TIKRIT UNIVERSITY
COLLEGE OF COMPUTER SCIENCE AND MATHEMATICS
DEPARTMENT OF COMPUTER SCIENCE

SUBJECT OF COMPILER1
DATE OF ISSUE: 2024 - 2025
CLASS: 3TH STAGE
SEMESTER 1
LECTURE NO. : 5

PREPARED BY

Lecturer:
Mohanad Dawood Al-Roomi

&

Assistant Lecturer:
Luay Ibrahim Klalif

Syntax Analysis phase (or parsing):

- ✓ The second phase of the compiler. The parser is software system uses the first components of the tokens produced by the lexical analyzer to create a syntax tree.
- ✓ It is responsible for validates rules governing the arrangement of instructions in a programming language.
- ✓ This phase identifies errors this phase according to a grammar of language.

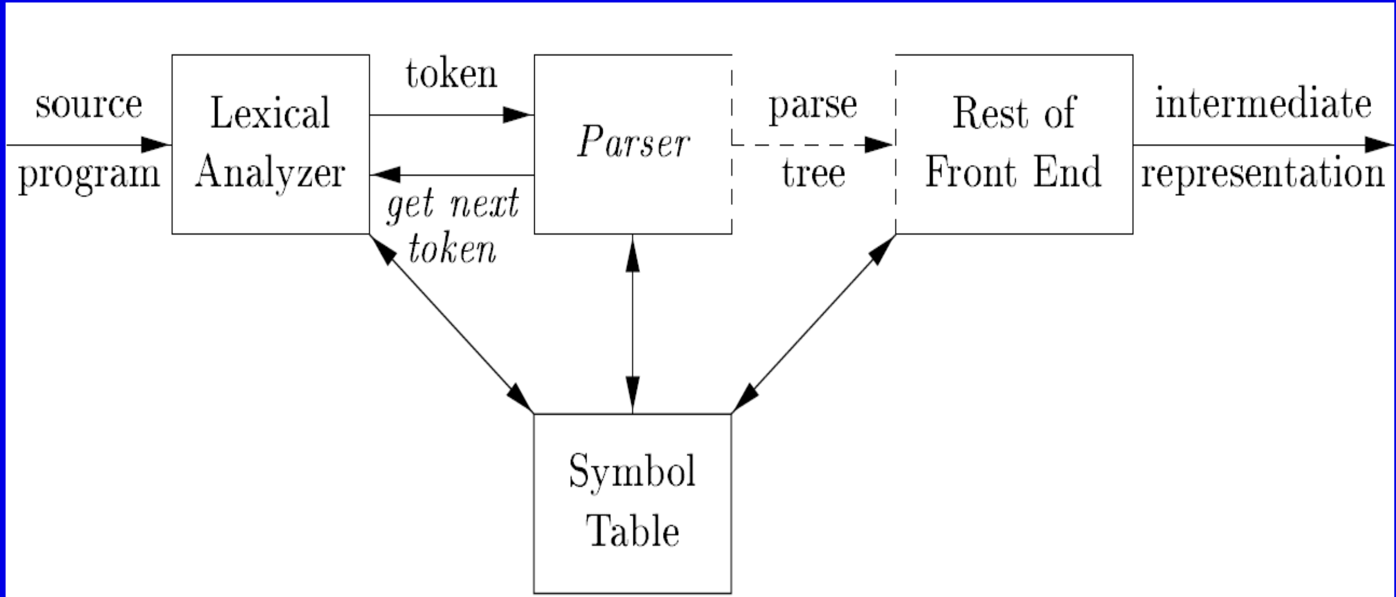


Figure 4.1: Position of parser in compiler model

The Role of the Parser

1. The parser obtains a string of tokens from the lexical analyzer.
2. The parser verifies that the string of token names can be generated by the grammar for the source language.
3. The parser builds parse tree and passes it to the rest of compiler phases for further processing.
4. The parser to report any syntax errors in an intelligible fashion.
5. Recovering from commonly occurring errors.
6. Collecting information about various tokens into the symbol table.

Error Handling

Common programming errors can occur at many different levels:

1. **Lexical errors:** include misspellings of identifiers, keywords, or operators.
2. **Syntactic errors:** include misplaced semicolons or extra or missing braces; that is, \{" or \}."
3. **Semantic errors:** include type mismatches between operators and operands, e.g., the return of a value in a Java method with result type void.
4. **Logical errors:** can be anything from incorrect reasoning on the part of the programmer.

Functions of error handler: (3)

1. It should report the presence of errors clearly and accurately.
2. It should recover from each error quickly enough to detect subsequent errors.
3. It should add minimal overhead to the processing of correct programs.

