# Chapter 3

## Describing Syntax and Semantics

# Topics

- Introduction
- The General Problem of Describing Syntax
- Formal Methods of Describing Syntax
- Attribute Grammars
- Describing the Meanings of Programs:

    Dynamic Semantics

# Introduction

- Syntax: the *form* or *structure* of the expressions, statements, and program units.
- 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
      - how the expressions, statements, and program units of a language are formed, and also their intended effect when executed.
    - Programmers (the users of the language)
      - Refer to a language reference manual.

# Introduction (cont.)

- Semantics: the *meaning* of the expressions, statements, and program units
- E.g.) `while (Boolean_expr) statement`
- The semantics:

  when the current value of the Boolean expression is true, the embedded statement is executed.
  Then control implicitly returns to the Boolean expression to repeat the process.
  If the Boolean expression is false, control transfers to the statement following the **`while`** construct.

# Static Semantics: Motivation

- Nothing to do with the meaning of programs; it has to do with the *syntax* rather than semantics.
- Context-Free Grammars (CFGs)[slide-#13] in BNF cannot describe all of the syntaxes of programming languages.

  → needs for a more powerful mechanism
- Categories of constructs that are trouble:

  - Context-Free, but cumbersome:

    e.g.) type compatibility of operands in expressions

    - In Java: `integer type var = floating-point value` – not allowed.

    Otherwise, it needs more nonterminal symbols and rules

    → too large grammar.

  - Non-context-free:

    e.g.) variables must be declared before they are used.

# Static Semantics: Motivation (cont.)

- *Static Semantics Rules:*
  - ← Need of the categories of language rules to handle such problems.
  - – Many of them state their *type constraints* in a language.
  - – The analysis required to check these specifications can be done at *compile time*. → *Static* semantics.
- Need for a variety of more powerful mechanisms for that task.

→ Attribute grammars designed by Knuth (1968) to describe both the syntax and the static semantics of programs.

# Attribute Grammars (AG) - skip

- A formal approach both to describe and check the correctness of the static semantics rules of a program.
  - An extension to context-free grammar (CFG) to describe more of the *structure* of a PL than those with CFG.
- CFGs + new to carry some *semantic information on parse tree nodes.*
  - attributes - associated with grammar symbols, ≈ variables
  - attribute computation function (=semantic functions)
    - associated with grammar rules
    - It specifies how attribute values are computed.
  - predicate function
    - It states the static semantic rules of the language.
- Primary value of AGs:
  - Static semantics specification
  - used in Compiler design (static semantics checking)

# (Dynamic) Semantics

- Describing the semantics - the *meaning*s of the program
  - no single widely acceptable notation or formalism.
- 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 (without testing).
  - Compiler generators would be possible.
  - Language designers could *detect ambiguities and inconsistencies*.
- Scheme (FL) - one of only a few PLs whose definition includes a formal semantics description.
- Operational Semantics (skip)
- Denotational Semantics (skip)
- Axiomatic Semantics

# Axiomatic Semantics (AS)

- The most abstract way of the semantics specification.
  - Specify *what can be proven* about the program.
- Based on formal logic, *predicate calculus*.
- Original purpose: formal program *verification*
  - *Prove the correctness of the program/statement.*
- In a proof, each statement of a program is both *preceded* and *followed* by a *logical expression* that specifies *constraints* on program variables, which are used to specify the meaning of the statement: *precondition/postcondition*
- The meaning is defined by the statement's *effect* on assertions about the data affected by the statement.
- The *logical expressions* used in AS are called predicates or *assertions*.

# Axiomatic Semantics (cont.)

- *Precondition*:
  - an assertion *before* a statement
  - it states the relationships and constraints among variables that are *true* at that point *in execution*.
- *Postcondition*: an assertion *following* a statement
  - it states the relationships and constraints among variables that are *true* at that point *right after execution.*
- The *weakest precondition* is the *least restrictive precondition* that will guarantee the postcondition.
- Axioms or inference rules are defined for each statement type in the language (to allow transformations of logic expressions into more formal logic expressions).

