

COMP1200 - Perspectives on Computing

These slides were based on those of C Johnson, E McCreath and W Liang.

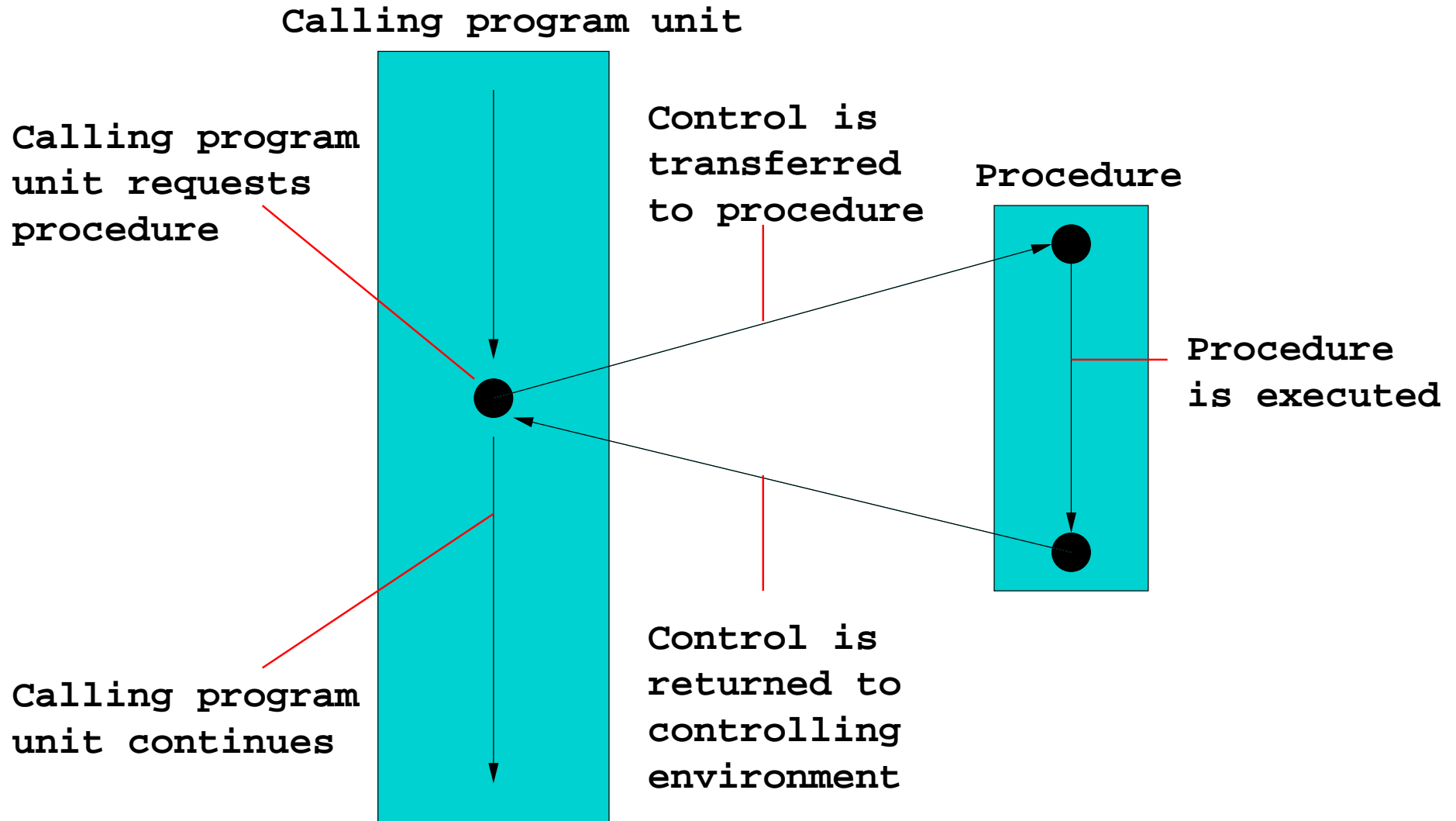
Outline

- Procedural Units
- Implementation

Procedural Units

- It is advantageous to divide large programs into procedural units
- Using procedures is a major technique for obtaining a modular representation of a program
- A procedure is a set of instructions for performing a task that can be used as an abstract tool by other program units
- Transferring control to a procedure is *calling* or *invoking* the procedure
- A procedure is a miniature program

