# CSCI312 Principles of Programming Languages

Chapter 3

Lexical and Syntactic Analysis

**Xu Liu**

# Recap

# 3.1 Chomsky Hierarchy

Regular grammar -- least powerful

Context-free grammar (BNF)

Context-sensitive grammar

Unrestricted grammar

# Lexical Analysis

Purpose: transform program representation

Input: printable ASCII characters

Output: tokens

Discard: whitespace, comments

Defn: A token is a logically cohesive sequence of characters representing a single symbol.

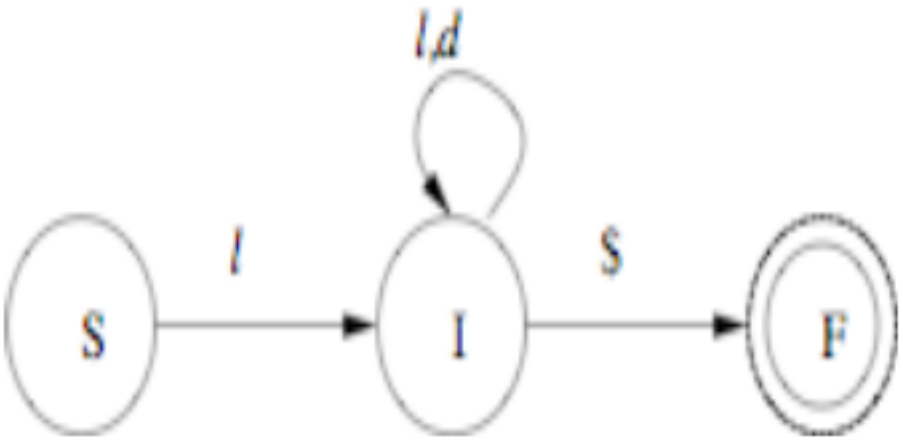
# Example

(S, a2i$) ├ (I, 2i$)

├ (I, i$)

├ (I, $)

├ (F, )




Thus: (S, a2i$) ├* (F, )

# Contents

# Syntactic Analysis

Phase also known as: parser

Purpose is to recognize source structure

Input: tokens

Output: parse tree or abstract syntax tree

A recursive descent parser is one in which each nonterminal in the grammar is converted to a function which recognizes input derivable from the nonterminal.

Program Structure consists of:

Expressions: `x + 2 * y`

Assignment Statement: `z = x + 2 * y`

Loop Statements:

```
while (i < n) a[i++] = 0;
```

Function definitions

Declarations: `int i;`

*Assignment* → *Identifier* = *Expression*

*Expression* → *Term* { *AddOp* *Term* }

*AddOp* → + | -

*Term* → *Factor* { *MulOp Factor* }

*MulOp* → * | /

*Factor* → [ *UnaryOp* ] *Primary*

*UnaryOp* → - | !

*Primary* → *Identifier* | *Literal* | ( *Expression* )

# LL & LR Parser

Two main types of parsers.

LL: left-to-right, leftmost-derivation

LR: left-to-right, rightmost-derivation

left-to-right: consume the input from the left to right.

# LL & LR Grammars

The grammars that can be parsed by LL (LR) parsers.

LL < LR: Some LR grammars cannot be parsed by LL parsers.

