Top-Down Parsing
Earley Parsing

And it's gonna be totally awesome!

Motivating Question

- Given this grammar G:
  - **E → E + T**
  - **E → T**
  - **T → T * int**
  - **T → int**
  - **T → (E)**
- Is the string **int * (int + int)** in L(G)?
  - Give a derivation or prove that it is not.

Revenge of Theory

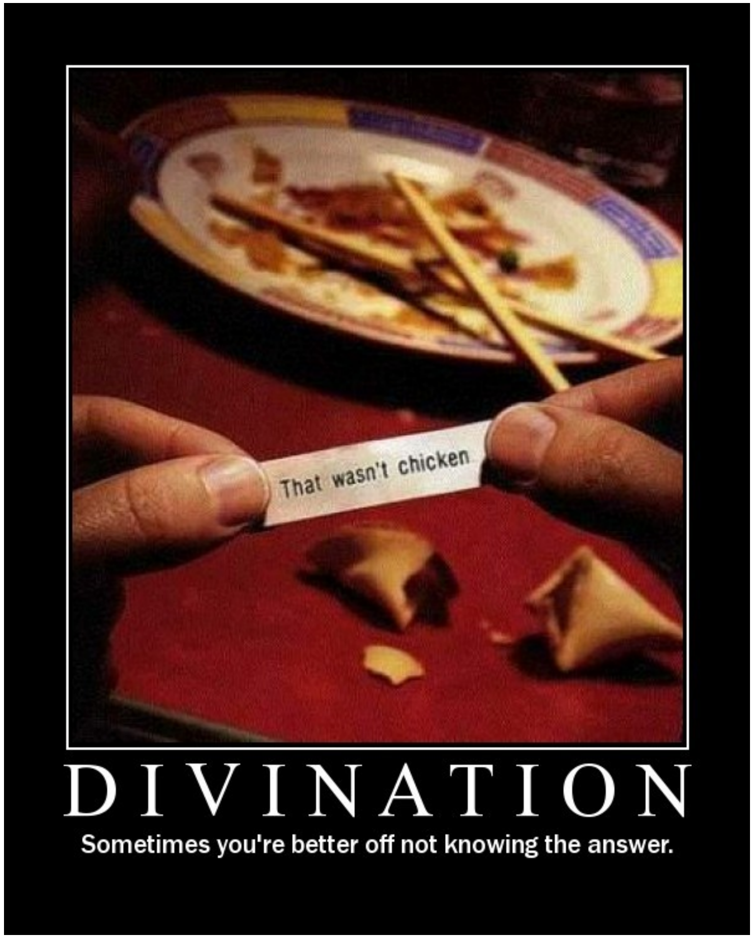
- How do we tell if DFA **P** is equal to DFA **Q**?
  - We can do: “is DFA **P** empty?”
    - How?
  - We can do: “**P** := not **Q**”
    - How?
  - We can do: “**P** := **Q** intersect **R**”
    - How?
  - So do: “is **P** intersect not **Q** empty?”
- Does this work for CFG **X** and CFG **Y**?
- Can we tell if **s** is in CFG **X**?

```
Acquiring Fruit. . . . . . . . . .
Scanning . . . . . . . . . . . . .
Analyzing perimeter:
      ROUND    94.2%
Analyzing color:
     ORANGE    93.7%
Analyzing surface:
     GLOSSY    11.8%

Probability:
     ORANGE 93.231%
```

Outline

- Recursive Descent Parsing
  - Left Recursion
- Historical Approaches
  - LL, LR, LALR
- Dynamic Programming
- Earley's Algorithm
  - Chart States
  - Operations
  - Example

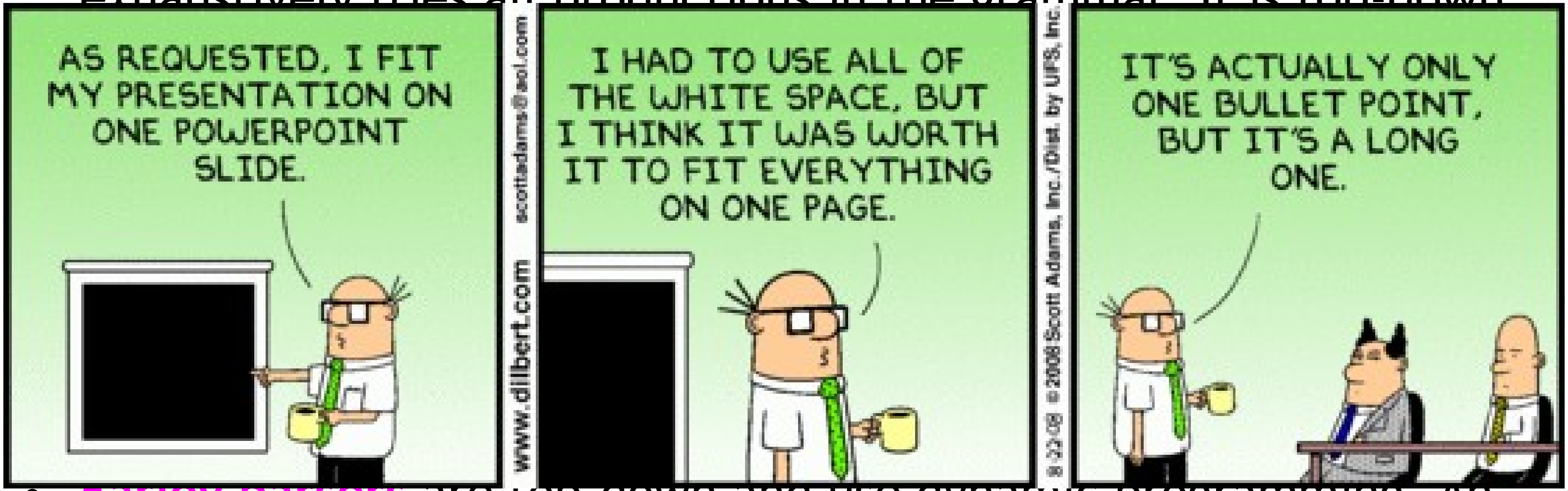


In One Slide

- A **top-down parser** starts to work from the initial grammar rules (rather than from the initial tokens). A **recursive descent parser** exhaustively tries all productions in the grammar. It is top-down, may backtrack and cannot handle left-recursive grammars. Left recursion can be eliminated. Historical approaches such as **LL, LR and LALR** cannot handle all context-free grammars. They can be efficient.
- **Dynamic programming** is a problem-solving technique based on optimal substructure: solutions to subproblems yield solutions to overall problems.
- **Earley parsers** are top-down and use dynamic programming. An Earley state records incremental information: when we started, what has been seen so far, and what we expect to see. The Earley chart holds a set of states for each input position. Shift, reduce and closure operations fill in the chart.
- **You** enjoy parsing. Parsing is easy and fun.

In One Slide

- A **top-down parser** starts to work from the initial grammar rules (rather than from the initial tokens). A **recursive descent parser** exhaustively tries all productions in the grammar. It is top-down



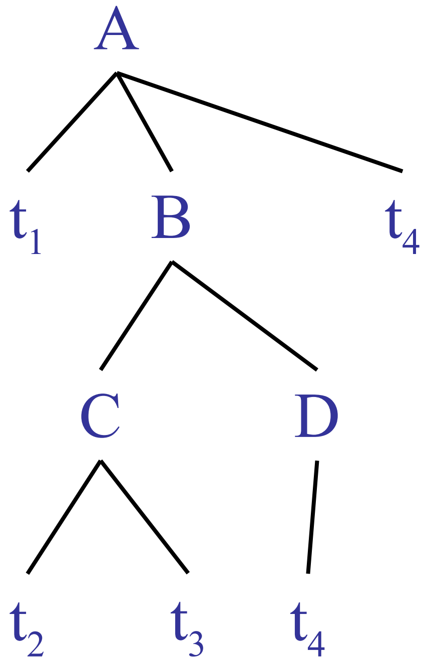
- **Earley parsers** are top-down and use dynamic programming. An Earley state records incremental information: when we started, what has been seen so far, and what we expect to see. The Earley chart holds a set of states for each input position. Shift, reduce and closure operations fill in the chart.
- **You** enjoy parsing. Parsing is easy and fun.

