Packrat Parsing: Simple, Powerful, Lazy, Linear Time

Bryan Ford
Massachusetts Institute of Technology

International Conference on Functional Programming, October 2002

Overview

- What Is Packrat Parsing?
- What is it Good (and not good) For?
- Practical Experience
- Related Work
- Conclusion

What is Packrat Parsing?

Answer:

Top-down parsing with backtracking –

except:

Uses memoization to achieve linear parse time

Example Grammar

Additive	→	Multitive '+' Additive
	|	Multitive
Multitive	→	Primary '*' Multitive
	|	Primary
Primary	→	'(' Additive ')'
	|	Decimal
Decimal	→	'0' | … | '9'

Recursive Descent Parser

Recursive Descent Parser

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

Recursive Descent Parser

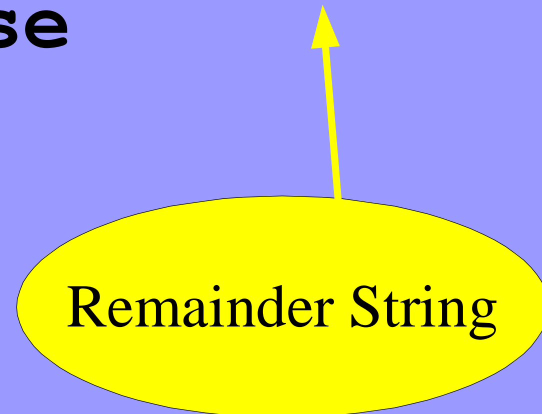
```
data Result v = Parsed v String
              | NoParse
```

Recursive Descent Parser

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

Recursive Descent Parser

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



Recursive Descent Parser

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

Recursive Descent Parser

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

pAdditive  :: String -> Result Int
pMultitive:: String -> Result Int
pPrimary   :: String -> Result Int
pDecimal   :: String -> Result Int
```

Recursive Descent Parser

```
pAdditive :: String -> Result Int
```

Recursive Descent Parser

-- Additive → Multitive '+' Additive
-- | Multitive

```
pAdditive :: String -> Result Int
```

Recursive Descent Parser

```
-- Additive →   Multitive '+' Additive
--          |   Multitive
pAdditive :: String -> Result Int
pAdditive =
       (do l <- pMultitive
           char '+'
           r <- pAdditive
           return (l + r))
   <|> (do pMultitive)
```

Recursive Descent Parser

```
-- Additive →   Multitive '+' Additive
--           |   Multitive
pAdditive :: String -> Result Int
pAdditive =
        (do l <- pMultitive
            char '+'
            r <- pAdditive
            return (l + r))
    <|> (do pMultitive)
```

Recursive Descent Parser

```
-- Additive →   Multitive '+' Additive
--          |   Multitive
pAdditive :: String -> Result Int
pAdditive =
       (do l <- pMultitive
           char '+'
           r <- pAdditive
           return (l + r))
   <|> (do pMultitive)
```

Recursive Descent Parser

```
-- Additive →    Multitive '+' Additive
--            |  Multitive
pAdditive :: String -> Result Int
pAdditive =
        (do l <- pMultitive
            char '+'
            r <- pAdditive
            return (l + r))
    <|> (do pMultitive)
```

Recursive Descent Parser

```
-- Additive →   Multitive '+' Additive
--            |   Multitive
pAdditive :: String -> Result Int
pAdditive =
        (do l <- pMultitive
            char '+'
            r <- pAdditive
            return (l + r))
    <|> (do pMultitive)
```

Recursive Descent Parser

```
-- Additive →    Multitive '+' Additive
--           |    Multitive
pAdditive :: String -> Result Int
pAdditive =
      (do l <- pMultitive
          char '+'
          r <- pAdditive
          return (l + r))
   <|> (do pMultitive)
```

Recursive Descent Parser

```
-- Additive →   Multitive '+' Additive
--            |   Multitive
pAdditive :: String -> Result Int
pAdditive =
      (do l <- pMultitive
          char '+'
          r <- pAdditive
          return (l + r))
  <|> (do pMultitive)
```

Parsing Example

pAdditive “2*(3+4)”

Parsing Example

pAdditive "2*(3+4)"

$A \rightarrow M$ '+' A

Parsing Example

pAdditive “2*(3+4)”

pMultitive “2*(3+4)”

A → M '+' A

Parsing Example

pAdditive "2*(3+4)"   A → M '+' A

pMultitive "2*(3+4)"   M → P '*' M

Parsing Example

pAdditive “2*(3+4)”

pMultitive “2*(3+4)”

pPrimary “2*(3+4)”

$A \rightarrow M$ '+' A

$M \rightarrow P$ '*' M

Parsing Example

pAdditive "2*(3+4)" A → M '+' A

pMultitive "2*(3+4)" M → P '*' M

pPrimary "2*(3+4)" P → '(' A ')'

Parsing Example

pAdditive “2*(3+4)” A → M '+' A

pMultitive “2*(3+4)” M → P '*' M

pPrimary “2*(3+4)” ~~P → '(' A ')'~~

Parsing Example

pAdditive “2*(3+4)” A → M '+' A

pMultitive “2*(3+4)” M → P '*' M

pPrimary “2*(3+4)” P → D

Parsing Example

pAdditive "2*(3+4)" $A \rightarrow M$ '+' A

pMultitive "2*(3+4)" $M \rightarrow P$ '*' M

pPrimary "2*(3+4)" $P \rightarrow D$

pDecimal "2*(3+4)"

Parsing Example

pAdditive “2*(3+4)” A → M '+' A

pMultitive “2*(3+4)” M → P '*' M

pPrimary “2*(3+4)” P → D

pDecimal “2*(3+4)” ⇒ Parsed 2 “*(3+4)”

Parsing Example

pAdditive “2*(3+4)” $A \rightarrow M$ '+' A

pMultitive “2*(3+4)” $M \rightarrow P$ '*' M

pPrimary “2*(3+4)” $\Rightarrow$ Parsed 2 “*(3+4)”

Parsing Example

pAdditive "2*(3+4)" $A \to M$ '+' A

pMultitive "2*(3+4)" $M \to P$ '*' M

pPrimary "2*(3+4)" $\Rightarrow$ Parsed 2 "*(3+4)"

pChar '*' "*(3+4)"

Parsing Example

pAdditive “2*(3+4)” $A \to M$ '+' A

pMultitive “2*(3+4)” $M \to P$ '*' M

pPrimary “2*(3+4)” $\Rightarrow$ Parsed 2 “*(3+4)”

pChar '*' “*(3+4)” $\Rightarrow$ Parsed () “(3+4)”

Parsing Example

pAdditive “2*(3+4)” — A → M '+' A

pMultitive “2*(3+4)” — M → P '*' M

pPrimary “2*(3+4)” ⇒ Parsed 2 “*(3+4)”

pChar '*' “*(3+4)” ⇒ Parsed () “(3+4)”

pMultitive “(3+4)”

Parsing Example

pAdditive “2*(3+4)” $A \to M$ '+' A

pMultitive “2*(3+4)” $M \to P$ '*' M

pPrimary “2*(3+4)” $\Rightarrow$ Parsed 2 “*(3+4)”

pChar '*' “*(3+4)” $\Rightarrow$ Parsed () “(3+4)”

pMultitive “(3+4)”

…

Parsing Example

pAdditive “2*(3+4)” A → M '+' A

pMultitive “2*(3+4)” M → P '*' M

pPrimary “2*(3+4)” ⇒ Parsed 2 “*(3+4)”

pChar '*' “*(3+4)” ⇒ Parsed () “(3+4)”

pMultitive “(3+4)” ⇒ Parsed 7 “”

Parsing Example

pAdditive “2*(3+4)” $A \rightarrow M$ '+' A

pMultitive “2*(3+4)” $\Rightarrow$ Parsed 14 “”

Parsing Example

pAdditive “2*(3+4)” $A \rightarrow M$ '+' A

pMultitive “2*(3+4)” $\Rightarrow$ Parsed 14 “”

pChar '+' “”

Parsing Example

pAdditive “2*(3+4)” $A \rightarrow M$ '+' A

pMultitive “2*(3+4)” $\Rightarrow$ Parsed 14 “”

~~pChar '+'~~ “”

Parsing Example

pAdditive “2*(3+4)”

$A \rightarrow M$

Parsing Example

pAdditive “2*(3+4)”

pMultitive “2*(3+4)”

$A \rightarrow M$

Parsing Example

pAdditive "2*(3+4)"

pMultitive "2*(3+4)"

...

$A \rightarrow M$

The Backtracking Problem

- Can yield exponential worst-case parse times

The Backtracking Problem

- Can yield exponential worst-case parse times
- **Typical solution:** avoid backtracking by
  - Prediction using one-token lookahead
  - Hacking the grammar
  - Designing the language for easy parsing

The Backtracking Problem

- Can yield exponential worst-case parse times
- **Typical solution:** avoid backtracking by
  - Prediction using one-token lookahead
  - Hacking the grammar
  - Designing the language for easy parsing
- **Alternate solution:** allow backtracking;
  - *memoize all intermediate results.*

Memoization of Results

Assumptions:

- Parsing functions depend *only* on input string.
- Parsing functions yield *at most one result*.

Implication:

- Requires results table of size $(m \times (n+1))$
  - m = number of nonterminals/parsing functions
  - n = length of input string

Building a Packrat Parser

Building a Packrat Parser

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

pAdditive  :: String -> Result Int
pMultitive:: String -> Result Int
pPrimary   :: String -> Result Int
pDecimal   :: String -> Result Int
```

Building a Packrat Parser

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

pAdditive  :: String -> Result Int
pMultitive:: String -> Result Int
pPrimary   :: String -> Result Int
pDecimal   :: String -> Result Int
```

Building a Packrat Parser

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

pAdditive  :: Derivs -> Result Int
pMultitive :: Derivs -> Result Int
pPrimary   :: Derivs -> Result Int
pDecimal   :: Derivs -> Result Int
```

Building a Packrat Parser

```
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}
```

Building the Derivs Structure

```
parse :: String -> Derivs
parse s = d where
```

Building the Derivs Structure

```
parse :: String -> Derivs
parse s = d where

  d   = Derivs add mult prim dec chr
```

Building the Derivs Structure

```
parse :: String -> Derivs
parse s = d where

  d   = Derivs add mult prim dec chr

  chr = case s of
          (c:s') -> Parsed c (parse s')
          []     -> NoParse
```

Building the Derivs Structure

```
parse :: String -> Derivs
parse s = d where

  d   = Derivs add mult prim dec chr

  chr = case s of
          (c:s') -> Parsed c (parse s')
          []     -> NoParse
```

Building the Derivs Structure

```
parse :: String -> Derivs
parse s = d where

  d   = Derivs add mult prim dec chr

  chr = case s of
          (c:s') -> Parsed c (parse s')
          []     -> NoParse
```

Building the Derivs Structure

```
parse :: String -> Derivs
parse s = d where

  d    = Derivs add mult prim dec chr

  chr = case s of
           (c:s') -> Parsed c (parse s')
           []     -> NoParse
```

Building the Derivs Structure

```
parse :: String -> Derivs
parse s = d where

  d   = Derivs add mult prim dec chr

  chr = case s of
          (c:s')-> Parsed c (parse s')
          []    -> NoParse
```

Building the Derivs Structure

```
parse :: String -> Derivs
parse s = d where

  d   = Derivs add mult prim dec chr

  chr = case s of
          (c:s') -> Parsed c (parse s')
          []     -> NoParse
```

Building the Derivs Structure

```
parse :: String -> Derivs
parse s = d where

  d   = Derivs add mult prim dec chr

  chr = case s of
            (c:s') -> Parsed c (parse s')
            []     -> NoParse
```