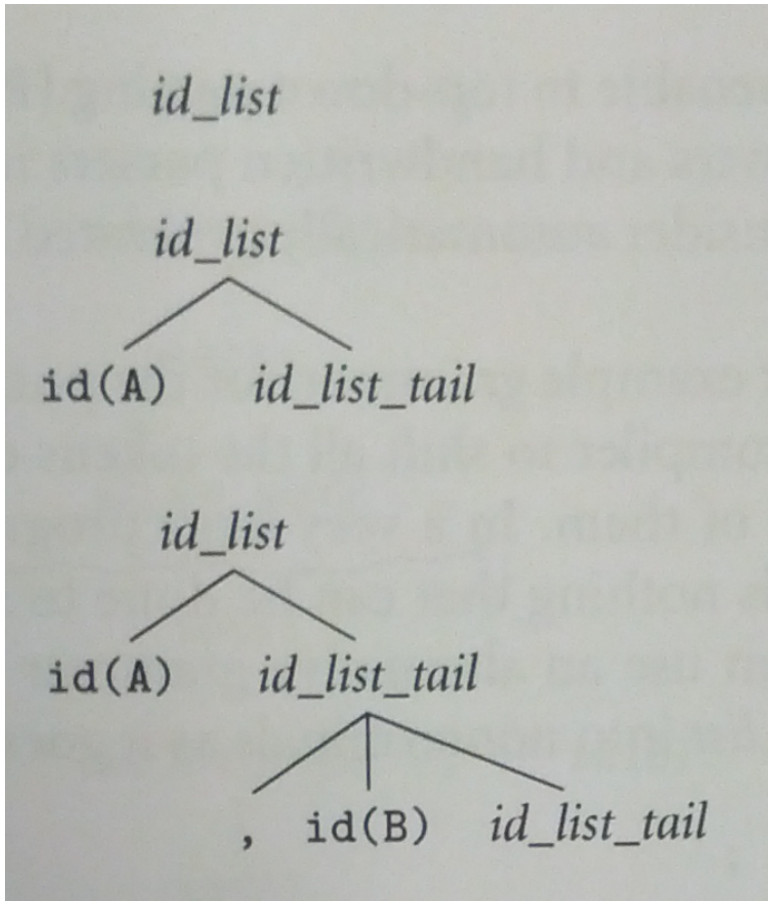
ε: empty.

# Example

IDList -> id  IDList_Tail
IDList_Tail -> ,  id  IDList_Tail
IDList_Tail -> ε

"A, B, C"

**LL parsing stages**



ε: empty.

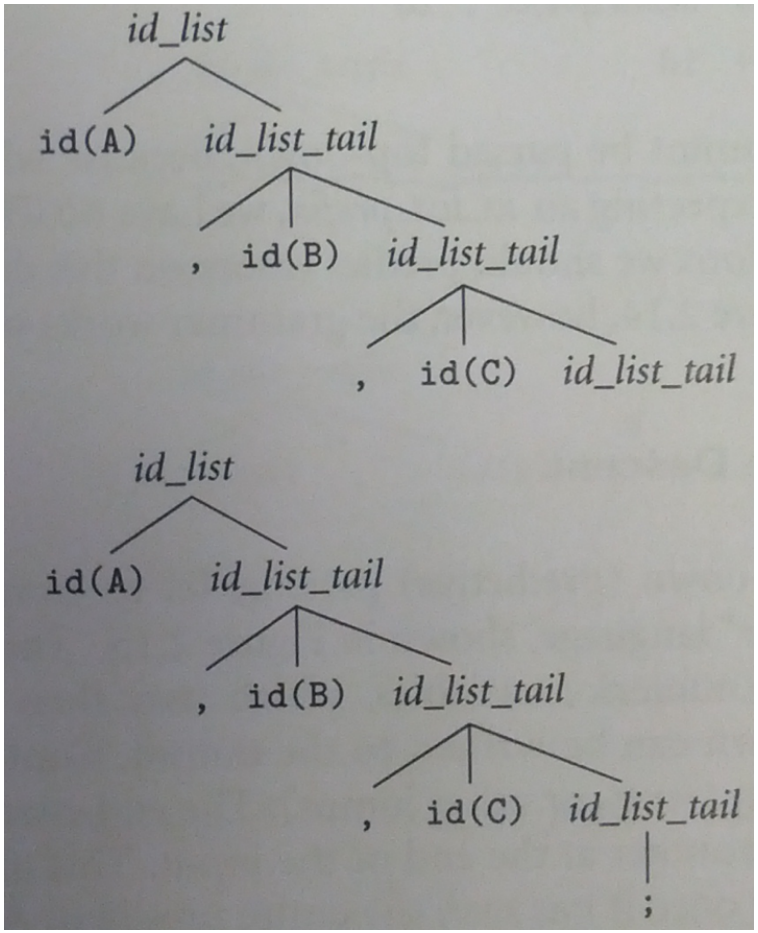
# Example

IDList -> id  IDList_Tail
IDList_Tail -> ,  id  IDList_Tail
IDList_Tail -> ε

"A, B, C"

**LL parsing stages**

(cont. from last slide)



