Chapter 3

- Syntax & Semantics Definition
- Methods of describing syntax
  - Context-free grammar using
    recursive descent parsing
  - Backus-Naur Form(BNF)
  - Syntax graphs
  - Attribute grammar describing
    syntax and static semantics of a PL
- Methods of describing semantics
  - Operational semantics
    recursive descent parsing
  - Axiomatic semantics
  - De-notational semantics
- Methods of regular grammar(補充)
  - Deterministic Finite Automata (DFA)

Chapter 3

***Syntax* - the form or structure of the expressions, statements, and program units described by CFG(BNF)**

***Semantics* - the meaning of the expressions, statements, and program units**

Who must use language definitions?(p.106)

1. Other language designers
2. Language implementors
3. Programmers (the users of the language)
   How to encode SW by a language manual

A *sentence* is a string of characters from the language's alphabet(match some syntax rules)
A *language* is a set of sentences
A *lexeme* is the lowest level syntactic unit of a language (e.g., `*`, `sum`, `begin`) (p.108)
A *token* is a category of lexemes (e.g., identifier)
A *syntax rule* is to specify which strings of characters from the language's alphabet are in the language

Formal approaches to describing syntax:

1. Recognizers(R) - used in compilers
2. Generators-generate the sentence of a language

$$\begin{tikzcd} & \text{in L(accept the string)} \\ \Sigma \arrow[r] & R \arrow[u] \arrow[d] \\ & \text{not in L(reject the string)} \end{tikzcd}$$

Syntax analysis part of a compiler is a recognizer!!!
Syntax analyzer(parsers) determines whether the given programs are syntactically correct!!!

Chapter 3

A Language generator likes a button push to produce the Sentences of a language
Syntax checking portion of a compiler can be used in a trial-and-error mode.(programmer just guess the syntax)

In the mid-1950s Chomsky (linguist) proposed 4 classes of generative devices or grammars that define 4 classes of Languages: (describe the language syntax)

Type 0:	Natural Language
Type 1:	Context Sensitive Language
Type 2:	Context Free Language
Type 3:	Regular Language(No memory)

Regular language is useful for describing the syntax of a PL(corresponding a regular grammar).
A regular grammar can be describing by a DFA. →補充

Context-Free Grammars(CFG)

- Define a class of languages called *context-free languages(CFL) recognized by CFG*
- CFG describes the PL with minor exception
- *A* new formal notation for specifying PL's syntax

Backus Normal Form (1959)

- Invented by John Backus of the ACM-GAMM to describe ALGOL 58(language syntax)
- BNF is equivalent to context-free grammars

A *meta-language* is a language used to describe another language. BNF is a meta-language.

Chapter 3

In BNF(CFG), *abstractions* are used to represent classes of syntactic structures -- they act like syntactic variables (also called *non-terminal symbols*)
e.g.(p. 110, 111)

<if_stmt> → if <logic_expre> then <stmt>
<if_stmt> → if <logic_expre> then <stmt> else <stmt>
Can be denoted as
<if_stmt> → if <logic_expre> then <stmt> |
if <logic_expre> then <stmt> else <stmt>

```
<while_stmt> → while <logic_expr> do <stmt>
```

This is a *rule*; it describes the structure of a `while` statement

A rule(production) has a left-hand side(LHS)[non-terminal] and a right-hand side (RHS), and consists of *terminal* and *non-terminal* symbols

A *grammar* is a finite nonempty set of rules

An abstraction(or production/rules/non-terminal) can have more than one RHS

The lexeme and tokens of the rules are terminal symbol

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

Syntactic lists are described in BNF using recursion

A rule(production) is *recursive* if its LHS appears in its RHS
ex. <ident_list> → identifier | identifier, <ident_list>

Chapter 3

A BNF(CFG) are denoted by terminal, non-terminal, production

A *derivation* is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

An example grammar:

1. <program> → <stmt_list> (<program> is start symbol)
2. <stmt_list> → <stmt> | <stmt> ; <stmt_list>
3. <stmt> → <var> = <expression>
4. <var> → a | b | c | d
5. <expression> → <var> + <var> | <var> - <var> | <var>

