvMultitive d
```

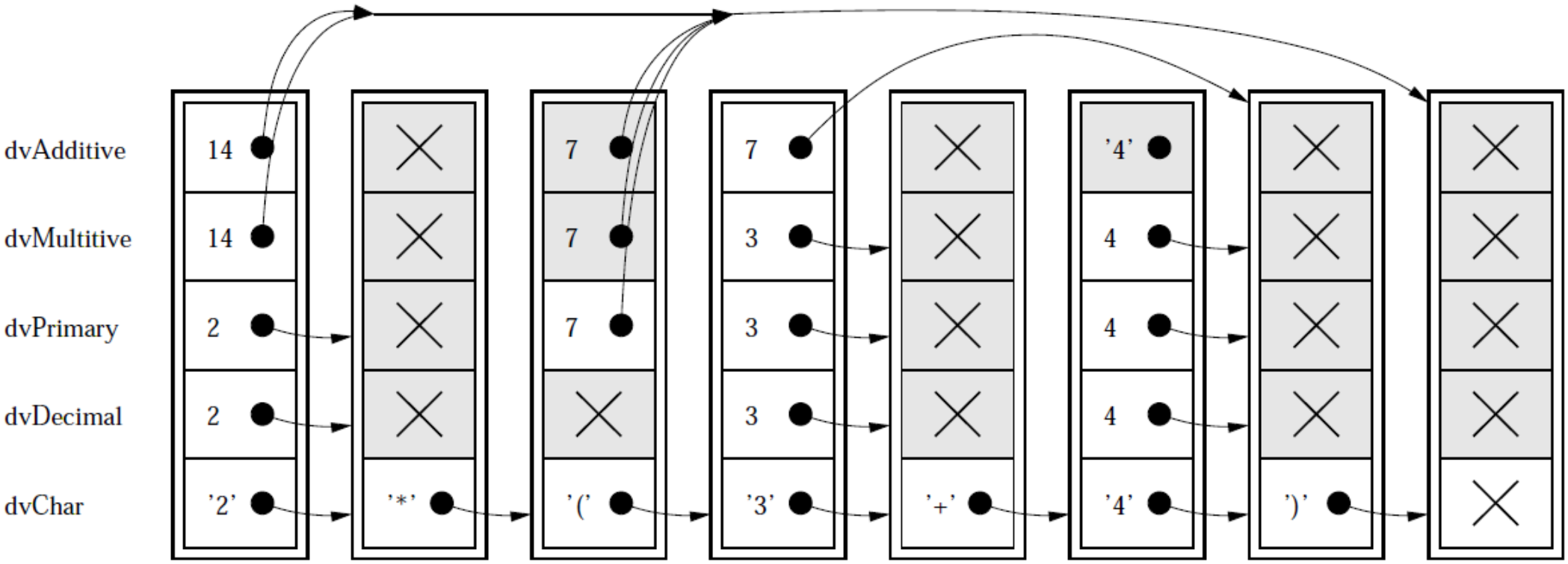
Packrat Parsing の例 (3/3)

- トップレベル関数
  - 表の組み立て
    - chr は表の現在位置以降を参照
    - chr 以外は表の読み残しを参照

```
--Create a result matrix for an input string
parse :: String -> Derivs
parse s = d where
    d = Derivs add mult prim dec chr
    add = pAdditive d
    mult = pMultitive d
    prim = pPrimary d
    dec = pDecimal d
    chr = case s of
            (c:s') -> Parsed c (parse s')
            [] -> NoParse
```

表の構造

- 影付きの部分は評価されない



Packrat Parser の利点 (1/3)

- しくみが単純
  - 人間が手で書くことも可能
    - yacc 要らず？
  - 後から文法を拡張しやすい
    - 文法上の細かい制限がない
      - shift/reduce conflict とか

S ← R | ID ‘=’ R
R ← A | A EQ A | A NE A
A ← P | P ‘+’ P | P ‘-’ P
P ← ID | ‘(’ R ‘)’

LR(1) grammar

ID ← ’a’ | ’a’ ID
EQ ← ’=’ ’=’
NE ← ’!’ ’=’

LR(1) では '=' と '==' の
区別がつかない。
かといって LR(2) は非現実的。

not LR(1) but LR(2) grammar

Packrat Parser の利点 (2/3)

- 字句解析との統合
  - longest-match を自然に書ける
    - 数字 / スペース / コメント / シンボル名などの tokenize に必要
    - CFG では非決定性のため explicit に書けない

