Faculty of Computer Science and Mathematics

Programming Paradigms: Syntax

Summer Semester 2023

Dr. Abhishek Tiwari, Prof. Dr. Christian Hammer

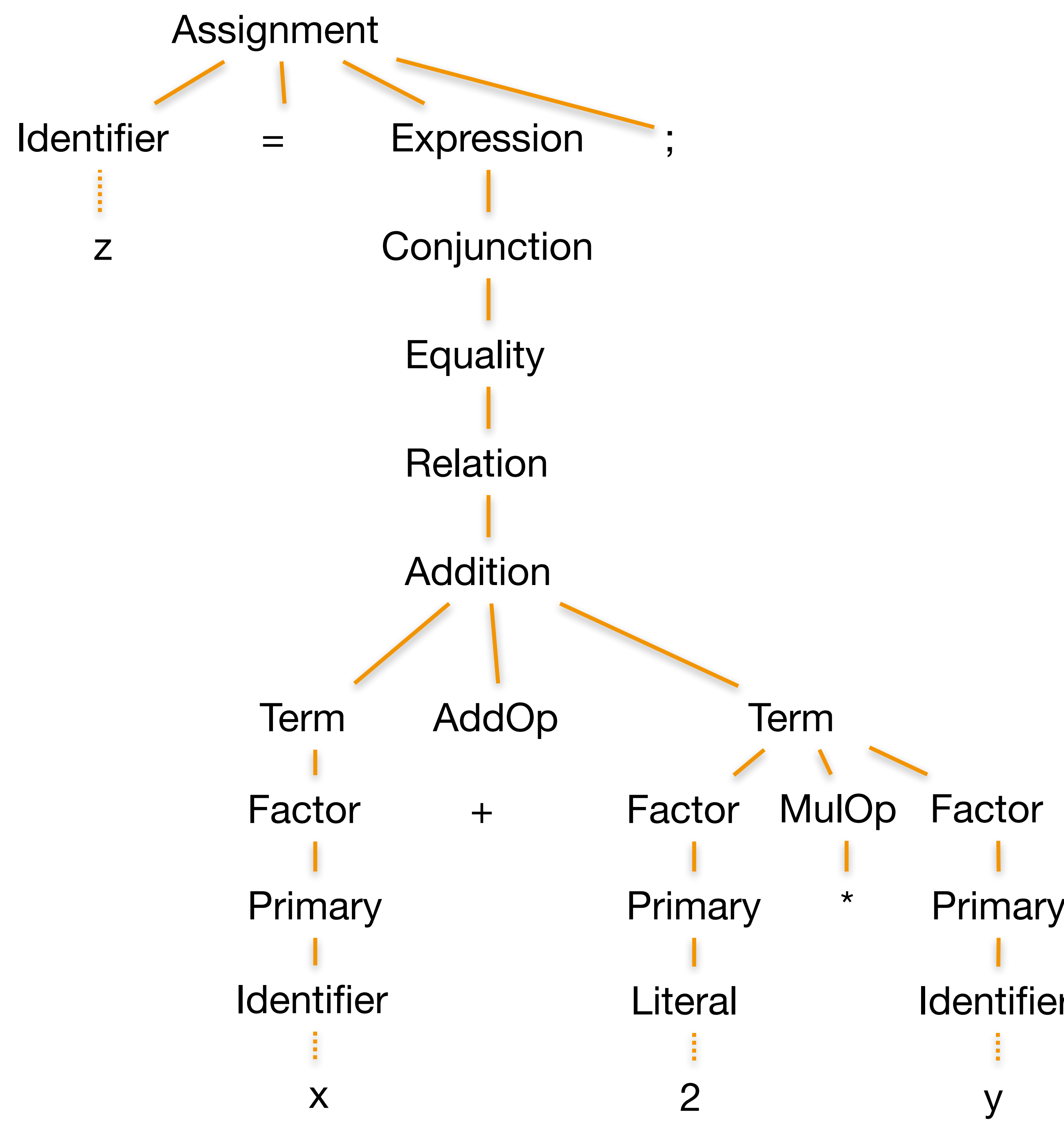
- Ambiguity of a context-free grammar is undecidable
- the ambiguity, however, can be removed by rewriting the grammar to have only one tree for the corresponding ‘string’