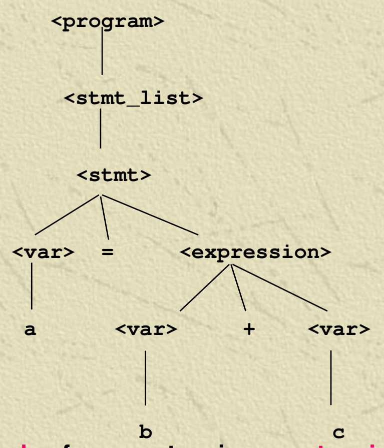
An example leftmost derivation

```
<program> ⇒ <stmt_list> ⇒ <stmt> ; <stmt_lsit>      (1,2)
          ⇒ <var> = <express>; <stmt_list>          ( 3 )
          ⇒ a = <expression>; <stmt_list>           ( 4 )
          ⇒ a = <var> + <var>; <stmt_list>          ( 5 )
          ⇒ a = b + <var>;  <stmt_list>             ( 4 )
          ⇒ a = b + c; <stmt_list>                  ( 4 )
          ⇒ a = b + c; <stmt>                       ( 2 )
          ⇒ a = b + c; <var> = <expression>         ( 3 )
          ⇒ a = b + c; b = <expression>             ( 4 )
          ⇒ a = b + c; b = <var>                    ( 5 )
          ⇒ a = b + c; b = c                        ( 4 )
```

Every string of symbols in the derivation is a *sentential form*
A *sentence* is a sentential form that has only terminal symbols

Chapter 3

A *leftmost derivation* is one in which the leftmost non-terminal in each sentential form is the one that is expanded
A derivation may be neither leftmost nor rightmost
A *parse tree* is a hierarchical representation of a derivation



Every internal node of a parse tree is a non-terminal

Every leaf of a parse tree is a terminal

Every sub-tree of a parse tree is an instance of an abstraction in the statement

A grammar is *ambiguous* iff it generates a sentence for which there are two or more distinct parse trees

Chapter 3

Ambiguous Grammar(p. 114)

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

Syntactic ambiguity of language structures is a problem. A grammar can produce two parse trees is called ambiguous. Operator Precedence is a problem.

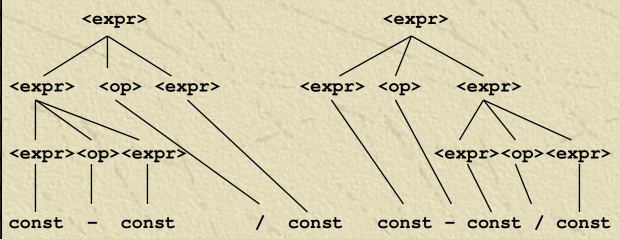
Operator in an arithmetic expression is generated *lower* in the parse tree can be used to indicate that it *has precedence over* an operator produced ***higher up*** in the tree.
A grammar can be written to separate the addition and multiplication operators so they are consistently in a higher to lower ordering in the parse tree.(p. 116)

<Assign> → <id> ：= <expr>
<id> → A | B | C
<expr> → <expr> + <term> | <expr> − <term>
| <term>
<term> → <term> * <factor> | <term> / <factor>
| <factor>
<factor> → (<expr>) | <id>

Chapter 3

An ambiguous expression grammar:

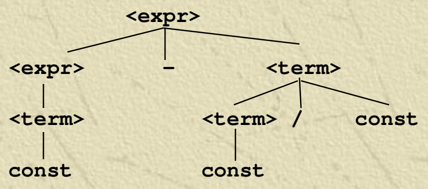
```
<expr> -> <expr> <op> <expr>  |  const
<op> -> /  |  -
```



If we use the parse tree to indicate precedence levels of the operators, it will be no ambiguous

An unambiguous expression grammar:

```
<expr> -> <expr> - <term>  |  <term>
<term> -> <term> / const  |  const
```

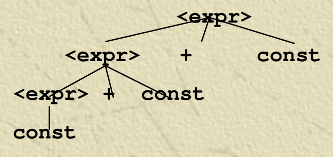


Chapter 3

```
<expr> => <expr> - <term> => <term> - <term>
       => const - <term>
       => const - <term> / const
       => const - const / const
```

Operator associativity can also be indicated by a Grammar(ex. + is left associative, but in mathematics + is both left and right associative)

`<expr> -> <expr> + <expr>   |   const` **(ambiguous)**
`<expr> -> <expr> + const   |   const` **(unambiguous)**



A BNF rule(production) has its LHS also appearing at the beginning of its RHS, the rule is said to be *left recursive.*
The Left recursive is meant to be left associative.(p. 119)
The right recursive is meant to be right associative.

