Chapter 3
Context-Free Grammars and Parsing

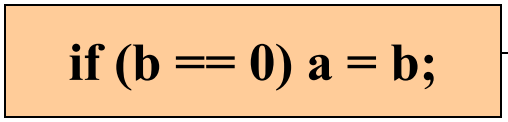
2022 Spring&Summer

Outline

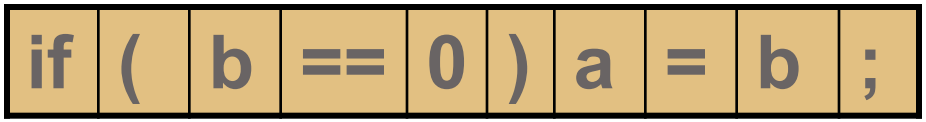
- Context-Free Grammars (CFGs)
- Derivations
- Parse trees and abstract syntax
- Ambiguous grammars

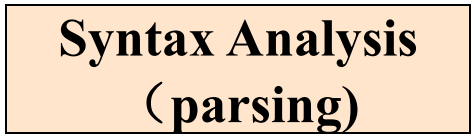
Where We Are

Source code
(character stream)

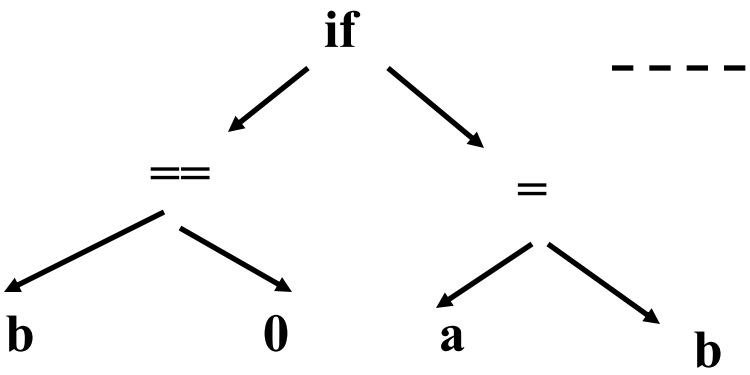


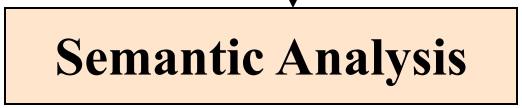
Token stream





Abstract Syntax Tree (AST)





3.1 the parsing process

The parser may be viewed as a *function* that takes as its input the sequence of tokens produced by the scanner and produces as its output the syntax tree.

1、The parser calls a scanner procedure such as *getToken* to fetch the next token from the input

2、In a single-pass compiler the parser will incorporate all the other phases of a compiler,

parse( ) ;

3、More commonly, a compiler will be multi-pass the further passes will use the syntax tree as their input.

3.1 the parsing process

- The structure of the syntax tree : dependent on the particular syntactic structure of the language.
- This tree is usually defined as a dynamic data structure

  each node consists of a record whose fields include the attributes needed for the remainder of the compilation process.

3.1 The parsing process

One problem that is more difficult: the treatment of errors.

1、Error in the scanner

Generate an error token and consume the offending character.

2、Error in the parser

Report an error message

Recover from the error and continue parsing (to find as many errors as possible).

Sometimes, a parser may perform error repair.

3.2 Context-free grammars

- A context-free grammar is a specification for the syntactic structure of a programming language.
- A context-free grammar involves recursive rules.