Intro to Top-Down Parsing

- Terminals are seen in order of appearance in the token stream:

  $t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5$

  The parse tree is constructed
  - From the top
  - From left to right



Recursive Descent Parsing

- We’ll try **recursive descent** parsing first
  - “Try all productions exhaustively, backtrack”
- Consider the grammar

$$E \rightarrow T + E \ \mid \ T$$

$$T \rightarrow (E) \ \mid \ \text{int} \ \mid \ \text{int} * T$$

- Token stream is: **int * int**
- Start with top-level non-terminal E

- Try the rules for E in order

Recursive Descent Example

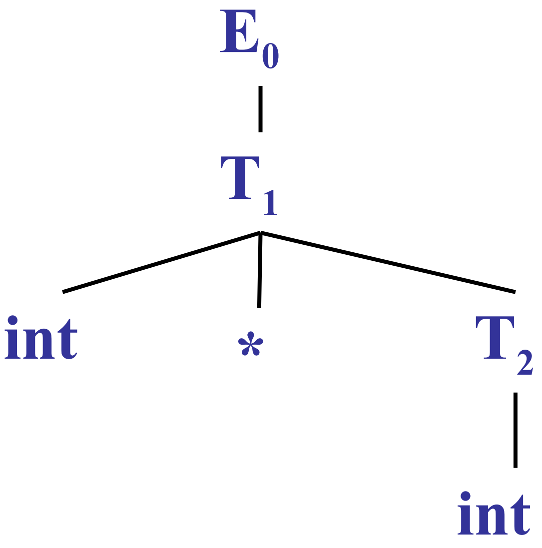
$E \rightarrow T + E \ | \ T$
$T \rightarrow (E) \ | \ int \ | \ int * T$
Input = int * int

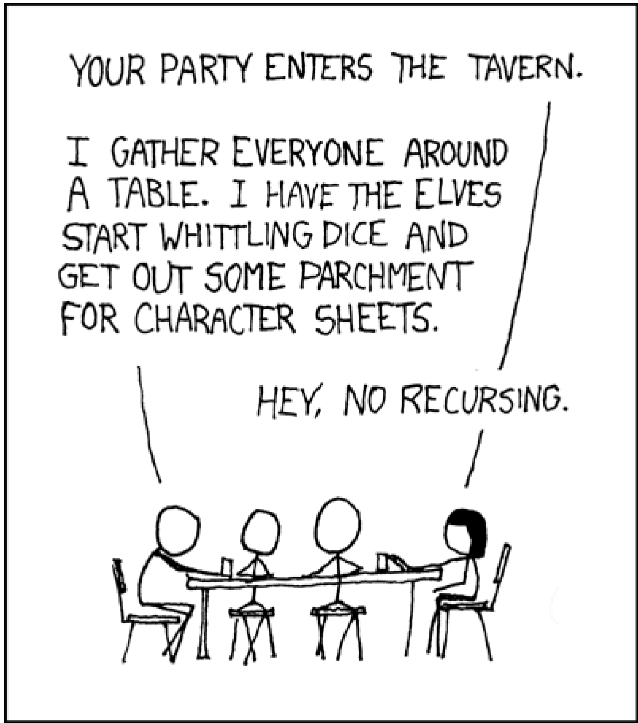
- Try $E_0 \rightarrow T_1 + E_2$
- Then try a rule for $T_1 \rightarrow (E_3)$
  - But (does not match input token int
- Try $T_1 \rightarrow$ int . Token matches.
  - But + after T_1 does not match input token *
- Try $T_1 \rightarrow$ int * T_2
  - This will match but + after T_1 will be unmatched
- Have exhausted the choices for T_1
  - **Backtrack** to choice for E_0

Recursive Descent Example (2)

$E \rightarrow T + E \ | \ T$
$T \rightarrow (E) \ | \ int \ | \ int * T$
Input = int * int

- Try $E_0 \rightarrow T_1$
- Follow same steps as before for T_1
  - And succeed with $T_1 \rightarrow$ int * T_2 and $T_2 \rightarrow$ int
  - With the following parse tree





Recursive Descent Parsing

- Parsing: given a string of tokens $t_1\ t_2\ \ldots\ t_n$, find its parse tree
- **Recursive descent parsing**: Try all the productions exhaustively
  - At a given moment the **fringe** of the parse tree is: $t_1\ t_2 \ldots t_k\ A \ldots$
  - Try all the productions for A: if $A \rightarrow BC$ is a production, the new fringe is $t_1\ t_2 \ldots t_k\ B\ C \ldots$
  - **Backtrack** if the fringe doesn't match the string
  - Stop when there are no more non-terminals

When Recursive Descent Does *Not* Work

- Consider a production S → S a:
    - In the process of parsing S we try the above rule
    - What goes wrong?

The page at http://www.youtube.com says:

something unexpected happened. please smash head against keyboard. when done, you may run around in circles, wave your hands, and scream.

When Recursive Descent Does *Not* Work

- Consider a production $S \to S\ a$:
  - In the process of parsing S we try the above rule
  - What goes wrong?

- A **left-recursive grammar** has

$$S \to^{+} S\alpha \quad \text{for some } \alpha$$

Recursive descent does not work in such cases

- It goes into an infinite loop

What's Wrong With That Picture?

Elimination of Left Recursion

- Consider the left-recursive grammar

$$S \to S\ \alpha \mid \beta$$

- S generates all strings starting with a β and followed by a number of α

- Can rewrite using **right-recursion**

$$S \to \beta\ T$$

$$T \to \alpha\ T \mid \varepsilon$$

Example of Eliminating Left Recursion

- Consider the grammar

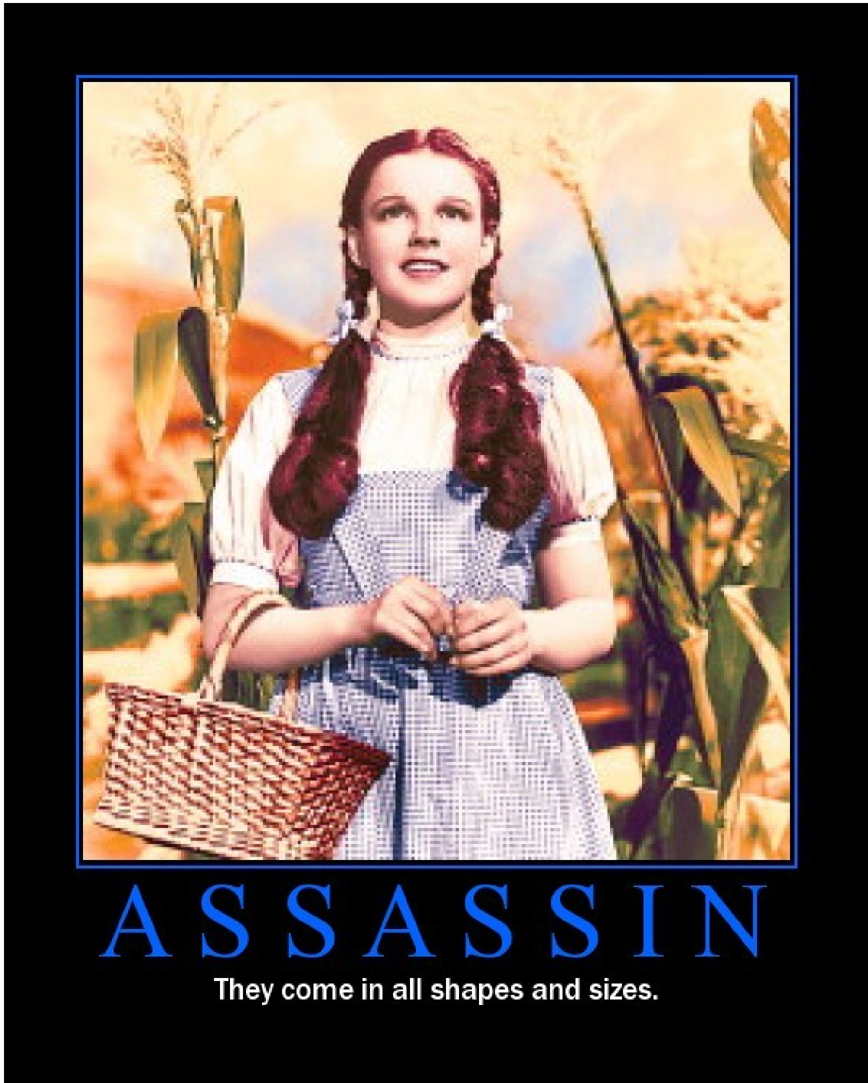
  S → 1 | S 0

  (β = 1 and α = 0)

  It can be rewritten as

  S → 1 T

  T → 0 T | ε



More Left Recursion Elimination

- In general

$$S \to S\,\alpha_1 \mid \ldots \mid S\,\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

- All strings derived from S start with one of $\beta_1,\ldots,\beta_m$ and continue with several instances of $\alpha_1,\ldots,\alpha_n$
- Rewrite as

$$S \to \beta_1\,T \mid \ldots \mid \beta_m\,T$$

$$T \to \alpha_1\,T \mid \ldots \mid \alpha_n\,T \mid \varepsilon$$

General Left Recursion

- The grammar

$$S \to A\,\alpha \mid \delta$$

$$A \to S\,\beta$$

is also left-recursive because

$$S \to^{+} S\,\beta\,\alpha$$

- This left-recursion can also be eliminated
- See book, Section 2.3
- Detecting and eliminating left recursion are *popular test questions*

Summary of Recursive Descent

- Simple and general parsing strategy
  - Left-recursion must be eliminated first
  - … but that can be done automatically
- Unpopular because of backtracking
  - Thought to be too inefficient (repetition)
- We can avoid backtracking
  - Sometimes ...