＜factor＞→＜expr＞** ＜factor＞|＜expr＞(consider power)
＜expr＞→ （＜expr＞）|＜id＞

Can be used to describe exponentiation as a right associative operator. Add is left associative, but float-point addition, subtraction and division are not associative.(pp. 118)
Ex． 10個1 + 1.0*10^7 and just 7 digits of accuracy

Chapter 3

＜Assign＞→＜id＞：＝＜expr＞（p. 116 & 119）
＜id＞→ A | B | C
＜expr＞→＜expr＞＋＜term＞|＜expr＞－＜term＞
|＜term＞|＜id＞
＜term＞→＜term＞*＜factor＞|＜term＞/＜factor＞
|＜factor＞
＜factor＞→＜expr＞^＜factor＞|＜expr＞|（＜expr＞）|
＋＜id＞| －＜id＞

Ambiguous grammar for if statement (p.119, p.120 Fig. 3.5)
＜stmt＞→＜if_stmt＞|＜Assign＞
＜if_stmt＞→ if ＜logic_expr＞ then ＜stmt＞
| if ＜logic_expr＞ then ＜stmt＞ else ＜stmt＞

We can develop an unambiguous grammar for if constructs that <u>an else clause is matched with the nearest previous unmatched then</u>. Statement can be distinguished by match and un-match clause.(p. 119)
＜stmt＞→＜if_stmt＞|＜Assign＞
＜if_stmt＞→＜matched＞|＜unmatched＞
＜matched＞→ if ＜logic_expr＞ then ＜matched＞ else ＜matched＞
| any non-if statement
＜unmatched＞→ if ＜logic_expr＞ then ＜stmt＞
| if ＜logic_expr＞ then ＜matched＞ else ＜unmatched＞

＜stmt＞→＜if_stmt＞→＜unmatched＞→ [Fig. 3.5]
if ＜logic_expr＞ then ＜stmt＞→
if ＜logic_expr＞ then ＜matched＞→ … results

Ambiguous VB statement for assign
c=A(I,j) → A is a function or array?(how to distinguish?)

Chapter 3

Extended BNF(EBNF ➔ just abbreviations) is not increased the BNF power but the readability and writability. The notation is just extended the RHS by the following forms:

1. Optional parts are placed in brackets ([])

<if_stmt>→if ＜logic_expr＞ then ＜stmt＞【else <stmt>】

2. Put multiple choices of RHSs in parentheses and separate them with vertical bars

<term> →<term> (* | /) <factor>

<for_stmt> → for <var>:=<expr> (to | downto) <expr> do <stmt>

3. Put repetitions (0 or more) in braces ({})

<ident> →letter {letter | digit}

<ident_list>→<identifier> {, <identifier>}

BNF: (left-recursive grammar)

<expr> →<expr> + <term> | <expr> - <term> | <term>

<term>→<term> * <factor> | <term> / <factor> | <factor>

EBNF: (see 補充)

<expr>→<term> {(+ | -) <term>}

<term> →<factor> {(* | /) <factor>}

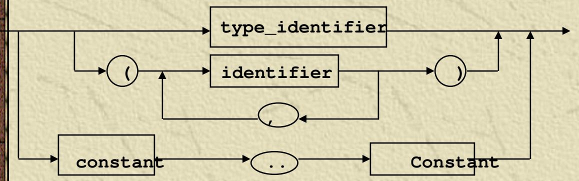
Some version of EBNF allow a numeric superscript to be attached to the right brace to indicate an upper limit to the number of times the enclosed part can be repeated.