  For example:

  $exp \rightarrow exp\ op\ exp \mid (exp) \mid \textbf{\textit{number}}$

  $op \rightarrow + \mid - \mid *$

3.2.1 Comparison to regular expression notation

The context-free grammar:

$exp \rightarrow exp\ op\ exp \mid (exp) \mid \mathbf{number}$

$op \rightarrow + \mid - \mid *$

The regular expression:

$number = digit\ \ digit^*$

$digit = \ 0|1|2|3|4|5|6|7|8|9$

3.2.2 Specification of context-free grammar rules

- A Context-Free Grammar is a 4-tuple $(V,\Sigma,S,\rightarrow)$, where
  - V is a finite set of nonterminal symbols
  - Σ is a finite set of terminal symbols
  - $S \in V$ is a distinguished nonterminal, the start symbol
  - $\rightarrow \subseteq V \times (V \cup \Sigma)^*$ is a finite relation, the productions
- Context Free Grammar is abbreviated CFG

3.2.2 Specification of context-free grammar rules

- The choice operation given by the vertical bar metasymbol is also not necessary in grammar rules.

  For example, simple expression grammar could be written as follows:

  exp → *exp op exp*

  exp → (*exp*)

  exp → ***number***

  op → +

  op → –

  op → *

3.2.2 Specification of context-free grammar rules

- Use uppercase letters for structure names and lowercase letters for individual token symbols (which often are just single characters).

$$E \rightarrow E\ O\ E \mid (E) \mid n$$

$$O \rightarrow + \mid - \mid *$$

3.2.3 Derivations and the language defined by a grammar

- Let G = (V,Σ,S,→) be a CFG. The "directly derives" relation (⇒) is defined as { (αAγ, αβγ) | A→β }.
- Example

  Let G be the grammar with productions S → aSbS | ε

  S ⇒ aSbS

  aSbS ⇒ aaSbSbS

  aaSbSbS ⇒ aabSbS

  aabSbS ⇒ aabbS

  aabbS ⇒ aabbaSbS

  aabbaSbS ⇒ aabbabS

  aabbabS ⇒ aabbab

3.2.3 Derivations and the language defined by a grammar

- **Derivation:** Grammar rules determine the legal strings of token symbols by means of derivations.
- A **derivation** is a sequence of replacements of structure names by choices on the right-hand sides(RHS) of grammar rules.
- A **derivation** begins with a single structure name and ends with a string of token symbols.
- At each step in a derivation, a single replacement is made using one choice from a grammar rule.

nonterminal is LHS of production
string is RHS of production

3.2.3 Derivations and the language defined by a grammar

Figure 3.1 : a derivation

- *exp* → *exp op exp* | *(exp)* | ***number***
- *op* → + | – | *

(1) *exp* => *exp op exp* [*exp* → *exp op exp*]

(2)=> *exp op* ***number*** [*exp* → number]

(3)=> *exp* * ***number*** [*op*→ *]

(4)=> (*exp*) * ***number*** [*exp*→ (*exp*)]

(5)=>{ *exp op exp*) * ***number*** [*exp* → *exp op exp*]

(6)=> *(exp op* number) * ***number*** [*exp*→ number]

(7)=> *(exp* - number) * ***number*** [*op* → -]

(8)=> (number - number) * ***number*** [*exp* → number]

3.2.3 Derivations and the language defined by a grammar

- Derivation steps use a different arrow from the arrow metasymbol in the grammar rules.

$$L(G) = \{\ s \mid exp \Rightarrow^* s\ \}$$

The set of all strings of token symbols obtained by derivations from the exp symbol is the language defined by the grammar of expressions.

3.2.3 Derivations and the language defined by a grammar

1. *G* represents the expression grammar
2. *s* represents an arbitrary string of token symbols (sometimes called a sentence)
3. The symbol s =>* stand for a derivation consisting of a sequence of replacements as described earlier.
4. Grammar rules are sometimes called productions because they "produce" the strings in *L(G)* via derivations.

3.2.3 Derivations and the language defined by a grammar

- **the start symbol:** the most general structure is listed first in the grammar rules.
- **nonterminals:** structure names are also called nonterminals, since they always must be replaced further on in a derivation .
- **terminals**: symbols in the alphabet are called terminals, since they terminate a derivation. Terminals are usually tokens in compiler applications.

3.2.3 Derivations and the language defined by a grammar

- The grammar for a programming language often defines a structure called ***program***
- Language of this structure is the set of all syntactically legal programs of the programming language.
- For example : a BNF for *pascal* language

program → *program-heading; program-block*

program-heading → ....

program-block → .....