Sometimes Things Are Perfect

- The “.ml-lex” format you emit in PA2
- Will be the input for PA3
  - actually the *reference* “.ml-lex” will be used
- It can be “parsed” directly
  - You always know just what to do next
- Ditto with the “.ml-ast” output of PA3
- Just write a few mutually-recursive functions
- They read in the input, one line at a time

Historical Approaches

- In the past, I/O was slow and memory was small. Many sacrifices were made to optimize parsing.
  - Basic idea: “If we don't handle all grammars, we can go faster on simpler grammars.” Also: table → no backtrack.
- **LL(*k*)** – Left to right scan of input, Leftmost derivation, predict using *k* tokens. Parse by making a table.
- **LR(*k*)** – Left to right scan of input, Rightmost derivation, predict using *k* tokens. Parse by making a table.
- **LALR(*k*)** – Left to right scan of input, Rightmost derivation, predict using *k* tokens. Parse by making a table, but merge some states in that table. Yacc, bison, etc. use LALR(1).

The Sacrifice

- **LL(1) languages** can be LL(1) parsed
  - A language Q is LL(1) if there exists an LL(1) table such the LL(1) parsing algorithm using that table accepts exactly the strings in Q
  - Essentially, the table has to be perfect: no entry can be ambiguous or multiply defined.
- Sadly, LL(1) != CFG.
- Sadly, LR(1) != CFG.
- Sadly, LALR(1) != CFG.
  - See textbook for definitive Venn diagram.

The Sacrifice

- **LL(1)**
  - A la
    tabl that
    tabl
  - Esse entry
    can

- Sadly, LL(1) != CFG.
- Sadly, LR(1) != CFG.
- Sadly, LALR(1) != CFG.
  - See textbook for definitive Venn diagram.

Q: Books (727 / 842)

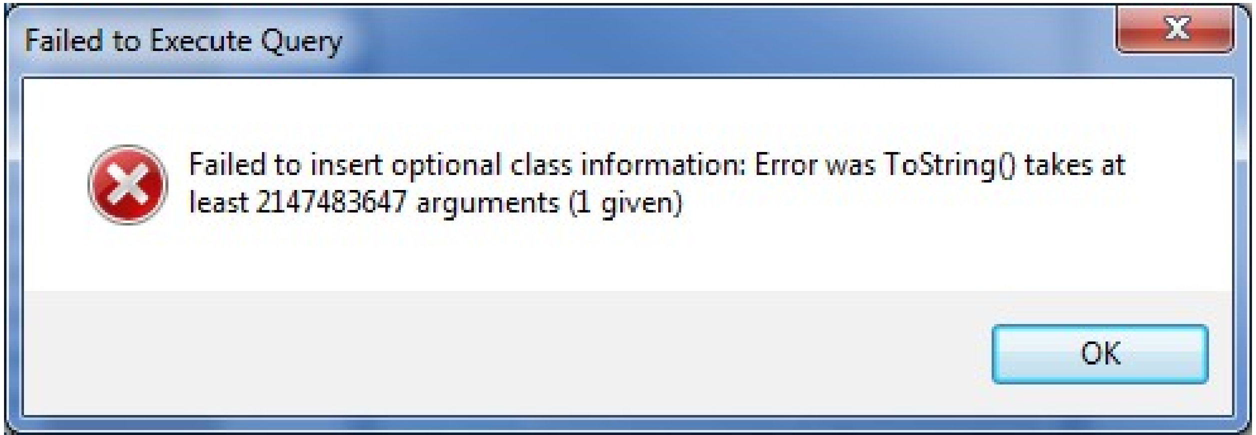
- Name 5 of the 9 major characters in A. A. Milne's 1926 books about a *"bear of very little brain*" who composes poetry and eats honey.

Dynamic Programming

- "Program" = old word for "table of entries"
  - cf. the program (= "schedule") at a concert
- **Dynamic Programming** means
  - "Fill in a chart or table at run-time."
  - Same idea as memoization.
- Works when the problem has the **optimal substructure** property
  - Solution to Big Problem can be constructed from solutions to Smaller Subproblems.
- Shortest path, spell checking, ...

Simple Dynamic Programming

- Dynamic Programming for Fibonacci
  - N 1 2 3 4 5 6
  - Chart 1 1 2 3 5 8
  - chart[N] = chart[N-1] + chart[N-2]
  - Reduces runtime of Fibo from Bad to Linear



Failed to insert optional class information: Error was ToString() takes at least 2147483647 arguments (1 given)

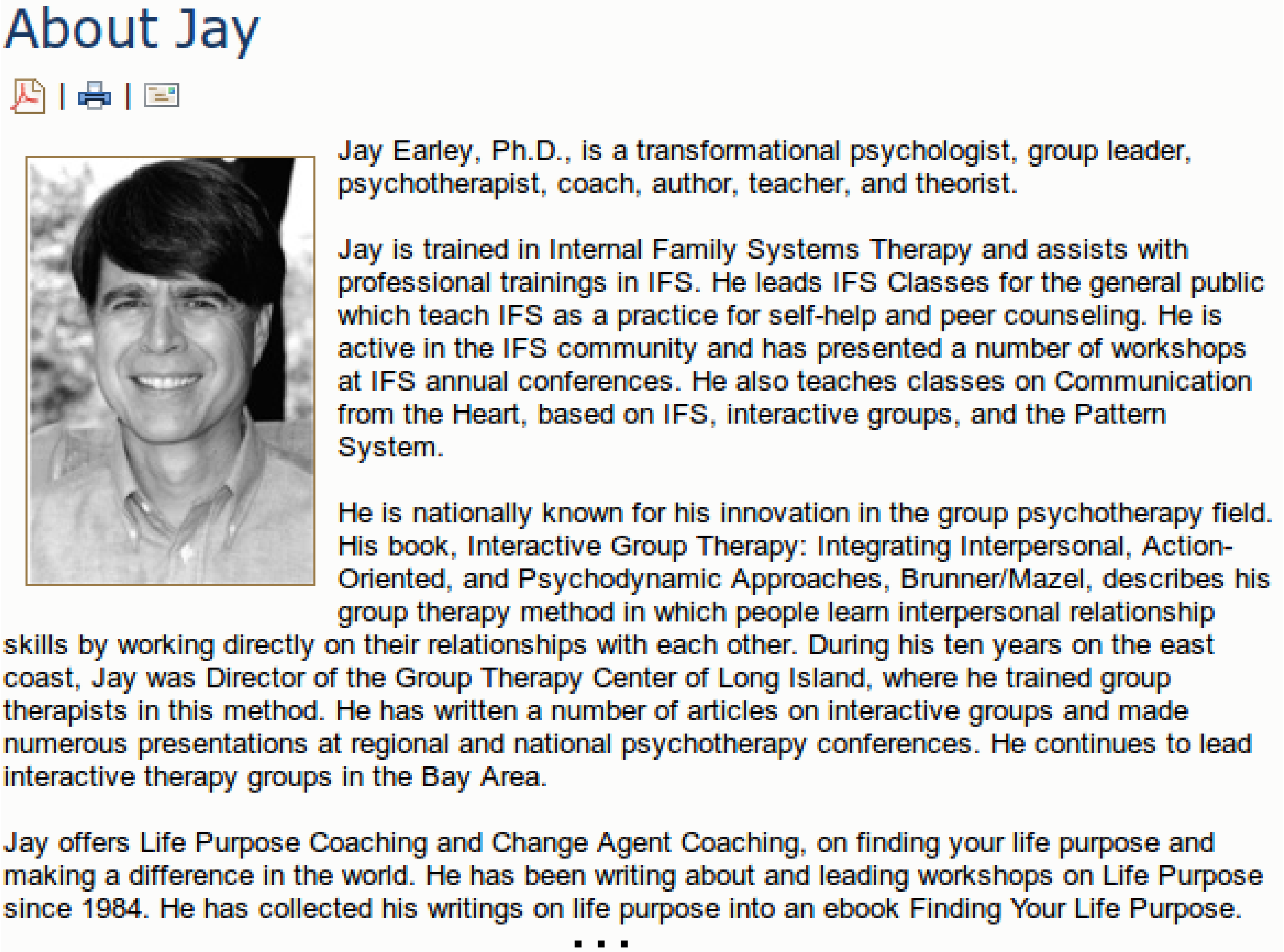
Dynamic Programming for Parsing

- Dynamic Programming for Fibonacci
  - N 1 2 3 4 5 6
  - Chart 1 1 2 3 5 8
  - chart[N] = chart[N-1] + chart[N-2]
  - Reduces runtime of Fibo from Bad to Linear
- Dynamic Programming for Parsing
  - N 1 2 3 4 5 6
  - Tokens x + (y * z
  - Chart < *I'll explain in a minute* >
  - chart[N] = “list of possible parses of tokens 1...N”

Earley's Algorithm

- **Earley's Algorithm** is an incremental left-to-right top-down parsing algorithm that works on arbitrary context-free grammars.