Building the Derivs Structure

```
parse :: String -> Derivs
parse s = d where

  d    = Derivs add mult prim dec chr

  chr = case s of
          (c:s') -> Parsed c (parse s')
          []     -> NoParse

  add = pAdditive  d
  mult= pMultitive d
  prim= pPrimary   d
  dec = pDecimal   d
```

Building the Derivs Structure

```
parse :: String -> Derivs
parse s = d where

  d    = Derivs add mult prim dec chr

  chr = case s of
          (c:s') -> Parsed c (parse s')
          []     -> NoParse

  add = pAdditive  d
  mult= pMultitive d
  prim= pPrimary   d
  dec = pDecimal   d
```

Modifying the Parsing Functions

```
pAdditive :: Derivs -> Result Int
pAdditive =
       (do l <- pMultitive
           char '+'
           r <- pAdditive
           return (l + r))
   <|> (do pMultitive)
```

Modifying the Parsing Functions

```
pAdditive :: Derivs -> Result Int
pAdditive =
       (do l <- pMultitive
           char '+'
           r <- pAdditive
           return (l + r))
   <|> (do pMultitive)
```

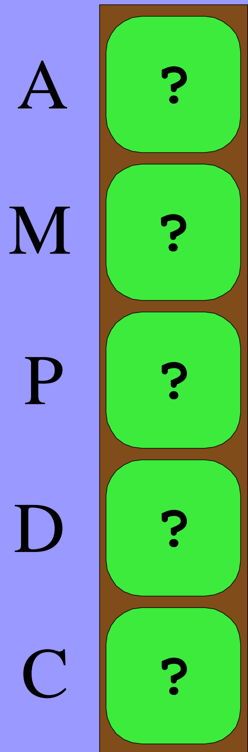
Modifying the Parsing Functions

```
pAdditive :: Derivs -> Result Int
pAdditive =
        (do l <- dvMultitive
            char '+'
            r <- dvAdditive
            return (l + r))
    <|> (do dvMultitive)
```

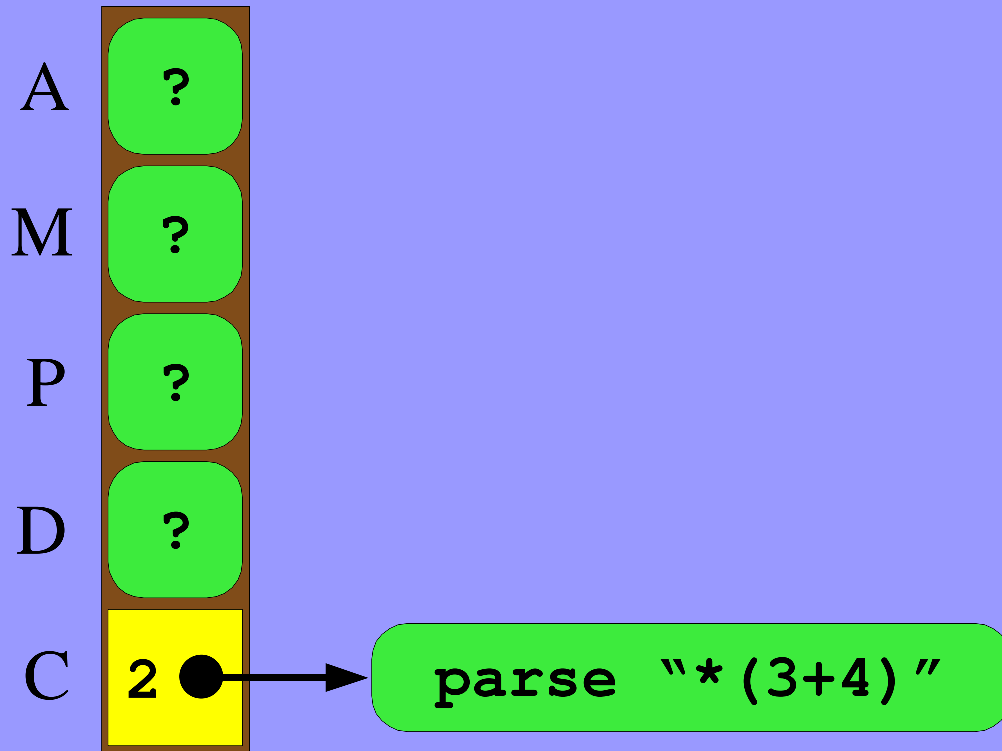
Packrat Parsing Example

```
parse "2*(3+4)"
```