3.2.3 Derivations and the language defined by a grammar

- Example : Consider the following extremely simplified grammar of statements:

  Statement → *if-stmt* | ***other***

  if-stmt → **if** (*exp*) *statement*

  | **if** (*exp*) *statement* **else** *statement*

  exp → 0 | 1

- Examples of strings in this language are

  if (0) other

  if (1) other

  if (0) other else other

  if (1) other else other

  if (0) if (0) other

  if (0) if (1) other else other

  if (1) other else if (0) other else other

3.2.3 Derivations and the language defined by a grammar

Recursion: the grammar rule $A \rightarrow A\ a \mid a$

or the grammar rule $A \rightarrow a\ A \mid a$

- Generates the language $\{a^n \mid$ n an integer $>=1\ \}$ (the set of all strings of one or more a's)
- (the regular expression a+)
- The string *aaaa* can be generated by the first grammar rule with the derivation

$A \Rightarrow Aa \Rightarrow Aaa \Rightarrow Aaaa \Rightarrow aaaa$

3.2.3 Derivations and the language defined by a grammar

- **Left Recursive:** the nonterminal *A* appears as the first symbol on the right-hand side of the rule defining *A*.

- **Right Recursive:** the nonterminal *A* appears as the last symbol on the right-hand side of the rule defining *A*.

3.2.3 Derivations and the language defined by a grammar

- Consider a rule of the form: $A \rightarrow A\alpha \mid \beta$

  where α and β represent arbitrary strings and β does not begin with A.
- This rule generates all strings of the form β, $\beta\alpha$, $\beta\alpha\alpha$, $\beta\alpha\alpha\alpha$, ...
- This grammar rule is equivalent in its effect to the regular expression $\beta\alpha^*$.
- $A \rightarrow \alpha A \mid \beta$ generates all strings β, $\alpha\beta$, $\alpha\alpha\beta$, $\alpha\alpha\alpha\beta$, ...

3.2.3 Derivations and the language defined by a grammar

- To generate the same language as the regular expression *a** we must have a notation for a grammar rule that generates the empty string (since the regular expression *a** matches the empty string).

  empty →

- Use the *epsilon* metasymbol for the empty string (similar to its use in regular expressions):

  empty → ε

3.2.3 Derivations and the language defined by a grammar

- Such a grammar rule is called an ε-production (an "epsilon production").
- A grammar that generates a language containing the empty string must have at least one *ε-production*.
- A grammar that is equivalent to the regular expression a^* as

  $A \rightarrow A\,a \mid \varepsilon$ *or* $A \rightarrow a\,A \mid \varepsilon$
- Both grammars generate the language

  $\{ a^n \mid n \text{ an integer} >= 0\} = L(a^*)$.

3.2.3 Derivations and the language defined by a grammar

- Example :

  statement → *if-stmt* | ***other***

  if-stmt → **if (** *exp* **)** *statement else-part*

  else-part → **else** *statement* | ε

  exp → **0 | 1**

  the ε-production indicates that the structure *else-part* is optional.

3.2.3 Derivations and the language defined by a grammar

- Example : $A \rightarrow (A)\, A \mid \varepsilon$

  This grammar generates the strings of all "balanced parentheses."

  For example, the string *(( ) (( ))) ( )*

  A => (A) A => (A)(A)A => (A)(A) =>(A)( ) =>
  ((A)A)( ) =>(( )A)() => (( ) (A)A) () => (( )(A))( )
  => (( )((A)A))( ) => (( )(( )A))( ) => (( )(( )))( )

3.2.3 Derivations and the language defined by a grammar

- Example : Consider the following grammar *G* for a sequence of statements:

  stmt-sequence → *stmt* ; *stmt-sequence* | *stmt*

  stmt → **s**

- This grammar generates sequences of one or more statements separated by semicolons