Procedural Units



Procedural Units

- A variable declared by a procedure is a *local variable*
- The portion of a program in which a variable can be referenced is the *scope* of the variable
- The scope of a local variable is the procedure in which it is declared
- Variables with unrestricted scope are called *global variables*

Procedural Units

- The definition of a procedure begins with the procedure's *header*
- This identifies the name of the procedure
- This is followed by statements that define the procedure's details

Parameters

- Procedures are written using generic terms that are made specific when the procedure is applied
 - **procedure** `Sort(List)`
- *List* is generic as opposed to being specific
- Such generic terms are called *parameters*
- The terms used within the procedure are *formal parameters*
- The precise meanings assigned to these formal parameters when the procedure is applied are called *actual parameters*

Procedural Units

```
void ProjPop (float Growth)
{
    int Year;

    Pop[0] = 100.0;
    for (Year = 0; Year ≤ 10; Year++)
        Pop[Year+1] = Pop[Year] + Pop[Year] * Growth;
}

ProjPop(0.03)
```

Parameters

- When there is more than one parameter involved, the actual parameters are associated, entry by entry, with the formal parameters listed in the procedure's header

```
procedure PrintCheque(Payee, Amount)
```

```
PrintCheque('John smith', 100)
```

Parameters

- In some languages, a duplicate of the data represented by the actual parameters is produced
- Any alteration made to the data is only made in the duplicate
- Such parameters are *passed by value*
- A more efficient technique is to give the procedure direct access to the parameters
- This is achieved by telling the procedure the address of the parameters
- Such parameters are *passed by reference*

Parameter

```
void ProjPop (float Growth)
{
    int Year;
    Growth = Growth * 2;
    Pop[0] = 100.0;
    for (Year = 0; Year ≤ 10; Year++)
        Pop[Year+1] = Pop[Year] + Pop[Year] * Growth;
}
```

`P=0.03; ProjPop(P); printf('‘p = %f\n’’,P) → 0.03`

Parameter

```
void ProjPop (float Growth)
{
    int Year;
    Growth = Growth * 2;
    Pop[0] = 100.0;
    for (Year = 0; Year ≤ 10; Year++)
        Pop[Year+1] = Pop[Year] + Pop[Year] * Growth;
}
```

`P=0.03; ProjPop(&P); printf('‘p = %f\n’',P) → 0.06`

Functions

- If the purpose of a procedure is to produce a value, then a procedure might be implemented as a *function*
- As a consequence of executing a function, a value will be computed and sent back (returned) to the calling unit

```
ProjJanSales = EstJanSales(January)
```

```
if (Sales < EstJanSales(January)) then ...
```

