Compilers

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

Today's Lecture

- Parsing (syntactic analysis)

Parsing

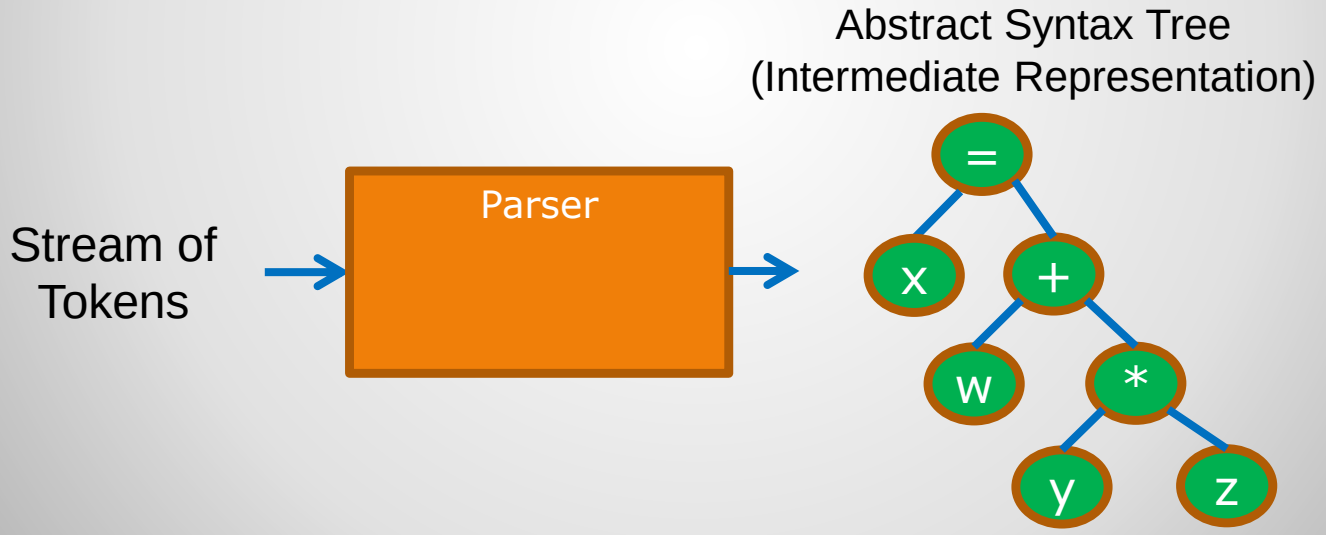
- Parsing - The process of constructing a derivation from a specific input sentence.

- Taken from Engineering a Compiler 2nd edition by Cooper and Torczon, 2012.

Syntactic Analysis

- **Syntatic Analysis** – Check if the source code conforms to the rules of the language.
- The rules of the language are defined using a context-free grammar.

- **Parser** - The parser takes a stream of tokens as input and produces an abstract syntax tree.
- The parser's input stream of tokens is generated by the scanner during the lexical analysis phase.



Parser

- Parse tree examples…

Parsing Tree Examples

Parsing Example 1

- Productions

A → id = Exp

Exp → Exp Op id | id

Op → + | - | * | /

- Parse the following: **w = x + y – z**

Terminals: id, =, +, -, *, /
Nonterminals: A, Exp, Op
Start: A

Parsing Example 1

- Productions

A → id = Exp
Exp → Exp Op id | id
Op → + | - | * | /
Parse the following: **w = x + y – z**

Terminals: id, =, +, -, *, /
Nonterminals: A, Exp, Op
Start: A

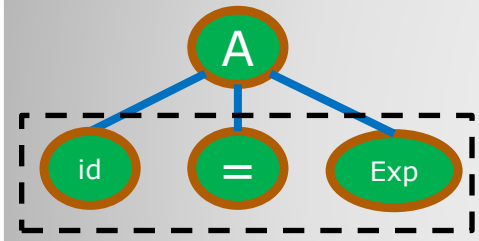
Start with A

- Productions

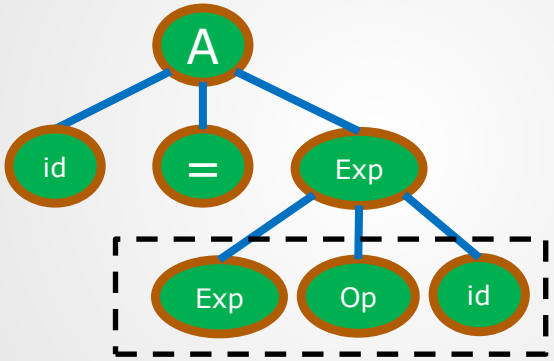
A → id = Exp
Exp → Exp Op id | id
Op → + | - | * | /
Parse the following: **w = x + y – z**

Terminals: id, =, +, -, *, /
Nonterminals: A, Exp, Op
Start: A

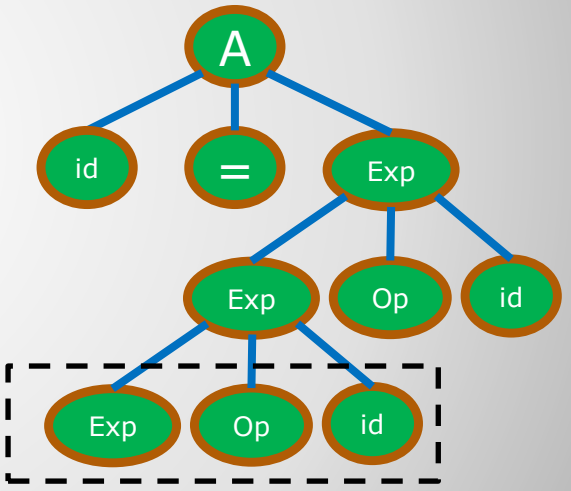
A→id=Exp



Exp→ Exp Op id



Exp →Exp Op id



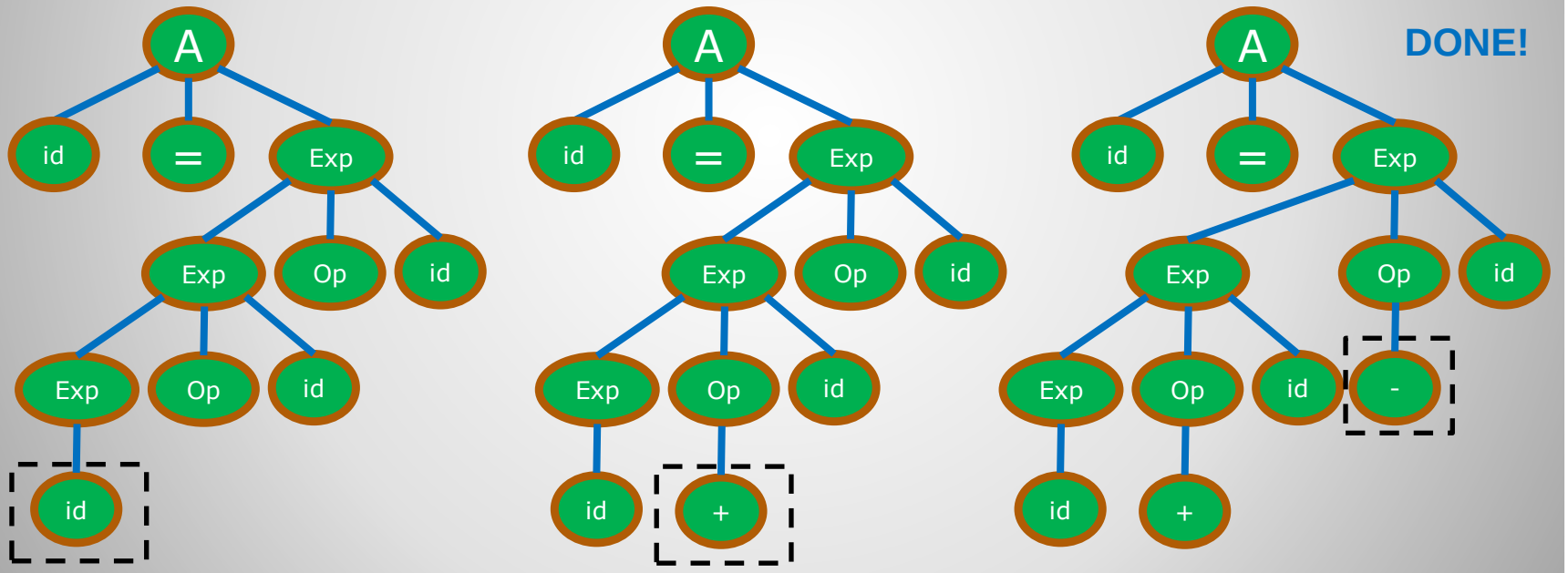
Parsing Example 1

- Productions

A → id = Exp
Exp → Exp Op id | id
Op → + | - | * | /
Parse the following: **w = x + y – z**

Terminals: id, =, +, -, *, /
Nonterminals: A, Exp, Op
Start: A

Exp→id **Op→+** **Op→-** **DONE!**



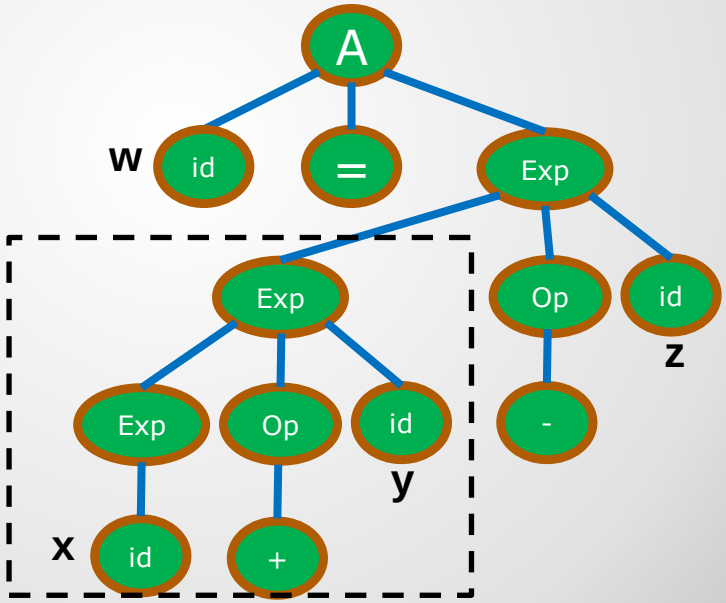
Parsing Example 1

- Productions

A → id = Exp
Exp → Exp Op id | id
Op → + | - | * | /
Parse the following: **w = x + y – z**

Terminals: id, =, +, -, *, /
Nonterminals: A, Exp, Op
Start: A

Variables w, x, y, z will respectively be associated with the id nodes



x+y will be evaluated first. z will be subtracted from the result of x+y because of the tree structure.

Parsing Example 1 - Observations

Parsing Example 2

- Productions

A → id = Exp

Exp → Exp Op id | id

Op → + | - | * | /

- Parse the following: **w = x + y * z**

Multiply (*) here instead of subtract (-)

Parsing Example 2

- Productions

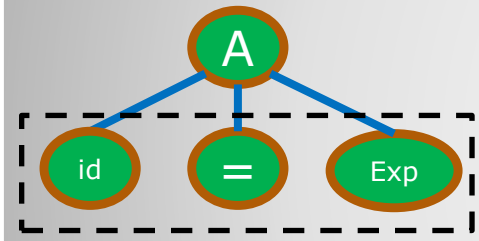
A → id = Exp
Exp → Exp Op id | id
Op → + | - | * | /
Parse the following: **w = x + y * z**

Start with A

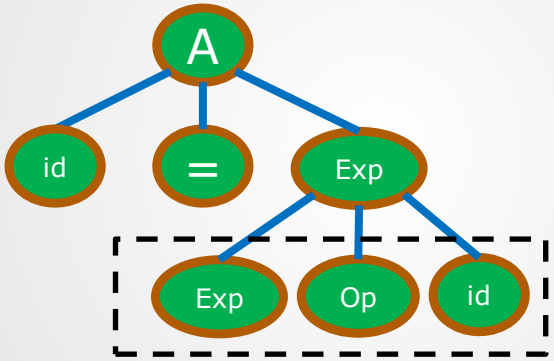
- Productions

A → id = Exp
Exp → Exp Op id | id
Op → + | - | * | /
Parse the following: **w = x + y * z**

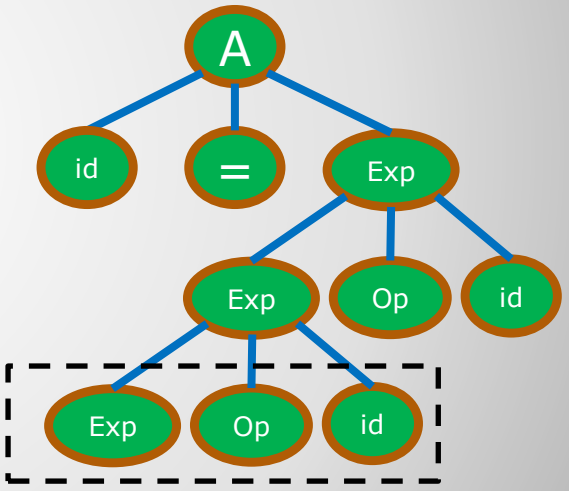
A→id=Exp;