  $L(G)$= { **s**, **s;s**, **s;s;s**... }

3.2.3 Derivations and the language defined by a grammar

If we want to allow statement sequences to also be empty, we could write the following grammar *G′*:

stmt-sequence → *stmt* ; *stmt-sequence* | ε

stmt → **s**

L(*G′*)= { ε, **s**;, **s**;**s**;, **s**;**s**;**s**; , ... }

3.2.3 Derivations and the language defined by a grammar

- This turns the semicolon into a statement **terminator** rather than a statement **separator**
- Allow statement sequences to be empty, but retain the semicolon as a statement separator.

stmt-sequence → *nonempty-stmt-sequence* | ε

nonempty-stmt-sequence → *stmt* **;** *nonempty-stmt-sequence* | *stmt*

stmt → **s**

- Care must be taken in the placement of the ε-production when constructing optional structures.

3.3 Parse trees and abstract syntax trees

- Derivations do not uniquely represent the structure of the strings they construct.
- There are many derivations for the same string.
  - the string of tokens *(number - number) * number* from our simple expression grammar using the derivation in Figure 3.1.
  - A second derivation for this string is given in Figure 3.2. (Page 107)

3.3.1 Parse trees

(1)	*exp* => *exp op exp*	[*exp* → *exp op exp*]
(2)	=> *exp op* ***number***	[*exp* → ***number***]
(3)	=> *exp* * ***number***	[*op*→ *]
(4)	=> (*exp*) * ***number***	[*exp*→ (*exp*)]
(5)	=>(*exp op exp*) * ***number***	[*exp* → *exp op exp*]
(6)	=> *(exp op* number) * ***number***	[*exp*→ ***number***]
(7)	=> *(exp* - number) * ***number***	[*op* → -]
(8)	=> (number - number) * ***number***	[*exp* → ***number***]

3.3.1 Parse trees

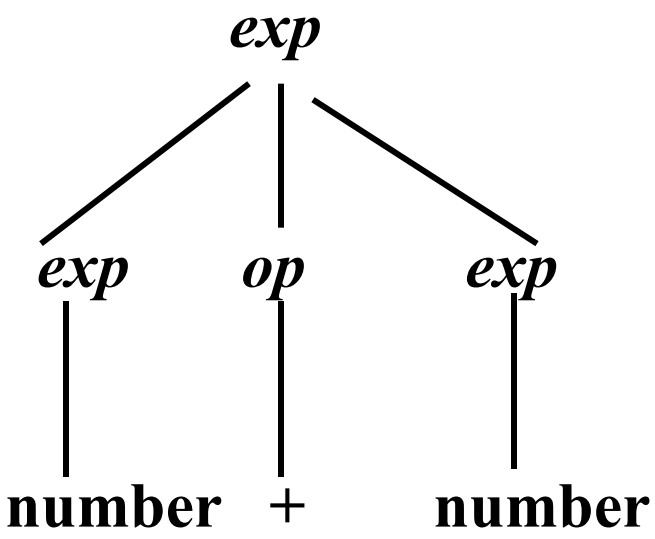
A parse tree corresponding to a derivation is a labeled tree

- the interior nodes are labeled by nonterminals,
- the leaf nodes are labeled by terminals
- the children of each internal node represent the replacement of the associated nonterminal in one step of the derivation.

.

3.3.1 Parse trees

corresponds to the parse tree



3.3.1 Parse trees

The above parse tree is corresponds to the three derivations:

1 (1) *exp* => *exp op exp*
(2) => ***number*** *op exp*
(3) => ***number*** + *exp*
(4) => ***number*** + ***number***

leftmost derivation

2 (1) *exp* => *exp op exp*
(2) => *exp op* ***number***
(3) => *exp* + ***number***
(4) => ***number*** + ***number***

rightmost derivation

3 (1) *exp* => *exp op exp*
(2) => *exp* + *exp*
(3) => ***number*** + *exp*
(4) => ***number*** + ***number***