Procedural Units

```
float CylinderVol (float Radius, float height)
{
    float Vol;

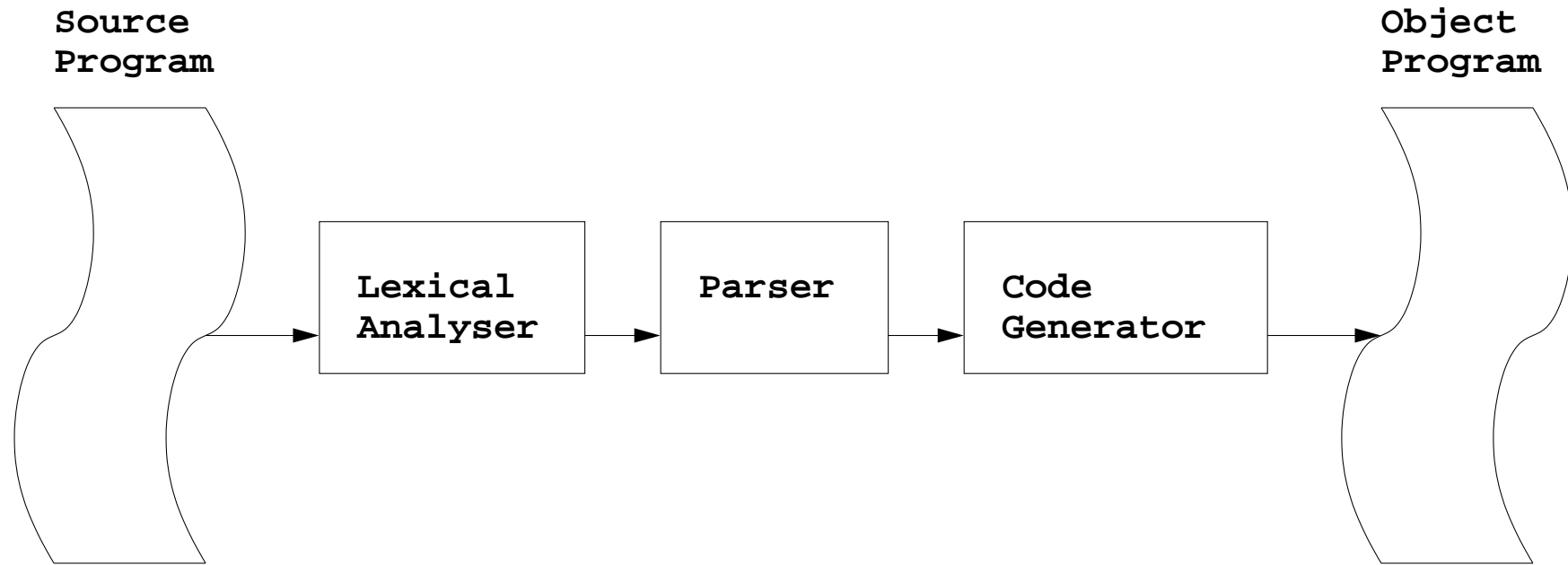
    Vol = 3.14 * Radius * Radius * Height;

    return Vol;
}
```

Translation

- The process of converting a program from one language to another is called *translation*
- The program in its original form is the *source program*
- The translated version is the *object program*
- The translation process consists of three activities:
 - lexical analysis - performed by the lexical analyser
 - parsing - performed by the parser
 - code generation - performed by the code generator

Translation



Lexical Analysis

- Lexical analysis is the process of recognising which strings of symbols from the source program represent a single entity e.g. 153, word
- The lexical analyser reads the source program symbol by symbol
- It identifies which groups of symbols represent single units

Lexical Analysis

- The lexical analyser classifies the units according to whether they are
 - numeric values
 - words
 - arithmetic operators
 - etc.
- As each unit is classified, the lexical analyser encodes the unit and its classification in a package known as a *token* and hands the token to the parser

Parsing

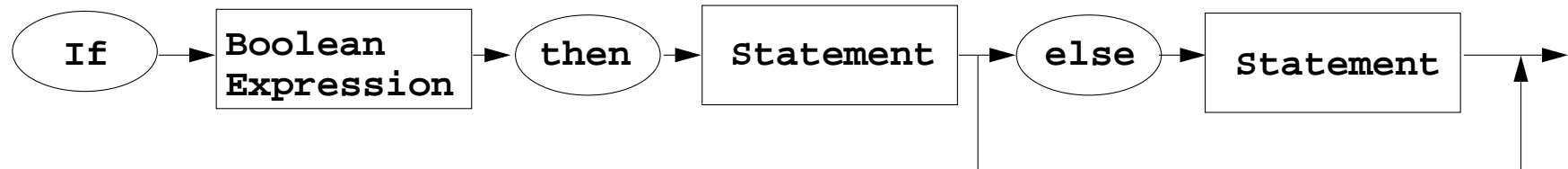
- The parser views the program in terms of lexical units (tokens) rather than individual symbols
- It is the parser's job to group these units into statements
- Parsing is the process of identifying the grammatical structure of the program and recognising the role of each component
- Today, most programming languages are *free-format languages*, meaning that the positioning of statements is not critical

Parsing

- For a machine to parse a program written in a free-format language, the syntax of the language must be designed so that the structure of a program can be identified regardless of the spacing used in the source program
- Most free-format languages use punctuation marks such as semicolons to mark the ends of statements
- They also use *key words* (such as if, then and else) to mark the beginning of individual phrases
- These key words are often *reserved words*

Parsing

- The parsing process is based on a set of rules that define the syntax of the programming language
- Collectively, these rules are called a *grammar*
- One way of expressing these rules is by means of *syntax diagrams*