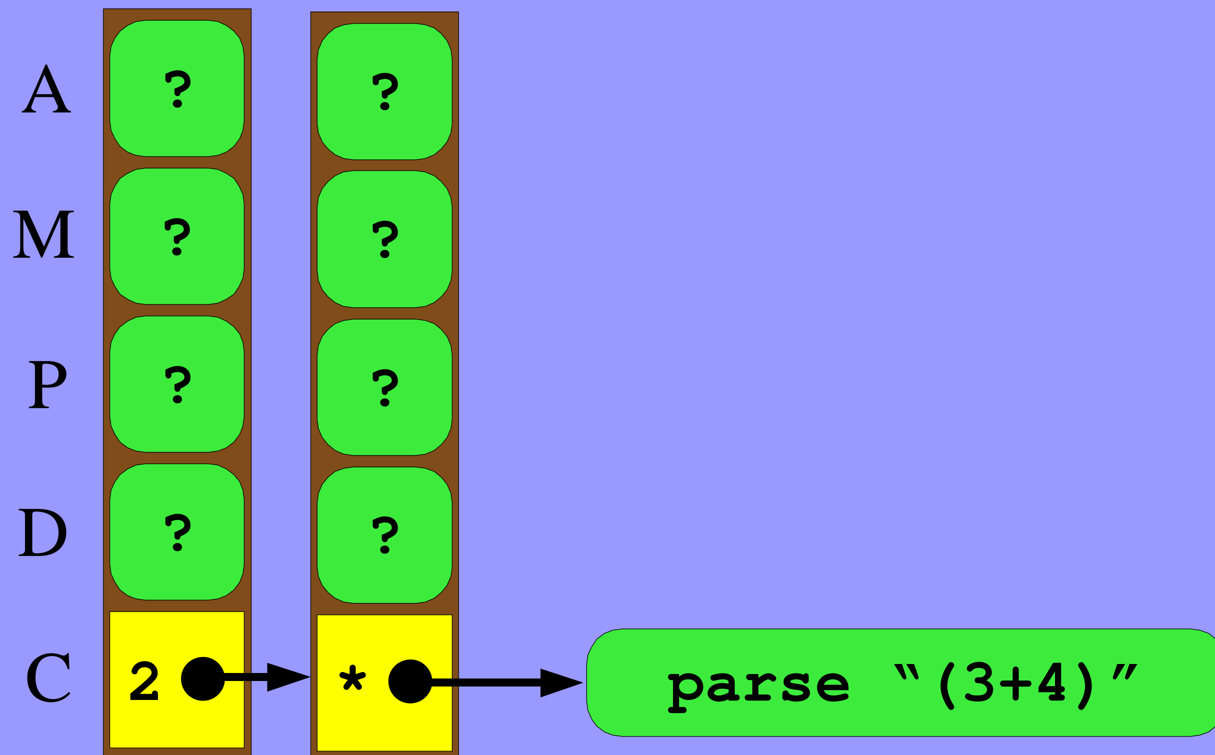
Packrat Parsing Example



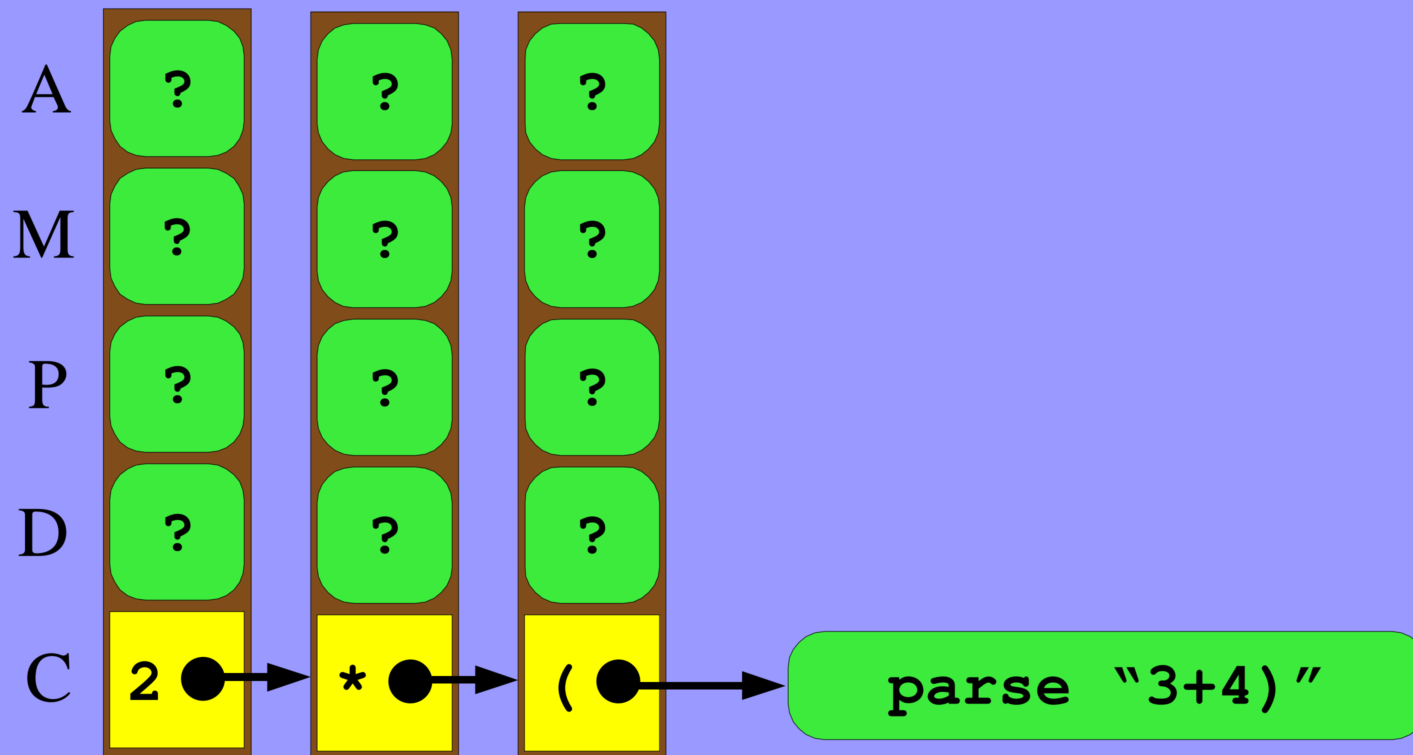
Packrat Parsing Example



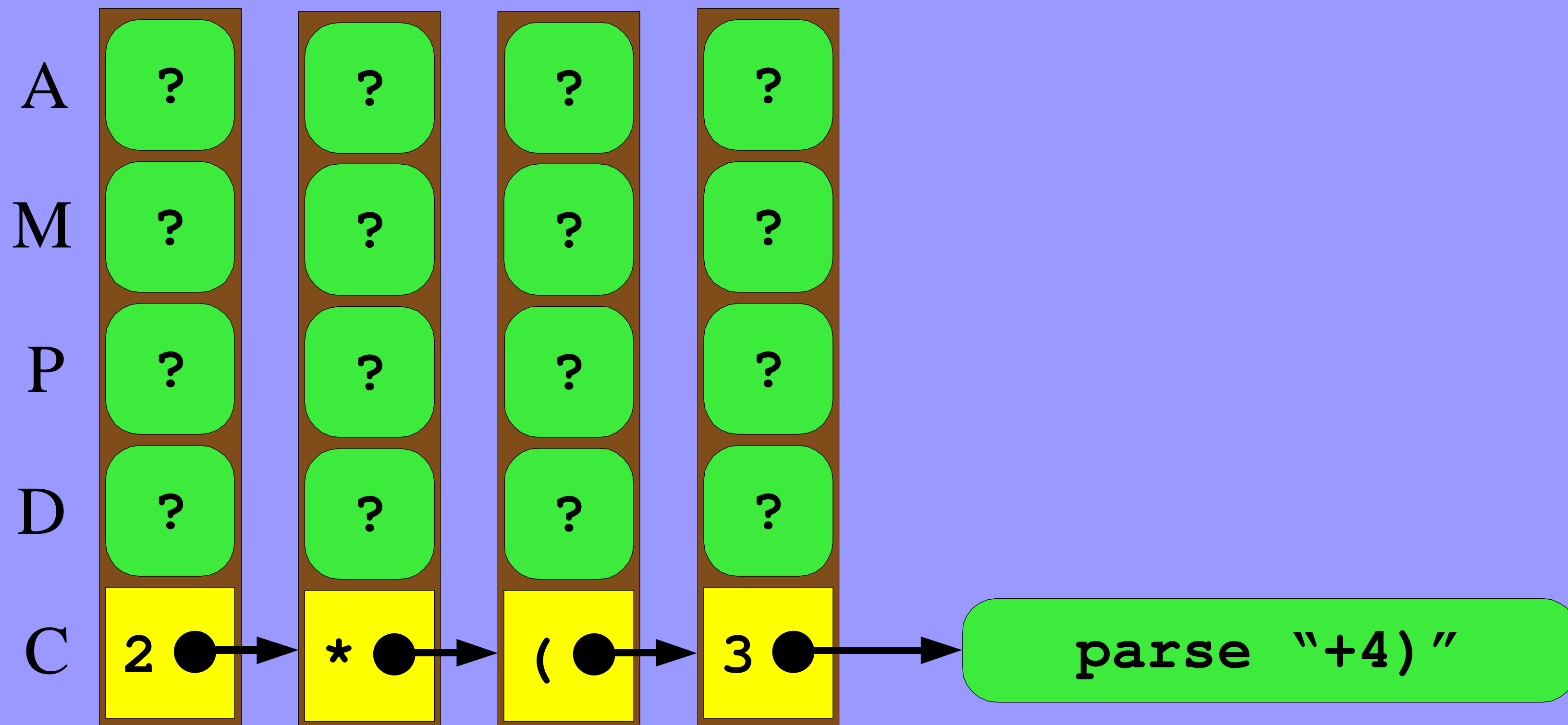
Packrat Parsing Example



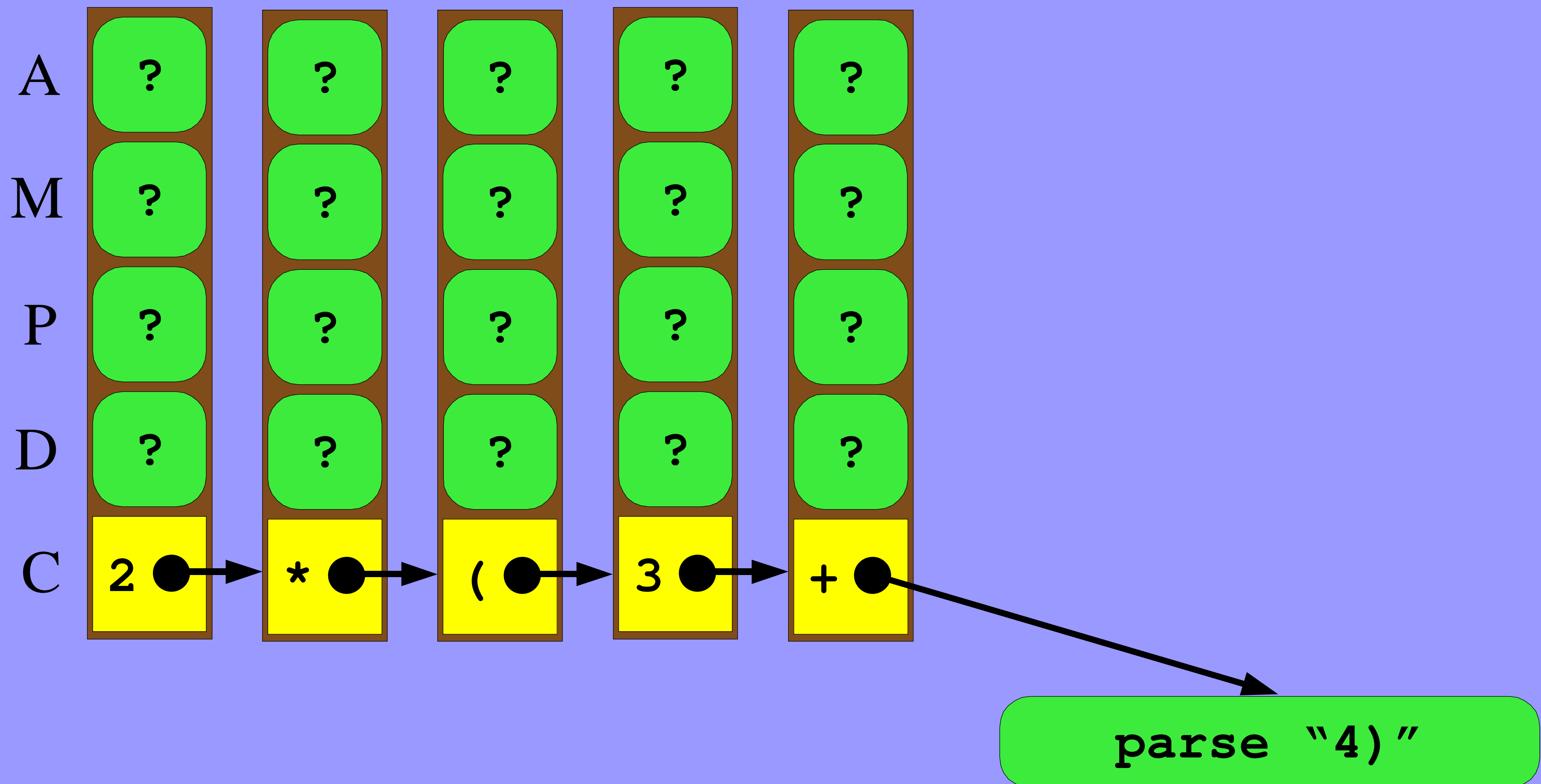