# Axiomatic Semantics Form

- Pre-, post form: `{P} statement {Q}`
- `{P}` : precondition
- `{Q}` : postcondition
- An example

```
a = b + 1
{a > 1}
```

  - One possible precondition: `{b > 10}`
  - Weakest precondition: `{b > 0}`

```
{P}
statement
{Q}
```

$\{P\} \overset{?}{\Leftrightarrow}$ the program specification

Entire Program

{Q} = desired output

# Program Proof Process

- The *postcondition* for the *entire program* is the *desired result of the program*.
  - Work back through the program to the first statement.
  - If the precondition on the 1st statement is the same as the program specification, the program is correct.
- *Inference rule*:
  - a method of inferring the truth of one assertion (consequent) on the basis of the values of other assertions (antecedent).
  - $\dfrac{S_1, S_2, \ldots\ldots, S_n}{S}$ $\quad \dfrac{:\text{antecedent}}{:\text{consequent}}$ $\quad (\text{antecedent} \Rightarrow \text{consequent})$
  - If all the antecedents $S_i$ are true, the truth of S can be inferred.
  - *Axiom*: a logical statement that is assumed to be true; an inference rule without an antecedent.

# Axiomatic Semantics: Assignment

- An axiom for assignment statements

  (x = E): $\{Q_{x\rightarrow E}\}$ x = E {Q}

- Its weakest precondition P is defined by the axiom

  $P = Q_{x\rightarrow E}$

  P is computed as Q with all instances of x replaced by E.

- Example: `a = b/2 - 1 {a < 10}`

  the weakest precondition: `b/2 – 1 < 10` → `{b < 22}.`

- Example: `{x > 3} x = x - 3 {x > 0}`

  If the assignment axiom, when applied to the postcondition and the assignment statement, produces the given precondition, the theorem is proven.

  the precondition: `x – 3 > 0` → `{x > 3}`

  the given precondition `{x > 3}` = `{x > 3}`.

  So, the given logical statement is proven.

# Axiomatic Semantics: Assignment

- Example: `{x > 5} x = x - 3 {x > 0}`

  the precondition: `x – 3 > 0` → `{x > 3}`

  \- assertion produced by the axiom.

  the given precondition `{x > 5}` ≠ `{x > 3}`,

  but `{x > 5}` implies `{x > 3}`.

- The Rule of Consequence:

$$\frac{\{P\}S\{Q\},\ P' \Rightarrow P,\ Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

- Example:

$$\frac{\{x>3\}\ x=x-3\ \{x>0\},\ (x>5) \Rightarrow (x>3),\ (x>0) \Rightarrow (x>-1)}{\{x>5\}\ x=x-3\ \{x>-1\}}$$

P’ is stronger than (or equal to) P, i.e. precondition can be strengthened.

and Q’ is weaker than (or equal to) Q, i.e. postcondition can be weakened.

# Axiomatic Semantics: Assignment

- The Rule of Consequence:

$$\frac{\{P\}S\{Q\},\ P' \Longrightarrow P,\ Q \Longrightarrow Q\prime}{\{P'\}\ S\ \{Q'\}}$$

  – If the logical statement {P} S {Q} is true,
    the assertion P’ implies the assertion P, and
    the assertion Q implies the assertion Q’,
  then it can be inferred that {P’} S {Q’}.

  – A *postcondition* can always be *weakened* and a *precondition* can always be *strengthened*.

