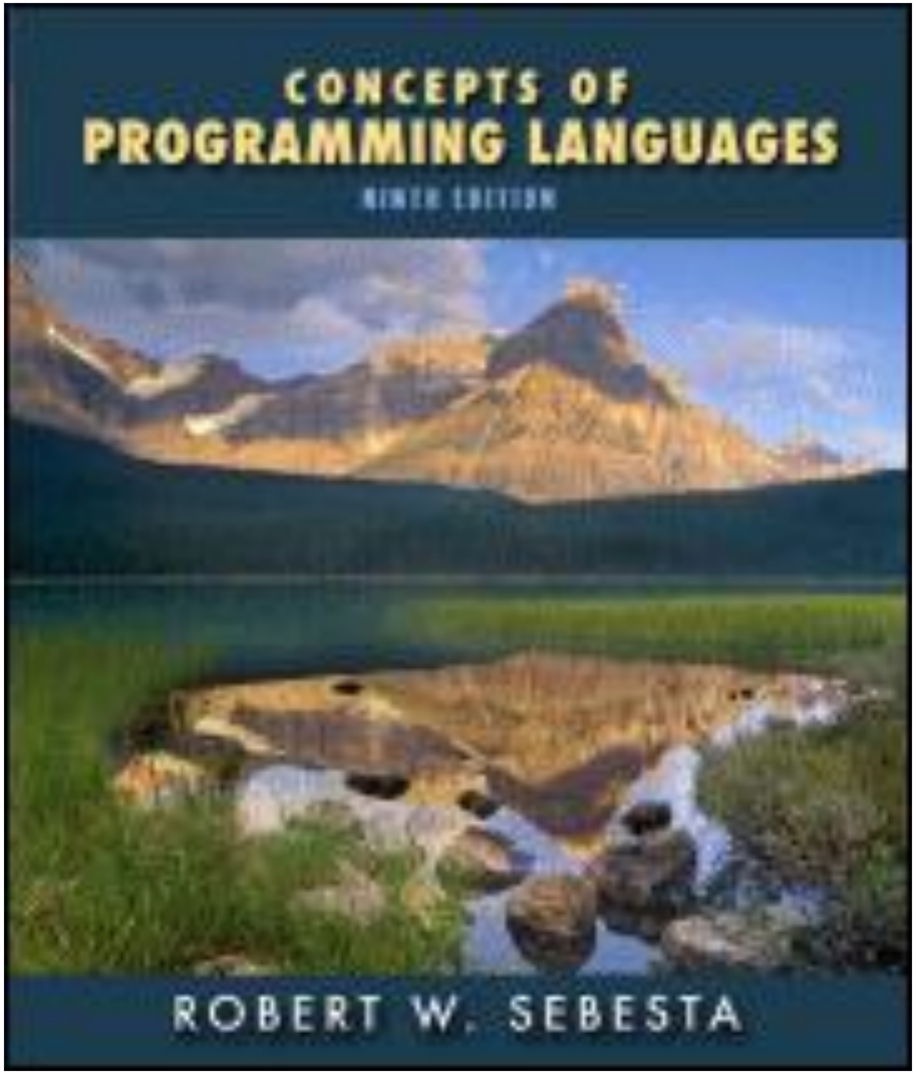
Chapter 3

Describing Syntax and Semantics



Ch03 – Describing Syntax and Semantics

3.1 Introduction

3.2 The General Problem of Describing Syntax

3.3 Formal Methods of Describing Syntax

3.4 Attribute Grammars

3.5 Describing the Meanings of Programs: Dynamic Semantics

3.1 Introduction

- Syntax and Semantics
  - Syntax

    The form or structure of expressions, statements, and program units.
  - Semantics

    The meaning of 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)

3.2 The General Problem of Describing Syntax

Terminology

- Formal languages

Σ	Alphabet
Σ^*	The set of all strings over the alphabet Σ
$L \subset \Sigma^*$	language
$x \in L$	sentence

- Example

Σ = All characters allowed in C++

C++ $\subset \Sigma^*$ is a language

int main() {} $\in$ C++ and is a sentence

int main() }{ $\notin$ C++ and isn't a sentence.

3.2 The General Problem of Describing Syntax

- Example

  $\Sigma = \{0, 1\}$

  L_1 = the language of all binary strings beginning with 101

  $= \{ 101, 1010, 1011, 10100, 10101, 10110, 10111, \ldots \}$

  L_2 = the language of all binary strings ending with 101

  $L_3 = L_1 \cap L_2$

  $L_4 = L_1 \cup L_2$

  $L_5 = \Sigma^* - L_1$

  $L_6 = L_1L_2 = \{ xy \mid x \in L_1, y \in L_2\}$

  Formal languages – "Formal" means "Mathematical"

  e.g. Is a class of languages closed under intersection, union, complement, concatenation, etc?

3.2 The General Problem of Describing Syntax

- Lexeme and token
  - A lexeme is the lowest level syntactic unit of a language.
  - A token is a category of lexemes.
Token	Lexeme
int_literal	25, 77
identifier	sum, begin
- Recognizer and Generator
  - A recognizer decides whether an input string belongs to the language, e.g. syntax analysis part of a compiler
  - A generator generates sentences of a language, e.g. grammar

3.3 Formal Methods of Describing Syntax

- Grammar G = <Σ,N,S,P>

Σ	A set of terminals	
N	A set of nonterminals	
$S \in N$	Start symbol	
P	A set of production rules or rewriting rules	
	$\alpha \rightarrow \beta$	$\alpha \in (\Sigma \cup N)^* N (\Sigma \cup N)^*$
		$\beta \in (\Sigma \cup N)^*$

- Languages generated by grammars

$\Rightarrow$	one-step derivation
$\Rightarrow^*$	zero or more step derivation

$L(G) = \{ \omega \mid \omega \in \Sigma^*, S \Rightarrow^* \omega \}$ = the language generated by G

3.3 Formal Methods of Describing Syntax

- Example

abbreviating $S \rightarrow 0S$

$S \rightarrow 1S$

$S \rightarrow 101$

  - Grammar G_1

    $S \rightarrow 0S \mid 1S \mid 101$

    $L(G_1)$ = All binary strings ending in 101

    e.g. $S \Rightarrow 0S \Rightarrow 01S \Rightarrow 01101$, i.e. $S \Rightarrow^* 01101$

  - Grammar G_2

    $S \rightarrow A101$

    $A \rightarrow A0 \mid A1 \mid \varepsilon$

    $L(G_2) = L(G_1)$

    e.g. $S \Rightarrow A101 \Rightarrow A1101 \Rightarrow A01101 \Rightarrow 01101$

    i.e. $S \Rightarrow^* 01101$

3.3 Formal Methods of Describing Syntax

- Regular grammar – left-linear or right-linear
  - Left-linear grammar $A \rightarrow \omega \mid B\omega \quad A, B \in N, \omega \in \Sigma^*$
  - Right-linear grammar $A \rightarrow \omega \mid \omega B \quad A, B \in N, \omega \in \Sigma^*$
- Regular language
  - Languages that can be generated by regular grammars
- Example
  - Non-regular Grammar G_3

    $S \rightarrow A101$

    $A \rightarrow 0A \mid 1A \mid \varepsilon$

    $L(G_3) = L(G_2) = L(G_1)$ is a regular language, since G_1 and G_2 are regular.

3.3 Formal Methods of Describing Syntax

- Lexical syntax

  The syntax of tokens. e.g. identifiers, constants, keywords, can be described by regular grammars.

- Example

  C/C++ identifiers

  $S \rightarrow aA \mid \text{'A'}A \mid \_A$

  $A \rightarrow aA \mid \text{'A'}A \mid \_A \mid 0A \mid \varepsilon$

  or

  $S \rightarrow a \mid \text{'A'} \mid \_ \mid Sa \mid S\text{'A'} \mid S\_ \mid S0$

3.3 Formal Methods of Describing Syntax

- Context-free grammar (CFG)

  $A \to \alpha \qquad A \in N, \alpha \in (\Sigma \cup N)^*$
- A regular language is also a CFL.
- Example

  The language of all nested balanced parentheses

  $S \to (S) \mid \varepsilon$

  This is a CFL, but not a regular language.

  It follows that programming languages are not regular.
- Phrase structure syntax

  The syntax of expressions, statements, program units, etc. can be described by CFGs.

3.3 Formal Methods of Describing Syntax

- Context-sensitive grammar (CSG)

  $\alpha \rightarrow \beta \qquad \alpha \in (\Sigma \cup N)^* N (\Sigma \cup N)^*,\ \beta \in (\Sigma \cup N)^*,\ |\alpha| \le |\beta|$

- A CFL without the empty string is also a CSL.
- Example

  L = { ωcω | ω is a string of a's and b's }

  $S \rightarrow AaS \mid BbS \mid c$

  $Aa \rightarrow aA \qquad Ba \rightarrow aB$

  $Ab \rightarrow bA \qquad Bb \rightarrow bB$

  $Ac \rightarrow ca \qquad Bc \rightarrow cb$

  $$
  \begin{aligned}
  S &\Rightarrow^* AaBbAaS \\
  &\Rightarrow AaBbAac \\
  &\Rightarrow^* abaABAc \\
  &\Rightarrow abaABca \\
  &\Rightarrow abaAcba \\
  &\Rightarrow abacaba
  \end{aligned}
  $$

  This is a CSL, but not a CFL.

  This implies that programming languages are not CFL's.