neither leftmost nor rightmost derivation

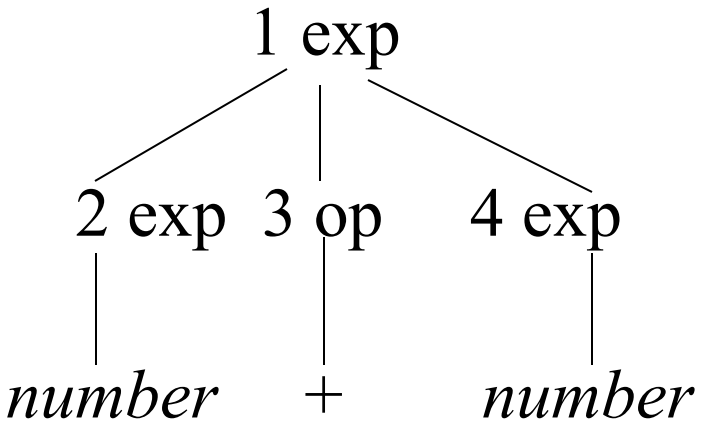
3.3.1 Parse trees

It is possible to distinguish particular derivations that are uniquely associated with the parse tree.

- **A leftmost derivation:** a derivation in which the leftmost nonterminal is replaced at each step in the derivation.
  - Corresponds to the preorder numbering of the internal nodes of its associated parse tree.
- **A rightmost derivation:** a derivation in which the rightmost nonterminal is replaced at each step in the derivation.
  - Corresponds to the postorder numbering of the internal nodes of its associated parse tree.