```
pWhitespace :: Derivs -> Result ()
pWhitespace d =
    case dvChar d of
      Parsed c d' -> if isSpace c then pWhitespace d' else Parsed () d
      _ -> Parsed () d
```

  - tokenization 不要の「先読み」機構を自然に実現
    - LL/LR 文法では先読み記号は終端記号に限られる
      →事前のトークン切り出しが必須
    - Packrat Parser は現在位置以降の構文情報を用いてパーズ
      →先読みトークンが構文木化していても問題無し

Packrat Parser の利点 (3/3)

- Syntactic predicates の実装が容易
  - 先読みするだけで実際に文字列を消費しない述語
    - followed-by A:

```
p :: Derivs -> Result A
p d = case dvA d of
        Parsed a d' -> Parsed a d
        NoParse -> NoParse
```

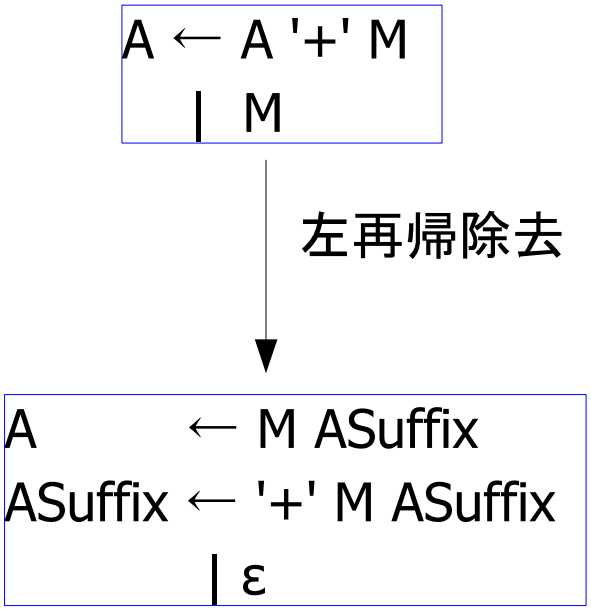
    - not-followed-by A:

```
p :: Derivs -> Result ()
p d = case dvA d of
        Parsed a d' -> NoParse
        NoParse -> Parsed () d
```

  - プログラマが明示的に先読みしたいときに使用

左再帰の処理 (1/2)

- まず左再帰でないものに変換
  - 文法に左再帰があるとパーズが止まらない
  - 変換アルゴリズムは既知



左再帰の処理 (2/2)

- 部分適用に相当する値は lambda 式で表現
  - infix operator の処理に使用

```
A       ← M ASuffix
ASuffix ← '+' M ASuffix
        | ε
```

```
pAdditive :: Derivs -> Result Int
pAdditive :: d = case dvM d of
            Parsed vl d' ->
              case dvAdditiveSuffix d' of
                Parsed f d'' ->
                  Parsed (f vl) d''
                _ -> NoParse
            _ -> NoParse
```