Exp→ Exp Op id



Exp →Exp Op id

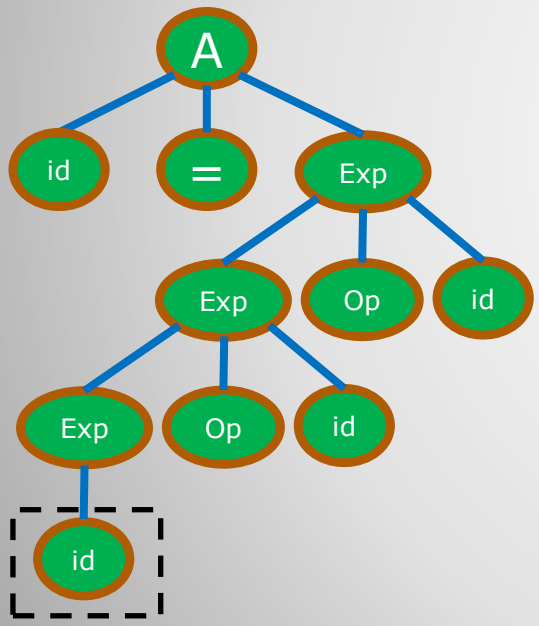


Parsing Example 2

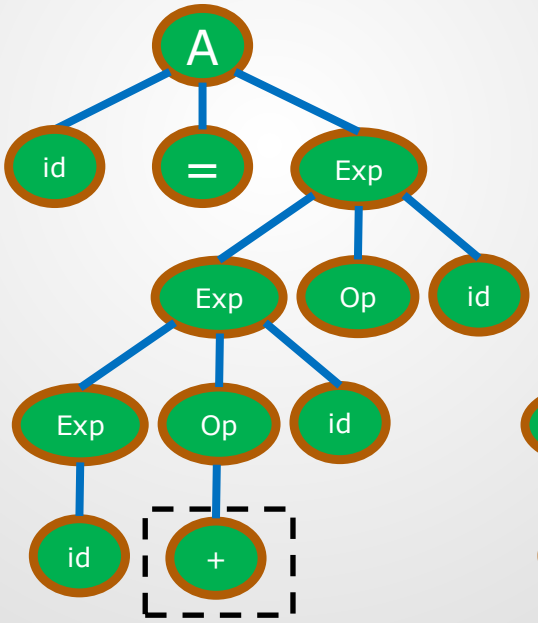
- Productions

A → id = Exp
Exp → Exp Op id | id
Op → + | - | * | /
Parse the following: **w = x + y * z**

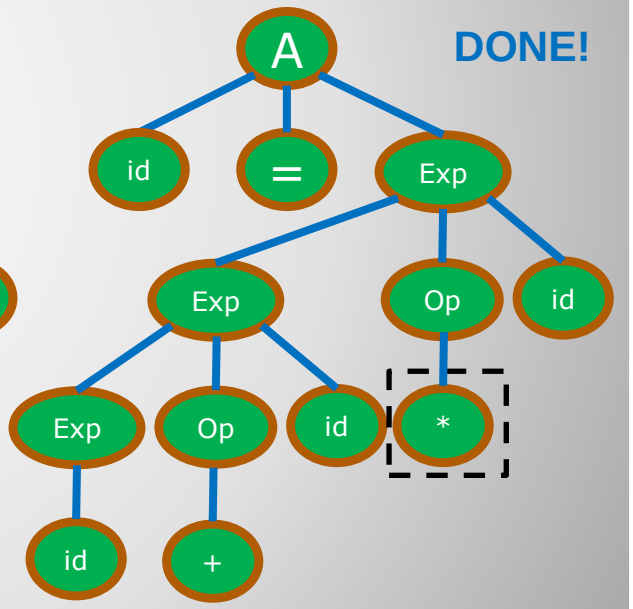
Exp→id



Op→+



Op→-



DONE!

Parsing Example 2

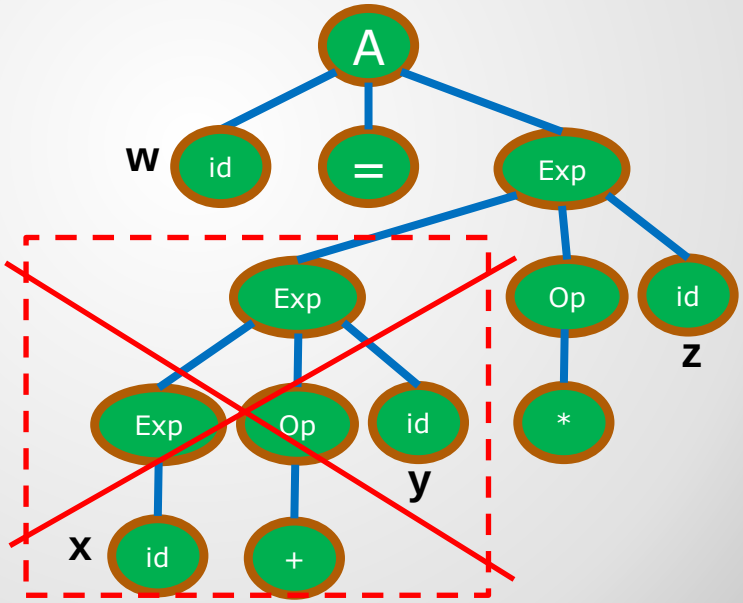
Parsing Example 2 - Observations

- Productions

A → id = Exp
Exp → Exp Op id | id
Op → + | - | * | /
Parse the following: **w = x + y * z**

Having + and * defined at the same level in the grammar means they have the same precedence

Variables w, x, y, z will respectively be associated with the id nodes



ERROR

x+y will STILL be evaluated first. This does not follow the normal order of operations. This is incorrect because z will be multiplied with the result of x+y (because of the tree structure). Need a grammar that expresses the correct precedence.

Grammar for Correct Expression Precedence

- Productions

A → id = Exp
Exp → Exp + Term | Exp – Term | Term
Term → Term * Fact | Term / Fact | Fact
Fact → id | num

- Parse the following: **w = x + y * z**

This grammar gives higher precedence to * and /.

Operations that are deeper in the parse tree have higher precedence.

Parsing Example 3

- Productions

A → id = Exp

Exp → Exp + Term | Exp – Term | Term

Term → Term * Fact | Term / Fact | Fact

Fact → id | num

Parse the following: **w = x + y * z**

Start with A

- Productions

A → id = Exp

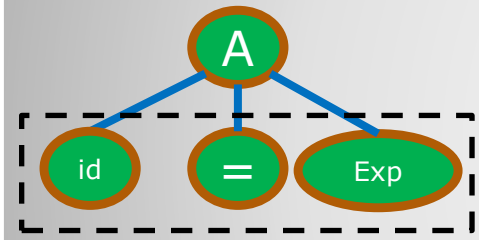
Exp → Exp + Term | Exp – Term | Term

Term → Term * Fact | Term / Fact | Fact

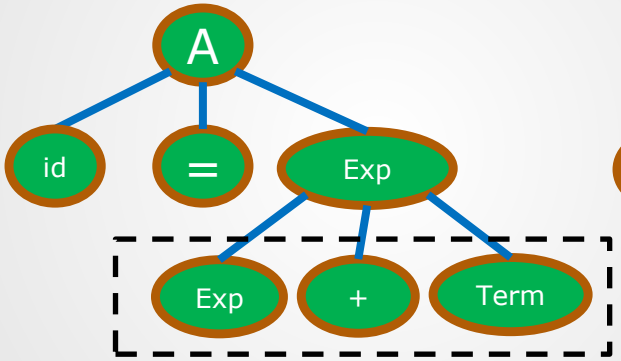
Fact → id | num

Parse the following: **w = x + y * z**

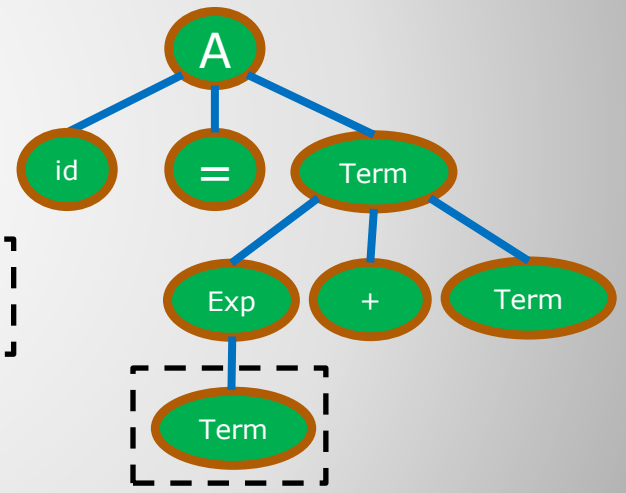
A→id=Exp;



Exp→ Exp + Term



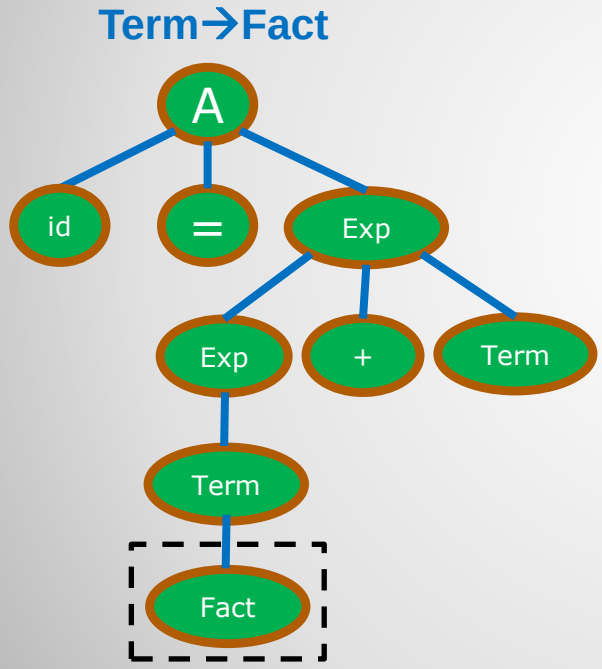
Exp→Term

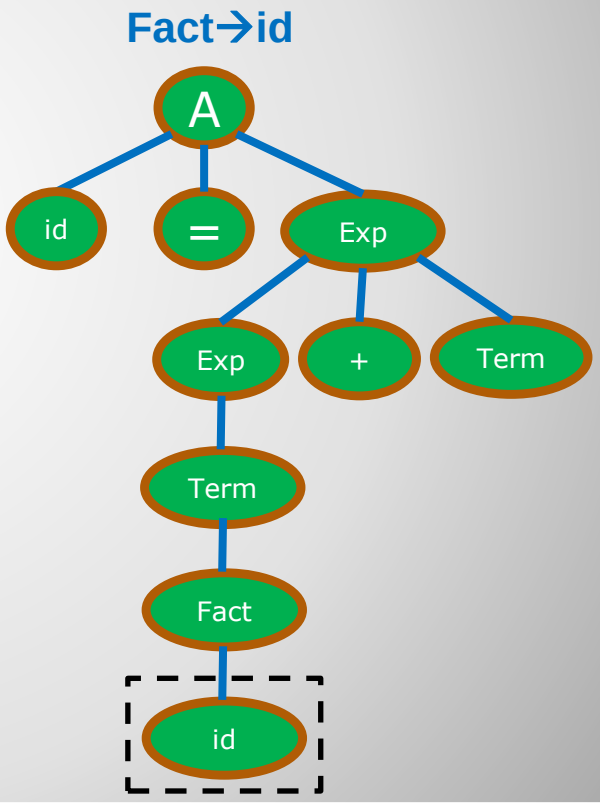


Parsing Example 3

- Productions

A → id = Exp
Exp → Exp + Term | Exp – Term | Term
Term → Term * Fact | Term / Fact | Fact
Fact → id | num
Parse the following: **w = x + y * z**

