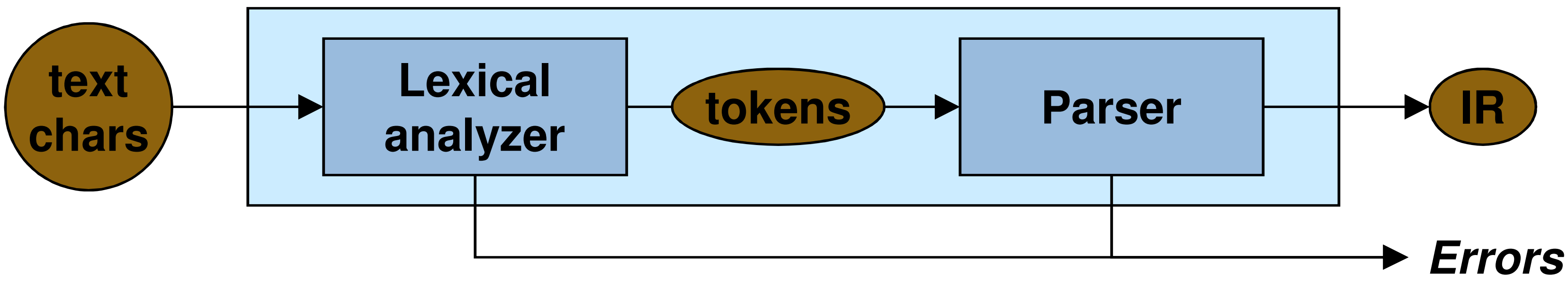
Compilers

Parsing

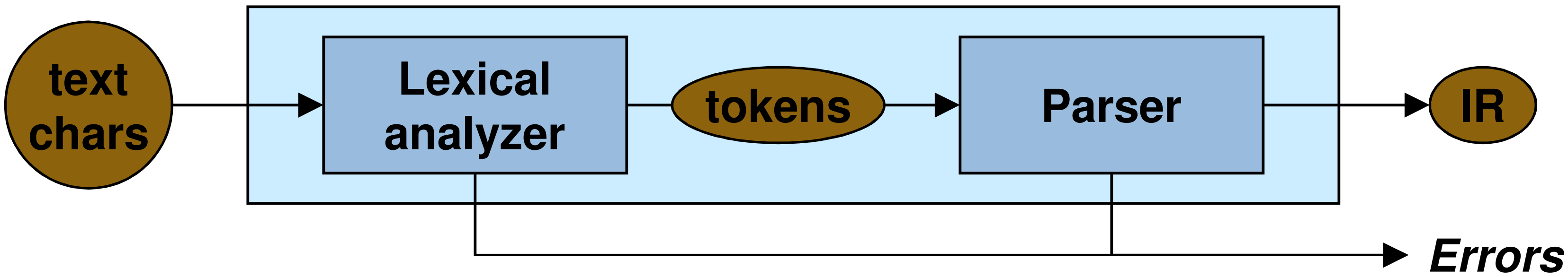
Yannis Smaragdakis, U. Athens
(original slides by Sam Guyer@Tufts)

Next step



- ***Parsing:*** Organize tokens into "sentences"
  - Do tokens conform to language ***syntax*** ?
  - ***Good news:*** token types are just numbers
  - ***Bad news:*** language syntax is fundamentally more complex than lexical specification
  - ***Good news:*** we can still do it in linear time in most cases

Parsing



- Parser
  - Reads tokens from the scanner
  - Checks organization of tokens against a ***grammar***
  - Constructs a ***derivation***
  - Derivation drives construction of IR

Study of parsing

- Discovering the derivation of a sentence
  - “Diagramming a sentence” in grade school
  - Formalization:
    - Mathematical model of syntax – a grammar G
    - Algorithm for testing membership in L(G)
- Roadmap:
  - Context-free grammars
  - Top-down parsers

    Ad hoc, often hand-coded, recursive decent parsers
  - Bottom-up parsers

    Automatically generated LR parsers

Specifying syntax with a grammar

- Can we use regular expressions?
  - For the most part, no
- Limitations of regular expressions
  - Need something more powerful
  - Still want formal specification *(for automation)*
- Context-free grammar
  - Set of rules for generating sentences
  - Expressed in ***Backus-Naur Form*** (BNF)

Context-free grammar

"produces" or "generates"

- Example:

#	*Production rule*
1	*sheepnoise* → *sheepnoise* `baa`
2	| `baa`

Alternative (shorthand)

- Formally: ***context-free grammar*** is
  - **G** = (s, N, T, P)
  - ***T*** : set of terminals *(provided by scanner)*
  - ***N*** : set of non-terminals *(represent structure)*
  - ***s*** $\in$ ***N*** : start or goal symbol
  - ***P*** : set of production rules of the form $N \rightarrow (N \cup T)^*$

Language L(G)

- Language L(G)

  L(G) is all sentences generated from start symbol

- Generating sentences
  - Use productions as ***rewrite rules***
  - Start with goal (or start) symbol – a non-terminal
  - Choose a non-terminal and “expand” it to the right-hand side of one of its productions
  - Only terminal symbols left → sentence in L(G)
  - Intermediate results known as ***sentential forms***

Expressions

- Language of expressions
  - Numbers and identifiers
  - Allow different binary operators
  - Arbitrary nesting of expressions

#	Production rule
1	*expr* → *expr* *op* *expr*
2	| `number`
3	| `identifier`
4	*op* → `+`
5	| `-`
6	| `*`
7	| `/`

Language of expressions

- What's in this language?

#	Production rule
1	*expr* → *expr op expr*
2	| number
3	| identifier
4	*op* → +
5	| −
6	| *
7	| /

Rule	Sentential form
-	*expr*
1	*expr op expr*
3	<id,x> *op expr*
5	<id,x> - *expr*
1	<id,x> - *expr op expr*
2	<id,x> - <num,2> *op expr*
6	<id,x> - <num,2> * *expr*
3	<id,x> - <num,2> * <id,y>

⇨ We can build the string "**x – 2 * y**"

This string is in the language

Derivations

- Using grammars
  - A sequence of rewrites is called a ***derivation***
  - Discovering a derivation for a string is ***parsing***
- Different derivations are possible
  - At each step we can choose any non-terminal
  - ***Rightmost derivation***: always choose right NT
  - ***Leftmost derivation***: always choose left NT

  (Other "random" derivations – not of interest)

Left vs right derivations

- Two derivations of "**`x – 2 * y`**"

Rule	*Sentential form*
-	*expr*
1	*expr op expr*
3	*<id, x> op expr*
5	*<id,x> - expr*
1	*<id,x> - expr op expr*
2	*<id,x> - <num,2> op expr*
6	*<id,x> - <num,2> * expr*
3	*<id,x> - <num,2> * <id,y>*

Left-most derivation

Rule	*Sentential form*
-	*expr*
1	*expr op expr*
3	*expr op <id,y>*
6	*expr * <id,y>*
1	*expr op expr * <id,y>*
2	*expr op <num,2> * <id,y>*
5	*expr - <num,2> * <id,y>*
3	*<id,x> - <num,2> * <id,y>*

Right-most derivation

Derivations and parse trees

- Two different derivations
  - Both are correct
  - Do we care which one we use?
- Represent derivation as a ***parse tree***
  - Leaves are terminal symbols
  - Inner nodes are non-terminals
  - To depict production $\alpha \rightarrow \beta\ \gamma\ \delta$
    show nodes β,γ,δ as children of α

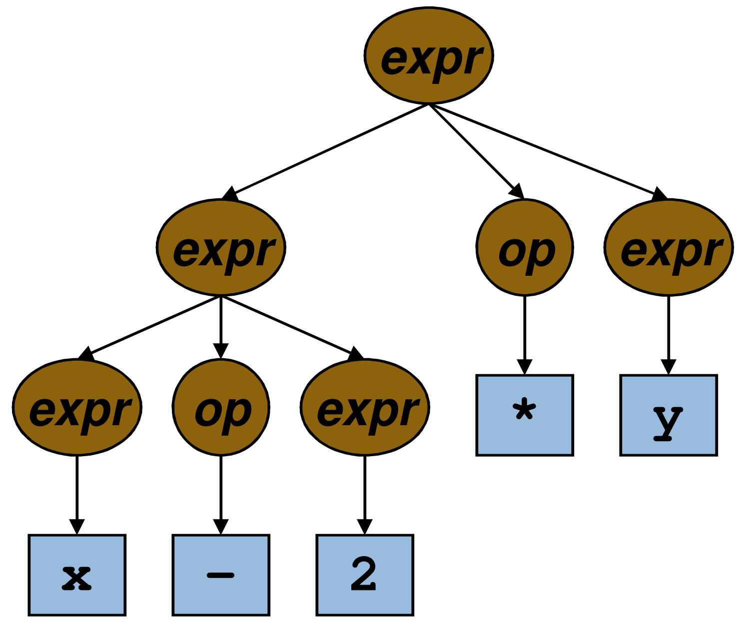
⇨ Tree is used to build internal representation