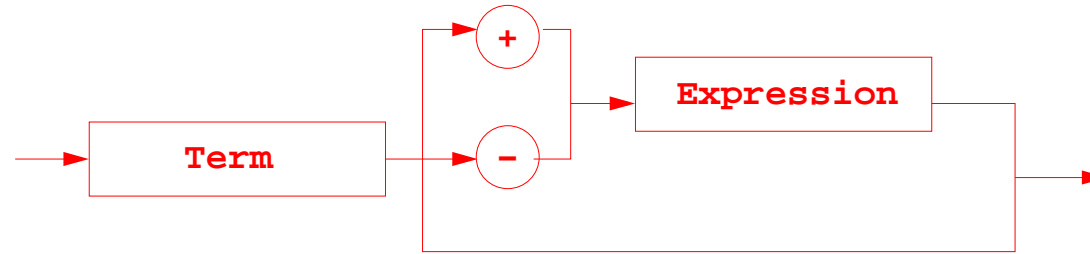


Syntax Diagrams

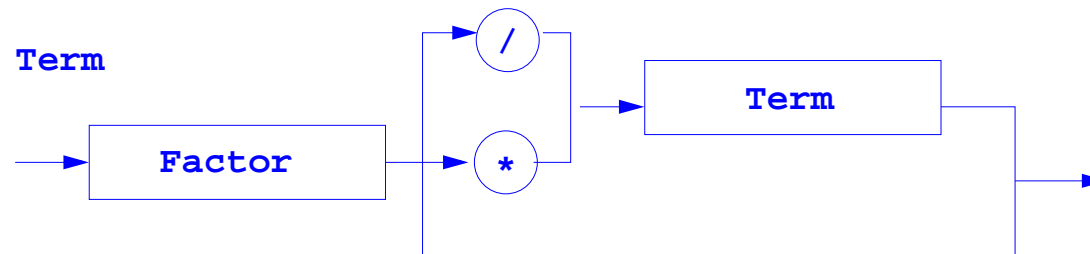
- An *if-then-else* structure begins with the word *if*, followed by a *Boolean expression*, followed by the word *then*, followed by a *Statement*
- This combination might or might not be followed by the word *else* and a *Statement*
- Actual terms in an *if-then-else* statement are enclosed in ovals and are called terminals
- Terms that require further description, such as *Boolean expression* and *Statement*, are enclosed in rectangles and are called non-terminals