Packrat Parsing Example



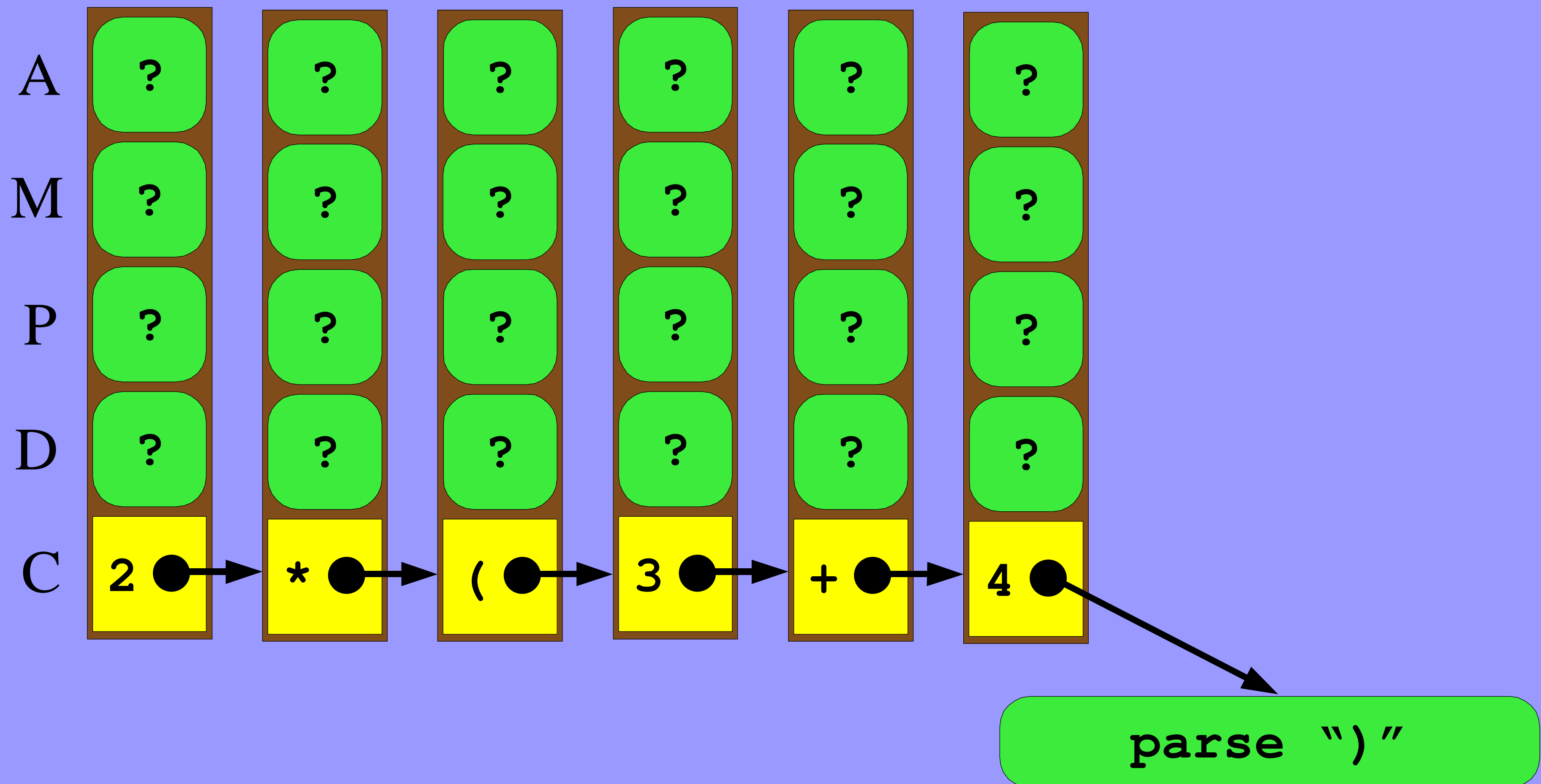
Packrat Parsing Example



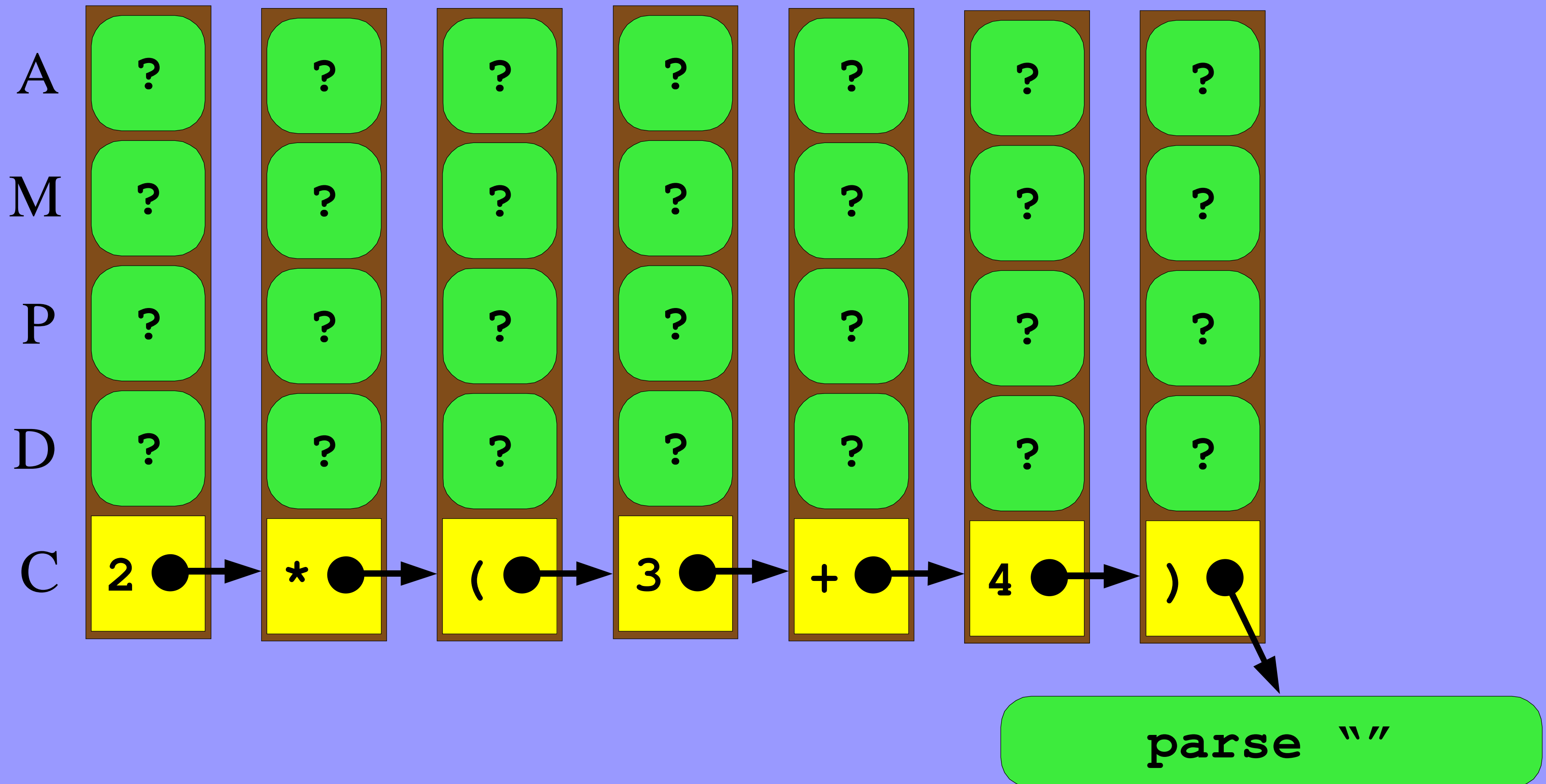
Packrat Parsing Example



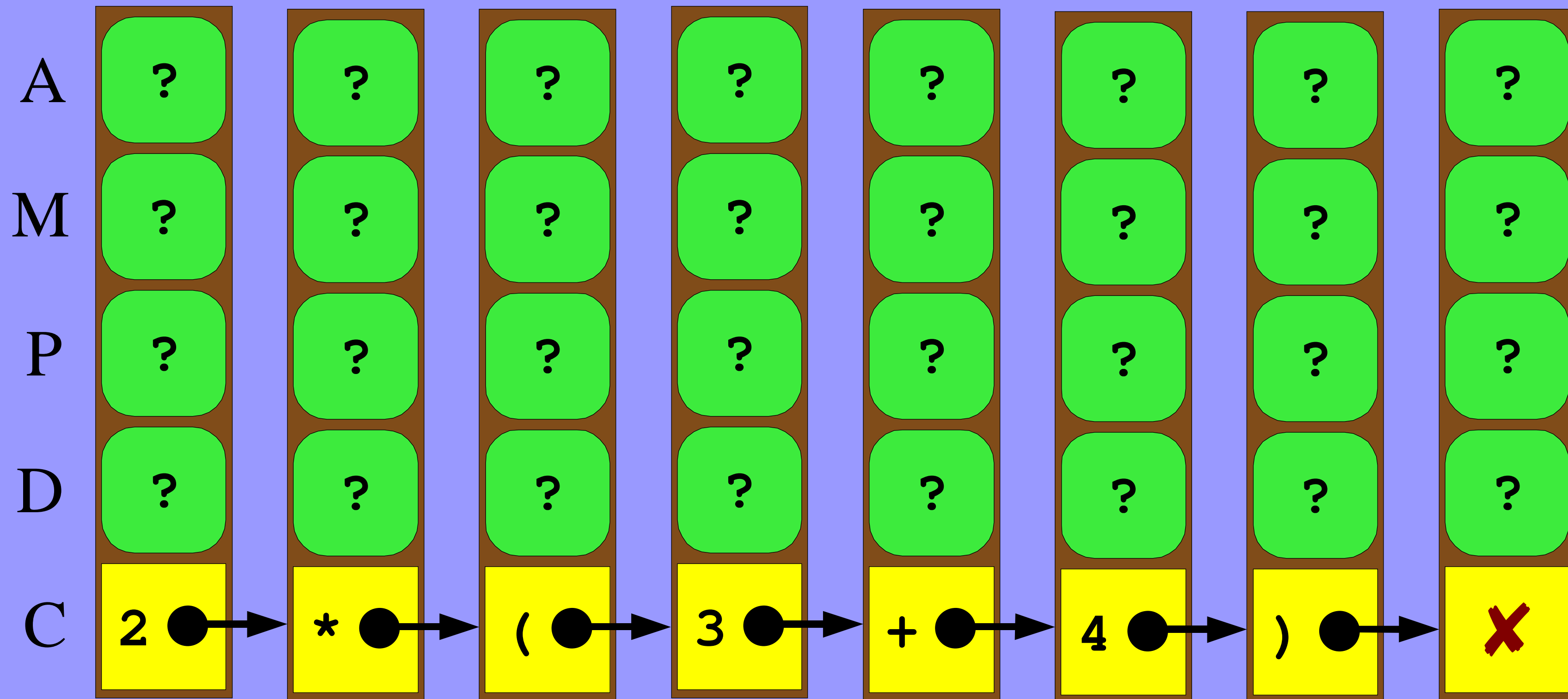
Packrat Parsing Example