Ex. <compound> → begin { <stmt>}$^+$ end

＜logic_expr＞ → <expr> (> | >= | < | <= | == | !=) <expr>

Syntax Graphs **- put the terminals in circles or ellipses and put the non-terminals in rectangles; connect with lines with arrowheads**

e.g., Pascal type declarations

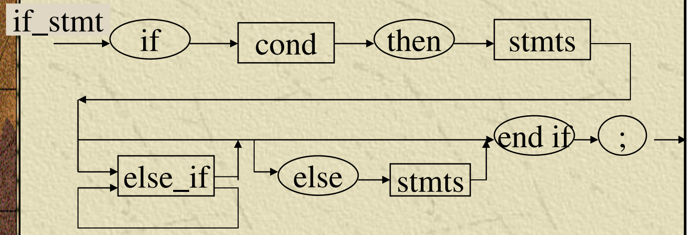


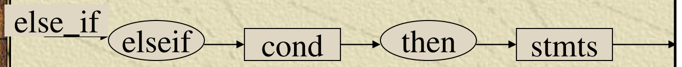
EBNF

<if_stmt>→if <condition> then <stmt>[else_if <stmt>]

<else_if> →elseif <condition> then ＜stmt＞

Syntax graphs





Chapter 3

Lexical analyzer is often called the scanner
Syntax Analyzer is often called the parser
Syntax Analyzer generator in UNIX is named yacc
Two kinds of parsers are top-down and bottom-up.
Top-down parser builds the parse tree from the root downto the leaves(Recursive Decent Parser)
Bottom-up parser builds the parse tree from the leaves upto the root

Recursive Descent Parsing(RDP)

- RDP is a grammar-based top-down parser(CFG)
- Parsing is the process of tracing or constructing a parse tree for a given input string
- Parsers usually do not analyze lexemes; that is done by a lexical analyzer, which is called by the parser
- A *recursive descent parser* traces out a parse tree in top-down order; it is a top-down parser
- Each non-terminal in the grammar has a subprogram associated with it; the subprogram parses all sentential forms that the non-terminal can generate
- The recursive descent parsing subprograms are built directly from the grammar rules(production)
- Recursive descent parsers, like other top-down parsers, cannot be built from left-recursive grammars

When given an input string, it traces out the parse tree that can be rooted at that non-terminal and whose leaves match the input string(p. 123)

Ex: For the EBNF:

```
1.  <expr>→<term> {(* | -) <term>}
2.  <term>→<factor> {(* | /) <factor>}
3.  <factor>→<id> | ( <expr> )
```

Chapter 3

We could use the following recursive descent parsing subprogram in C are shown followed:

```
void expr() { //<expr>→<term> {(* | -) <term>}
 term();  /* parse the first factor*/
 while (next_token =='+'|| next_token == '-') {
   lexical();  /* get next token from input */
   term();  /* parse the next term */
 }
}
void term(){//<term>→<factor> {(* | /) <factor>}
 factor();  /* parse the first factor*/
 while (next_token == '*'|| next_token == '/') {
   lexical();  /* get next token from input */
   factor();  /* parse the next factor */
 }
}
void factor() { // <factor>→<id> | ( <expr> )
 if (next_token == id_code) { lexical(); return;}
 else if (next_token == '(') {
     lexical(); expr();
     if (next_token == ')') {
       lexical(); return;
     }
     else error(); // not just stop should cont.
 }
 else error();//not just stop & show syntax error
}

void error() {
  printf("some errors are found!!!\n");
}
```

Chapter 3

Static Semantics(p. 126)

Some characteristics of PL are difficult to describe with BNF.

