Lecture 2: Syntax and Semantics

Part A: Describing Syntax

Part B: Describing Semantics

Chapter 3

Part A: Describing Syntax

Introduction

- What is syntax? What does syntax describe?
  - Syntax: the form or structure of the expressions, statements, and program units
- Syntax vs. Semantics
  - What is semantics? What does semantics describe?
  - Semantics: the meaning of the expressions, statements, and program units
  - Syntax and semantics provide a language's definition
  - Users of a language definition
    - Other language designers
    - Implementers
    - Programmers (the users of the language)

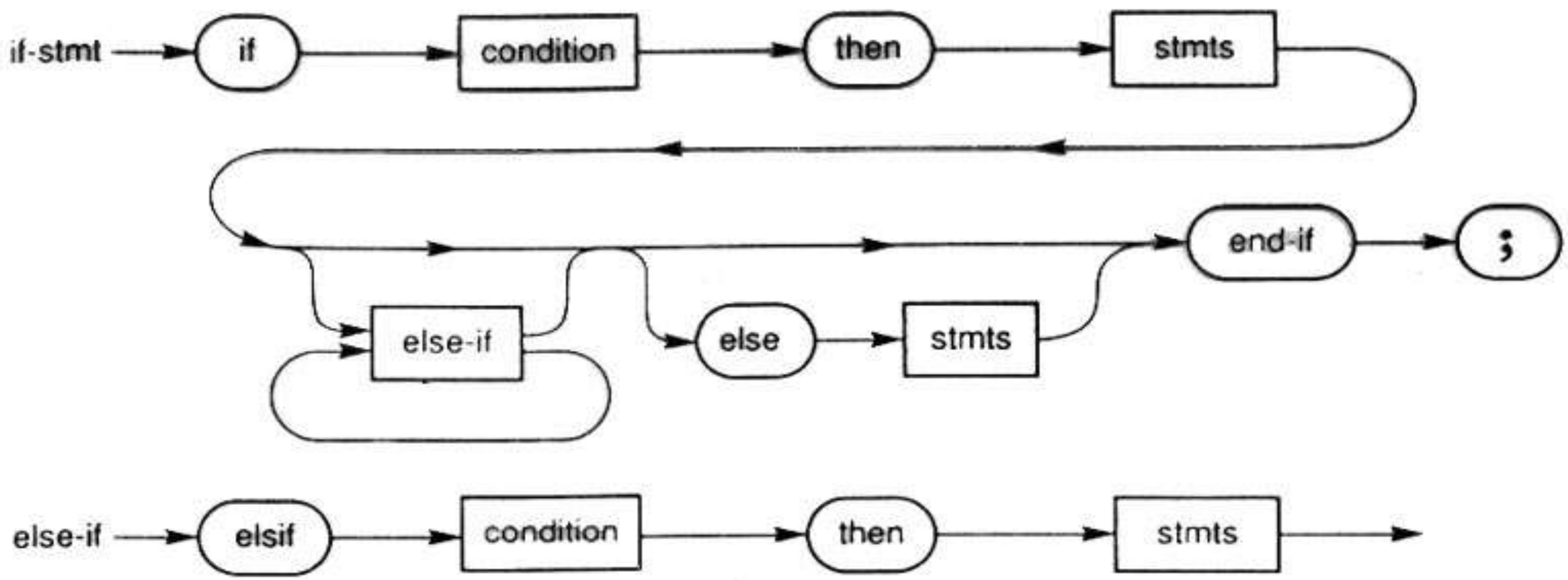
Methods for Describing Syntax

- Syntax Diagrams
  - Informal, used in early days
- Formal descriptions
  - Context Free Grammar (CFG)
    - CS3110
  - Backus-Naur Form (BNF)
    - Equivalent to CFG
    - More readable notations
      - E.g. E -> EaT | T in CFG could be described in BNF as:
        <Expr> -> <Expr> + <Term> | <Term>
  - Extended BNF (EBNF)
    - A non-recursive way (iterative way) to express CFG

Describing the syntax

- Syntax diagram
  - Simple graphical representation, easy to read but not powerful enough to describe complexity in syntax, not support for compiler generator

Syntax diagram: Example



More examples

[Database SQL]

[Java syntax diagrams]

