CSCI312 Principles of Programming Languages

Chapter 2

Syntax

Xu Liu

Project: derive

Work in pairs: follow the rule

Terminal string length: number of terminals in the derivation:

e.g., (id) has length 3

(id + id) has length 5

Recap

BNF Grammar

Set of *productions*: P

terminal symbols: T

nonterminal symbols: N

start symbol: $S \in N$

A *production* has the form

$$A \rightarrow \omega$$

where $A \in N$ and $\omega \in (N \cup T)^*$

Derivation of 352 from a BNF Grammar

Integer → *Digit* | *Integer Digit*

Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Integer ⇒ *Integer Digit*

⇒ *Integer Digit Digit*

⇒ *Digit Digit Digit*

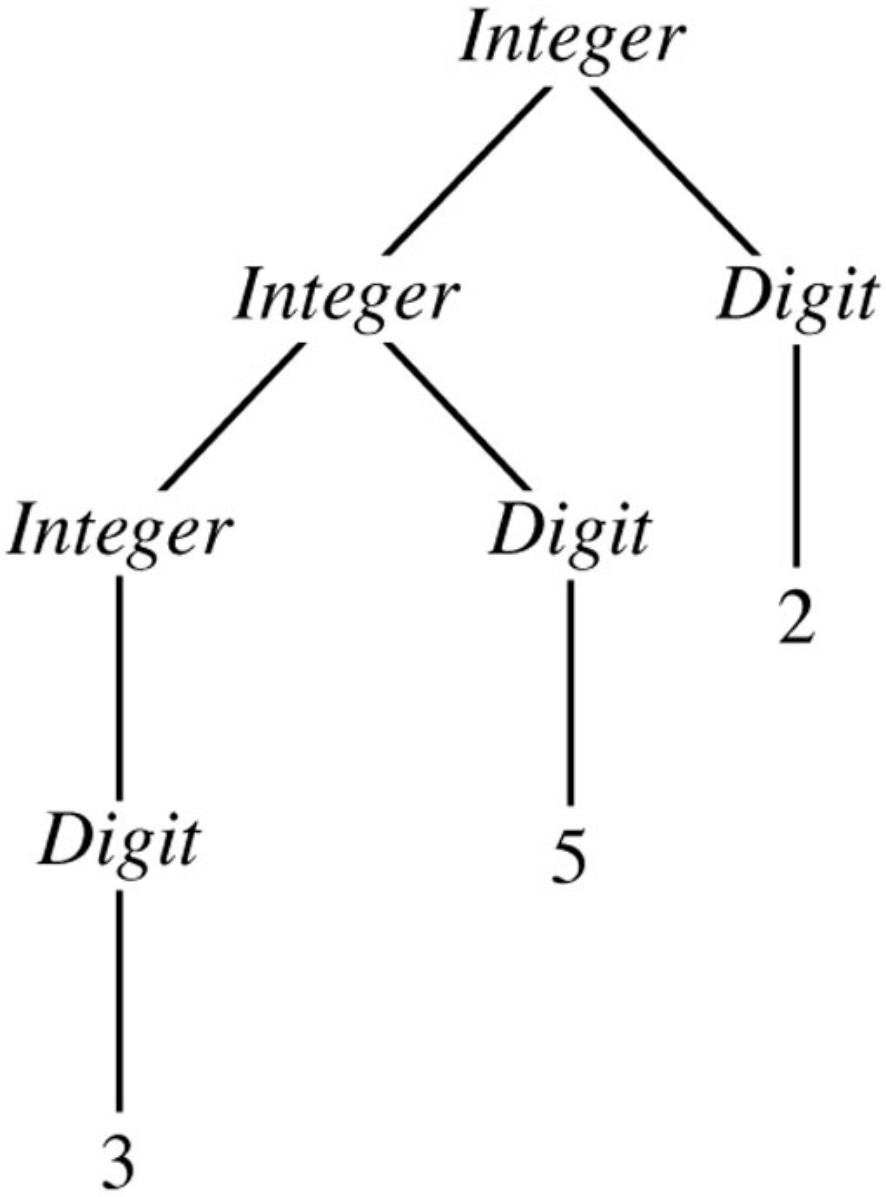
⇒ 3 *Digit Digit*

⇒ 3 5 *Digit*

⇒ 3 5 2