1. Context-free but cumbersome (e.g. type checking)
   Ex. Variables assign values
   limitation: floating-point value cannot put into int var.
2. Non-context-free (e.g. variables must be declared before they are used)

The static semantics of a language is related to the meaning of programs during execution.(type constraints of a language)

Attribute Grammars (AGs) (Knuth, 1968)

- CFG cannot describe all of the syntax of a PL
- Additions to CFGs can carry some semantic info along through parse trees

Primary value of AGs:

1. Describing static semantics with BNF
2. Describing both syntax and the static semantics
3. Compiler design(static semantics checking)
   a. Attribute computation function(function)
   b. Predicate function(state syntax and semantic rules)

Def: An *attribute grammar* is a CFG G = (V, T, S, P) with the following additions:

1. For each grammar symbol x there is a set of attribute values A(x) (Synthesized or Inherited attributes)
2. Each grammar production's non-terminal symbol has a set of semantic functions

Chapter 3

For a rule $X_0 \rightarrow X_1 \ldots X_n$

The synthesize attribute of X_0 are computed with a semantic function $S(X_0)=f(A(X_1),\ldots,A(X_n))$

The attribute value depends on the children nodes (pass semantic information up a parse tree)

The inherited attribute of X_j, $1<=j<=n$ are computed with a semantic function $I(X_j)=f(A(X_0),\ldots,A(X_n))$

To avoid circularity, inherited attribute are prevented from depending on itself or on attributes to the right in the parse tree $I(X_j)=f(A(X_0),\ldots,A(X_{j-1}))$

The attribute value depends on the parent and those of its sibling nodes (pass semantic information down a parse tree)

3. Each rule has a (possibly empty) set of predicates to check for attribute consistency

A false predicate function value indicates a violation of the syntax or static semantics rules of the language

If all the attribute values in a parse tree have been computed, the tree is said to be fully attributed

Intrinsic attributes are synthesized attributes of a leaf nodes whose values are determined outside the parse tree.

Initially, there are *intrinsic attributes* on the leaves and an un-attribute parse tree has been constructed and computed

Chapter 3

Example of Attribute grammars：

Q1. The rule that the name on the *end* of an Ada procedure must match the procedure name.

String attribute of <proc_name>： <proc_name>.string

Actual string of characters were found by the lexical analyzer

Syntax rule:

<proc_def> → procedure <proc_name>[1]
<proc_body> *end* <proc_name>[2]

Semantic rule:

<proc_name>[1].string = <proc_name>[2].string

Q2. The assignment of variable. The variables on the LHS and RHS of a assignment they need not be the same type.

Example: expressions of the form id + id
- id's can be either int_type or real_type
- types of the two id's must be the same
- type of the expression must match it's expected type

BNF(Syntax rule) ：
1. *<assign> → <var> = <expr>*
2. and 3.<expr> → <var> + <var> | <var>
4. <var> → A | B | C

Attributes:
actual_type - synthesized attributes for <var> and <expr>. A variable the actual type is intrinsic. An expression is determined from the actual types of the child node or children nodes of the <expr> non-terminal
expected_type - inherited attributes for <expr> determined by the type of variable on the LHS of the assignment statement

Chapter 3

Attribute Grammar

1. Syntax rule: <assign> → <var> = <expr>
   Semantic rules:
   <expr>.expected_type ← <var>.actual_type 由LHS type決定

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

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)

   Look-up function looks up a given variable name in the symbol table and returns the variable's type

How are attribute values computed?
1. If all attributes were inherited, the tree could be decorated in top-down order.
2. If all attributes were synthesized, the tree could be decorated in bottom-up order.
3. In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.

Chapter 3

Ex. A=A+B (P. 130 Fig. 3.7 & 3.8)

1. <var>.actual_type ← Look-up(A)   [Rule 4]
2. <expr>.expect_type ← <var>.actual_type   [Rule 1]
3. <var>[2].actual_type ← Look-up(A)   [Rule 4]
<var>[3].actual_type ← Look-up(B)   [Rule 4]
4. <expr>.actual_type ← either int or real   [Rule 2]
5. <expr>.expected_type =<expr>.actual_type is either TRUE or FALSE   [Rule 2]

Characteristics:
Attribute grammar is used to describe the syntax and the static semantics of a PL.

Advantage:
Attribute grammar can be used as the formal definition of a language that can be input to a compiler generation System.

Dis-advantage:
Hard to describe all of the syntax and static semantics of a real PL because its size and complexity.
The number of attributes and semantics rules make such grammars difficult to read and write.
The attribute values on a large parse tree are expensive.

