المقدمة:

عند تصميم وبناء البرمجيات، غالبًا ما نواجه تحديات وتعقيدات متعددة تتعلق بطريقة التعامل مع أوامر الحاسوب وتعليماته .ففي البدايات، كان المبرمجون يكتبون البرامج بلغة الآلة (Machine Language)وهي اللغة الوحيدة التي يستطيع الحاسوب فهمها وتنفيذها مباشرة، وتتكون من سلاسل طويلة من الأرقام الثنائية مثل.(10101001 ,00000001) على فرض القيام بعملية تخزين الرقم (5) في سجل معيّن داخل المعالج؛ سيكون عليك استخدام كود ثنائي معقّد مثل:

10111000 00000101

مما يجعل عملية كتابة وتطوير البرامج صعبًا باستخدام مثل هذه الطريقة حيث تعتبر هذه الطريقة:

1. صعبة القراءة والكتابة.
2. عُرضة للأخطاء البشرية الكثيرة.
3. غير مناسبة لتطوير تطبيقات كبيرة ومعقدة.

لحل هذه المشكلة، ظهرت لغات البرمجة عالية المستوى (High-Level Languages) مثل C, Python, Java, Pascalوغيرها، والتي تمكّن المبرمج من كتابة الأوامر بطريقة أقرب إلى اللغة البشرية، باستخدام كلمات مفهومة مثل...if, while, print, int, float

لكن بالرغم من سهولة هذه اللغات بالنسبة للبشر، فإن الحاسوب لا يزال لا يفهم إلا لغة الآلة، وهنا كان لا بد من وجود وسيط يقوم بترجمة الأوامر المكتوبة بلغة عالية المستوى إلى أوامر يمكن للآلة تنفيذها ويتعبر **المترجم** هو الوسيط بين اللغات عالية المستوى ولغة الالة.

1.1المترجم ((Compiler

المترجم هو برنامج يأخذ الشيفرة المصدرية المكتوبة بلغة عالية المستوى، ويحوّلها إلى شيفرة بلغة الآلة قابلة للتنفيذ مباشرة من قبل المعالج.

مثال:

```
int a = 5;
```

يتحول بواسطة المترجم إلى كود بلغة الآلة

```
MOV AX, 0005
```

ويعتبر السبب الرئيسي للحاجة الى المترجمات

1. لتمكين الحاسوب من تنفيذ أوامر مفهومة للبشر.
2. للكشف عن الأخطاء البرمجية قبل التنفيذ.
3. لتحسين أداء البرامج من خلال توليد كود عالي الكفاءة.
4. لتحقيق استقلالية عن بنية الجهاز (من خلال التمثيل الوسيط).

2.1 أهمية المترجمات في علوم الحاسوب وهندسة البرمجيات The Importance of Compilers in Computer Science and Software Engineering

المترجمات (Compilers) هي برامج متخصصة تأخذ كمدخل برنامجًا مكتوبًا بلغة برمجة عالية المستوى مثل (C, (java تُنتج كمخرج برنامجًا مكافئًا بلغة أخرى – غالبًا ما تكون لغة الآلة (Machine Language)أو ما يُعرف بـبرنامج الهدف.(Target Program)

مثال:

إذا كتبت برنامجًا بلغة C ، فإن المترجم يقوم بتحويله إلى ملف تنفيذي بصيغة exe. في نظام Windows، وهذا الملف هو الذي يُنفَّذ مباشرة من قبل المعالج.

لماذا لا نكتب مباشرة بلغة الآلة؟

لغة الآلة هي اللغة الوحيدة التي "يفهمها" المعالج، وهي مكوّنة فقط من تسلسل طويل من الأرقام الثنائية (0 و1) فكتابة البرامج بهذه الطريقة تكون:

- صعبة جدًا.
- معرّضة للأخطاء.
- لا توفّر إمكانية القراءة أو الصيانة البرمجية.
- تتطلب معرفة دقيقة ببنية المعالج ومكوناته مثل السجلات (Registers) والذاكرة المؤقتة.