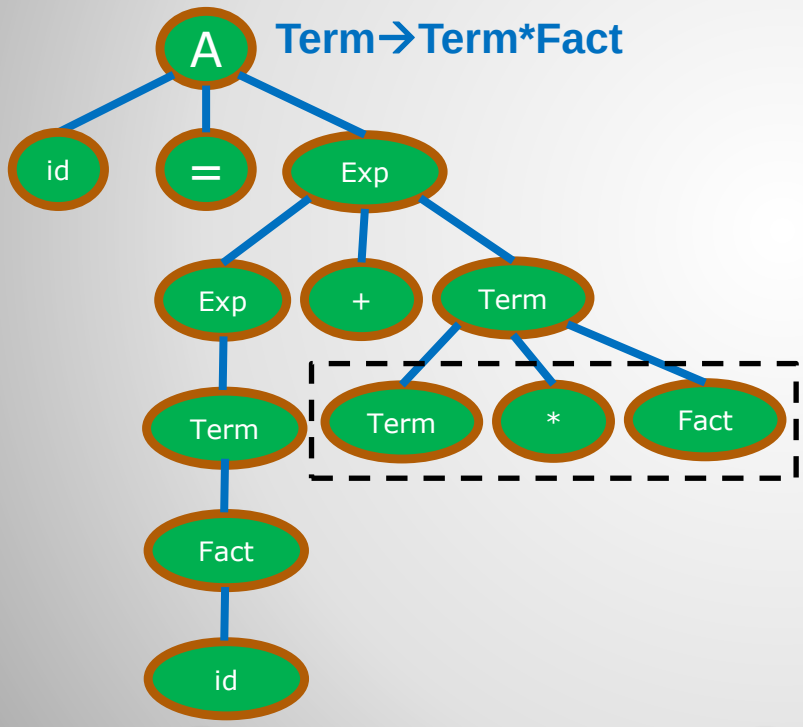


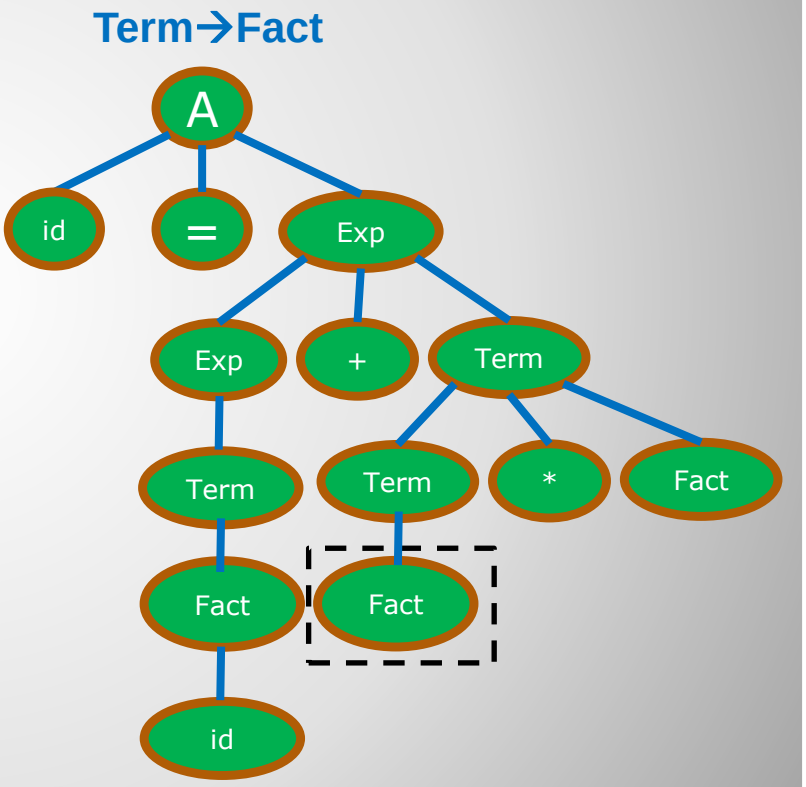


Parsing Example 3

- Productions

A → id = Exp
Exp → Exp + Term | Exp – Term | Term
Term → Term * Fact | Term / Fact | Fact
Fact → id | num
Parse the following: **w = x + y * z**





Parsing Example 3

- Productions

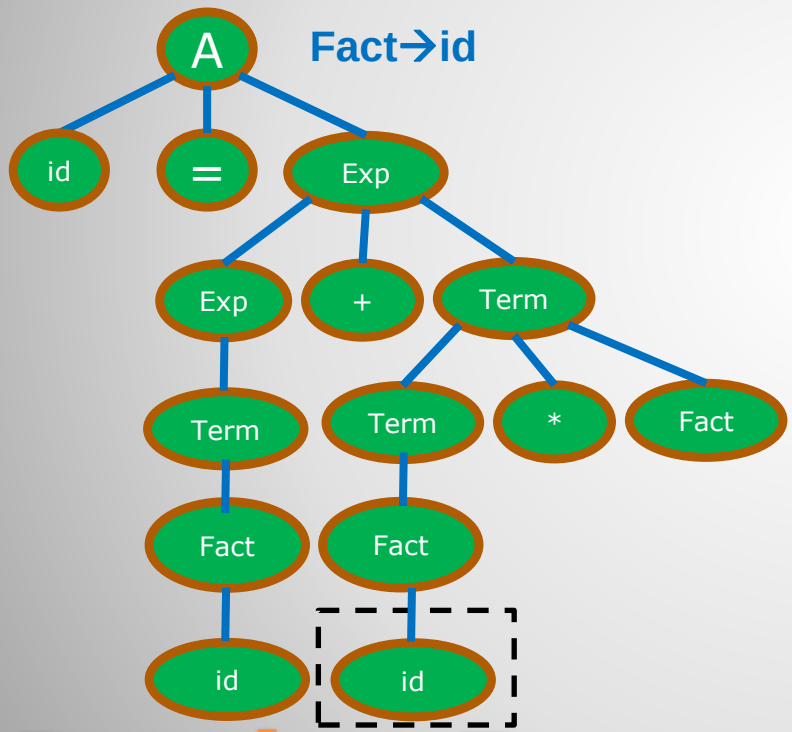
A → id = Exp

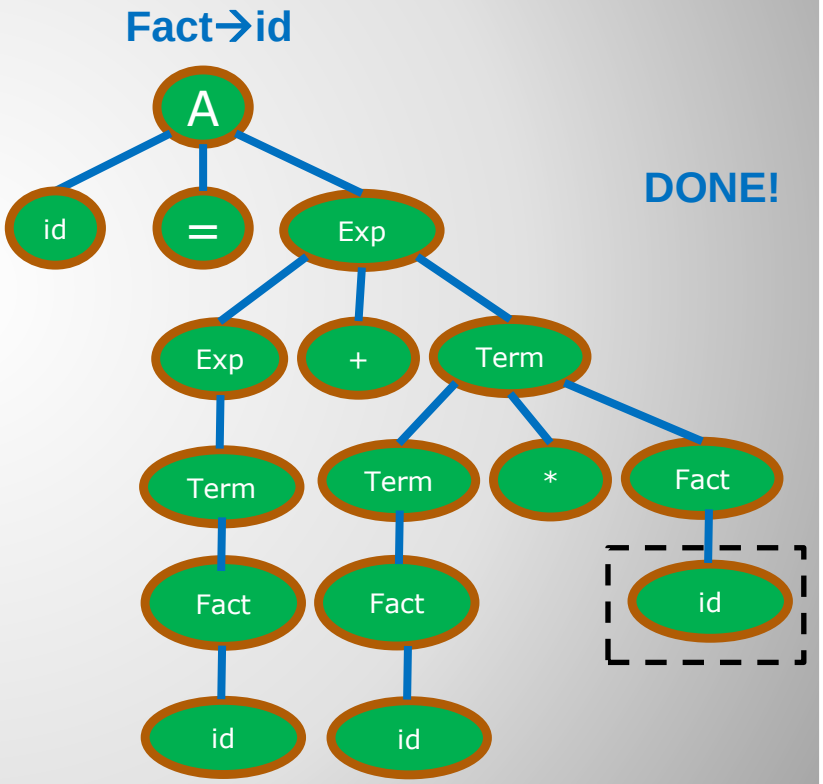
Exp → Exp + Term | Exp – Term | Term

Term → Term * Fact | Term / Fact | Fact

Fact → id | num

Parse the following: **w = x + y * z**



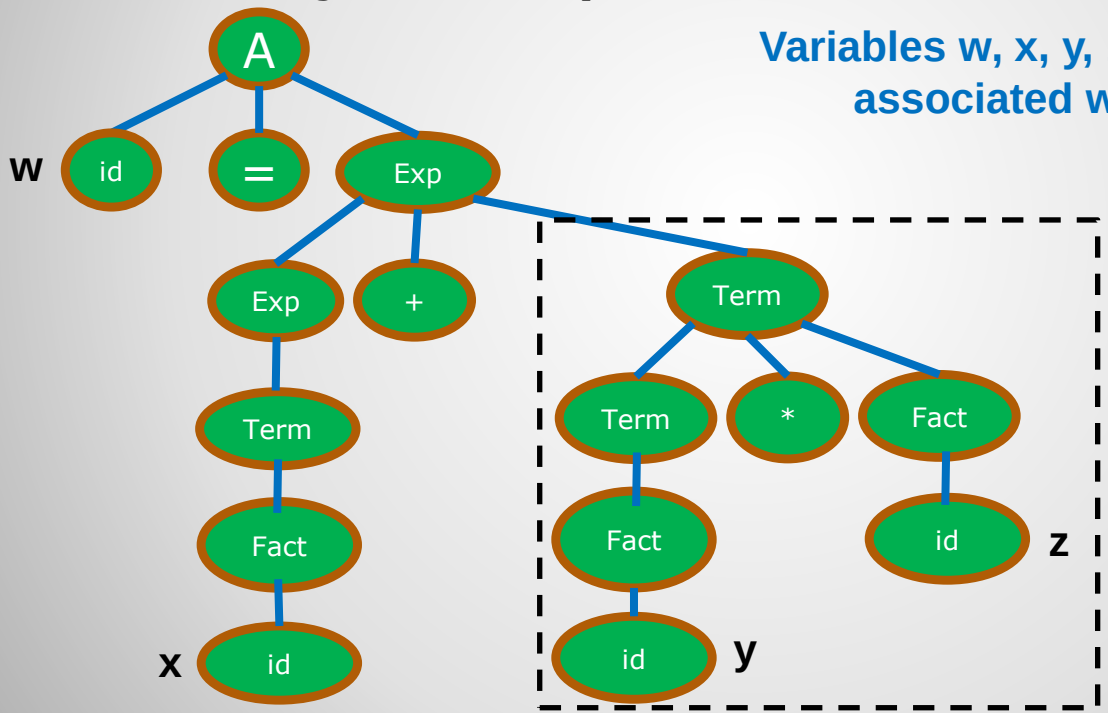


Parsing Example 3

- Productions

A → id = Exp
Exp → Exp + Term | Exp – Term | Term
Term → Term * Fact | Term / Fact | Fact
Fact → id | num
Parse the following: **w = x + y * z**

*** And / are deeper in the parse tree compared to + and -**



Variables w, x, y, z will respectively be associated with the id nodes

y*z will be evaluated first. x will be added to the result of y*z because of the tree structure.

This is because the + operator needs a value for its rhs (it must wait for the value from the *)

Parsing Example 3 - Observations

Add Parenthesis to the Expression Grammar

- Productions

A → id = Exp

Exp → Exp + Term | Exp – Term | Term

Term → Term * Fact | Term / Fact | Fact

Fact → id | num | (Exp)

- Parse the following: **w = (x + y) * z**

Add parenthesis to the grammar to create higher precedence for + and – when necessary.

Adding the parenthesis at a deeper level in the grammar compared to Exp and Term gives the parenthesis higher precedence.

Parsing Example 4

- Productions

A → id = Exp

Exp → Exp + Term | Exp – Term | Term

Term → Term * Fact | Term / Fact | Fact

Fact → id | num | (Exp)

Parse the following: **w = (x + y) * z**

Start with A

- Productions

A → id = Exp

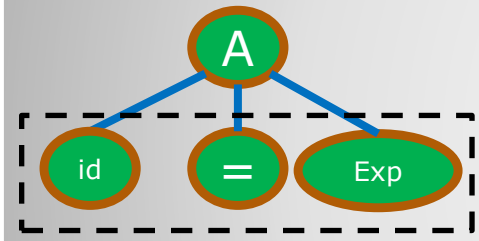
Exp → Exp + Term | Exp – Term | Term

Term → Term * Fact | Term / Fact | Fact

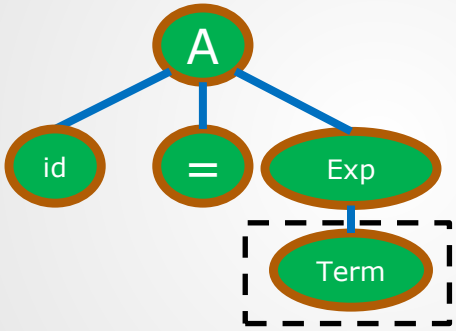
Fact → id | num | (Exp)

Parse the following: **w = (x + y) * z**

A→id=Exp;