Decidability of Ambiguous Grammar

- Consider a grammar:
  - <expr> ::= <expr> + <expr> | <expr> - <expr> | <factor>
  - <factor> ::= <expr> * <expr> | <id>
  - <id> ::= x | y | z
- Considering the parse tree for string x-y-z
  - the grammar is ambiguous
- Rewriting the as follows would disambiguate the grammar:
  - <expr> ::= <expr> + <factor> | <expr> - <factor> | <factor>
  - <factor> ::= <expr> * <expr> | <id>
  - <id> ::= x | y | z
- Consider the parse tree for x+y*z
- Rewriting the as follows would disambiguate the grammar:
  - <expr> ::= <expr> + <term> | <expr> - <term> | <term>
  - <term> ::= <term> * <factor> | <factor>
  - <factor> ::= <id>
  - <id> ::= x | y | z

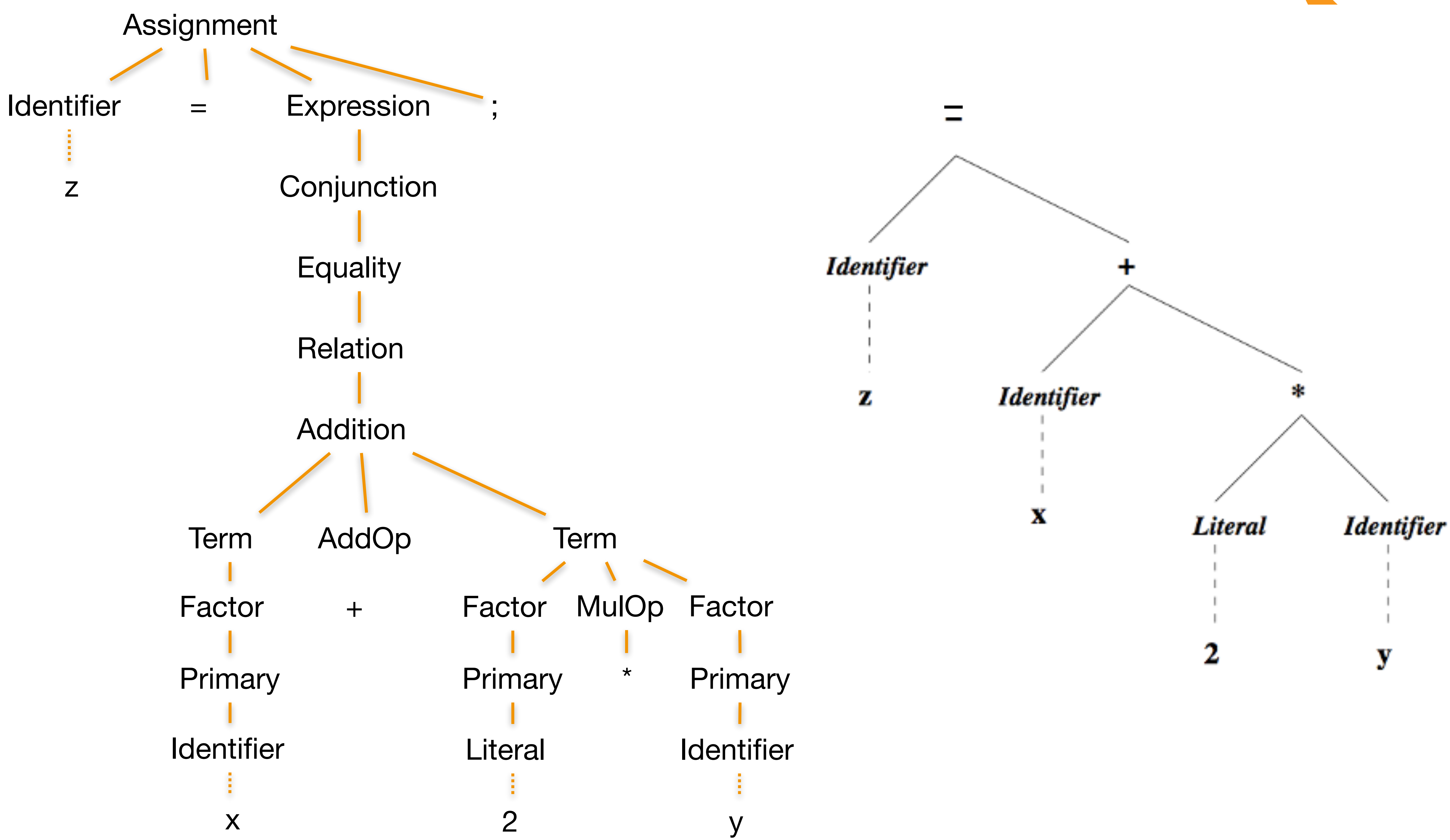
Recap: Parse Tree

Parse Tree for
z = x + 2*y;