Example (I)

Right-most derivation

Rule	*Sentential form*
-	*expr*
1	*expr op expr*
3	*expr op <id,y>*
6	*expr * <id,y>*
1	*expr op expr * <id,y>*
2	*expr op <num,2> * <id,y>*
5	*expr - <num,2> * <id,y>*
3	*<id,x> - <num,2> * <id,y>*

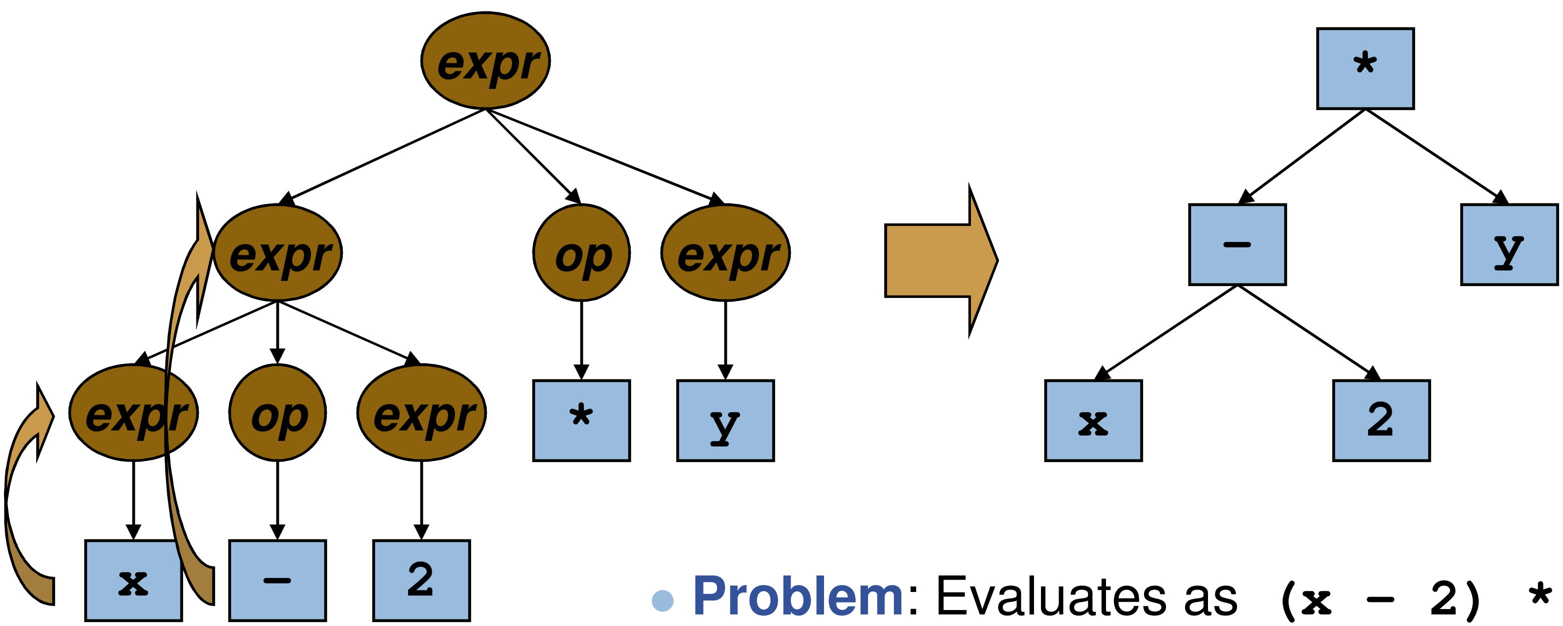
Parse tree



- *Concrete* syntax tree
  - Shows all details of syntactic structure
- What's the problem with this tree?

Abstract syntax tree

- Parse tree contains extra junk
  - Eliminate intermediate nodes
  - Move operators up to parent nodes
  - Result: ***abstract syntax tree***



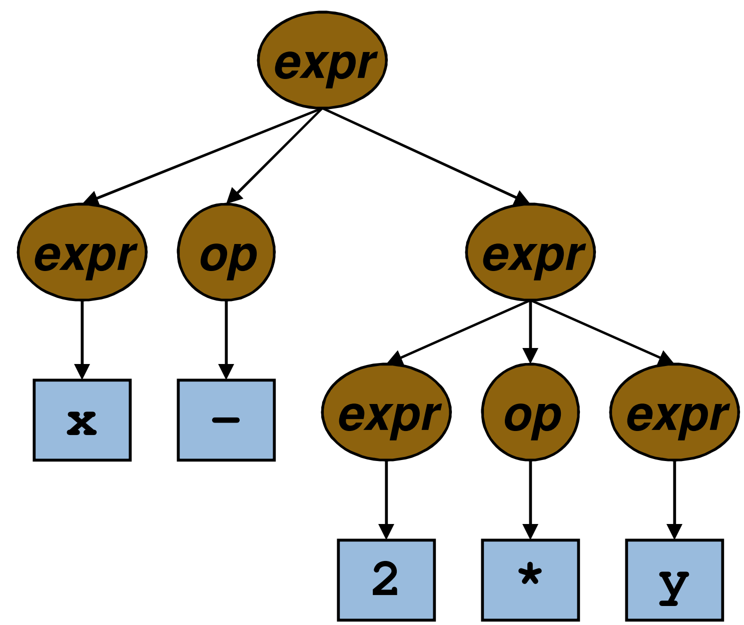
- **Problem**: Evaluates as `(x - 2) * y`

Example (II)

Left-most derivation

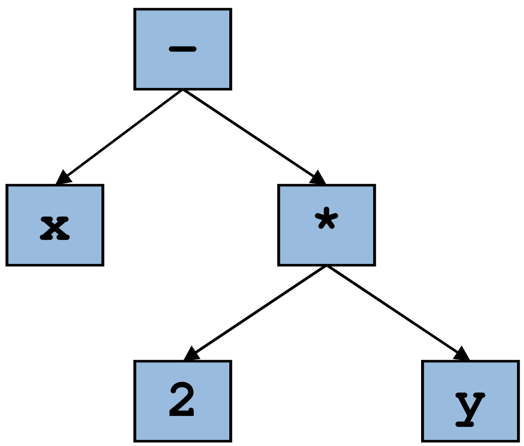
Rule	*Sentential form*
-	*expr*
1	*expr op expr*
3	*<id, x> op expr*
5	*<id,x> - expr*
1	*<id,x> - expr op expr*
2	*<id,x> - <num,2> op expr*
6	*<id,x> - <num,2> * expr*
3	*<id,x> - <num,2> * <id,y>*

Parse tree

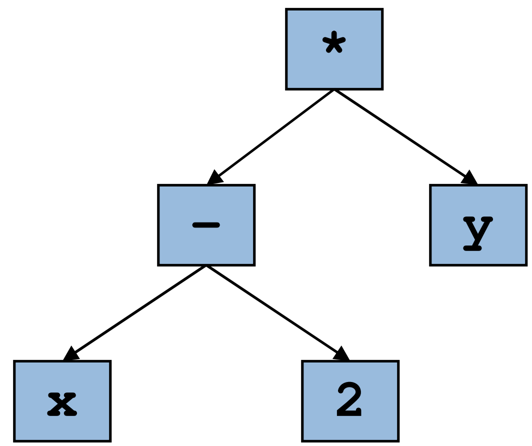


- **Solution**: evaluates as `x – (2 * y)`

Derivations



Left-most derivation



Right-most derivation

Derivations and semantics

- **Problem**:
  - Two different valid derivations
  - One captures “meaning” we want
    (What specifically are we trying to capture here?)
  - **Key idea**: shape of tree implies its meaning
- Can we express precedence in grammar?
  - Notice: operations deeper in tree evaluated first
  - **Solution**: add an intermediate production
    - New production isolates different levels of precedence
    - Force higher precedence “deeper” in the grammar

Adding precedence

- Two levels:

Level 1: lower precedence – higher in the tree

Level 2: higher precedence – deeper in the tree

#	Production rule				
1	*expr*	→	*expr*	+	*term*
2		|	*expr*	-	*term*
3		|	*term*		
4	*term*	→	*term*	*	*factor*
5		|	*term*	/	*factor*
6		|	*factor*		
7	*factor*	→	`number`		
8		|	`identifier`		