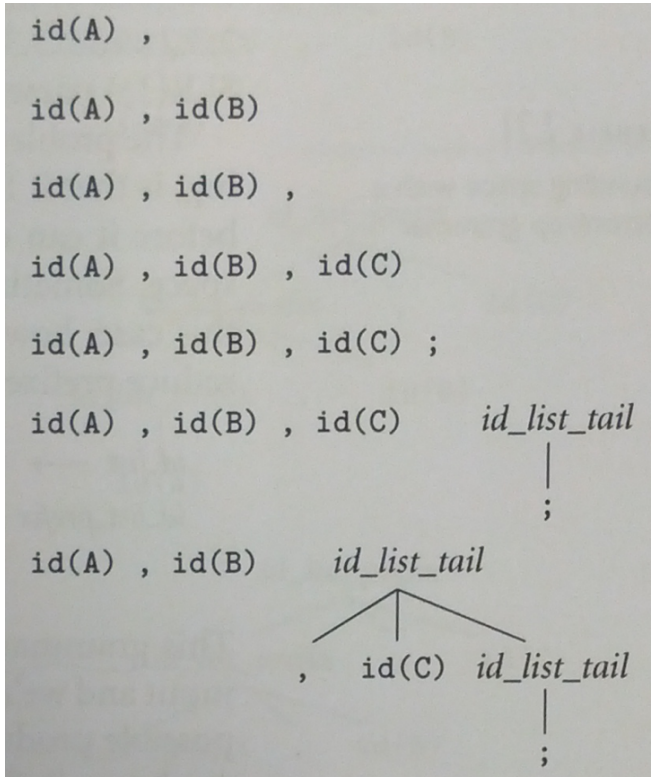
ε: empty.

# Example

IDList -> id  IDList_Tail
IDList_Tail -> ,  id  IDList_Tail
IDList_Tail -> ε

"A, B, C"

**LR parsing stages**



ε: empty.

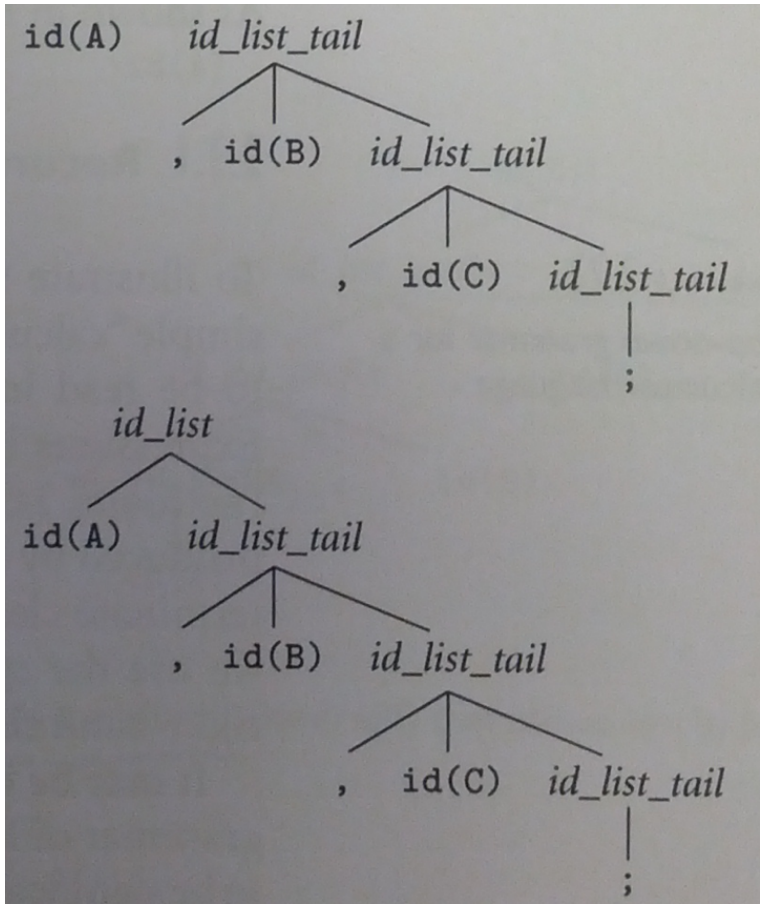
# Example

IDList -> id  IDList_Tail
IDList_Tail -> ,  id  IDList_Tail
IDList_Tail -> ε

"A, B, C"

**LR parsing stages**

(cont. from last slide)



# First Set

Augmented form: S – start symbol

$S' \rightarrow S \ \$$

First(X) is set of leftmost terminal symbols derivable from X

$First(X) = \{ a \in T \mid X \rightarrow^* a\, w,\ w \in (N \cup T)^* \}$

Since there are no production rules for terminal symbols:

$First(a) = a,\ a \in T$

For w = X1 ... Xn V ...