BNF and Context-Free Grammars

- Context-Free Grammars (CFG)
  - Developed by Noam Chomsky in the mid-1950s
  - Language generators, meant to describe the syntax of natural languages
  - Define a class of languages called context-free languages
  - CS3110: Formal Languages and Automata
- Backus-Naur Form BNF
  - Invented by John Backus to describe Algol 58
  - Published in 1959
  - BNF is equivalent to context-free grammars

Sample BNF Rules

- An abstraction (or nonterminal symbol) can have more than one RHS

```
<stmt> → <single_stmt>
       | begin <stmt_list> end
<single_stmt> → …
<stmt_list> → …
```

A name inside <>, e.g. <stmt> is a nonterminal symbol to be defined

An Example Grammar

```
<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const
```

- Ambiguous and unambiguous grammars
  - The above is an ambiguous grammar
    - Details discussed in CS3110, omitted here.

Extended BNF (EBNF)

- Optional parts are placed in brackets [ ]

```
<fun_call> -> ident '('[<expr_list>]')'
```

- Alternative parts of RHSs are placed inside parentheses () and separated via vertical bars |

```
<term> → <term> (+|-) const
```

- Repetitions (0 or more) are placed inside braces { }

```
<ident> → letter {letter|digit}
```

- Note: similar to regular expression notation

Example

- The following are a few examples of function calls or a Java-like static method calls

  f();   f(a, a+b);   f(x*y, 3*x, 4);

- From above we summarize the function call as:

  A function name followed by (), or

  A function name followed by a list of parameters inside the () where each parameter could be an expression.

- Thus, the EBNF syntax rule for the function call could be

  <fun_call> -> id ‘(‘[<expr_list>]’)’

  - Optional parts are placed in brackets [ ]
    - Parameters are optional

BNF and EBNF

- BNF

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
```

- EBNF

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
```

Recent Variations in EBNF

- Alternative right-hand sides are put on separate lines
  - Note: we have followed this convention
- Use of a colon : instead of =>

  `<expr> : <term> {(+ | -)`$_{\texttt{oneof}}$ `<term>}`

- Use of $_{\texttt{opt}}$ for optional parts
- Use of `oneof` for choices

(For information only.)

Examples

- Java Syntax Rules
  - Java Syntax by Oracle
  - Java Syntax Diagram and EBNF

- Python Syntax Rules

  Python 3 Syntax

Note: Since BNF no longer practically used, we now often use BNF referring to EBNF syntax description.

Interpreting Syntax Rules: if-stmt

- Java Syntax Rules
  - Java Syntax by Oracle
  - Java Syntax Diagram and EBNF

Oracle:

IfThenStatement:
  if (Expression) Statement

IfThenElseStatement:
  if (Expression) StatementNoShortIf else Statement

IfThenElseStatementNoShortIf:
  if (Expression) StatementNoShortIf else StatementNoShortIf

Cui:

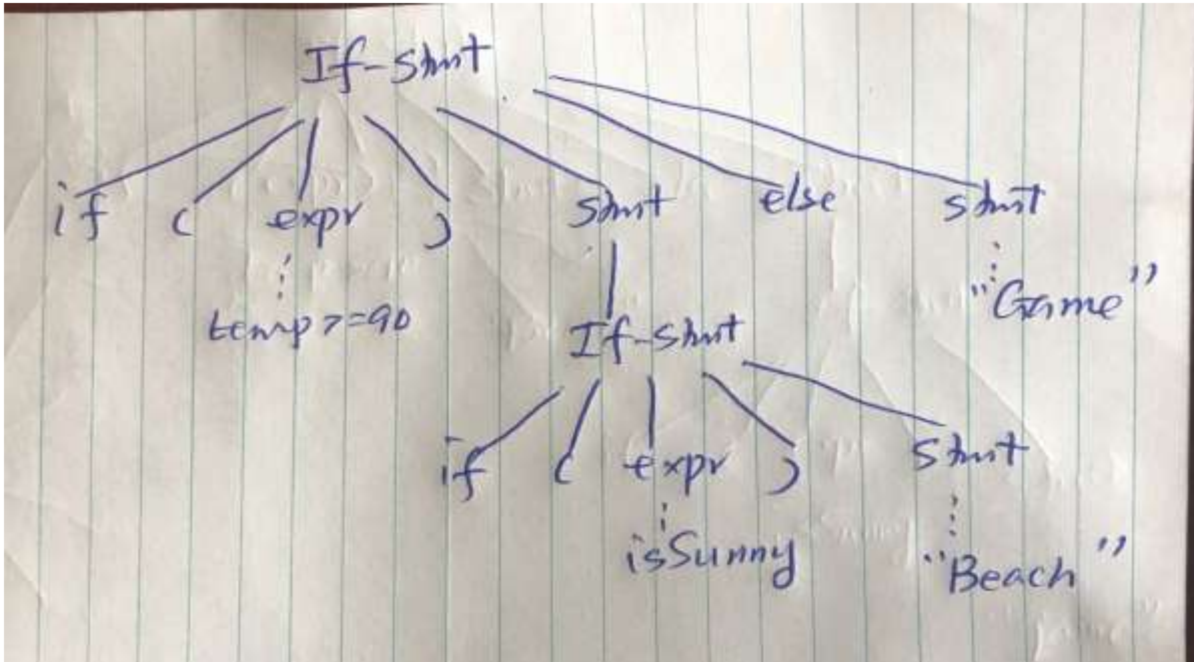
if_statement ::= **"if" "("** expression **")"** statement [**"else"** statement]