# Axiomatic Semantics: Sequences

- An inference rule for sequences of the form S1; S2

  {P1} S1 {P2}

  {P2} S2 {P3}

- The Rule of Consequence:

$$\frac{\{P1\}\ S1\ \{P2\},\ \ \{P2\}\ S3\ \{P3\},}{\{P1\}\ S1;\ \ S2\ \{P3\}}$$

# Axiomatic Semantics: Sequence

- An axiom for sequence of assignment statements
  (x1 = E1; x2 = E2 ) : {P1} x1 = E1; x2 = E2 {P3}
- Its weakest precondition P is defined by the axiom

$$\{P3_{x2 \to E2}\}\ x2 = E2\ \{P3\}$$

$$\{(P3_{x2 \to E2})_{x1 \to E1}\}\ x1 = E1\ \{P3_{x2 \to E2}\}$$

The weakest precondition for the sequence x1 = E1; x2 = E2 with postcondition P3 is $\{(P3_{x2 \to E2})_{x1 \to E1}\}$.

- Example:

```
y=3*x+1;    - S1
x=y+3;      - S2
{x < 10}
```

Precondition for S2: y+3< 10 → y < 7 – postcondition for S1

Precondition for S1: 3*x+1 < 7 → x < 2

Thus, {x < 2} is the precondition of both S1;S2.

# Axiomatic Semantics: Selection

- An inference rule for selection
  - {P} **if** B **then** S1 **else** S2 {Q}

$$\frac{\{B \text{ and } P\}\ S1\ \{Q\},\quad \{(\text{not } B) \text{ and } P\}\ S2\ \{Q\},}{\{P\}\ \textbf{if}\ \ B\ \textbf{then}\ S1\ \textbf{else}\ \ S2\ \{Q\}}$$

- Example:

  **if** x>0 **then** y=y-1 **else** y=y+1 **{y>0}**

  – Precondition of then clause: y-1> 0 → {y > 1} (stronger)

  – Precondition of else clause: y+1> 0 → {y > -1}

  – {y > -1} ⇒ {y > 1} because both condition must be true.

  – **P = {y > 1}**

# Axiomatic Semantics: Loops

- Find an assertion, called a *loop invariant*, to find the *weakest precondition*.
- An inference rule for logical *pretest* loops

{P} **while** B **do** S **end** {Q}

$$\frac{\{I \text{ and } B\}\ S\ \{I\}}{\{I\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{end}\ \{I \text{ and } (\text{not } B)\}}$$

where I is the loop invariant (the inductive hypothesis)

# Axiomatic Semantics: Axioms for a loop

- Characteristics of the loop invariant: I must meet the following conditions:

$$\frac{\{I \text{ and } B\}\ S\ \{I\}}{\{I\}\ \texttt{while}\ B\ \texttt{do}\ \ S\ \texttt{end}\ \{I \text{ and } (\text{not } B)\}}$$

  - $P \Rightarrow I$ -- the loop invariant must be true initially
  - {I} B {I} -- evaluation of the Boolean must not change the validity of I
  - {I and B} S {I} -- I is not changed by executing the body of the loop
  - {I and (not B)} $\Rightarrow$ Q -- if I is true and B is false, Q is implied
  - The loop terminates -- If Q is true immediately after loop exit, then a precondition P for the loop is one that guarantees Q at loop exit and also guarantees that the loop terminates.

# Axiomatic Semantics: loops (cont.)

$$\frac{\{I \text{ and } B\}\ S\ \{I\}}{\{I\}\ \textbf{while}\ B\ \textbf{do}\ \ S\ \textbf{end}\ \{I \text{ and } (\text{not } B)\}}$$

- <u>Example</u>: (pg. 150 – 151)

**while** $y \neq x$ **do** y=y+1 **end** {y=x}

I : {y $\leq$ x} for P    {**y** $\leq$ **x** and $y \neq x$} y=y+1 **{y<=x}**

i.e. {I and B} S {I}

1) Prove P $\Rightarrow$ I.

Find a loop invariant I for a precondition and a postcondition.

- Before iteration ($0^{th}$) : the weakest precondition is {y=x}$\equiv${y=x}
- $1^{st}$ iteration: y=y+1 {y=x} $\Rightarrow$ {y+1=x} $\equiv$ {y=x-1}
- $2^{nd}$ iteration: y=y+1 {y=x-1} $\Rightarrow$ {y+1=x} $\equiv$ {y=x-2}
- $3^{rd}$ iteration: y=y+1 {y=x-2} $\Rightarrow$ {y+1=x} $\equiv$ {y=x-3}

….