An Algebraic Expression

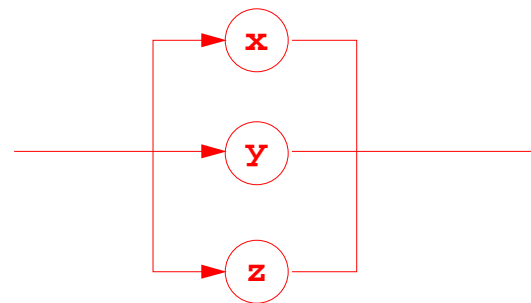
Expression



Term



Factor



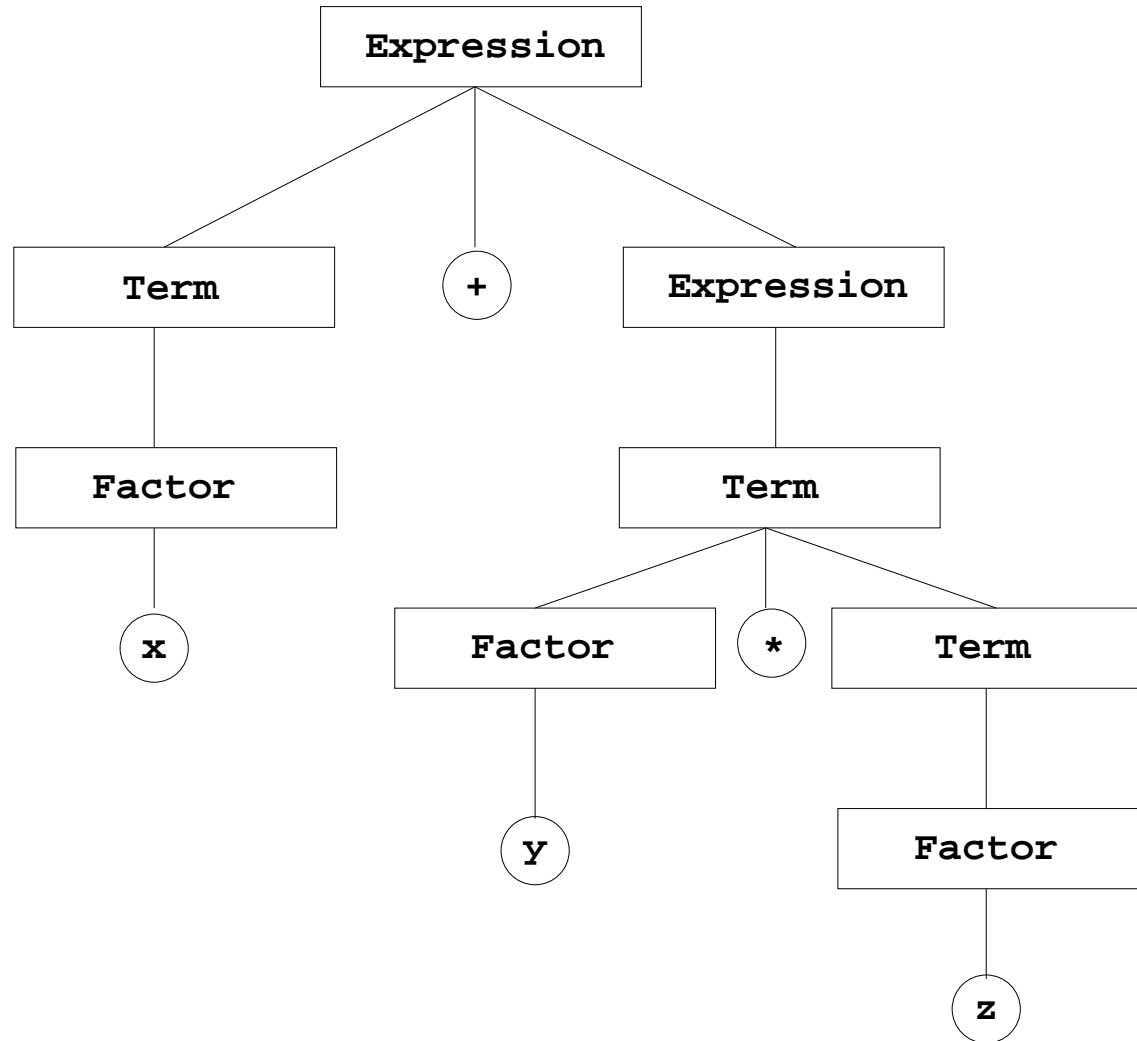
Parse Trees

- The manner in which a particular string conforms to a set of syntax diagrams can be represented in a pictorial form by a *parse tree*
- The tree starts at the top with the non-terminal *Expression*
- At each level it shows how the non-terminals at that level are decomposed until the symbols in the string itself are obtained

Parse Trees

- An *Expression* can be decomposed as a *Term*, followed by the $+$ symbol, followed by an *Expression*
- The *Term* can be decomposed as a *Factor* (which turns out to be the symbol x)
- The final *Expression* can be decomposed as a *Term* (which turns out to be $y \times z$)

