جَامعة المَنارة
MANARA UNIVERSITY

Compilers Techniques
Lecture2
المحاضرة الثانية

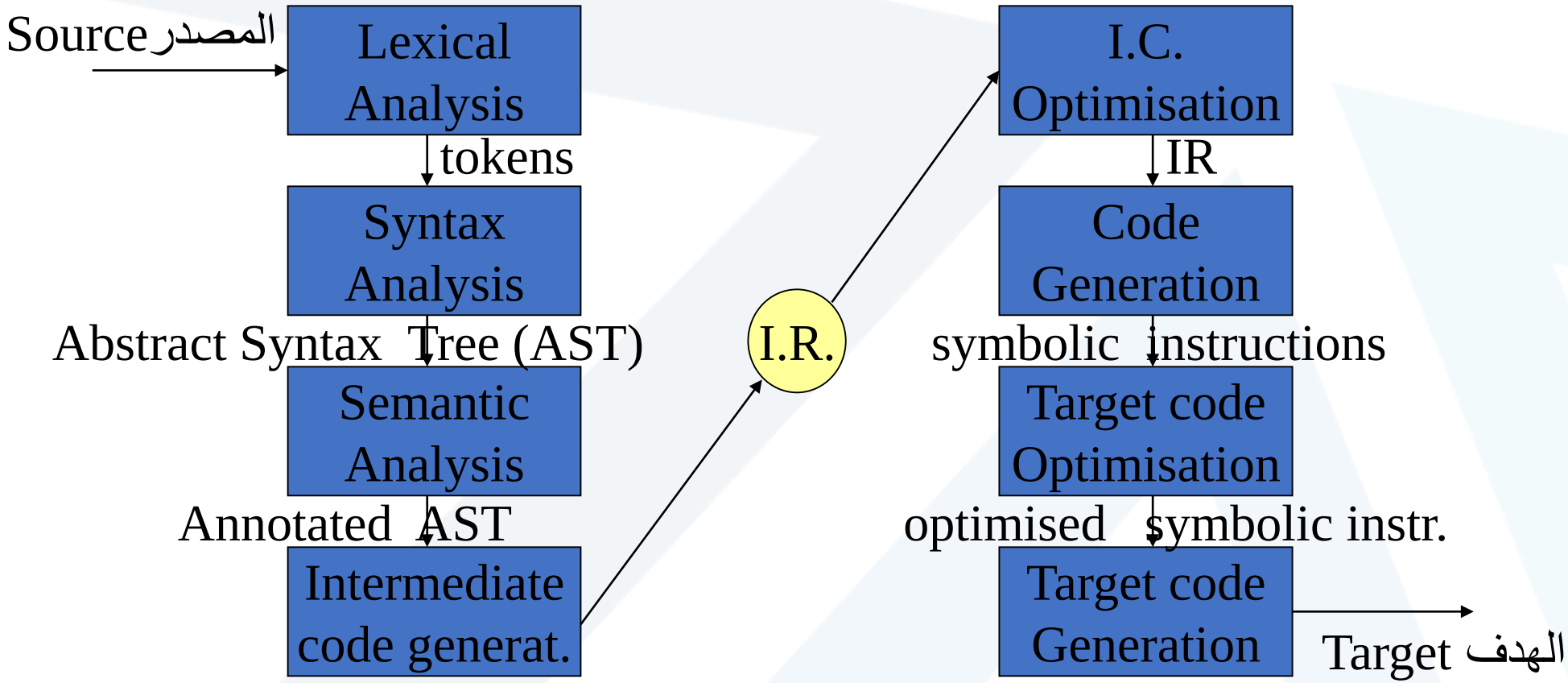
Compiler Architecture (Compile stages)

بنية المترجم (مراحل الترجمة)

Reference: Alfred, V. Aho, S. Lam Monica, and D. Ullman Jeffrey. "Compilers Principles, Techniques & Tools." (2007), *Pages 1-24 & 109-146 (Chapter 3)*

السنة الرابعة – المستوى السابع - الهندسة المعلوماتية

البنية العامة للمترجم – تذكير من المحاضرة السابقة -



التحليل المعجمي (المسح)
Lexical Analysis (Scanning)

- تقرأ المحارف في البرنامج المصدري وتجمعها بكلمات (الوحدات القواعدية الأساسية).
- تولد الكلمات وتميز أنواعها.
- يسمى خرج هذه المرحلة معلِّمات token ويخرج كأزواج من الشكل التالي:

<type, lexeme> or *<token_class, attribute>*

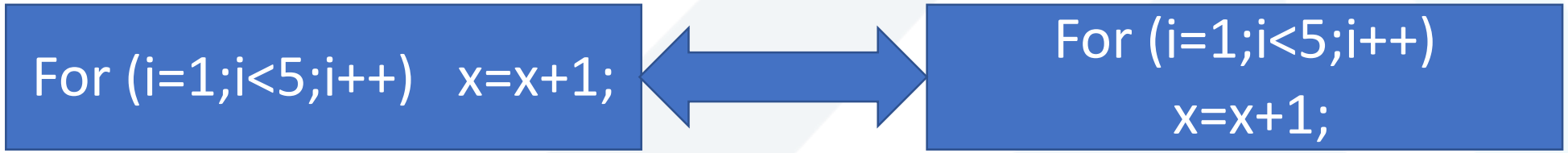
- مثال:

a=b+c تصبح

<id,**a**> <=,> <id,**b**> <+,> <id,**c**>

1- التحليل المعجمي (المسح) Lexical Analysis (Scanning)

- يجب حفظ معرّف المتغير id attribute في جدول الرموز.**symbol table**
- يلغي (يتجاهل) التحليل المعجمي Lexical analysis الفراغات white space ومثيلاتها
- Example;



- يمكن استخدام أدوات tools خاصة مثل **flex** الذي هو أداة لتوليد الماسحات **scanners**:
  وهي برمجيات تقرأ تسلسل الكلمات في البرنامج المصدري وتطابقها وفق قوالب محددة لتقوم بتقسيم هذه التعليمات إلى كلمات (معلِّمات) Tokens.

Syntax Analysis

2- التحليل القواعدي (أو الإعراب Parsing)

- يشكل بنية هرمية معتمدة على تسلسل المعلِّمات token stream. القادم من مرحلة التحليل المعجمي
- يعبر عن البنية الهيكلية لقواعد اللغة عادةً من خلال القواعد العودية recursive rules.
- توصف القواعد العودية رياضياً من خلال **نحو السياق الحر** Context-free grammars والتي تقود بدورها عملية التحليل القواعدي

```
expression → expression '+' term
```

2- التحليل القواعدي Syntax Analysis (أو الإعراب Parsing)

الإعراب Parsing:
نتاج الإعراب Production

- **مثال: وفق قواعد النحو التالية هل تندرج العبارة a*b+c ضمن قواعد اللغة؟**

```
expression → expression ‘+’ term | expression ‘-’ term | term
term → term ‘*’ factor | term ‘/’ factor | factor
factor → identifier | constant | ‘(‘ expression ‘)’

expression → expression ‘+’ term
           → term ‘+’ term
           → term‘*’ factor ‘+’ term
           → factor ‘*’ factor ‘+’ term
           → factor ‘*’ factor ‘+’ factor
           → identifier ‘*’ factor ‘+’ factor
           → identifier ‘*’ identifier ‘+’ factor
           → identifier ‘*’ identifier ‘+’ identifier
```

(yes it’s under grammar)