*First(w) = First(X1) ∪ ... ∪ First(Xn) ∪ First(V)*

*where X1, ..., Xn are nullable*

*and V is not nullable*

A is *nullable* if it derives the empty string.

# Nullable Algorithm

```
Set nullable = new Set( );
do { oldSize = nullable.size( );
    for (Production p : grammar.productions( ))
            if (p.length( ) == 0)
                nullable.add(p.nonterminal( ));
            else
                << check righthand side >>
} while (nullable.size( ) > oldSize);
```

# Check Righthand Side

```
{
    boolean  allNull = true;
    for (Symbol t : p.rule( ))
            if (! nullable.contains(t))
                allNull = false;
    if (allNull)
            nullable.add(p.nonterminal( ));
}
```

# Recursive Descent Parser (cont.)

Outline:

- Grammar rewriting for convenience of parser development
- Apply FirstSet
- Parser development with an example
  - Basic coding
  - Output abstract syntax tree

# Rewrite Grammar

Augmented form

Abbreviate symbols

Replace meta constructs with nonterminals

# Original Grammar

*Assignment* → *Identifier* = *Expression*

*Expression* → *Term* { *AddOp* *Term* }

*AddOp* → + | -

*Term* → *Factor* { *MulOp Factor* }

*MulOp* → * | /

*Factor* → [ *UnaryOp* ] *Primary*

*UnaryOp* → - | !

*Primary* → *Identifier* | *Literal* | ( *Expression* )

# Transformed Grammar

S → A $

A → i = E ;

E → T E'

E' → | AO T E'

AO → + | -

T → F T'

T' → |MO F T'

MO → * | /

F → F' P

F' → | UO

UO → - | !

P → *i* | *l* | ( E )

*Assignment* → *Identifier* = *Expression*
*Expression* → *Term* { *AddOp* *Term* }
*AddOp* → + | -
*Term* → *Factor* { *MulOp* *Factor* }
*MulOp* → * | /
*Factor* → [ *UnaryOp* ] *Primary*
*UnaryOp* → - | !
*Primary* → *Identifier* | *Literal* | ( *Expression* )

# Compute Nullable Symbols

| Pass | Nullable |
|---|---|
| 1 | E' T' F' |
| 2 | E' T' F' |

S → A $
A → i = E ;
E → T E'
E' → | AO T E'
AO → + | -
T → F T'
T' → | MO F T'
MO → * | /
F → F' P
F' → | UO
UO → - | !
P → *i* | *l* | ( E )

# Left Dependency Graph

For each production:

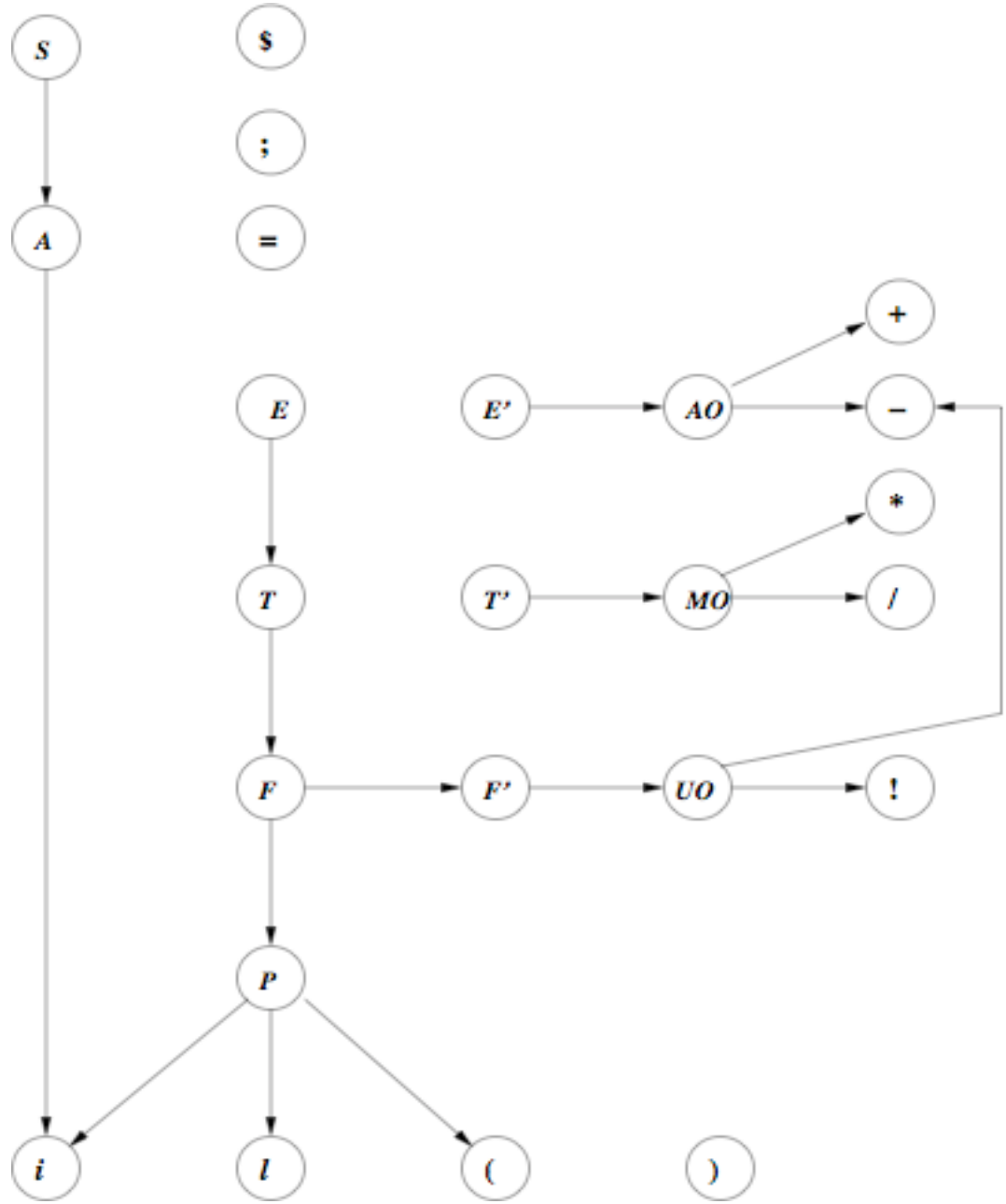
$A \rightarrow V1 \ldots Vn\ X\ w$

$V1, \ldots Vn$ *nullable*

draw an arc from A to X.

Note: you also get arcs from A to V1, ..., A to Vn

S → A $
A → i = E ;
E → T E'
E' → | AO T E'
AO → + | -
T → F T'
T' → MO F T'
MO → * | /
F → F' P
F' → | UO
UO → - | !
P → *i* | *l* | ( E )



| Nonterminal | First | Nonterminal | First |
|---|---|---|---|
| A | i | UO | ! - |
| E | ! - i l ( | P | i l ( |
| E’ | + - | | |
| AO | + - | | |
| T | ! - i l ( | | |
| T’ | * / | | |
| MO | * / | | |
| F | ! - i l ( | | |
| F’ | ! - | | |

# Features of Recursive Descent Parsing

Method/function for each nonterminal

Recognize longest sequence of tokens derivable from the nonterminal

Need an algorithm for converting productions to code

Based on EBNF

# T(EBNF) = Code: A → w

1 If w is nonterminal, call it.

2 If w is terminal, match it against given token.

3 If w is { w' }:

```
while (token in First(w'))  T(w')
```

4 If w is: w1 | ... | wn,

```
switch (token) {
    case First(w1):  T(w1); break;
    ...
    case First(wn):  T(wn); break;
```

5 Switch (cont.): If some wi is empty, use:

default: break;

*Otherwise*

default: error(token);

6 If w = [ w' ], rewrite as ( | w' ) and use rule 4.

7 If w = X1 ... Xn, T(w) =

T(X1); ... T(Xn);