- Systematic Hypothesis Testing
  - Before processing token #X, it has already considered all hypotheses consistent with tokens #1 to #X-1.
  - Substructure: parsing token #X is solved in terms of parsing sequences of tokens from #1 to #X-1



About Jay

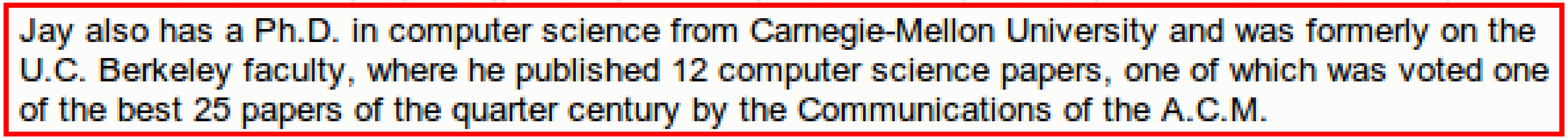
Jay Earley, Ph.D., is a transformational psychologist, group leader, psychotherapist, coach, author, teacher, and theorist.

Jay is trained in Internal Family Systems Therapy and assists with professional trainings in IFS. He leads IFS Classes for the general public which teach IFS as a practice for self-help and peer counseling. He is active in the IFS community and has presented a number of workshops at IFS annual conferences. He also teaches classes on Communication from the Heart, based on IFS, interactive groups, and the Pattern System.

He is nationally known for his innovation in the group psychotherapy field. His book, Interactive Group Therapy: Integrating Interpersonal, Action-Oriented, and Psychodynamic Approaches, Brunner/Mazel, describes his group therapy method in which people learn interpersonal relationship skills by working directly on their relationships with each other. During his ten years on the east coast, Jay was Director of the Group Therapy Center of Long Island, where he trained group therapists in this method. He has written a number of articles on interactive groups and made numerous presentations at regional and national psychotherapy conferences. He continues to lead interactive therapy groups in the Bay Area.

Jay offers Life Purpose Coaching and Change Agent Coaching, on finding your life purpose and making a difference in the world. He has been writing about and leading workshops on Life Purpose since 1984. He has collected his writings on life purpose into an ebook Finding Your Life Purpose.

. . .



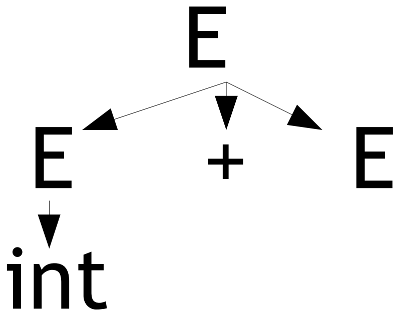
Jay also has a Ph.D. in computer science from Carnegie-Mellon University and was formerly on the U.C. Berkeley faculty, where he published 12 computer science papers, one of which was voted one of the best 25 papers of the quarter century by the Communications of the A.C.M.

Parsing State

- Earley is incremental and left-to-right
- Consider: E → E + E | E * E | (E) | int
- We use a "•" to mark "where we are now"
- Example: E → E + • E
  - I am trying to parse E → E + E
  - I have already seen E + (before the dot)
  - I expect to see E (after the dot)
- General form: X → a • b
  - a, b: sequences of terminals and non-terminals

Dot Example

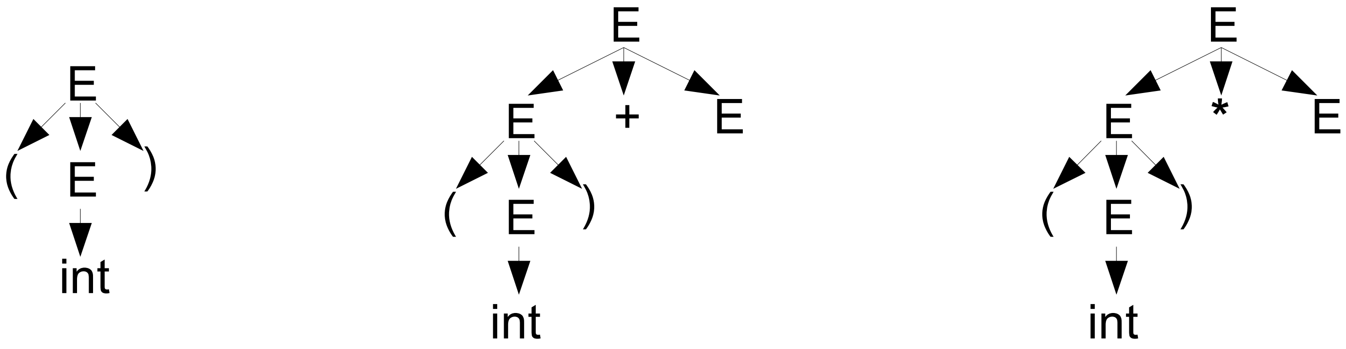
- E → E + E | E * E | (E) | int



- Input so far: int +
- Current state: E → E + • E

Dotty Example

- E → E + E | E * E | (E) | int
- Input so far: (int)
- State could be: E → (E) •



- But also: E → E • + E
- Or: E → E • * E

Origin Positions

- E → E + E | E * E | (E) | int
- Example Input-So-Far #1: int
  - Possible Current State: E → int •
- Example Input-So-Far #2: int + int
  - Possible Current State: E → int •
- Must track the input position just before the matching of this production began.

Origin Positions

- E → E + E | E * E | (E) | int
- Example Input-So-Far #1: int
  - Possible Current State: E → int • , from 0
- Example Input-So-Far #2: int + int
  - Possible Current State: E → int • , from 2
- Must track the input position just before the matching of this production began.

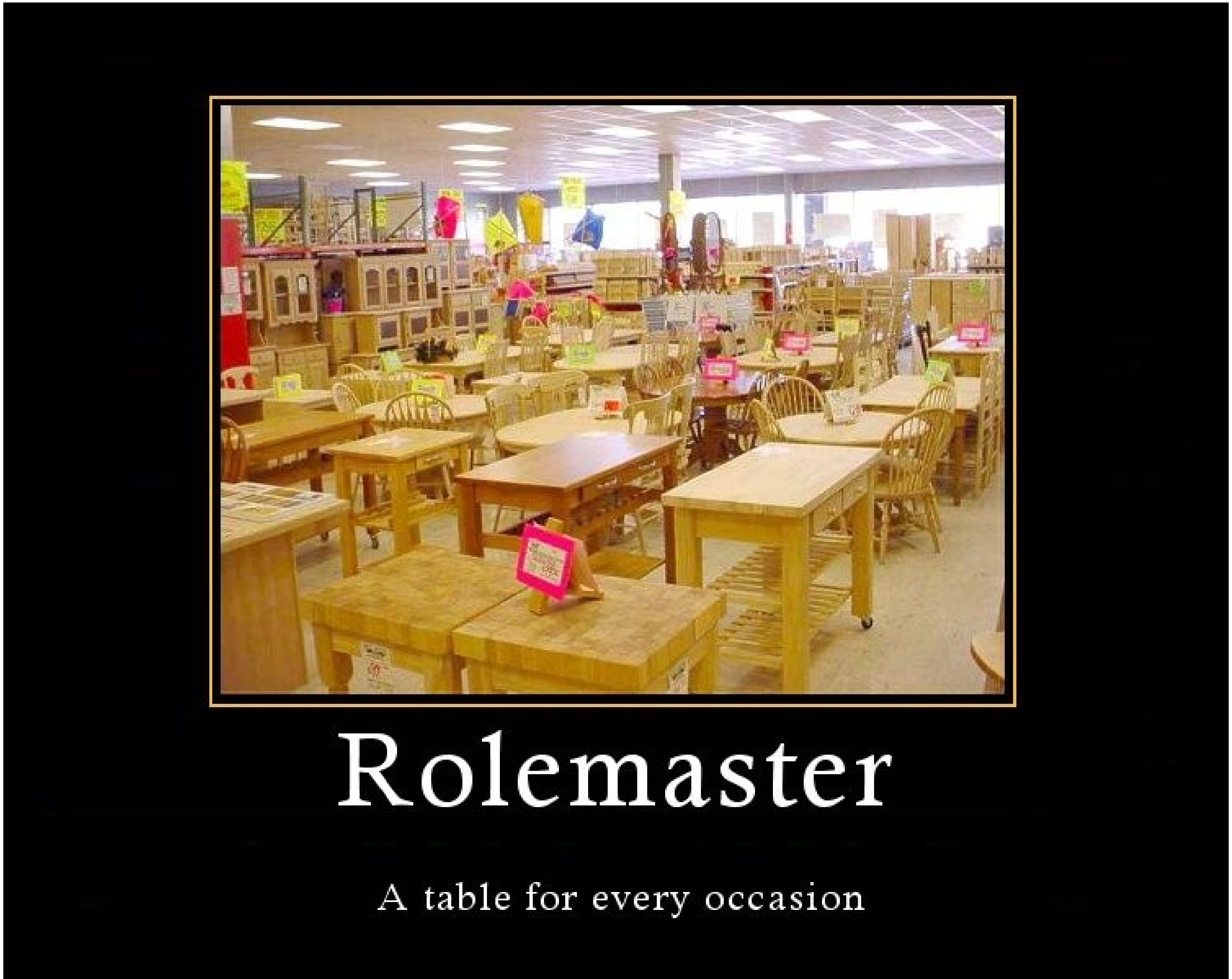
int + int

0 1 2 3

Earley States

- Let X be a non-terminal
- Let a and b be (possibly-empty) sequences of terminals and non-terminals
- Let $X \rightarrow ab$ be a production in your grammar
- Let j be a position in the input
- Each **Earley State** is a tuple $< X \rightarrow a \bullet b \; , j >$
  - We are currently parsing an X
  - We have seen a, we expect to see b
  - We started parsing this X after seeing the first j tokens from the input.