```
pAdditiveSuffix :: Derivs -> Result (Int -> Int)
pAdditiveSuffix :: d = alt1 where
  alt1 = case dvChar d of
           Parsed '+' d' ->
             case dvMultitive d' of
               Parsed vr d'' ->
                 case dvAdditiveSuffix d'' of
                   Parsed f d''' ->
                     Parsed (\vl -> f (vl + vr))
                            d'''
                   _ -> alt2
               _ -> alt2
           _ -> alt2
  alt2 = Parsed (\v -> v) d
```

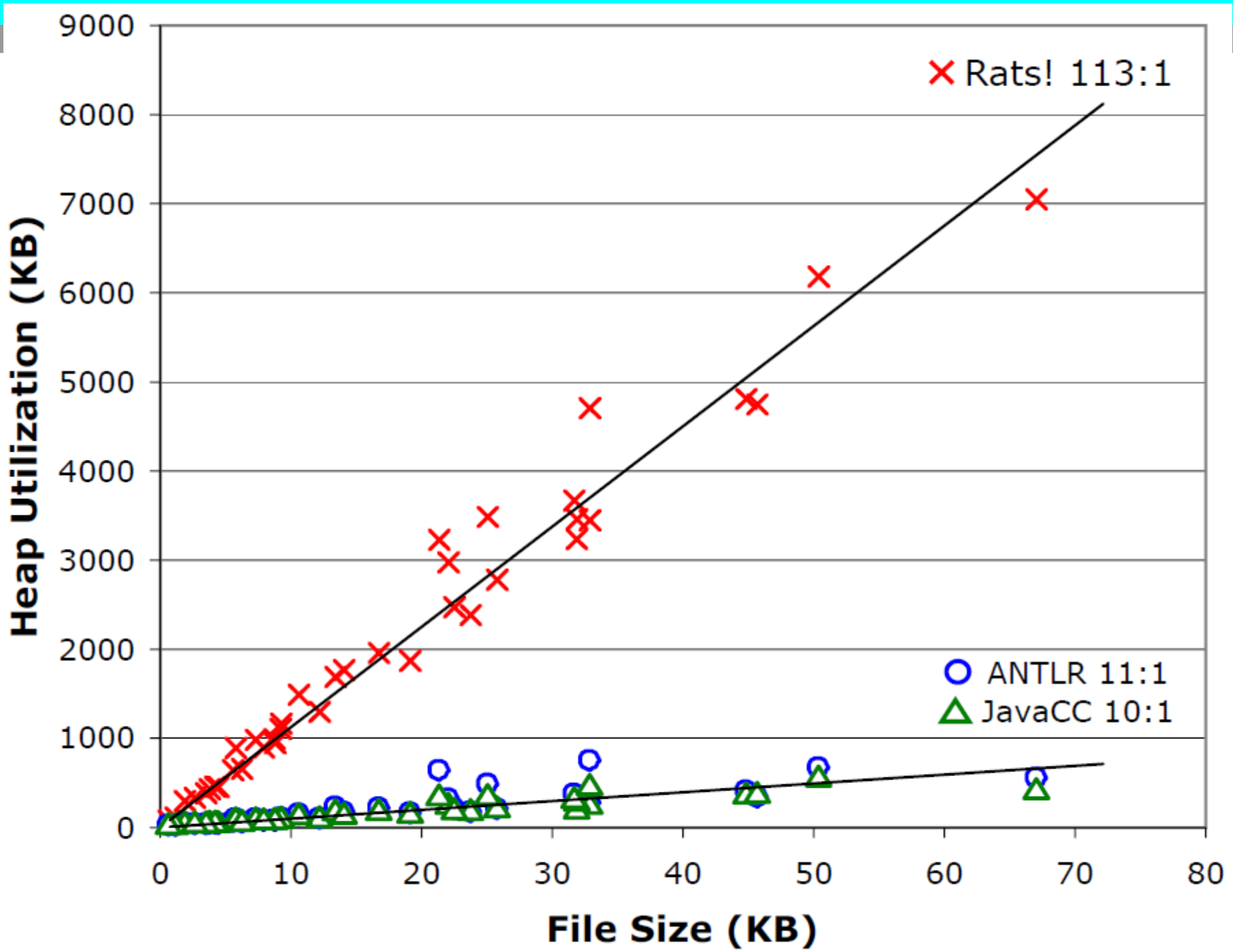
状態つき Packrat Parsing

- 今の実装：パーズ結果と評価時の状態とは独立
  - だから昔の評価結果をいつでも使いまわしできる
- 問題：文脈依存文法 (CSG) を扱えない
  - 例 :C/C++ での型名と変数名の区別
    - ユーザ定義型をまとめたテーブルを逐次更新する必要性
  - 状態を持ちながらのパーズが不可能
- 状態を扱えるようにするには？
  - 状態が変わる度に表中の依存部分を更新
    - 線形時間では収まらない

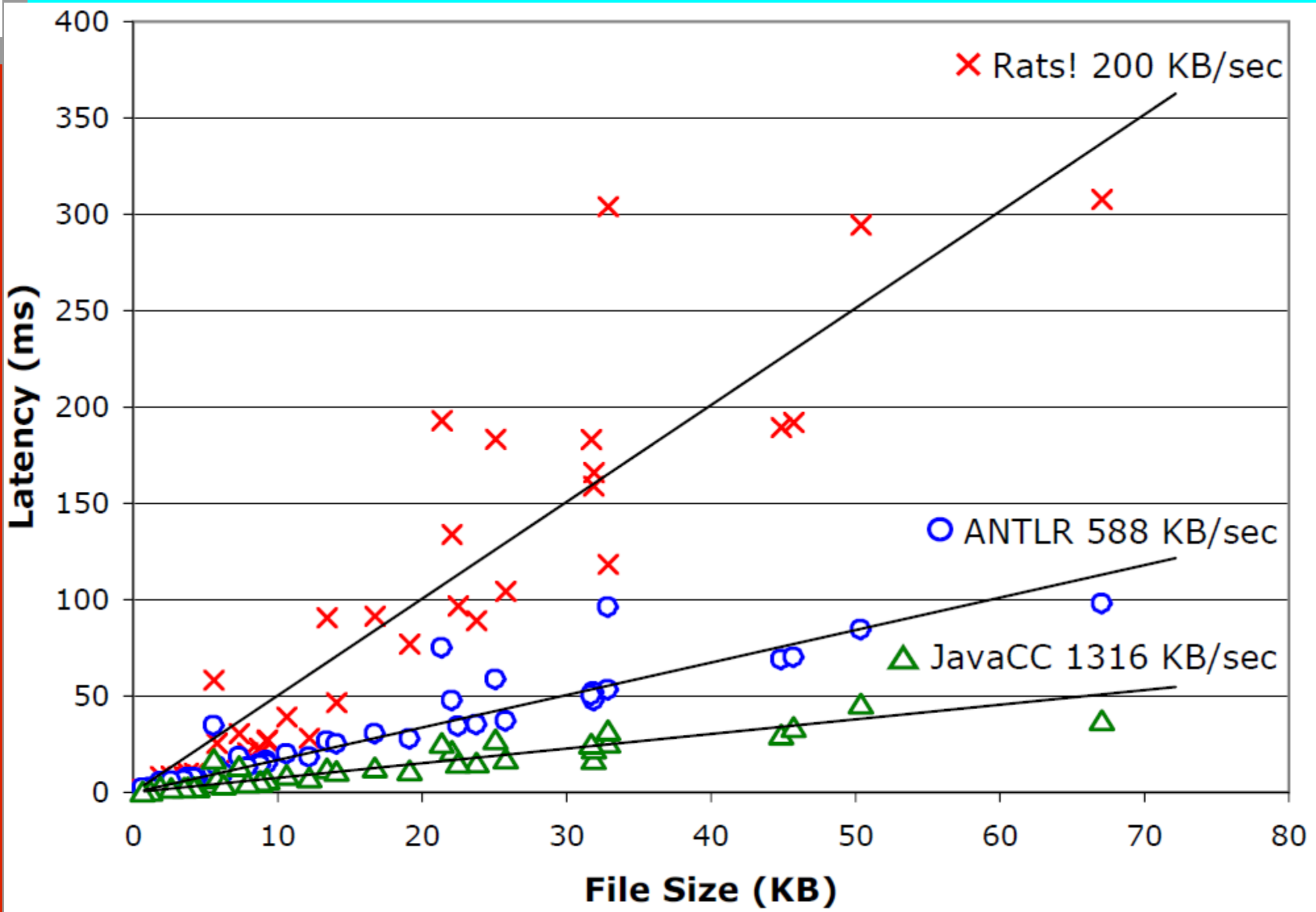
欠点

- Deterministic parsing
  - 高々１つの答えしか返さない
    - 自然言語のような曖昧な言語には不向き
    - プログラミング言語への用途なら十分だろう
- 大きいメモリ使用量
  - 表のサイズが大きい
    - 大きな xml ファイルの処理には不向き
    - プログラミング言語のソースファイルは小さいから大丈夫だろう
- 実は既存アルゴリズムより遅い
  - Java で実装した人の報告 (Rats!)
    - 実装の洗練が必要？
      - 受け取ったのをそのまま返すだけなのにいちいち new してた

メモリ使用量の比較


Heap Utilization (KB)
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