3.3 Formal Methods of Describing Syntax

- Programming languages are CSL's
- Context-sensitive features of programming languages
  - Identifiers must be declared before use.
  - An identifier can't be declared twice in a block.
  - A two-dimensional array cannot be accessed with three indices.
  - The number/order/type of actual parameters must agree with that of formal parameters.
  - And so on
  - E.g.

int x;	int a[2][3];	void p() {}	int p;
⋮	⋮	⋮	⋮
int x;	a[0][1][2]	p(2,3);	*p;

3.3 Formal Methods of Describing Syntax

- BNF (Backus-Naur Form, Backus Normal Form)
  - Invented by John Backus to describe Algol 58
  - BNF = CFG
  - BNF and grammars are *metalanguages* used to describe another language.
  - Non-terminals: syntactic categorires
  - Terminals: lexemes and tokens
  - Rules

    <if_stmt> → **if** <logic_expr> **then** <stmt>

    <ident_list> → identifier | identifier, <ident_list>

    or, ::=

3.3 Formal Methods of Describing Syntax

- Parse tree
  - A hierarchical representation of derivations
  - Example 3.2

A Grammar for Simple Assignment Statements

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
  | <id> * <expr>
  | (<expr>)
  | <id>

3.3 Formal Methods of Describing Syntax

- Derivations of A = B*(A+C)

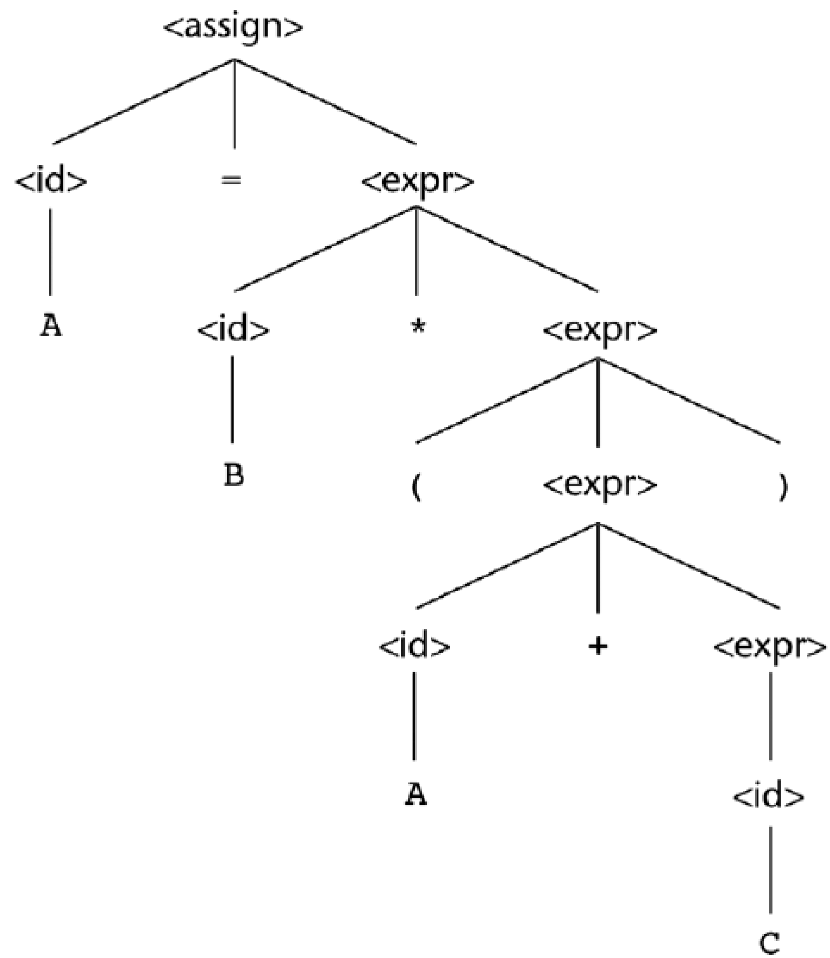
Leftmost derivation	Rightmost derivation
<assign>	<assign>
⇒ <id> = <expr>	⇒ <id> = <expr>
⇒ A = <expr>	⇒ <id> = <id>*<expr>
⇒ A = <id>*<expr>	⇒ <id> = <id>*(<expr>)
⇒ A = B*<expr>	⇒ <id> = <id>*(<id>+<expr>)
$\Rightarrow^{*}$ A = B*(A+C)	$\Rightarrow^{*}$ A = B*(A+C)

Arbitrary-order derivation

All these derivations correspond to a single parse tree.

3.3 Formal Methods of Describing Syntax

- Figure 3.1: Parse tree of A = B*(A+C)



3.3 Formal Methods of Describing Syntax

- Ambiguous grammar

  A grammar is *ambiguous* if it generates a sentential form that has two or more distinct parse trees

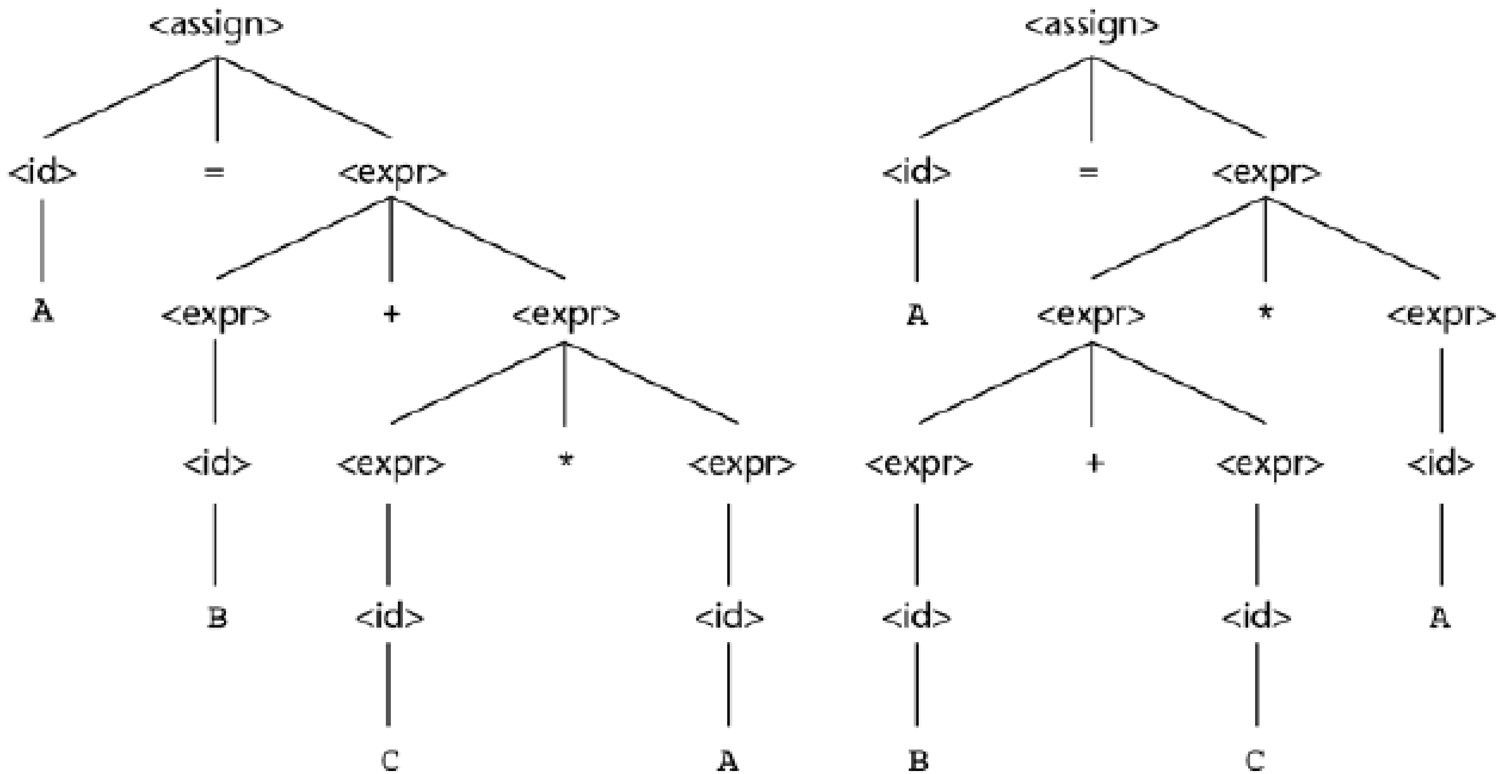
- An ambiguous grammar for arithmetic expressions
  - Example 3.3

An Ambiguous Grammar for Simple Assignment Statements

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
  | <expr> * <expr>
  | (<expr>)
  | <id>

3.3 Formal Methods of Describing Syntax

- Figure 3.2: Two parse trees for A = B+C*A



- Also, there are two parse trees for A = B+C+A
- Parse trees determines the semantics of expressions.

3.3 Formal Methods of Describing Syntax

- An unambiguous grammar for arithmetic expressions
  - Key points

    Operators generated earlier are evaluated later.

    Operators generated later are evaluated earlier.
  - Precedence

    Each precedence level is handled by a nonterminal.