Chapter 3

Dynamic Semantics(meaning)

Describe the syntax is a relative simple
No single widely acceptable notation or formalism for describing semantics

Why need to describe the semantics?
a. Programmers need to know what statements of a language do
b. Compiler writer determine the semantics of a language accroding the English descriptions.
c. A research goal to find a semsntics formalism that can be used by programmers and compiler writers.
d. Program correctness proofs rely on some formal description of the language semantics.

1. *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
- To use operational semantics for a high-level language, a virtual machine is needed
- A *hardware* pure interpreter would be too expensive
- A *software* pure interpreter also has problems:
a. The complexity of HW and OS were used to run the pure intepreter would make actions difficult to understand
b. Such a semantic definition would be machine-dependent
- *A better alternative*: A complete computer simulation
- *The process:*
a. Build a translator (converts statement in L to the chosen low-level language)
b. Build a virtual machine for the low-level language(state changes in virtual machine by translating a statement in the high-level language)

Chapter 3

Ex.
C statement:
 for (expr1; expr2; expr3) { … }

Operational semantics
```
          expr1;
Loop: if expr2 = 0  goto out
         …
         expr3;
         goto loop
Out:  …
```

Simple control statement components:
```
  ident=var
  ident=ident+1
  ident=ident-1
  goto label
  if var relop var goto label      relop: >, >=, <, <=, =, <>
```

Assignment statement
```
  ident=var bin_op var        bin_op: +, -, *, /
  ident=un_op var             un_op: +, -
```

Evaluation of operational semantics:
- Depends on algorithm not mathematics
- Provide an effective means of describing semantics for language users and implement-or
- Good if used informally
- Extremely complex if used formally (e.g., VDL)
  VDL: abstract machine translation rule for PL/I

Chapter 3

Axiomatic Semantics

-Develop a method to prove the correctness of a program
In a proof, each statement of a program is both ***preceded*** and ***followed*** by a logical expression that specifies constraints on program variable
- Based on formal logic (first order predicate calculus)
- *Approach:*
  Define axioms or inference rules for each statement type in the language (to allow transformations of expressions to other expressions)
- The logical expressions are called ***assertions or predicates***
- An assertion before a statement is a ***precondition***
  Precede a statement that states the relationships and constraints among variables at that point in the program

-An assertion following a statement is a ***post-condition***
  Follow a statement that states the new constraints on those variables(p. 135)

-A ***weakest precondition*** is the least restrictive precondition that will guarantee the validity of the associated post-condition

- Pre-post form: {P} statement {Q}

- *An example:* a := b + 1 {a > 1}
  One possible precondition: {b > 10}
  Weakest precondition: {b > 0}
  { b > 0} ⊃ {b > 10} or { b > 0} => {b > 10}

Chapter 3

Program proof process:

The post-condition of the last statement as the the desired results of the program's execution. By working backward through the program, computing the weakest preconditions for each statement until the beginning of the program is reached. At this point, the first precondition states the condition under which the program will compute the desired results

An axiom is a logical statement that is assumed to be true
An inference rule is a method of inferring the truth of one assertion on the basis of the value of other assertions.

- An axiom for assignment statements:

$\{Q_{x\text{->}E}\}$ x = E {Q}

Ex.

a=b/2-1 {a < 10}
{a < 10} → post-condition
b/2-1 < 10 → weakest precondition is computed
{b < 22} → precondition

x=2*y-3 {x > 25}
{x > 25} → post-condition
2*y-3 > 25 → weakest precondition is computed
{y > 14} → precondition

x=x+y-3 {x > 10}
{x > 10} → post-condition
x+y-3 > 10 → weakest precondition is computed
{y > 13-x} → precondition

See p. 137 for statement proof!!!

Chapter 3

- The Rule of Consequence:

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

A post-condition can always be weakened.
A precondition can always be strengthened.