Parse Tree for 352 as an *Integer*

Figure 2.1



Ambiguity

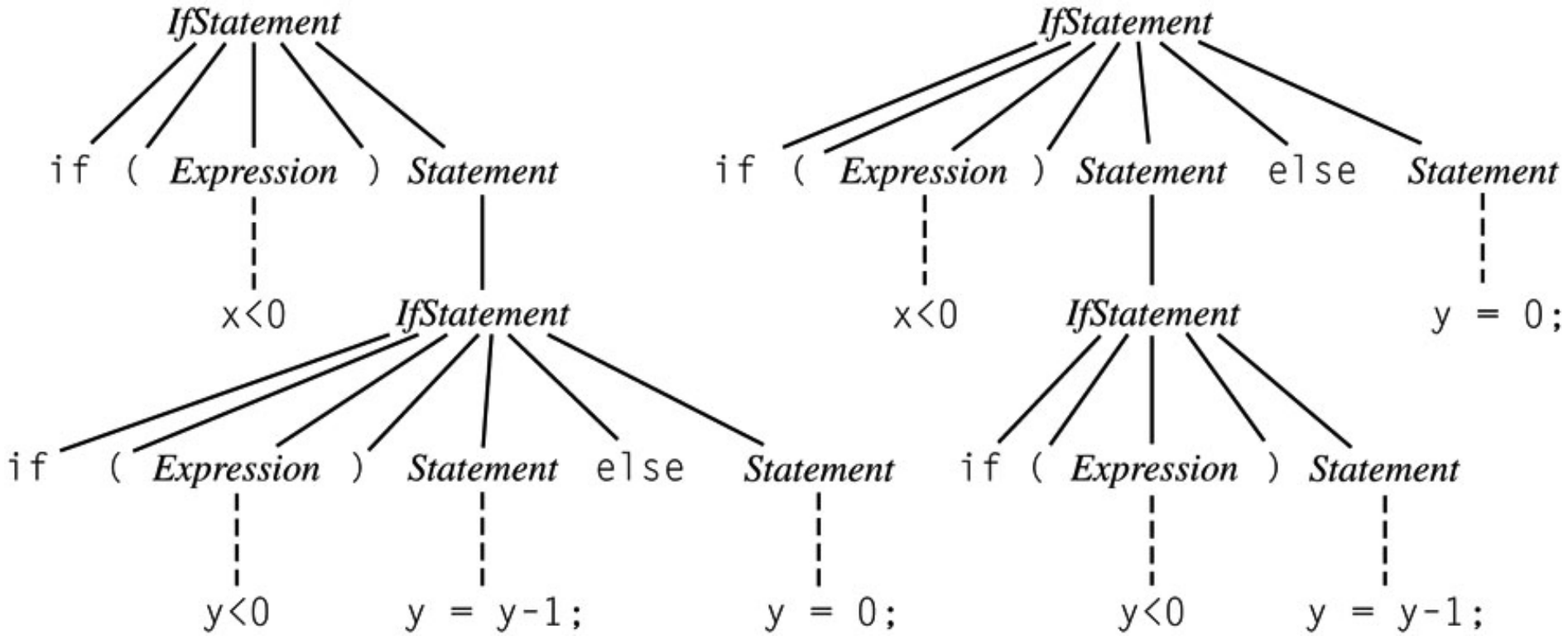
With which ‘if’ does the following ‘else’ associate

```
if (x < 0)
        if (y < 0)  y = y  - 1;
        else y = 0;
```

Answer: *either one!*

The *Dangling Else* Ambiguity

Figure 2.5



Extended BNF (EBNF)

BNF:

- *recursion for iteration*
- *nonterminals for grouping*

EBNF: additional metacharacters

- **{ }** for a series of zero or more
- **( )** for a list, must pick one
- **[ ]** for an optional list; pick none or one

Contents

2.3 Syntax of a Small Language: *Clite*

Motivation for using a subset of C:

Language	*Grammar (pages)*	*Reference*
Pascal	5	Jensen & Wirth
C	6	Kernighan & Richie
C++	22	Stroustrup
Java	14	Gosling, et. al.