Operator	Precedence	Generated order	Nonterminal
+	low	early	<expr>
*	↓	↓	<term>
()	high	late	<factor>

3.3 Formal Methods of Describing Syntax

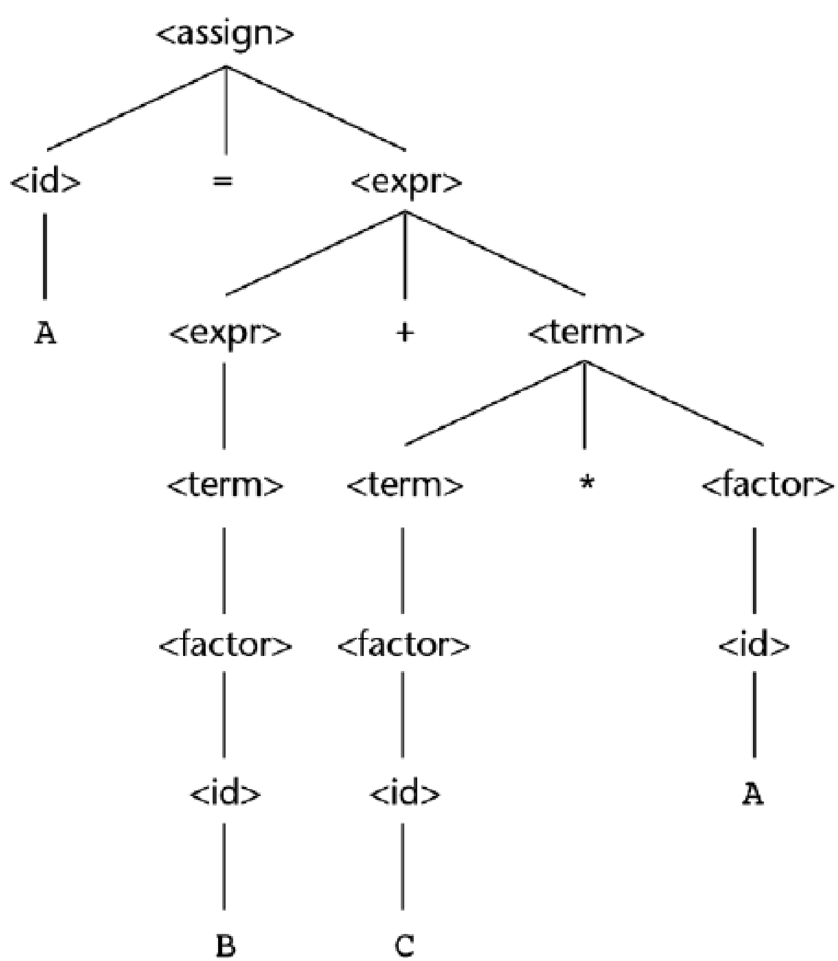
- Example 3.4

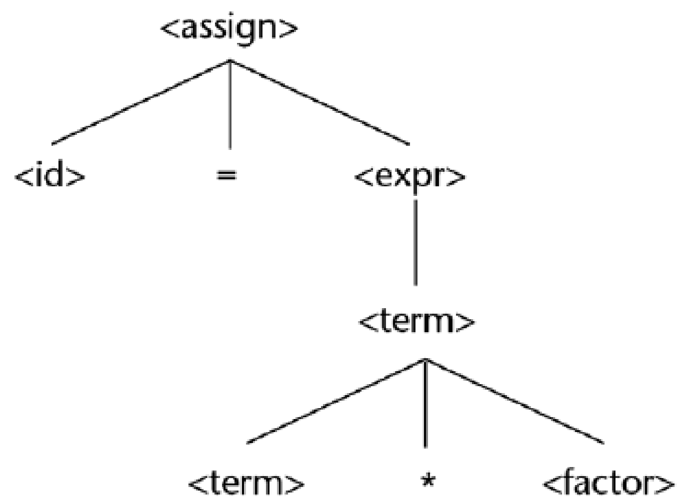
An Unambiguous Grammar for Expressions

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
        | <term>
<term> → <term> * <factor>
        | <factor>
<factor> → (<expr>)
        | <id>

3.3 Formal Methods of Describing Syntax

- Figure 3.3: A single parse tree for A = B+C*A





Can produce (B+C)

But not B+C

3.3 Formal Methods of Describing Syntax

- Associativity

  <expr> → <expr> + <expr>

  Double recursion makes the grammar ambiguous.

  <expr> → <expr> + <term>

  Left recursion specifies left associativity.

  <expr> → <term> + <expr>

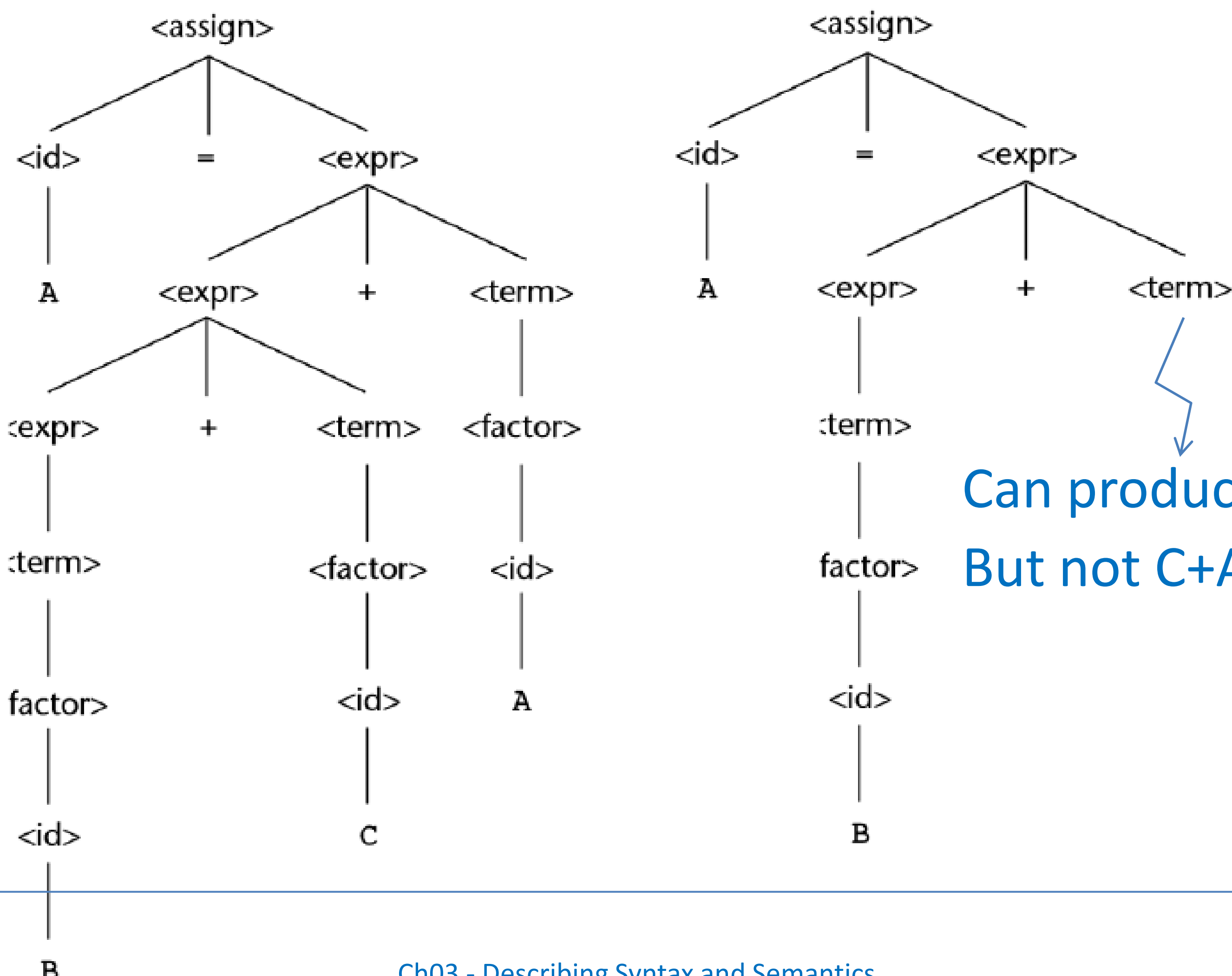
  Right recursion specifies right associativity.

  <expr> → <term> + <term>

  No recursion specifies non-associativity, e.g. B+C+A is illegal.