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

where
{x>3} → P, {x>0} → Q, {x>5} → P', {x>0} → Q'

- An inference rule for sequences

- For a sequence S1;S2:
  {P1} S1 {P2}
  {P2} S2 {P3}

the inference rule is:

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

Ex.

y=3*x+1;
x=y+3; {x < 10}
{x < 10} → post-condition
y+3 < 10 → weakest precondition is computed
{y < 7} → precondition for x=y+3;

{ ??? } y=3*x+1; {y < 7}
{y < 7} x=y+3; {x < 10}
{y < 7} → post-condition
3*x+1 < 7 → weakest precondition is computed
{x < 2} → precondition for y=3*x+1;

{x < 2} y=3*x+1; {y < 7}
{y < 7} x=y+3; {x < 10}
{x < 2} ⊃ {x < 10} or {x < 2} => {x < 10}

Chapter 3

- The Rule of Selection:

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

Ex.

if (x>0) y=y-1 else y=y+1 {y > 0}

for then clause y=y-1 {y > 0} → {y > 1}

for else clause y=y+1 {y > 0} → {y > -1}

{y > -1} ⊃ {y > 1} or {y > 1} => {y > -1}

∴ {y > 1} if (x>0) y=y-1 else y=y+1 {y > 0}

- An inference rule for logical pretest loops

Axiomatic description for the loop construct:

{P} while B do S end {Q}

the inference rule is:

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

where I is the loop invariant, B is loop-controlling Boolean expression and S is loop body.

Characteristics of the loop invariant

I must meet the following conditions(check every time):

① P => I (the loop invariant must be true initially)
②{I} B {I} (evaluation of the Boolean must not change the validity of I)
③{I and B} S {I} (I is not changed by executing the body of the loop)
④(I and (not B)) => Q (if I is true and B is false, Q is implied)
⑤ The loop terminates (this can be difficult to prove)

Chapter 3

Ex.

While y <> x do y=y+1 end { y=x}

By inductive hypothesis,

For 0 iteration, the weakest precondition is obvious

For 1 iteration,

wp(y=y+1, {y=x}) = { y+1=x } or { y=x-1}

For 2 iteration,

wp(y=y+1, {y=x-1}) = { y+1=x-1 } or { y=x-2}

For 3 iteration,

wp(y=y+1, {y=x-2}) = { y+1=x-2 } or { y=x-3}

Combine all of these, we get <u>{ y <= x }</u> as the loop invariant

① Because P=I, P => I
② I is unaffected by the evaluation of the loop Boolean expression, which is y <> x. This expression can't affect I
③ {I and B} S {I}
  {y <= x and y <> x} y=y+1 {y <= x}
  Apply the assign axiom to y=y+1 {y <= x}
  {y+1 <= x} ≅ {y < x} can be implied by {y <= x and y <> x}
④ {I and (not B)} => Q
  {y <= x and not (y <> x)} => {y=x}
  {y <= x and y = x} => {y=x}
⑤ {y <= x} While y <> x do y=y+1 end { y=x}
  The loop does terminate. The precondition guarantee that y initially is not greater than x. The loop body increase y with each iteration until y is equal to x.

Chapter 3

Ex.

While s > 1 do s=s/2 end {s=1}

By inductive hypothesis,

For 0 iteration, the weakest precondition is {s=1}

For 1 iteration,

wp(s=s/2, {s=1}) = { s/2=1} or {s=2}

For 2 iteration,

wp(s=s/2, {s=2}) = { s/2=2} or {s=4}

For 3 iteration,

wp(s=s/2, {s=4}) = { s/2=4} or {s=8}

Combine all of these, we get { s is a nonnegative power of 2} as the loop invariant I

① Because P=I, P => I

② Consider the precondition {s > 1}, The logical statement {s > 1} While s > 1 do s=s/2 end {s=1} can be easy proven

…

Because of the difficulty of proving loop termination, the requirement is often ignored.

If the loop termination can be shown, the axiomatic description of the loop is called total correctness.

If the other conditions can be met but termination is not guaranteed,, it is called partial correctness.

- The loop invariant I is a weakened version of the loop post-condition, and it is also a precondition.