Lexical error.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      cout << "Hello world!" << endl;
6      return 0# ;
7  }
```

Syntactic error.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      cout << "Hello world!" << endl;
6      return 0 ,
7  }
```

Semantic error.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      float X[4.5]= {0.0 , 1 , 2.5 , 3.5};
6      float S = 0 ;
7      for (int i = 0 ; i <=3  ; i++)
8        S = S + X[i] ;
9      cout << "S = " << S << endl;
10     return 0;
11 }
```

Logical error.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      float X[5]= {0.0 , 1 , 2.5 , 3.0};
6      int S = 0 ;
7      for (int i = 0 ; i <=3  ; i++)
8        S = S + X[i] ;
9      cout << "S = " << S << endl;
10     return 0;
11 }
```

النتيجة = 6 والمفروض = 6.5

The Benefits of syntax grammar:

1. A grammar gives a perfect, easy-to-understand, syntactic specification of a programming language.
2. It can construct automatically an efficient parser that determines the syntactic structure of a source program.
3. A properly designed grammar is useful for translating source programs into correct object code and for detecting errors discover syntactic ambiguities.
4. A grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks.

Syntax tree (Parse tree):

A tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.

Representative Grammars (التمثيل النحوي)

There are three general types of parsers for grammars:

1. **Top-down methods:** build parse trees from the top (root) to the bottom (leaves), using instructions.
2. **Bottom-up methods:** start from the leaves and work their way up to the root, using Expression.
3. **Universal methods:** these general methods are, however, too inefficient to use in production compilers.

- In either case, the input to the parser is scanned from left to right, one symbol at a time.
- The most efficient top-down and bottom-up methods work only for subclasses of grammars.

Source program

Ex.1:

```
if ( x > y )
x = 10 ;
while …
```

1

Lexemes
if
(
x
>
y
)
x
=
10
;
while

2

Token table	
Name	Attribute
if	Kw
(	Pun
x	ID
>	Op
y	ID
)	Pun
x	ID
=	Op
10	Num
;	Pun
while	K.w