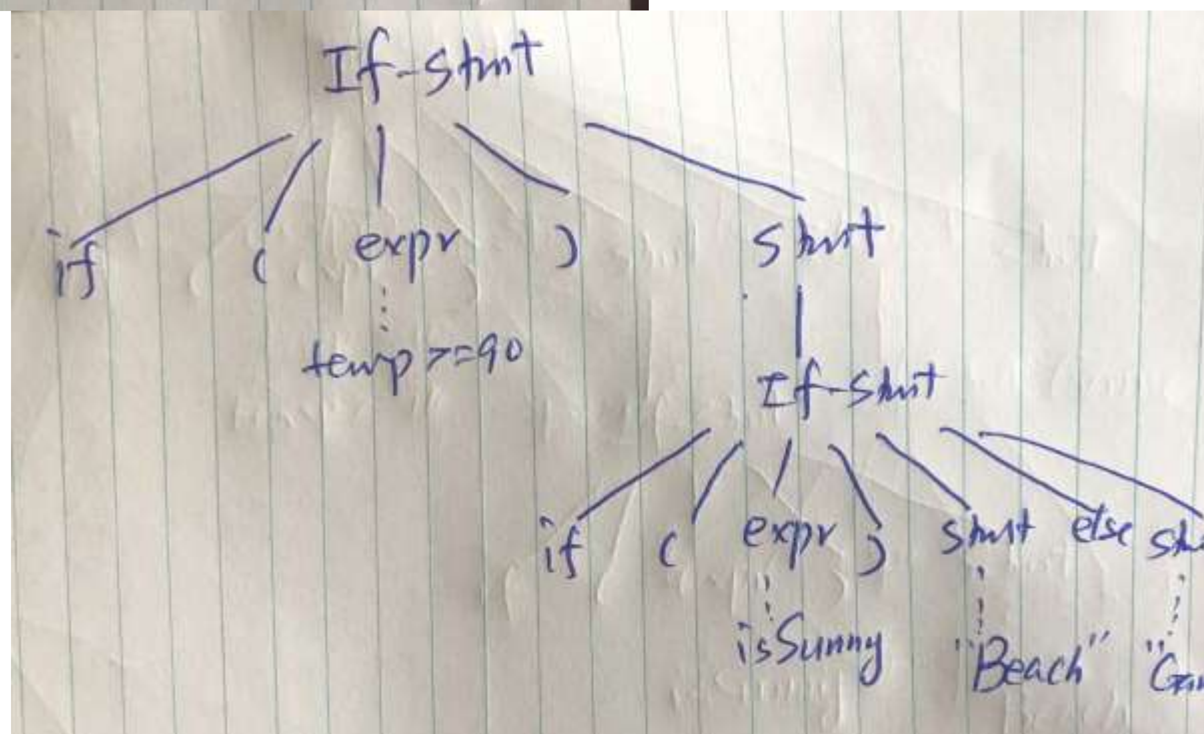
```
if (temperature >= 90) {
        if (isSunny)
                System.out.println("Beach");
}  else
        System.out.println("Game");
```

Today's temperature is 80 and isSunny is True, what do you do?