- I must be weak enough to be satisfied prior to the beginning of the loop, but when combined with the loop exit condition, it must be strong enough to force the truth of the post-condition

Evaluation of axiomatic semantics:

1. Developing axioms or inference rules for all of the statements in a language is difficult
2. It is a good tool for correctness proofs, and an excellent framework for reasoning about pograms, but it is not as useful for language users and compiler writers

Chapter 3

Denotational Semantics

-Based on recursive function theory

-A thorough discussion of the use of denotational semantics to describe the semantics of PLs is long and complex

- The most abstract semantics description method
- Originally developed by Scott and Strachey (1970)
- The process of building a de-notational spec for a language:
  1. Define a mathematical object for each language entity
  2. Define a function that maps instances of the language entities onto instances of the corr. mathematical objects

-There are rigorous way of manipulating mathematical objects but not for PL constructs

-The difficulty is to create the objects and mapping functions

- Mathematical objects denoted the meaning of their corresponding syntactic entities ➔ de-notational semantics

Ex. CFG production: <bin_num> → 0|1|<bin_num>(0|1)

The semantic value of the objects be N, the set of non-negative decimal integer value. The semantic function name M_{bin} maps the syntactic objects to the objects in N.

The function defined as followed:

M_{bin}('0')=0 , M_{bin}('1')=1 ,

M_{bin}(<bin_num>'0')=2* M_{bin}(<bin_num>),

M_{bin}(<bin_num>'1')=2* M_{bin}(<bin_num>)+1

When a program reads a number as a string, it must be converted to a mathematical number before it can be used as a number in the program(p. 146, 147)

Chapter 3

Ex. Similar example of Decimal Numbers

CFG production:

<dec_num> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
 | <dec_num> (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)

De-notational mapping for the above production is followed:

M_{dec}('0') = 0, M_{dec} ('1') = 1, …, M_{dec} ('9') = 9
M_{dec} (<dec_num> '0') = 10 * M_{dec} (<dec_num>)
M_{dec} (<dec_num> '1') = 10 * M_{dec} (<dec_num>) + 1
…
M_{dec} (<dec_num> '9') = 10 * M_{dec} (<dec_num>) + 9

- The denotational semantics of a program could be defined in term of state change
- The difference between de-notational and operational semantics: In operational semantics, the state changes are defined by <u>coded algorithms</u>; In de-notational semantics, they are defined by rigorous <u>mathematical functions</u>
- The *state* <u>s</u> of a program can be represented as a set of <u>ordered pairs</u> as followed:

  s = {<i_1, v_1>, <i_2, v_2>, …, <i_n, v_n>}

  i is the name of a variable and the associated V are the current values of those variables
- Let VARMAP be a function that, when given a variable name and a state, returns the current value of the variable VARMAP(i_j, s) = v_j
- Any of the v can have the special value <u>*undef*</u>, which indicates that its associated variable is <u>*undefined.*</u>

Chapter 3

1. Expression BNF description:

<expr> → <dec_num> | <var> | <binary_expr>
<binary_expr> → <left_expr> <operator> <right_expr>
<operator> → + | *

A variable has an undefined value is considered.

Δ= is to define the mathematical functions, s is the state.

```
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)
```

2. Assignment Statements

```
M_a(x := E, s) Δ= if M_e(E, s) = error  then error
               else s' = {<i_1',v_1'>,<i_2',v_2'>,...,<i_n',v_n'>},
                    where for j = 1, 2, ..., n,
                          v_j' = VARMAP(i_j, s) if i_j <> x;
                       = M_e(E, s) if i_j = x
```

Chapter 3

3 Logical Pretest Loops

```
M_l(while B do L, s) Δ= if M_b(B, s) = undef
                            then error
                            else if M_b(B, s) = false
                                then s
                                else if M_sl(L, s) = error
                                    then error
                                    else M_l(while B do L, M_sl(L, s))
```

-The meaning of the loop is the value of the program variables after the statements in the loop have been executed the prescribed number of times, assuming there have been no errors

-In essence, the loop has been converted from iteration to recursion, where the recursive control is mathematically defined by other recursive state mapping functions

-Recursion, when compared to iteration, is easier to describe with mathematical rigor

Evaluation of denotational semantics:

- Can be used to prove the correctness of programs
- Provides a rigorous way to think about programs
- Can be an aid to language design
-Has been used in compiler generation systems
-Because of the complexity of de-notational descriptions, they are of little use to language users