3

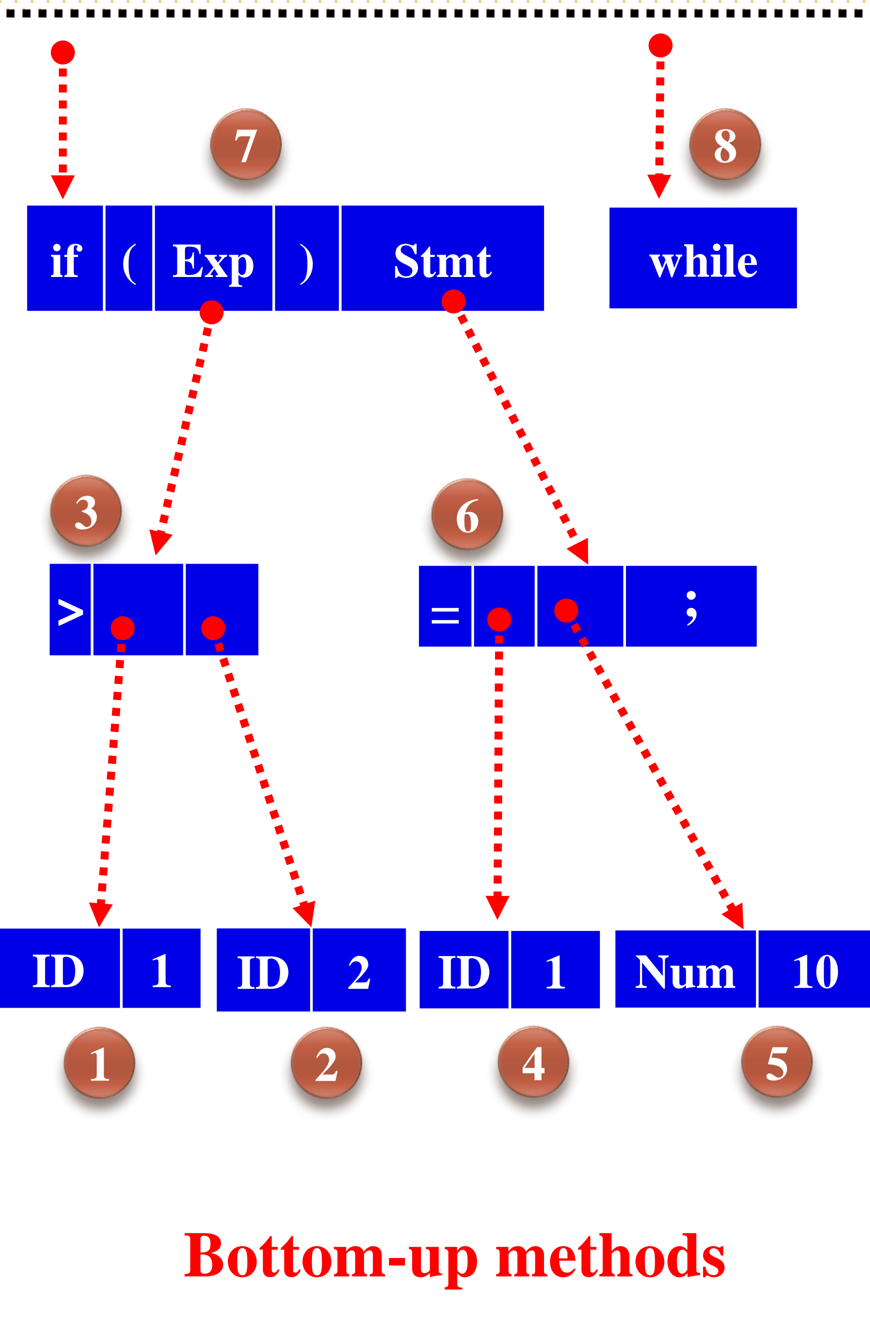
Symbol table		
Sq.	Token	
1	x	
2	y	

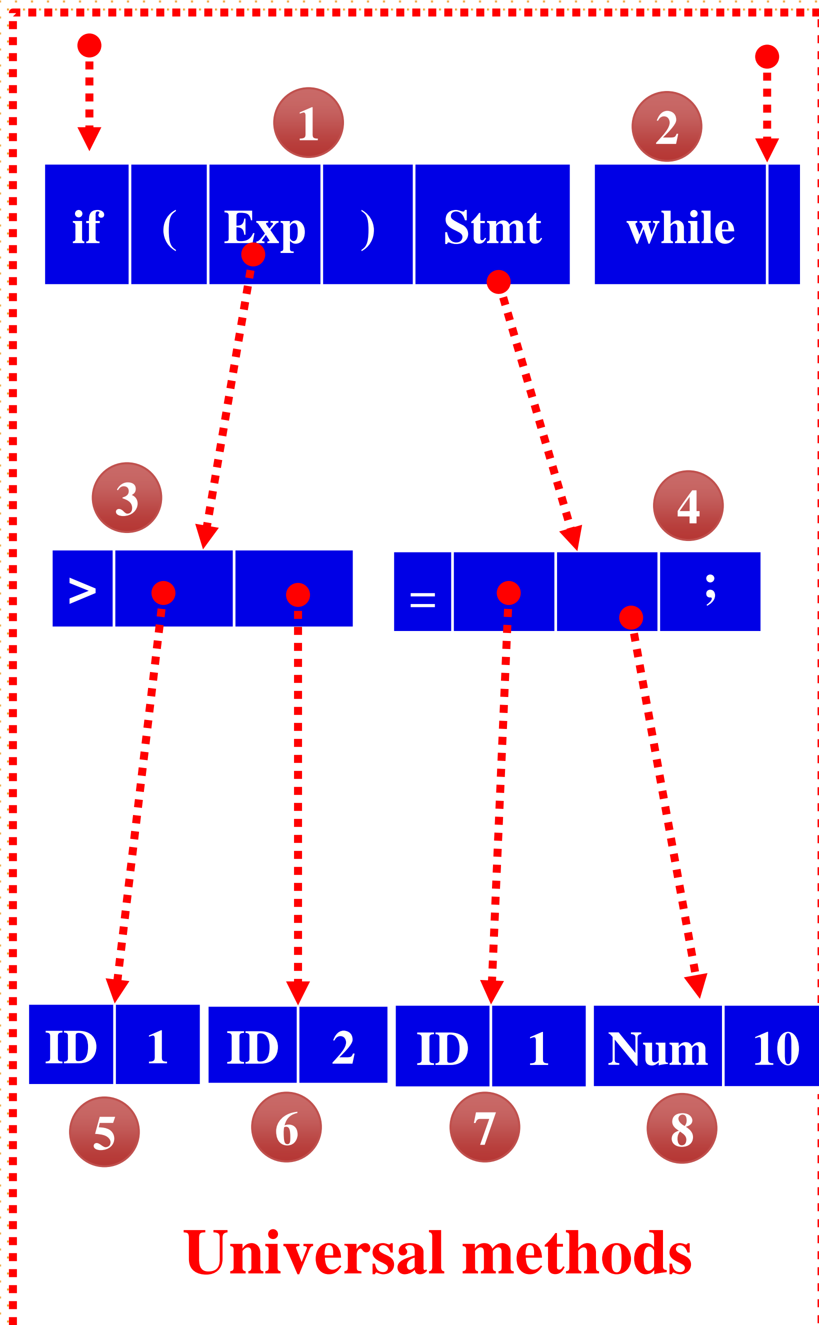
Lexical analyzer phase

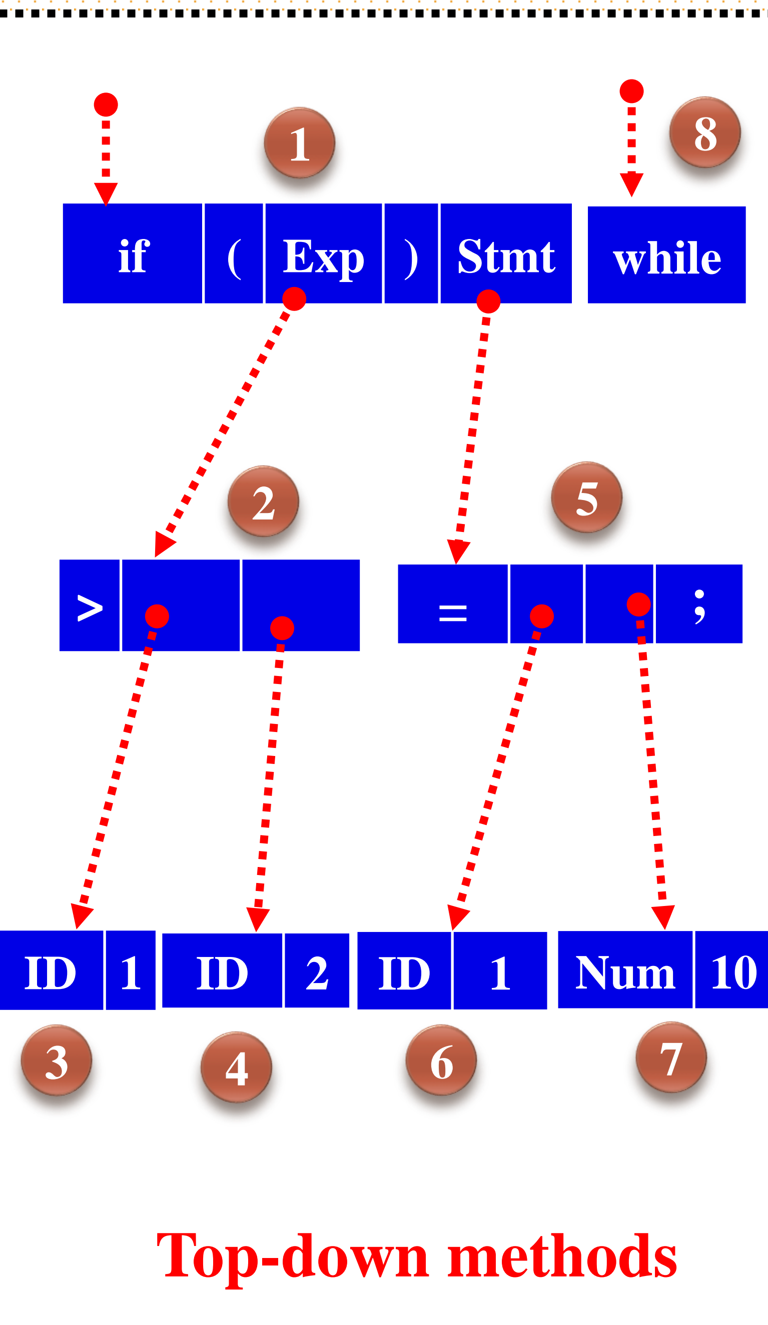
4

Token stream:
<if,k.w> <(,pun> <Id,1>
<>,Op_GT> <Id,2> <),pun>
<Id,1> <=,Op_Ass.>
<10,Num> <;,pun>
< while,k.w>

Syntax Analysis phase







التحليل من اعلى الى اسفل (Top-down methods):

ان احدى اكثر المهمات صعوبة عند كتابة محلل من اعلى الى اسفل هي مهمة اعداد قواعد اللغة لتصبح مناسبة للتحليل من اعلى الى اسفل اعتمادا عل شروط معينة. وبعد انجاز هذه المهمة يكون من السهل بناء محلل قواعد لغوية كمجموعة من الانهجة المتداخلة تبادليا (Mutual Recursive) حيث يخصص نهج معالجة لكل رمز غير ختامي (Non Terminal) في قواعد اللغة.

توجد مشكلتان يجب التغلب عليهما عند كتابة محلل من اعلى الى اسفل هما:

1. مشكلة النهج المعاكس (Back Tracking problem).
2. مشكلة التداخل من اليسار (Left overlap problem).

