CSCI 2320
Principles of Programming Languages

Syntax
Reading: Chapter 2 of Tucker-Noonan

MOHAMMAD T. IRFAN

See class notes on CFG/BNF

Expr → Expr + Prod | Expr – Prod | Prod
Prod → Prod * Term | Prod / Term | Term
Term → 0 | 1 | … | 9 | (Expr)

Lessons

1. Left recursion for left associativity and right recursion for right associativity
2. Push higher precedence operators further down from the start symbol by creating new non-terminal symbols

Terminology

- Syntax (form): expressed by grammar
- CFG/BNF: (1) terminal (2) nonterminal (3) production
- Left-recursive vs. right-recursive production
- Derivation & parse tree
  - Left-most vs. right-most derivation
- Sentential form
- Sentence
- Language

Ambiguous Grammar

SEE CLASS NOTES

Resolving ambiguity: C++

selection-statement:

`if (` *condition* `)` *statement*

`if (` *condition* `)` *statement* `else` *statement*

C++ Grammar (on Canvas)

"The else ambiguity is resolved by connecting an else with the last encountered else-less if." [Stroustrup, 1991]

Resolving ambiguity: Java

https://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html#jls-14.5

- Statement:

  IfThenStatement

  IfThenElseStatement

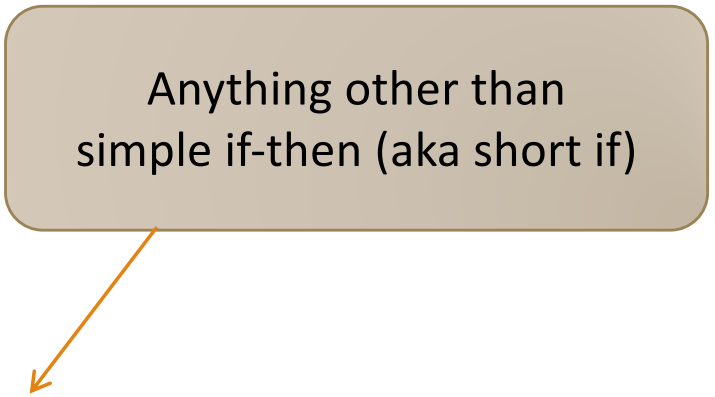
  ...

- IfThenStatement:

  if (Expression) Statement

- IfThenElseStatement:

  if (Expression) StatementNoShortIf else Statement



Java:

Stmt → IfThenStmt | IfThenElseStmt | ...

IfThenStmt → if (Expression) Stmt

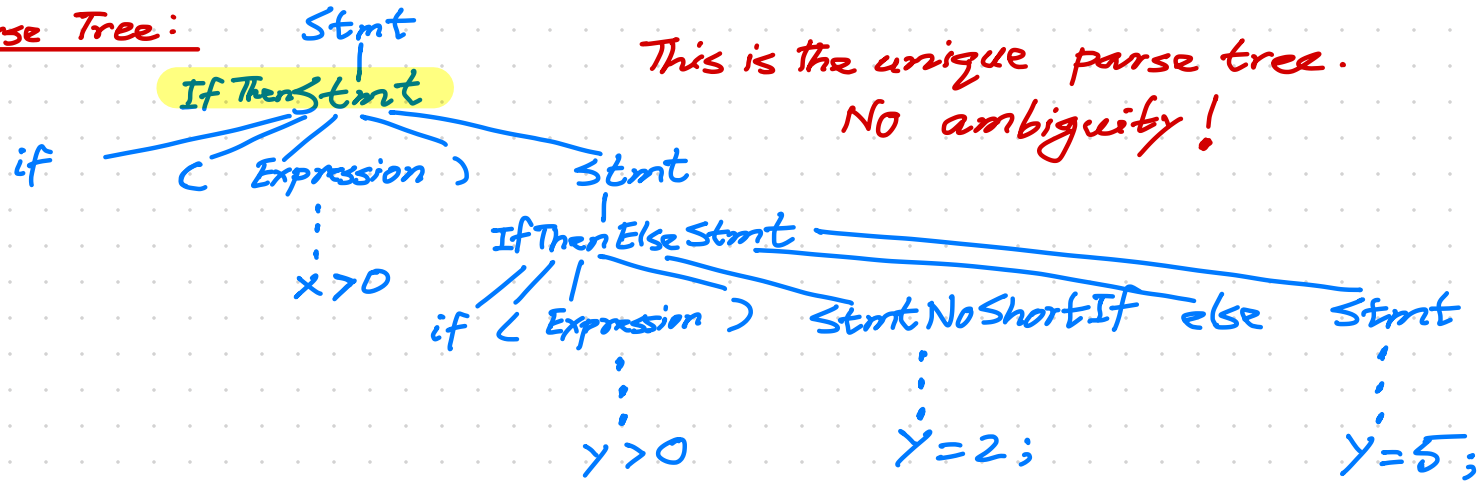
IfThenElseStmt → if (Expression) StmtNoShortIf else Stmt