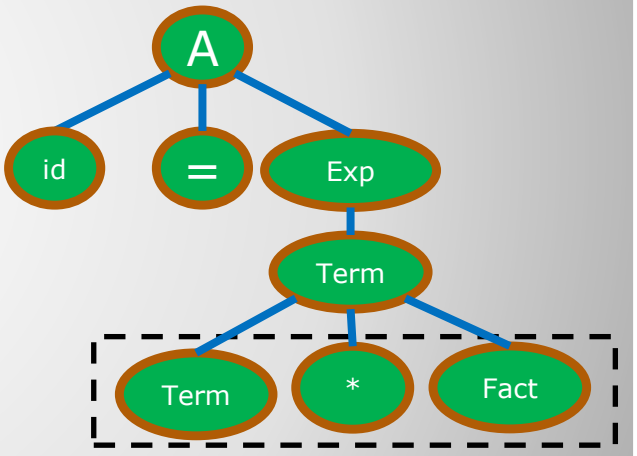


Exp→ Term



(x+y)*z is the Term

Term→Term*Fact



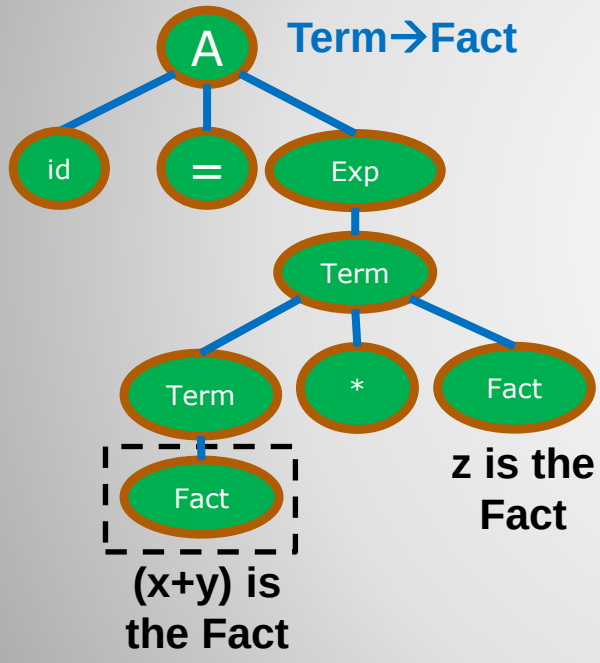
(x+y) is the Term

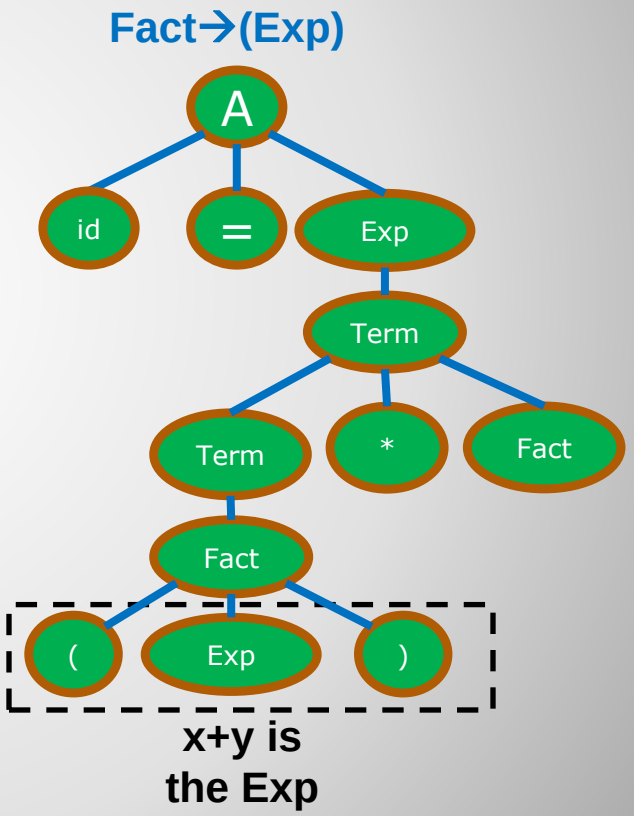
z is the Fact

Parsing Example 4

- Productions

A → id = Exp
Exp → Exp + Term | Exp – Term | Term
Term → Term * Fact | Term / Fact | Fact
Fact → id | num | (Exp)
Parse the following: **w = (x + y) * z**

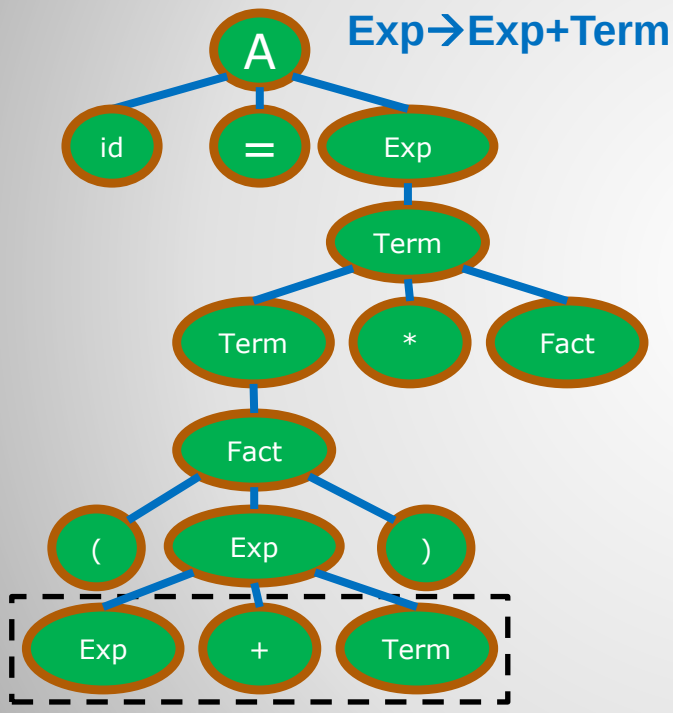


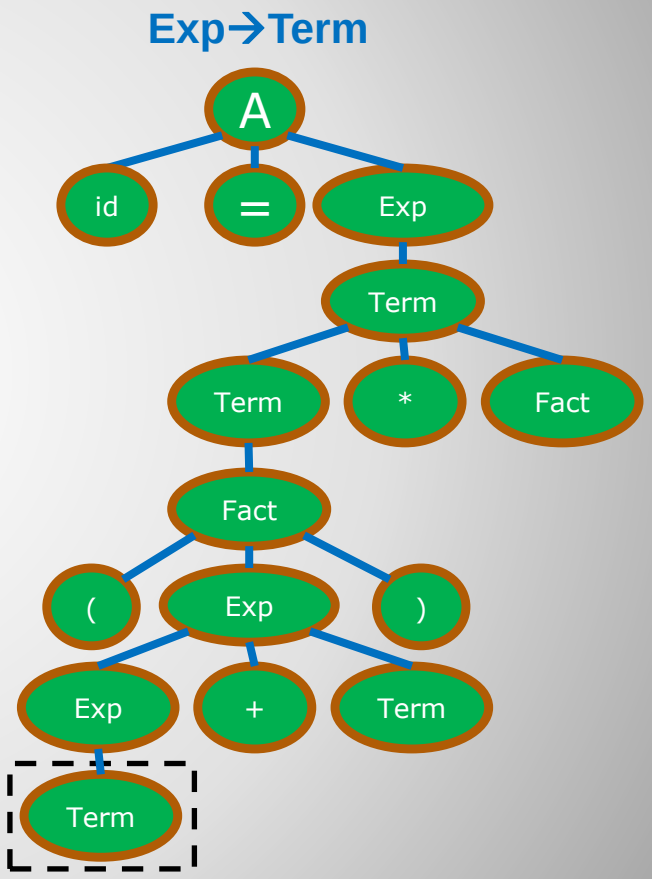


Parsing Example 4

- Productions

A → id = Exp
Exp → Exp + Term | Exp – Term | Term
Term → Term * Fact | Term / Fact | Fact
Fact → id | num | (Exp)
Parse the following: **w = (x + y) * z**

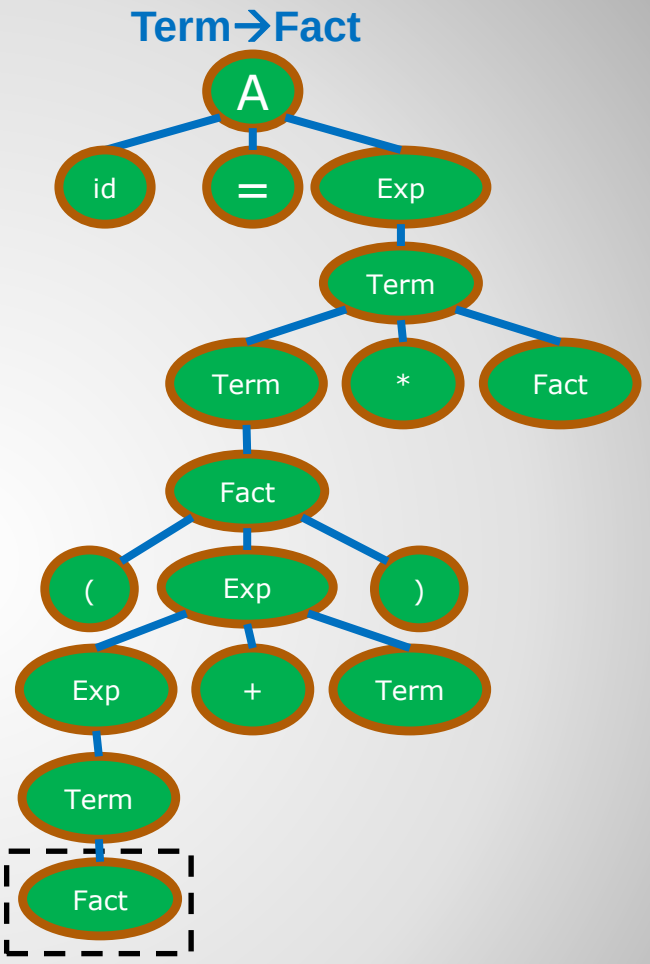




Parsing Example 4

- Productions

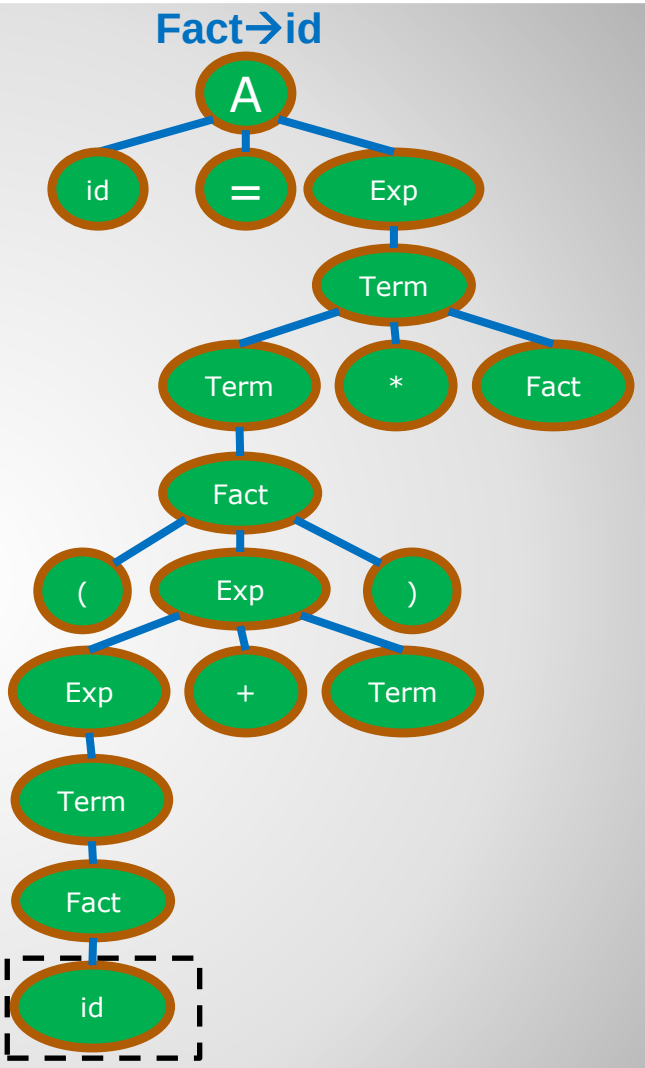
A → id = Exp
Exp → Exp + Term | Exp – Term | Term
Term → Term * Fact | Term / Fact | Fact
Fact → id | num | (Exp)
Parse the following: **w = (x + y) * z**