وسيتم التطرق الى تلك المشكلتين بشكل مفصل وكما يلي:

اولا: مشكلة النهج المعاكس: يحصل النهج المعاكس عندما يبدأ بديلين او اكثر من البدائل المتنافسة بالبداية نفسها (نفس الرمز)

مثال: لنفترض قاعدة (if) الشرطية التالية:

if_Statement ⟶ if (Expression) Statement else Statement | if (Expression) Statement

بدائل متنافسة

ولنفترض العبارة التالية كمدخلات لمحلل القواعد اللغوية:

if (X > Y)
Z = X + 1 ;

اعتمادا على القاعدة اعلاه والتي يبدأ فيها كل من البديلين المتنافسين بالبداية نفسها فان المحلل يتبع البدائل حسب تسلسلها فينجح في تحليله عندما يصادف if (Expression) Statement else Statement ولكنه سيفشل لأنه لم يصادف else ولان هناك بديل اخر فان المحلل سيلغي التحليل السابق ناهجاً نهجاً معاكساً الى بداية العبارة المدخلة ليبدأ التحليل مرة أخرى معتمداً على البديل الثاني في القاعدة.

✓ **التحليل الى عوامل لتقليص النهج المعاكس:**

يمكن استخدام طريقة التحليل الى العوامل لمحاولة تقليص النهج المعاكس كما في الامثلة التالية:

B ⟶ D,B | D,S

نلاحظ بان القاعدة اعلاه تبدأ ببديلين متشابهين (نفس الرمز) ولحل تلك المشكلة نقوم بالتقليص الى عوامل وكما يلي:

B ⟶ D,[B|S]

وتكون قاعدة if الشرطية السابقة بعد التحليل الى عوامل بالشكل التالي:

Statement → if (Expression) Statement [else Statement]?

ثانياً: التداخل من اليسار يحدث هذا النوع عندما يظهر رمز الطرف الايسر عند النهاية اليسرى لاحد بدائل الطرف الايمن.

Ex: S ⟶ abc | def | Srx

هناك صعوبة في بناء محلل للقواعد المتداخلة من اليسار لان المحلل لا يستطيع التعرف على العبارة التي يقوم بتحليلها عندما يختار البديل المتداخل من اليسار حيث اول رمز في هذا البديل يسبب تداخل. وتوجد طريقتان لحذف التداخل من اليسار هما:

أ- جعل التداخل من اليمين:

EX: <int> ⟶ <dig> | <int> <dig>

نلاحظ بان القاعدة متداخلة من اليسار وعلية يتم تحويل التداخل من اليمين

<int> ⟶ <dig> | <dig> <int>

هذه القاعدة اصبحت متداخلة من اليمين ولكن ظهرت مشكلة اخرى وهي ان هناك بديلين لهما نفس البداية وهذا ما نسميه النهج المعاكس ولذا يجب التحليل الى عوامل لتقليص ذلك النهج المعاكس وكما يلي:

<int> ⟶ <dig> (<int>)*

ب- حذف التداخل وجعل القاعدة تكرارية:

EX: <int> ⟶ <dig> | <int> <dig>

<int> ⟶ <dig>
⟶ <int> <dig> ⟶ <dig> <dig>
⟶ <int> <dig> <dig>

وهكذا يستمر التحليل فنلاحظ من تفسير القاعدة بان <int> يمكن ان تكون <dig> واحدة او اثنان او ثلاثة...الخ وبهذا يمكن ان نحذف التداخل ونحول القاعدة الى تكرار وكما يلي:

<int> ⟶ <dig> (<dig>)*

Syntax Analysis phase

طور تحليل القواعد اللغوية:

هو نظام برمجي، وهو ثاني طور في المترجم كما يسمى هذا الطور المعرب (Parser). مدخلات هذا الطور هو (Tokens stream) التي يحصل عليها من الطور Lexical analyzer . يقوم محلل القواعد اللغوية بجمع (Tokens) مع بعضها البعض لتكوين هياكل اللغة مثل التعابير (Expression) والعبارات (Statement). والتي بدورها ترتبط فيما بينها لتكون الهياكل شجرة الاعراب لجميع اجزاء البرنامج (Parse tree). ثم يقوم بتمريرها الى باقي اطوار البرنامج بعد ان يتأكد هذا الطور من صحة عبارات وتعابير البرنامج اعتماداً على قواعد اللغة الانماط (القواعد المايكروية) والمحلل اللغوي بالاعتماد على أول فقرة في العبارة يحدد القاعدة المطلوبة.

مثلا فعندما يصادف المعرب أول فقرة في العبارة if فإنه اعتماداً على القاعدة if

Statement ⟶ if (Expression) Statement | if (Expression) Statement else Statement

هذا الصيغة خطأ تسبب مشكلة النهج المعاكس Back Tracking problem

هذا الصيغة الصحيحة

Statement → if (Expression) Statement (else Statement)?