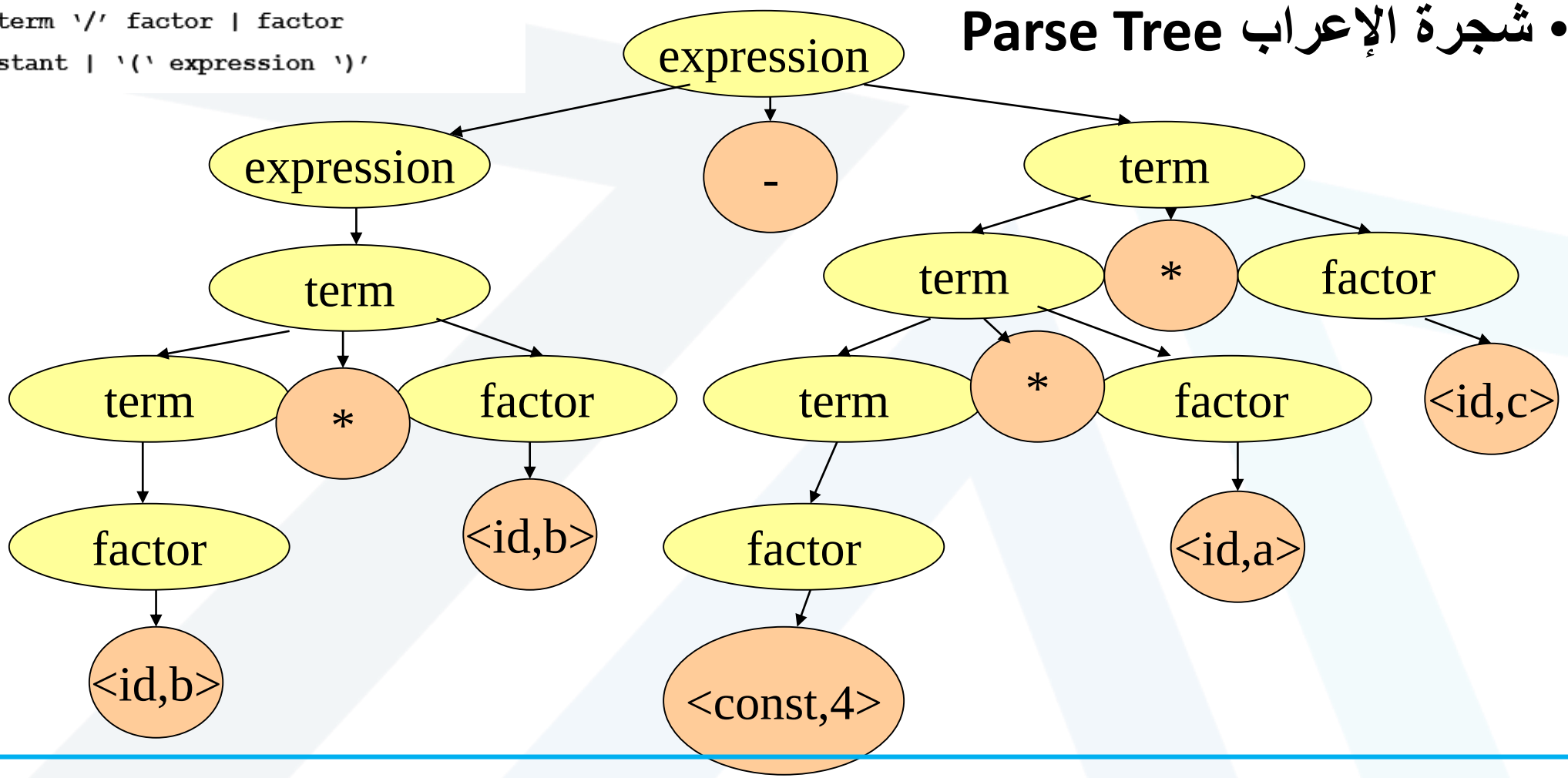
2- التحليل القواعدي Syntax Analysis (أو الإعراب Parsing)