Parsing Example 4

- Productions

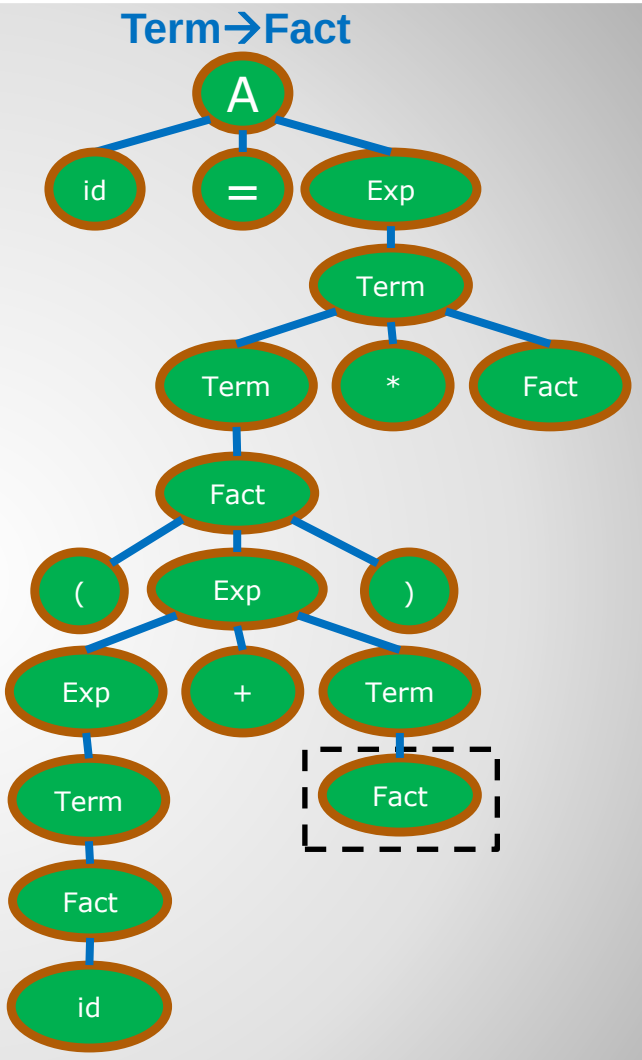
A → id = Exp
Exp → Exp + Term | Exp – Term | Term
Term → Term * Fact | Term / Fact | Fact
Fact → id | num | (Exp)
Parse the following: **w = (x + y) * z**



Parsing Example 4

- Productions

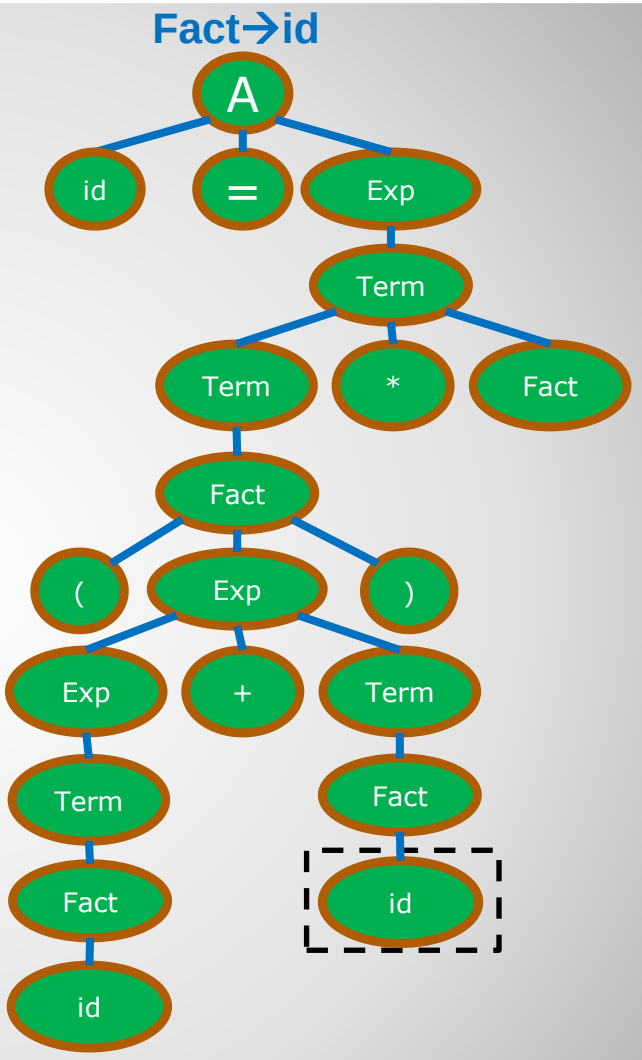
A → id = Exp
Exp → Exp + Term | Exp – Term | Term
Term → Term * Fact | Term / Fact | Fact
Fact → id | num | (Exp)
Parse the following: **w = (x + y) * z**



Parsing Example 4

- Productions

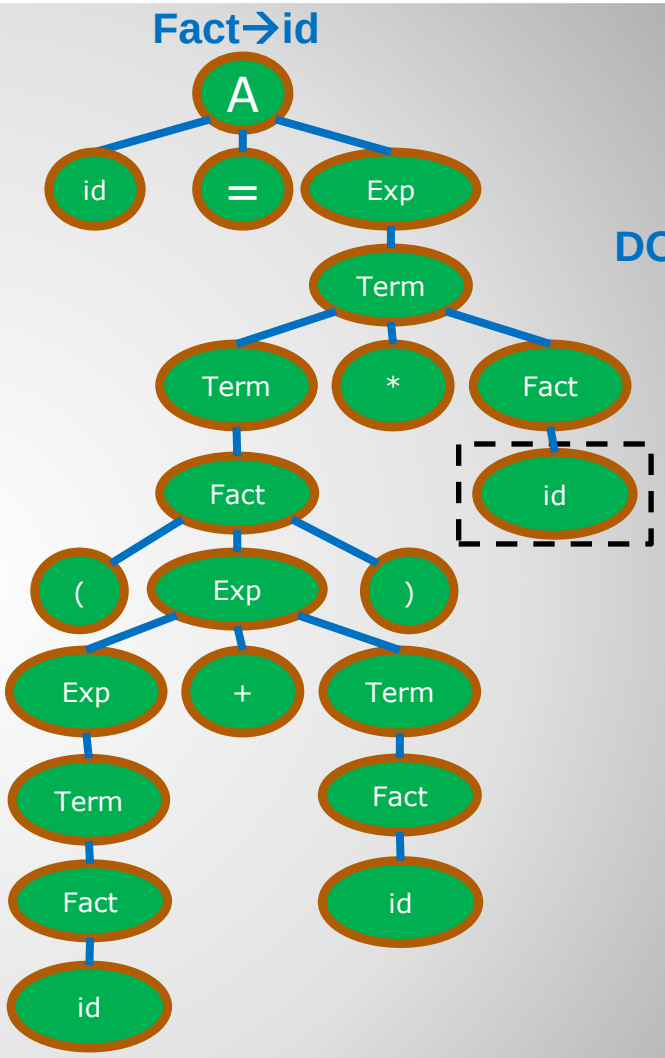
A → id = Exp
Exp → Exp + Term | Exp – Term | Term
Term → Term * Fact | Term / Fact | Fact
Fact → id | num | (Exp)
Parse the following: **w = (x + y) * z**



Parsing Example 4

- Productions

A → id = Exp
Exp → Exp + Term | Exp – Term | Term
Term → Term * Fact | Term / Fact | Fact
Fact → id | num | (Exp)
Parse the following: **w = (x + y) * z**

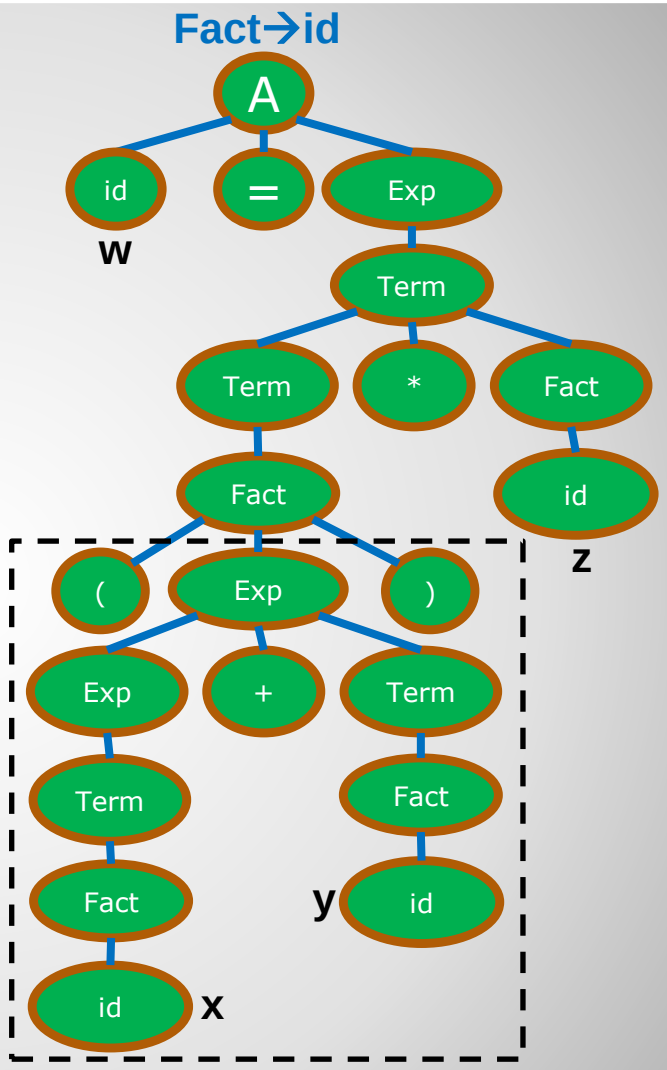


DONE!

Parsing Example 4

- Productions

A → id = Exp
Exp → Exp + Term | Exp – Term | Term
Term → Term * Fact | Term / Fact | Fact
Fact → id | num | (Exp)
Parse the following: **w = (x + y) * z**



Variables w, x, y, z will respectively be associated with the id nodes

(x+y) will be evaluated first. z will be multiplied by the result of (x+y) because of the tree structure. The * operator must wait for its lhs value to be computed.

Parsing Example 4 - Observations

Left Recursion

- **Left recursion** – The first symbol of the rhs is the same as the lhs nonterminal.
- For example:

A → Ay | z

Left recursion because the lhs symbol A is the first symbol on the rhs

- Productions

A → Ay | z

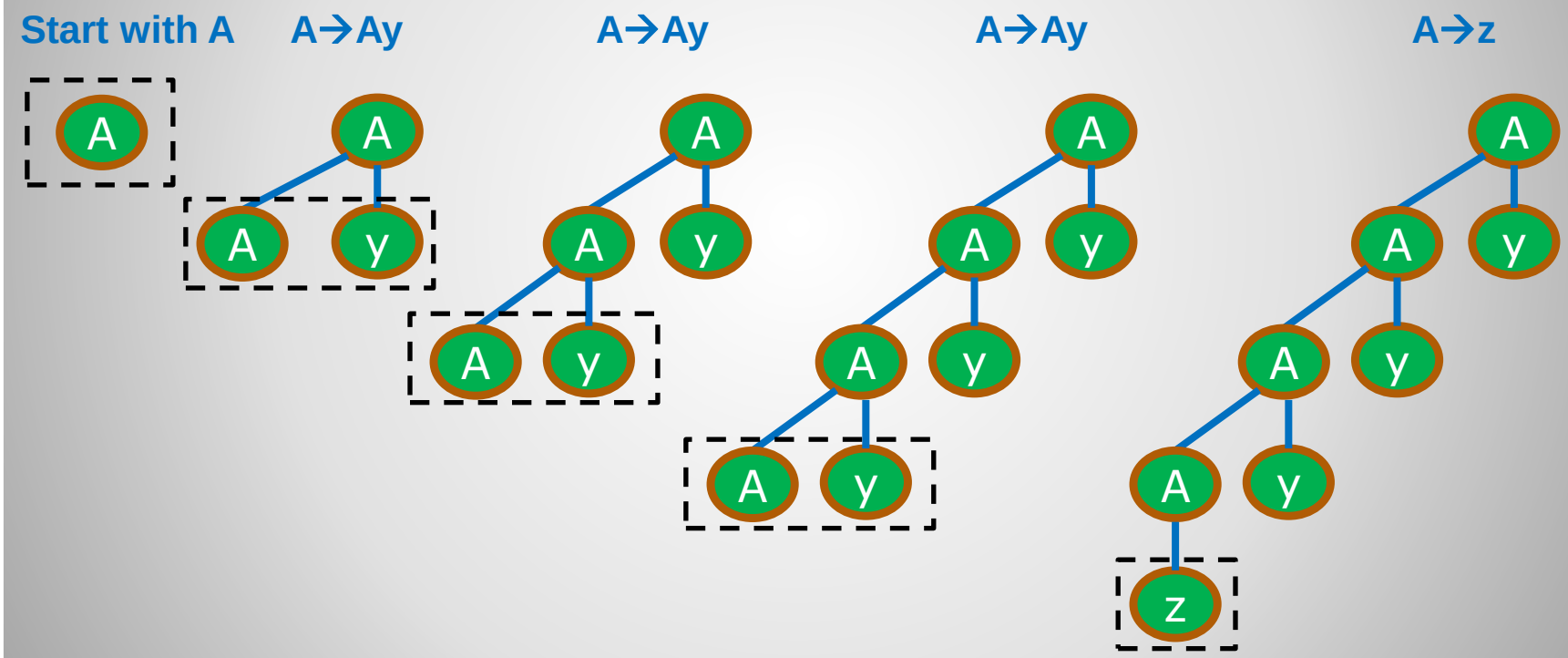
- Parse the following: zyyy

Left Recursive Grammar

- Productions

A → Ay | z

- Parse the following: zyyy



Left Recursive Grammar

Right Recursion

- **Right recursion** – The last symbol of the rhs is the same as the lhs nonterminal.