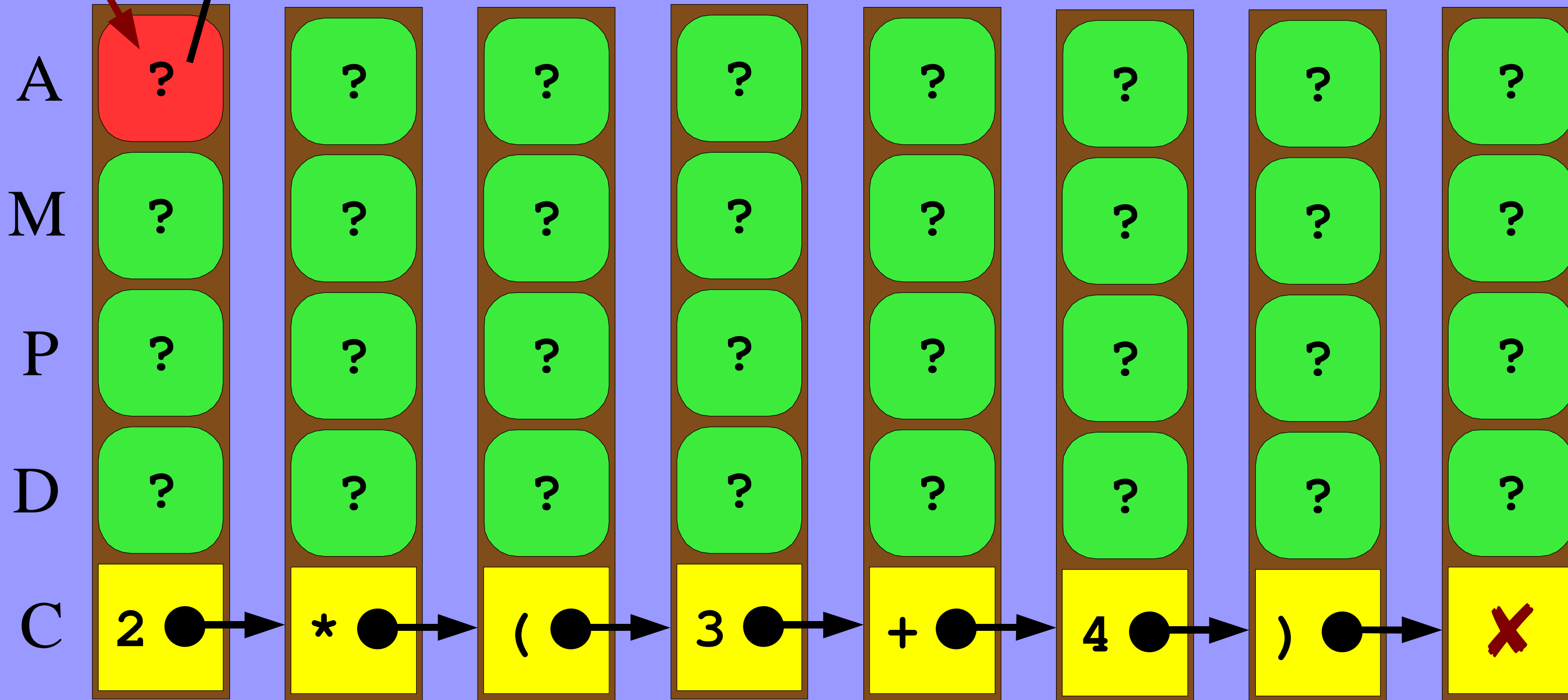
Packrat Parsing Example



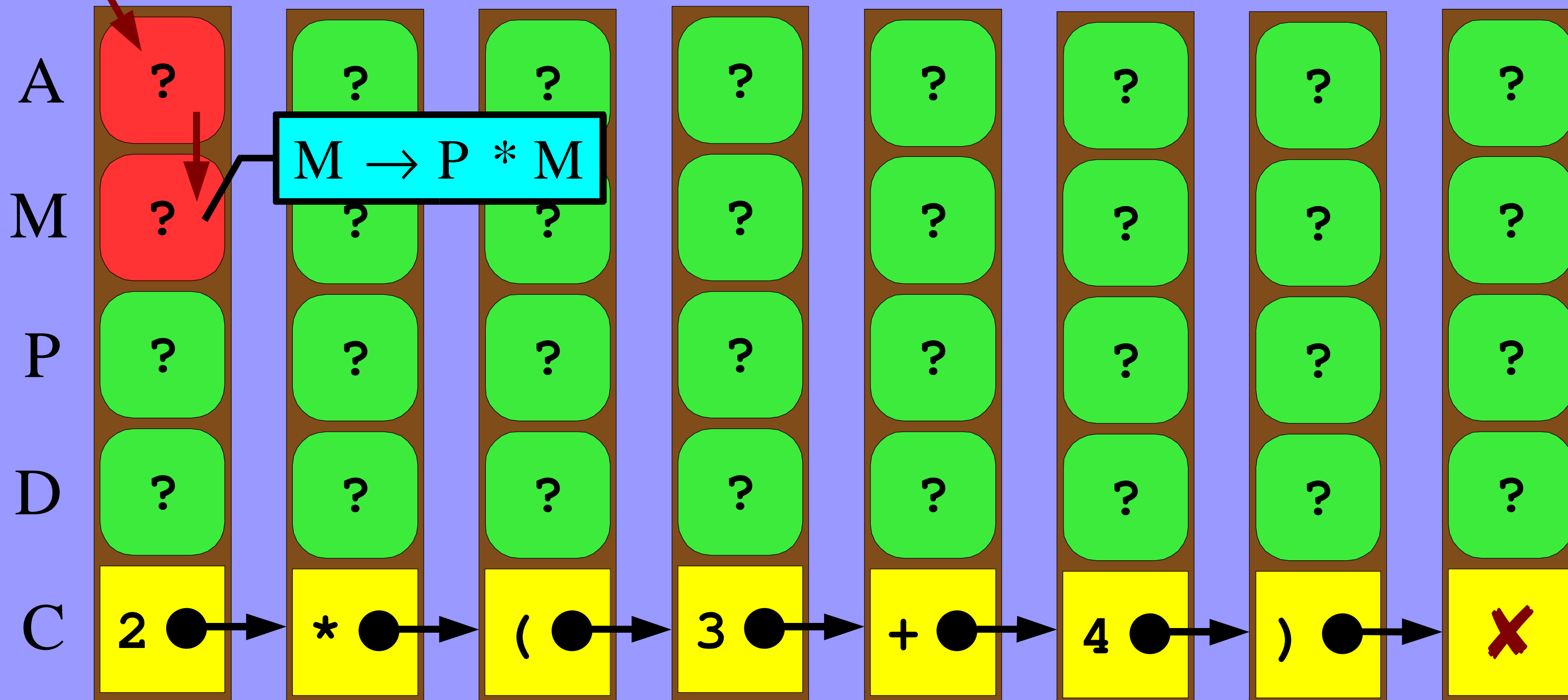
Packrat Parsing Example



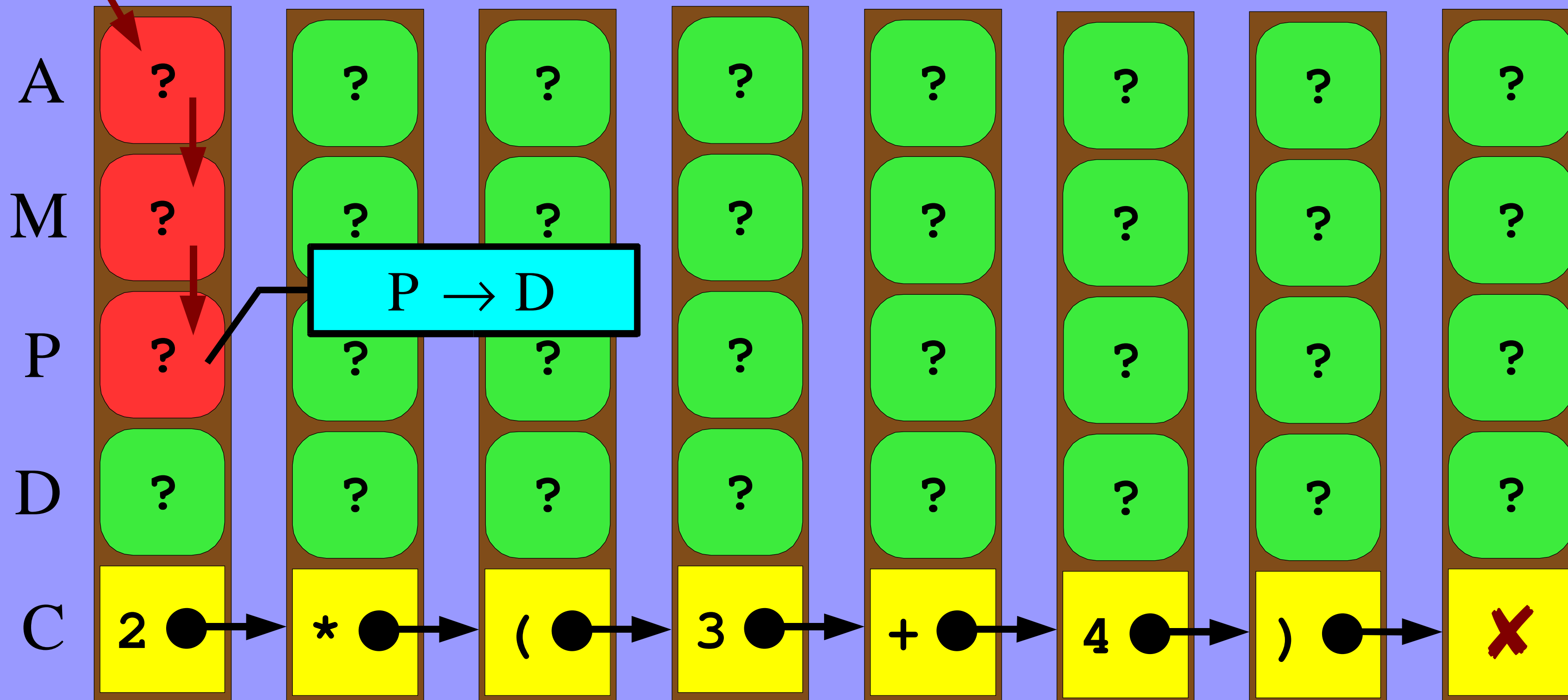
Packrat Parsing Example



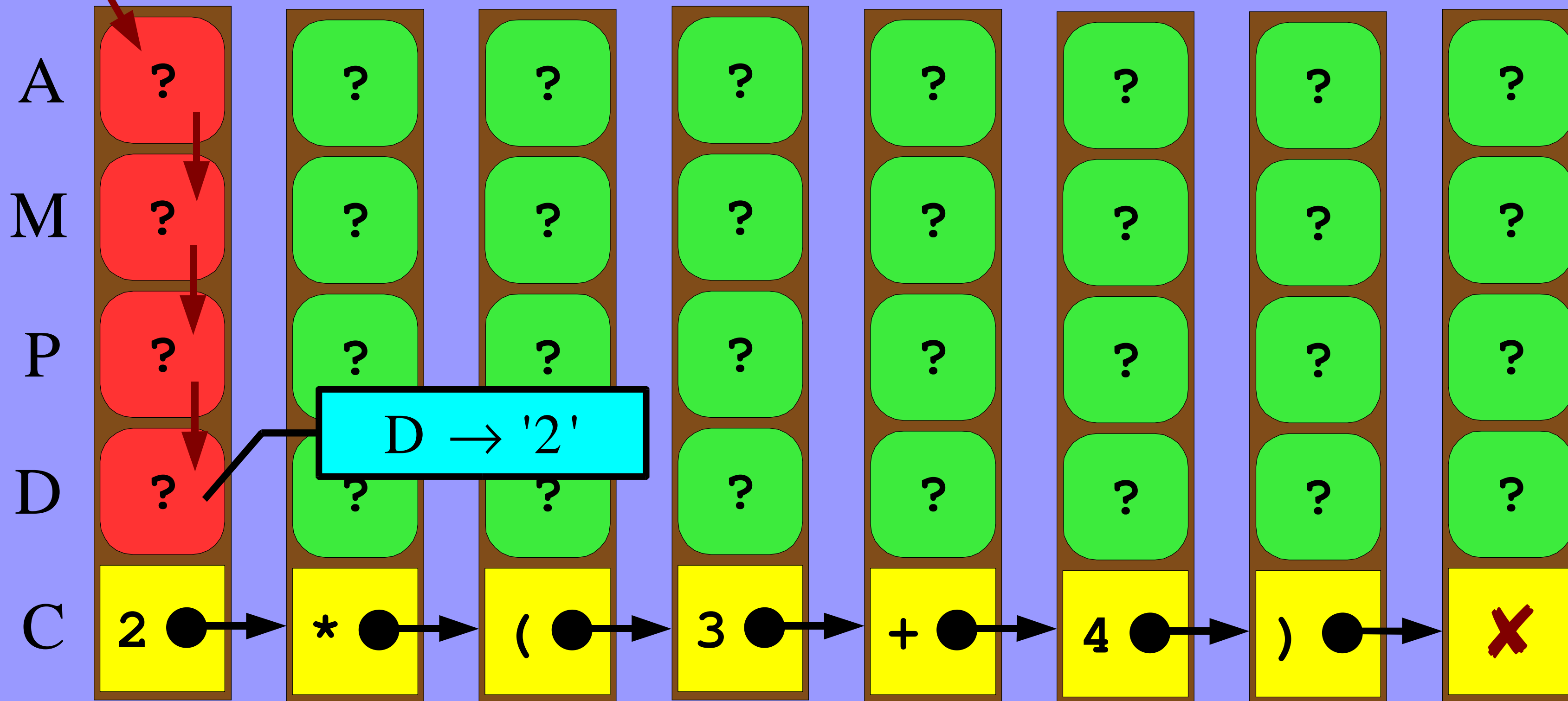
Packrat Parsing Example