The *Clite* grammar fits on one page (next 3 slides), so it's a far better tool for studying language design.

Fig. 2.7 *Clite* Grammar: Statements

Program → `int main ( ) {` *Declarations Statements* `}`

Declarations → { *Declaration* }

Declaration → *Type Identifier* [[*Integer*]] { , *Identifier* [[*Integer*]] }

Type → `int` | `bool` | `float` | `char`

Statements → { *Statement* }

Statement → `;` | *Block* | *Assignment* | *IfStatement* | *WhileStatement*

Block → { *Statements* }

Assignment → *Identifier* [[*Expression*]] = *Expression* `;`

IfStatement → `if (` *Expression* `)` *Statement* [`else` *Statement*]

WhileStatement → `while (` *Expression* `)` *Statement*

Fig. 2.7 *Clite* Grammar: 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*)

Fig. 2.7 *Clite* grammar: lexical level

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* ‘

Issues Not Addressed by this Grammar

- Comments
- Whitespace
- Distinguishing one token <= from two tokens < =
- Distinguishing identifiers from keywords like if

These issues are addressed by identifying two levels:

  - *lexical level*
  - *syntactic level*

2.3.1 Lexical Syntax

Input: a stream of characters from the ASCII set, keyed by a programmer.

Output: a stream of *tokens* or basic symbols, classified as follows:

- *Identifiers* e.g., Stack, x, i, push
- *Literals* e.g., 123, 'x', 3.25, true
- *Keywords* bool char else false float if int main true while
- *Operators* = || && == != < <= > >= + - * / !
- *Punctuation* ; , { } ( )

Whitespace

Whitespace is any space, tab, end-of-line character (or characters), or character sequence inside a comment

No token may contain embedded whitespace (unless it is a character or string literal)

Example:

`>=` *one token*

`> =` *two tokens*

Whitespace Examples in Pascal

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

Comments

Not defined in grammar

Clite uses // comment style of C++

Identifier

Sequence of letters and digits, starting with a letter

- `if` is both an identifier and a keyword
- Most languages require identifiers to be distinct from keywords

In some languages, identifiers are merely predefined (and thus can be redefined by the programmer)

Redefining Identifiers can be dangerous

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

Should Identifiers be case-sensitive?

Older languages: no

- *Pascal: no*
- *Modula: yes*
- *C, C++: yes*
- *Java: yes*
- *PHP: partly yes, partly no*

2.3.2 Concrete Syntax

Based on a parse of its *Tokens*

; is a statement terminator

(Algol-60, Pascal use ; as a separator)

Rule for *IfStatement* is ambiguous:

"The else ambiguity is resolved by connecting an **else** with the last encountered else-less if."

[Stroustrup, 1991]

Expressions in *Clite*

13 grammar rules

Use of meta braces – operators are left associative

C++ expressions require 4 pages of grammar rules
[Stroustrup]

C uses an ambiguous expression grammar
[Kernighan and Ritchie]

Associativity and Precedence

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

Clite Equality, Relational Operators

… are non-associative.

(an idea borrowed from Ada)

Why is this important?