Introducing: Parse Tables



Earley Parse Table (= Chart)

- An **Earley parsing table** (or **chart**) is a one-dimensional array. Each array element is a set of Earley states.
  - chart[i] holds the set of valid parsing states we could be in after seeing the first *i* input tokens

Earley Parse Table (= Chart)

- An **Earley parsing table** (or **chart**) is a one-dimensional array. Each array element is a set of Earley states.
  - chart[i] holds the set of valid parsing states we could be in after seeing the first *i* input tokens
- Then the string $tok_1 \ldots tok_n$ is in the language of a grammar with start symbol S *iff*
  - chart[n] contains < S → ab• , 0 > for some production rule S → ab in the grammar.
  - We then say the parser **accepts** the string.

Earley Parsing Algorithm

- Input:
  - Grammar G (start symbol S, productions X → ab)
  - Input Tokens $tok_1 \ldots tok_n$
- Work:

```
chart[0] = { < S → •ab , 0 > }
for i = 0 to n
  complete chart[i] using G and chart[0]...chart[i]
```

- Output:
  - true iff < S → ab• , 0 > in chart[n]

Filling In The Chart

- Three operations build up chart[n]
- The first is called **shift** or **scan**.
  - It corresponds to “seeing the next expected token” or “helping to confirm the current hypothesis” or “we're winning”.
- Example:
  - chart[1] contains < E → E • + E , 0 >
  - 2nd token is “+”
  - Then put < E → E + • E , 0 > in chart[2]

Formal shift operation

- Whenever
  - chart[i] contains < X → ab • cd , j >
  - c is a terminal (*not* a non-terminal)
  - the (i+1)th input token is c
- The **shift** operation
  - Adds < X → abc • d , j > to chart[i+1]

Filling In The Chart (2)

- The second operation is the **closure** or **predictor**.
  - It corresponds to "expanding rewrite rules" or "substituting in the definitions of non-terminals"
- Suppose the grammar is:

  S → E   E → E + E | E – E | int
- If chart[0] has < S → • E , 0 > then add

  < E → • E + E , 0 >

  < E → • E – E , 0 >

  < E → • int, 0 >

Formal closure operation

- Whenever
  - chart[i] contains < X → ab • cd , j >
  - c is a non-terminal
  - The grammar contains < c → p q r >
- The **closure** operation
  - Adds < c → • p q r , i > to chart[i]

  - Note < c → • p q r , i > because "we started parsing this *c* after seeing the first *i* tokens from the input."

Filling In The Chart (3)

- The third operation is **reduction** or **completion**.
  - It corresponds to “finishing a grammar rewrite rule” or “being done parsing a non-terminal” or “doing a rewrite rule in reverse and then shifting over the non-terminal”.
- Suppose:
  - E → int | E + E | E – E | (E) , input is “(int”
  - chart[2] contains < E → int • , 1 >
  - chart[1] contains < E → (• E) , 0 >
  - Then chart[2] += < E → (E •) , 0 >

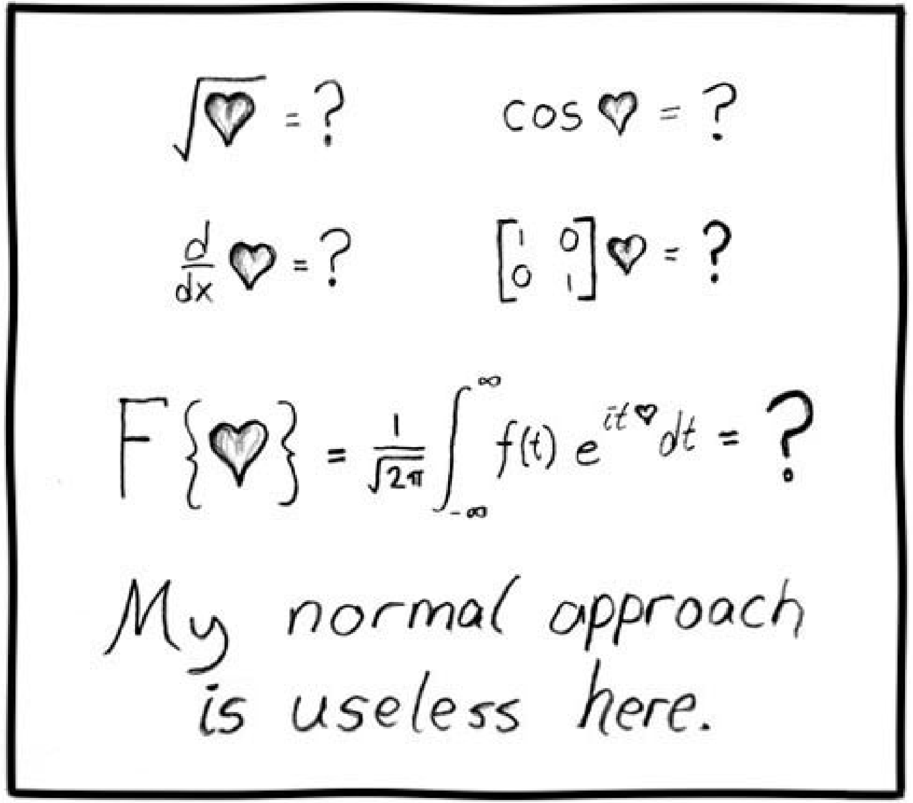
Formal reduce operation

- Whenever
  - chart[i] contains < X → ab • , j >

    (The dot must be all the way to the right!)
  - chart[j] contains < Y → q • X r , k >
- The **reduce** operation
  - Adds < Y → q X • r , k > to chart[i]
  - Note < Y → q X • r , k > because "we started parsing this *Y* after seeing the first *k* tokens from the input."

This is easy and fun.

- This is not as hard as it may seem.



- Let's go practice it!

Shift Practice

- chart[3] contains

  < S → E • , 0 >

  < E → E • + E , 0>

  < E → E • - E , 2>

  < E → int • , 2 >

  < E → E • - E , 0 >

  < E → E - E • , 0 >

  < E → E • + E , 2 >

- The 4th token is “+”. What does shift bring in?

Shift Practice

- chart[3] contains

  < S → E • , 0 >

  < E → E • + E , 0>

  < E → E • - E , 2>

  < E → int • , 2 >

  < E → E • - E , 0 >

  < E → E - E • , 0 >

  < E → E • + E , 2 >

- The 4th token is “+”. What does shift bring in?

  < E → E + • E , 0>

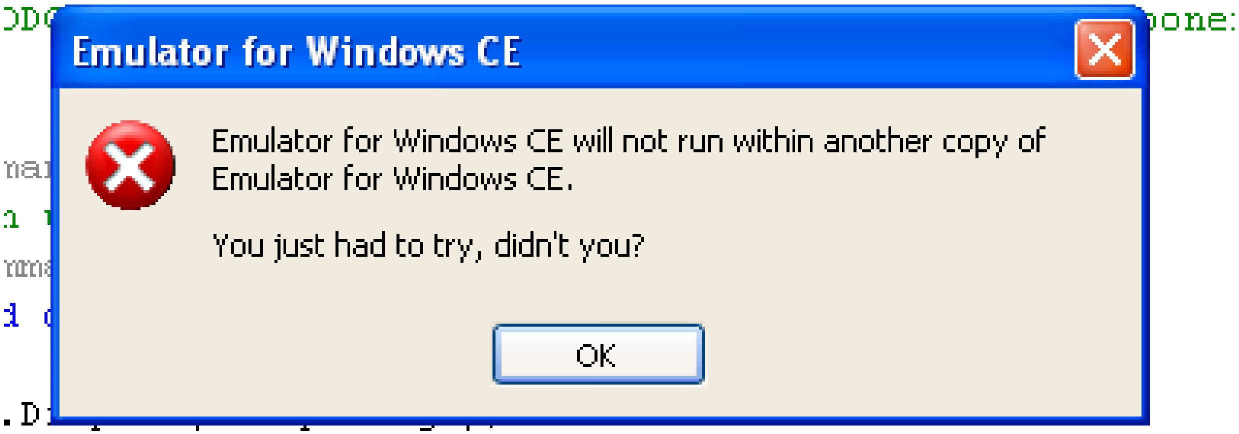
  < E → E + • E , 2 >

  … are both added to chart[4].

Closure Practice

- Grammar is
  - $S \to E$ $E \to E + E \mid E - E \mid (E) \mid int$
- chart[4] contains:

  $< E \to E + \bullet E , 0 > < E \to E + \bullet E , 2 >$
- What does the closure operation bring in?



Emulator for Windows CE

Emulator for Windows CE will not run within another copy of Emulator for Windows CE.

You just had to try, didn't you?

Closure Practice

- Grammar is
  - S → E   E → E + E | E – E | (E) | int
- chart[4] contains:

  < E → E + • E , 0 >< E → E + • E , 2 >
- What does the closure operation bring in?