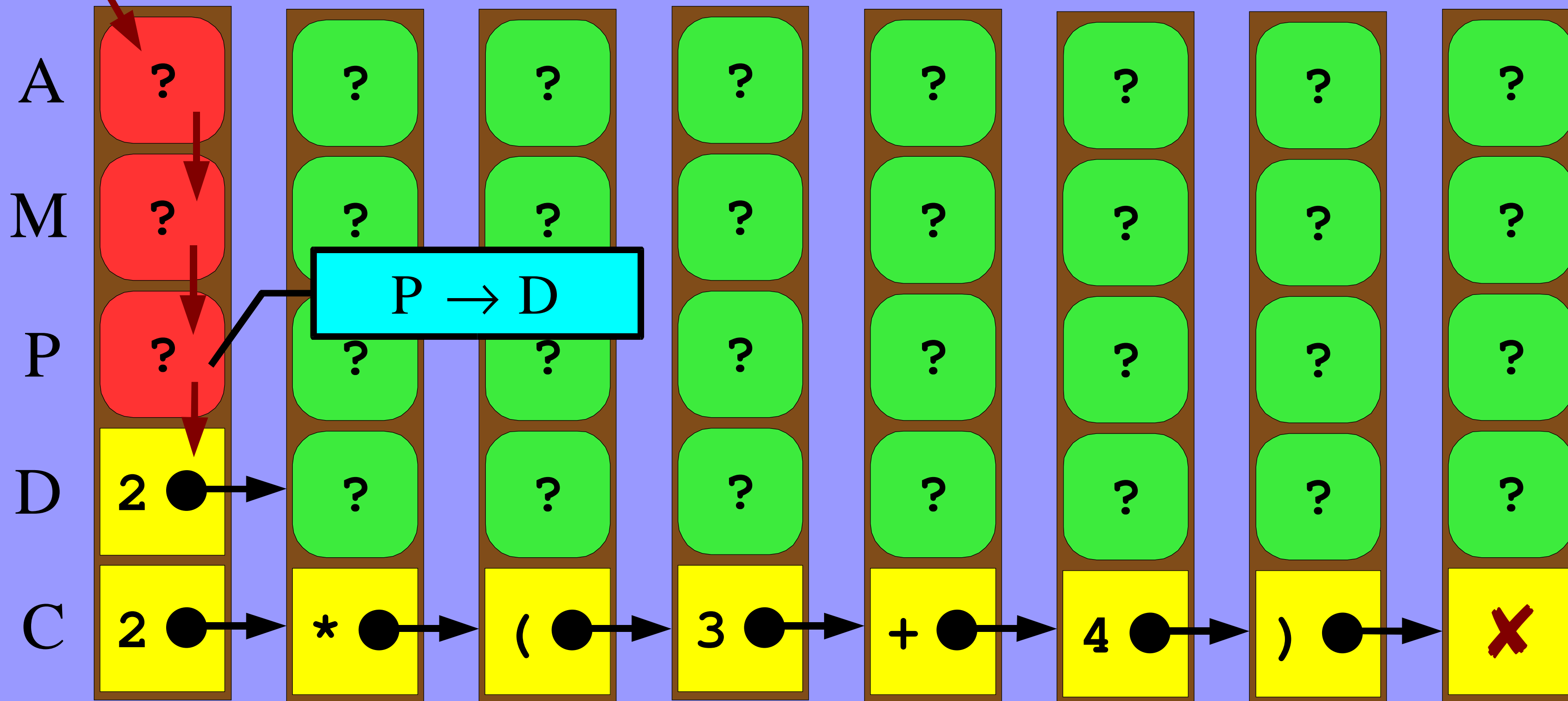
Packrat Parsing Example



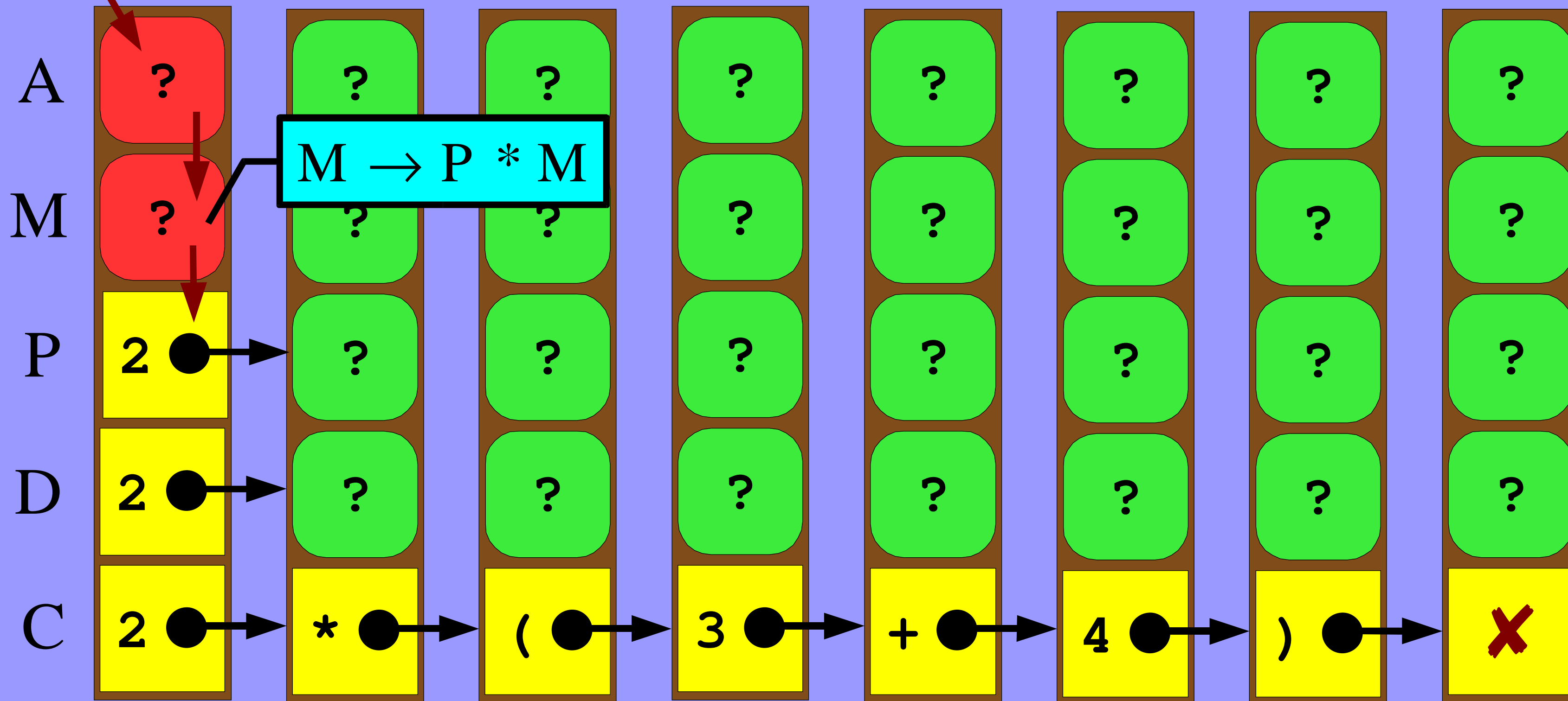
Packrat Parsing Example



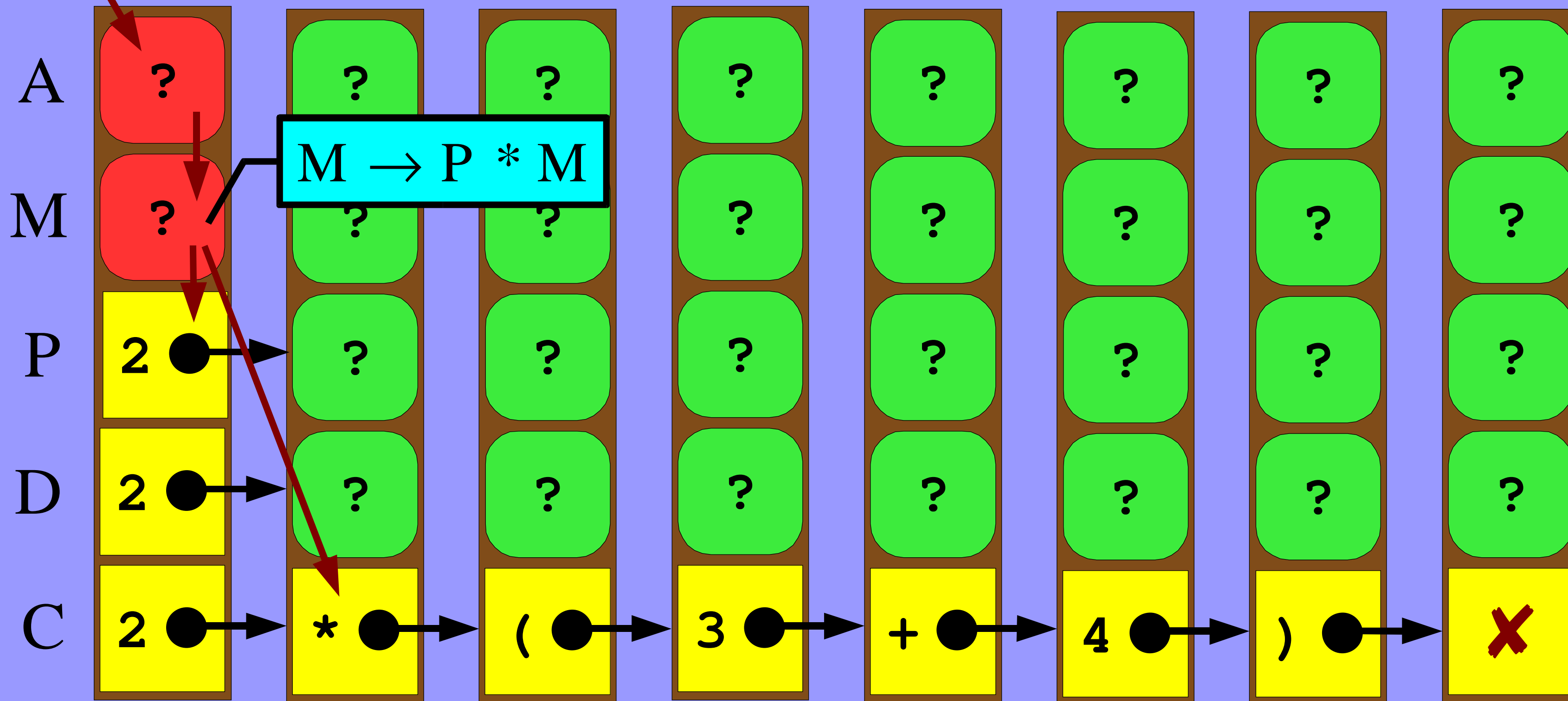
Packrat Parsing Example



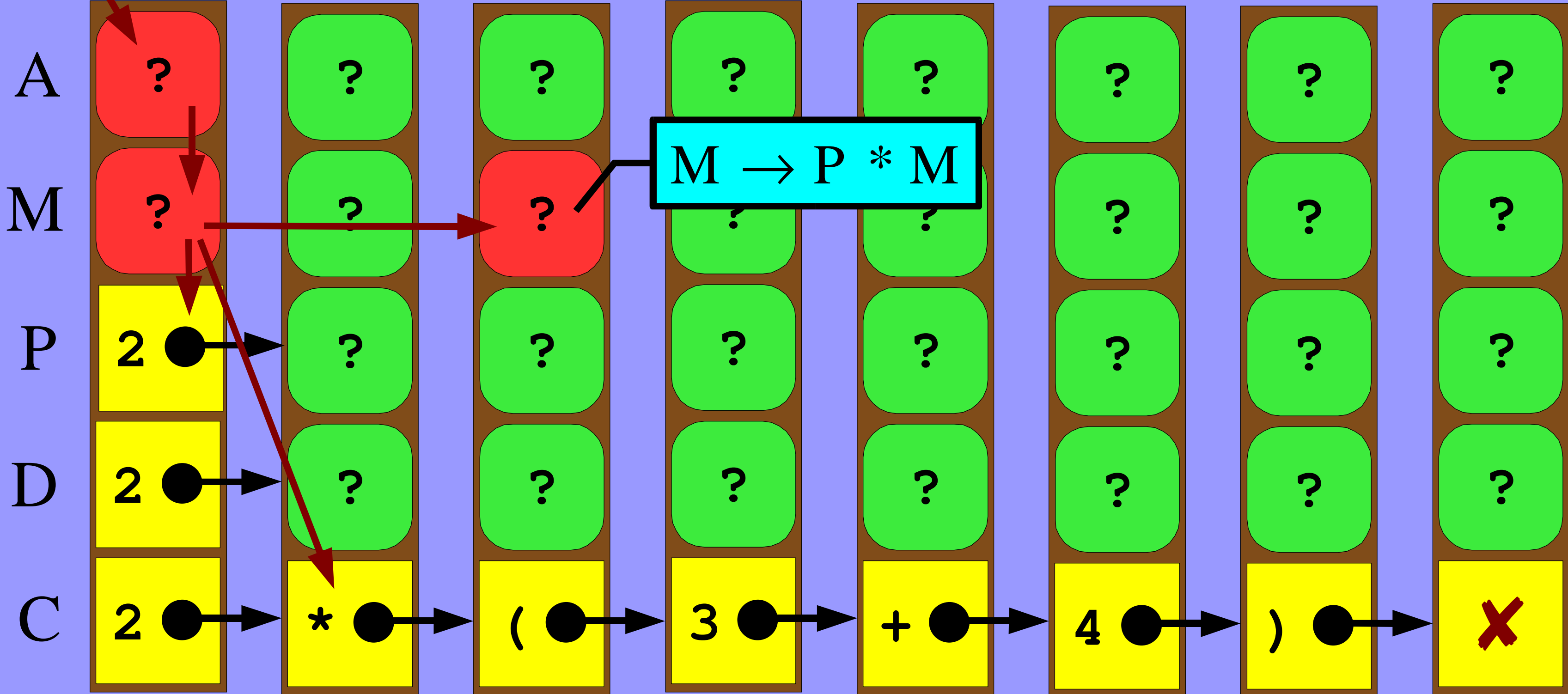