In C++, the expression:

```
if (a < x < b)
```

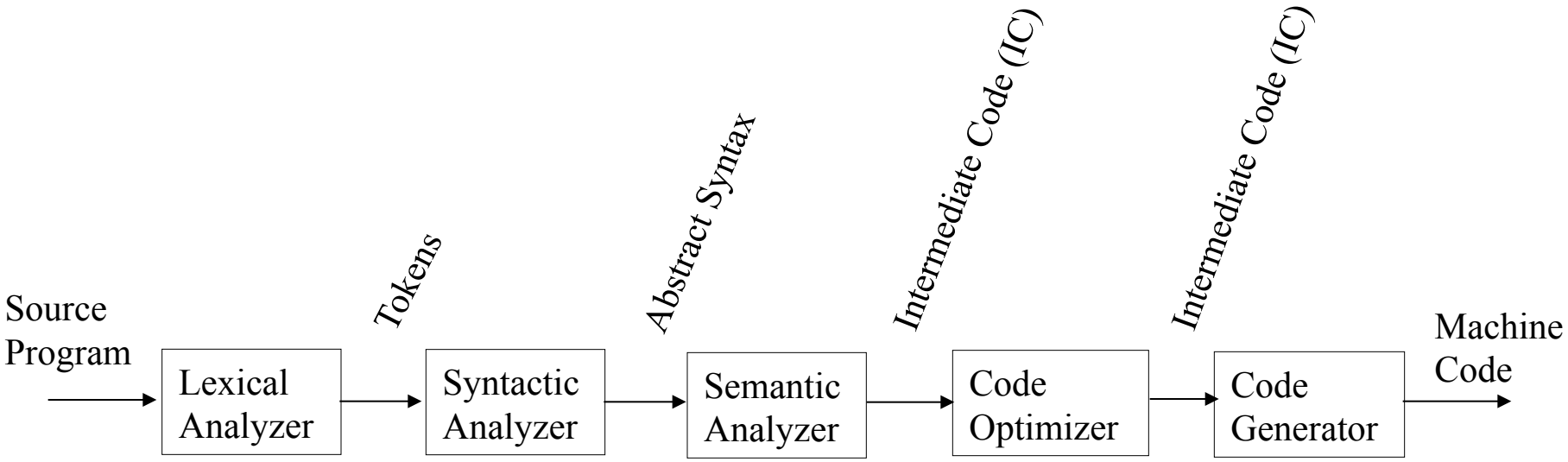
is *not* equivalent to

```
if (a < x && x < b)
```

But it is error-free!

So, what does it mean?

2.4 Compilers and Interpreters



Lexer

- Input: characters
- Output: tokens
- Separate:
  - *Speed: 75% of time for non-optimizing compilers*
  - *Simpler design*
  - *Character sets*
  - *End of line conventions*

Parser

- Based on BNF/EBNF grammar
- Input: tokens
- Output: abstract syntax tree (parse tree)
- Abstract syntax: parse tree with punctuation, many nonterminals discarded

Semantic Analysis

- Check that all identifiers are declared
- Perform type checking
- Insert implied conversion operators

  (i.e., make them explicit)

Code Optimization

- Evaluate constant expressions at compile-time
- Reorder code to improve cache performance
- Eliminate common subexpressions
- Eliminate unnecessary code

Code Generation

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

Interpreter

Replaces last 2 phases of a compiler

Input:

- *Mixed: intermediate code*
- *Pure: stream of ASCII characters*

Mixed interpreters

- *Java, Perl, Python, Haskell, Scheme*

Pure interpreters:

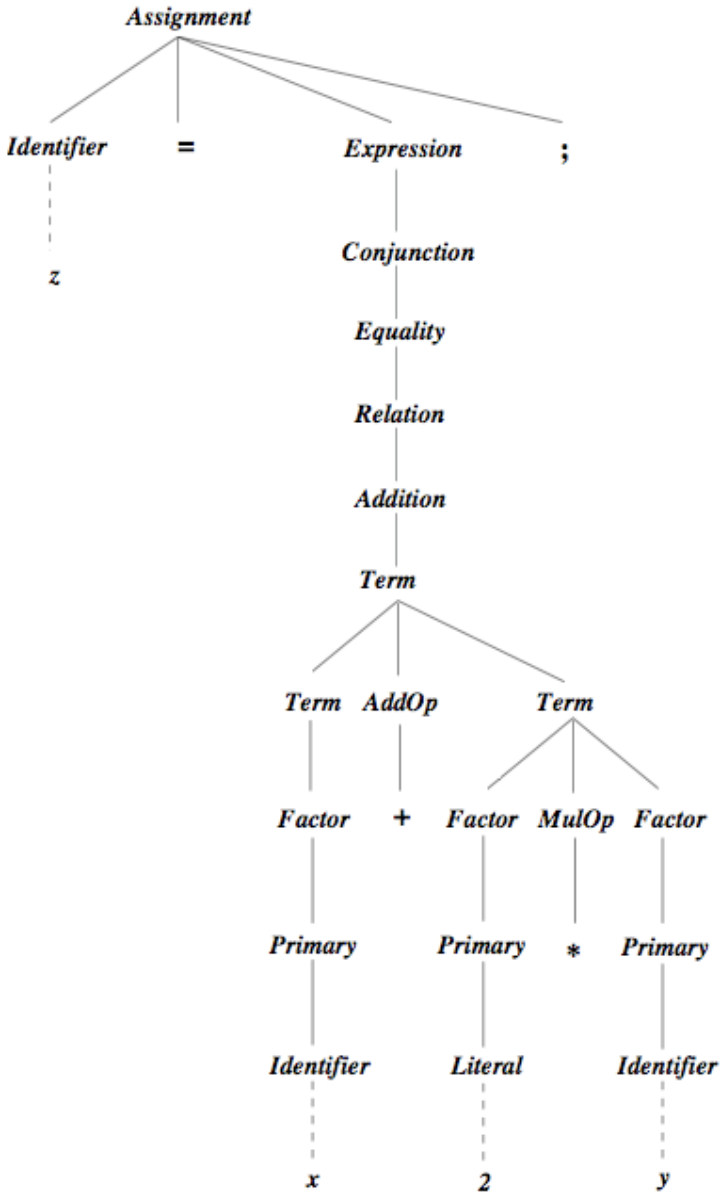
- *most Basics, shell commands*

2.5 Linking Syntax and Semantics

Output: parse tree is inefficient

Example: Fig. 2.9

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



Finding a More Efficient Tree

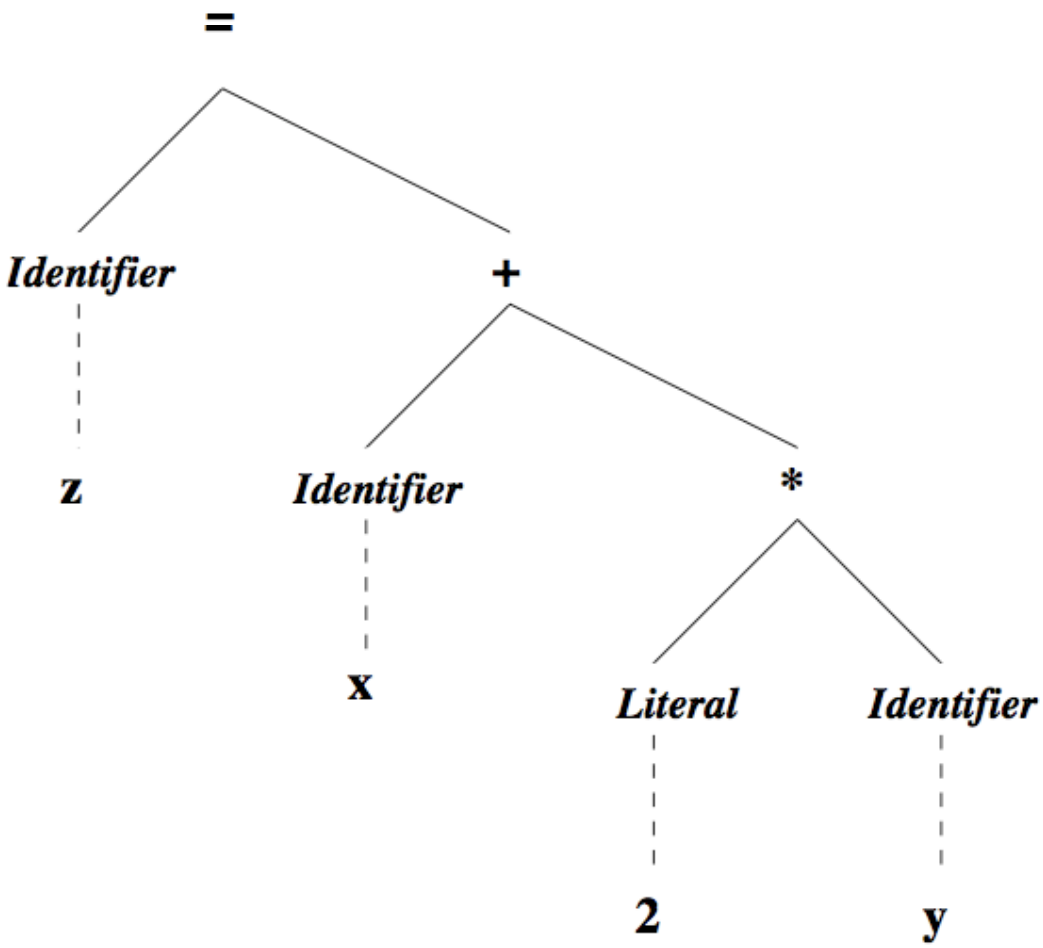
The *shape* of the parse tree reveals the meaning of the program.

