UNIVERSITY OF PASSAU

Faculty of Computer Science and Mathematics

# Programming Paradigms: Syntax

Summer Semester 2023

Dr. Abhishek Tiwari, Prof. Dr. Christian Hammer

# Wake up Exercise

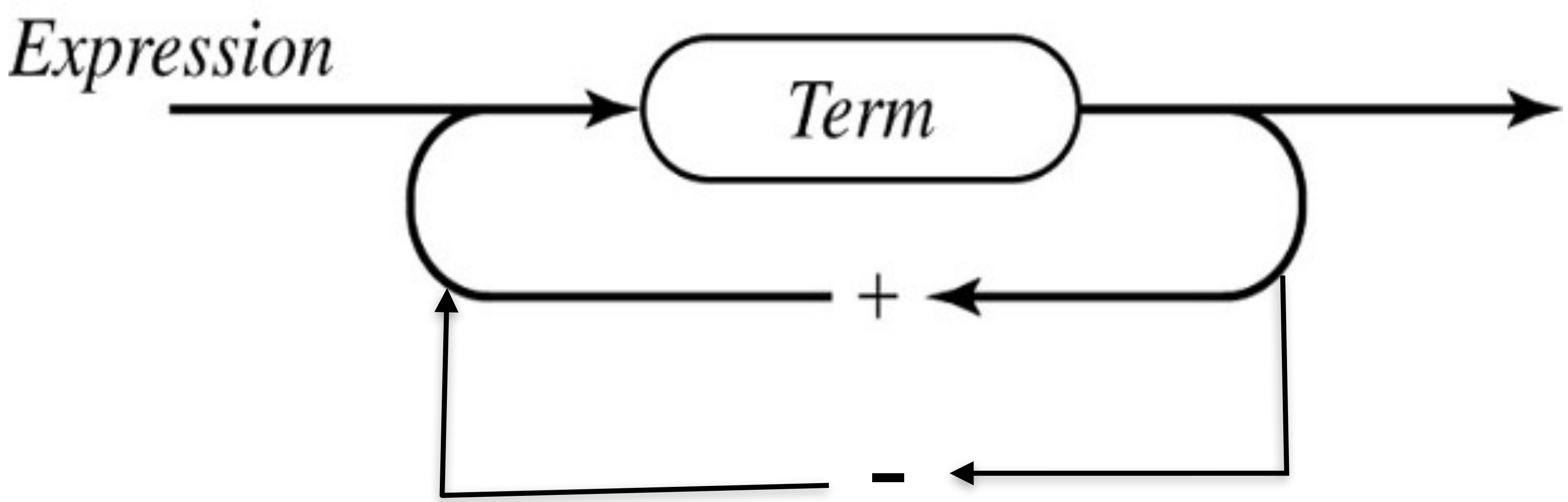
$$
\begin{array}{rcl}
s \in Stmt & ::= & D \mid v = e \mid \textbf{allocate}(p, c) \mid \textbf{free}\ (p) \mid \\
& & \textbf{arraystore}(arr, i, v) \mid \textbf{arrayload}(arr, i, v) \mid \textbf{input}(\mathbf{x}) \mid \\
& & if\ (e)\ \{\overrightarrow{s_1}\}\ else\ \{\overrightarrow{s_2}\} \mid while(e)\ \{\overrightarrow{s}\} \mid v_n = func(\overrightarrow{v}) \mid \\
& & s_1; s_2 \mid skip \\
e \in Exp & ::= & v \mid c \mid v_1 \odot v_2 \mid p \pm c \mid *p \\
D \in Decl & ::= & T\,var \mid T\ arr[n] \mid T\ *p
\end{array}
$$

Draw a parse tree for: **x = x + y ; x = x - y**

- Extend the grammar with metasymbols for iteration, option, and choices
  - {} for a series of zero or more occurrences
  - () for a list of alternatives to be picked
  - [] for an optional sequence; pick one or none

- Consider BNF grammar
  - <IfStatement> ::= <if (Expression)><Statement> | <if (Expression)><Statement> <else> <Statement>
- Corresponding EBNF grammar
  - <IfStatement> ::= <if (Expression)><Statement> [else <Statement>]
- Consider BNF grammar
  - <Expr> ::= <Expr> + <Term> | <Expr> — <Term> | <Term>
  - <Term> ::= 0|…|9
- Corresponding EBNF grammar
  - <Expr> ::= <Term> { (+|-) <Term>}
  - <Term> ::= 0|…|9