فمن المفترض به في البداية ان يجد كلمة if ومن ثم يجب ان يصادف الرمز بادئة قوس (وثم يصادف Expression حسب قاعدته وبعد ذلك يجب ان يصادف الرمز نهاية القوس) ومن ثم يجب أن يصادف Statement وأخيراً يجب أن يصادف الفارزة منقوطة ; فإذا تكللت كل العمليات السابقة بنجاح يقوم ببناء عقدة في شجرة الإعراب لهذه العبارة. كما هو موضح في المثال رقم (Ex.1):

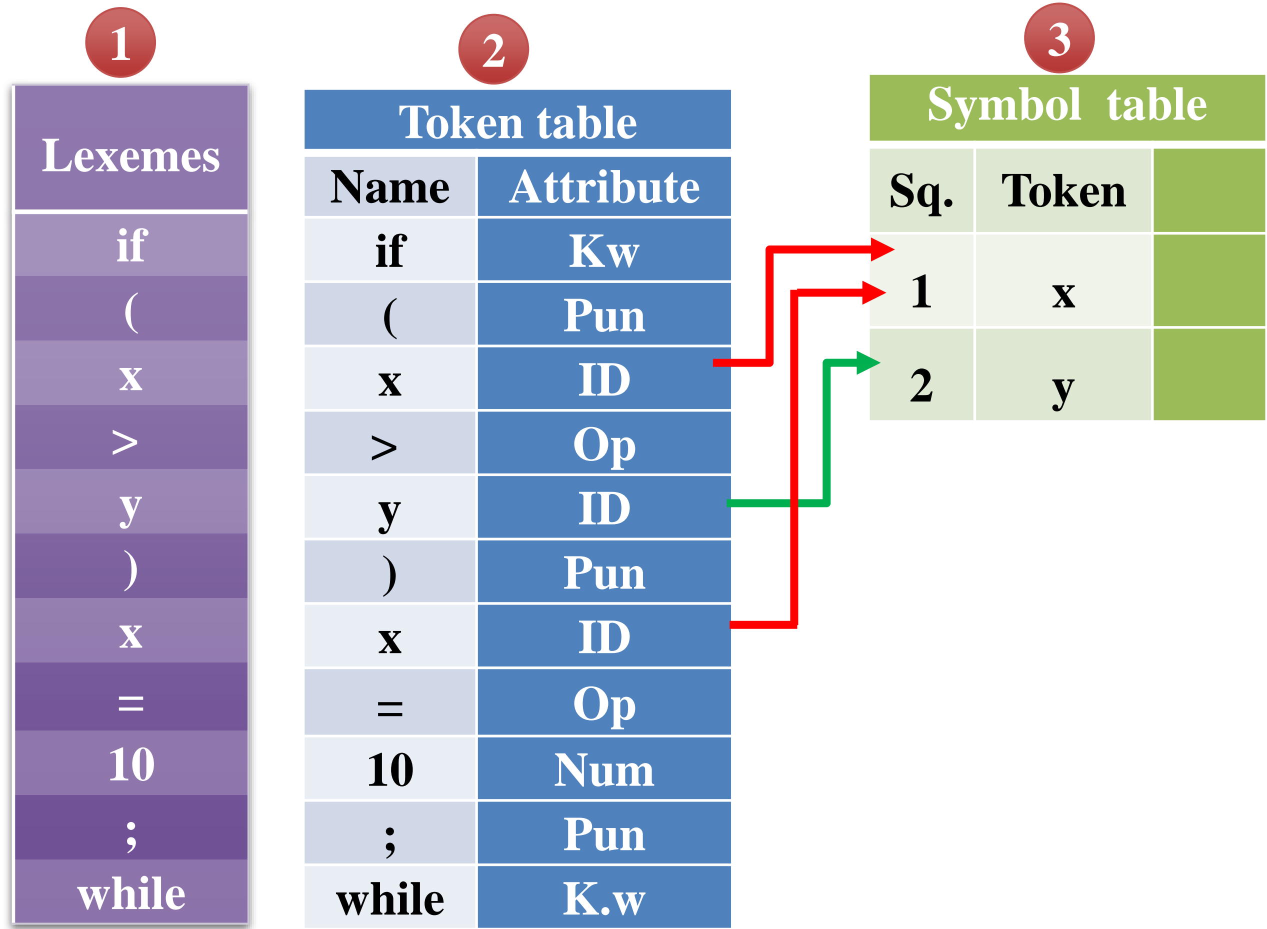
(10)