Any stmt except simple if-then

Example:

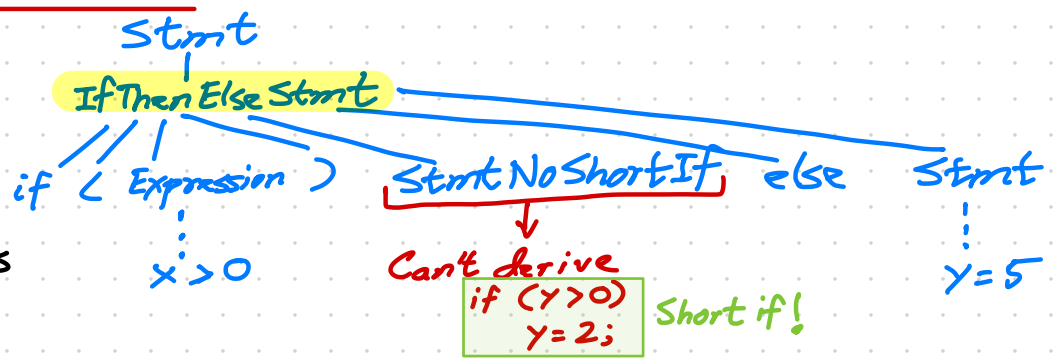
```
if (x>0)
  if (y>0)
    y=2;
  else y=5;
```

Parse Tree:



This is the unique parse tree.
No ambiguity!

Failed Parse Tree:



This parse tree works in C++, but not in Java due to Java's StmtNoShortIf.

Resolving ambiguity: Algol 68

- By syntax of if-fi
- Example:

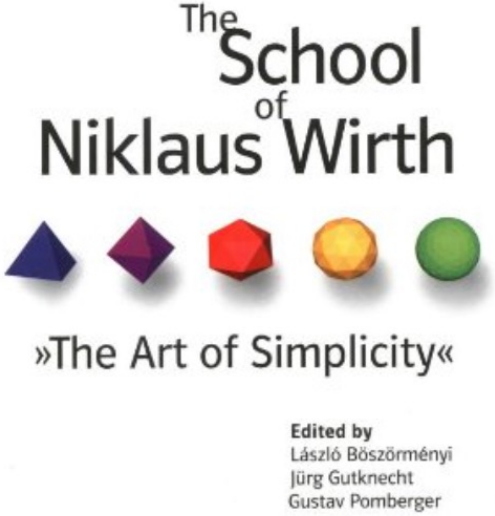
```
if (x > 0)
    if (y > 0)
        y := -5;
    else
        y := 5;
    fi
fi
```

What about Python?

New term

- Ambiguous grammar: 2 parse trees for the same sentence

Extended BNF (EBNF)



http://pascal.hansotten.com/niklaus-wirth/

ITERATIONS, OPTIONS, AND CHOICES

BNF IS AS POWERFUL AS EBNF

BNF is as powerful as EBNF

Proof.

	EBNF	BNF
Iteration	$A \to x\{y\}z$	$A \to xA'z$ $A' \to \in \mid yA'$
Option	$A \to x[y]z$	$A \to xA'z$ $A' \to y \mid \in$
Choice	$A \to w(x \mid y)z$	$A \to wxz \mid wyz$

Grammar for CLite

LEXICAL SYNTAX (LOW LEVEL: MOSTLY TYPED THINGS)

CONCRETE SYNTAX (HIGHER LEVELS OF ABSTRACTION)

