More Finite Automata/ Lexical Analysis /Introduction to Parsing

Lecture 7

Programming a lexer in Lisp "by hand"

- (actually picked out of comp.lang.lisp when I was teaching CS164 3 years ago, an example by Kent Pitman).
- Given a string like **"foo+34-bar*g(zz)"** we could separate it into a lisp list of strings:

("foo" "+" "34" …) or we could try for a list of Lisp symbols like **(foo + 34 - bar * g |(| zz |)|).**

Huh? What is |(| ? It is the way lisp prints the symbol with printname "(" so as to not confuse the Lisp read program, and humans too.

Set up some data and predicates

```
(defvar *whitespace* '(#\Space #\Tab #\Return #\Linefeed))

(defun whitespace? (x) (member x *whitespace*))

(defvar *single-char-ops* '(#\+ #\- #\* #\/ #\( #\) #\. #\, #\=))

(defun single-char-op? (x) (member x *single-char-ops*))
```

Tokenize function…

```
(defun tokenize (text) ;; text is a string "ab+cd(x)"
  (let ((chars '()) (result '()))
      (declare (special chars result)) ;;explain scope
       (dotimes (i (length text))
         (let ((ch (char text i))) ;;pick out ith character of string
           (cond ((whitespace? ch)
                  (next-token))
                 ((single-char-op? ch)
                  (next-token)
                  (push ch chars)
                  (next-token))
                 (t
                  (push ch chars)))))
       (next-token)
       (nreverse result)))
```

Next-token / two versions

```
(defun next-token () ;;simple version
  (declare (special chars result))
  (when chars
    (push (coerce (nreverse chars) 'string) result)
    (setf chars '())))

(defun next-token () ;; this one “parses” integers magically
  (declare (special chars result))
  (when chars
    (let((st (coerce (reverse chars) 'string))) ;keep chars around
   to test
    (push (if (every #'digit-char-p chars)
             (read-from-string st)
           (intern st))
          result))
    (setf chars '())))
```

Example

- `(tokenize "foo(-)+34")` → `(foo |(| - |)| + 34)`

- `(Much) more info in file: pitmantoken.cl`

- `Missing: line/column numbers, 2-char tokens, keyword vs. identifier distinction. Efficiency here is low (but see file for how to use hash tables for character types!)`

- `Also note that Lisp has a programmable read-table so that its own idea of what delimits a token can be changed, as well as meanings of every character.`

Introduction to Parsing

Outline

- Regular languages revisited
- Parser overview
- Context-free grammars (CFG's)
- Derivations

Languages and Automata

- Formal languages are very important in CS
  - Especially in programming languages
- Regular languages
  - The weakest class of formal languages widely used
  - Many applications
- We will also study context-free languages

Limitations of Regular Languages

- Intuition: A finite automaton with N states that runs N+1 steps must revisit a state.
- Finite automaton can't remember # of times it has visited a particular state. No way of telling how it got here.
- Finite automaton can only use finite memory.
  - Only enough to store in which state it is
  - Cannot count, except up to a finite limit
- E.g., language of balanced parentheses is not regular: $\{ (^i\ )^i \mid i > 0 \}$

Context Free Grammars are more powerful

- Easy to parse balanced parentheses and similar nested structures
- A good fit for the vast majority of syntactic structures in programming languages like arithmetic expressions.
- Eventually we will find constructions that are not CFG, or are more easily dealt with outside the parser.

The Functionality of the Parser

- **Input:** sequence of tokens from lexer
- **Output:** parse tree of the program