ولهذا السبب، ظهرت لغات البرمجة عالية المستوى (High-Level Languages) التي تتيح للمبرمج التركيز على حل المشكلة بدلاً من التفكير بطريقة عمل الآلة.

يعتبر المترجم كوسيط بين العقل البشري والآلة حيث يعمل بمثابة "المفسر التقني" الذي يحوّل الأفكار المجرّدة المكتوبة بلغة مفهومة للبشر، إلى أوامر دقيقة تُفهم من قبل الآلة. إنه ليس مجرد أداة ترجمة، بل يُجري أيضًا فحصًا وتحققًا من صحة البرنامج، ويحسن الكود، ويتعامل مع الموارد بشكل فعال.

حيث تعتبر عملية بناء المترجم مهمة برمجية معقدة لأنه يحتوي عادةً على مئات الآلاف أو ملايين الأسطر البرمجية.

يتكوّن المترجم من عدة وحدات فرعية مترابطة مثل:

- التحليل المعجمي والنحوي والدلالي.
- توليد التمثيل الوسيط.
- تحسين الكود.
- إدارة الذاكرة.
- توليد الكود النهائي.

يعد بناء المترجم مثالًا معقدًا يُجسّد التداخل العميق بين قرارات التصميم وتأثيراتها المتبادلة على مختلف مكونات النظام.

فكل قرار يُتخذ في مرحلة معينة من مراحل بناء المترجم قد ينعكس بشكل مباشر أو غير مباشر على أداء أو تعقيد مراحل أخرى، مما يستدعي اتباع نهج متكامل في التصميم والتنفيذ.

يتطلب هذا النهج توظيف تقنيات وأساليب من مختلف فروع علم الحاسوب، مثل:

1. البرمجة الديناميكية :(Dynamic Programming) لحل المسائل القابلة للتقسيم إلى مشكلات فرعية متداخلة، كما في تحليل السياق النحوي أو تحسين الكود.

2. نظرية القواعد والتحليل النحوي :(Grammar Theory and Parsing) لتحديد البنية النحوية الصحيحة للبرامج المكتوبة بلغة المصدر.
3. خوارزميات الجدولة وتحليل تدفق البيانات (Scheduling Algorithms and Data Flow Analysis):لضمان الاستخدام الأمثل للموارد وتحسين كفاءة التنفيذ.
4. هياكل البيانات المعقدة وإدارة الموارد (Advanced Data Structures and Resource Management):لتنظيم المعلومات المتعلقة بالمتغيرات والأنواع والنطاقات وغيرها، بكفاءة عالية.

من هذا المنطلق، يمكن اعتبار المترجم نموذجًا مصغرًا يُجسّد علم الحاسوب بمختلف فروعه النظرية والعملية، مما يجعله أداة تعليمية وتطبيقية مهمة لفهم الترابط بين مفاهيم علم الحاسوب وتطبيقاته الفعلية.

3.1 أنواع اللغات البرمجية(Types of Programming Languages)

يُقسم تطور لغات البرمجة إلى ثلاث مستويات رئيسية، كل منها يقترب أكثر من طريقة فهم الإنسان ويبتعد تدريجياً عن البنية المعقدة التي يتعامل بها الحاسوب مباشرة:

أ. لغة الآلة(Machine Level Language)
- هي أبسط وأدنى مستوى من اللغات.
- تُمثل التعليمات على شكل سلسلة من الأرقام الثنائية (0 و 1).
- تعتمد على عناوين الذاكرة ومعالجة مباشرة للبيانات.
- لا يمكن للإنسان قراءتها بسهولة، لكنها تُفهم مباشرة من قِبل وحدة المعالجة المركزية.(CPU)

ب. لغات التجميع(Assembly Languages)
- تمثل تحسينًا مباشرًا على لغة الآلة.
- تستخدم رموزًا نصية (mnemonics) بدلًا من التعليمات الرقمية.
- أكثر وضوحًا من لغة الآلة لكنها لا تزال مرتبطة بالبنية المادية للمعالج.