- 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;



- Remove “syntactic sugar” and keep essential elements of a language
  - consider the following two loops:

Pascal

```
while i < n do begin
    i := i + 1;
end;
```

C/C++

```
while (i < n) {
    i = i + 1;
}
```

  - Essential information
    - both program contain a loop
    - the terminating condition, i.e., i < n
    - the inside statement, i.e., i is incremented

- Set of rules of the form: Lhs = Rhs
  - Lhs is the name of abstract syntactic class
  - Rhs defines the class as:
    - A list of one or more alternatives
      - e.g., Expression: Variable | Value | Binary | Unary
    - A list of essential components separated by semicolons (;)
    - Each component has the form of an ordinary class declaration, a list of one or more fields separated by ,
      - e.g., Binary = Operator op ; Expression term1, term2

Example Abstract Syntax of Clite Grammar

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

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

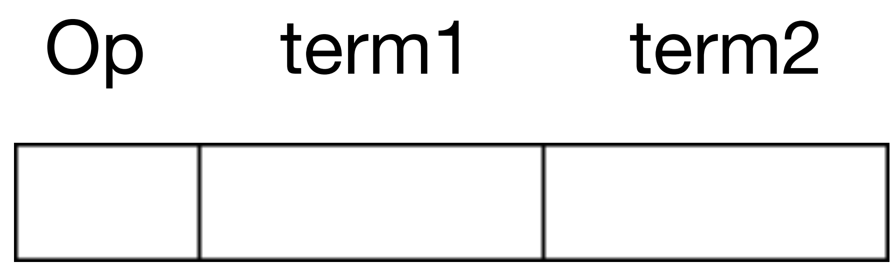
```
class Binary extends Expression {
   Operator op;
   Expression term1, term2;
}
```

```
abstract class Expression { }
```

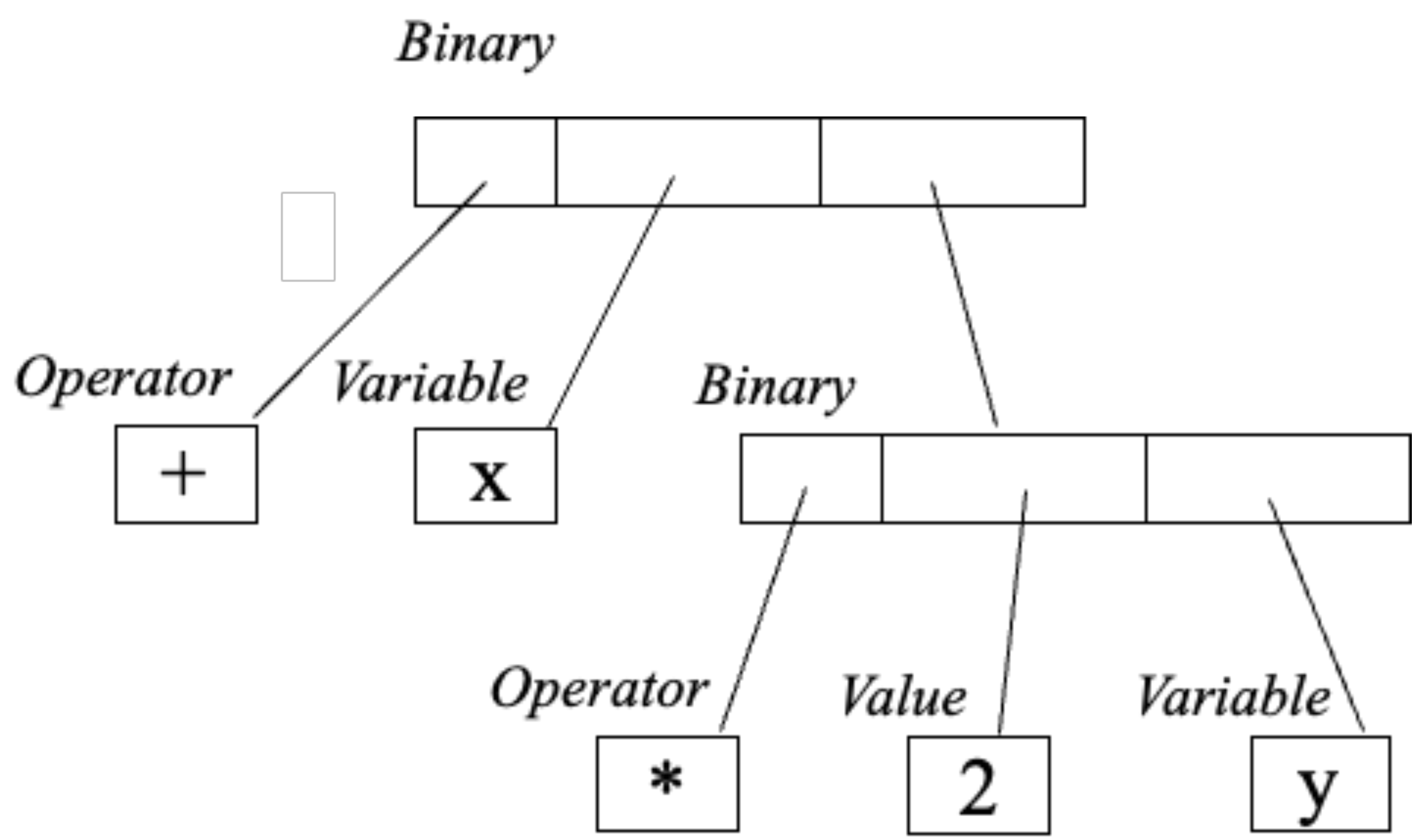
```
class Unary extends Expression {
   UnaryOp op;
   Expression term;
}
```