Example

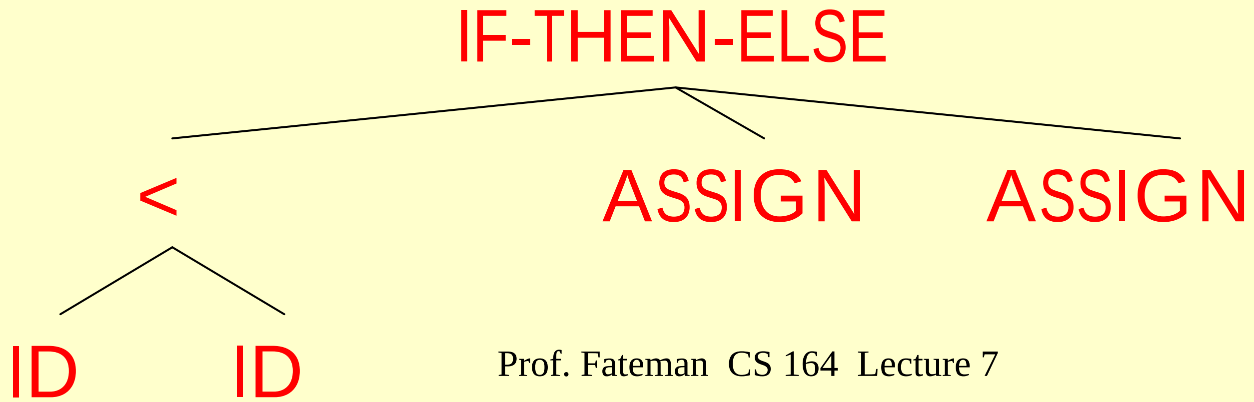
- Program Source

  if (x < y) a=1; else a=2;

  Lex output = parser input (simplified)

  IF lpar ID < ID rpar ID = ICONST ; ID= ICONST ICONST

- Parser output (simplified)



Example

- MJSource

if (x<y) a=1; else a=2;

- Actual lex output (from lisp…)

```
(fstring "          if (x<y) a=1; else a=2;") →

(if if (1 . 10))
(#\( #\( (1 . 12))
(id x (1 . 13))
(#\< #\< (1 . 14))
(id y (1 . 15))
(#\) #\) (1 . 16))
(id a (1 . 18))
(#\= #\= (1 . 19))
(iconst 1 (1 . 20))
(#\; #\; (1 . 21))
(else else (1 . 26)) …
```

Example

- MJSource

  if (x < y) a=1; else a=2;

- Actual Parser output ; lc = line&column

```
(If (LessThan (IdentifierExp x) (IdentifierExp y))
        (Assign (id a lc) (IntegerLiteral 1))
        (Assign (id a lc) (IntegerLiteral 2))))
```

  – Or cleaned up by taking out "extra" stuff …

```
(If (< x y) (assign a 1)(assign a 2))
```

Comparison with Lexical Analysis

Phase	*Input*	*Output*
Lexer	Sequence of characters	Sequence of tokens
Parser	Sequence of tokens	Parse tree

The Role of the Parser

- Not all sequences of tokens are programs . . .
- . . . Parser must distinguish between valid and invalid sequences of tokens
- Some sequences are valid only in some context, e.g. MJ requires framework.

- We need
  - A formal technique G for describing exactly and only the valid sequences of tokens (i.e. describe a language L(G))
  - An "implementation" of a recognizer for L, preferably based on automatically transforming G into a program. *G for grammar.*

A test framework for trivial MJ line of code

```
class Test {
   public static void main(String[ ] S){
    {   }   }}

class fooClass  {
   public int aMethod(int value) {
      int     a;
      int     x;
      int     y;

      if (x<y) a=1; else a=2;
      return 0;

   }}
```

Context-Free Grammars: Why

- Programming language constructs often have an underlying recursive structure

- An EXPR is EXPR + EXPR , … , or
  A statement is if EXPR statement; else statement , or
  while EXPR statement
  …

- Context-free grammars are a natural notation for this recursive structure

Context-Free Grammars: Abstractly

- A CFG consists of
  - A set of *terminals* T
  - A set of *non-terminals* N
  - A *start symbol* S (a non-terminal)
  - A set of *productions* , *or PAIRS of* $N \times (N \cup T)^*$

Assuming $X \in N$

$X \rightarrow \varepsilon$, or

$X \rightarrow Y_1 Y_2 \ldots Y_n$ where $Y_i \in N \cup T$

Notational Conventions

- In these lecture notes
  - Non-terminals are written upper-case
  - Terminals are written lower-case
  - The start symbol is the left-hand side of the first production

  ε production; vaguely related to same symbol in RE. X → ε means there is a rule by which X can be replaced by "nothing"

Examples of CFGs

A fragment of MiniJava

STATE→ if (EXPR) STATE;

STATE → LVAL = EXPR

EXPR → id

Examples of CFGs

A fragment of MiniJava

STATE → if (EXPR) STATE;

| LVAL = EXPR

EXPR → id

Shorthand notation with* | *.

Examples of CFGs (cont.)