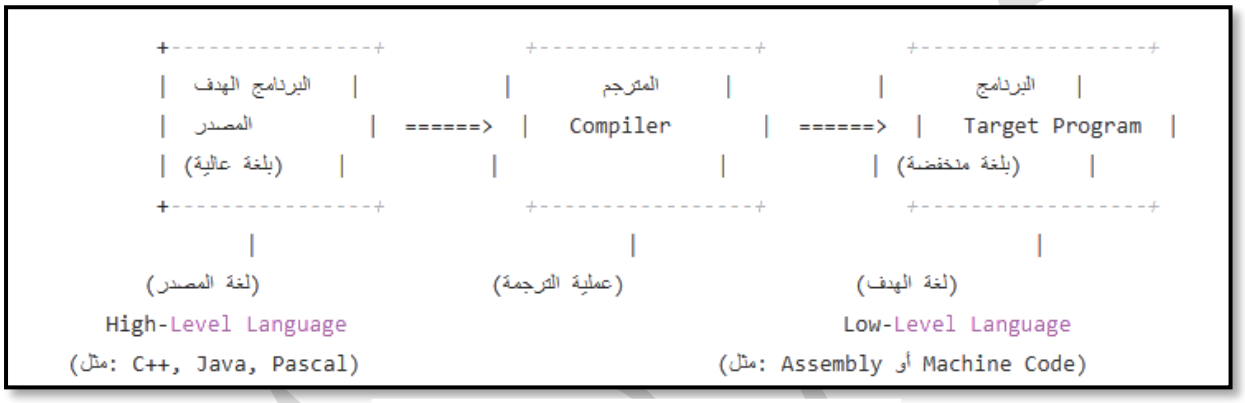
أمثلة للرموز :
- MOV → لنقل البيانات.
- ADD → للجمع.
- SUB → للطرح.
- MUL أوMULT → للضرب
- CMP → المقارنة.
- CALL → استدعاء اجراء .

ج. اللغة عالية المستوى(High-Level Language)
- أقرب ما تكون إلى اللغة البشرية.
- تعتمد على كلمات إنجليزية وأوامر منطقية واضحة.
- لا تعتمد على المعالج، ويمكن نقلها وتشغيلها على أي نظام (بعد الترجمة).

أمثلة على اللغات Java, Python ,pascal, C+

4.1المترجمات وبرامج الترجمة (Compilers and Translator Programs)

تُعرف برامج الترجمة (Translator Programs) بأنها أنظمة برمجية تُستخدم لتحويل البرامج المكتوبة بإحدى لغات البرمجة (وتُسمى لغة المصدر Source Language) إلى برامج مكافئة مكتوبة بلغة برمجة أخرى (وتُعرف بـ لغة الهدف Target Language)
ويتم هذا التحويل من خلال تطبيق قواعد محددة للحفاظ على البنية الدلالية والمنطقية للبرنامج الأصلي.
وعندما تكون لغة المصدر إحدى اللغات عالية المستوى مثل Pascal، C++، Javaوغيرها، وتكون لغة الهدف لغة منخفضة المستوى مثل لغة التجميع (Assembly Language) أو لغة الآلة (Machine Language)، فأن البرنامج الذي يؤدي هذه المهمة يعرف بالمترجم (Compiler).



مخطط يوضح عمل المترجم

يقوم المترجم بقراءة البرنامج المصدر كاملاً، ثم يُجري عليه مجموعة من التحليلات والتحويلات البنيوية والدلالية، ليُنتج في النهاية برنامجًا جديدًا بلغة الهدف، يكون قابلاً للتنفيذ مباشرة أو بعد معالجات إضافية.