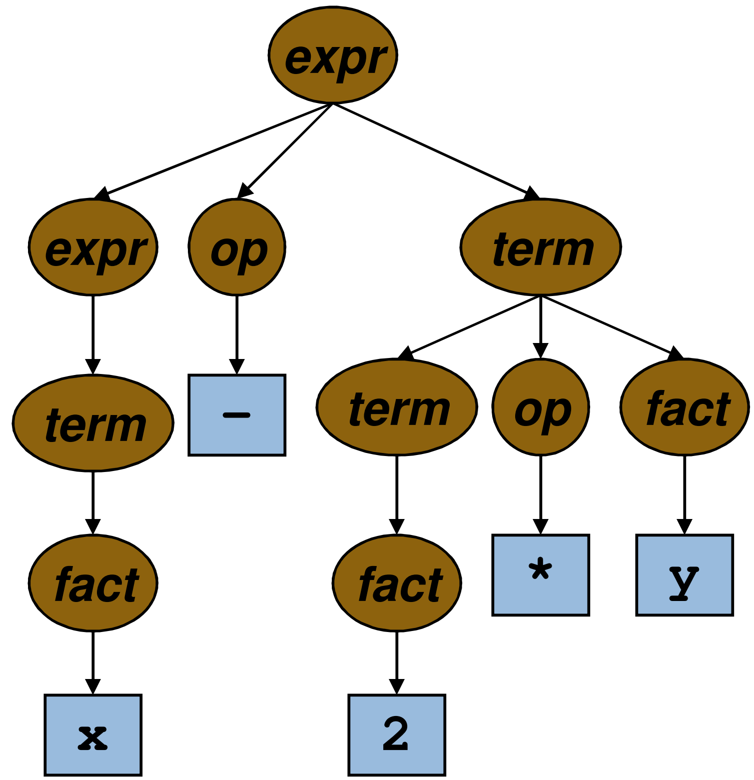
- Observations:
  - Larger: requires more rewriting to reach terminals
  - **But**, produces same parse tree under both left and right derivations

Expression example

Right-most derivation

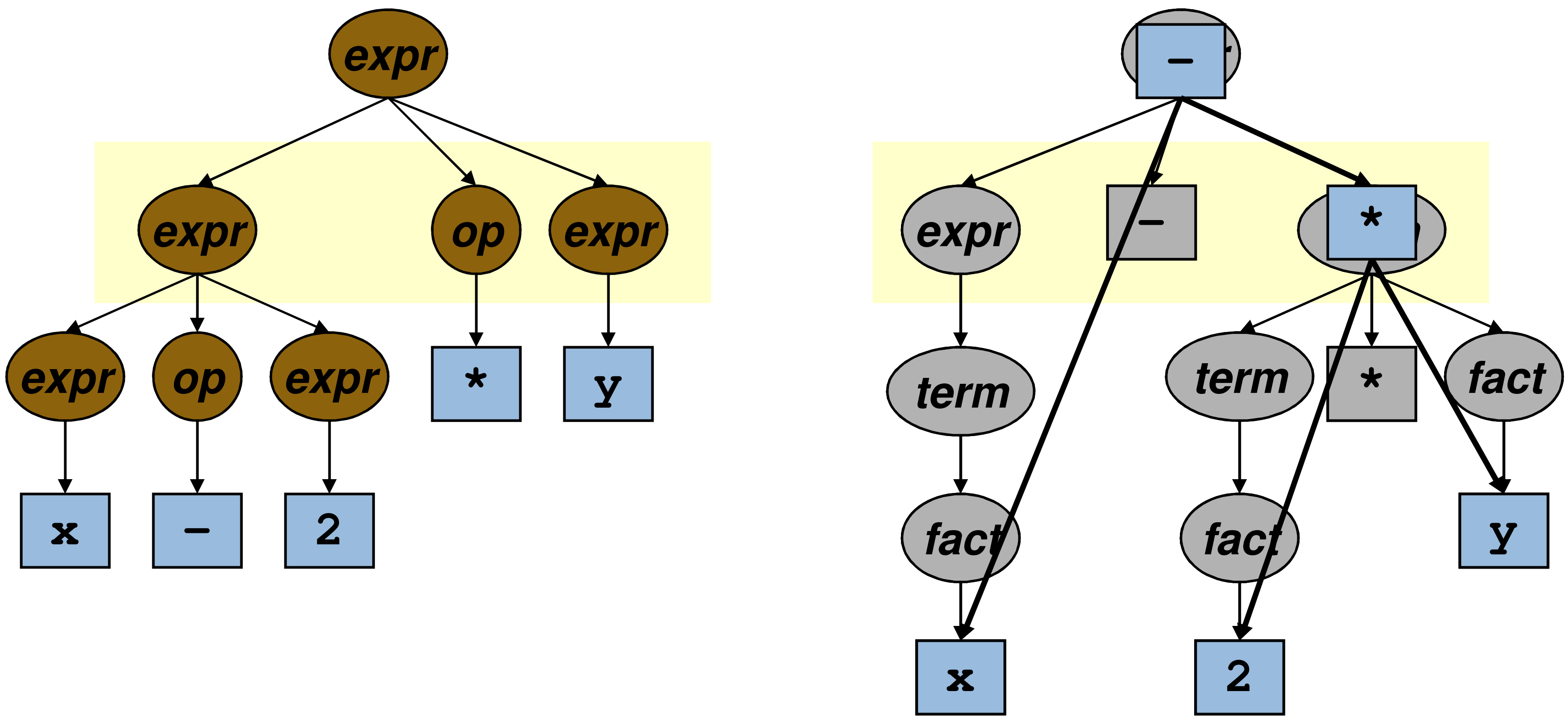
Rule	*Sentential form*
-	*expr*
2	*expr - term*
4	*expr - term * factor*
8	*expr - term * <id,y>*
6	*expr - factor * <id,y>*
7	*expr - <num,2> * <id,y>*
3	*term - <num,2> * <id,y>*
6	*factor - <num,2> * <id,y>*
8	*<id,x> - <num,2> * <id,y>*

Parse tree



Now right derivation yields `x – (2 * y)`

With precedence



Another issue

- Original expression grammar:

#	*Production rule*
1	*expr* → *expr* *op* *expr*
2	| <u>`number`</u>
3	| <u>`identifier`</u>
4	*op* → +
5	| −
6	| *
7	| /

- Our favorite string: `x – 2 * y`

Another issue

Rule	Sentential form
-	*expr*
1	*expr op expr*
1	*expr op expr op expr*
3	*<id, x> op expr op expr*
5	*<id,x> - expr op expr*
2	*<id,x> - <num,2> op expr*
6	*<id,x> - <num,2> * expr*
3	*<id,x> - <num,2> * <id,y>*

Rule	Sentential form
-	*expr*
1	*expr op expr*
3	*<id, x> op expr*
5	*<id,x> - expr*
1	*<id,x> - expr op expr*
2	*<id,x> - <num,2> op expr*
6	*<id,x> - <num,2> * expr*
3	*<id,x> - <num,2> * <id,y>*

- Multiple leftmost derivations
- Such a grammar is called ***ambiguous***
- Is this a problem?
  - Very hard to automate parsing

Ambiguous grammars

- A grammar is ambiguous *iff*:
  - There are multiple leftmost or multiple rightmost derivations for a single sentential form
  - ***Note***: leftmost and rightmost derivations may differ, even in an unambiguous grammar
  - **Intuitively**:
    - We can choose different non-terminals to expand
    - But each non-terminal should lead to a unique set of terminal symbols
- What's a classic example?
  - If-then-else ambiguity

If-then-else

- Grammar:

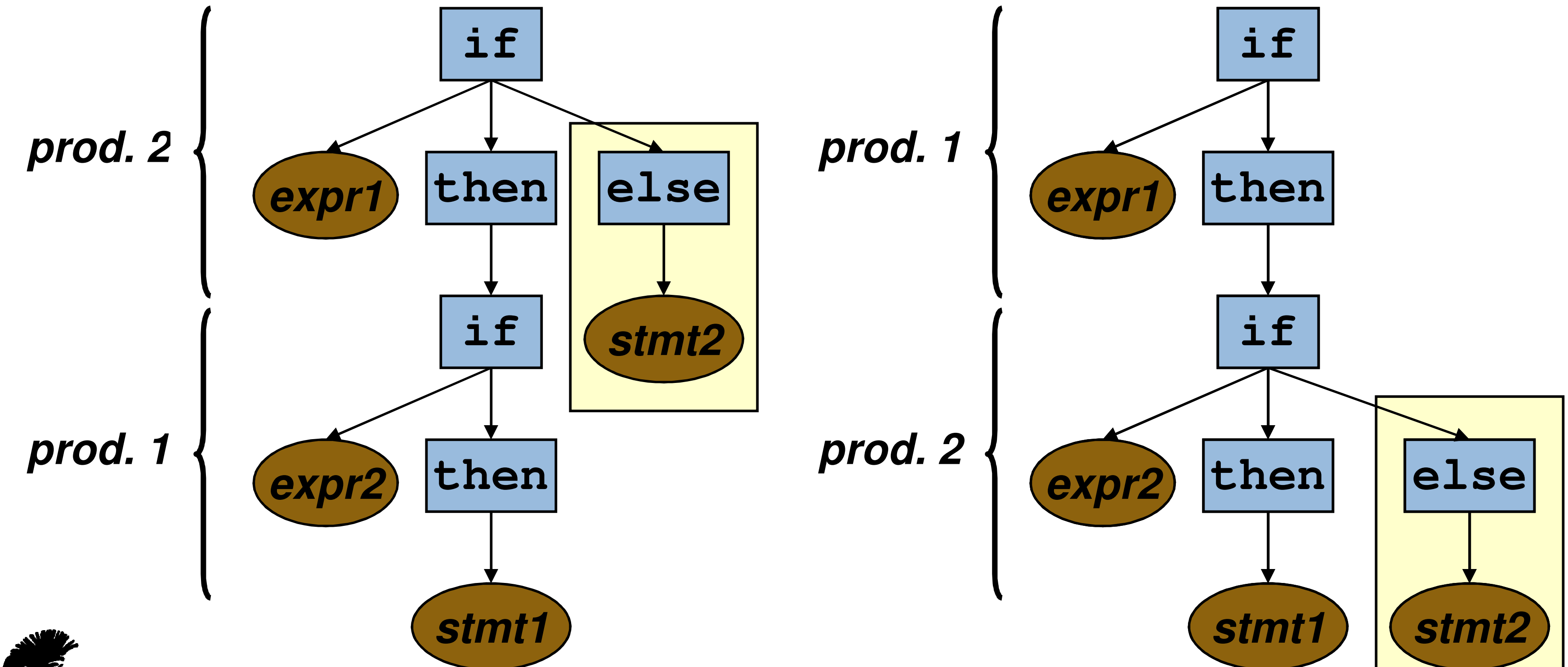
#	Production rule
1	*stmt* → <u>`if`</u> *expr* <u>`then`</u> *stmt*
2	| <u>`if`</u> *expr* <u>`then`</u> *stmt* <u>`else`</u> *stmt*
3	| *…other statements…*

- **Problem**: nested if-then-else statements
  - Each one may or may not have **`else`**
  - How to match each **`else`** with **`if`**

If-then-else ambiguity

- Sentential form with two derivations:

  `if` ***expr1*** **`then if`** ***expr2*** **`then`** ***stmt1*** **`else`** ***stmt2***



Removing ambiguity

- Restrict the grammar
  - Choose a rule: “else” matches innermost “if”
  - Codify with new productions

#	Production rule
1	*stmt* → `if` *expr* `then` *stmt*
2	| `if` *expr* `then` *withelse* `else` *stmt*
3	| *…other statements…*
4	*withelse* → `if` *expr* `then` *withelse* `else` *withelse*
5	| *…other statements…*

- **Intuition**: when we have an “else”, all preceding nested conditions must have an “else”

Ambiguity

- Ambiguity can take different forms
  - Grammatical ambiguity *(if-then-else problem)*
  - Contextual ambiguity
    - In C: `x * y;` could follow `typedef int x;`
    - In Fortran: `x = f(y);` f could be function or array

⇒ *Cannot be solved directly in grammar*

- Issues of ***type*** (later in course)

- Deeper question:

  How much can the parser do?

Parsing

- What is parsing?
  - Discovering the derivation of a string
    - *If one exists*
  - Harder than generating strings
    - *Not surprisingly*
- Two major approaches
  - Top-down parsing
  - Bottom-up parsing
- Don’t work on all context-free grammars
  - Properties of grammar determine parse-ability
  - **Our goal**: make parsing efficient
  - We may be able to transform a grammar

Two approaches

- Top-down parsers **LL(1), recursive descent**
  - Start at the root of the parse tree and grow toward leaves
  - Pick a production and try to match the input
  - What happens if the parser chooses the wrong one?

- Bottom-up parsers **LR(1), operator precedence**
  - Start at the leaves and grow toward root
  - Issue: might have multiple possible ways to do this
  - Key idea: encode possible parse trees in an internal state *(similar to our NFA → DFA conversion)*
  - Bottom-up parsers handle a large class of grammars

Grammars and parsers

- LL(1) parsers
  - **L**eft-to-right input
  - **L**eftmost derivation
  - **1** symbol of look-ahead

Grammars that they can handle are called LL(1) grammars

- LR(1) parsers
  - **L**eft-to-right input
  - **R**ightmost derivation
  - **1** symbol of look-ahead

Grammars that they can handle are called LR(1) grammars

- Also: LL(k), LR(k), SLR, LALR, …

Top-down parsing

- Start with the root of the parse tree
  - Root of the tree: node labeled with the start symbol
- **Algorithm**:

  Repeat until the fringe of the parse tree matches input string
  - At a node A, select one of A's productions

    Add a child node for each symbol on rhs
  - Find the next node to be expanded ***(a non-terminal)***
- Done when:
  - Leaves of parse tree match input string ***(success)***

Example

- Expression grammar *(with precedence)*

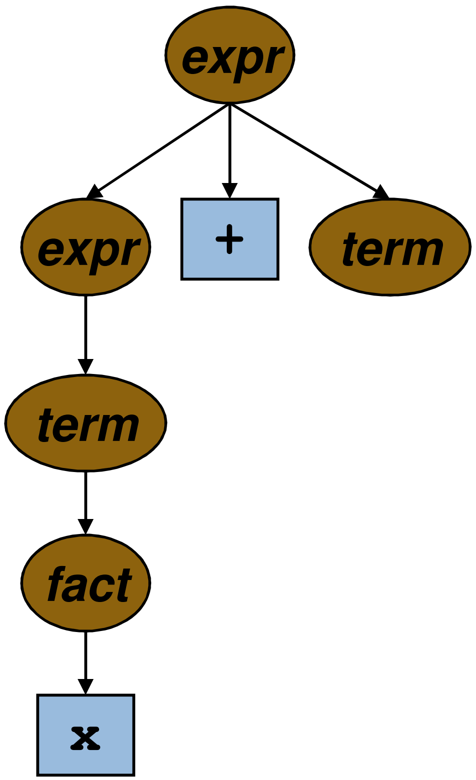
#	Production rule
1	*expr* → *expr* + *term*
2	| *expr* - *term*
3	| *term*
4	*term* → *term* * *factor*
5	| *term* / *factor*
6	| *factor*
7	*factor* → <u>`number`</u>
8	| <u>`identifier`</u>

- Input string **`x - 2 * y`**

Example

Current position in the input stream

Rule	*Sentential form*	*Input string*
-	*expr*	↑ x – 2 * y
1	*expr* + *term*	↑ x – 2 * y
3	*term* + *term*	↑ x – 2 * y
6	*factor* + *term*	↑ x – 2 * y
8	*<id>* + *term*	x ↑ – 2 * y
-	*<id,x>* + *term*	x ↑ – 2 * y



- **Problem**:
  - Can’t match next terminal
  - We guessed wrong at step 2
  - What should we do now?