# Syntax Diagram



- Neither EBNF nor the syntax diagram is anymore powerful than BNF
- Let A be a nonterminal and x, y, z be arbitrary sequences of terminal and nonterminals.
  - A EBNF Grammar rule: A ::= x {y} z
  - Corresponding BNF rule:
    - A ::= x A' z
    - A' ::= | y A'
    - A' is a unique new nonterminal

# *Grammar Size of Languages*

| Language | Grammar Size (Pages) | Reference |
|---|---|---|
| Pascal | 5 | Jensen & Wirth (1975) |
| C | 6 | Kernighan & Ritchie (1988) |
| C++ | 22 | Stroustrup (1997) |
| Java | 14 | Gosling et al. (1996) |

$$
\begin{aligned}
\mathit{Program} &::= \texttt{int main ( ) \{ } \mathit{Declarations\ Statements}\ \texttt{\}} \\
\mathit{Declarations} &::= \{\ \mathit{Declaration}\ \} \\
\mathit{Declaration} &::= \mathit{Type\ Identifier}\ [\ \texttt{[}\ \mathit{Integer}\ \texttt{]}\ ]\ \{\ ,\ \mathit{Identifier}\ [\ \texttt{[}\ \mathit{Integer}\ \texttt{]}\ ]\ \} \\
\mathit{Type} &::= \texttt{int} \mid \texttt{bool} \mid \texttt{float} \mid \texttt{char} \\
\mathit{Statements} &::= \{\ \mathit{Statement}\ \} \\
\mathit{Statement} &::= \texttt{;} \mid \mathit{Block} \mid \mathit{Assignment} \mid \mathit{IfStatement} \mid \mathit{WhileStatement} \\
\mathit{Block} &::= \texttt{\{}\ \mathit{Statements}\ \texttt{\}} \\
\mathit{Assignment} &::= \mathit{Identifier}\ [\ \texttt{[}\ \mathit{Expression}\ \texttt{]}\ ]\ \texttt{=}\ \mathit{Expression}\ \texttt{;} \\
\mathit{IfStatement} &::= \texttt{if (}\ \mathit{Expression}\ \texttt{)}\ \mathit{Statement}\ [\ \texttt{else}\ \mathit{Statement}\ ] \\
\mathit{WhileStatement} &::= \texttt{while (}\ \mathit{Expression}\ \texttt{)}\ \mathit{Statement}
\end{aligned}
$$

## Clite (Expressions)

```
Expression ::= Conjunction { || Conjunction }
Conjunction ::= Equality { && Equality }
Equality ::= Relation [ EquOp Relation ]
EquOp ::= == | !=
Relation ::= Addition [ RelOp Addition ]
RelOp ::= < | <= | > | >=
Addition ::= Term { AddOp Term }
AddOp ::= + | -
Term ::= Factor { MulOp Factor }
MulOp ::= * | / | %
Factor ::= [ UnaryOp ] Primary
UnaryOp ::= - | !
Primary ::= Identifier [ [ Expression ] ] | Literal | ( Expression ) |
        Type ( Expression )
```

*Identifier* ::= *Letter* { *Letter* | *Digit* }

*Letter* ::= `a` | `b` | … | `z` | `A` | `B` | … | `Z`

*Digit* ::= `0` | `1` | … | `9`

*Literal* ::= *Integer* | *Boolean* | *Float* | *Char*

*Integer* ::= *Digit* { *Digit* }

*Boolean* ::= `true` | `False`

*Float* ::= *Integer* `.` *Integer*

*Char* ::= ' *ASCII Char* '

- Comments
- Whitespaces
- Differentiating one token <= from two token < =
- Differentiating identifiers from keywords like $if$
- These issues are addressed by identifying two levels:
    - lexical level
    - syntactic level

## *Lexical Syntax*

- Input alphabet: a stream of character from the ASCII set, keyed by a programmer
- The derivable terminal strings are called *tokens*, classified as follows:
    - Identifiers: e.g., dummy, x, y
    - Literals: e.g., 123, ‘x’, 3.25, true
    - Keywords: *bool char else false float if int main true while*
    - Operators: = || && == != < <= > >= + - * / !
    - Punctuation: ; , { } ( )

# *Whitespace*

- Any space, tab, end-of-line character (or characters), or character sequence inside a comment
- No token may contain embedded whitespace
    - >= one token
    - > = two tokens