# Augmentation Production

Gets first token

Calls method corresponding to original start symbol

Checks to see if final token is end token

*E.g.: end of file token*

```
private void match (int t) {
    if (token.type() == t)
            token = lexer.next();
    else
            error(t);
}
```

```
private void error(int tok) {
    System.err.println(
            "Syntax error: expecting"
            + tok + "; saw: "  + token);
    System.exit(1);
}
```

```
private void assignment( ) {
   // Assignment → Identifier = Expression ;
   match(Token.Identifier);
   match(Token.Assign);
   expression( );
   match(Token.Semicolon);
}
```

```
private void expression( ) {
   // Expression → Term { AddOp Term }
   term( );
   while (isAddOp()) {
      token = lexer.next( );
      term( );
   }
}
```

# Linking Syntax and Semantics

Output: parse tree is inefficient

- One nonterminal per precedence level

Shape of parse tree that is important

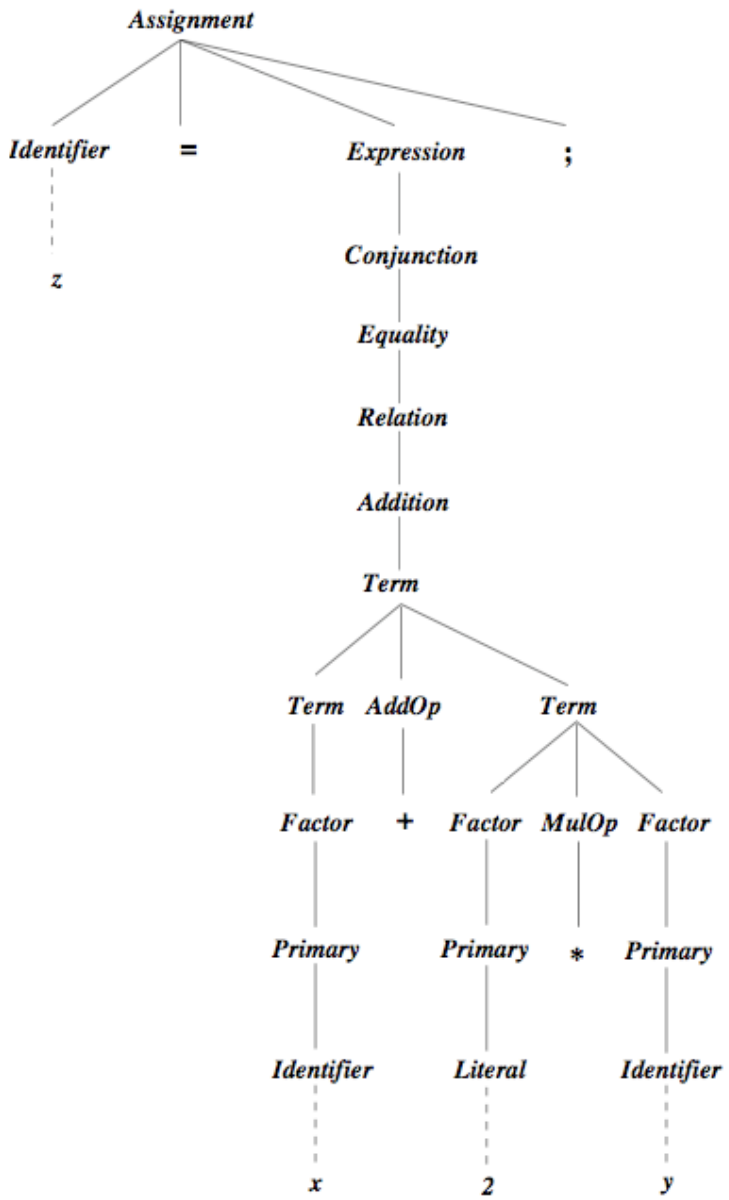
# Discard from Parse Tree

Separator/punctuation terminal symbols

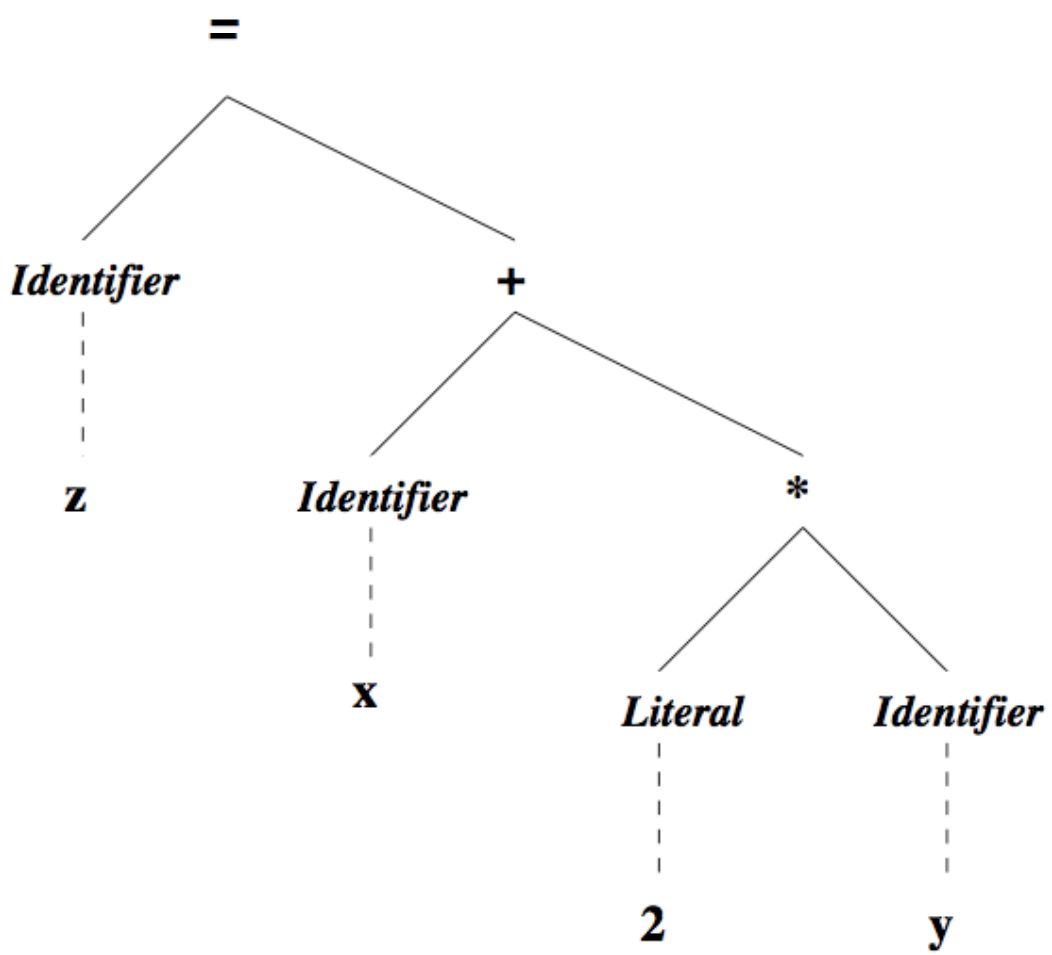
All trivial root nonterminals

Replace remaining nonterminal with leaf terminal

Parse Tree for
z = x + 2*y;
Fig. 2.9



Abstract Syntax Tree for
z = x + 2*y;
Fig. 2.10