Backtracking

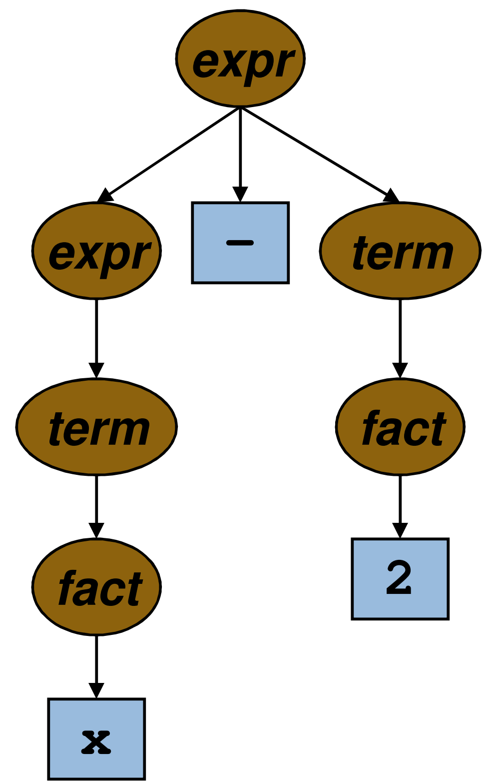
Rule	*Sentential form*	*Input string*
-	*expr*	↑ x – 2 * y
1	*expr* + *term*	↑ x – 2 * y
3	*term* + *term*	↑ x – 2 * y
6	*factor* + *term*	↑ x – 2 * y
8	*<id>* + *term*	x ↑ – 2 * y
?	*<id,x>* + *term*	x ↑ – 2 * y

Undo all these productions

- If we can't match next terminal:
  - Rollback productions
  - Choose a different production for *expr*
  - *Continue*

Retrying

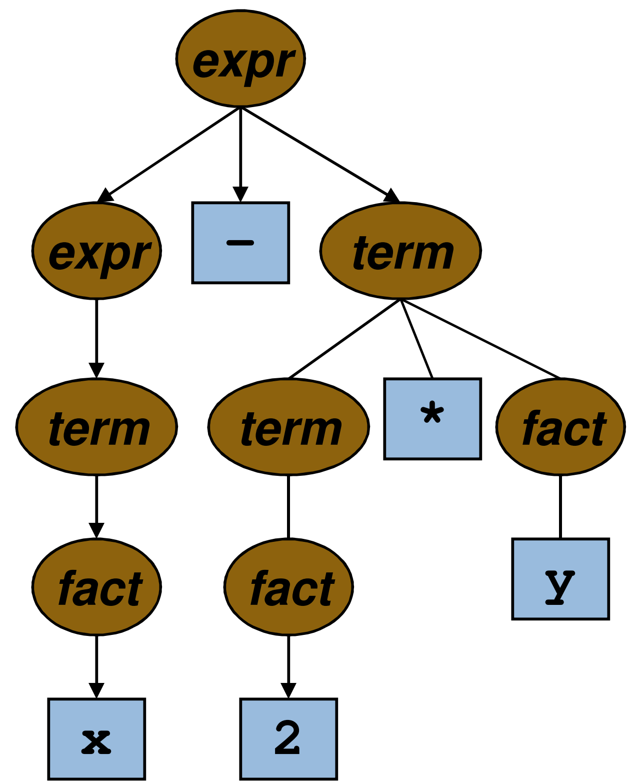
Rule	*Sentential form*	*Input string*
-	*expr*	↑ x – 2 * y
2	*expr* - *term*	↑ x – 2 * y
3	*term* - *term*	↑ x – 2 * y
6	*factor* - *term*	↑ x – 2 * y
8	*<id>* - *term*	x ↑ – 2 * y
-	*<id,x>* - *term*	x – ↑ 2 * y
3	*<id,x>* - *factor*	x – ↑ 2 * y
7	*<id,x>* - *<num>*	x – 2 ↑ * y



- **Problem**:
  - More input to read
  - Another cause of backtracking

Successful parse

Rule	Sentential form	Input string
-	*expr*	↑ `x – 2 * y`
2	*expr - term*	↑ `x – 2 * y`
3	*term - term*	↑ `x – 2 * y`
6	*factor - term*	↑ `x – 2 * y`
8	*<id> - term*	`x` ↑ `– 2 * y`
-	*<id,x> - term*	`x –` ↑ `2 * y`
4	*<id,x> - term * fact*	`x –` ↑ `2 * y`
6	*<id,x> - fact * fact*	`x –` ↑ `2 * y`
7	*<id,x> - <num> * fact*	`x – 2` ↑ `* y`
-	*<id,x> - <num,2> * fact*	`x – 2 *` ↑ `y`
8	*<id,x> - <num,2> * <id>*	`x – 2 * y` ↑



Other possible parses

Rule	Sentential form	Input string
-	*expr*	↑ `x – 2 * y`
2	***expr - term***	↑ `x – 2 * y`
2	***expr - term - term***	↑ `x – 2 * y`
2	***expr - term - term - term***	↑ `x – 2 * y`
2	***expr - term - term - term - term***	↑ `x – 2 * y`

- **Problem**: termination
  - Wrong choice leads to infinite expansion

    (More importantly: without consuming any input!)
  - May not be as obvious as this
  - Our grammar is ***left recursive***

Left recursion

- Formally,

  A grammar is ***left recursive*** if $\exists$ a non-terminal A such that

  A $\rightarrow^*$ A α ***(for some set of symbols α)***

What does $\rightarrow^*$ mean?

A $\rightarrow$ B <u>x</u>
B $\rightarrow$ A <u>y</u>

- **Bad news**:

  Top-down parsers cannot handle left recursion

- **Good news**:

  We can systematically eliminate left recursion

Notation

- Non-terminals
  - Capital letter: A, B, C

- Terminals
  - Lowercase, underline: $\underline{x}$, $\underline{y}$, $\underline{z}$

- Some mix of terminals and non-terminals
  - Greek letters: α, β, γ
  - Example:

#	Production rule
1	$A \rightarrow B \ \underline{+} \ \underline{x}$
1	$A \rightarrow B \ \alpha$

$$\alpha = \underline{+} \ \underline{x}$$

Eliminating left recursion

- Fix this grammar:

#	Production rule
1	*foo* $\rightarrow$ *foo* α
2	$\mid$ β

Language is β followed by zero or more α

- Rewrite as

#	Production rule
1	*foo* $\rightarrow$ β *bar*
2	*bar* $\rightarrow$ α *bar*
3	$\mid$ ε

This production gives you one β

These two productions give you zero or more α

New non-terminal

Back to expressions

- Two cases of left recursion:

#	Production rule
1	*expr* → *expr* + *term*
2	| *expr* - *term*
3	| *term*

#	Production rule
4	*term* → *term* * *factor*
5	| *term* / *factor*
6	| *factor*

- How do we fix these?

#	Production rule
1	*expr* → *term expr2*
2	*expr2* → + *term expr2*
3	| - *term expr2*
4	| ε

#	Production rule
4	*term* → *factor term2*
5	*term2* → * *factor term2*
6	| / *factor term2*
	| ε

Eliminating left recursion

- Resulting grammar
  - All right recursive
  - Retain original language and associativity
  - Not as intuitive to read
- Top-down parser
  - Will always terminate
  - May still backtrack

#	Production rule
1	*expr* → *term expr2*
2	*expr2* → **+** *term expr2*
3	| **-** *term expr2*
4	| ε
5	*term* → *factor term2*
6	*term2* → ***** *factor term2*
7	| / *factor term2*
8	| ε
9	*factor* → `number`
10	| `identifier`

There's a lovely algorithm to do this automatically, which we will skip

Top-down parsers

- ***Problem***: Left-recursion
- ***Solution***: Technique to remove it

- What about backtracking?

  Current algorithm is brute force

- ***Problem***: how to choose the right production?
  - **Idea**: use the next input token *(duh)*
  - How? Look at our right-recursive grammar…

Right-recursive grammar

#	Production rule
1	*expr* → *term expr2*
2	*expr2* → + *term expr2*
3	| - *term expr2*
4	| ε
5	*term* → *factor term2*
6	*term2* → * *factor term2*
7	| / *factor term2*
8	| ε
9	*factor* → `number`
10	| `identifier`

Two productions with no choice at all

All other productions are uniquely identified by a terminal symbol at the start of RHS

- We can choose the right production by looking at the next input symbol
  - This is called ***lookahead***
  - BUT, this can be tricky…

Lookahead

- **Goal**: avoid backtracking
  - Look at future input symbols
  - Use extra context to make right choice
- How much lookahead is needed?
  - In general, an arbitrary amount is needed for the full class of context-free grammars
  - Use fancy-dancy algorithm ***CYK algorithm, $O(n^3)$***
- Fortunately,
  - Many CFGs can be parsed with limited lookahead
  - Covers most programming languages ***not C++ or Perl***

Top-down parsing

- **Goal**:

  Given productions $A \rightarrow \alpha \mid \beta$, the parser should be able to choose between α and β

- Trying to match A

  How can the next input token help us decide?

- **Solution**: ***FIRST*** sets ***(almost a solution)***
  - Informally:

    FIRST(α) is the set of tokens that could appear as the first symbol in a string derived from α
  - ***Def:*** $\underline{x}$ in FIRST(α) iff $\alpha \rightarrow^{*} \underline{x}\ \gamma$

Top-down parsing

- Building FIRST sets

  We'll look at this algorithm later

- The LL(1) property
  - Given $A \rightarrow \alpha$ and $A \rightarrow \beta$, we would like:

    $$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \varnothing$$

    - we will also write $\text{FIRST}(A \rightarrow \alpha)$, defined as $\text{FIRST}(\alpha)$
  - Parser can make right choice by with one lookahead token
  - ..almost..
  - What are we not handling?