Source program

Ex.1:

if (x > y)
x = 10 ;
while …

Lexical analyzer phase

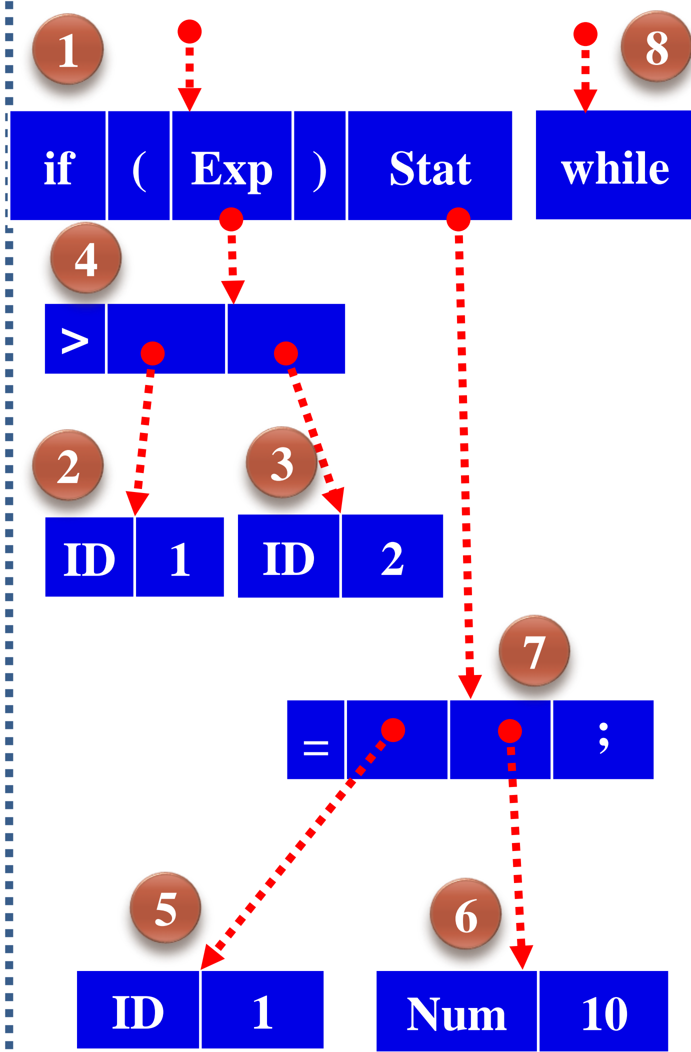


4

Token stream:

<if,k.w> <(,pun> <Id,1> <>,Op_GT> <Id,2> <),pun>
<Id,1> <=,Op_Ass.> <10,Num> <;,pun>
< while,k.w>

Syntax Analysis phase



Top-down methods
&
Bottom-up methods

Syntax Analysis phase

- الاخطاء التي يكتشفها الطور هي في تركيب الهياكل اللغوية (عبارات البرنامج). فمثلاً لو كانت بادئة قوس (غير موجودة في المثال السابق فسوف يطبع رسالة: expected " (" .
- شجرة الاعراب (Parse tree) تمثل طريقة ربط اجزاء البرنامج (Tokens) مع بعضها لتكوين تراكيب أكبر (Expression & Statements) وكيفية ترابط هذه التركيب لتكوين تراكيب أكبر منها أي ان شجرة الاعراب توضح هيكل البرنامج ككل.
- مهمة هذا الطور هو جمع الـ Tokens stream الناتجة من طور تحليل الفقرات (Lexical analyzer phase) وهل هي مرتبة حسب قواعد اللغة وفق نمط معيناً معلوم لها ومن ثم تكوين شجرة الاعراب.
- يقسم هيكل البرنامج لغة C و ++C الى اربعة اقسام:-

أولاً: التضمين.

ثانياً: التعريف.

ثالثاً: البرامج الرئيسي (الدالة الرئيسية).

1. الاعلانات.
2. التعليمات.

رابعاً: الدوال الفرعية.

1. الاعلانات.
2. التعليمات.

Ex.2: If we a source program by C++ in below. How to analyze it by the lexical analyzer phase and the syntax analyzer phase. Find the following:

(12)

1. **Errors correction in for the lexical analyzer phase?**
2. **Lexemes?**
3. **Token table?**
4. **Symbol table?**
5. **Token stream?**
6. **Errors correction in for the syntax analyzer phase?**
7. **Schema of parse tree?**

```
int  main (  )
{
    int  x ;
    x  =  10 ;
    if  (  x  >   5  )
        x  =  x  +  1 ;
    x  =  x  –  2 ;
    cout  <<  x  ;
    return  0 ;
}
```

2

Lexeme
int
main
(
)
{
int
x
;
x
=
10
;
if
(
x
>
5
)
x
=
x
+
1
;
x
=
x
-
2
;
cout
<<
x
;
return
0
;
}

3

Token table	
Name	**Attribute**
int	K.w
main	K.w
(	Pun
)	Pun
{	Pun
int	K.w
x	Id
;	Pun
x	Id
=	Op_ASS
10	Num
;	Pun
if	K.w
(	Pun
x	Id
>	Op_GT
5	Num
)	Pun
x	Id
=	Op_ASS
x	Id
+	OP_ADD
1	Num
;	Pun
x	Id
=	Op_ASS
x	Id
-	OP_SUB
2	Num
;	Pun
cout	K.w
<<	Pun
x	Id
;	Pun
return	K.w
0	Num
;	Pun
}	Pun