$\Rightarrow$ {y < x} $\Rightarrow$ {y < x} and {y=x} $\Rightarrow$ **{y $\leq$ x}**

**I : {y $\leq$ x}** for P    i.e. P = I

# Axiomatic Semantics: loops (cont.)

- <u>Example</u> (cont.):

  **while** $y \neq x$ **do** $y=y+1$ **end** $\{y=x\}$

  2) Prove {I and B} S {I},

  i.e. $\{\mathbf{y} \leq \mathbf{x}$ and $y \neq x\}$ $y=y+1$ $\{\mathbf{y} \leq \mathbf{x}\}$.

  $y=y+1$ $\{y \leq x\}$

  $\{y+1 \leq x\} \equiv \{y < x\}$

  So, $\{\mathbf{y} \leq \mathbf{x}$ and $y \neq x\}$ is asserted from P as $P \Rightarrow I$ (because $P = I$)

  thus, $\{\mathbf{y} \leq \mathbf{x}$ and $y \neq x\}$ $y=y+1$ $\{\mathbf{y} \leq \mathbf{x}\}$ is proven.

  3) By inference rule, prove {I and (not B)} $\Rightarrow$ Q.

  $\{\mathbf{y} \leq \mathbf{x}$ and $\text{not}(y \neq x)\} \Rightarrow \{\mathbf{y} \leq \mathbf{x}$ and $y=x\} \Rightarrow \{y = x\}$

  $\{y = x\} \Rightarrow \{\mathbf{y = x}\}$ **-- postcondition**

  So, {I and (not B)} $\Rightarrow$ Q is proven.

# Axiomatic Semantics: loops (cont.)

- Example (cont.):

  **while** y≠x **do** y=y+1 **end** {y=x}

  4) Prove Loop termination in

  {y ≤ x} **while** y≠x **do** y=y+1 **end** {y=x}.

  The precondition {y ≤ x} guarantees that initially y ≯ x.

  The loop body increments y with each iteration, until y=x.

  Regardless of the initial y (≤ x), y will eventually become equal to x (y=x).

  So the loop will terminate.

  Because our choice of I satisfies all 4 criteria (slide#20-#22), it is a satisfactory loop invariant and loop precondition.

  Thus, the statement {y ≤ x} **while** y≠x **do** y=y+1 **end** {y=x}

  with P and Q is a correct statement.

# Axiomatic Semantics: loop (cont.)

## Example 2: Find a loop invariant

**while** `s>1` **do** `s=s/2` **end** {s=1}

Find a loop invariant I for a precondition and a postcondition.

- Before iteration: the weakest precondition is {s=1}≡{s=1}
- $1^{st}$ iteration: s=s/2 {s=1} ⇒ {s/2=1} ≡ {s=2}
- $2^{nd}$ iteration: y=y+1 {s=2} ⇒ {s/2=1} ≡ {s=4}
- $3^{rd}$ iteration: y=y+1 {s=4} ⇒ {s/2=1} ≡ {s=8}

…. ⇒ $\{s = 2^k\}$ for $k \geq 0$

**Invariant** : $\{s = 2^k\}$ for P – I can serve as P,

but not the weakest precondition

Consider using {s > 1} for the weakest P in {I and B} S {I}.

{s > 1}⇒ $s=2^k$ ; {$s=2^k$ and s>1} s=s/2 {$s=2^k$}

# Loop Invariant (I)

- The loop invariant I is

  a *weakened* version of the loop *postcondition*,

  and a precondition for the loop.
- I must be *weak enough* to be satisfied *prior to* the beginning of the loop,
- but when I is combined with the loop exit condition,

  it must be *strong enough* to force the truth of the

  postcondition.
- The axiomatic description of the loop is called
  - total correctness: if loop termination can be shown,
  - partial correctness: if other conditions can be met but termination is not guaranteed.