Simple arithmetic expression language:

$$
\begin{aligned}
E \quad \rightarrow \quad & E * E \\
| \quad & E + E \\
| \quad & (E) \\
| \quad & id
\end{aligned}
$$

The Language of a CFG

Read productions as replacement rules in generating sentences in a language:

$X \rightarrow Y_1 \ldots Y_n$

Means X can be replaced by $Y_1 \ldots Y_n$

$X \rightarrow \varepsilon$

Means X can be erased (replaced with empty string)

Key Idea

1. Begin with a string consisting of the start symbol "S"
2. Pick a non-terminal X in the string by a right-hand side of some production e.g. $X \rightarrow YZ$

…string1 X string2… $\Rightarrow$ …string1 YZ string2 …

1. Repeat (2) until there are no non-terminals in the string. i.e. do $\Rightarrow^*$

The Language of a CFG (Cont.)

More formally, write

$X_1 \ldots X_i \ldots X_n \Rightarrow X_1 \ldots X_{i-1}\ y_1\ y_2 \ldots y_m\ X_{i+1} \ldots X_n$

if there is a production

$X_i \rightarrow y_1\ y_2 \ldots y_m$

Note, the double arrow denotes rewriting of strings is $\Rightarrow$

The Language of a CFG (Cont.)

Write $u \Rightarrow^* v$

If $u \Rightarrow \ldots \Rightarrow v$

in 0 or more steps

The Language of a CFG

Let *G* be a context-free grammar with start symbol *S*. Then the language of *G* is:

$$\{a_1 \ldots a_n \mid S \Rightarrow a_1 \ldots a_n \text{ and every } a_i \text{ is a terminal symbol}\}$$

Terminals

Terminals are called that because there are no rules for replacing them. (terminated..)

- Once generated, terminals are permanent.
- Terminals ought to be tokens of the language, numbers, ids, not concepts like "statement".

Examples

L(G) is the language of CFG G

Strings of balanced parentheses $\left\{ (^i)^i \mid i \geq 0 \right\}$

A simple grammar:

$$S \rightarrow (S)$$

$$S \rightarrow \varepsilon$$

To be more formal..

- The alphabet Σ for G is { (,)} , the set of two characters left and right parenthesis. This is the set of terminal symbols.
- The non-terminal symbols, **N** on the LHS of rules is here, a set of one element: {S}
- There is one distinguished non-terminal symbol, often S for “sentence” or “start” which is what you are trying to recognize.
- And then there is the finite list of rules or productions, technically a subset of $N \times (N \cup \Sigma)^*$

Let's produce some sentential forms of a MJgrammar

A fragment of a Tiger grammar:

STATE $\rightarrow$ if (EXPR) STATE ; else STATE

| while EXPR do STATE

| id

MJ Example (Cont.)

Some sentential forms of the language

id

if (expr) state; else state

while id do state;

if if id then id else id then id else id

Arithmetic Example

Simple arithmetic expressions:

$$E \rightarrow E+E \mid E * E \mid (E) \mid id$$

Some elements of the language:

id	id + id
(id)	id * id
(id) * id	id * (id)

Notes

The CFG idea for describing languages is a powerful concept. Understanding its complexities can solve many important Programming Language problems.

- Membership in a CFG's language is "yes" or "no".
- But to be useful to us, a CFG parser
  - Should show how a sentence corresponds to a parse tree.
  - Should handle non-sentences gracefully (pointing out likely errors).
  - Should be easy to generate from the grammar specification "automatically" (e.g., YACC, Bison, JCC, LALR-generator)

More Notes

- Form of the grammar is important
  - Different grammars can generate the identical language
  - Tools are sensitive to the form of the grammar
  - Restrictions on the types of rules can make automatic parser generation easier

Simple grammar (3.1 in text)

1: S → S ; S

2: S → id := E

3: S → print (L)

4: E → id

5: E → num

6: E → E + E

7: E → (S , E)

8: L → E

9: L → L , E

Derivations and Parse Trees

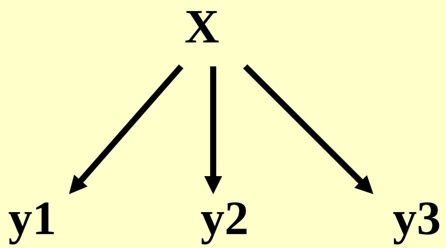
A *derivation* is a sequence of sentential forms starting with S, rewriting one non-terminal each step. A left-most derivation rewrites the left-most non-terminal.

	Using rules
<u>S</u>	**2**
id := <u>E</u>	**6**
id := <u>E</u> + E	**5**
id := num + <u>E</u>	**5**
id := num + num	

The sequence of rules tells us all we need to know! We can use it to generate a tree diagram for the sentence.