A. Beach B. Game C. Nothing D. Other

Today's temperature is 95 and isSunny is False, what do you do?


If-Stmt
if
(
expr
)
Stmt
else
Stmt
temp >= 90
"Game"
If-Stmt
if
(
expr
)
Stmt
isSunny
"Beach"


If-Stmt
if
(
expr
)
Stmt
temp >= 90
If-Stmt
if
(
expr
)
Stmt
else
isSunny
"Beach"

Strategies for resolving "dangling else"

- Adding additional keyword
  - Ruby, VB
  - Pseudo code for demonstrating the concept

```
if (temperature >= 90)

        if (isSunny)

                "Beach";
        end if;
else
        "Game";
end if;
```

```
if (temperature >= 90)

        if (isSunny)

                "Beach";
        else
                "Game";
        end if;
end if;
```

Strategies for resolving "dangling else"

- Using indentation
  - Python
  - Pseudo code for demonstrating the concept

```
if temperature >= 90:
        if isSunny:
                "Beach"
else:
        "Game";
```

```
if temperature >= 90:
        if isSunny:
                "Beach";
        else:
                "Game";
```

Strategies for resolving “dangling else”

- Requiring { } (block) for then & else branch
  - Go, Rust, Swift
  - Pseudo code for demonstrating the concept

```
if (temperature >= 90)   {

     if (isSunny) {
          “Beach”;
      }
     else {
               “Game”;
      }    //end of inside if
};
```

```
if (temperature >= 90) {

          if (isSunny) {
                    “Beach”;
           }
    } //end of inside if
else {
          “Game”;
};
```

Strategies for resolving "dangling else"

- Writing unambiguous grammar or additional rule
  - Kotlin, C#, JavaScript
  - Same as Java

  Syntax rule (e.g. Kotlin)

```
ifExpression (used by primaryExpression) :
      'if' '(' expression ')' (controlStructureBody | ';')
      | 'if' '(' expression ')' controlStructureBody? ';'? 'else' (controlStructureBody | ';')
;
```

  "else matches the closest if" or similar sentence

  --Did you find it in language documentation?

  --Some languages take this as de facto, not good.

  --If part (2) syntax rule is an ambiguous grammar, must add additional note/rule to resolve the ambiguity

  some may consider this addition as a semantic rule

Group Activity and Discussion

- Find a (formal/trustable) document that describe the syntax of your group's language in a EBNF or similar way (give a link to the document.)
- Show the syntax rule for the if statement in your language
- Rewrite the Java-like if statement (see next slide) in your group's language
- Check if your language's syntax for if statement is ambiguous or not (i.e. able to produce two parse trees for the given if statement or not.)
  - If ambiguous, discuss how the language resolves the ambiguity
  - If not, discuss how the Java-like ambiguity is avoided

Java-like if statement

if (temperature >= 90) if (isSunny) System.out.println("Beach");
else System.out.println("Game");

Summary

- Syntax diagram is an easy, graphical method to describe syntax of ”simple” programming languages
- BNF and context-free grammars are equivalent meta-languages
  - Well-suited for describing the syntax of programming languages
- EBNF (now often simply referred to as BNF) is the method now popularly used to describe the syntax of programming languages.

Part B: Describing Semantics

Topics

- Introduction
- Attribute grammars
  - Syntax + semantics
- Semantics descriptions
  - Operational semantics
  - Denotational semantics
  - Axiomatic semantics

Introduction

- **Semantics:** the meaning of the expressions, statements, and program units
- There is no single widely acceptable notation or formalism for describing semantics
- Several needs for a methodology and notation for semantics:
  - Programmers need to know what statements mean
  - Compiler writers must know exactly what language constructs do
  - Correctness proofs would be possible
  - Compiler generators would be possible
  - Designers could detect ambiguities and inconsistencies

Describing Semantics

- Attribute grammar
  - Syntax-directed translations
- Operational semantics
  - Informal way of describing the semantics
  - Popularly used
- Denotational semantics
  - Formal description of semantics
- Axiomatic Semantics
  - Based on formal logic

Attribute Grammars: Concept

- Semantic meanings (i.e. actions) associated with CFG rules
- Limited power
- Commonly used cases
  - Arithmetic expression evaluation
  - Type checking of expressions and statements
- CS4110: Compilers and Interpreters
  - Discusses details of “syntax-directed translation”
  - In this class, we will give an example to illustrate the idea behind.