Lexical Syntax

Identifier → *Letter* { (*Letter* | *Digit*) }

Blue color: EBNF meta-symbol

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 → `'`*ASCIIChar*`'`

ASCIIChar is an ASCII character

Concrete Syntax (EBNF): Hierarchy of categories/abstractions

Program → *Type* `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*

Blue color: EBNF meta-symbol

Higher precedence

Precedence of operators in C

1	( ), [ ], Type ( )	Parentheses, array subscript, type casting
2	+, –	Unary + or –
3	* / %	Multiplication, division, and remainder
4	+ -	Addition and subtraction
5	<< >>	Bitwise left shift and right shift — We ignore in CLite
6	< <= > >=	For relational operators < and ≤ respectively For relational operators > and ≥ respectively
7	== !=	For relational = and ≠ respectively
8	&	Bitwise AND
9	^	Bitwise XOR (exclusive or) — We ignore in CLite
10	|	Bitwise OR (inclusive or)
11	&&	Logical AND
12	||	Logical OR

http://en.cppreference.com/w/c/language/operator_precedence

Concrete Syntax (cont…)

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*)

Note: Operators and punctuation symbols are part of "lexical syntax" (next)

Draw the parse tree

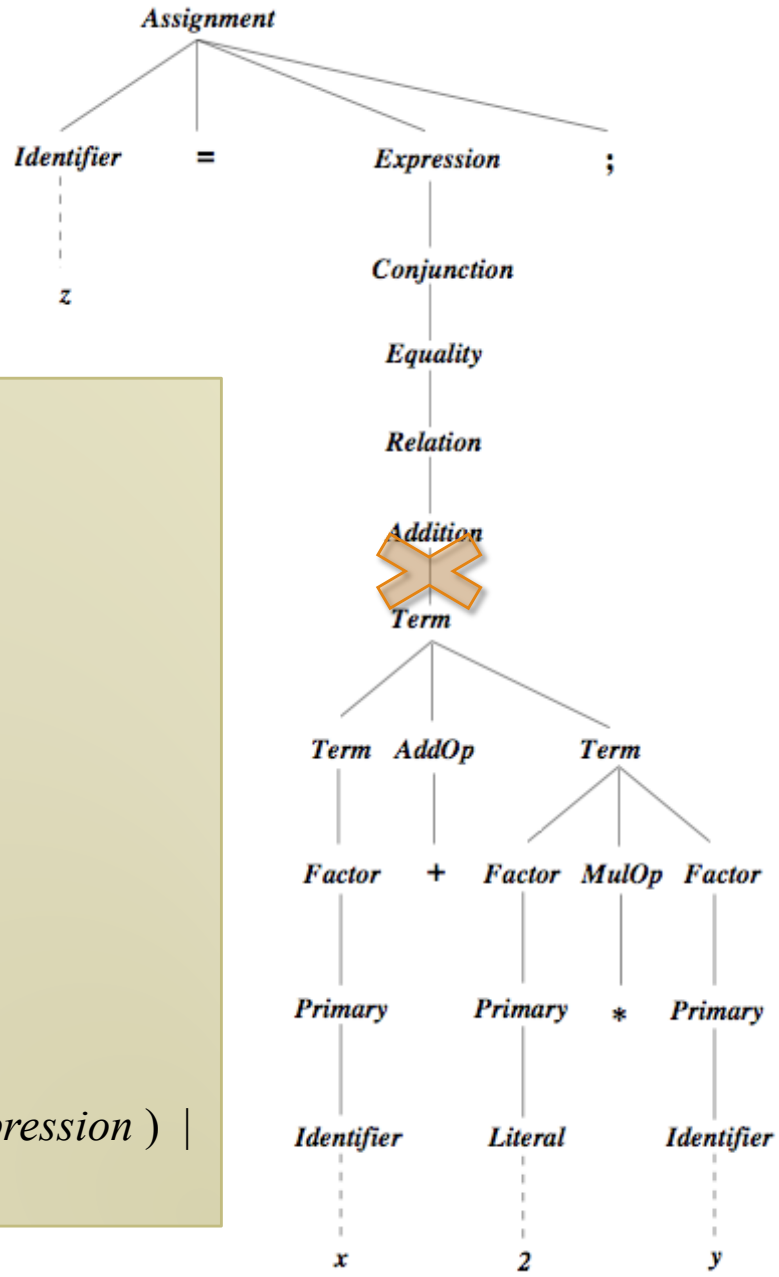
```
1  int main()
2  {
3      int i, j;
4      int k;
5      i = 10;
6      j = i;
7  }
```

Abstract Syntax Tree (AST)

LINKING SYNTAX WITH SEMANTICS

Parse Tree for z = x + 2*y;