So we want a tree that removes its inefficiency and keeps its shape.

- *Remove separator/punctuation terminal symbols*
- *Remove all trivial root nonterminals*
- *Replace remaining nonterminals with leaf terminals*

Example: Fig. 2.10

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



Abstract Syntax

Removes "syntactic sugar" and keeps essential elements of a language. E.g., consider the following two equivalent loops:

Pascal

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

C/C++

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

The only essential information in each of these is 1) that it is a *loop*, 2) that its terminating condition is `i < n`, and 3) that its body increments the current value of `i`.

Abstract Syntax of *Clite* Assignments

Assignment = *Variable* target; *Expression* source

Expression = *VariableRef* | *Value* | *Binary* | *Unary*

VariableRef = *Variable* | *ArrayRef*

Variable = *String* id

ArrayRef = *String* id; *Expression* index

Value = *IntValue* | *BoolValue* | *FloatValue* | *CharValue*

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

Unary = *UnaryOp* op; *Expression* term

Operator = *ArithmeticOp* | *RelationalOp* | *BooleanOp*

IntValue = *Integer* intValue

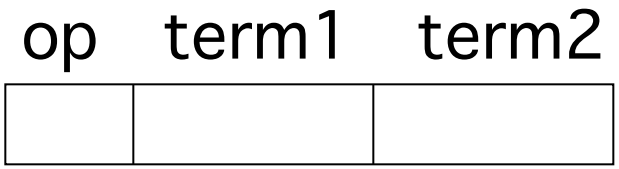
...