3.3 Formal Methods of Describing Syntax

- Figure 3.4: A single parse tree for A = B+C+A



Can produce (C+A)

But not C+A

3.3 Formal Methods of Describing Syntax

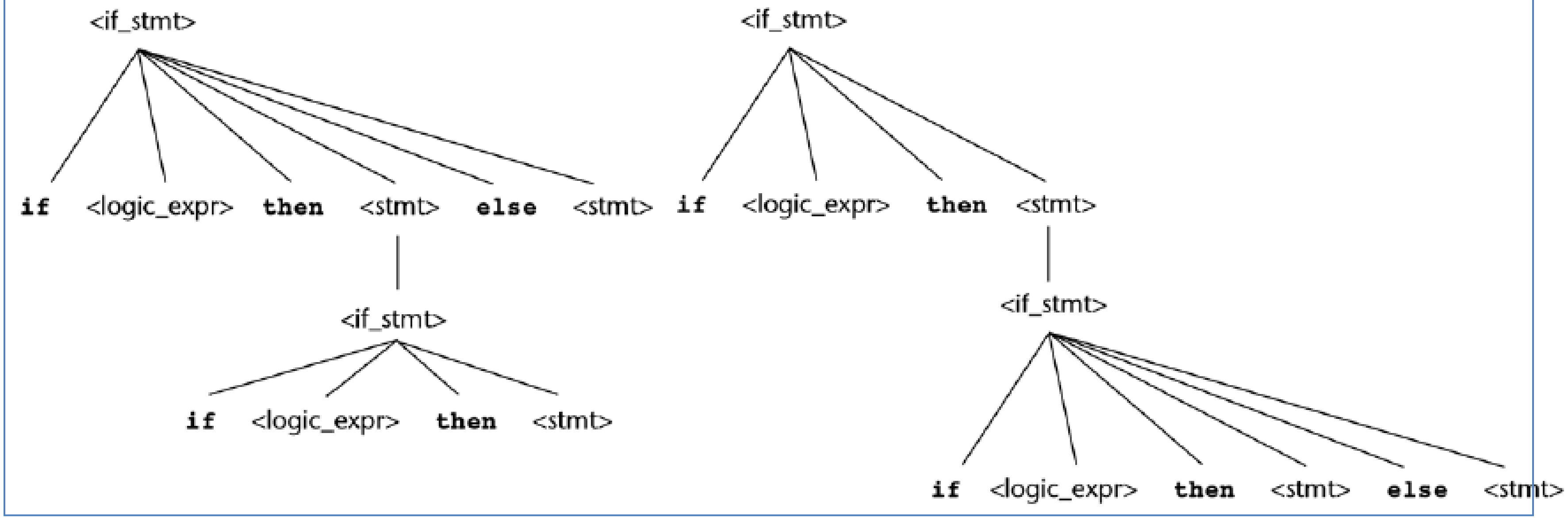
- An ambiguous grammar for if statements

  <stmt> → <if_stmt> | *other non-*if *statement*

  <if_stmt> → if <logic_exp> then <stmt>

  | if <logic_exp> then <stmt> else <stmt>

  Figure 3.5: Dangling else



3.3 Formal Methods of Describing Syntax

Unambiguous grammars for if statements

- Nearest unmatched approach

  An else is matched with the nearest previous unmatched then.

  <stmt> → <matched> | <unmatched>

  <matched> → *other non-*if *statement*

  | if <logic_exp> then <matched> else <matched>

  <unmatched> → if <logic_exp> then <stmt>

  | if <logic_exp> then <matched> else <unmatched>

  With this grammar,

  if <logic_exp> then if <logic_exp> then <stmt> else <stmt>

  has only one parse tree.

3.3 Formal Methods of Describing Syntax

- Terminating keyword approach

  Each if statement is terminated with an "endif".

  <stmt> → <if_stmt> | *other non*-if *statement*

  <if_stmt> → if <logic_exp> then <stmt> endif

  | if <logic_exp> then <stmt> else <stmt> endif

  - if <logic_exp> then

    if <logic_exp> then <stmt> else <stmt> endif

    endif
  - if <logic_exp> then

    if <logic_exp> then <stmt> endif

    else <stmt> endif

3.3 Formal Methods of Describing Syntax

- Drawback of the preceding IF statement

  if <logic_exp> then <stmt>

  else if <logic_exp> then <stmt>

  else if <logic_exp> then <stmt>

  else <stmt>

  endif endif endif ← too many endif's

- Ada's IF statement

  <if_stmt> → if <logic_exp> then <stmt>

  {elsif <logic_exp> then <stmt>} ← Rule A

  [else <stmt>] ← Rule B

  end if

3.3 Formal Methods of Describing Syntax

- Why the special keyword "elsif" in Ada?

Alternative: "elseif" (Hard to read)

if <logic_exp> then <stmt>
elseif <logic_exp> then <stmt>
else <stmt> endif

if <logic_exp> then <stmt>
else if <logic_exp> then <stmt>
else <stmt> endif endif

Alternative: "else if" (Hard to compile)

if <logic_exp> then <stmt>
else if <logic_exp> then <stmt>
else <stmt> endif

Rule A

if <logic_exp> then <stmt>
else if <logic_exp> then <stmt>
else <stmt> endif endif

Rule B

3.3 Formal Methods of Describing Syntax

- Fortran has no special keyword
  - It uses "elseif" or equivalently "else if"
  - "elseif" (or "else if") in a line is different from "else; if" in a line or "else" and "if" in two separated lines.

```
if <logic_exp> then
    <stmt>
else if <logic_exp> then
    <stmt>
else
    <stmt>
endif
```

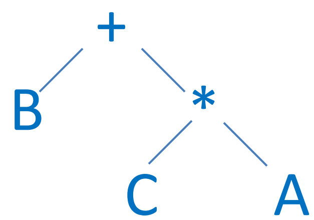
```
if <logic_exp> then
    <stmt>
else
    if <logic_exp> then
        <stmt>
    else
        <stmt>
    endif
endif
```

3.3 Formal Methods of Describing Syntax

- Concrete syntax
  - Concrete syntax concerns how sentences are actually written.
  - Concrete syntax trees retain all the information (e.g. the rules applied) used in parsing sentences.
  - Concrete syntax trees are too complex for semantic analysis and code generation
- Abstract syntax
  - Abstract syntax concerns only the structural properties of sentences.
  - Abstract syntax trees capture the structural properties of sentences in a simpler form.

3.3 Formal Methods of Describing Syntax

- Compilers usually generate abstract syntax trees.



- The semantics of programming languages is usually formulated with abstract syntax.
- An abstract syntax for expressions

  <expr> ::= <expr> + <expr> | <expr> * <expr> | <id>

  <id> ::= A | B | C

  The abstract syntax contains no parentheses and has no ambiguity problem.

3.3 Formal Methods of Describing Syntax

- EBNF (Extended BNF)
  - [] optional
    (..|..) selection
    {} repetition
  - BNF
    <expr> → <expr> + <term> | <expr> - <term> | <term>
    EBNF
    <expr> → [<expr> (+|-)] <term>
    <expr> → <term> { (+|-) <term> }
    <expr> → { <term> (+|-) } <term>
    The last two rules don’t express associativity.

3.3 Formal Methods of Describing Syntax

- EBNF is equivalent to BNF

$$\alpha \to \beta\ (\lambda \mid \pi)\ \theta \quad \equiv \quad \alpha \to \beta\ \lambda\ \theta \mid \beta\ \pi\ \theta$$

$$\alpha \to \beta\ [\ \lambda\ ]\ \theta \quad \equiv \quad \alpha \to \beta\ \theta \mid \beta\ \lambda\ \theta$$

$$\alpha \to \beta\ \{\ \lambda\ \}\ \theta \quad \equiv \quad \alpha \to \beta\ \mu\ \theta$$

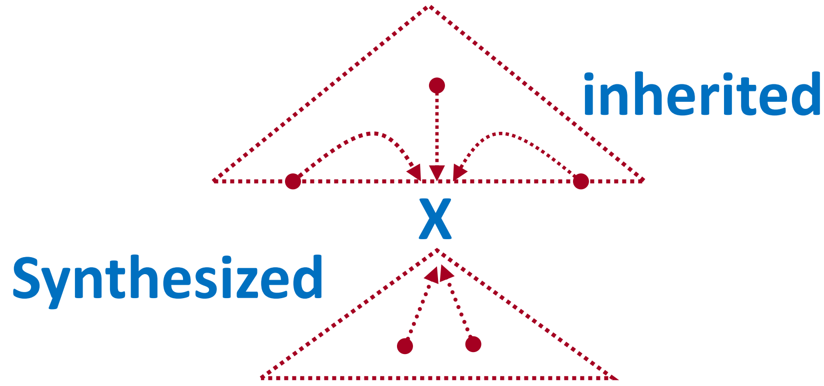
$$\mu \to \lambda\ \mu \mid \varepsilon$$

3.4 Attribute Grammars

- Static semantics
  - This deals with context-sensitive features and type constraints of programming languages.
  - It has nothing to do with the meaning of a program.
- Dynamic semantics (or Semantics)
  - This deals with the meaning of a program.
- Attribute Grammars (AG)
  - AG = CFG + attributes, semantic functions, and predicates
  - Primary value of attribute grammars
    - Static semantics specification
    - Compiler design (static semantics checking)

3.4 Attribute Grammars

- Attributes
  - For each grammar symbol X, there is a set of attributes
    A(X)
    = S(X) ∪ I(X)
    = { synthesized attributes } ∪ { inherited attributes }



  - It is possible that A(X) = Ø
  - Initially, there may be *intrinsic attributes* on the leaves

3.4 Attribute Grammars

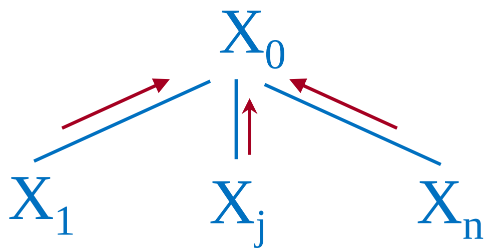
- Semantic functions
  - Semantic functions are attached to grammar rules.
  - Let $X_0 \rightarrow X_1 \ldots X_n$ be a rule

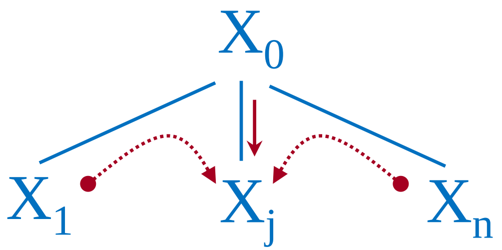
    Semantic function f defines synthesized attributes

    $S(X_0) = f(A(X_1), \ldots, A(X_j), \ldots, A(X_n))$

    Semantic function f defines inherited attributes

    $I(X_j) = f(A(X_0), \ldots, A(X_n))$





3.4 Attribute Grammars

- Predicate functions (Conditions)
  - Predicate functions are attached to grammar rules.
  - Let $X_0 \rightarrow X_1 \ldots X_n$ be a rule
    Predicate function **p** is a boolean-valued function
    p$(A(X_0), A(X_1), \ldots, A(X_n))$
- Languages generated by attribute grammars
  - A string ω of terminals is generated by an attribute grammar iff
    - it is generated by the CFG, and
    - all predicates are satisfied.