Lexical analyzer phase

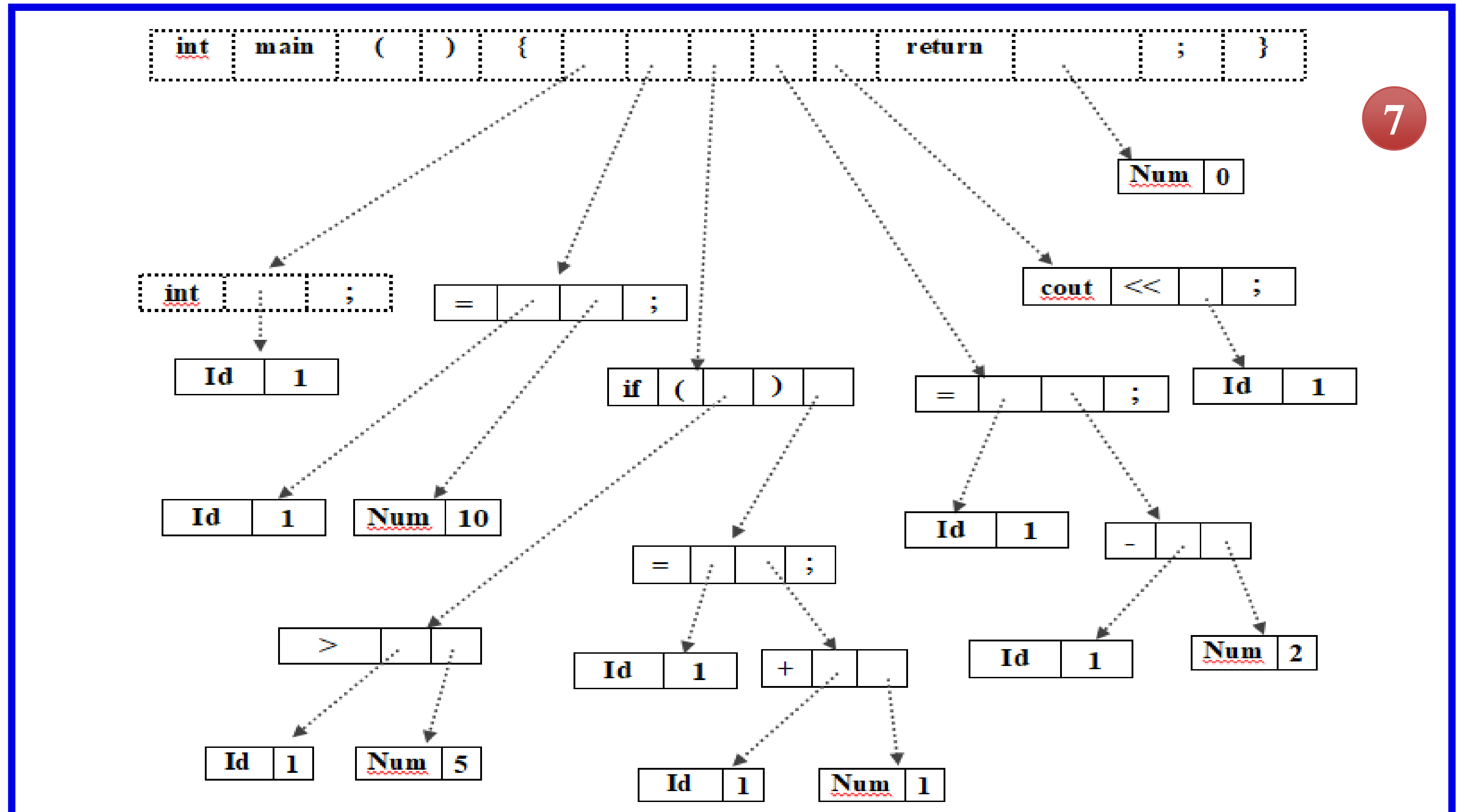
4

Token table		
Sq.	**Token**	**Descriptions**
1	x	فراغ

5

Token stream: (H.W.)

Syntax Analysis phase



H.W.: If we a source program by C++ in below. How to analyze it by the lexical analyzer phase and the syntax analyzer phase. Find the following:

1. **Errors correction in for the lexical analyzer phase?**
2. **Lexemes?**
3. **Token table?**
4. **Symbol table?**
5. **Token stream?**
6. **Errors correction in for the syntax analyzer phase?**
7. **Schema of parse tree?**

```
#include  <iostream.h>
using namespace std ;
int main (  )
  {
     int  x  ,   y  ;
     x1  =  10  ;  ;  ;
     if  ( x  >  5  )
        if  ( y  ==  0  )
           x  =  x  +  1  ;
     x  =  x  –  2  ;
     cout  <<  x  ;
     return 0  ;
  }
```

Source program

THANK YOU