Abstract Syntax as Java Classes

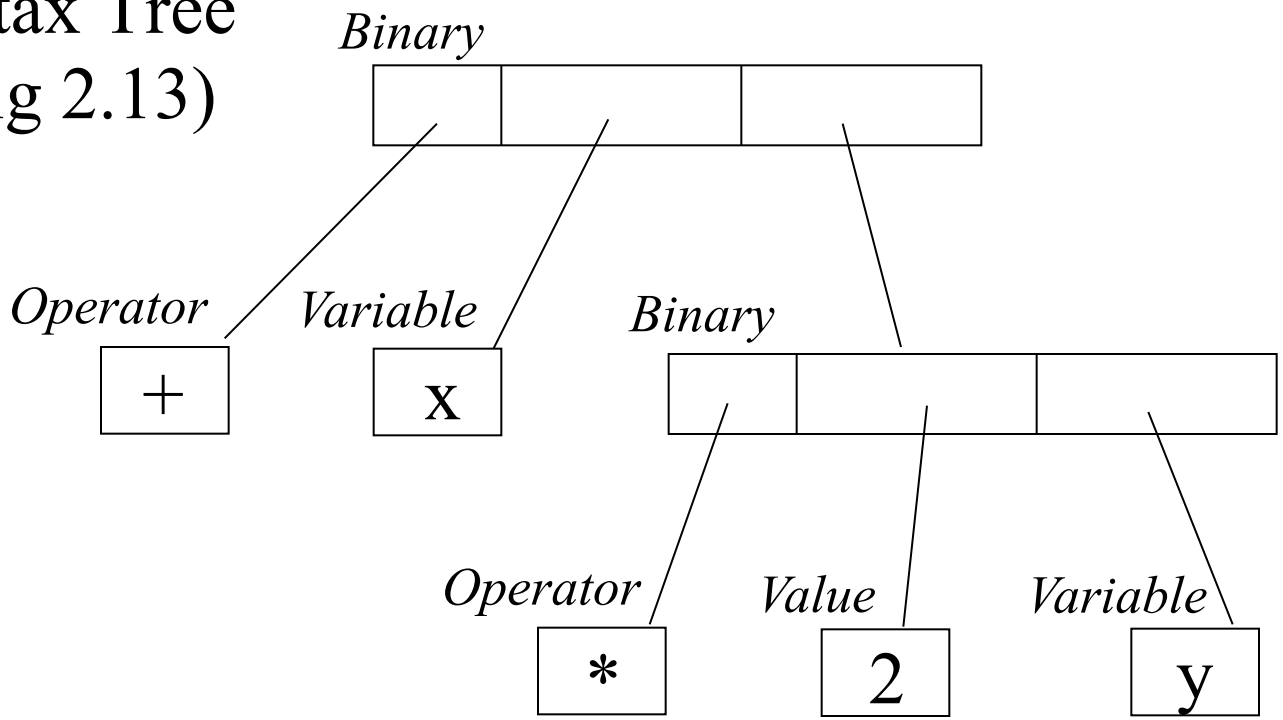
```
abstract class Expression { }
abstract class VariableRef extends Expression { }
class Variable extends VariableRef { String id; }
class Value extends Expression { ... }
class Binary extends Expression {
    Operator op;
    Expression term1, term2;
}
class Unary extends Expression {
    UnaryOp op;
    Expression term;
}
```

Example Abstract Syntax Tree

Binary node



Abstract Syntax Tree for x+2*y (Fig 2.13)



Remaining Abstract Syntax of *Clite* (*Declarations* and *Statements*) Fig 2.14

$$
\begin{aligned}
\mathit{Program} &= \mathit{Declarations}\ \texttt{decpart;}\ \ \mathit{Statements}\ \texttt{body;} \\
\mathit{Declarations} &= \mathit{Declaration}^{*} \\
\mathit{Declaration} &= \mathit{VariableDecl} \mid \mathit{ArrayDecl} \\
\mathit{VariableDecl} &= \mathit{Variable}\ \texttt{v;}\ \mathit{Type}\ \texttt{t} \\
\mathit{ArrayDecl} &= \mathit{Variable}\ \texttt{v;}\ \mathit{Type}\ \texttt{t;}\ \mathit{Integer}\ \texttt{size} \\
\mathit{Type} &= \ \texttt{int} \mid \texttt{bool} \mid \texttt{float} \mid \texttt{char} \\
\mathit{Statements} &= \mathit{Statement}^{*} \\
\mathit{Statement} &= \mathit{Skip} \mid \mathit{Block} \mid \mathit{Assignment} \mid \mathit{Conditional} \mid \mathit{Loop} \\
\mathit{Skip} &= \\
\mathit{Block} &= \mathit{Statements} \\
\mathit{Conditional} &= \mathit{Expression}\ \texttt{test;}\ \mathit{Statement}\ \texttt{thenbranch, elsebranch} \\
\mathit{Loop} &= \mathit{Expression}\ \texttt{test;}\ \mathit{Statement}\ \texttt{body}
\end{aligned}
$$