- For example:

A → zB
B → yB | ε

Right recursion because the lhs symbol B is the last symbol on the rhs

Right Recursive Grammar

- Productions

A → zB

B → yB | ε

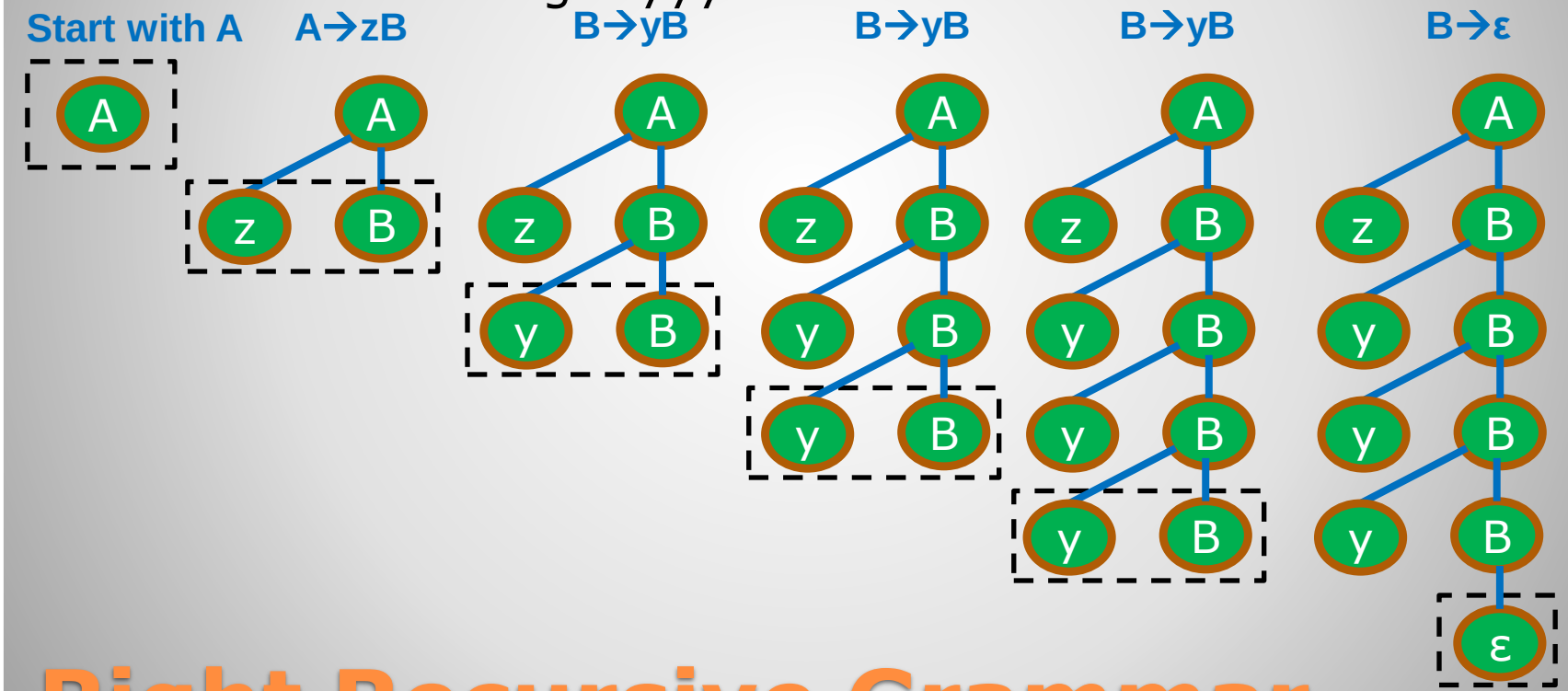
- Parse the following: zyyy

- Productions

A → zB

B → yB | ε

- Parse the following: zyyy



Right Recursive Grammar

- Now on to LL(1) grammars…

LL(1) Grammar

LL(1)

- LL(1) is a top-down parsing technique. We begin with the start symbol and apply substitutions.
- First L in LL(1). The first L means to process the input from left to right.
- Second L in LL(1). The second L stands for a leftmost derivation (this means always expand the leftmost nonterminal).
- The 1 in LL(1). The 1 means there is a 1 symbol lookahead.

- LL(1) grammars are only able to parse a subset of all context-free grammars.

LL(1)

Requirements for Parsing LL(1)

- LL(1) parsing requires that when we are doing a substitution(expanding a nonterminal), the choice of production to use is unambiguous.
- When applying a substitution for a nonterminal, there should only be one possible substitution given one token of lookahead.
- For example, assume we are expanding nonterminal B in the following grammar and the lookahead is 'c'. The choice of which production to use is unambiguous.

A → Bw
B → cs ←
B → dz

The only valid substitution is to use B→c since c is the lookahead. This grammar is valid for LL(1) parsing.

Requirements for Parsing LL(1)

Requirements for Parsing LL(1)

- Assume the following grammar.

A → Bw
B → sC
B → sz
C → y

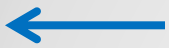
Question
If we are expanding nonterminal B and the lookahead is 's', is this valid when using LL(1) parsing on the grammar above?

Requirements for Parsing LL(1)

Requirements for Parsing LL(1)

- Assume the following grammar.

A → Bw
B → sC
B → sz
C → y



Ambiguous. If we are expanding B and the lookahead is 's', we do NOT know which production to use. It is ambiguous (both production start with 's'). This grammar is not valid for LL(1) parsing.

Question

If we are expanding nonterminal B and the lookahead is 's', is this valid when using LL(1) parsing on the grammar above?

Answer

No

Requirements for Parsing LL(1)

Requirements for Parsing LL(1)

- The first+ sets of productions for a given nonterminal must be disjoint.
- In the following grammar the first+ sets for B's productions are not disjoint (they both include s).

A → Bw
B → sC
B → sz
C → y

First+ sets for B right sides are not disjoint

There is no way to know which substitution to apply since s is in both first+ sets!!!

First+(B→sC) = { s }
First+(B→sD) = { s }

Requirements for Parsing LL(1)

Requirements for Parsing LL(1)

- Assume the following grammar.

A → Bw
B → Cs
B → sz
C → y
C → ε

Question
Are the First+ sets of the productions for B disjoint?

Requirements for Parsing LL(1)

Requirements for Parsing LL(1)

- Assume the following grammar.

A → Bw
B → Cs ←
B → sz
C → y
C → ε

C can disappear in this production because of C→ε.
Since C can disappear s is in the First+ of B→Cs.
Note:First(C)={y, ε}

First+(B→Cs) = {y, ε, s}
First+(B→sz) = { s }

Question
Are the First+ sets of the productions for B disjoint?

Answer
No. The First+ sets of B→Cs and B→ sD intersect (they are not disjoint). This grammar is NOT LL(1).

Requirements for Parsing LL(1)

Requirements for Parsing LL(1)

- Assume the following grammar.

A → Bs
B → C
B → sz
C → y
C → ε

Question
Are the First+ sets of the productions for B disjoint?

Requirements for Parsing LL(1)

Requirements for Parsing LL(1)

- Assume the following grammar.

A → Bs
B → C
B → sz
C → y
C → ε

C can disappear in this production because of C→ε. Since the right side of this production can disappear, we must include members of Follow(B) in First+(B→C).
Note:First(C)={y, ε} and Follow(B)={s}.

First+(B→C) = First(C) + Follow(B) = {y, ε, s}
First+(B→sz) = {s}

Question
Are the First+ sets of the productions for B disjoint?

Answer
No. The First+ sets of B→C and B→ sz intersect (they are not disjoint). This grammar is NOT LL(1).

Requirements for Parsing LL(1)

Requirements for Parsing LL(1)

- Assume the following grammar.

A → Bw
B → C
B → sz
C → y
C → ε

Question
Are the First+ sets of the productions for B disjoint?

Requirements for Parsing LL(1)

Requirements for Parsing LL(1)

- Assume the following grammar.

A → Bw
B → C
B → sz
C → y
C → ε

Follow(B) is in First+(B→C) since C→ε.
Note: Follow(B)={w}.
First+(B→C) = First(C) + Follow(B) = {y, ε ,w}
First+(B→sz) = {s}

The C productions are also valid:
First+(C→y) = {y}
First+(C→ε) = {ε} + Follow(C) = {ε} + Follow(B) = {w, ε}

Question

Are the First+ sets of the productions for B disjoint?

Answer

Yes. The First+ sets of B→C and B→ sz are disjoint. This grammar is LL(1) because all productions for each nonterminal have disjoint First+ sets.

Requirements for Parsing LL(1)

Transforming Grammars to LL(1)

- You can try and transform a grammar to LL(1) using the following techniques:
  - Change to right recursive.
  - Use left factoring
- Every context free grammar is not LL(1), so these techniques are not guaranteed to work.

Transforming Grammars to LL(1)

Predictive Parser

- A predictive parser is able to determine the correct production to use given a small number of lookahead symbols.
- The right recursive form of a grammar might work for LL(1).
- Converting to right recursive form will allow you to use a predictive parser.
- If a grammar is LL(1) then you can build a recursive descent parser for it. Recursive descent parsing means parsing from the start symbol down (top-down parsing).

Predictive Parser

Convert the Expression Grammar to Right Recursive

- Original Expression Grammar (without the assignment)

Exp → Exp + Term | Exp – Term | Term
Term → Term * Fact | Term / Fact | Fact
Fact → id | num | (Exp)

- Right Recursive Expression Grammar

Exp → Term ExpEnd
ExpEnd → + Term ExpEnd | - Term ExpEnd | ε
Term → Fact TermEnd
TermEnd → * Fact TermEnd | / Fact TermEnd | ε
Fact → id | num | (Exp)

Right Recursive Expression Grammar

- Now on to right recursive grammars…

Right Recursive Grammar

Right Recursive Grammar

- Productions (Exp is start symbol)

Exp → Term ExpEnd
ExpEnd → + Term ExpEnd | - Term ExpEnd | ε
Term → Fact TermEnd
TermEnd → * Fact TermEnd | / Fact TermEnd | ε
Fact → id | num | (Exp)

- Parse the following: a+2*b

Right Recursive Grammar

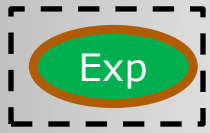
- Productions (Exp is start symbol)

Exp → Term ExpEnd
ExpEnd → + Term ExpEnd | - Term ExpEnd | ε
Term → Fact TermEnd
TermEnd → * Fact TermEnd | / Fact TermEnd | ε
Fact → id | num | (Exp)

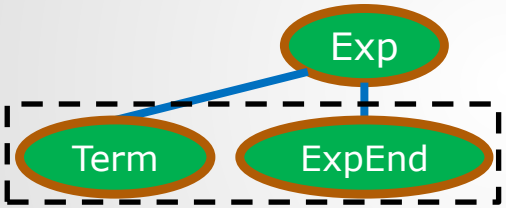
- Parse the following: **a+2*b**

Parse Example Part 1 (continued on next slide)

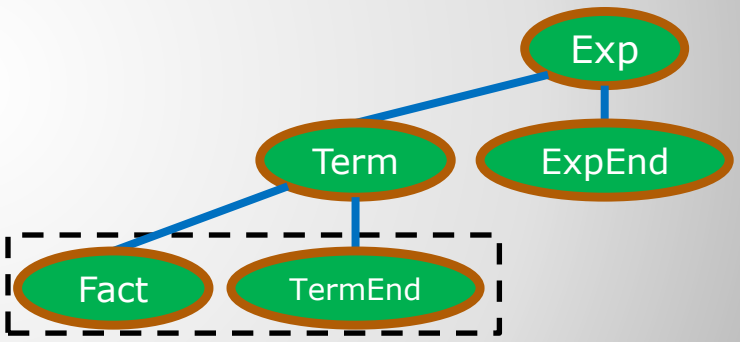
Start with Exp



Exp→Term ExpEnd



Term→Fact TermEnd



Right Recursive Grammar

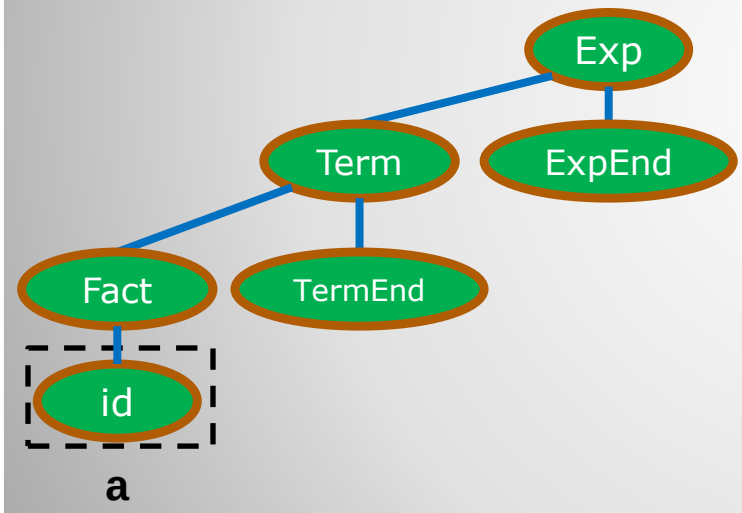
- Productions (Exp is start symbol)