جَامعة المَنارة MANARA UNIVERSITY

الإعراب Parsing:
شجرة الإعراب للتعبير:
*b*b-4*a*c*

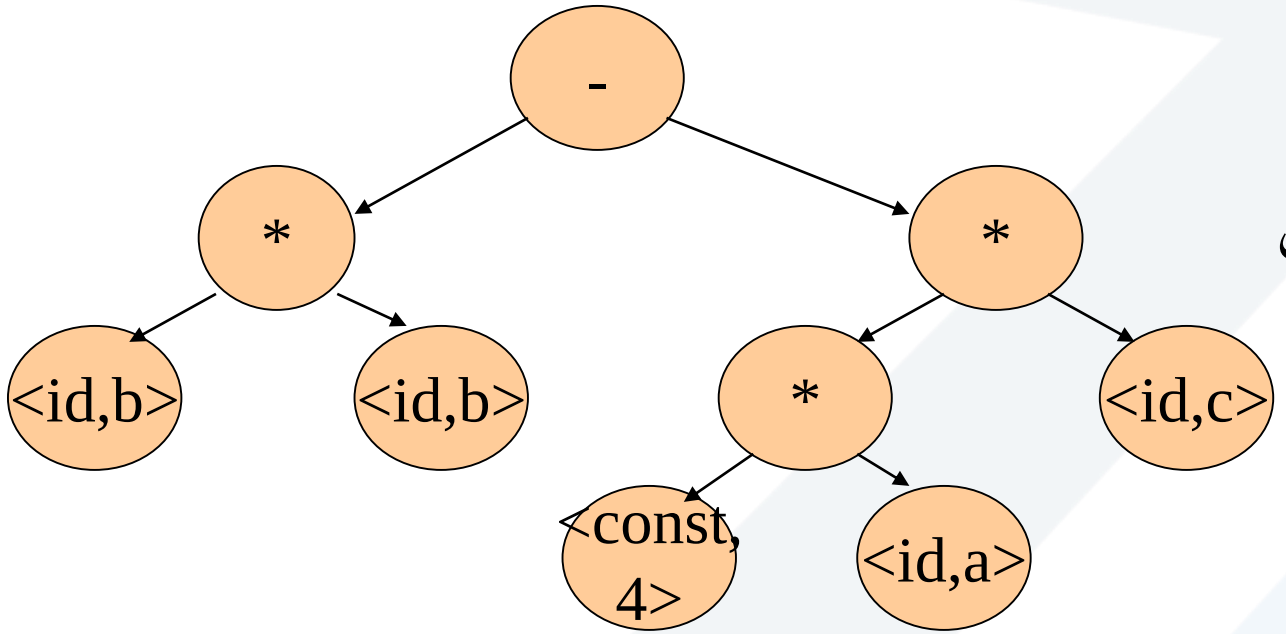
```
expression → expression '+' term | expression '-' term | term
term → term '*' factor | term '/' factor | factor
factor → identifier | constant | '(' expression ')'
```

- شجرة الإعراب **Parse Tree**



Abstract Syntax Tree
شجرة الإعراب المجردة

2- التحليل القواعدي Syntax Analysis (أو الإعراب Parsing)



- AST for *b*b-4*a*c*
- تعتبر شجرة القواعد المجردة Abstract Syntax Tree (AST) بنية معطيات أكثر نفعاً للتمثيل الداخلي.
- وهي نسخة مضغوطة من شجرة الإعراب parse tree (تحتوي على خلاصة البنى القواعدية دون تفاصيل عن كيفية اشتقاقها)
- تعتبر ASTs أحد أشكال التمثيل الوسيطي IR.

التحليل الدلالي
Semantic Analysis

(التعامل مع السياق context handling)

- تختبر وجود أخطاء دلالية
- وتنتج مجموعة ملاحظات عن عقد الشجرة الناتجة.
- **أمثلة: Examples:**
  - اختبار النمط type checking:

```
void func(int x) {}
int main() {    func("string");  // Type check error
return 0;}
```

  - اختبارات التصريح لكشف أخطاء الإسناد قبل التصريح

```
int x=1; z=x*2;
```

  - اختبار التدفقات التحكمية check flow-of-controls.
  - اختبار كون أسماء المتحولات غير مكررة.uniqueness or name-related checks

```
int x=1; double x=2.2;
```

توليد الشيفرة الوسيطية
Intermediate representation IR or intermediate code generation

- يتم في هذه المرحلة توليد شيفرة وسيطية وهي غير نهائية أي أنها تحتاج لعمليات لاحقة (أمثلة وتحسين).
- ذكرنا سابقاً أن شجرة AST أحد أشكال التمثيل الوسيطي.
- مثال عن أحد أشكال IR (الشيفرة ثلاثية العناوين 3-address code):

```
tmp1=4
tmp2=tmp1*a
tmp3=tmp2*c
tmp4=b*b
tmp5=tmp4-tmp3
```

أمثلة (تحسين) الشيفرة
Code Optimisation

- هدف هذه المرحلة هو تحسين الشيفرة الوسيطية وبالتالي فعالية توليد الشيفرة وإنجاز الشيفرة الهدف target code.
- تتفاوت الأمثلة بين البدائية trivial (تضمين الثوابت) إلى شديدة التعقيد (توابع in-lining).
- مثلاً استبدال أول تصريحين في المثال السابق.

```
tmp1=4
tmp2=tmp1*a
```

```
tmp2=4*a
```

- تأخذ المترجمات الحديثة على عاتقها إنجاز مجال الأمثلة هذا.