Top-down parsing

- What about ε productions?
  - Complicates the definition of LL(1)
  - Consider A → α and A → β and α may be empty
  - In this case there is no symbol to identify α

#	Production rule
1	*S* → *A* <u>z</u>
2	*A* → <u>x</u> B
3	| <u>y</u> C
4	| *ε*

- Example:
  - What is FIRST(#4)?
  - = { ε }
  - What would tells us we are matching production 4?

Top-down parsing

#	Production rule
1	$S \rightarrow A$ z
2	$A \rightarrow$ x B
3	| y C
4	| ε

- If A was empty
  - What will the next symbol be?
  - Must be one of the symbols that immediately *follows* an A
- **Solution**
  - Build a ***FOLLOW*** set for each symbol that could produce ε
  - Extra condition for LL:

FIRST(A→β) *must be disjoint from FIRST*(A→α) *and FOLLOW*(A)

FOLLOW sets

- Example:
  - FIRST(#2) = { x }
  - FIRST(#3) = { y }
  - FIRST(#4) = { ε }

#	Production rule
1	*S* → *A* **z**
2	*A* → **x** **B**
3		**y** **C**
4		*ε*

- What can follow A?
  - Look at the context of all uses of A
  - FOLLOW(A) = { z }
  - Now we can uniquely identify each production:

    If we are trying to match an A and the next token is z, then we matched production 4

FIRST and FOLLOW more carefully

- *Notice*:
  - FIRST and FOLLOW are sets
  - FIRST may contain ε in addition to other symbols

- **Question**:
  - What is FIRST(#2)?
  - = FIRST(B) = { x, y, ε }?
  - and FIRST(C)

- **Question**:

  When would we care about FOLLOW(A)?

  Answer: if FIRST(C) contains ε

#	Production rule
1	*S* → *A* z
2	*A* → B C
3	| D
4	*B* → x
5	| y
6	| *ε*
7	*C* → . . .

LL(1) property

- ***Key idea:***
  - Build parse tree top-down
  - Use look-ahead token to pick next production
  - Each production must be uniquely identified by the terminal symbols that may appear at the start of strings derived from it.
- ***Def**:* FIRST+($A \rightarrow \alpha$) as
  - FIRST(α) U FOLLOW(A), if $\varepsilon \in$ FIRST(α)
  - FIRST(α), otherwise
- ***Def**:* a grammar is ***LL(1)*** *iff*

  $A \rightarrow \alpha$ and $A \rightarrow \beta$ and

  FIRST+($A \rightarrow \alpha$) $\cap$ FIRST+($A \rightarrow \beta$) $= \varnothing$

Parsing LL(1) grammar

- Given an LL(1) grammar
  - Code: simple, fast routine to recognize each production
  - Given $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, with

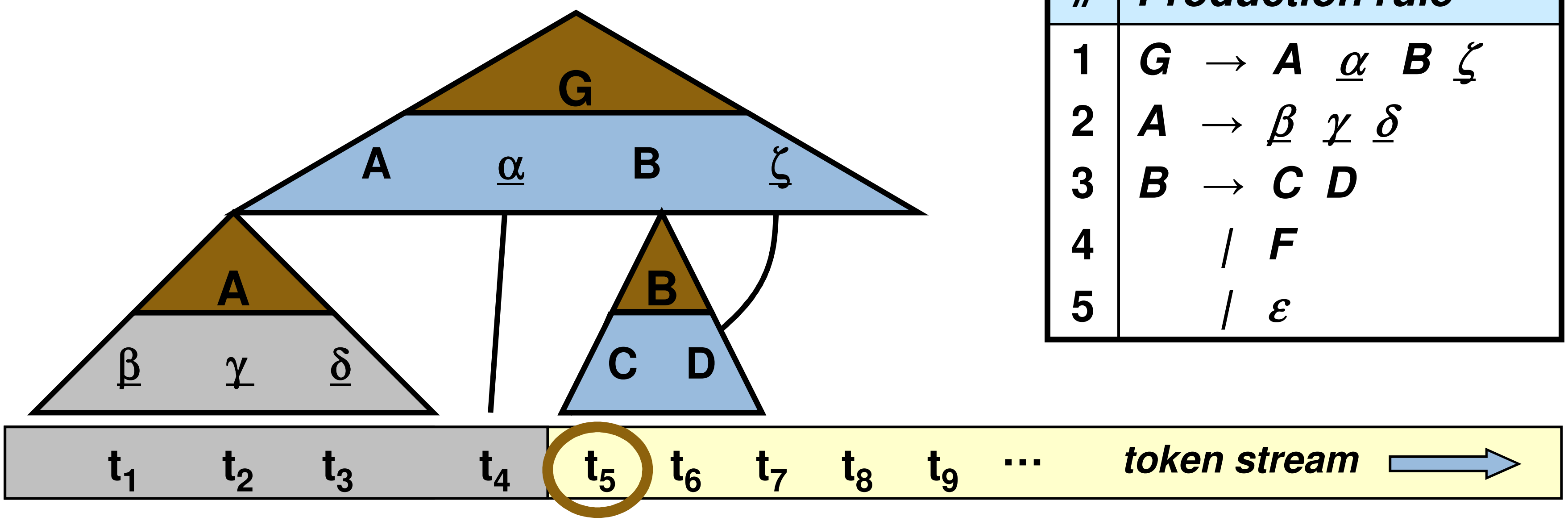
    $\text{FIRST}^+(\beta_i) \cap \text{FIRST}^+(\beta_j) = \varnothing$ *for all i != j*

```
/* find rule for A */
if (current token ∈ FIRST+(β1))
   select A → β1
else if (current token ∈ FIRST+(β2))
   select A → β2
else if (current token ∈ FIRST+(β3))
   select A → β3
else
   report an error and return false
```

Top-down parsing

- Build parse tree top down

#	Production rule
1	$G \rightarrow A\ \underline{\alpha}\ B\ \underline{\zeta}$
2	$A \rightarrow \underline{\beta}\ \underline{\gamma}\ \underline{\delta}$
3	$B \rightarrow C\ D$
4	$\quad \mid F$
5	$\quad \mid \varepsilon$



Is "CD"? **Consider all possible strings derivable from "CD"**
What is the set of tokens that can appear at start?

$t_5 \in$ FIRST(C D)
$t_5 \in$ FIRST(F)
disjoint?
$t_5 \in$ FOLLOW(B)

FIRST and FOLLOW sets

The right-hand side of a production

FIRST(α)

For some $\alpha \in (T \cup NT)^*$, define FIRST(α) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $\underline{x} \in$ FIRST(α) *iff* $\alpha \Rightarrow^* \underline{x}\,\gamma$, for some γ

and $\varepsilon \in$ FIRST(α) *iff* $\alpha \Rightarrow^* \varepsilon$

FOLLOW(A)

For some $A \in NT$, define FOLLOW(A) as the set of symbols that can occur immediately after A in a valid sentence.

FOLLOW(G) = {EOF}, where G is the start symbol

Computing FIRST sets

- *Idea*:

  Use FIRST sets of the right side of production

$$\mathbf{A} \rightarrow \mathbf{B_1} \quad \mathbf{B_2} \quad \mathbf{B_3} \ldots$$

- *Cases*:
  - FIRST(A→B) = FIRST(B_1)
    - What does FIRST(B_1) mean?
    - Union of FIRST($B_1 \rightarrow \gamma$) for all γ
  - What if ε in FIRST(B_1)?

    $\Rightarrow$ FIRST(A→B) $\cup$= FIRST(B_2) ***repeat as needed***
  - What if ε in FIRST(B_i) for all i?

    $\Rightarrow$ FIRST(A→B) $\cup$ = $\{\varepsilon\}$ ***leave {ε} for later***

Why $\cup$ = ?

Algorithm

- For one production: $p = A \rightarrow \beta$

```
if (β is a terminal t)
        FIRST(p) = {t}
else if (β == ε)
        FIRST(p) = {ε}
else
        Given   β = B1 B2 B3 … Bk
        εInAll = true
        for (i ← 1 to k)
                FIRST(p) += FIRST(Bi)  -  {ε}
                if (ε not in FIRST(Bi))
                        εInAll = false
                        break
        if (εInAll) FIRST(p) += {ε}
```

Why do we need to remove ε from FIRST(B_i)?

Algorithm

- For one production:
  - Given **A** $\rightarrow$ $\mathbf{B_1}$ $\mathbf{B_2}$ $\mathbf{B_3}$ $\mathbf{B_4}$ $\mathbf{B_5}$
  - Compute FIRST(**A**→**B**) using FIRST(**B**)
  - How do we get FIRST(**B**)?