  < E → • E + E , 4 >< E → • E – E , 4 >

  < E → • (E) , 4 >   < E → • int , 4 >

  … are all added to chart[4].

Reduction Practice

- chart[4] contains:

  < E → E + • E , 0 >< E → E + • E , 2 >

  < E → • E + E , 4 >< E → • E – E , 4 >

  < E → • (E) , 4 >   < E → • int , 4 >
- chart[5] contains:
  - < E → int • , 4 >
- What does the reduce operator bring in?

Reduction Practice

- chart[4] contains:

  < E → E + • E , 0 >< E → E + • E , 2 >

  < E → • E + E , 4 >< E → • E – E , 4 >

  < E → • (E) , 4 >   < E → • int , 4 >

- chart[5] contains:
  - < E → int • , 4 >
- What does the reduce operator bring in?

  < E → E + E • , 0 >< E → E + E • , 2 >

  < E → E • + E , 4 >< E → E • – E , 4 >

  - … are all added to chart[5]. (Plus more in a bit!)

Earley Parsing Algorithm

- Input: CFG G, Tokens $tok_1 \ldots tok_n$
- Work:

```
chart[0] = { < S → •ab , 0 > }
for i = 0 to n
  repeat
    use shift, reduce and closure on chart[i]
  until no new states are added
```

- Output:
  - true iff < S → ab• , 0 > in chart[n]

Massive Earley Example

Grammar

S → F

F → id (A)

A → N

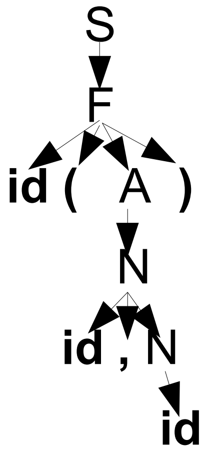
A → ε

N → id

N → id , N

Input

id (id , id)



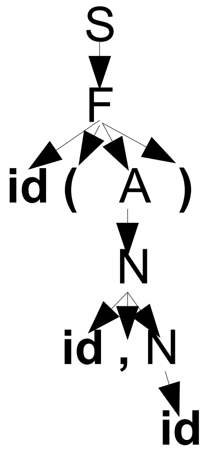
Massive Earley Example

Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)



id (id , id)

chart[0]	chart[1]	chart[2]	chart[3]	chart[4]	chart[5]	chart[6]

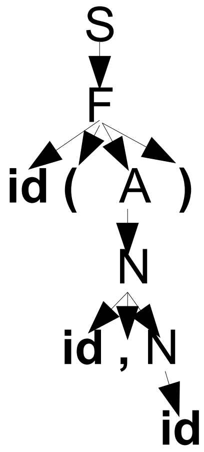
Massive Earley Example

Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)



id (id)

chart[0] **chart[1]** **chart** **chart[5]** **chart[6]**

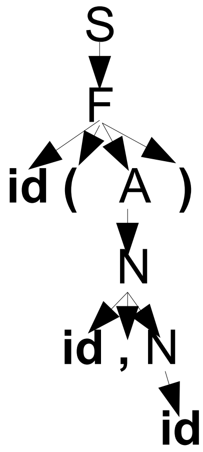
Massive Earley Example

Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)



id (id , id)

chart[0]	chart[1]	chart[2]	chart[3]	chart[4]	chart[5]	chart[6]
S→•F ,0						

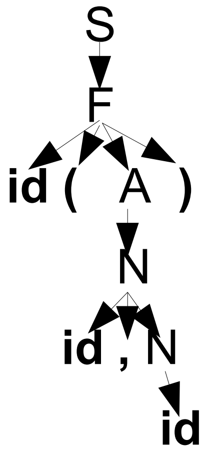
Massive Earley Example

Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)



id (id , id)

chart[0]	chart[1]	chart[2]	chart[3]	chart[4]	chart[5]	chart[6]
S→•F ,0						
F→•id(A) ,0						

Closure on F

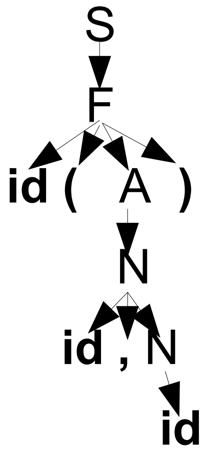
Massive Earley Example

Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)



id (id , id)

chart[0]	chart[1]	chart[2]	chart[3]	chart[4]	chart[5]	chart[6]
S→•F ,0	F→id•(A) ,0					
F→•id(A) ,0						

Shift on “id”

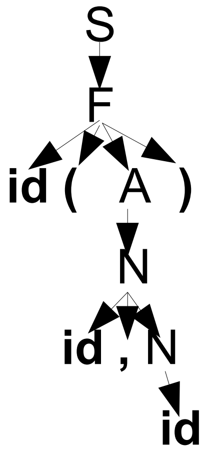
Massive Earley Example

Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)



id (id , id)

chart[0]	chart[1]	chart[2]	chart[3]	chart[4]	chart[5]	chart[6]
S→•F ,0	F→id•(A) ,0	F→id(•A) ,0				
F→•id(A) ,0						

Shift on “(”

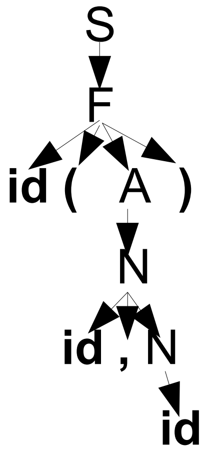
Massive Earley Example

Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)



id (id , id)

chart[0]	chart[1]	chart[2]	chart[3]	chart[4]	chart[5]	chart[6]
S→•F ,0	F→id•(A) ,0	F→id(•A) ,0				
F→•id(A) ,0		A→•N ,2				
		A→• ,2				

Closure on A

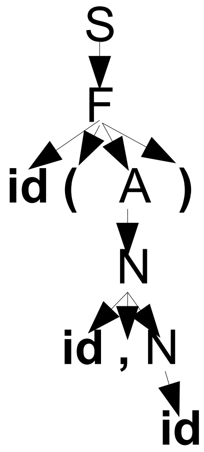
Massive Earley Example

Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)



id (id , id)

chart[0]	chart[1]	chart[2]	chart[3]	chart[4]	chart[5]	chart[6]
S→•F ,0	F→id•(A) ,0	F→id(•A) ,0				
F→•id(A) ,0		A→•N ,2				
		A→• ,2				
		N→•id ,2				
		N→•id,N ,2				

Closure on N

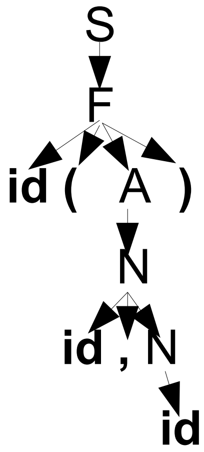
Massive Earley Example

Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)



id (id , id)

chart[0]	chart[1]	chart[2]	chart[3]	chart[4]	chart[5]	chart[6]
S→•F ,0	F→id•(A) ,0	F→id(•A) ,0				
F→•id(A) ,0		A→•N ,2				
		A→• ,2				
		N→•id ,2				
		N→•id,N ,2				
		F→id(A•) ,0				

Reduce on A

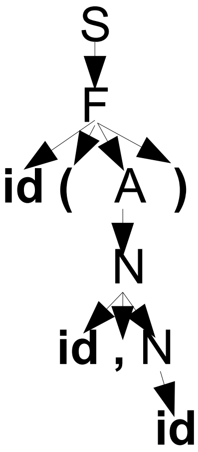
Massive Earley Example

Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)



id (id , id)

chart[0]	chart[1]	chart[2]	chart[3]	chart[4]	chart[5]	chart[6]
S→•F ,0	F→id•(A) ,0	F→id(•A) ,0	N→id• ,2			
F→•id(A) ,0		A→•N ,2	N→id•,N ,2			
		A→• ,2				
		N→•id ,2				
		N→•id,N ,2				
		F→id(A•) ,0				

Shift on “id”

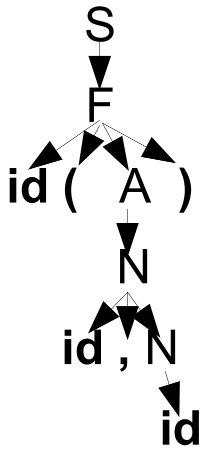
Massive Earley Example

Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)



id (id , id)

chart[0]	chart[1]	chart[2]	chart[3]	chart[4]	chart[5]	chart[6]
S→•F ,0	F→id•(A) ,0	F→id(•A) ,0	N→id• ,2			
F→•id(A) ,0		A→•N ,2	N→id•,N ,2			
		A→• ,2	A→N• ,2			
		N→•id ,2				
		N→•id,N ,2				
		F→id(A•) ,0				