Building a Parse Tree

- Start symbol is the tree's root
- For a production $X \rightarrow y_1\, y_2\, y_3$ we draw



Another Derivation Example

- Grammar Rules

$$E \rightarrow E+E \mid E * E \mid (E) \mid id$$

- Sentential Form (input to parser)

$$id * id + id$$

Derivation Example (Cont.)

$$E$$

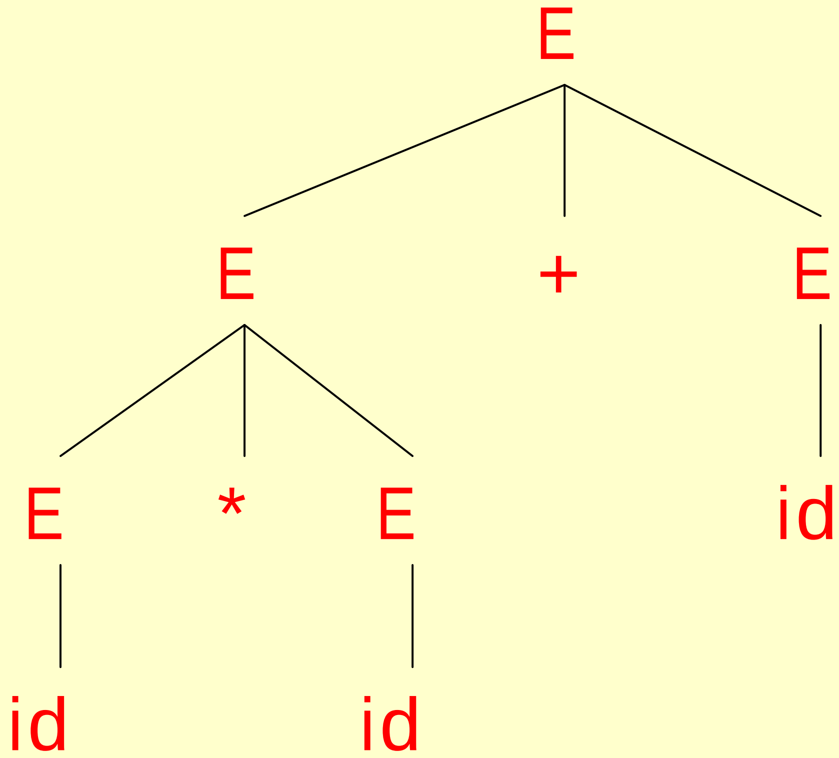
$$\rightarrow E+E$$

$$\rightarrow E * E+E$$

$$\rightarrow id * E + E$$

$$\rightarrow id * id + E$$

$$\rightarrow id * id + id$$

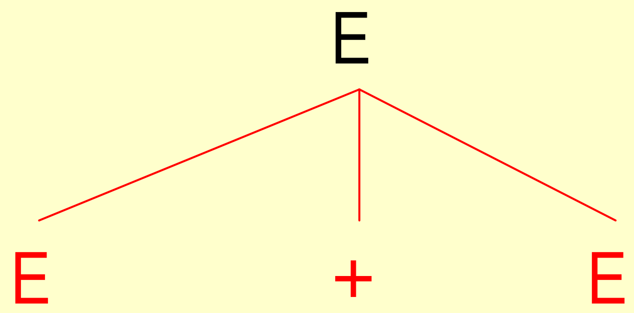


Left-Most Derivation in Detail (1)

E

E

Derivation in Detail (2)