# *Whitespace Examples in Pascal*

- while a < b do — legal — spacing between tokens
- while a<b do — legal — spacing not needed for <
- whilea<bdo — illegal — can’t tell boundaries
- whilea < bdo

# Comments

- Not defined in grammar
- Uses C++ style comments, i.e., //

$$\mathit{Identifier} ::= \mathit{Letter} \ \{ \ \mathit{Letter} \mid \mathit{Digit} \ \}$$

- Sequence of Letters and digits, starting with a letter
  - if is both an identifier and a keyword
  - Most language require the identifiers to be distinct from keywords
  - In some language, these are merely predefined and can be redefined

## *Redefining Identifiers can be Dangerous*

```
program confusing;
const true = false;
begin
  if (a<b) == true then
    f(a)
  else
  …
```

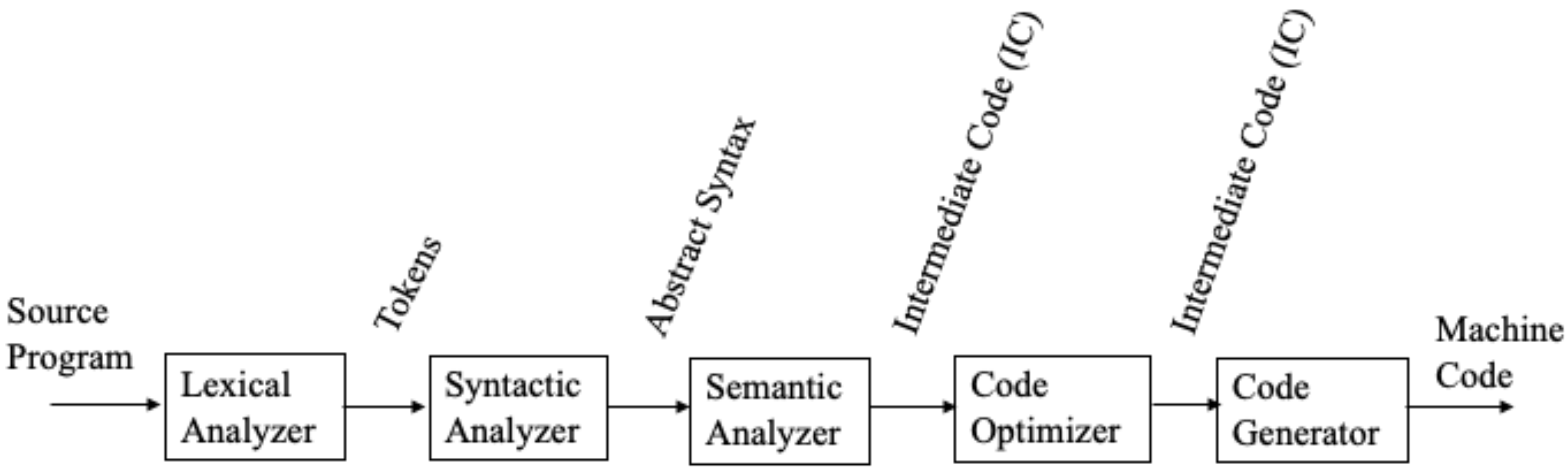
- Based on a parse of its *tokens*
  - ; is a statement terminator
- A declaration consists of a type followed by a list of identifiers separated by **,**
  - e.g., int i, j;
- Rule for IfStatement is ambiguous
  - The else ambiguity is resolved by connecting an else with the last encountered else-less if” [Stroustrup, 1991]

- 13 grammar rules
- Use of metabraces ({ }) — operators are left associative
- C++ requires 4 pages of grammar rules
- C uses an ambiguous expression grammar
- Clite has many fewer operators and resulting precedence levels

| Clite Operator | Associativity |
| --- | --- |
| Unary - ! | none |
| * / | left |
| + - | left |
| < <= > >= | none |
| == != | none |
| && | left |
| \|\| | left |

- Equality and relational operators are non-associative
  - idea borrowed from Ada
- Why is this important?
  - In C++, the expression
    - if (a < b < c) is not equivalent to
    - if (a < b && b < c)

```
#include<stdio.h>

int main(){

int a = 3, b = 4, c = 2;

if(a<b<c)
printf("in If\n");
else
printf("in else\n");

return 0;

}
```