Exp → Term ExpEnd
ExpEnd → + Term ExpEnd | - Term ExpEnd | ε
Term → Fact TermEnd
TermEnd → * Fact TermEnd | / Fact TermEnd | ε
Fact → id | num | (Exp)

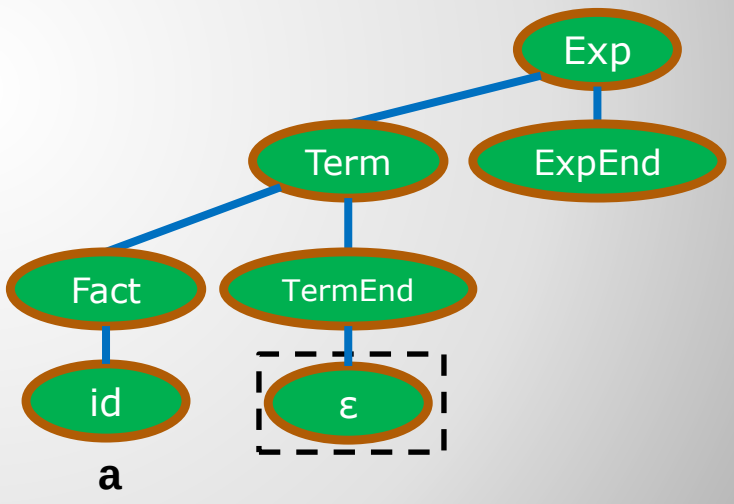
- Parse the following: **a+2*b**

Parse Example Part 2 (continued on next slide)

Fact→id



TermEnd→ ε



Right Recursive Grammar

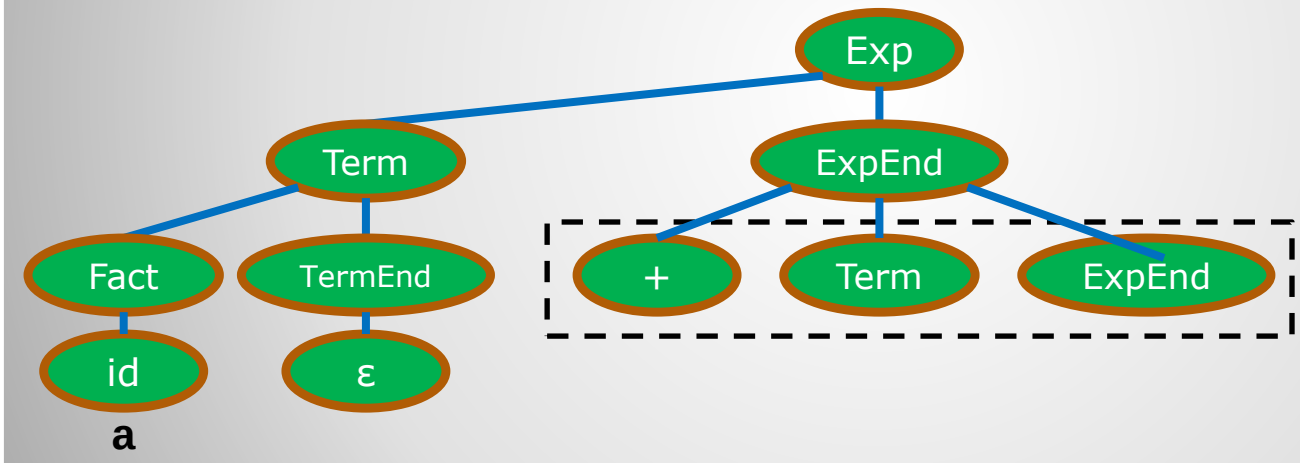
- Productions (Exp is start symbol)

Exp → Term ExpEnd
ExpEnd → + Term ExpEnd | - Term ExpEnd | ε
Term → Fact TermEnd
TermEnd → * Fact TermEnd | / Fact TermEnd | ε
Fact → id | num | (Exp)

- Parse the following: **a+2*b**

Parse Example Part 3 (continued on next slide)

ExpEnd→ + Term ExpEnd



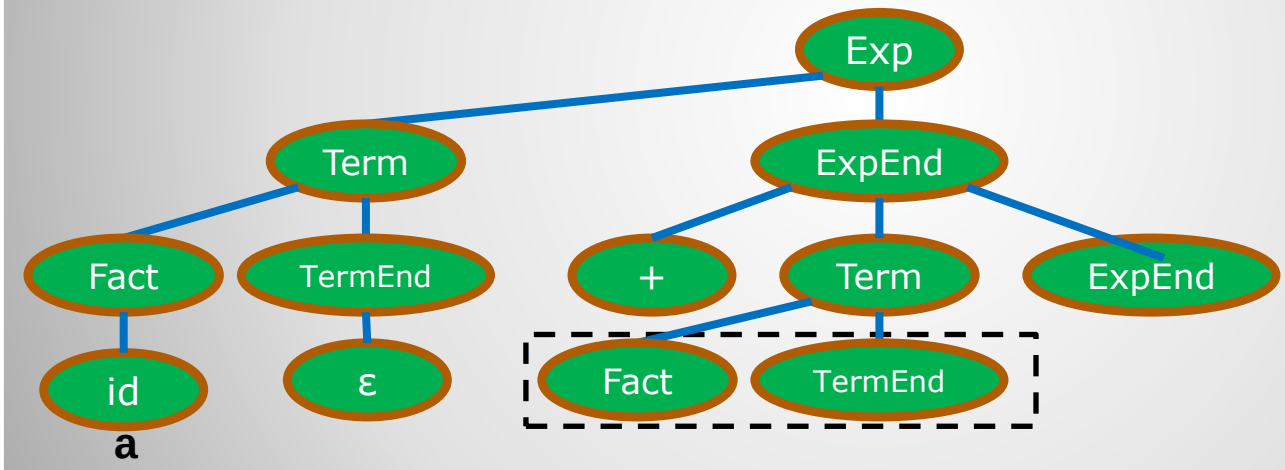
- Productions (Exp is start symbol)

Exp → Term ExpEnd
ExpEnd → + Term ExpEnd | - Term ExpEnd | ε
Term → Fact TermEnd
TermEnd → * Fact TermEnd | / Fact TermEnd | ε
Fact → id | num | (Exp)

- Parse the following: **a+2*b**

Parse Example Part 4 (continued on next slide)

Term→ Fact TermEnd



Right Recursive Grammar

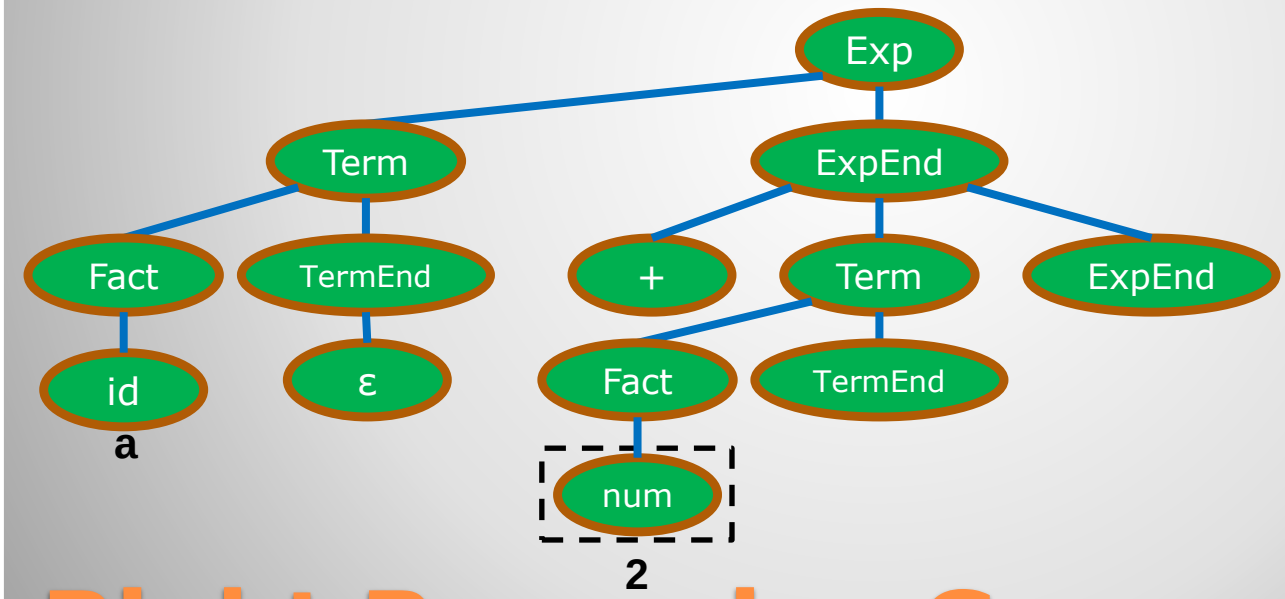
- Productions (Exp is start symbol)

Exp → Term ExpEnd
ExpEnd → + Term ExpEnd | - Term ExpEnd | ε
Term → Fact TermEnd
TermEnd → * Fact TermEnd | / Fact TermEnd | ε
Fact → id | num | (Exp)

- Parse the following: **a+2*b**

Parse Example Part 5 (continued on next slide)

Fact→ num



Right Recursive Grammar

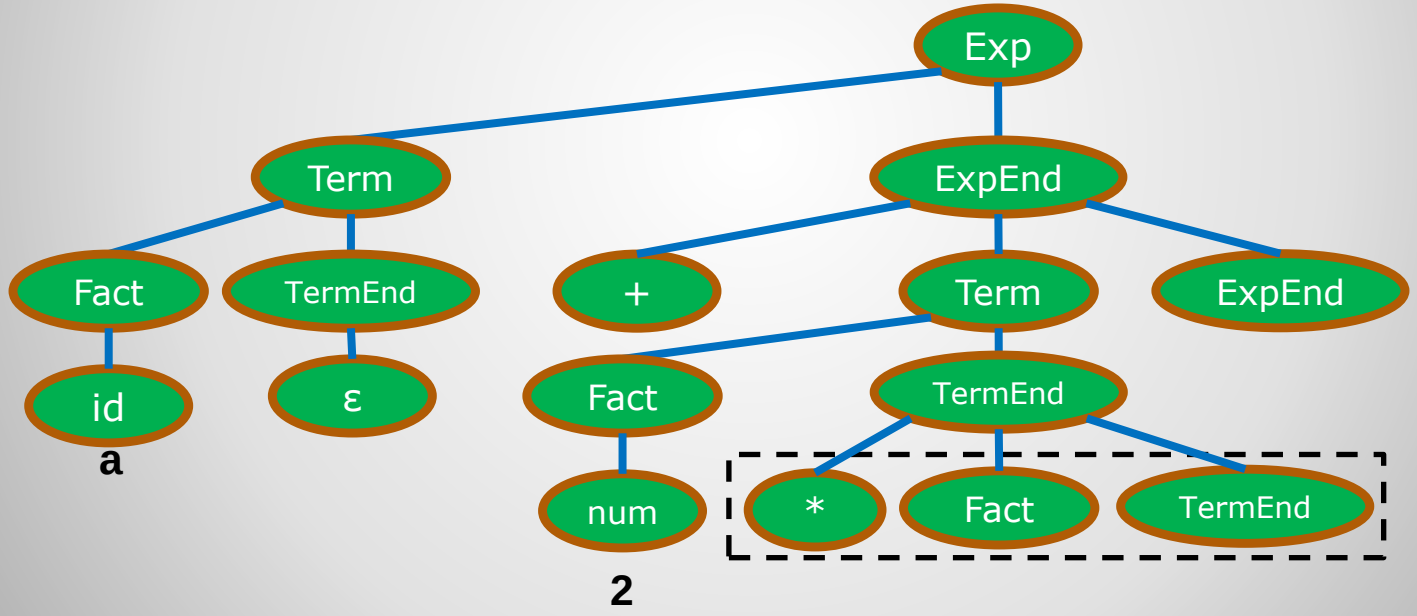
- Productions (Exp is start symbol)

Exp → Term ExpEnd
ExpEnd → + Term ExpEnd | - Term ExpEnd | ε
Term → Fact TermEnd
TermEnd → * Fact TermEnd | / Fact TermEnd | ε
Fact → id | num | (Exp)

- Parse the following: **a+2*b**

Parse Example Part 6 (continued on next slide)