# Two Example: Program Proofs (pg. 152 – 155)

```
{x = A AND y = B}
t = x;
x = y;
y = t;
{x = B AND y = A}
```

1. $\{P3_{x3\to E3}\}$ x2 = E2 {P3}: $\{P3_{y\to t}\}$ **y=t** {x = B AND y = A}

   → $\{P3_{y\to t}\}$ = {x = B AND t = A}

2. $\{(P3_{x3\to E3})_{x2\to E2}\}$ **x = y** $\{P3_{x3\to E3}\}$:

   $\{(y=B \text{ AND } t=A)_{x\to y}\}$ **x = y** {x=B AND t=A}

   → $\{(y=B \text{ AND } t=A)_{x\to y}\}$ = {y=B AND t=A}

3. $\{((P3_{x3\to E3})_{x2\to E2})_{x1\to E1}\}$ **t = x** $\{P3_{x1\to E1}\}$:

   $\{(y=B \text{ AND } x=A)_{t\to x}\}$ **t = x** {y=B AND t=A}

   → $\{(y=B \text{ AND } x=A)_{t\to x}\}$ = {y=B AND x=A}

   ={x = A AND y = B}

# Two Example: Program Proofs (pg. 152 – 155)

```
{n >= 0}
count = n;
fact = 1;
while count ≠ 0 do
    fact = fact * count;  (L1)
    count = count - 1;    (L2)
end
{fact = n!}
```

1. Loop invariant I = ( fact=$n$*($n$-1)* ....(count+2)*(count+1)) AND (count ≥ 0)
2. {I and B} S {I}:

   { fact=1*n*(n-1)* ....(count+2)*(count+1)) AND (count ≥ 0) AND (count ≠0)}

→ { fact=n*(n-1)* ....(count+2)*(count+1)) AND (count > 0)

3. {P} L2 {I}: {P} count = count -1 {I}

→ {P} = {fact=1*n*(n-1)* ....(count+2)*(count+1)*count) AND (count ≥ 1)} = {Q} of (L1)

So, {P}={fact= 1*n*(n-1)* ....(count+2)*(count+1) AND (count ≥ 1) of L1.

Thus, {I and B} implies P; therefore, {I and B} S {I} is true.

# Two Example: Program Proofs (pg. 152 – 155)

```
{n >= 0}
count = n;                (S1)
fact = 1;                 (S2)
while count <> 0 do
   fact = fact * count;  (L1)
   count = count – 1;    (L2)
end
{fact = n!}
```

4. {I and (not B)} $\Rightarrow$ Q:

{ fact=n*(n-1)* ….(count+2)*(count+1)) AND (count $\geq$ 0) AND (count =0)}

$\Rightarrow$ fact=n*(n-1)* …. 2 * 1 = n! True.

{P} of while loop: from {Q} of S2

{ *fact*=n*(n-1)* ….(count+2)*(count+1)) AND (count >=0)

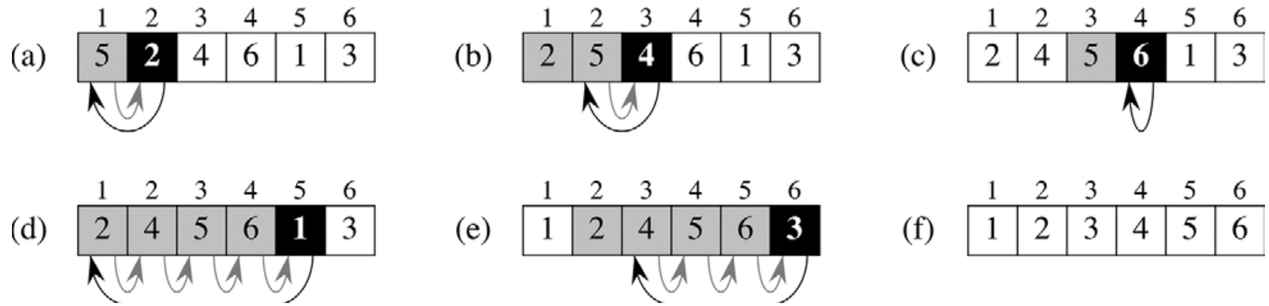
→ P : {*1* = n*(n-1)* ….(count+2)*(count+1)) AND (count $\geq$ 0) } → {Q} of S1