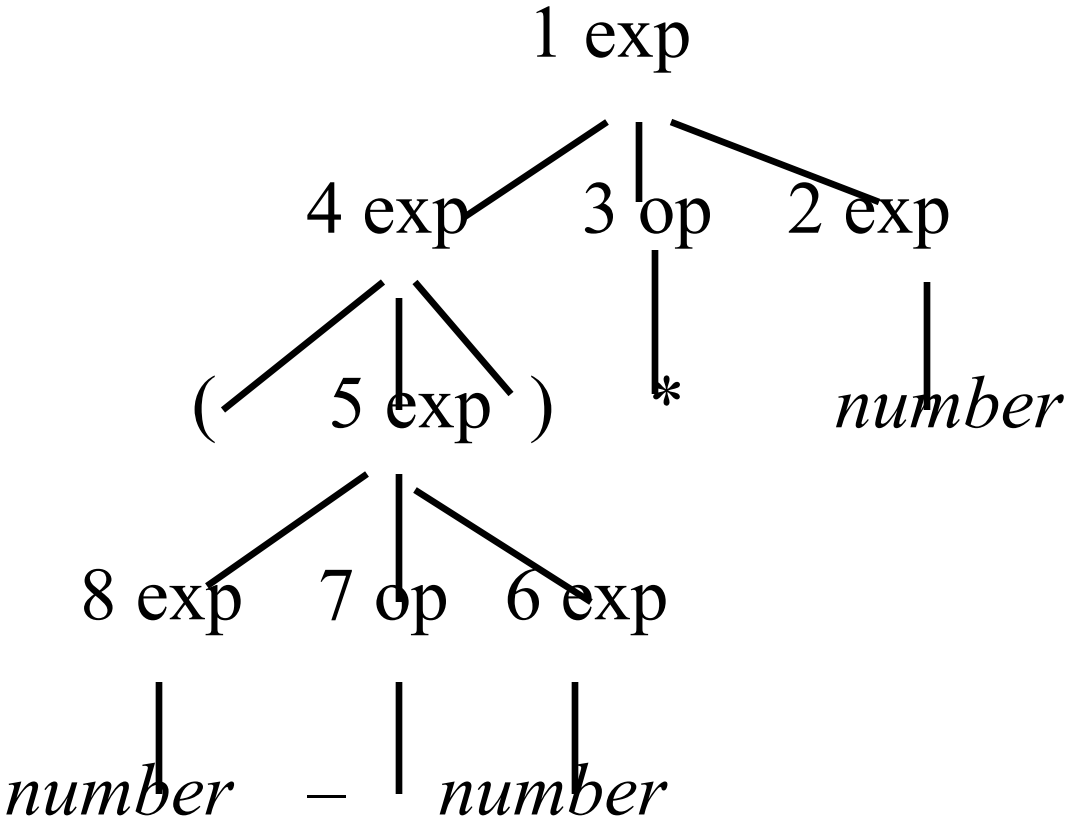
3.3.1 Parse trees

The parse tree corresponds the first derivation



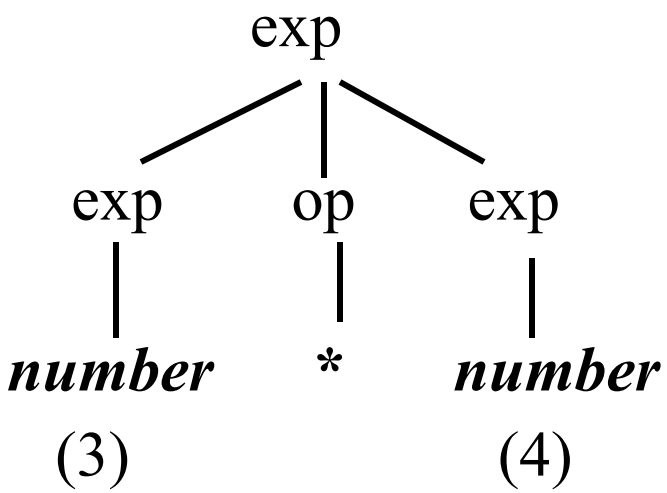
3.3.1 Parse trees

The parse tree for the arithmetic expression (34-3)*42



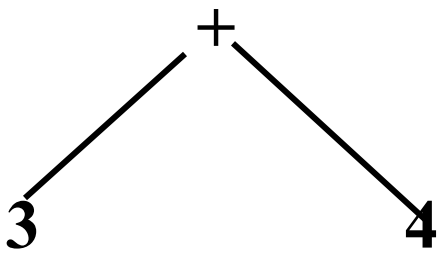
3.3.2 Abstract syntax trees

- The principle of syntax-directed translation states that the meaning, or semantics, of the string *3+4* should be directly related to its syntactic structure as represented by the parse tree.
- The principle of syntax-directed translation means that the parse tree should imply that the value *3* and the value *4* are to be added.



3.3.2 Abstract syntax trees

A much simpler way to represent this same information, namely, as the tree



3.3.2 Abstract syntax trees

- Abstract Syntax Trees, or Syntax Trees:

  1. Such trees represent abstractions of the actual source code token sequences, and *the token sequences cannot be recovered from them (unlike parse trees)*.

  2. A parse tree is a representation for the structure of ordinary called concrete syntax when comparing it to abstract syntax.

  3. Abstract syntax can be given a formal definition using a BNF-like notation, just like concrete syntax.

3.3.2 Abstract syntax trees

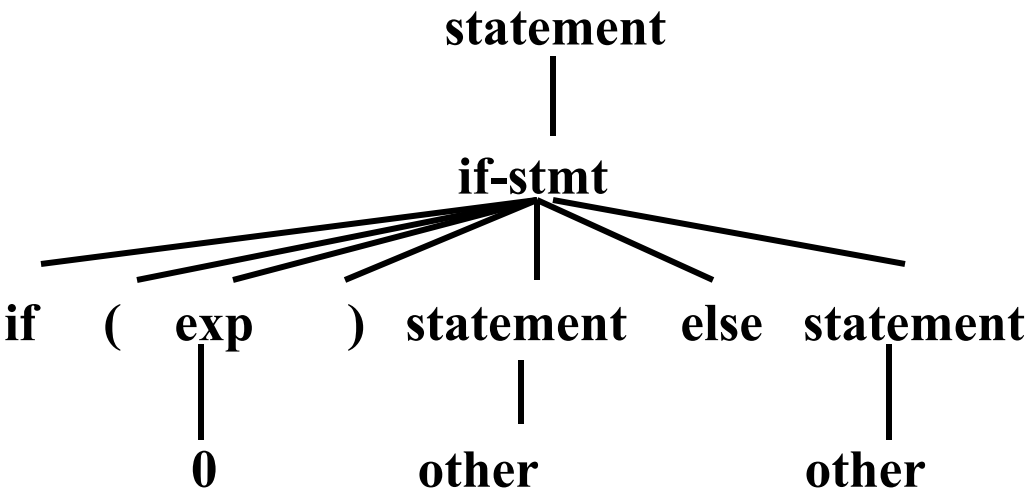
Example 3.8: Consider the grammar for simplified if-statements of Example 3.4:

statement → *if-stmt* | ***other***

if-stmt → **if** (*exp*) *statement*

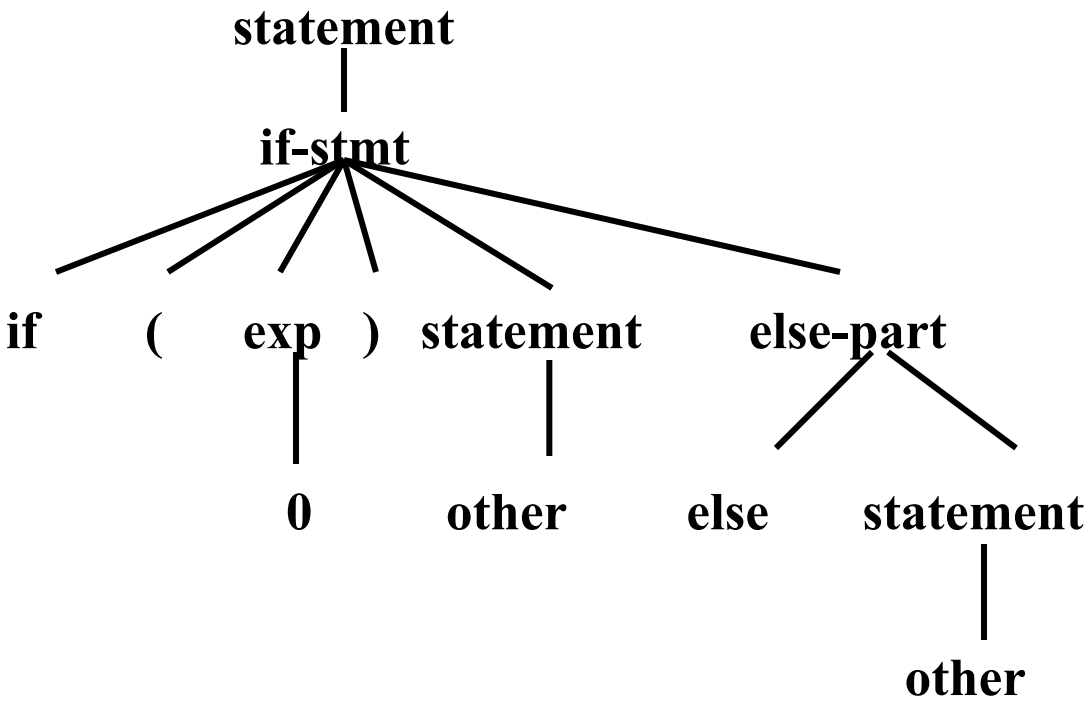
| **if (** *exp* **)** *statement* **else** *statement*

exp → **0 | 1**



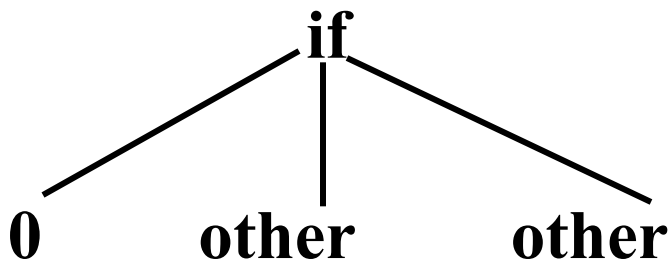
3.3.2 Abstract syntax trees

statement → *if-stmt* | ***other***
if-stmt → **if (** *exp* **)** *statement else-part*
else-part → **else** *statement* | ε
exp → **0 | 1**