# Abstract Syntax Example

*Assignment* = *Variable* target; *Expression* source

*Expression* = *Variable* | *Value* | *Binary* | *Unary*

*Binary* = *Operator* op; *Expression* term1, term2

*Unary* = *Operator* op; *Expression* term

*Variable* = *String* id

*Value* = *Integer* value

*Operator* = + | - | * | / | !

```
abstract class Expression { }

class Binary extends Expression {
   Operator op;
   Expression term1, term2;
}

class Unary extends Expression {
   Operator op; Expression term;
}
```

# Modify T(A → w) to Return AST

1. Make A the return type of function, as defined by abstract syntax.
2. If w is a nonterminal, assign returned value.
3. If w is a non-punctuation terminal, capture its value.
4. Add a return statement that constructs the appropriate object.

```
private Assignment assignment( ) {
   // Assignment → Identifier = Expression ;
   Variable target = match(Token.Identifier);
   match(Token.Assign);
   Expression source = expression( );
   match(Token.Semicolon);
   return new Assignment(target, source);
}
```

```
private String match (int t) {
    String value = Token.value( );
    if (token.type( ) == t)
            token = lexer.next( );
    else
            error(t);
    return value;
}
```

# Abstract Syntax Tree for
# z = x + 2*y;
# Fig. 2.10