10
20
30
40
50
60
70
80
File Size (KB)
Rats! 113:1
ANTLR 11:1
JavaCC 10:1

実行時間の比較


Rats! 200 KB/sec
ANTLR 588 KB/sec
JavaCC 1316 KB/sec
Latency (ms)
File Size (KB)
0
50
100
150
200
250
300
350
400
0
10
20
30
40
50
60
70
80

様々な最適化

- 表の階層化
  - 表のサイズを小さくする
- 共通 prefix をまとめる
  - terminal の string を if の連鎖でなく switch でさばくなど
- 静的に確定する評価結果はわざわざ返さない
  - 成功したら返ってくる値が常に同じである場合
- repetition を再帰呼び出しで実装しない
  - text を処理する際の stack overflow を防ぐ

Parsing Expression Grammar

- CFG の拡張
  - 優先順序付き choice
    - determinism を表現
  - syntactic predicates
    - And-predicate: followed-by
    - Not-predicate: not-followed-by
  - repetition
    - longest-match
- PEG を Parser generator で packrat parser に変換

PEG のオペレータ

Operator	Type	Precedence	Description
' '	primary	5	Literal string
" "	primary	5	Literal string
[ ]	primary	5	Character class
.	primary	5	Any character
(*e*)	primary	5	Grouping
e?	unary suffix	4	Optional
*e**	unary suffix	4	Zero-or-more
e+	unary suffix	4	One-or-more
&*e*	unary prefix	3	And-predicate
!*e*	unary prefix	3	Not-predicate
e_1 e_2	binary	2	Sequence
e_1 / e_2	binary	1	Prioritized Choice

PEG の利点 (1/2)

- C++ の nested template 問題