مرحلة توليد الشيفرة
Code Generation Phase

- تجري مطابقة شجرة القواعد المجردة AST مع قائمة تعليمات الآلة التي سيتم تنفيذه عليها:
  - اختيار التعليمة: مسألة مطابقة نماذج. يوجد نوعان من المعالجات CICS, RISC وتختلف قائمة التعليمات لكل منهما.
  - الربط مع المسجلات Register allocation: يجب إلحاق كل قيمة بمسجل عند استعمالها (مع ملاحظة أنه يوجد عدد محدود من المسجلات)NP-Complete problem.
  - جدولة التعليمات (ترتيب التعليمات): يحدد مولد الشيفرة ترتيب تنفيذ التعليمات وينشئ عملية جدولة لها.
- يمكن استخدام خصائص خاصة بالآلة لأمثلة الشيفرة.
- وأخيراً يجري توليد شيفرة الآلة والمعلومات المرفقة المطلوبة بواسطة نظام التشغيل.

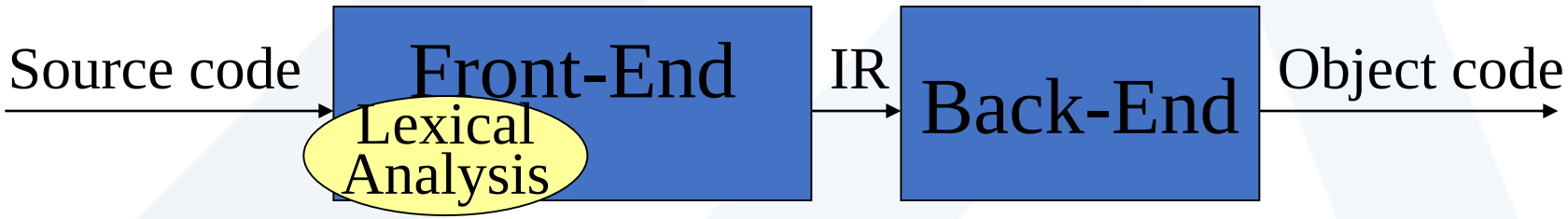
التحليل المعجمي
Lexical Analysis

الخطوة الأولى في أية عملية ترجمة هي تحديد فيما إذا كان البرنامج المصدري مناسباً وفقاً لمصطلحات اللغة المصدر.
وتحدد القواعد كقواعد اختبار (قوالب) لمقاطع الكلام التي تطابق بين كلمات البرنامج المصدري والقواعد.
أي يتم مطابقة تسلسل الدخل مع قوالب محددة مسبقاً لمعرفة نوع مفردات الدخل ومدى تطابقها مع مفردات اللغة.

- عملية حجز الكلمات المفتاحية هامة جداً reserved keywords are important

What does lexical analysis do? *مالذي يقوم به التحليل المعجمي*

Recognises the language's parts of speech. ***يميز المقاطع الكلامية للغة***



التحليل المعجمي
Lexical Analysis

- المفردات vocabulary (الأحرف الهجائية) هي مجموعة منتهية من الرموز مثلاً مجموعة الأحرف A-Z.
- السلسلة string هي أية سلسلة منتهية من الرموز المنتقاة من المفردات.
- اللغة language هي أية مجموعة من السلاسل المعرفة على مفردات محددة.
- القواعد grammar هي طريقة محددة (منتهية) من أجل وصف اللغة.
- القواعد حرّة السياق context-free grammar، *G*، *هي رباعية* 4-tuple،حيث *G=(S,N,T,P)*,
  - *S*: رمز بداية starting symbol
  - *N*: *مجموعة من الرموز غير التحديدية* non-terminal symbols
  - *T*: *مجموعة من الرموز التحديدية* terminal symbols
  - *P*: *مجموعة من قواعد الإنتاج* production rules
- اللغة language: هي مجموعة من جميع النواتج التحديدية المعرفة على *G*.

Example مثال

مثال عن النحو حر السياق:

S=Exp;
N={Exp,Term,Factor};
T={+,,(,),x}*
P={Exp →Term|Exp+Term,
*Term →Factor|Term*Factor,*
Factor →(Exp)|id}

بالتعويض المتكرر نشتق الصيغ المتعلقة بالجملة:

Exp ⇒Exp+Term
⇒Term+Term
⇒Factor+Term
⇒id+Term
*⇒id+Term*Factor*
*⇒id+Factor*Factor*
*⇒id+id*Factor ⇒id+id*id*

وهذا مثال على الاشتقاق من اليسار إلى اليمين (حيث في كل خطوة نقوم باستبدال وتوسيع القيمة غير Non-terminal اليسرى)