3.3.2 Abstract syntax trees

A syntax tree for the previous string:



3.3.2 Abstract syntax trees

A set of C declarations that would be appropriate for the structure of the statements and expressions in this example is as follows:

```
typedef enum  {ExpK,StmtK} NodeKind;
typedef enum  {Zero,One} ExpKind;
typedef enum {IfK,OtherK) StmtKind;
typedef struct streenode
        { NodeKind kind;
          ExpKind ekind;      .
          StmtKind skind;
          struct streenode
           *test,*thenpart,*elsepart;
        } STreeNode;
typedef STreeNode * SyntaxTree;
```

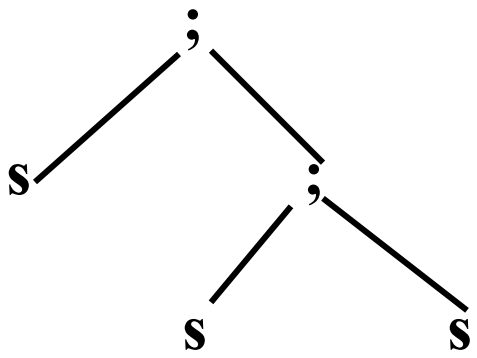
3.3.2 Abstract syntax trees

Example 3.9: Consider the grammar of a sequence of statements separated by semicolons from Example 3.7:

stmt-sequence $\rightarrow$ ***stmt*** **;** ***stmt-sequence*** | ***stmt***

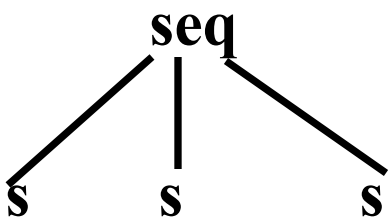
stmt $\rightarrow$ **s**

A possible syntax tree for this same string is:



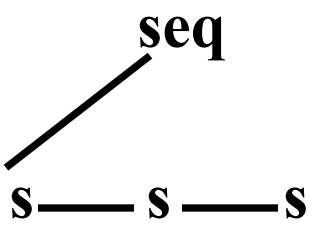
3.3.2 Abstract syntax trees

To bind all the statement nodes in a sequence together with just one node



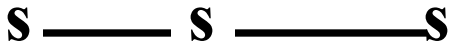
The problem: a *seq* node may have an arbitrary number of children

The solution: use the standard *leftmost-child right-sibling* representation for a tree (presented in most data structures texts).



3.3.2 Abstract syntax trees

With this arrangement, we can also do away with the connecting *seq* node, and the syntax tree then becomes simply:



Homework of Chapter 3

- 3.2 Given the grammar A -> AA | (A) | ε ,
- 1). Describe the language it generates,
- 2). Show that it is ambiguous.
- 3.3 Given the grammar
- exp -> exp addop term | term
- addop -> + | -
- term -> term mulop factor | factor
- mulop -> *
- factor -> (exp) | number
- write down leftmost derivations, parse trees, and abstract syntax trees for the following expressions:
- a. 3+4*5-6 b. 3*(4-5+6) c. 3-(4+5*6)

Homework of Chapter 3

- 3.4 The following grammar generates all regular expressions over the alphabet of letters (we have used quotes to surround operators,since the vertical bar is an operator as well as a metasymbol):
- rexp -> rexp “ | ” rexp
- | rexp rexp
- | rexp “*”
- | “(“ rexp ”)”
- | letter
- a.Give a derivation for the regular expression *(ab|b)** using this grammar.
- b.Show that this grammar is ambiguous.
- c.Rewrite this grammar to establish the correct precedences for the operators.
- d.What associativity does your answer in part (c) give to the binary operators? Why?