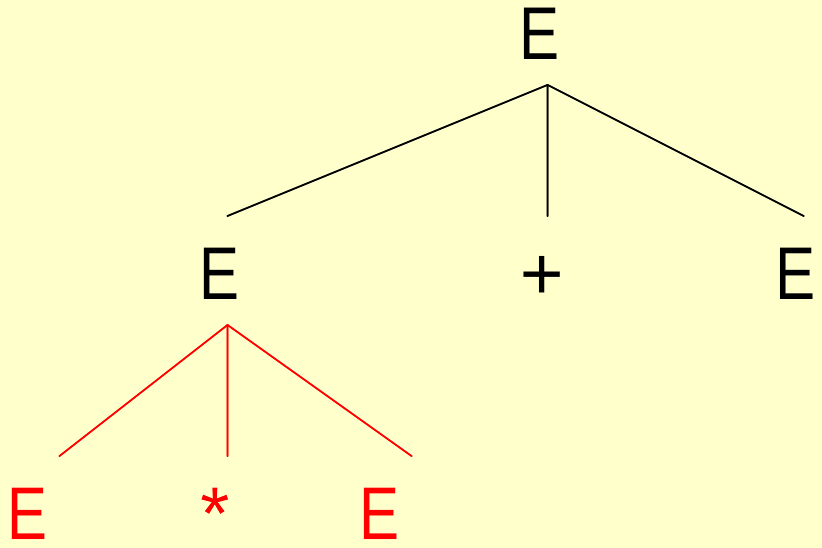


$$E$$

$$\rightarrow \quad E+E$$

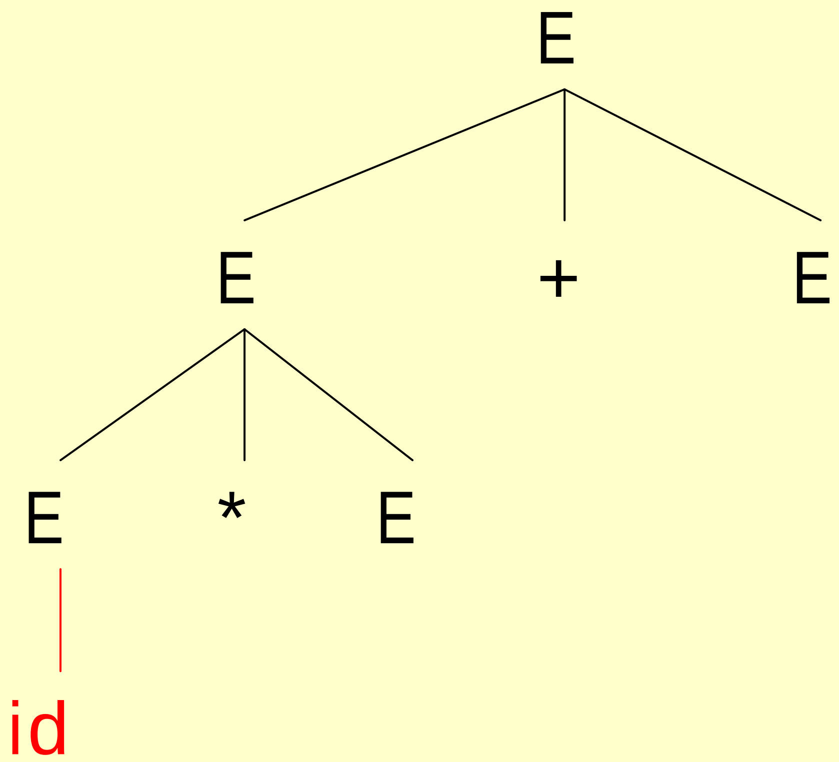
Derivation in Detail (3)

$$
\begin{aligned}
& E \\
\rightarrow\quad & E+E \\
\rightarrow\quad & E * E+E
\end{aligned}
$$



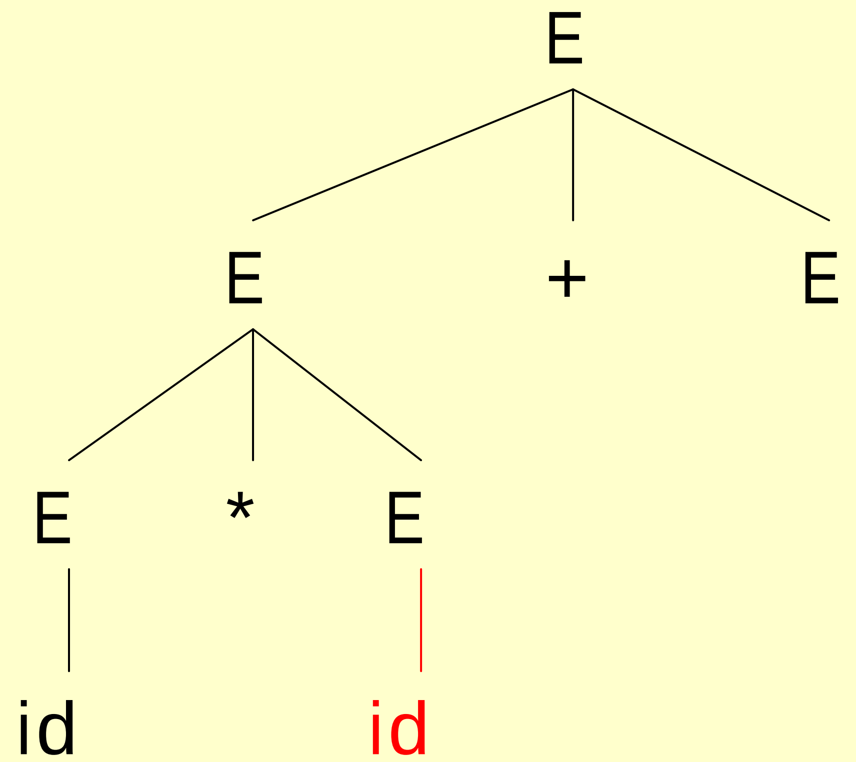
Derivation in Detail (4)