TermEnd→ * Fact TermEnd



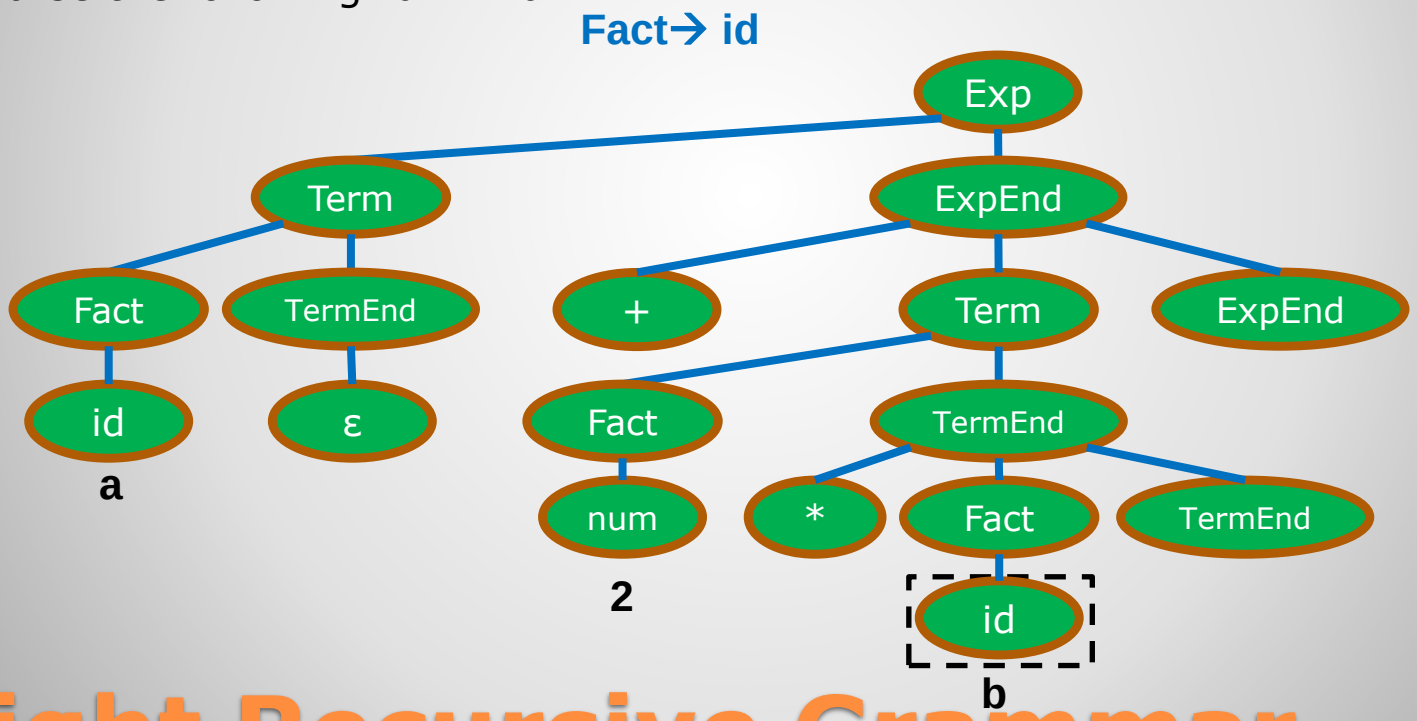
Right Recursive Grammar

- Productions (Exp is start symbol)

Exp → Term ExpEnd
ExpEnd → + Term ExpEnd | - Term ExpEnd | ε
Term → Fact TermEnd
TermEnd → * Fact TermEnd | / Fact TermEnd | ε
Fact → id | num | (Exp)

- Parse the following: **a+2*b**

Parse Example Part 7 (continued on next slide)



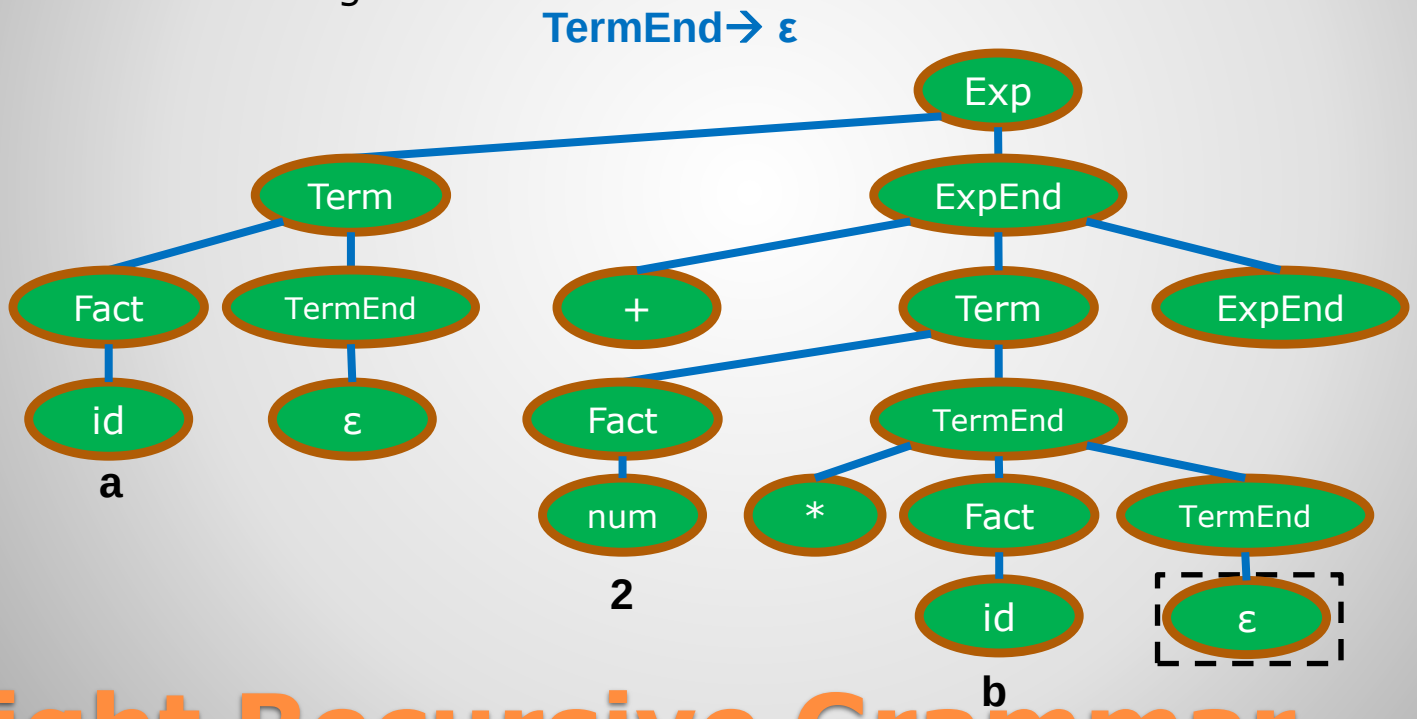
Right Recursive Grammar

- Productions (Exp is start symbol)

Exp → Term ExpEnd
ExpEnd → + Term ExpEnd | - Term ExpEnd | ε
Term → Fact TermEnd
TermEnd → * Fact TermEnd | / Fact TermEnd | ε
Fact → id | num | (Exp)

- Parse the following: **a+2*b**

Parse Example Part 8 (continued on next slide)



Right Recursive Grammar

- Productions (Exp is start symbol)

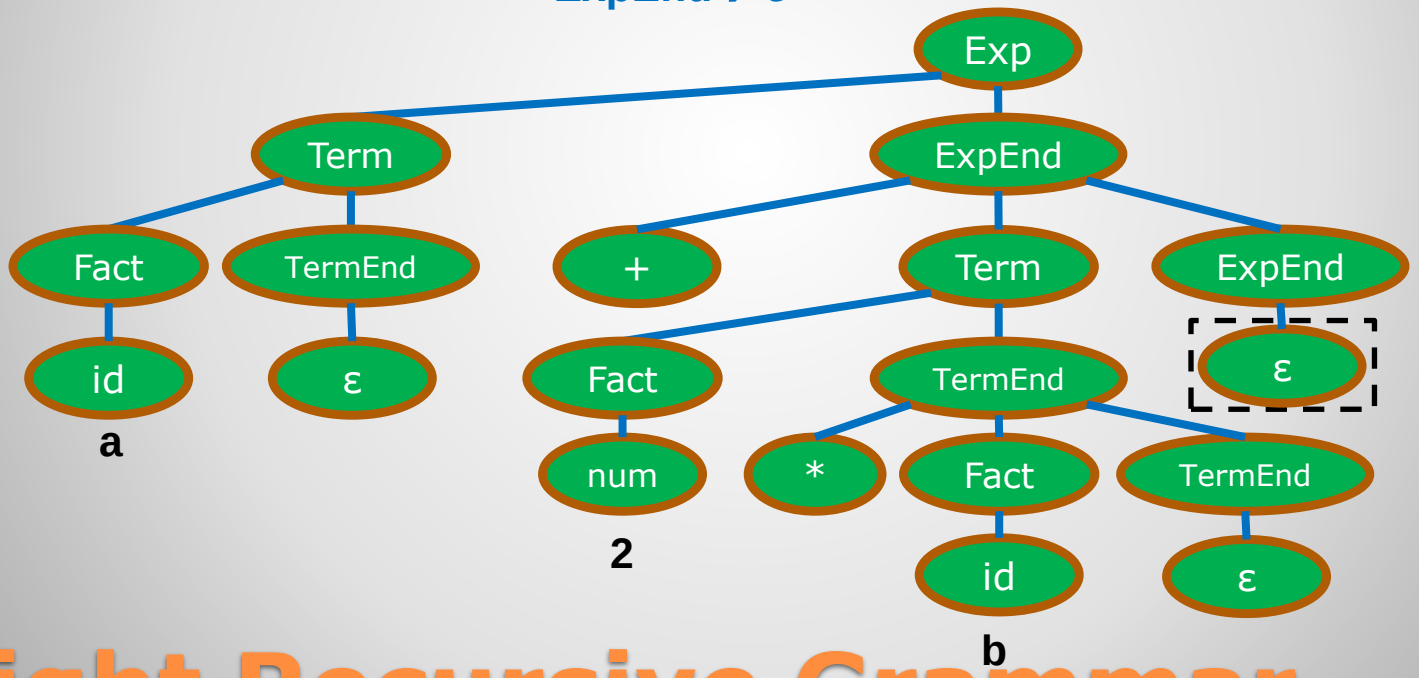
Exp → Term ExpEnd
ExpEnd → + Term ExpEnd | - Term ExpEnd | ε
Term → Fact TermEnd
TermEnd → * Fact TermEnd | / Fact TermEnd | ε
Fact → id | num | (Exp)

- Parse the following: **a+2*b**

Parse Example Part 9

FINISHED!

ExpEnd→ ε



Right Recursive Grammar

- Now on to left factoring…

Left Factoring

Requirement for Parsing LL(1) - Revisited

- Assume the following grammar.

A → cd
A → ce
A → cf

Question
Are the First+ sets of the productions for nonterminal A disjoint?

Requirements for Parsing LL(1) - Revisited

Requirement for Parsing LL(1) - Revisited

- Assume the following grammar.

A → cd
A → ce
A → cf

First+ sets are NOT disjoint (grammar is not LL(1))

First+(A→ cd) = {c}
First+(A→ce) = {c}
First+(A→cf) = {c}

Question
Are the First+ sets of the productions for nonterminal A disjoint?

Answer
No. They all contain c. This grammar is not LL(1).

We can transform these productions so that they are disjoint though!

Requirements for Parsing LL(1) - Revisited

Left Factoring Rule

Greek symbols:
α is alpha
β is beta
γ is gamma

- Assume the following grammar.

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$

- To left factor A move the symbols following α into new productions.

- Grammar after left factoring.

$A \rightarrow \mathbf{\alpha B} \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$

$B \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

α originally appeared in multiple A productions but it has now been factored out into one A production. The γ productions in A remain unchanged because they do not contain α.

- Taken from Engineering a Compiler 2nd edition by Cooper and Torczon, 2012.

Left Factoring Rule

Left Factoring

- Assume the following grammar.

A → cd

A → ce

A → cf

Question

What does the grammar look like after left factoring nonterminal A's productions?

Left Factoring

Left Factoring

- Assume the following grammar.

A → cd
A → ce
A → cf

Leave c in production A and move d, e, and f into the new B productions

Question

What does the grammar look like after left factoring nonterminal A's productions?

Answer

A → cB
B → d
B → e
B → f

The First+ sets are now disjoint

Left Factoring

Left Factoring

- Assume the following grammar.

A → cd
A → ce
A → x
A → y

Question
What does the grammar look like after left factoring nonterminal A's productions?

Left Factoring

Left Factoring

- Assume the following grammar.

A → cd
A → ce
A → x
A → y

Leave c in production A and move d, e, and f into the new B productions

Question

What does the grammar look like after left factoring nonterminal A's productions?

When applying the left factoring rule:

c is α
d is β_1
e is β_2
x is γ_1
y is γ_2

Answer

A → cB
A → x
A → y
B → d
B → e

Left Factoring

- **End of Slides**

End of Slides