3.4 Attribute Grammars

- Example

  L = { ωcω | ω is a string of a's and b's }

  - CFG G

    S → XcY

    X → aX | bX | ε

    Y → aY | bY | ε

    L(G) = { ωcμ | ω and μ are strings of a's and b's }

  - Attributes

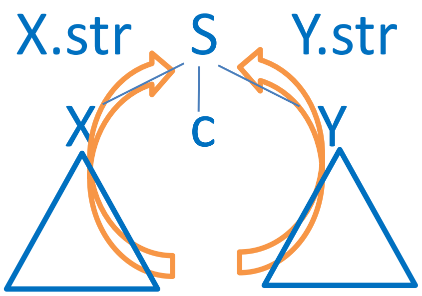
    X.str (synthesized), the string generated by X

    Y.str (synthesized), the string generated by Y

3.4 Attribute Grammars

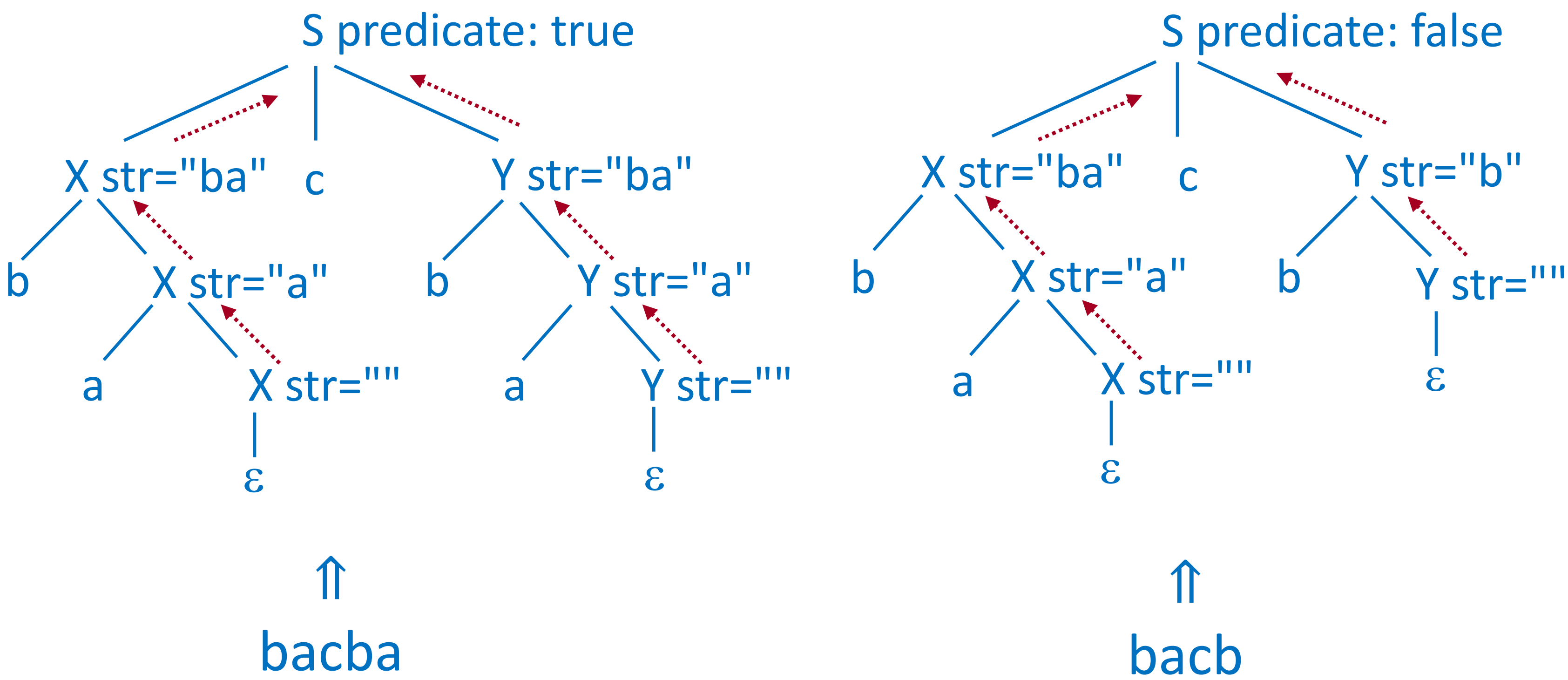
- Semantics and predicate functions

$S \rightarrow XcY$	predicate: $X.str == Y.str$
$X_1 \rightarrow aX_2$	$X_1.str = "a" + X_2.str$
$X_1 \rightarrow bX_2$	$X_1.str = "b" + X_2.str$
$X \rightarrow \varepsilon$	$X.str = ""$
$Y_1 \rightarrow aY_2$	$Y_1.str = "a" + Y_2.str$
$Y_1 \rightarrow bY_2$	$Y_1.str = "b" + Y_2.str$
$Y \rightarrow \varepsilon$	$Y.str = ""$



3.4 Attribute Grammars

- Fully attributed parse trees



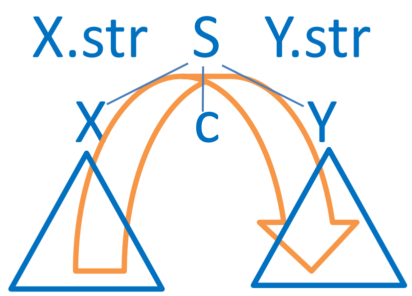
3.4 Attribute Grammars

- Example – Another attribute grammar
  - Attributes

    X.str (synthesized), the string generated by X

    Y.str (inherited), the string *expected* to be generated by Y

  - Semantic and predicate functions

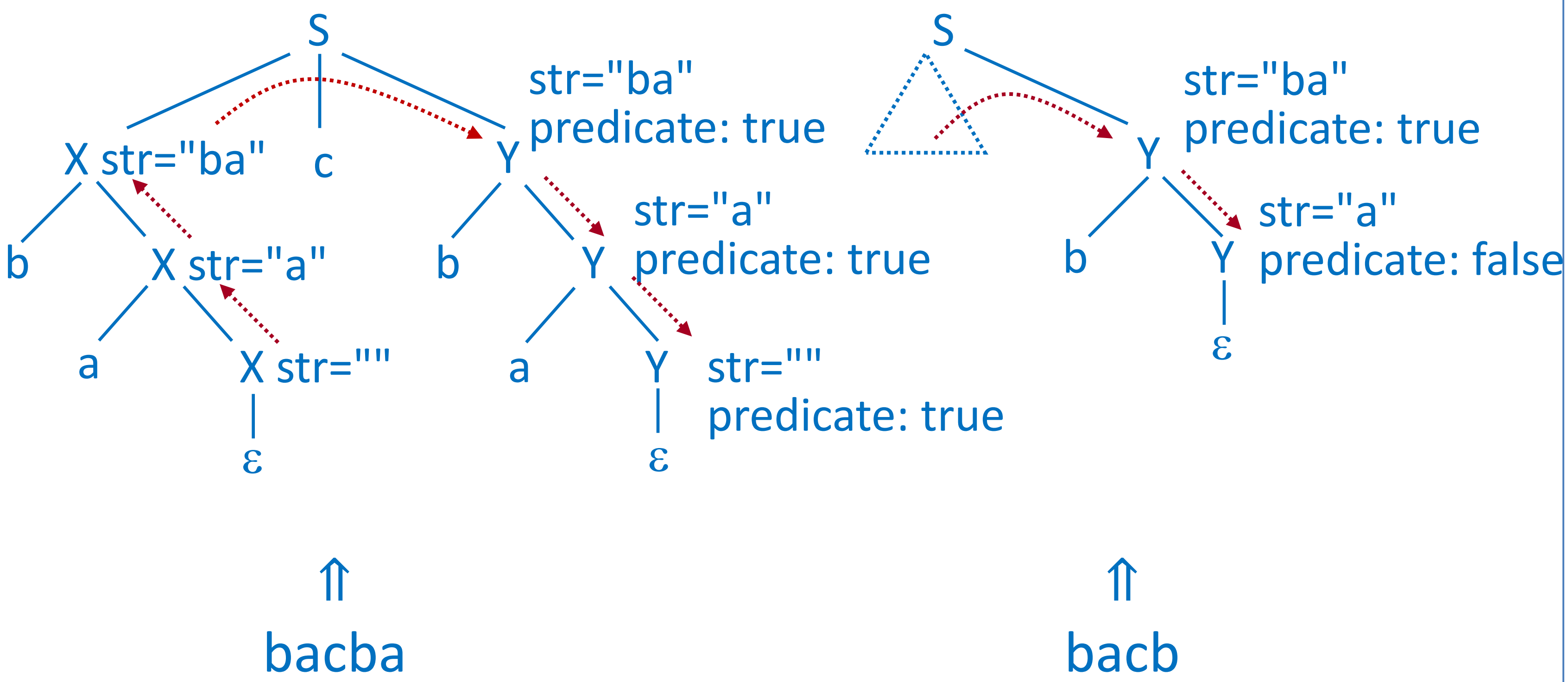


S	$\rightarrow$ XcY	Y.str = X.str
Y_1	$\rightarrow aY_2$	predicate: head(Y_1.str) == "a"
		Y_2.str = tail(Y_1.str)
Y_1	$\rightarrow bY_2$	predicate: head(Y_1.str) == "b"
		Y_2.str = tail(Y_1.str)
Y	$\rightarrow \varepsilon$	predicate: Y.str == ""

head("abb") = "a"
tail("abb") = "bb"

3.4 Attribute Grammars

- Fully attributed parse trees



3.4 Attribute Grammars

- Example 3.6
  - BNF

    <assign> → <var> = <expr>

    <expr> → <var> + <var> | <var>

    <var> → A | B | C
  - Attributes

    actual_type: synthesized for <var> and <expr>

    expected_type: inherited for <expr>
  - A, B, and C have *intrinsic attributes* which are their declared types.