$$E$$
$$\rightarrow E+E$$
$$\rightarrow E*E+E$$
$$\rightarrow id*E+E$$



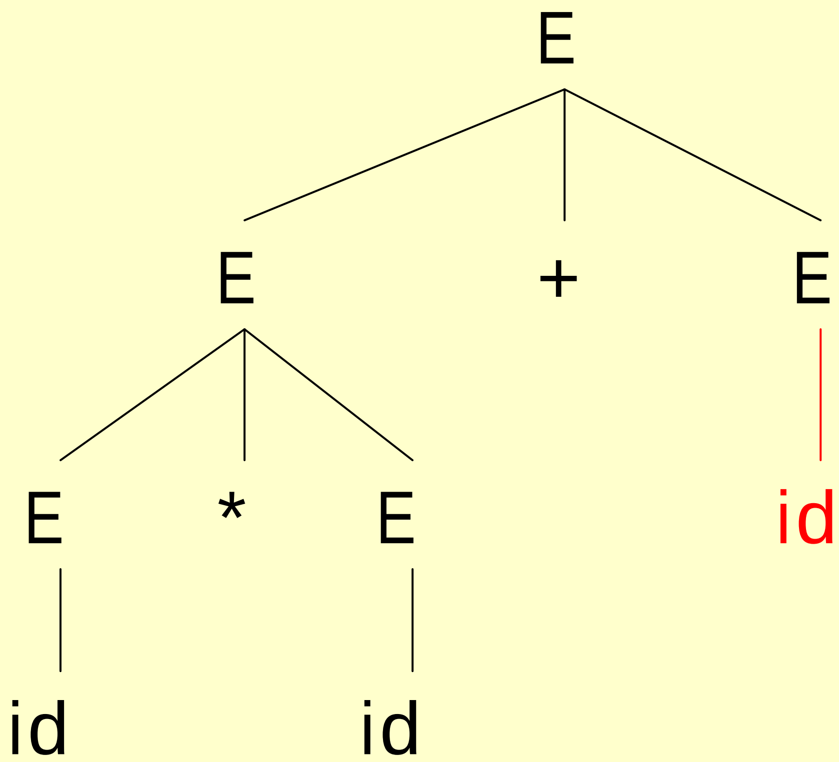
Derivation in Detail (5)

$$
\begin{array}{ll}
 & E \\
\rightarrow & E+E \\
\rightarrow & E * E+E \\
\rightarrow & id * E + E \\
\rightarrow & id * id + E
\end{array}
$$



Derivation in Detail (6)

$$
\begin{aligned}
& E \\
\rightarrow\; & E+E \\
\rightarrow\; & E * E+E \\
\rightarrow\; & id * E + E \\
\rightarrow\; & id * id + E \\
\rightarrow\; & id * id + id
\end{aligned}
$$



Notes on Derivations

- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- An in-order traversal of the leaves is the original input
- The parse tree shows the association of operations, even if the input string does not

What is a Right-most Derivation?

- Our examples were *left-most* derivations
  - At each step, replace the left-most non-terminal
- There is an equivalent notion of a *right-most* derivation

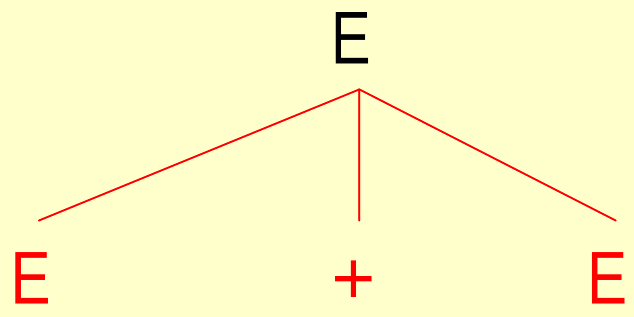
$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E * E + id \\
\rightarrow\ & E * id + id \\
\rightarrow\ & id * id + id
\end{aligned}
$$