لماذا ندرس التحليل المعجمي؟

- لتجنب كتابة المحللات المعجمية (الماسحات) يدوياً.
- لتحويل تسلسل الدخل إلى تسلسل من المعرفات Tokens ذات قدرة أكثر على الفهم والتعامل معها.
- نحتاج لتحديد نماذج معجمية **lexical patterns**(لاشتقاق العلّامات tokens)
- بعض الأجزاء سهل التعريف مثل:

- *WhiteSpace → blank | tab | WhiteSpace blank | WhiteSpace tab*
  - الكلمات المفتاحية والمعاملات (if, then, =, +).
  - التعليقات Comments (/* followed by */ in C, // in C++, % in latex)
- بعضها أكثر تعقيداً مثل:

- Identifiers (letter followed by - up to *n* - alphanumerics…)
- Numbers

نحتاج إلى مجموعة قواعد وقوالب محددة تقودنا إلى التنفيذ.

هو صيغة رياضية رمزية تصف مجموعة غير منتهية من السلاسل يمكن استخدامها لتقسيم تسلسل البرنامج المصدري إلى مجموعة المعرفات Tokens وبالتالي يمكن استخدامها لتوصيف اللغات

مثل `ls [x-z]*`

التعبير النظامي Regular Expression (RE) المعرف على المفردات V

- ε is a RE denoting the empty set $\{\varepsilon\}$.
- If r_1, r_2 are REs then:
  - r_1* denotes zero or more occurrences of r_1; نسخة واحدة أو أكثر أو لا شيء
  - $r_1 r_2$ denotes concatenation; r1 و r2 معاً
  - r_1 | r_2 denotes either r_1 or r_2; إما .... أو

- **اختصارات لبعض التعابير المنتظمة:**
  - *[a-d]* for *a | b | c | d*; — تعبير منتظم يضم الأحرف من a وحتى d
  - r^+ for *rr**; — نسخة واحدة أو أكثر من الحرف r
  - *r?* for *r* | ε — نسخة واحدة على الأكثر من r

وصّف اللغة التي تشير إلى التعابير النظامية التالية:

a;
a | b;
*a**;
*(a | b)**;
(a | b)(a | b);
*(a*b*)**;
*(a | b)*baa*;

- *integer → (+ | − | ε) (0 | 1 | 2 | … | 9)+*
- *integer → (+ | − | ε) (0 | (1 | 2 | … | 9) (0 | 1 | 2 | … | 9)*)*
- *decimal → integer.(0 | 1 | 2 | … | 9)**
- *identifier → [a-zA-Z] [a-zA-Z0-9]**

لا نستطيع توصيف جميع اللغات باستخدام التعابير النظامية

بناء المحلل المعجمي يدوياً

يمكننا كتابة المحلل المعجمي اعتماداً على خصائص العلّاماتtokens باستخدام التعابير النظامية.
تتلخص هذه المقاربة باختبار حالة تلو الأخرى والتقسيم إلى مسائل أصغر يمكننا حل كل منها على حدة
مثال:

```
void get_next_token() {
  c=input_char();
  if (is_eof(c)) { token ← (EOF,"eof"); return}
  if (is_letter(c)) {recognise_id()}
  else if (is_digit(c)) {recognise_number()}
        else if (is_operator(c))||is_separator(c))
              {token ← (c,c)}  //single char assumed
             else {token ← (ERROR,c)}
  return;
}
...
do {
  get_next_token();
  print(token.class, token.attribute);
} while (token.class != EOF);
```

من الممكن أن تكون هذه الطريقة فعالة إلا أنها الكثير من الجهد والوقت وصعبة التعديل.

بناء المحلل المعجمي ألياً

الفكرة Idea: نختبر التعابير النظامية واحداً تلو الآخر ونوجد المطابقة الأطول.

```
set (token.class, token.length) ←(NULL, 0)
// first
find max length such that input matches T1→RE1
  if max length > token.length
      set (token.class, token.length) ←(T1, max_length)
// second
find max length such that input matches T2→RE2
  if max length > token.length
      set (token.class, token.length) ←(T2, max_length)
...
// n-th
find max length such that input matches Tn→REn
  if max length > token.length
      set (token.class, token.length) ←(Tn, max_length)
// error
if (token.class == NULL) { handle no_match }
```

المساوئ تعتمد على عدد أصناف العلاّمات بشكل خطي وتحتاج إعادة تشغيل البحث عن كل تعبير نظامي.