3.4 Attribute Grammars

- Semantic and predicate functions

1. Syntax rule: <assign> → <var> = <expr>
   Semantic rule: <expr>.expected_type ← <var>.actual_type

2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rule: <expr>.actual_type ←
```
                    if (<var>[2].actual_type = int) and
                            (<var>[3].actual_type = int)
                       then int
                    else real
                    end if
```
   Predicate: <expr>.actual_type == <expr>.expected_type

3.4 Attribute Grammars

- Semantic and predicate functions

3. Syntax rule: <expr> → <var>
   Semantic rule: <expr>.actual_type ← <var>.actual_type
   Predicate: <expr>.actual_type == <expr>.expected_type

4. Syntax rule: <var> → `A` | `B` | `C`
   Semantic rule: <var>.actual_type ← look-up(<var>.string)

   The look-up function looks up a given variable in the symbol table and returns the variable's type.

3.5 Describing the Meanings of Programs

- Operational semantics
  - Describe the meaning of a program by executing its statements on a machine, either simulated or actual.
  - Informal operational semantics is the normal means of describing programming languages.

```
for (exp1;exp2;exp3)    ➡          exp1;
    stmt;                   loop:  if (!exp2) goto exit;
                                   stmt; exp3;
                                   goto loop;
                            exit:  ;
```

  - Formal methods: Vienna Definition Language (VDL) (IBM for PL/I); Structural operational semantics (ML)

3.5 Describing the Meanings of Programs

Axiomatic semantics

- Axiomatic specification (Hoare triple)
  - {P} S {Q}

    P = precondition, Q = postcondition
  - Correct axiomatic specifications reflect meanings.

    $\{x \geq 0\}\ x = x+1\ \{x \geq 1\}$ correct

    $\{x < 0\}\ x = x+1\ \{x \geq 1\}$ incorrect
  - How does one know if an axiomatic specification is correct?

    In axiomatic semantics, each language construct is given an axiom or inference rule to define its meaning.

3.5 Describing the Meanings of Programs

- Inference rule

  $$\frac{C}{\{P\}\ S\ \{Q\}}$$

  if C then {P} S {Q}

  C is called a verification condition (VC)

- Axiom – An inference rule without an antecedent.

- **Assignment axiom**
  - $\{Q_{x \to e}\}\ x = e\ \{Q\}$
  - Example

afterward

$\{\ x \geq 0\ \}\ x = x+1\ \{\ x \geq 1\ \}$ $\qquad (x \geq 1)_{x \to x+1} = x+1 \geq 1 = x \geq 0$

beforehand

3.5 Describing the Meanings of Programs

- Strength of conditions
  - If $P \rightarrow Q$, P is stronger than Q or Q is weaker than P
  - E.g. $x = 9 \rightarrow x \geq 5 \rightarrow x \geq 0$

    0,1,2,...,5,6,...,9,10,..............

  - $\{ x \geq 0 \}\ x = x+1\ \{ x \geq 1 \}$

    $\{ x \geq 5 \}\ x = x+1\ \{ x \geq 1 \}$

    $\{ x = 9 \}\ x = x+1\ \{ x \geq 1 \}$

    All are correct.

    Among them, $x \geq 0$ is the weakest precondition.

3.5 Describing the Meanings of Programs

- Strengthen precondition

$$\frac{P \rightarrow Q,\ \{Q\}\ S\ \{R\}}{\{P\}\ S\ \{R\}}$$

Weaken postcondition

$$\frac{\{\ P\}\ S\ \{Q\},\ Q \rightarrow R}{\{P\}\ S\ \{R\}}$$

- Together, rule of consequence

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

- Assignment rule

$$\frac{P \rightarrow Q_{x \rightarrow e}}{\{P\}\ x = e\ \{Q\}}$$

$\because$ by strengthen precondition

$$\frac{P \rightarrow Q_{x \rightarrow e},\ \{Q_{x \rightarrow e}\}\ x = e\ \{Q\}}{\{P\}\ x = e\ \{Q\}}$$

3.5 Describing the Meanings of Programs

- Example

  $\{ x \geq 5 \}\ x = x+1\ \{ x \geq 1 \}$

  Proof 1

  $x \geq 5 \rightarrow (x \geq 1)_{x\rightarrow x+1} = x \geq 0$, by assignment rule

  Proof 2

  $\{ x \geq 5 \}\ x = x+1\ \{ x \geq 6 \}$, assignment axiom

  $x \geq 6 \rightarrow x \geq 1$, weaken postcondition

- Weakest precondition transformer
  - wp(S,Q)
  - E.g. $wp(x = e,Q) = Q_{x\rightarrow e}$

3.5 Describing the Meanings of Programs

- Sequence rule
  - $$\frac{\{P\}\ S_1\ \{Q\},\ \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1;\ S_2\ \{R\}}$$

    May choose $Q = wp(S_2,R)$ and prove $P \rightarrow wp(S_1,wp(S_2,R))$

  - Example

    $\{ x = x_0 \wedge y = y_0 \}\ z = x;\ x = y;\ y = z\ \{ x = y_0 \wedge y = x_0 \}$

    $x = y_0 \wedge z = x_0$

    $y = y_0 \wedge z = x_0$

    $y = y_0 \wedge x = x_0$

    VC: $x = x_0 \wedge y = y_0 \rightarrow y = y_0 \wedge x = x_0$

    Trivial, $\wedge$ is commutative

3.5 Describing the Meanings of Programs

- IF rules
  - $$\frac{\{P \land B\}\ S_1\ \{Q\},\ \{P \land \neg B\}\ S_2\ \{Q\}}{\{P\}\ \text{if } B \text{ then } S_1 \text{ else } S_2\ \{Q\}} \qquad \frac{\{P \land B\}\ S\ \{Q\},\ P \land \neg B \rightarrow Q}{\{P\}\ \text{if } B \text{ then } S\ \{Q\}}$$
  - Example

    { true } if x > y then z = x else z = y { z = max(x,y) }

    VC1: true $\land$ x > y $\rightarrow$ x = max(x,y)

    VC2: true $\land$ $\neg$(x > y) $\rightarrow$ y = max(x,y)

    - true is the weakest condition, since P $\rightarrow$ true $\forall$P
      false is the strongest condition, since false $\rightarrow$ P $\forall$P
    - No precondition is needed.
      $\equiv$ The precondition is the weakest.

3.5 Describing the Meanings of Programs

- Example

  $\{ x = x_0 \}$ if $x < 0$ then $x = -x$ $\{ x = |x_0| \}$

  VC1: $x = x_0 \wedge x < 0 \rightarrow -x = |x_0|$

  Proof

  $x = x_0 \wedge x < 0 \Rightarrow x_0 < 0 \Rightarrow |x_0| = -x_0 \Rightarrow x = |x_0| \quad \because x = x_0$

  VC2: $x = x_0 \wedge \neg(x < 0) \rightarrow x = |x_0|$

  Proof

  $x = x_0 \wedge \neg(x < 0) \Rightarrow x_0 \geq 0 \Rightarrow |x_0| = x_0 \Rightarrow x = |x_0| \quad \because x = x_0$

  `

3.5 Describing the Meanings of Programs

- While rule
  - $$\frac{\{I \land B\}\ S\ \{I\}}{\{I\}\ \text{while } B \text{ do } S\ \{I \land \neg B\}} \qquad \frac{P \rightarrow I,\ \{I \land B\}\ S\ \{I\},\ I \land \neg B \rightarrow Q}{\{P\}\ \text{while } B \text{ inv } I \text{ do } S\ \{Q\}}$$

    where I is the loop invariant
  - Example

    $\{ n \geq 0 \}$

    k = n; f = 1;

    P $\rightarrow$ while k <> 0 inv $I \equiv f*k! = n! \land k \geq 0$ do

    f = f*k; k = k-1

    $\{ f = n! \}$

    Other invariants don't help, e.g. $f \neq 0$, $k \leq n$

n	f	k
5	1	5
5	5	4
5	5*4	3

3.5 Describing the Meanings of Programs

- Example (Cont'd)

  We may choose

  $P \equiv n \geq 0 \wedge k = n \wedge f = 1$

  and show that

  $P \rightarrow I$

  i.e. $n \geq 0 \wedge k = n \wedge f = 1 \rightarrow f{*}k! = n! \wedge k \geq 0$