Right-most Derivation in Detail (1)

E

E

Right-most Derivation in Detail (2)



$$E$$

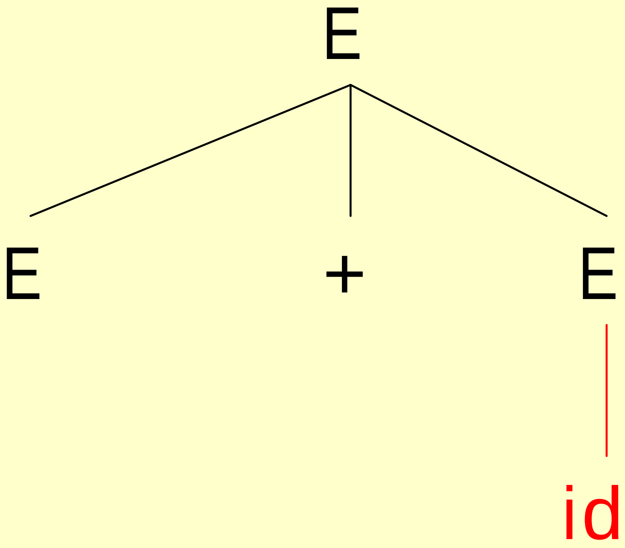
$$\rightarrow \quad E+E$$

Right-most Derivation in Detail (3)

E

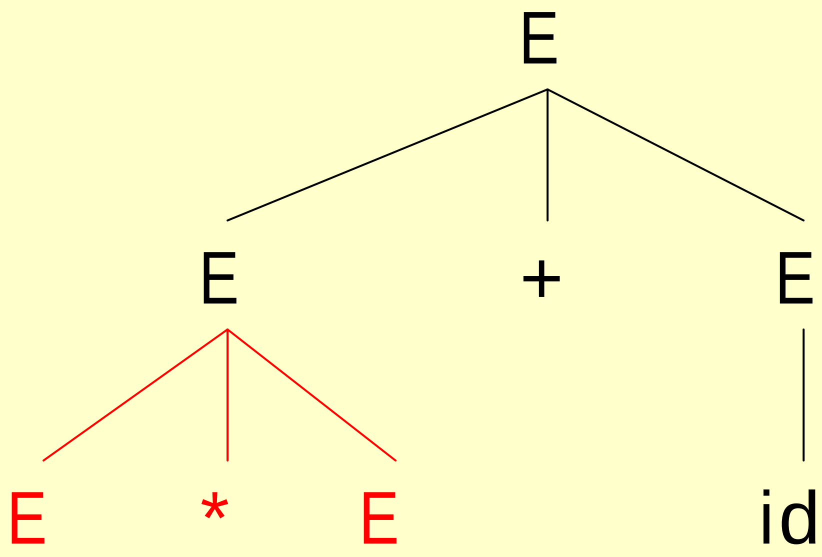
$\rightarrow$ E+E

$\rightarrow$ E+id



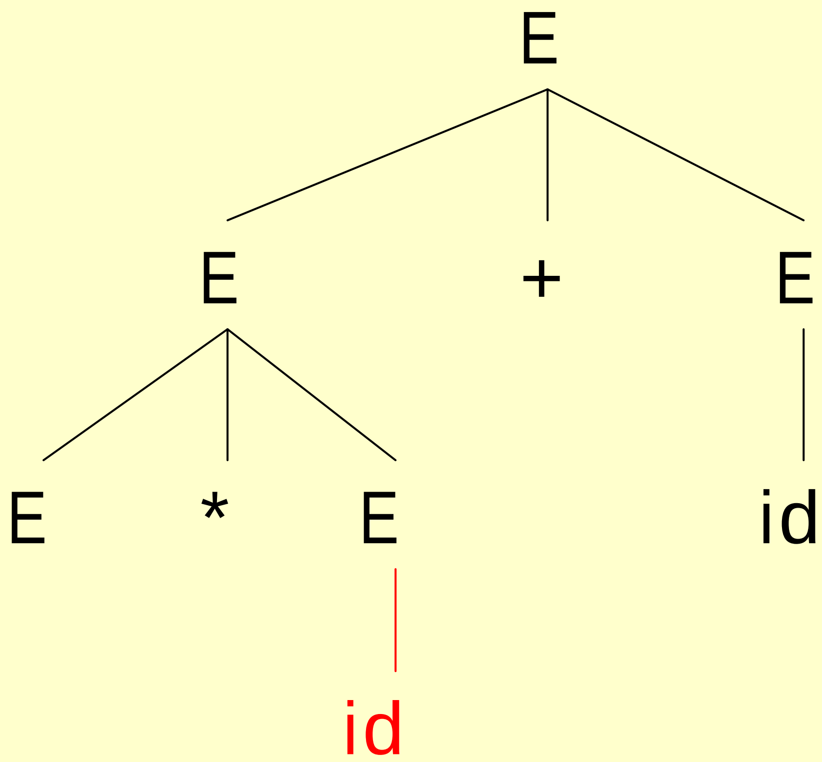
Right-most Derivation in Detail (4)