- What kind of algorithm does this suggest?
  - Recursive?
  - Like a depth-first search of the productions

- **Problem**:
  - What about recursion in the grammar?
  - **A** → **x B y** and **B** → **z A w**

Algorithm

- **Solution**
  - Start with FIRST(B) empty
  - Compute FIRST(A) using empty FIRST(B)
  - Now go back and compute FIRST(B)
    - What if it's no longer empty?
    - Then we recompute FIRST(A)
    - What if new FIRST(A) is different from old FIRST(A)?
    - Then we recompute FIRST(B) again…
- When do we stop?
  - When no more changes occur – we reach ***convergence***
  - FIRST(A) and FIRST(B) both satisfy equations
- This is another ***fixpoint*** algorithm

Algorithm

- Using fixpoints:

```
forall p   FIRST(p) = {}

while (FIRST sets are changing)
        pick a random p
        compute FIRST(p)
```

- Can we be smarter?
  - Yes, visit in special order
  - Reverse post-order depth first search

    Visit all children (all right-hand sides) before visiting the left-hand side, whenever possible

Example

#	Production rule
1	*goal* → *expr*
2	*expr* → *term expr2*
3	*expr2* → + *term expr2*
4		- *term expr2*
5		ε
6	*term* → *factor term2*
7	*term2* → * *factor term2*
8		/ *factor term2*
9		ε
10	*factor* → `number`
11		`identifier`

FIRST(3) = { + }
FIRST(4) = { − }
FIRST(5) = { ε }

FIRST(7) = { * }
FIRST(8) = { / }
FIRST(9) = { ε }

FIRST(1) = ?

FIRST(1) = FIRST(2)
= FIRST(6)
= FIRST(10) ∪ FIRST(11)
= { `number`, `identifier` }

Computing FOLLOW sets

- *Idea*:

  Push FOLLOW sets down, use FIRST where needed

$$\mathbf{A \rightarrow B_1 \quad B_2 \quad B_3 \quad B_4 \quad \ldots B_k}$$

- *Cases*:
  - What is FOLLOW(B_1)?
    - FOLLOW(B_1) = FIRST(B_2)
    - In general: FOLLOW(B_i) = FIRST(B_{i+1})
  - What about FOLLOW(B_k)?
    - FOLLOW(B_k) = FOLLOW(A)
  - What if $\varepsilon \in$ FIRST(B_k)?

  $\Rightarrow$ FOLLOW(B_{k-1}) $\cup$= FOLLOW(A) *extends to k-2, etc.*

Example

#	Production rule
1	*goal* → *expr*
2	*expr* → *term expr2*
3	*expr2* → + *term expr2*
4	| - *term expr2*
5	| ε
6	*term* → *factor term2*
7	*term2* → * *factor term2*
8	| / *factor term2*
9	| ε
10	*factor* → <u>`number`</u>
11	| <u>`identifier`</u>

FOLLOW(*goal*) = { EOF }

FOLLOW(*expr*) = FOLLOW(*goal*) = { EOF }

FOLLOW(*expr2*) = FOLLOW(*expr*) = { EOF }

FOLLOW(*term*) = ?

FOLLOW(*term*) += FIRST(*expr2*)

+= { +, -, ε }

+= { +, -, FOLLOW(*expr*)}

+= { +, -, EOF }

Example

#	Production rule
1	*goal* → *expr*
2	*expr* → *term expr2*
3	*expr2* → + *term expr2*
4	| - *term expr2*
5	| ε
6	*term* → *factor term2*
7	*term2* → * *factor term2*
8	| / *factor term2*
9	| ε
10	*factor* → number
11	| identifier

FOLLOW(*term2*) += FOLLOW(*term*)

FOLLOW(*factor*) = ?

FOLLOW(*factor*) += FIRST(*term2*)

+= { *, / , ε }

+= { *, / , FOLLOW(*term*)}

+= { *, / , +, -, EOF }

Computing FOLLOW Sets

```
FOLLOW(G) ← {EOF }
for each A ∈ NT, FOLLOW(A) ← Ø

while (FOLLOW sets are still changing)
  for each p ∈ P, of the form A→ … B1B2…Bk
    FOLLOW(Bk) ← FOLLOW(Bk) ∪ FOLLOW(A)
    TRAILER ← FOLLOW(A)
     for i ← k down to 2
       if ε ∈ FIRST(Bi ) then
            TRAILER ← TRAILER ∪ ( FIRST(Bi ) – { ε } )
       else
            TRAILER ← FIRST(Bi)
       FOLLOW(Bi-1 ) ← FOLLOW(Bi-1) ∪ TRAILER
```

LL(1) property

- ***Def***: a grammar is LL(1) iff
  $A \rightarrow \alpha$ and $A \rightarrow \beta$ and
  $\text{F}{\scriptstyle\text{IRST}}+(A \rightarrow \alpha) \cap \text{F}{\scriptstyle\text{IRST}}+(A \rightarrow \beta) = \varnothing$
- **Problem**
  - What if my grammar is not LL(1)?
  - May be able to fix it, with transformations
- Example:

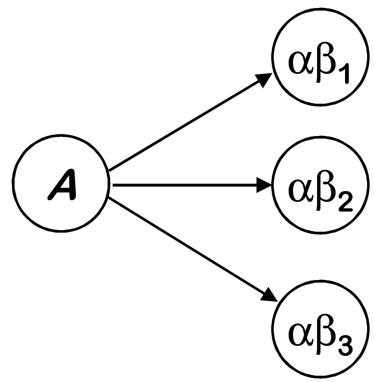
#	Production rule
1	$A \rightarrow \underline{\alpha}\ \beta_1$
2	$\quad\mid \underline{\alpha}\ \beta_2$
3	$\quad\mid \underline{\alpha}\ \beta_3$

#	Production rule
1	$A \rightarrow \underline{\alpha}\ Z$
2	$Z \rightarrow \beta_1$
3	$\quad\mid \beta_2$
4	$\quad\mid \beta_3$

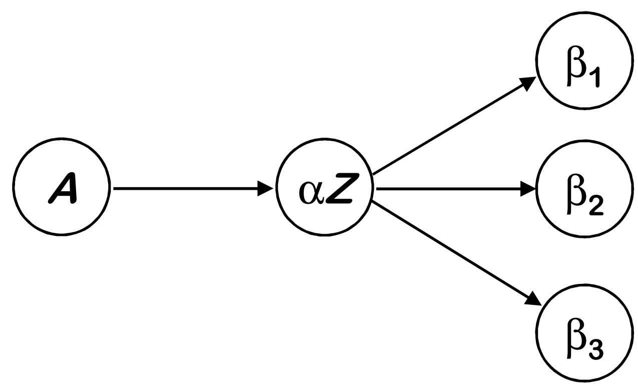
Left factoring

- Graphically

#	*Production rule*
1	$A \rightarrow \alpha\ \beta_1$
2	$\mid\ \alpha\ \beta_2$
3	$\mid\ \alpha\ \beta_3$



#	*Production rule*
1	$A \rightarrow \alpha\ Z$
2	$Z \rightarrow \beta_1$
3	$\mid\ \beta_2$
	$\mid\ \beta_3$



Expression example

#	*Production rule*
1	*factor* → identifier
2	| identifier [*expr*]
3	| identifier (*expr*)

First+(1) = {identifier}

First+(2) = {identifier}

First+(3) = {identifier}

After left factoring:

#	*Production rule*
1	*factor* → identifier *post*
2	*post* → [*expr*]
3	| (*expr*)
4	| ε

First+(1) = {identifier}

First+(2) = { [}

First+(3) = { (}

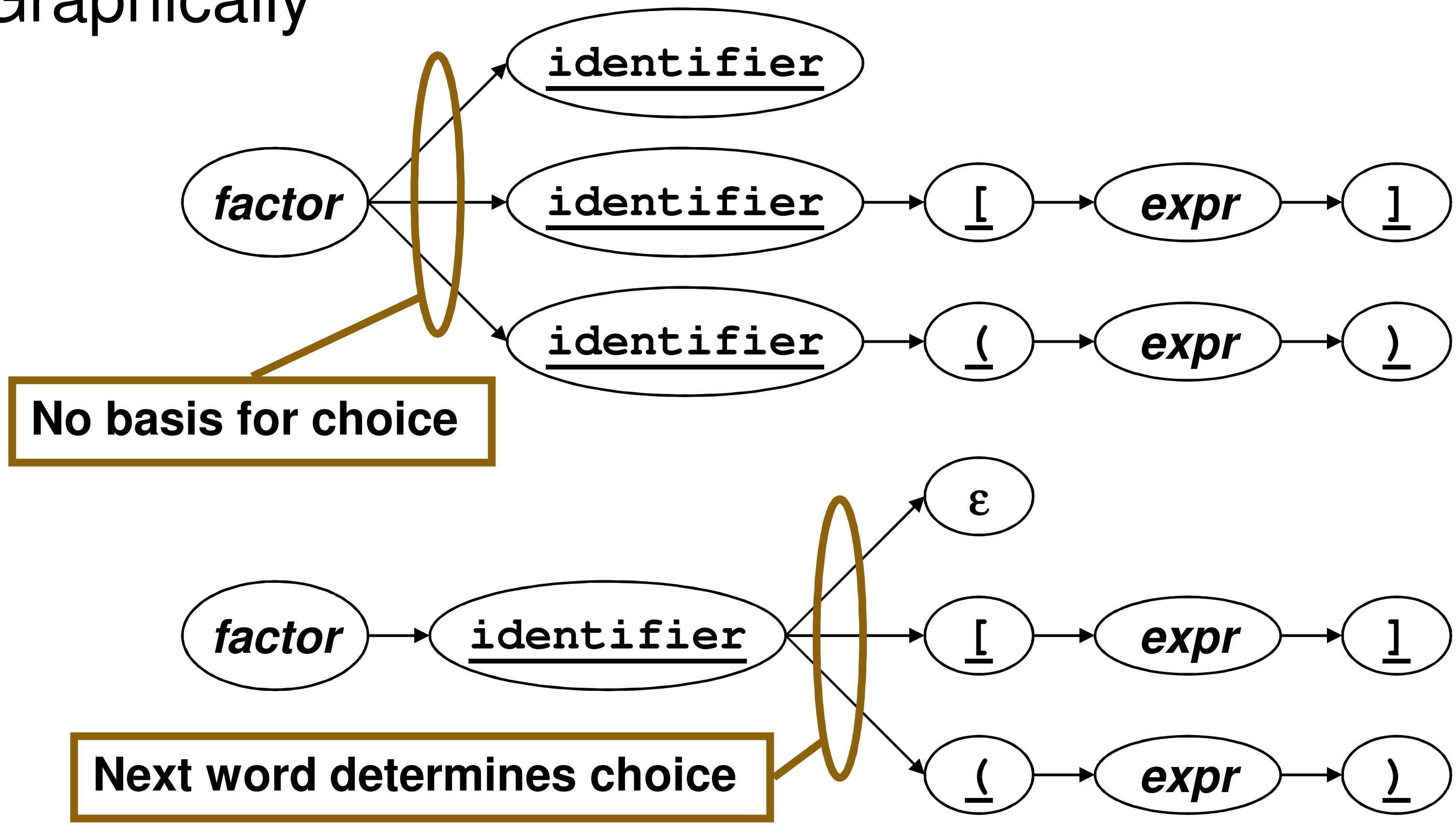
First+(4) = ?

= Follow(*post*)
= {operators}

In this form, it has LL(1) property

Left factoring

- Graphically



Left factoring

- ***Question***

  Using left factoring and left recursion elimination, can we turn an arbitrary CFG to a form where it meets the LL(1) condition?

- ***Answer***

  Given a CFG that does not meet LL(1) condition, it is ***undecidable*** whether or not an LL(1) grammar exists

- ***Example***

  $\{a^n 0\, b^n \mid n \geq 1\} \cup \{a^n 1\, b^{2n} \mid n \geq 1\}$ has no *LL(1)* grammar

  aaa0bbb

  aaa1bbbbbb

Limits of LL(1)

- No LL(1) grammar for this language:

$\{a^n 0\, b^n \mid n \geq 1\} \cup \{a^n 1\, b^{2n} \mid n \geq 1\}$ has no *LL(1)* grammar

#	*Production rule*
1	*G* → a *A* b
2	/ a *B* bb
3	*A* → a *A* b
4	/ 0
5	*B* → a *B* bb
6	/ 1

<u>Problem</u>: need an unbounded number of <u>a</u> characters before you can determine whether you are in the A group or the B group

Predictive parsing

- ***Predictive parsing***
  - The parser can “predict” the correct expansion
  - Using lookahead and FIRST and FOLLOW sets
- Two kinds of predictive parsers
  - Recursive descent

    Often hand-written
  - Table-driven

    Generate tables from First and Follow sets

Recursive descent

#	Production rule
1	*goal* → *expr*
2	*expr* → *term expr2*
3	*expr2* → + *term expr2*
4	| - *term expr2*
5	| ε
6	*term* → *factor term2*
7	*term2* → * *factor term2*
8	| / *factor term2*
9	| ε
10	*factor* → `number`
11	| `identifier`
12	| (*expr*)

- This produces a parser with six *mutually recursive* routines:
  - *Goal*
  - *Expr*
  - *Expr2*
  - *Term*
  - *Term2*
  - *Factor*
- Each recognizes one *NT* or *T*
- The term *descent* refers to the direction in which the parse tree is built.

Example code

- Goal symbol:

```
main()
  /* Match goal --> expr */
  tok = nextToken();
  if (expr() && tok == EOF)
    then proceed to next step;
    else return false;
```

- Top-level expression

```
expr()
  /* Match expr --> term expr2 */
  if (term() && expr2());
    return true;
  else return false;
```

Example code

- Match expr2

```
expr2()
  /* Match expr2 --> + term expr2 */
  /* Match expr2 --> - term expr2 */

  if (tok == '+' or tok == '-')
    tok = nextToken();
    if (term())
      then if (expr2())
              return true;
      else return false;

  /* Match expr2 --> empty */
  return true;
```

Check FIRST and FOLLOW sets to distinguish

Example code

```
factor()
  /* Match factor --> ( expr ) */
  if (tok == `(`)
    tok = nextToken();
    if (expr() && tok == `)')
      return true;
    else
      syntax error: expecting )
      return false

  /* Match factor --> num */
  if (tok is a num)
    return true

  /* Match factor --> id */
  if (tok is an id)
    return true;
```

Top-down parsing

- So far:
  - Gives us a yes or no answer
  - Is that all we want?
  - We want to build the parse tree
  - How?

- Add actions to matching routines
  - Create a node for each production
  - How do we assemble the tree?

Building a parse tree

- Notice:
  - Recursive calls match the shape of the tree

```
main
  expr
    term
      factor
    expr2
      term
```

- **Idea**: use a stack
  - Each routine:
    - Pops off the children it needs
    - Creates its own node
    - Pushes that node back on the stack

Building a parse tree

- With stack operations

```
expr()
  /* Match expr --> term expr2 */
  if (term() && expr2())
    expr2_node = pop();
    term_node = pop();
    expr_node = new exprNode(term_node,
                             expr2_node)
    push(expr_node);
    return true;
  else return false;
```

Generating (automatically) a top-down parser

#	Production rule
1	*goal* → *expr*
2	*expr* → *term expr2*
3	*expr2* → **+** *term expr2*
4	| **-** *term expr2*
5	| ε
6	*term* → *factor term2*
7	*term2* → * *factor term2*
8	| / *factor term2*
9	| ε
10	*factor* → `number`
11	| `identifier`

- Two pieces:
  - Select the right RHS
  - Satisfy each part
- First piece:
  - FIRST+() for each rule
  - Mapping:

    $NT \times \Sigma \rightarrow \textit{rule\#}$

    Look familiar? Automata?

Generating (automatically) a top-down parser

#	Production rule
1	*goal* → *expr*
2	*expr* → *term expr2*
3	*expr2* → + *term expr2*
4	| - *term expr2*
5	| *ε*
6	*term* → *factor term2*
7	*term2* → * *factor term2*
8	| / *factor term2*
9	| *ε*
10	*factor* → `number`
11	| `identifier`

- Second piece
  - Keep track of progress
  - Like a depth-first search
  - Use a stack
- **Idea**:
  - Push *Goal* on stack
  - Pop stack:
    - Match terminal symbol, *or*
    - Apply NT mapping, push RHS on stack

This will be clearer once we see the algorithm

Table-driven approach

- Encode mapping in a table
  - Row for each non-terminal
  - Column for each terminal symbol

  Table[NT, symbol] = rule#

  if symbol $\in$ FIRST+(NT -> rhs(#))

	+,-	***, /**	**id, num**
expr2	***term expr2***	error	error
term2	ε	***factor term2***	error
factor	error	error	***(do nothing)***

Code

```
push the start symbol, G, onto Stack
top ← top of Stack
loop forever
  if top = EOF and token = EOF then break & report success
  if top is a terminal then
     if top matches token then
        pop Stack                           // recognized top
        token ← next_token()
  else                                      // top is a non-terminal
     if TABLE[top,token] is A→ B1B2…Bk then
        pop Stack                           // get rid of A
        push Bk, Bk-1, …, B1                // in that order
  top ← top of Stack
```

Missing else's for error conditions