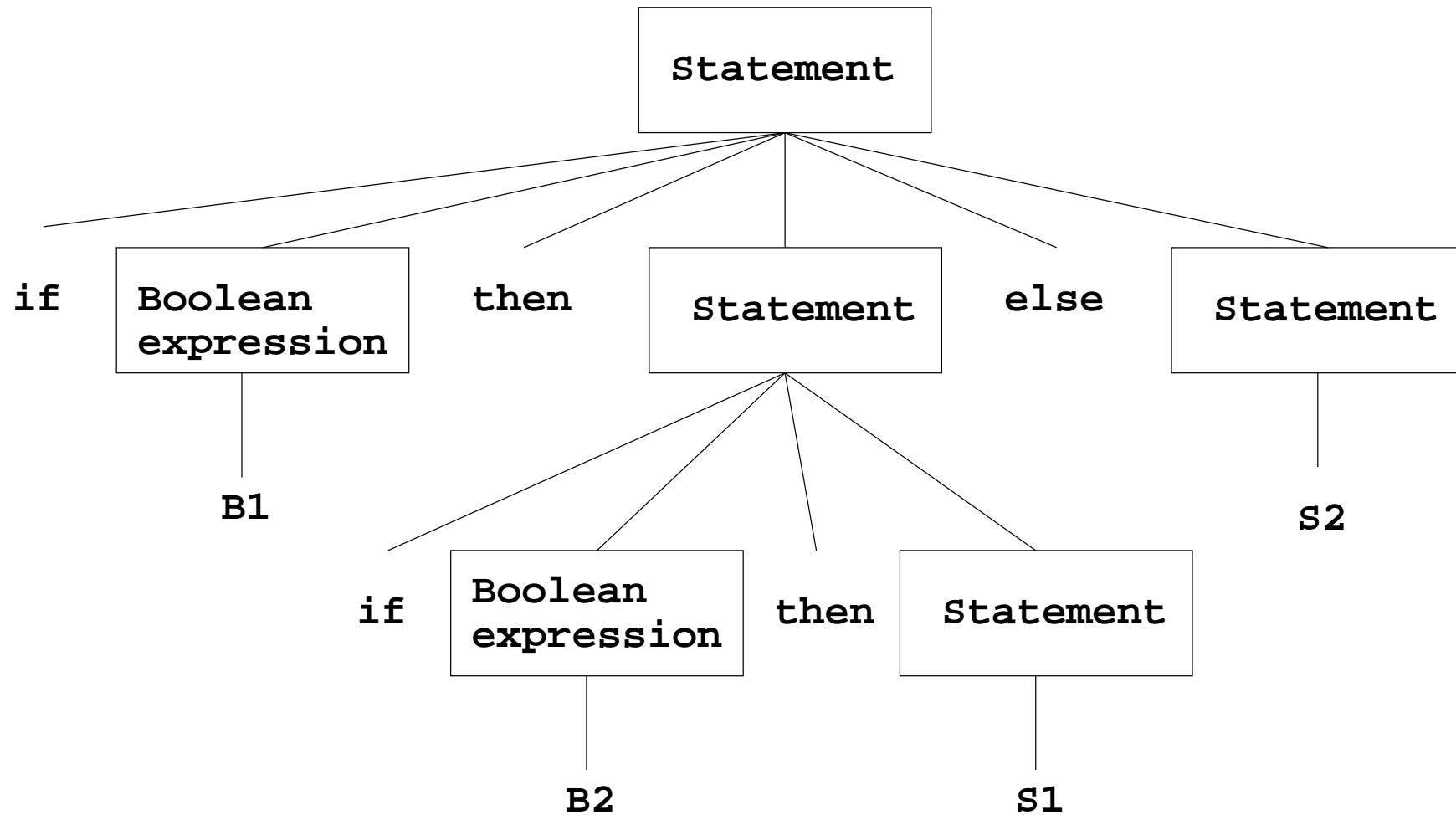
A Parse Tree for $x + y \times z$



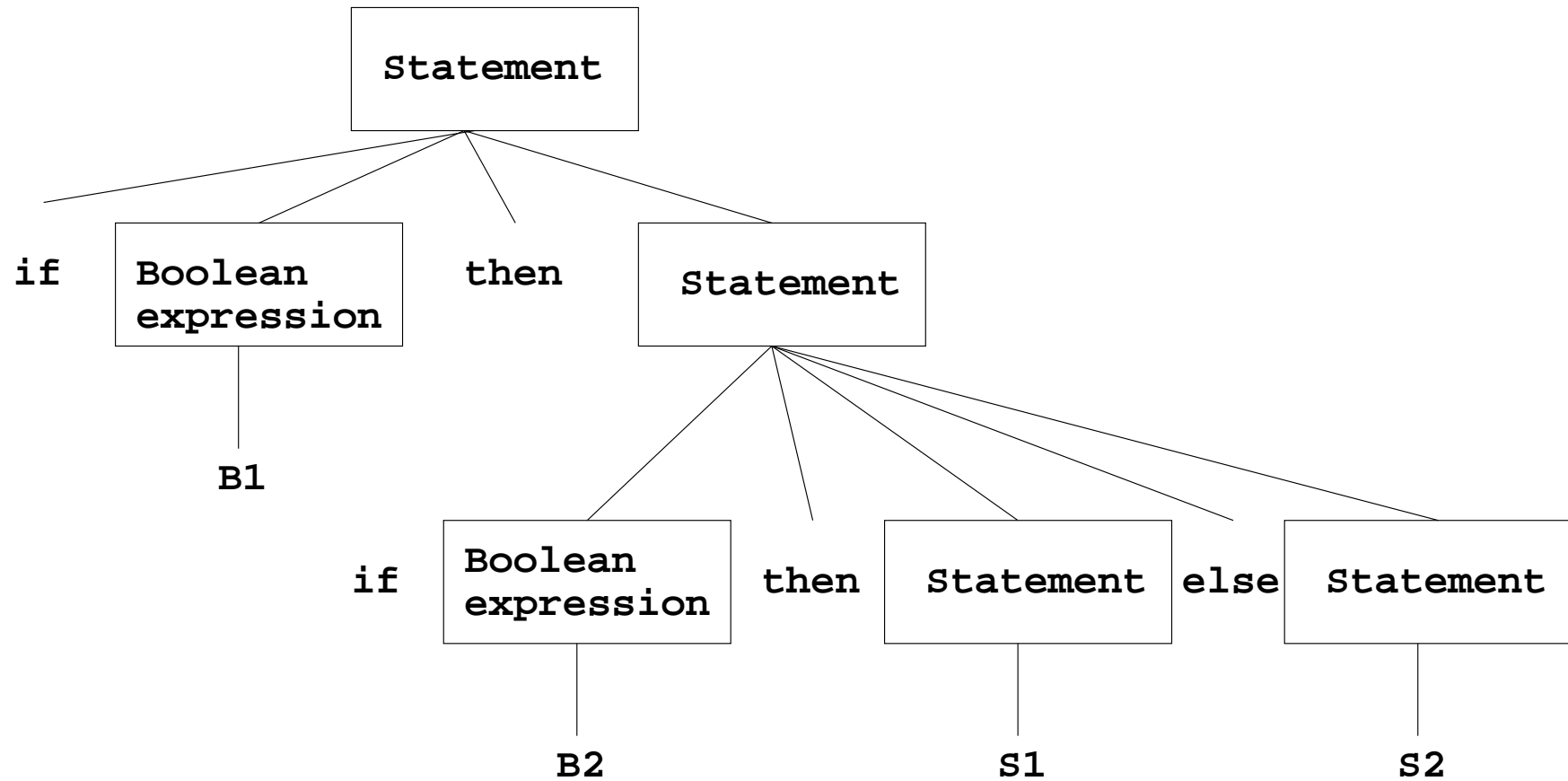
Parsing

- The process of parsing a program is essentially that of constructing a parse tree for the source program
- A parse tree represents the parser's interpretation of the program's grammatical composition
- The syntax rules must not allow two distinct parse trees for one string
- A grammar that does allow two distinct parse trees for one string is said to be an *ambiguous grammar*

If B1 then if B2 then S1 else S2



If B1 then if B2 then S1 else S2



Symbol Tables

- As a parser analyses the grammatical structure of a program, it is able to
 - identify individual statements
 - distinguish between the declarative and imperative statements
- As it recognises the declarative statements, it records the information in a table called the *symbol table*
- The symbol table contains the names of the variables as well as what data types and data structures they are associated with

Symbol Tables

- The parser then relies on this information when analysing imperative statements such as $z \leftarrow x + y$
- To determine the meaning of the symbol $+$, the parser must know the data type associated with x and y
- If x is of type real and y is of type character, then adding x and y makes little sense and should be reported as an error

Symbol Tables

- If x and y are both of type integer, then the parser will request that the code generator build a machine-language instruction using the machine's integer addition op-code
- If both are of type real, the parser will request that floating-point addition op-code be used
- If both are of type character, the parser might request that the code generator build the sequence of machine-language instructions needed to perform the concatenation operation

Symbol Tables

- If x is of type integer and y is of type real the concept of addition is applicable but the values are not encoded in compatible forms
- The parser might choose to have the code generator build the instructions to convert one value to the other type and then perform the addition
- Such implicit conversion between types is called *coercion*