Packrat Parsing Example



Packrat Parsing Example

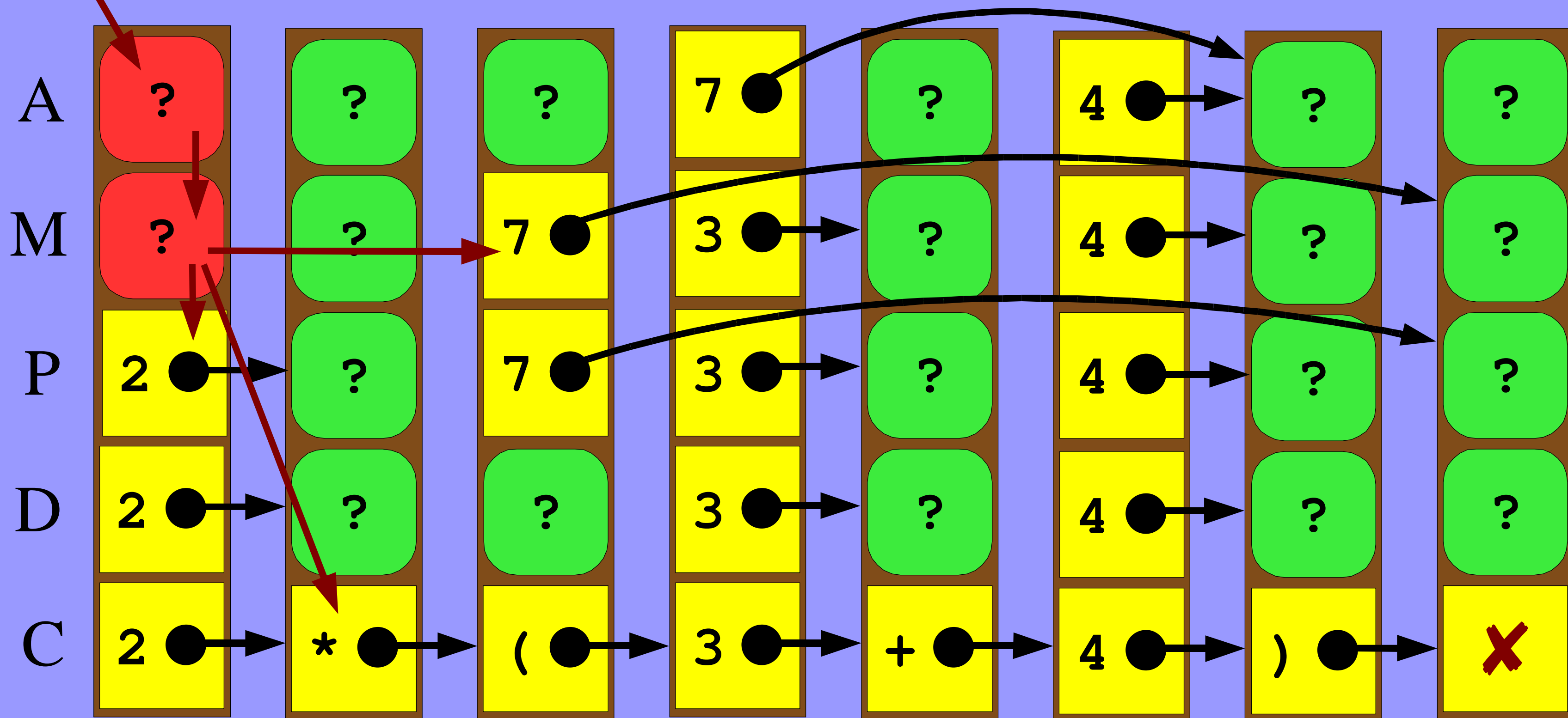


Packrat Parsing Example

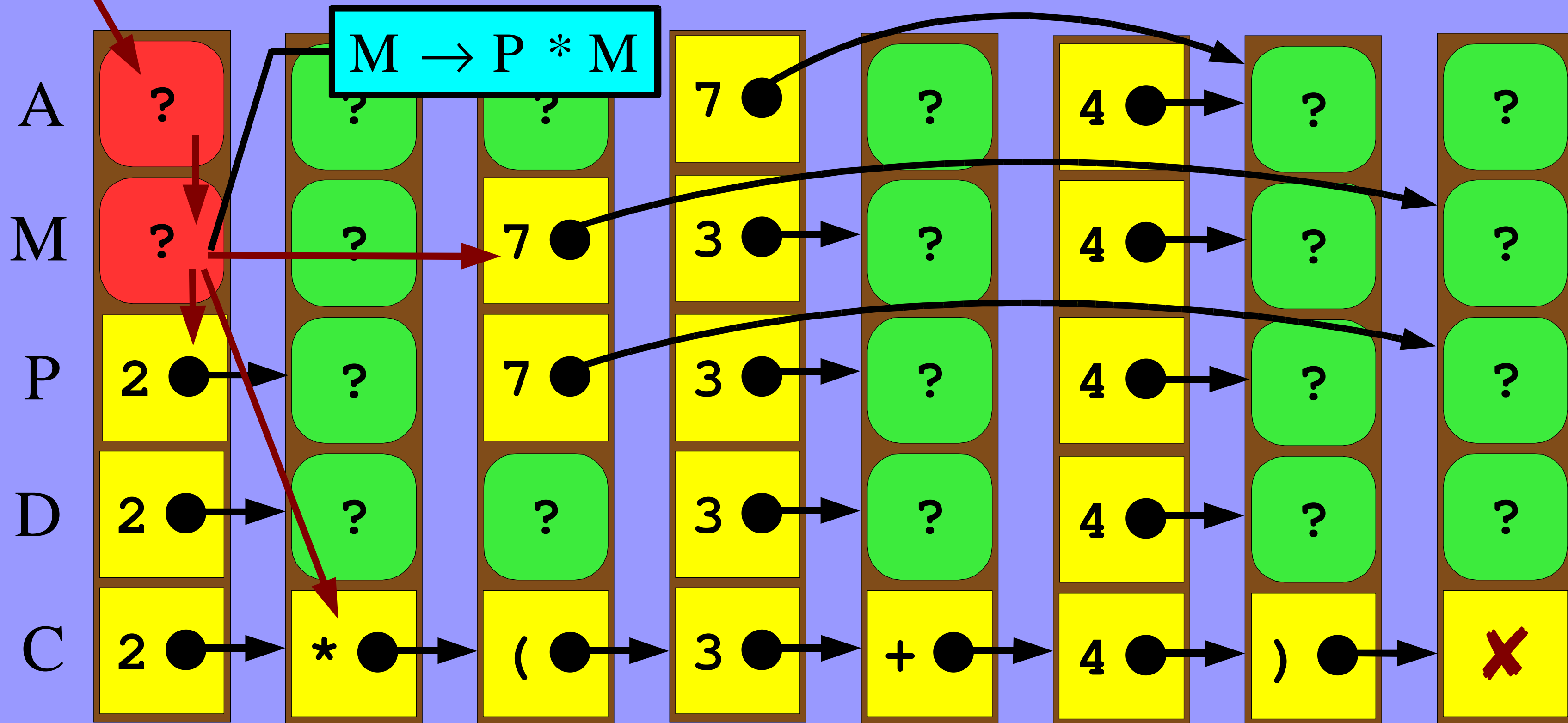


Packrat Parsing Example

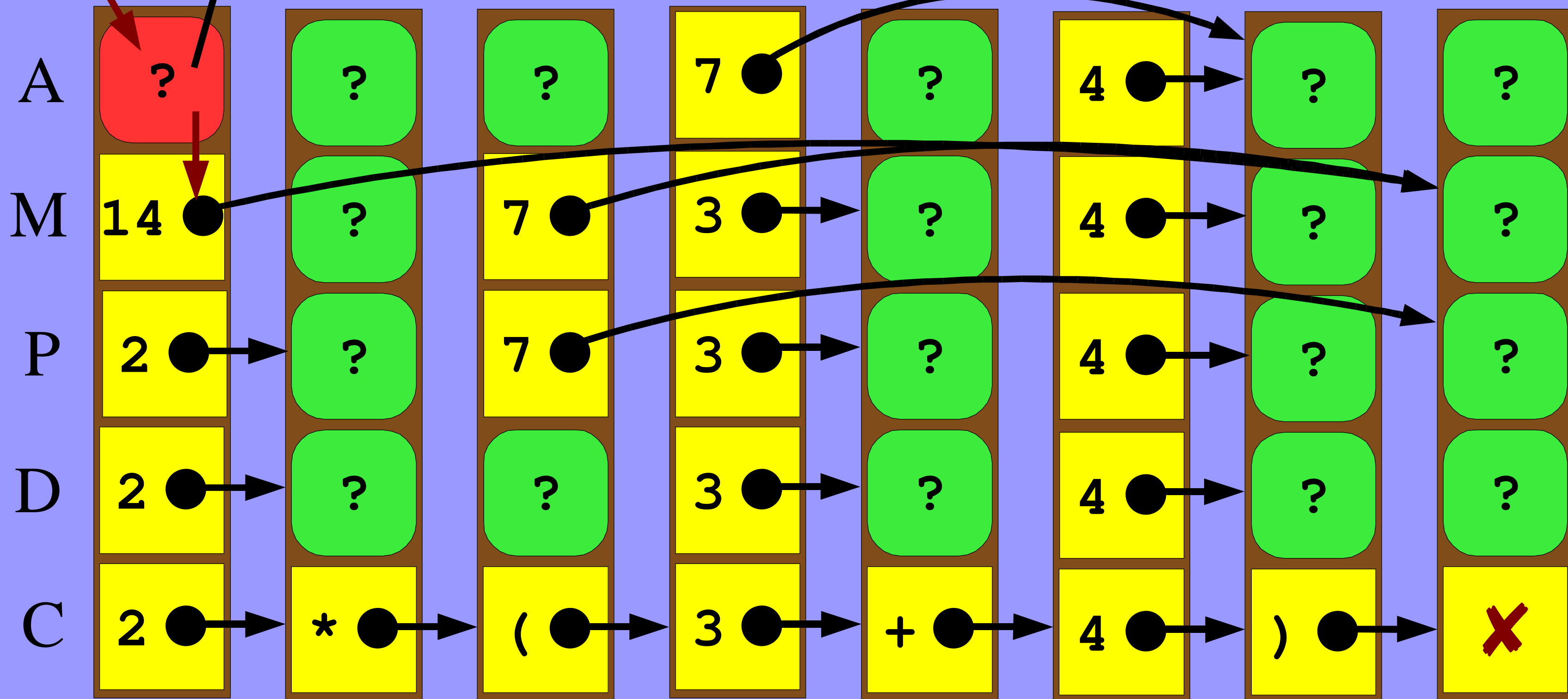
(eval)