Fig. 2.9



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* }

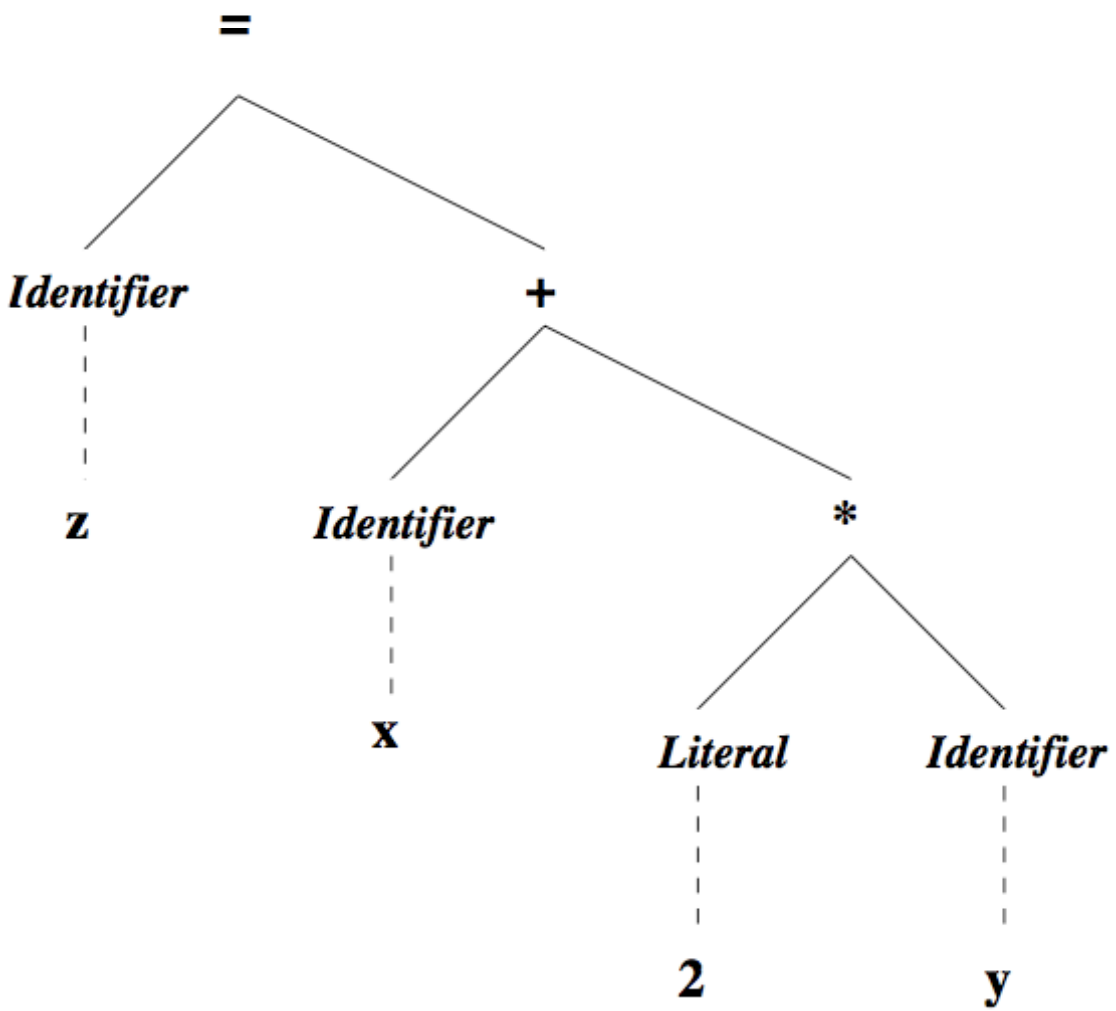
MulOp → * | / | %

Factor → [*UnaryOp*] *Primary*

UnaryOp → − | !

Primary → *Identifier* [[*Expression*]] | *Literal* | (*Expression*) | *Type* (*Expression*)

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



Optional Reading

Implementation of AST

Pages 49 -- 54

Vocabulary

- Syntax (form): expressed by grammar
- CFG/BNF: (1) terminal (2) nonterminal (3) production
- Left-recursive vs. right-recursive production
- Derivation: left-most vs. right-most derivation
- Parse tree
- Sentential form
- Sentence
- Language
- Ambiguous grammar
- Concrete syntax and lexical syntax
- Abstract syntax tree