  A better way

  Choose $P \equiv I$, making $P \rightarrow I$ trivial

  1) $\{ n \geq 0 \}$ k = n; f = 1 $\{ I \}$

     $n \geq 0 \rightarrow (f{*}k! = n! \wedge k \geq 0)_{f\rightarrow 1, k\rightarrow n} \equiv 1{*}n! = n! \wedge n \geq 0$

     Trivial

3.5 Describing the Meanings of Programs

- Example (Cont'd)

  2) $\{ I \wedge k \neq 0 \}$ f = f*k; k = k-1 $\{ I \}$

     $f*k! = n! \wedge k \geq 0 \wedge k \neq 0$

     $\rightarrow (f*k! = n! \wedge k \geq 0)_{k \rightarrow k-1, f \rightarrow f*k}$

     $\equiv f*k*(k-1)! = n! \wedge (k-1) \geq 0$

     Proof

     $k \geq 0 \wedge k \neq 0$

     $\Rightarrow k > 0$

     $\Rightarrow k! = k*(k-1)! \wedge (k-1) \geq 0$

     $\Rightarrow f*k! = f*k*(k-1)! \wedge (k-1) \geq 0$

     $\Rightarrow n! = f*k*(k-1)! \wedge (k-1) \geq 0 \quad \because f*k! = n!$

3.5 Describing the Meanings of Programs

- Example (Cont'd)

  3) $I \land \neg(k \neq 0) \rightarrow f = n!$

     $f*k! = n! \land k \geq 0 \land \neg(k \neq 0) \rightarrow f = n!$

     Proof

     $\neg(k \neq 0)$

     $\Rightarrow k = 0$

     $\Rightarrow f*0! = n! \qquad \because f*k! = n!$

     $\Rightarrow f = n!$

3.5 Describing the Meanings of Programs

- Total and Partial correctness
  - Partial correctness

    If P is true and S terminates, Q must be true.

    Precondition + Termination $\Rightarrow$ Postcondition
  - Total correctness

    If P is true, S must terminate and Q must be true.

    Total Correctness = Partial Correctness + Termination
  - Example (Cont'd)

    the value of k decreases on each iteration $\wedge$ $k \geq 0$ is invariant

    $\Rightarrow$ the decreasing sequence is finite

    $\Rightarrow$ the loop will eventually terminate

3.5 Describing the Meanings of Programs

- Example – Partial correctness

  { true }

  k = n; f = 1;

  while k <> 0 inv I do where $I \equiv n < 0 \lor (f*k! = n! \land k \geq 0)$

  f = f*k; k = k-1

  { $n < 0 \lor f = n!$ }

  Observe that the loop won't terminate if $n < 0$.

  Replacing <> by > gives rise to total correctness.

  1) { true } k = n; f = 1 { I }

     $true \rightarrow n < 0 \lor (1*n! = n! \land n \geq 0)$

     Proof: If $n < 0$, we are done.

     Otherwise, $n \geq 0$ and $1*n! = n!$, as desired

3.5 Describing the Meanings of Programs

- Example (Continued)

  2) $\{ I \land k \neq 0 \}$ f = f*k; k = k-1 $\{ I \}$

     $(n < 0 \lor (f*k! = n! \land k \geq 0)) \land k \neq 0$

     $\rightarrow n < 0 \lor (f*k*(k-1)! = n! \land k-1 \geq 0)$

     Proof

     If $n < 0$, we are done.

     Otherwise, $f*k! = n! \land k \geq 0 \land k \neq 0$ is true.

     $k \geq 0 \land k \neq 0$

     $\Rightarrow k > 0$

     $\Rightarrow k! = k*(k-1)! \land (k-1) \geq 0$

     $\Rightarrow f*k! = f*k*(k-1)! \land (k-1) \geq 0$

     $\Rightarrow f*k*(k-1)! = n! \land (k-1) \geq 0 \quad \because f*k! = n!$

3.5 Describing the Meanings of Programs

- Example (Continued)

  3) $I \land \neg(k \neq 0) \rightarrow n < 0 \lor f = n!$

     $(n < 0 \lor (f*k! = n! \land k \geq 0)) \land \neg(k \neq 0) \rightarrow n < 0 \lor f = n!$

     Proof

     If $n < 0$, we are done.

     Otherwise, $f*k! = n! \land k \geq 0 \land \neg(k \neq 0)$ is true.

     Thus,

     $\neg(k \neq 0)$

     $\Rightarrow k = 0$

     $\Rightarrow f*0! = n! \quad \because f*k! = n!$

     $\Rightarrow f = n!$

3.5 Describing the Meanings of Programs

- Pro and Con of axiomatic semantics

  Pro

  - Loop invariants are the most valuable comments

  Con

  - Need program prover = VC generator + theorem prover
  - "Specification is correct" doesn't necessarily mean "program is correct".
  - Axioms are not so easy to define.

    e.g. the previous assignment axiom doesn't always work.

    { x = 5 } y = x++ { x = 6 } due to side effect

    { i = j } a[i] = 7 { a[j] = 7 } due to array

3.5 Describing the Meanings of Programs

- Array assignment axiom
  - Notation

    Let a be an array, then <a,i,e> is an array defined as

    $<a,i,e>[i] = e$

    $<a,i,e>[j] = a[j]$, where $j \neq i$
  - Array assignment axiom and rule

    $\{ Q_{a \rightarrow <a,i,e>} \}\ a[i] = e\ \{ Q \}$

    $\{ Q_{a \rightarrow <a,i,e>} \}\ a = <a,i,e>\ \{ Q \}$

    $$\frac{P \rightarrow Q_{a \rightarrow <a,i,e>}}{\{ P \}\ a[i] = e\ \{ Q \}}$$
  - Example

    $\{ i = j \}\ a[i] = 7\ \{ a[j] = 7 \}$

    VC: $i = j \rightarrow <a,i,7>[j] = 7$

3.5 Describing the Meanings of Programs

Denotational semantics

- Denotational semantics
  - Each language construct is given a semantic function to specify the value denoted by the construct.
  - Meaning: Syntax → Semantic
  - Example 3.5.2.1

    <bin_num> → '0' | '1' | <bin_num> '0' | <bin_num> '1'

    $M_{bin}('0') = 0$

    $M_{bin}('1') = 1$

    $M_{bin}(\text{<bin\_num> '0'}) = 2 \times M_{bin}(\text{<bin\_num>})$

    $M_{bin}(\text{<bin\_num> '1'}) = 2 \times M_{bin}(\text{<bin\_num>}) + 1$

3.5 Describing the Meanings of Programs

- De facto standard notation
  - Example (Continued)

    Num ::= 0 | 1 | Num 0 | Num 1

    Syntactic domain

    Num = the set of all binary numerals generated by Num

    (Convention: Use the same name for the syntactic category and the syntactic domain)

    N: Num, meaning that N is an element of Num

    (Convention: Use the first letter of the syntactic domain)

    Semantic domain

    $\mathbb{N}$ = the set of nonnegative integers

3.5 Describing the Meanings of Programs

- Example (Cont'd)

  Semantic function $\mathcal{N}$: Num $\rightarrow \mathbb{N}$

  $\mathcal{N}[\![0]\!] = 0$

  $\mathcal{N}[\![1]\!] = 1$

  $\mathcal{N}[\![N0]\!] = 2 \times \mathcal{N}[\![N]\!]$

  $\mathcal{N}[\![N1]\!] = 2 \times \mathcal{N}[\![N]\!] + 1$

  (Convention: Entities inside $[\![\ ]\!]$ are syntactic.)

  $\mathcal{N}[\![101]\!]$
  $= 2 \times \mathcal{N}[\![10]\!] + 1$
  $= 2 \times 2 \times \mathcal{N}[\![1]\!] + 1 = 2 \times 2 \times 1 + 1 = 5$

  So, 101 denotes 5; or 5 is the denotation of 101.

3.5 Describing the Meanings of Programs

- Semantic functions are defined according to the grammar.

  Num ::= 0 | 1 | 0 Num| 1 Num

  Semantic functions

  $\mathcal{N} : \text{Num} \rightarrow \mathbb{N}$

  $\mathcal{N}[\![0]\!] = 0$

  $\mathcal{N}[\![1]\!] = 1$

  $\mathcal{N}[\![0\text{N}]\!] = \mathcal{N}[\![\text{N}]\!]$

  $\mathcal{N}[\![1\text{N}]\!] = 2^{\mathcal{L}[\![\text{N}]\!]} + \mathcal{N}[\![\text{N}]\!]$ e.g. $\mathcal{N}[\![101]\!] = 2^{\mathcal{L}[\![01]\!]} + \mathcal{N}[\![01]\!]$

  $\mathcal{L} : \text{Num} \rightarrow \mathbb{N}$

  $\mathcal{L}[\![\text{N}]\!]$ = the number of binary digits in N

  $\mathcal{L}[\![0]\!] = \mathcal{L}[\![1]\!] = 1$

  $\mathcal{L}[\![0\text{N}]\!] = \mathcal{L}[\![1\text{N}]\!] = 1 + \mathcal{L}[\![\text{N}]\!]$

3.5 Describing the Meanings of Programs

- A very simple example in book
  - Grammar for very simple expressions

    <expr> → <dec_num> | <var> | <binary_expr>

    <binary_expr> → <left_expr> <operator> <right_expr>

    <left_expr> → <dec_num> | <var>

    <right_expr> → <dec_num> | <var>

    <operator> → + | *

    <dec_num> → 0 | …| 9 | <dec_num> (0 | … | 9)

    <var> :: = *left unspecified*

3.5 Describing the Meanings of Programs

- State

  The state of a program is the values of all its variables, e.g.

  state $s = \{<i_1,v_1>, <i_2,v_2>, \ldots, <i_n,v_n>\}$

  where the v_i's are integers or the special value undef.