{P} of S1: { (1 = n*(n-1)* …2*1 AND (n $\geq$ 0) }

The (1 = n*(n-1)* …2*1 is true (because 1 = 1) and the (n $\geq$ 0) is exactly the precondition of the whole code segment, {*n* $\geq$ 0}.

Therefore, the program has been proven to be correct

# Example (CSci 242): Loop Invariant (I) and correctness of insertion-sort algorithm




INSERTION-SORT$(A, n)$

**for** $j = 2$ **to** $n$
    $key = A[j]$
    // Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.
    $i = j - 1$
    **while** $i > 0$ and $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$

# Example (CSci 242): Loop Invariant (I) and correctness of insertion-sort algorithm (cont.)

Use a *Loop Invariant* (LI): a statement that always holds in the loop.

***Loop Invariant*:** At the start of each iteration of the "outer" **for** loop (indexed by $j$), the subarray $A[1 \,..\, j\text{-}1]$ consists of the elements originally in $A[1 \,..\, j\text{-}1]$ but in sorted order.

Show three things about Loop Invariant to prove the correctness of algorithm :

**Initialization:**

LI is true prior to the first iteration of the loop.

**Maintenance:**

If LI is true before an iteration of the loop,
then LI remains true before the next iteration.

**Termination:**

When the loop terminates, the invariant(LI) remains true;
thus,
LI gives us a useful property that helps show that the algorithm is correct.

# Example (CSci 242): Loop Invariant (I) and correctness of insertion-sort algorithm (cont.)

Proof: Prove the correctness of Insertion algorithm using the Loop Invariant(LI).

**Initialization:** Show that the LI holds before the 1st iteration, when $j=2$.

When $j=2$, the subarray $A[1 .. j\text{-}1]$ (i.e. $A[1,1]$) consists of just the single element $A[1]$, which is the original element in $A[1]$. Moreover, the subarray $A[1,1]$ (=$A[1]$) is sorted.

Thus, the LI holds prior to the 1st iteration of the loop.

**Maintenance:** Show that each iteration maintains the LI.

The body of for loop works by moving $A[j\text{-}1]$,$A[j\text{-}2]$,$A[j\text{-}3]$, etc. by one position to the right until it finds the proper position for $A[j]^{\text{line-4-7}}$, at which point it inserts the value of $A[j]^{\text{line-8}}$.

The subarray $A[1..j]$ then consists of the elements in $A[1..j]$ in sorted order.

Then, Incrementing $j$ for the next iteration of the for loop preserves the LI.

**Termination:** Show what happens when the loop terminates: LI holds.

The condition of for loop termination is that $j > n$.

Because each loop iteration increases $j$ by 1, we must have $j=n+1$ at the termination.

Substituting $n+1$ for $j$ in the wording of LI, we have that the subarray $A[1..n]$ consists of the elements in sorted order. Since the subarray A[1..n] is the entire array, we conclude that the entire array A[1..n] is sorted. *Hence, the insertion algorithm is correct.*

# Evaluation of Axiomatic Semantics

- Developing axioms or inference rules for all of the statements in a language is difficult.
- It is a good tool for correctness proofs, and an excellent framework for reasoning about programs, but it is not as useful for language users and compiler writers.
- Its usefulness in describing the meaning of a programming language is limited for language users or compiler writers.

# Summary

- The grammar of programming language is Context-Free Grammar (CFG).
- BNF is a meta-language or a form that is used to express a CFG.
  - Well-suited for describing the syntax of programming languages.
- An attribute grammar is a descriptive formalism that can describe both the syntax and the semantics of a language.
- Three primary methods of semantics description
  - Operation, axiomatic, denotational