Example Abstract Syntax Tree

Binary node



Abstract Syntax Tree for
x + 2*y;



Names, Scope, & Bindings

- Binding is an association between an entity (e.g., a variable) and a property (e.g,. it’s value)
- A binding is static if the association occurs before runtime
- A binding is dynamic if the association occurs at runtime
    - the lifetime of a variable name refers to the time interval during which memory is allocated

Syntactic Issues

- Lexical rules for names
- Collection of reserved words or keywords
- Case sensitivity
  - C-like: yes
  - Early languages: no
  - PHP: partly yes

- Cannot be used as Identifiers
- Usually identify major constructs: if while switch
- Predefined identifiers, e.g., library routines

Variables

- Basic bindings
  - name
  - address
  - type
  - value
  - lifetime

- L-value: use of a variable name to denote its address
  - e.g., x = …
- R-value: use of a variable name to denote its value
  - e.g., … = …x…
- Some languages require explicit dereferencing
  - e.g., x := !y + 1

Pointer Example

```
int x, y;
int *p;
x = *p;
*p = y;
```

- Scope defines the collection of statements which can access the name binding.
- In *static* scoping, a name is bound to a collection of statements according to *its position* in the source program
  - can be performed at compile time
  - independent of the execution history of the program
- Most modern languages (C/C++, Java) use static (or lexical) scoping
  - improves program readability
  - enhanced compile-time checking

Scope

- Two different scopes are either nested or disjoint
- In disjoint scopes, same name can be bound to different entities without interference
- What constitutes a scope?

Higher-level Picture

	Algol	C	Java	Ada
Package	n/a	n/a	yes	yes
Class	n/a	n/a	nested	yes
Function	nested	yes	yes	nested
Block	nested	nested	nested	nested
For Loop	no	no	yes	automatic

- The scope in which a name is defined or declared is called its *defining scope*
- A reference to a name is *nonlocal* if it occurs in *a nested scope* of the defining scope; otherwise, it is local

Example in C

```c
void sort (float a[ ], int size) {
  int i, j;
  for (i = 0; i < size; i++)  // i, size local
    for (j = i + 1; j < size; j++)
      if (a[j] < a[i]) {  // a, i, j local
          float t;
          t = a[i];       // t local; a, i nonlocal
          a[i] = a[j];
          a[j] = t;
      }
}
```

```
for (int i = 0; i < 10; i++) {
      System.out.println(i);
      ...
}
... i ... // invalid reference to i
```

- A symbol table is a data structure kept by a translator that allows it to keep track of each declared name and its binding
- Assume for now that each name is unique within its local scope
- The data structure can be any implementation of a dictionary, where the name is the key

- Each time a scope is entered, push a new dictionary onto the stack
- Each time a scope is exited, pop a dictionary off the top of the stack
- For each name declared, generate an appropriate binding and enter the name-binding pair into the dictionary on the top of the stack
- Given a name reference, search the dictionary on top of the stack
  - if found, return its binding
  - else, repeat the process on the next dictionary down in the stack
  - if name is not found in any dictionary in the stack, report an error