```
vector<vector<float> > MyMatrix;
```

  - 1 文字空けないと右シフト演算子になってしまう仕様
  - PEG なら簡潔に記述可能
    - not-followed-by を活用

```
LANGLE  <-  !LSHIFT  '<'  Spacing
RANGLE  <-  !RSHIFT  '>'  Spacing
LSHIFT  <-  '<<'  Spacing
RSHIFT  <-  '>>'  Spacing
```

- nested comments も簡潔に表現

```
Comment  <-  '(*'  (  Comment  /  !'*)' .  )*  '*)'
```

PEG の利点 (2/2)

- ぶら下がり else 問題

if c1 then if c2 then A else B

```
Statement <- IF Cond THEN Statement ELSE Statement
           |  IF Cond THEN Statement
```

  - CFG だと曖昧になってしまう
    - else 節はどちらの if 節に属するか？
    - 実装に informal な修正を加えて対処するしかない
  - PEG なら簡潔に表現可能
    - 優先順位付き choice を活用

```
Statement <- IF Cond THEN Statement ELSE Statement
           /  IF Cond THEN Statement
```

PEG の simplification

- syntax sugar の除去
  - [c1-cn] --> c1/c2/.../cn
  - e? --> e/ε
  - e+ --> ee*
  - e* --> e* を非終端記号 A で置き換え、A←eA/ε を追加
  - &e --> !(!e)
  - 実は ! も除去できる
    - アルゴリズムは複雑なので割愛
- 結局 PEG は CFG に優先順序付き choice を入れただけのもの

まとめ

- Packrat Parsing
  - 遅延評価を用いた構文解析
    - descent parser
    - 動的計画法
  - 様々な利点
    - しくみの単純化
    - 線形時間アルゴリズム
    - 字句解析との統合
- Parsing Expression Grammar
  - packrat parser を生成する文法の定式化
    - 優先順位付き choice
    - syntactic predicates

各種ソースコード

- http://pdos.csail.mit.edu/~baford/packrat/thesis/
  - Java の language specification
    - PEG で記述
  - parser generator
    - Haskell で実装
  - Java の packrat parser
    - PEG 記述から parser generator を用いて生成
    - 動作環境は Haskell
  - monad を使って書いた Java の packrat parser

References(1/2)

- Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time. In Proceedings of the 2002 International Conference on Functional Programming, Oct 2002.
- Bryan Ford. Packrat Parsing: A practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, Sept. 2002.

References(2/2)

- Bryan Ford. Parsing Expression Grammars: A recognition-based syntactic foundation. In Proceedings of the 31st ACM Symposium on Principles of Programming Languages, pages 111-122, Venice, Italy, Jan. 2003.
- Robert Grimm. Practical packrat parsing. Technical Report TR2004-854, New York University, Mar. 2004.