Code Generation

- The final activity in the translation process is code generation
- This is the process of constructing the machine-language instructions to implement the statements recognised by the parser
- This process involves numerous issues, one being that of producing efficient machine-language versions of programs

Code Generation

- For example, consider the task of translating the two-statement sequence
 - $x \leftarrow y + z$
 - $w \leftarrow x + z$
- If these statements are translated as individual statements, each would require that data be transferred from main memory into the CPU before the indicated addition takes place

Code Generation

- However, efficiency can be gained by recognising that once the first statement has been executed, the values of x and z will already be in the CPU's general-purpose registers and therefore need not be loaded from memory before performing the second addition.
- Implementing insights such as this is called code optimisation and is an important task of the code generator

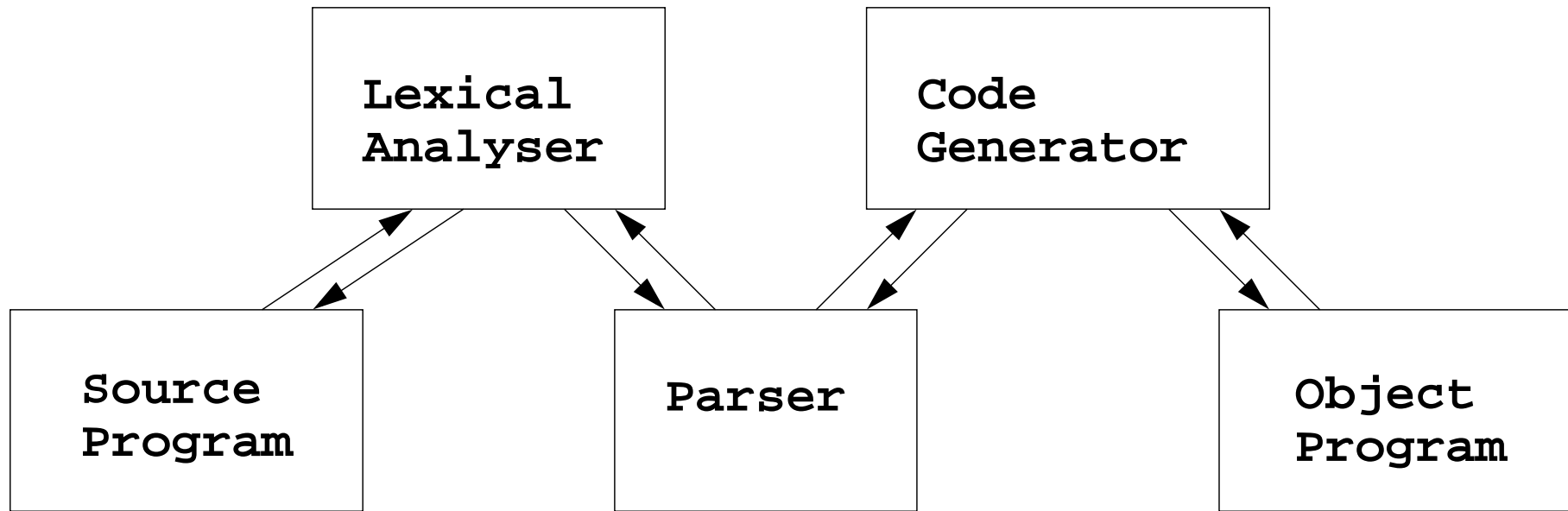
Implementation

- The steps of lexical analysis, parsing, and code generation are not carried out in a strict sequential order - these activities are intertwined
- The lexical analyser begins by reading characters from the source program and identifying the first token
- It hands this token to the parser
- Each time the parser receives a token from the lexical analyser, it analyses the grammatical structure being read

Implementation

- At this point it might request another token from the lexical analyser or, if the parser recognises that a complete phrase or statement has been read, it calls on the code generator to produce the proper machine instructions
- Each such request causes the code generator to build machine instructions that are added to the object program

Translation Process



Reading and Self Assessment

- Required reading:
 - Chapter 6
- Questions:
 - 6.1: 1, 2, 3, 4
 - 6.2: 1, 2, 3, 4, 5
 - 6.3: 1, 2, 3, 4, 5
 - 6.4: 1, 2, 3, 4