Example in Blackboard

- For static scoping, the *referencing environment* for a name is its defining scope and all nested sub-scopes
- The referencing environment defines the set of statements which can validly reference a name

```
1 int h, i;
2 void B(int w) {
3         int j, k;
4         i = 2*w;
5         w = w+1;
6         ...
7 }
8 void A (int x, int y) {
9         float i, j;
10      B(h);
11      i = 3;
12      ...
13 }
14 void main() {
15        int a, b;
16        h = 5; a = 3; b = 2;
17        A(a, b);
18        B(h);
19        ...
20 }
```

- Outer scope: <h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
- Function B: <w, 2> <j, 3> <k, 3>
- Function A: <x, 8> <y, 8> <i, 9> <j, 9>
- Function main: <a, 15> <b, 15>

Example contd.

- Symbol table stack for function B:
  - <w, 2> <j, 3> <k, 3>
  - <h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
- Symbol table stack for function A:
  - <x, 8> <y, 8> <i, 9> <j, 9>
  - <h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
- Symbol table stack for function main:
  - <a, 15> <b, 15>
  - <h, 1> <i, 1> <B, 2> <A, 8> <main, 14>

Resolving references

```
1 int h, i;
2 void B(int w) {
3       int j, k;
4       i = 2*w;
5       w = w+1;
6       ...
7 }
8 void A (int x, int y) {
9     float i, j;
10     B(h);
11     i = 3;
12     ...
13 }
14 void main() {
15       int a, b;
16       h = 5; a = 3; b = 2;
17       A(a, b);
18       B(h);
19       ...
20 }
```

- Outer scope: <h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
- Function B: <w, 2> <j, 3> <k, 3>
- Function A: <x, 8> <y, 8> <i, 9> <j, 9>
- Function main: <a, 15> <b, 15>

Line	Reference	Declaration
4	i	1
10	h	1
11	i	9
16	h	1
18	h	1

Dynamic Scoping

- A name is bound to its most recent declaration based on program's call history
- Used in early versions of Lisp, APL, Snobol, Perl
- Symbol table for each scope built at compile time, but managed at runtime
- Scope pushed/popped on stack when entered/exited

Example

```
1 int h, i;
2 void B(int w) {
3         int j, k;
4         i = 2*w;
5         w = w+1;
6         ...
7 }
8 void A (int x, int y) {
9      float i, j;
10      B(h);
11      i = 3;
12      ...
13 }
14 void main() {
15         int a, b;
16         h = 5; a = 3; b = 2;
17         A(a, b);
18         B(h);
19         ...
20 }
```

call history:
main (17) -> A (10) -> B

Function:	**Dictionary**
B	<w, 2> <j, 3> <k, 3>
A	<x, 8> <y, 8> <i, 9> <j, 9>
main	<a, 15> <b, 15> <h, 1> <i, 1> <B, 2> <A, 8> <main, 14>

Reference to i (4) resolves to ?   <i, 9> in A

Example

```
1 int h, i;
2 void B(int w) {
3         int j, k;
4         i = 2*w;
5         w = w+1;
6         ...
7 }
8 void A (int x, int y) {
9       float i, j;
10      B(h);
11      i = 3;
12      ...
13 }
```

```
14 void main() {
15        int a, b;
16        h = 5; a = 3; b = 2;
17        A(a, b);
18        B(h);
19        ...
20 }
```

call history:
main (18) -> B

Function:	**Dictionary**
B	<w, 2> <j, 3> <k, 3>
main	<a, 15> <b, 15> <h, 1> <i, 1> <B, 2> <A, 8> <main, 14>

Reference to i (4) resolves to ? <i, 1> in Global scope

Visibility

- A name is visible if its referencing environment includes the reference and the name is not redeclared in an inner scope
- A name redeclared in an inner scope effectively hides the outer declaration
- Some language provide a mechanism for referencing a hidden name, e.g., this.field in C++/Java

```
public class Student {
  private String name;
  public Student (String name, ...) {
    this.name = name;
    ...
  }
```

- Overloading uses the number or type of parameters to distinguish among identical function names or operators
  - +, - , *, / can be float or int
  - + can be float or int addition or string concatenation on Java
  - System.out.println(x) in Java

- Modula: library functions
  - Read() for characters
  - ReadReal() for floating point
  - ReadInt() for integers
  - ReadString() for strings

Reference Materials

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