Example: Attribute Grammar for (simple) Arithmetic Expressions

Syntax	Semantics
E -> E1 + T	E.v = E1.v + T.v
E -> T	E.v = T.v
T -> T1 * F	T.v = T1.v * F.v
T -> F	T.v = F.v
F -> num	F.v = num.lexval

Notes:

(1) For simplicity we use E, T, F instead of <Expr>, <Term>, <Factor>

(2) E, E1 are same symbols in syntax, but need to distinguish them semantically (e.g. 3+4+8, 4+8 syntactically both are expressions, semantically they're different, one means 15, the other 12.)

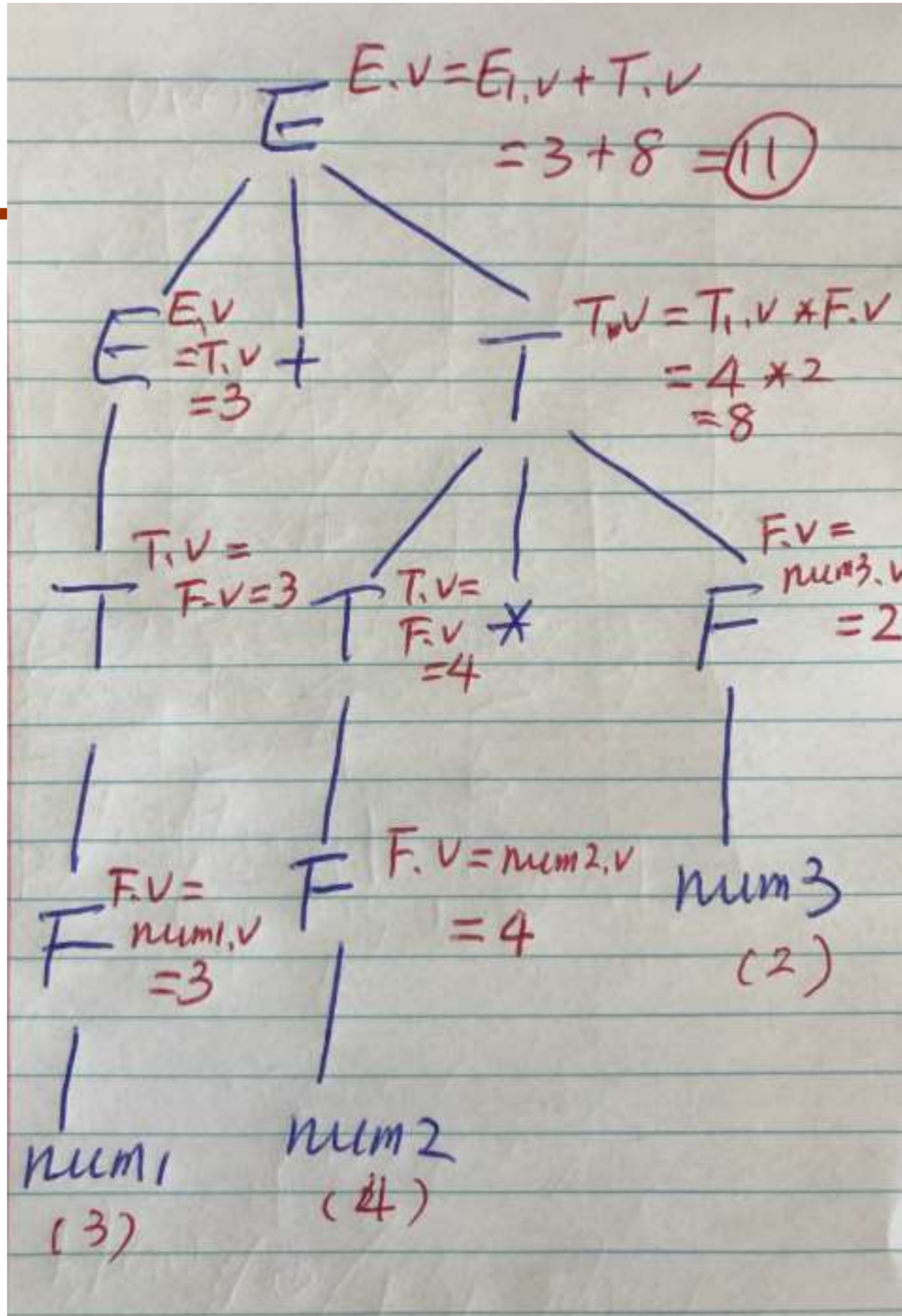
(3) E.v here v is the attribute for symbol E. num.lexval refers to the lexical value of a number, e.g. 3 and 4 syntactically both are num but their lexical values are different.