Reduce on N

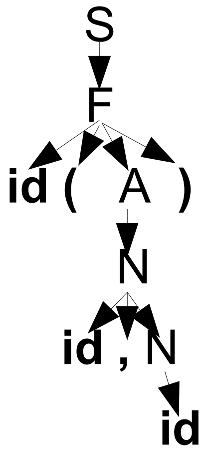
Massive Earley Example

Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)



id (id , id)

chart[0]	chart[1]	chart[2]	chart[3]	chart[4]	chart[5]	chart[6]
S→•F ,0	F→id•(A) ,0	F→id(•A) ,0	N→id• ,2			
F→•id(A) ,0		A→•N ,2	N→id•,N ,2			
		A→• ,2	A→N• ,2			
		N→•id ,2	F→id(A•) ,0			
		N→•id,N ,2				
		F→id(A•) ,0				

Reduce on A

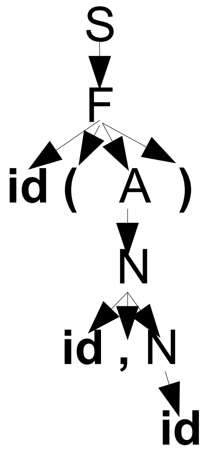
Massive Earley Example

Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)



id (id , id)

chart[0]	chart[1]	chart[2]	chart[3]	chart[4]	chart[5]	chart[6]
S→•F ,0	F→id•(A) ,0	F→id(•A) ,0	N→id• ,2	N→id,•N ,2		
F→•id(A) ,0		A→•N ,2	N→id•,N ,2			
		A→• ,2	A→N• ,2			
		N→•id ,2	F→id(A•) ,0			
		N→•id,N ,2				
		F→id(A•) ,0				

Shift on “,”

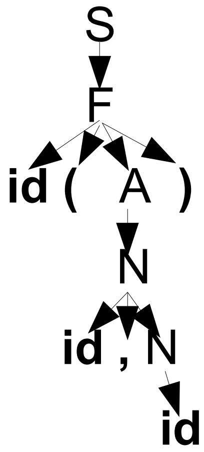
Massive Earley Example

Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)



id (id , id)

chart[0]	chart[1]	chart[2]	chart[3]	chart[4]	chart[5]	chart[6]
S→•F ,0	F→id•(A) ,0	F→id(•A) ,0	N→id• ,2	N→id,•N ,2		
F→•id(A) ,0		A→•N ,2	N→id•,N ,2	N→•id ,4		
		A→• ,2	A→N• ,2	N→•id,N ,4		
		N→•id ,2	F→id(A•) ,0			
		N→•id,N ,2				
		F→id(A•) ,0				

Closure on N

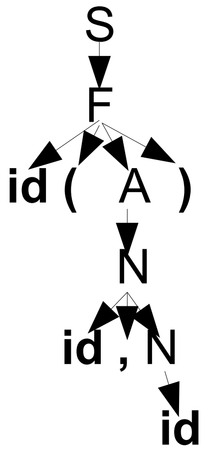
Massive Earley Example

Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)



id (id , id)

chart[0]	chart[1]	chart[2]	chart[3]	chart[4]	chart[5]	chart[6]
S→•F ,0	F→id•(A) ,0	F→id(•A) ,0	N→id• ,2	N→id,•N ,2	N→id• ,4	
F→•id(A) ,0		A→•N ,2	N→id•,N ,2	N→•id ,4	N→id•,N ,4	
		A→• ,2	A→N• ,2	N→•id,N ,4		
		N→•id ,2	F→id(A•) ,0			
		N→•id,N ,2				
		F→id(A•) ,0				

Shift on “id”

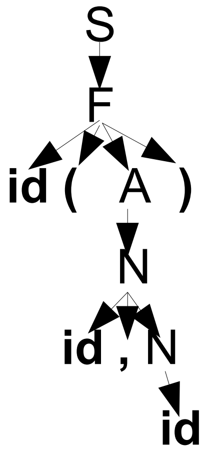
Massive Earley Example

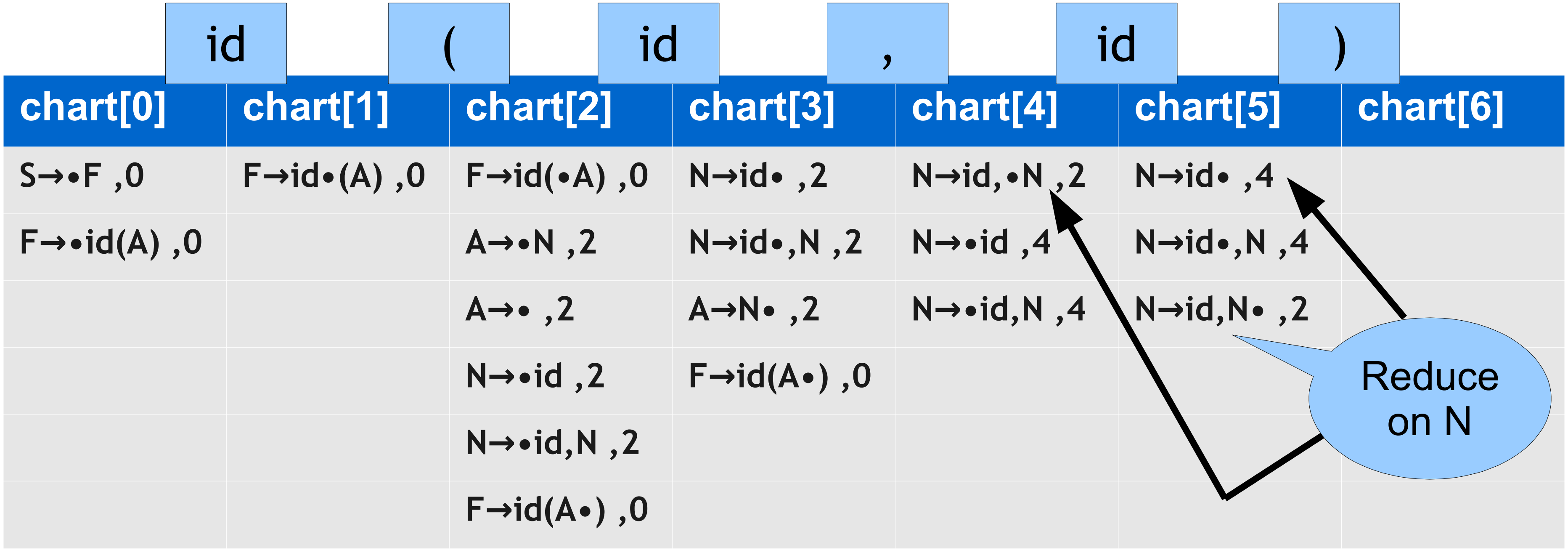
Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)





chart[0]	chart[1]	chart[2]	chart[3]	chart[4]	chart[5]	chart[6]
S→•F ,0	F→id•(A) ,0	F→id(•A) ,0	N→id• ,2	N→id,•N ,2	N→id• ,4	
F→•id(A) ,0		A→•N ,2	N→id•,N ,2	N→•id ,4	N→id•,N ,4	
		A→• ,2	A→N• ,2	N→•id,N ,4	N→id,N• ,2	
		N→•id ,2	F→id(A•) ,0			
		N→•id,N ,2				
		F→id(A•) ,0				

Reduce on N

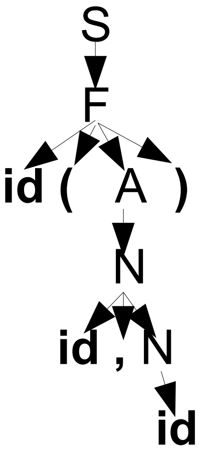
Massive Earley Example

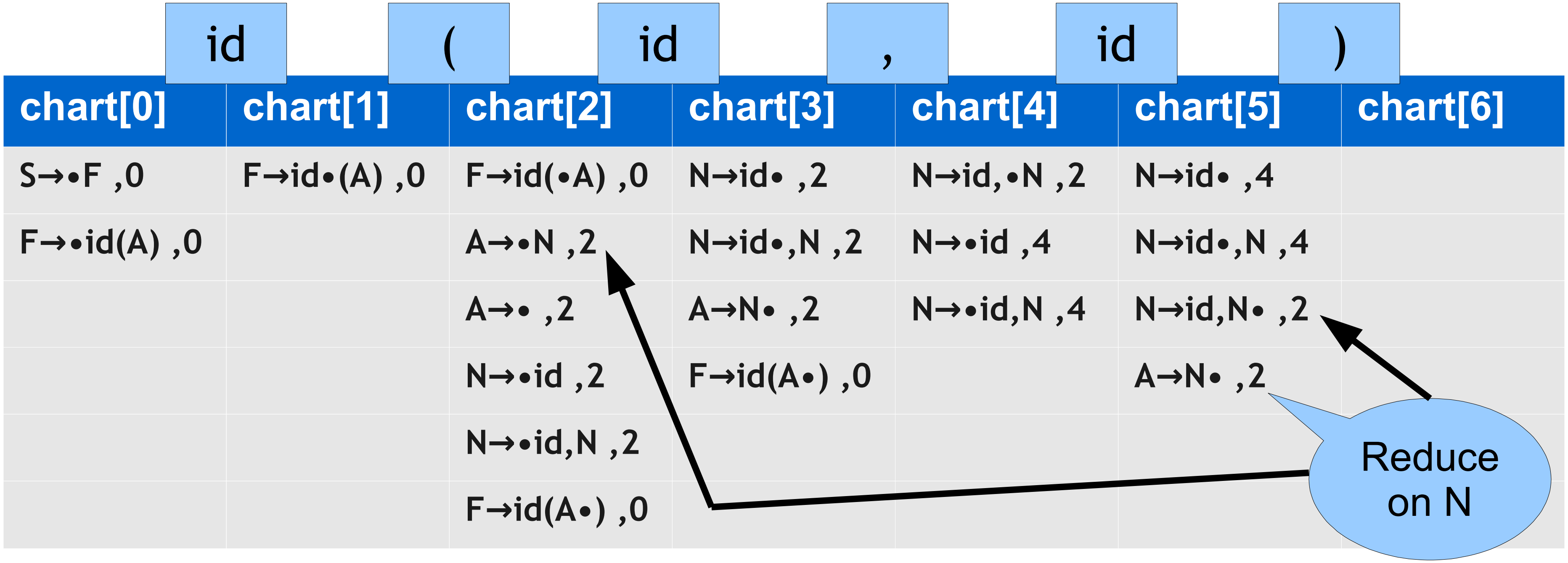
Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)