Source Program
Tokens
Abstract Syntax
Intermediate Code (IC)
Intermediate Code (IC)
Machine Code
Lexical Analyzer
Syntactic Analyzer
Semantic Analyzer
Code Optimizer
Code Generator

- A source program is processed as a stream of characters
- Scan the program and transform into a stream of tokens
    - discard all whitespaces, comments.
    - throw errors for all invalid character sequences
- Potential benefits
    - may improve up to 75% time for non-optimizing
    - simpler design
    - handle OS specific end of line conventions

- Based on BNF/EBNF grammar
- Input: tokens from phase 1
- Output: abstract syntax tree (parse tree)
- Abstract syntax: parse tree with punctuations, many nonterminal discarded

- Check that all used identifiers are declared
- Perform type checking
- Insert implied conversion operators (make them explicit)

- Evaluate constant expression at compile-time
- Reorder code to improve cache performance
- Eliminate common subexpression
- Eliminate unnecessary code

- Output: machine code
- Instruction selection
- Register management
- Peephole optimization

- Replaces last 2 phases of the compiler process
- Input:
    - Mixed: intermediate code
    - Pure: stream of ASCII characters
- Mixed interpreter
    - Java, perl, Python, Haskell, Scheme
- Pure interpreter
    - most shell commands

Parse Tree for

**z = x + 2*y**

*Assignment* ::= *Identifier* [ [ *Expression* ] ] = *Expression* **;**

*Expression* ::= *Conjunction* { || *Conjunction* }

*Conjunction* ::= *Equality* { && *Equality* }

*Equality* ::= *Relation* [ *EquOp Relation* ]

*Relation* ::= *Addition* [ *RelOp Addition* ]