$$
\begin{aligned}
& E \\
\rightarrow\quad & E+E \\
\rightarrow\quad & E+id \\
\rightarrow\quad & E * E + id
\end{aligned}
$$



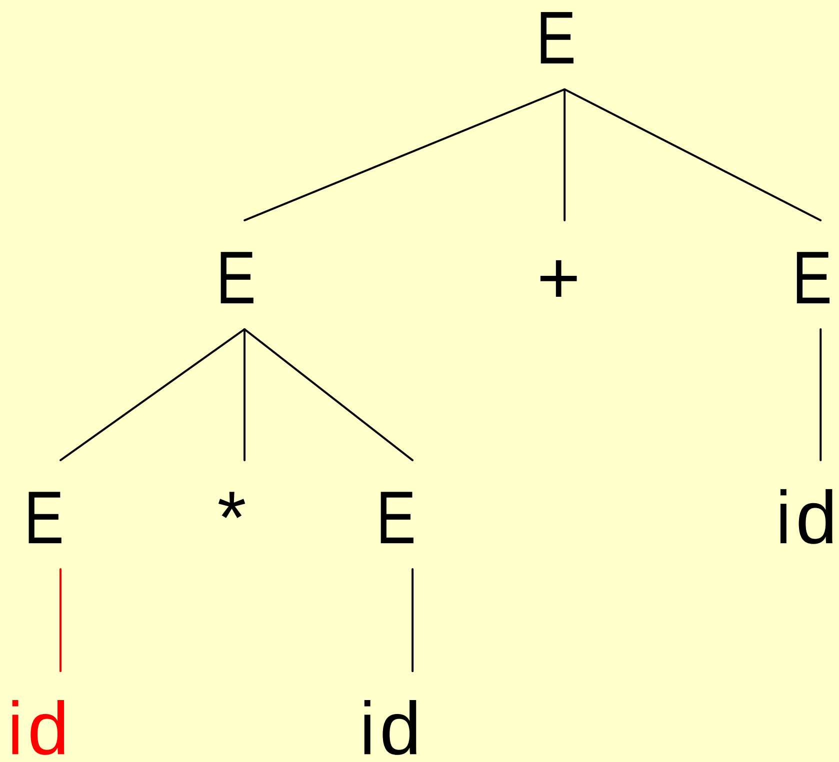
Right-most Derivation in Detail (5)

$$
\begin{aligned}
& E \\
\rightarrow\; & E+E \\
\rightarrow\; & E+id \\
\rightarrow\; & E * E + id \\
\rightarrow\; & E * id + id
\end{aligned}
$$



Right-most Derivation in Detail (6)

$$E$$
$$\rightarrow E+E$$
$$\rightarrow E+id$$
$$\rightarrow E * E + id$$
$$\rightarrow E * id + id$$
$$\rightarrow id * id + id$$



Derivations and Parse Trees

- Note that right-most and left-most derivations have the same parse tree

- The difference is the order in which branches are added

Summary: Objectives of Parsing

- We are not just interested in whether

  $s \in L(G)$

  – We need a parse tree for *s*

- A derivation defines a parse tree
  – But one parse tree may have many derivations

- Left-most and right-most derivations are important in parser implementation

Question from 9/21: grammar for /* */

- The simplest way of handling this is to write a program to just suck up characters looking for */, and "count backwards".
- Here's an attempt at a grammar
- C → / *A * /
- C → / * A C A * /
- A1 → a | b | c | 0 |…9 | … all chars not /
- B1 → a | b | c | 0 |…9 | … all chars not *
- A → A B1 | A1 B1 A B1 A1 | ε
- --To make this work, you'd need to have a grammar that covered both "real programs" and comments concatenated.