  VARMAP is a function for looking up the value of a variable, e.g. $\text{VARMAP}(i_j,s) = v_j$
- Semantic domain

  $\mathbb{Z} \cup \{\text{error}\}$, where $\mathbb{Z}$ = the set of integers
- Semantic functions

  M_{dec}: decimal numerals $\rightarrow \mathbb{Z}$

  M_e : expressions $\times$ states $\rightarrow \mathbb{Z} \cup \{\text{error}\}$

3.5 Describing the Meanings of Programs

```
M_e(<expr>, s) Δ =
    case <expr> of
        <dec_num> => M_dec(<dec_num>,s)
        <var> => if VARMAP(<var>,s)==undef then error
                 else VARMAP(<var>,s)
        <binary_expr> =>
            if (M_e(<binary_expr>.<left_expr>,s)==undef
                or M_e(<binary_expr>.<right_expr>,s)==undef) then error
            else
                if (<binary_expr>.<operator>=='+' then
                    M_e(<binary_expr>.<left_expr>,s) +
                            M_e(<binary_expr>.<right_expr>,s)
                else
                    M_e(<binary_expr>.<left_expr>, s) *
                            M_e(<binary_expr>.<right_expr>, s)
```

3.5 Describing the Meanings of Programs

- A rewritten and extended example
  - *Abstract* syntax

    Exp ::= Num | Var | Exp Op Exp

    Num ::= Digit | Num Digit

    Digit ::= 0 |1 | … | 9

    Op ::= + | - | * | / | %

    Var :: = *left unspecified*
  - Syntax domains

    Exp = the language generated by Exp

    Num = the language generated by Num

    And so on

3.5 Describing the Meanings of Programs

- Semantic domains

  $\mathbb{Z}$ = the set of integers

  $\mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$

  Store = Var $\rightarrow \mathbb{Z}_\perp$ = { s | s : Var $\rightarrow \mathbb{Z}_\perp$}

  $\text{Store}_\perp$ = Store $\cup \{\perp\}$

  Operator = $\{+, -, \times, \text{div}, \text{mod}\}$

  - The bottom $\perp$ denotes an undefined value or an error.
  - A store (or state) $s = \{ (i_1,v_1), (i_2,v_2), (i_3,v_3), \ldots \}$ is viewed as a function that maps a variable to the value stored in it.
  - Var $\rightarrow \mathbb{Z}_\perp$ denotes the set of all stores.

3.5 Describing the Meanings of Programs

- Semantic functions

  $\mathcal{D}$: Digit $\rightarrow \mathbb{Z}$

  $\mathcal{D}[\![0]\!] = 0$, $\mathcal{D}[\![0]\!] = 1$, …, $\mathcal{D}[\![9]\!] = 9$

  $\mathcal{N}$: Num $\rightarrow \mathbb{Z}$

  $\mathcal{N}[\![D]\!] = \mathcal{D}[\![D]\!]$, $\mathcal{N}[\![ND]\!] = 10 \times \mathcal{N}[\![N]\!] + \mathcal{D}[\![D]\!]$

  $\mathcal{O}$: Op $\rightarrow$ Operator

  $\mathcal{O}[\![+]\!] = +$, $\mathcal{O}[\![-]\!] = -$, $\mathcal{O}[\![*]\!] = \times$, $\mathcal{O}[\![/]\!] = \text{div}$, $\mathcal{O}[\![\%]\!] = \text{mod}$

  $\mathcal{E}$: Exp $\rightarrow$ Store $\rightarrow \mathbb{Z}_\perp$

  $\mathcal{E}[\![N]\!]\ s = \mathcal{N}[\![N]\!]$

  $\mathcal{E}[\![V]\!]\ s = s(V)$

  $\mathcal{E}[\![E_1\ O\ E_2]\!]\ s = (\mathcal{E}[\![E_1]\!]\ s)\ \mathcal{O}[\![O]\!]\ (\mathcal{E}[\![E_2]\!]\ s)$

3.5 Describing the Meanings of Programs

- Comment on $\mathcal{E} : \text{Exp} \rightarrow \text{Store} \rightarrow \mathbb{Z}_{\perp}$

  a function of the type $\text{Store} \rightarrow \mathbb{Z}_{\perp}$

  $$\underline{\mathcal{E}\overbrace{[\![E]\!]}\ s}$$

  apply the function $\mathcal{E}[\![E]\!]$ to the store s
- Convention

  The parentheses in $\mathcal{E}[\![E]\!](s)$ are removed.
- Example

  The expression

  $x + 2$

  denotes a function $e : \text{Store} \rightarrow \mathbb{Z}_{\perp}$ defined by

  $e(s) = s(x) + 2$

3.5 Describing the Meanings of Programs

- Example (Cont'd)

  In detail,

  $\mathcal{E}[\![x+2]\!]\ s$

  $= (\mathcal{E}[\![x]\!]\ s)\ \mathcal{O}[\![+]\!]\ (\mathcal{E}[\![2]\!]\ s)$

  $= s(x) + \mathcal{N}[\![2]\!]$

  $= s(x) + \mathcal{D}[\![2]\!]$

  $= s(x) + 2$

  Observe that the meaning of x + 2 depends on the value of x in a store, e.g.

  $s_1 = \{(x,3)\} \Rightarrow \mathcal{E}[\![x+2]\!]\ s_1 = s_1(x) + 2 = 3 + 2 = 5$

  $s_2 = \{(x,\perp)\} \Rightarrow \mathcal{E}[\![x+2]\!]\ s_2 = s_2(x) + 2 = \perp + 2 = \perp$

3.5 Describing the Meanings of Programs

- Adding commands (i.e. statements)

  Grammar

  Com ::= Var = Exp | while Exp do Com | Com; Com

  Semantic function

  $\mathcal{C}$: Com → Store → $\text{Store}_\perp$

  $\mathcal{C}$⟦V = E⟧ s

  = ⊥, if $\mathcal{E}$⟦E⟧ s = ⊥

  = s′, where s′(V) = $\mathcal{E}$⟦E⟧ s, and s′(I) = s(I), I ≢ V, otherwise

  $\mathcal{C}$⟦C_1; C_2⟧ s

  = ⊥ , if $\mathcal{C}$⟦C_1⟧ s = ⊥

  = $\mathcal{C}$⟦C_2⟧ ($\mathcal{C}$⟦C_1⟧ s), otherwise

3.5 Describing the Meanings of Programs

- Adding commands (Cont'd)

  $\mathcal{C}[\![\text{while E do C}]\!]\ s$

  $= \perp$, if $\mathcal{E}[\![\text{E}]\!]\ s = \perp$

  $= s$, if $\mathcal{E}[\![\text{E}]\!]\ s = 0$

  $= \perp$, if $\mathcal{E}[\![\text{E}]\!]\ s \neq 0$ and $\mathcal{C}[\![\text{C}]\!]\ s = \perp$

  $= \mathcal{C}[\![\text{while E do C}]\!]\ (\mathcal{C}[\![\text{C}]\!]\ s)$, otherwise

  - 0 is treated as false and any non-zero value as true.
  - Nontermination of a while command isn't handled directly in our semantics.

    Indeed, $\mathcal{C}$ is a partial function, e.g. for any store s

    $\mathcal{C}[\![\text{while 1 do x = 2}]\!]\ s$

    gives no value at all.

3.5 Describing the Meanings of Programs

- Example

  The program

  a = 1; b = a+2

  denotes a function d : Store $\rightarrow$Store$_\perp$ defined by

  d(s) = s′, where s′(a) = 1, s′(b) = 3, and s′(I) = s(I), I $\not\equiv$ a,b

  In detail,

  $\mathcal{C}[\![$a = 1; b = a+2$]\!]$ s

  = $\mathcal{C}[\![$b = a+2$]\!]$ s_1

  where s_1 = $\mathcal{C}[\![$a = 1$]\!]$ s satisfies

  s_1(a) = $\mathcal{E}[\![1]\!]$ s = $\mathcal{N}[\![1]\!]$ = $\mathcal{D}[\![1]\!]$ = 1

  and

  s_1(I) = s(I), I $\not\equiv$ a

3.5 Describing the Meanings of Programs

- Example (Cont'd)

  $\mathcal{C}[\![a = 1;\ \ b = a+2]\!]\ s$

  $= \mathcal{C}[\![b = a+2]\!]\ s_1$

  $= s'$

  where $s' = \mathcal{C}[\![b = a+2]\!]\ s_1$ satisfies

  $s'(b) = \mathcal{E}[\![a+2]\!]\ s_1$

  $\quad = (\mathcal{E}[\![a]\!]\ s_1)\ \mathcal{O}[\![+]\!]\ (\mathcal{E}[\![2]\!]\ s_1)$

  $\quad = s_1(a) + 2 = 1 + 2 = 3$

  and

  $s'(I) = s_1(I),\ I \not\equiv b$

  $\Rightarrow s'(a) = s_1(a) = 1,\ s'(I) = s_1(I) = s(I),\ I \not\equiv a, b$