*Addition* ::= *Term* { *AddOp Term* }

*AddOp* ::= + | -

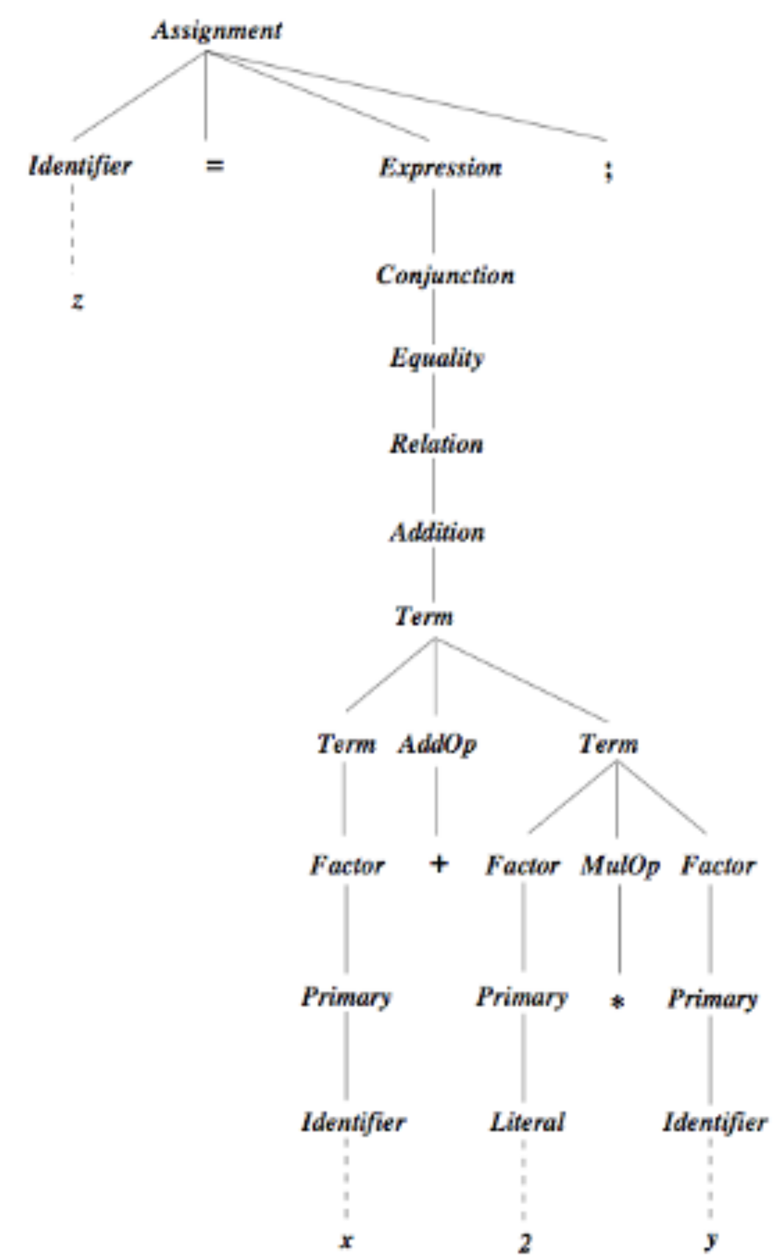
*Term* ::= *Factor* { *MulOp Factor* }

*MulOp* ::= * | / | %

*Factor* ::= [ *UnaryOp* ] *Primary*

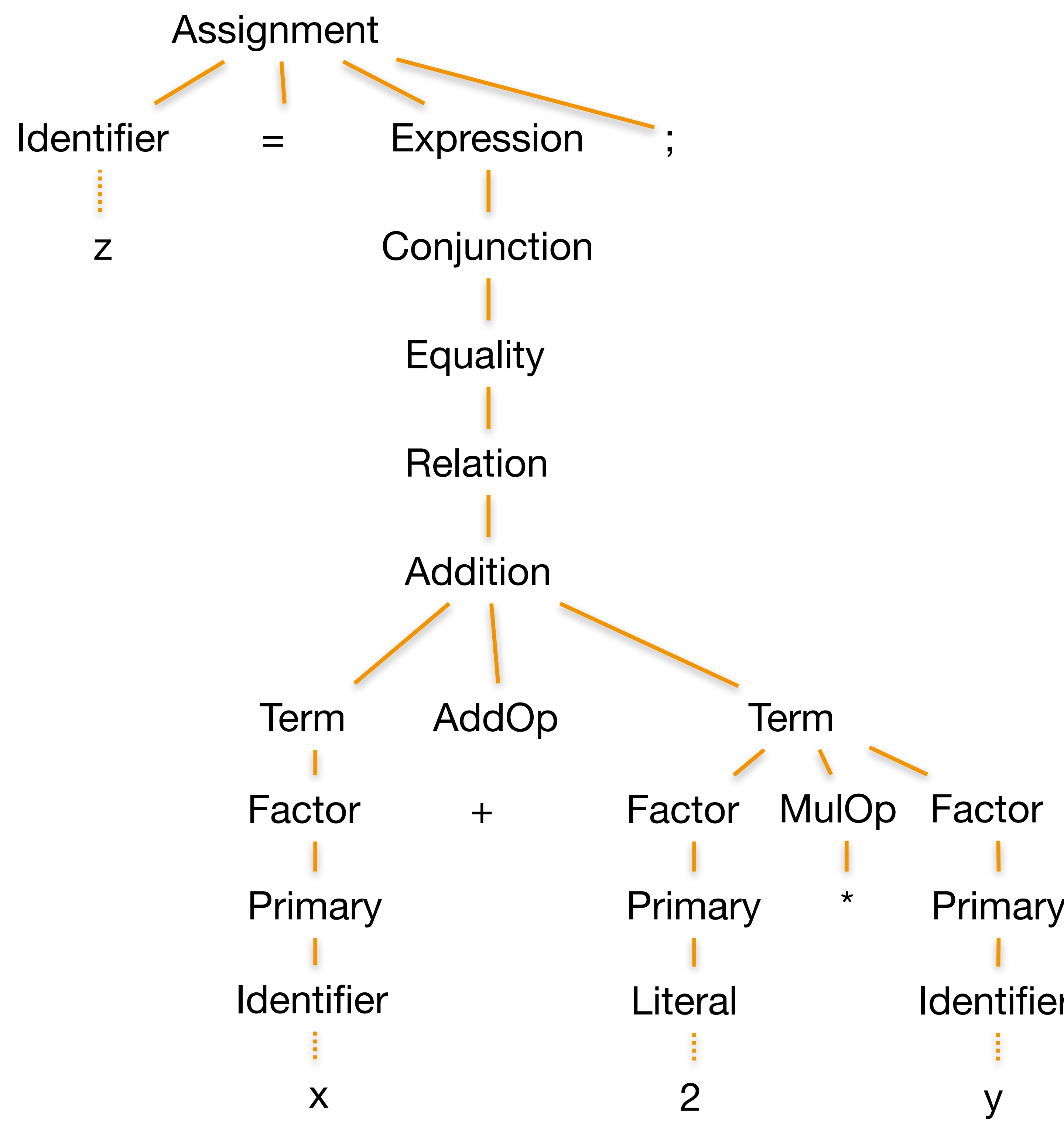
*UnaryOp* ::= - | !