Attributed Grammar: Semantics Analysis

- Syntax Directed Translation
  - Evaluate semantic meaning/values along parse tree
- Example:
  - Expression: 3 + 4 * 2
  - After lexical analysis: num + num * num
    - For semantic analysis, we need to distinguish the symbols, so

      num1 + num2 * num3
    - Here we simply use num.v to refer to a number's lexical value

      num1.v = 3 num2.v = 4 num3.v = 2
  - Now we build a parse tree for num + num * num
    - And evaluate the semantic values along the parse tree



This illustrates that based on semantic rules defined in the attribute grammar, the meaning of 3+4*2 is 11.

Operational Semantics

- Operational Semantics
  - Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement
- Levels of uses
  - At the highest level, the interest is in the final result of the execution of a complete program, called *natural operational semantics.*
  - At the lowest level, operational semantics can be used to determine the precise meaning of a program through an examination of the complete sequence of state changes that occur when the program is executed, called *structural operational semantics.*

Basic Process

1. design an appropriate intermediate language
   - Here we use assembly like pseudo code as intermediate language, e.g.

     a = b + c; //corresponds to an add instruction

     if a < b goto L //corresponds to branch less than

     goto L //corresponds to jump instruction
2. Translate HL program to low-level program
   - See next slide for example
3. The meaning of low-level program (as executed on an ideal machine) is the meaning/semantics of the high-level program

Operational Semantics – Example

- Translate the following for-statement into assembly (pseudo-code)

HL code:

```
count = 0
while (++count < max)
        cost = cost +count * unit_price;
```

LL equivalence: (for illustration of basic idea, not optimized)

```
       count = 0
Loop:  count = count + 1
       if count >= max goto Exit
       temp = count * unit_price
       cost = cost + temp
       goto Loop
Exit:
```

Operational Semantics - Practice

- Translate the following for-statement into assembly (pseudo-code)

```
for (count = 0; count < max; count ++)
  cost = cost + count * unit_price;
```

(note: knowledge of CS2640 very helpful.)

Use case of operational semantics

- In teaching programming languages
  - Particularly for 1st programming language
  - E.g. in explaining the meaning of a C++ for loop

C++ statement	meaning
for (expr1; expr2; expr3) { …stmts … }	expr1 Loop: v= expr2 if v == 0 goto Out … stmts … expr3 goto Loop Out:

Operational Semantics: Practice

- Please define the meaning of a C++ if statement

```
if (a < b) {
        count--;
        sum += count * b;
}
else {
        a *= b;
        b /= count;
}
```

Denotational Semantics

- Originally developed by Scott and Strachey (1970)
- Based on recursive function theory
- Provides a rigorous way to think about programs
- The most abstract semantics description method
- The basic idea is to recursively map the syntactic units to semantic domain which mimics memory states
- Because of its complexity, of little use to language users
- Details omitted here

Axiomatic Semantics

- Based on formal logic (predicate calculus)
- Original purpose: formal program verification
- Axioms or inference rules are defined for each statement type in the language (to allow transformations of logic expressions into more formal logic expressions)
  - Recall inference rules from CS1300
- The logic expressions are called *assertions*
  - *Preconditions: prior to the assertion*
  - *Postconditions: after the assertion*
- The semantics of a program is a mapping from preconditions to postconditions
- Details omitted here

Summary

- An attribute grammar is a descriptive formalism that can describe both the syntax and the semantics of a language
  - Limited power in semantics description
- Three primary methods of semantics description
  - Operational, denotational, axiomatic
  - Operational semantics widely used to (informally) describe the meaning of program constructs

Learning Objectives

- After Lecture 3, you should be able to
  - Name the methods used to describe syntax and semantics
  - Explain the meaning of a BNF or EBNF rule
  - Write EBNF rules for simple language constructs
  - Discuss the ambiguity issues in language constructs
  - Translate simple Java-like statements into intermediate codes with equivalency in semantics