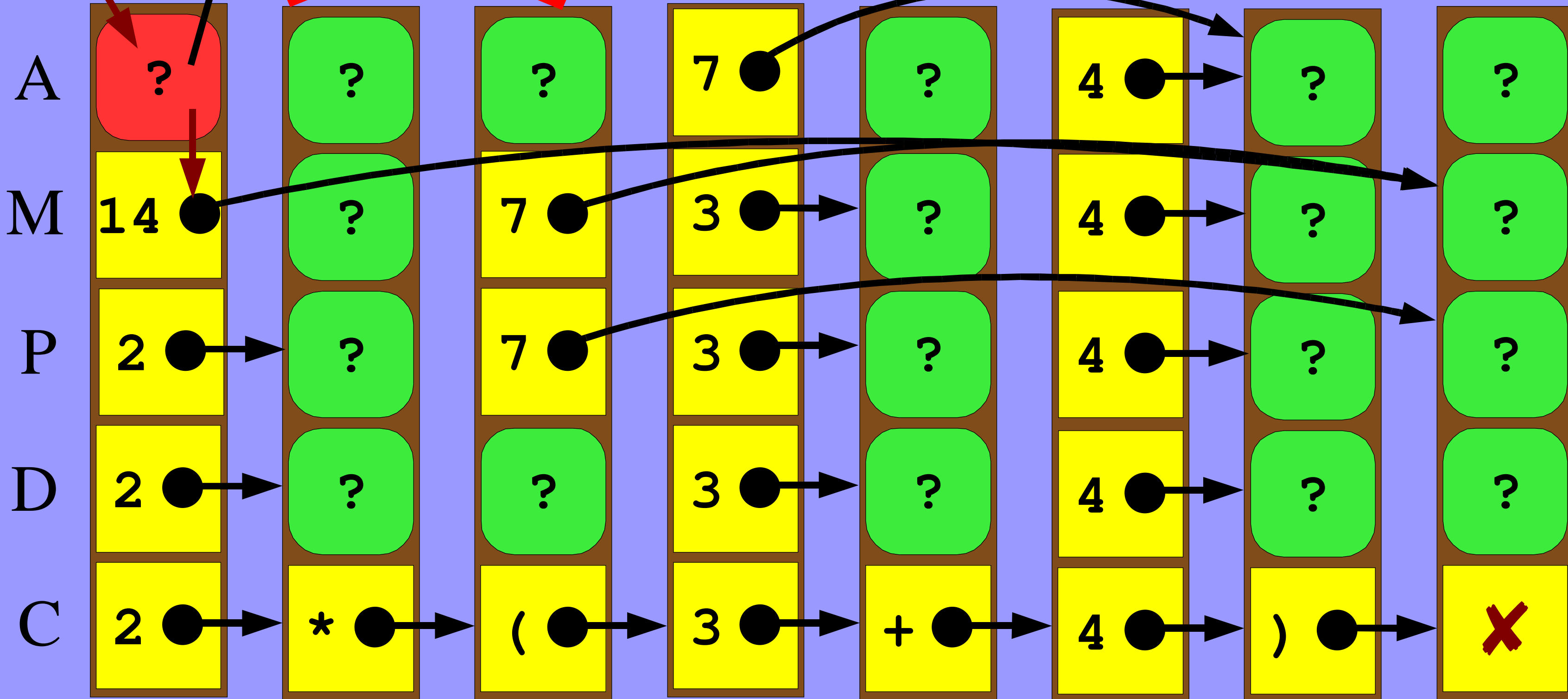
Packrat Parsing Example



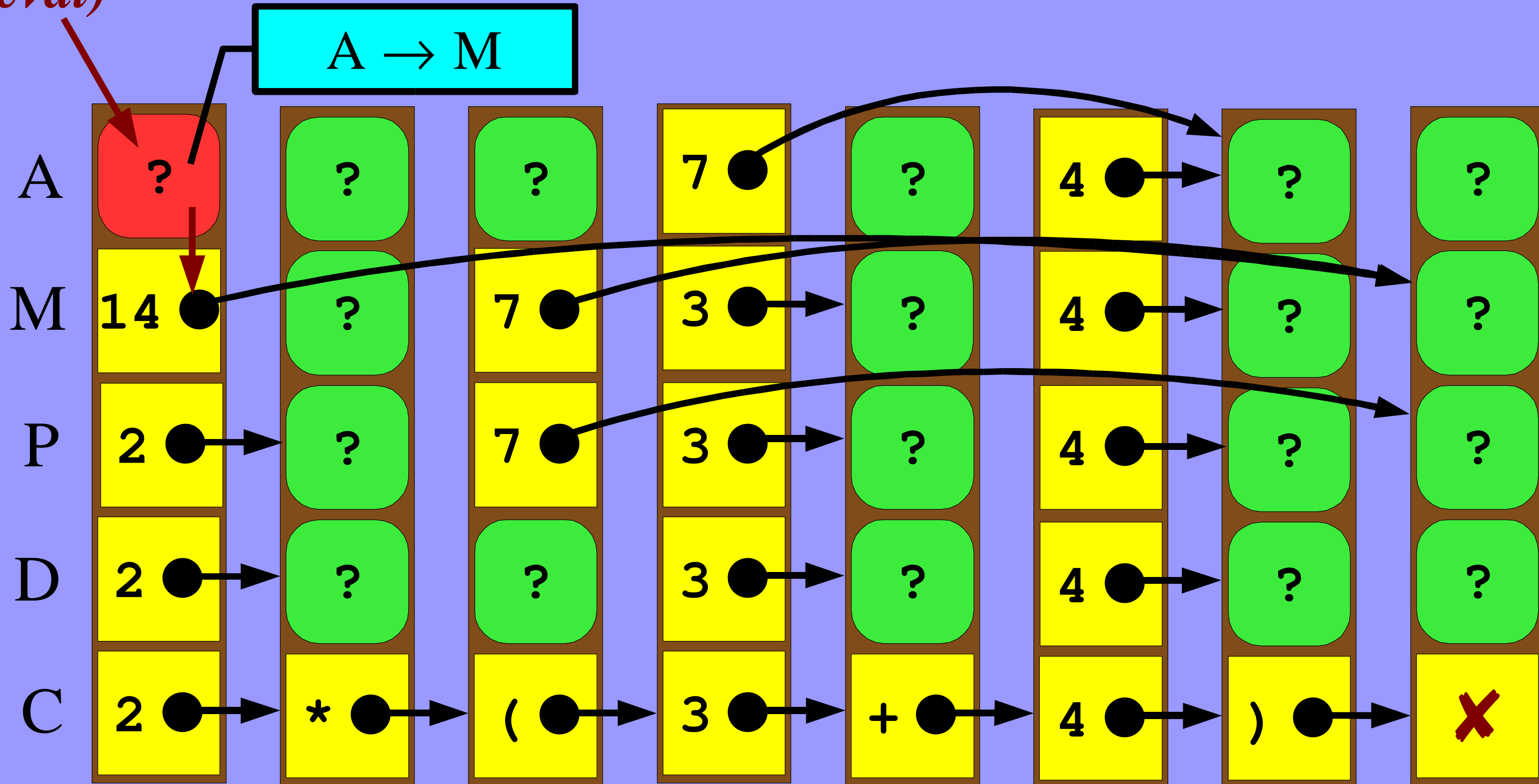
Packrat Parsing Example



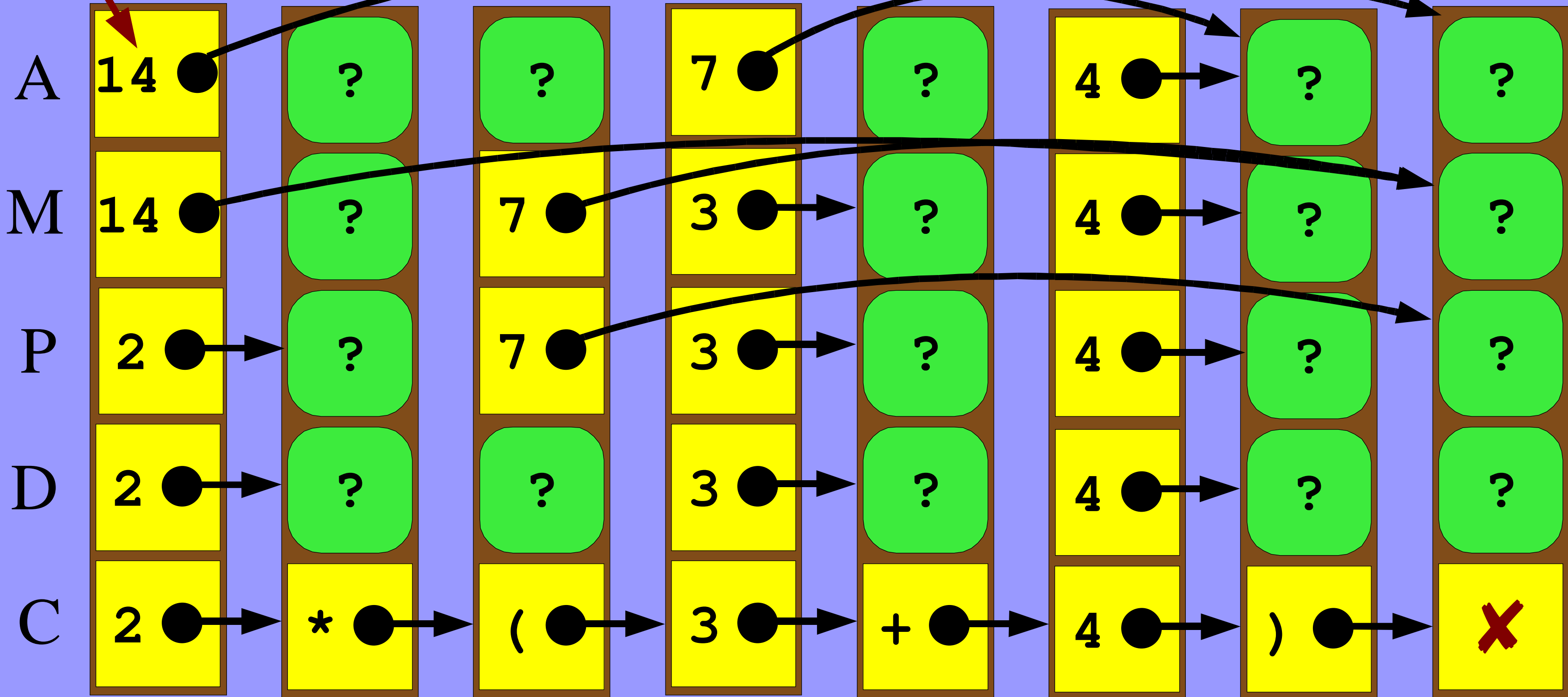
Packrat Parsing Example



Packrat Parsing Example



Packrat Parsing Example



What is Packrat Parsing Good (and not good) For?

Theoretical Properties

- Formally developed by Birman in 1970s
  - Proved existence of linear-time parsing algorithm
  - *...but apparently never implemented*
- Recognizable languages:
  - Strictly larger than deterministic parsing algorithms: e.g., LL(k), LR(k)
  - Incomparable to class of context-free languages

Scannerless Parsing

- Traditional linear-time parsers limited by fixed (e.g., one-token) lookahead
  - If we only have one lookahead token, then it's easier if tokens are *big*.
- Packrat parsers provide unlimited lookahead
  - No longer need to separate lexical analysis
- Why scannerless parsing?
  - Simplicity: unified grammar for entire language
  - Power: lexical elements with complex syntax

Syntactic Flexibility

- Syntactic predicates
  - Parse X only if Y also matches
  - Parse X only if followed by Y
- Subtractive syntax
  - Parse X only if Y doesn't match
  - Parse X only if *not* followed by Y
- Semantic predicates
  - Parse X if its semantic value satisfies condition

Limitations

What is a packrat parser *not* good for?

- General CFG parsing: e.g., ambiguous grammars

  (because of “at-most-one result” limitation)

- Parsing highly “stateful” syntax: e.g., C, C++

  (memoization depends on statelessness)

- Parsing in minimal space

  (LL/LR parser grows with stack depth, not input size)

Practical Parsers

Example packrat parser for the Java language:

- Unified (scannerless) parser
- Implemented in Haskell
- Three versions:
  1. Hand-coded with monadic combinators
  2. Hand-coded with primitive pattern-matching
  3. Automatically built by prototype parser generator

Performance Results (Summary)

Parse Time:

- Reliably linear growth with input size
- 26-52KB/s (600-1200 lines/sec)
  (GHC 5.04, 1.2GHz Athlon)
- Comparable to Happy-generated LR parser
  (faster for average-size Java sources)

- Heap usage:
  - Reliably linear growth with input size
  - 300-600× expansion ratio

Related Work

Functional/ monadic Parsing:

- Wadler, Fokker, Hutton, Meijer, etc.

Scannerless Parsing:

- Tai, Salomon, Cormack – NSLR(1)
  - (linear time, but restrictive of grammar)
- Visser et al. – Generalized-LR
  - (not linear time)

Syntactic & Semantic Predicates:

- Parr, Quong – pred-LL(k)

Conclusion

Packrat parsing:

- Uses memoization to provide backtracking and unlimited lookahead in a linear-time parser
- Is easily expressed as a lazy data structure
- Provides more flexibility than LL or LR parsing
- Enables practical scannerless parsing
- Has substantial storage cost, but often reasonable

For More Information

Papers, Master's Thesis
Prototype Packrat Parser Generator
Source Code for Example Parsers
Test Suite for Example Java Parsers

available at:

`http://pdos.lcs.mit.edu/~baford/packrat/`