*Primary* ::= *Identifier* [ [ *Expression* ] ] | *Literal* | ( *Expression* ) | *Type* ( *Expression* )



Parse Tree for

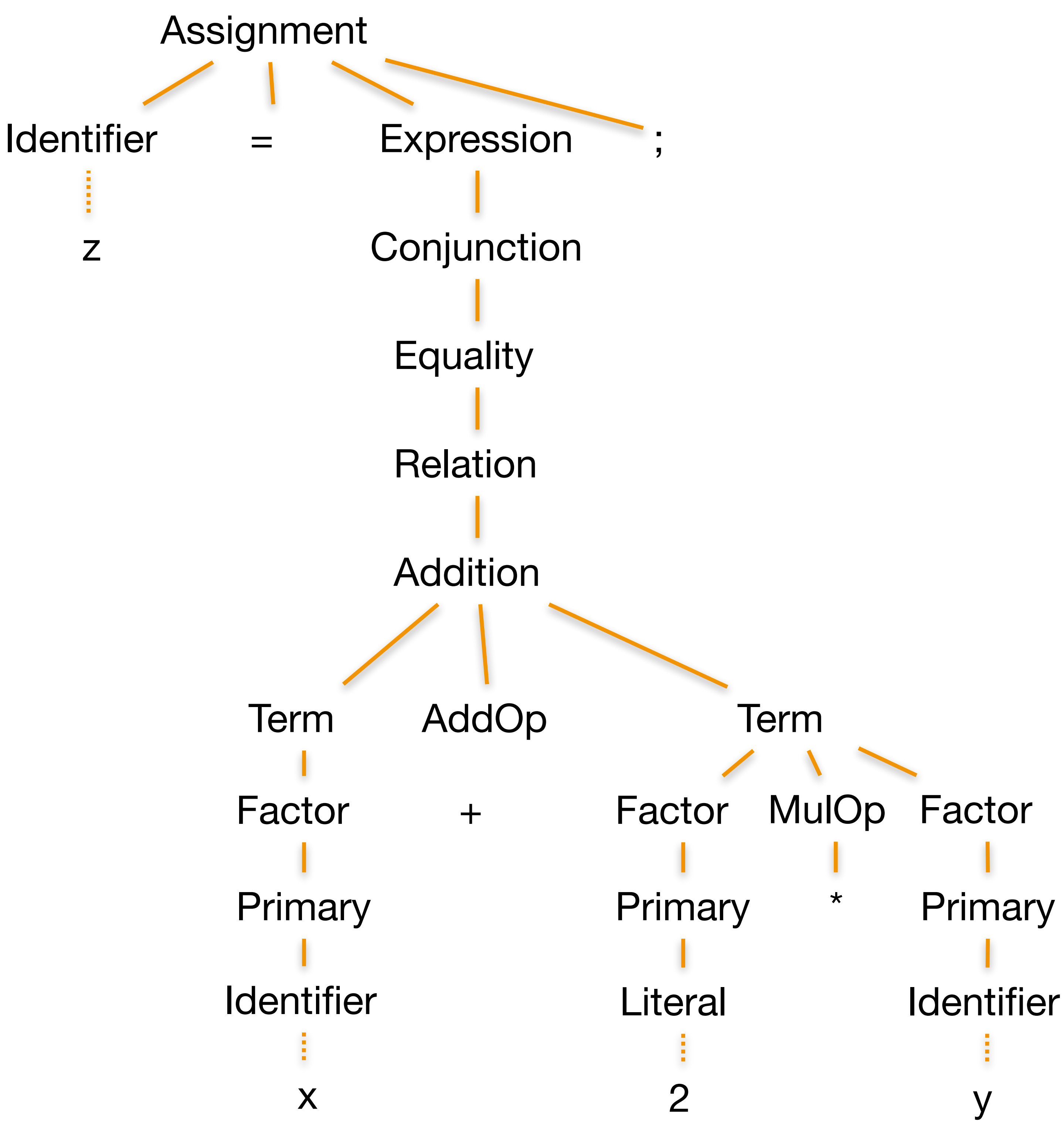
**z = x + 2*y;**

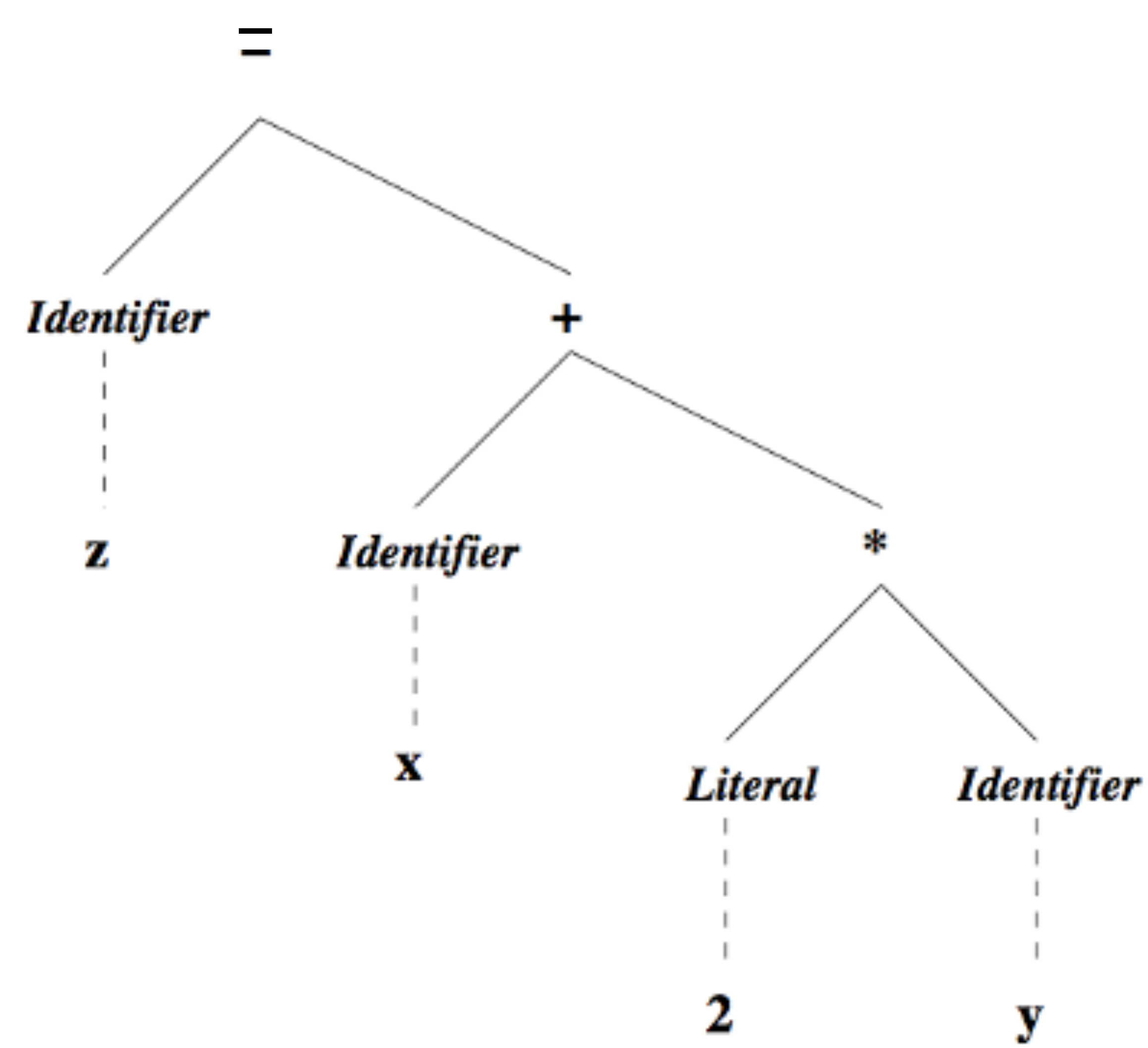


# *Finding a More Efficient Tree*

- The *shape* of the parse tree reveals the meaning of program
- Contains many redundant and inefficient nodes
  - Remove separator/punctuation terminal symbols
  - Remove all trivial root nonterminals
  - Replace remaining nonterminals with leaf terminals

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



- Programming Languages: Principles and Paradigms by Allen B. Tucker and Robert E. Noonan