id (id , id)

chart[0]	chart[1]	chart[2]	chart[3]	chart[4]	chart[5]	chart[6]
S→•F ,0	F→id•(A) ,0	F→id(•A) ,0	N→id• ,2	N→id,•N ,2	N→id• ,4	
F→•id(A) ,0		A→•N ,2	N→id•,N ,2	N→•id ,4	N→id•,N ,4	
		A→• ,2	A→N• ,2	N→•id,N ,4	N→id,N• ,2	
		N→•id ,2	F→id(A•) ,0		A→N• ,2	
		N→•id,N ,2				
		F→id(A•) ,0				

Reduce on N

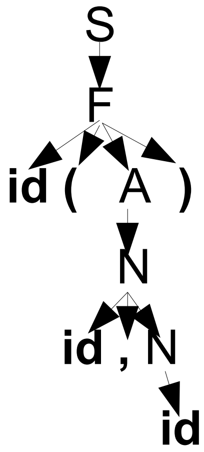
Massive Earley Example

Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)



id (id , id)

chart[0]	chart[1]	chart[2]	chart[3]	chart[4]	chart[5]	chart[6]
S→•F ,0	F→id•(A) ,0	F→id(•A) ,0	N→id• ,2	N→id,•N ,2	N→id• ,4	
F→•id(A) ,0		A→•N ,2	N→id•,N ,2	N→•id ,4	N→id•,N ,4	
		A→• ,2	A→N• ,2	N→•id,N ,4	N→id,N• ,2	
		N→•id ,2	F→id(A•) ,0		A→N• ,2	
		N→•id,N ,2			F→id(A•) ,0	
		F→id(A•) ,0				

Reduce on A

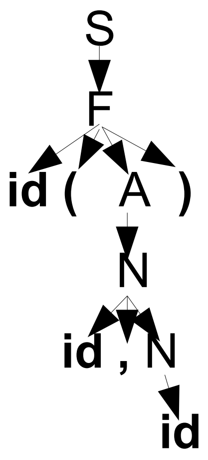
Massive Earley Example

Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)



id (id , id)

chart[0]	chart[1]	chart[2]	chart[3]	chart[4]	chart[5]	chart[6]
S→•F ,0	F→id•(A) ,0	F→id(•A) ,0	N→id• ,2	N→id,•N ,2	N→id• ,4	F→id(A)• ,0
F→•id(A) ,0		A→•N ,2	N→id•,N ,2	N→•id ,4	N→id•,N ,4	
		A→• ,2	A→N• ,2	N→•id,N ,4	N→id,N• ,2	
		N→•id ,2	F→id(A•) ,0		A→N• ,2	
		N→•id,N ,2			F→id(A•) ,0	
		F→id(A•) ,0				

Shift on ")"

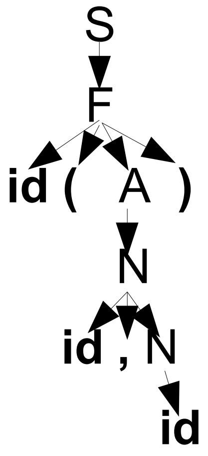
Massive Earley Example

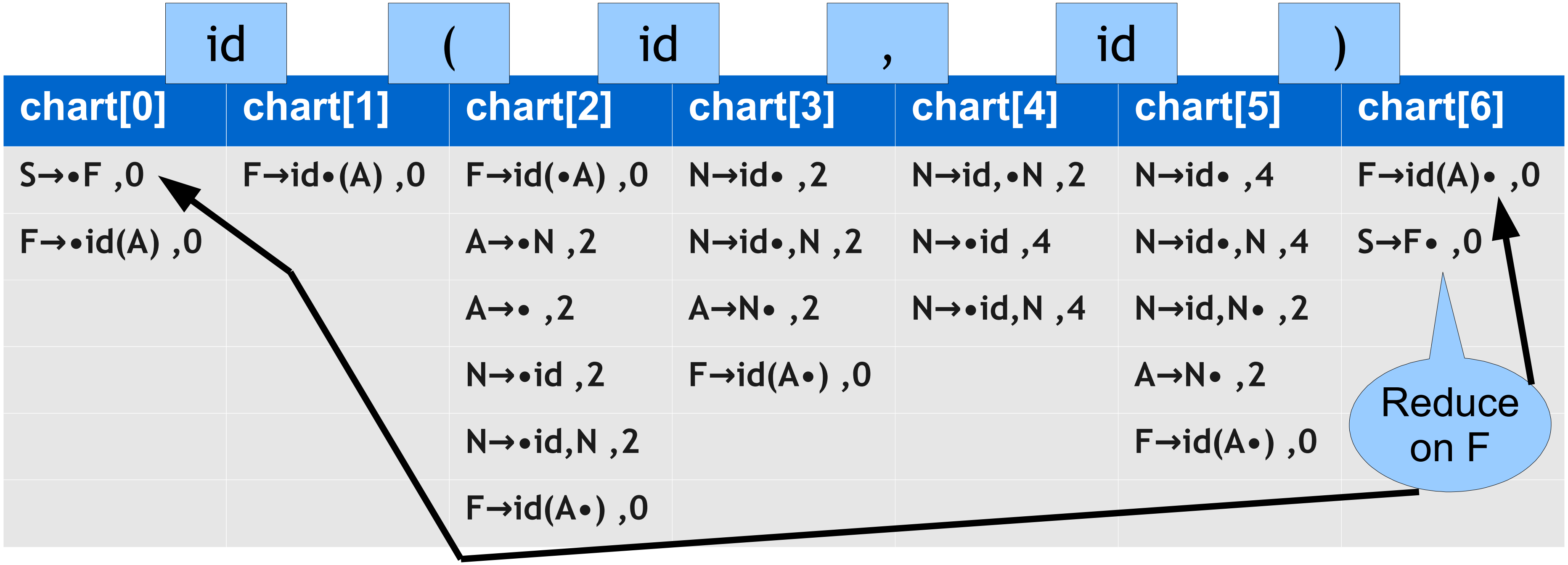
Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)





chart[0]	chart[1]	chart[2]	chart[3]	chart[4]	chart[5]	chart[6]
S→•F ,0	F→id•(A) ,0	F→id(•A) ,0	N→id• ,2	N→id,•N ,2	N→id• ,4	F→id(A)• ,0
F→•id(A) ,0		A→•N ,2	N→id•,N ,2	N→•id ,4	N→id•,N ,4	S→F• ,0
		A→• ,2	A→N• ,2	N→•id,N ,4	N→id,N• ,2	
		N→•id ,2	F→id(A•) ,0		A→N• ,2	
		N→•id,N ,2			F→id(A•) ,0	
		F→id(A•) ,0				

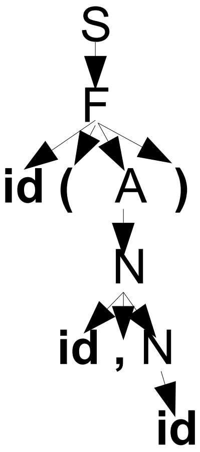
Massive Earley Example

Grammar

S → F
F → id (A)
A → N
A → ε
N → id
N → id , N

Input

id (id , id)



id (id , id)

chart[0]	chart[1]	chart[2]	chart[3]	chart[4]	chart[5]	chart[6]
S→•F ,0	F→id•(A) ,0	F→id(•A) ,0	N→id• ,2	N→id,•N ,2	N→id• ,4	F→id(A)• ,0
F→•id(A) ,0		A→•N ,2	N→id•,N ,2	N→•id ,4	N→id•,N ,4	S→F• ,0
		A→• ,2	A→N• ,2	N→•id,N ,4	N→id,N• ,2	
		N→•id ,2	F→id(A•) ,0		A→N• ,2	
		N→•id,N ,2			F→id(A•) ,0	
		F→id(A•) ,0				

Accept!

Homework

- Start PA3 “Early” (all the jokes are this bad)
- Study for Midterm 1
  - Complete Review Set 2
- Reading!

- Midterm 1 Next Thursday