الفرق بين "المترجم" و"برنامج الترجمة":

- "برنامج الترجمة" مصطلح عام يشمل أي أداة تقوم بتحويل الكود من لغة إلى أخرى.
- "المترجم" هو نوع خاص من برامج الترجمة التي تحول من لغة عالية المستوى إلى لغة منخفضة المستوى

5.1 مراحل تنفيذ البرامج المكتوبة بلغة عالية المستوى (Stages of Executing High-Level Language Programs)

إن تنفيذ البرنامج المكتوب بإحدى لغات البرمجة عالية المستوى لا يتم بشكل مباشر من قبل الحاسوب، وإنما يمر بمرحلتين رئيسيتين:

1. الترجمة:(Compilation)

في هذه المرحلة، يتم تحويل البرنامج المكتوب بلغة المصدر لغة عالية المستوى مثل C++ أو

Javaإلى برنامج مكافئ مكتوب بلغة الهدف، والتي تكون عادةً لغة منخفضة المستوى مثل لغة الآلة أو لغة التجميع. تتم هذه العملية بواسطة برنامج المترجم(Compiler) ، الذي يقوم بتحليل الكود المصدر، واكتشاف الأخطاء، ثم إنتاج ملف تنفيذي قابل للتنفيذ.

2. التحميل والتنفيذ:(Loading and Execution)
بعد ترجمة البرنامج، يتم تحميل البرنامج الهدف (Target Program) إلى الذاكرة الرئيسية للحاسوب(RAM) ، ومن ثم يبدأ نظام التشغيل بتنفيذه خطوة بخطوة باستخدام وحدة المعالجة المركزية.(CPU)

6.1 بنية المترجم(Compiler Structure)

بنية المترجم تشير إلى **الطريقة التي يُبنى بها المترجم داخليًا** من حيث المكونات (Modules) والتفاعل بينها لأداء عملية الترجمة من برنامج مكتوب بلغة عالية المستوى (High-Level Language)إلى لغة الآلة.(Machine Code)

أولًا: نظرة عامة على بنية المترجم

يمكن تمثيل بنية المترجم على شكل سلسلة من المراحل، وكل مرحلة تؤدي وظيفة معينة وتتسلم مدخلات من المرحلة السابقة وتنتج مخرجات تستخدمها المرحلة التالية.

ثانيًا: مكونات المترجم بالتفصيل

1. الواجهة الأمامية(Front-End)

تهتم بتحليل البرنامج والتأكد من صحته لغويًا ودلاليًا، وتشمل:

- Lexical Analyzer يحول النص إلى وحدات لغوية.(Tokens)
- Syntax Analyzer يبني شجرة التحليل ويتحقق من قواعد اللغة.
- Semantic Analyzer يتحقق من المعاني ويملأ جدول الرموز.
- Intermediate Code Generator يُنشئ تمثيلًا وسيطًا للبرنامج.

الهدف من الواجة الامامية : كشف الأخطاء وتنظيم البرنامج في شكل وسيط موحد.

2. الواجهة الخلفية(Back-End)

تهتم بتحويل الشيفرة الوسيطة إلى شيفرة نهائية ذات كفاءة عالية.

- Code Optimizer يحسّن الشيفرة الوسيطة لتقليل الزمن أو الذاكرة.
- Code Generator يحول التمثيل الوسيط إلى شيفرة الآلة.

الهدف من الواجهة الخلفية :إنتاج برنامج هدف عالي الكفاءة.

3. جداول البيانات الداخلية:

- جدول الرموز :(Symbol Table) يخزن معلومات عن المتغيرات والدوال.
- جدول الأخطاء :(Error Handler) يحتفظ برسائل الأخطاء وتعليمات المعالجة.

- وحدة إدارة الأنواع :(Type Checker) تتأكد من صحة الأنواع المستخدمة.

ثالثًا: أنواع البنى في تصميم المترجمات

1. المترجم أحادي الطور:(Single-Pass Compiler)

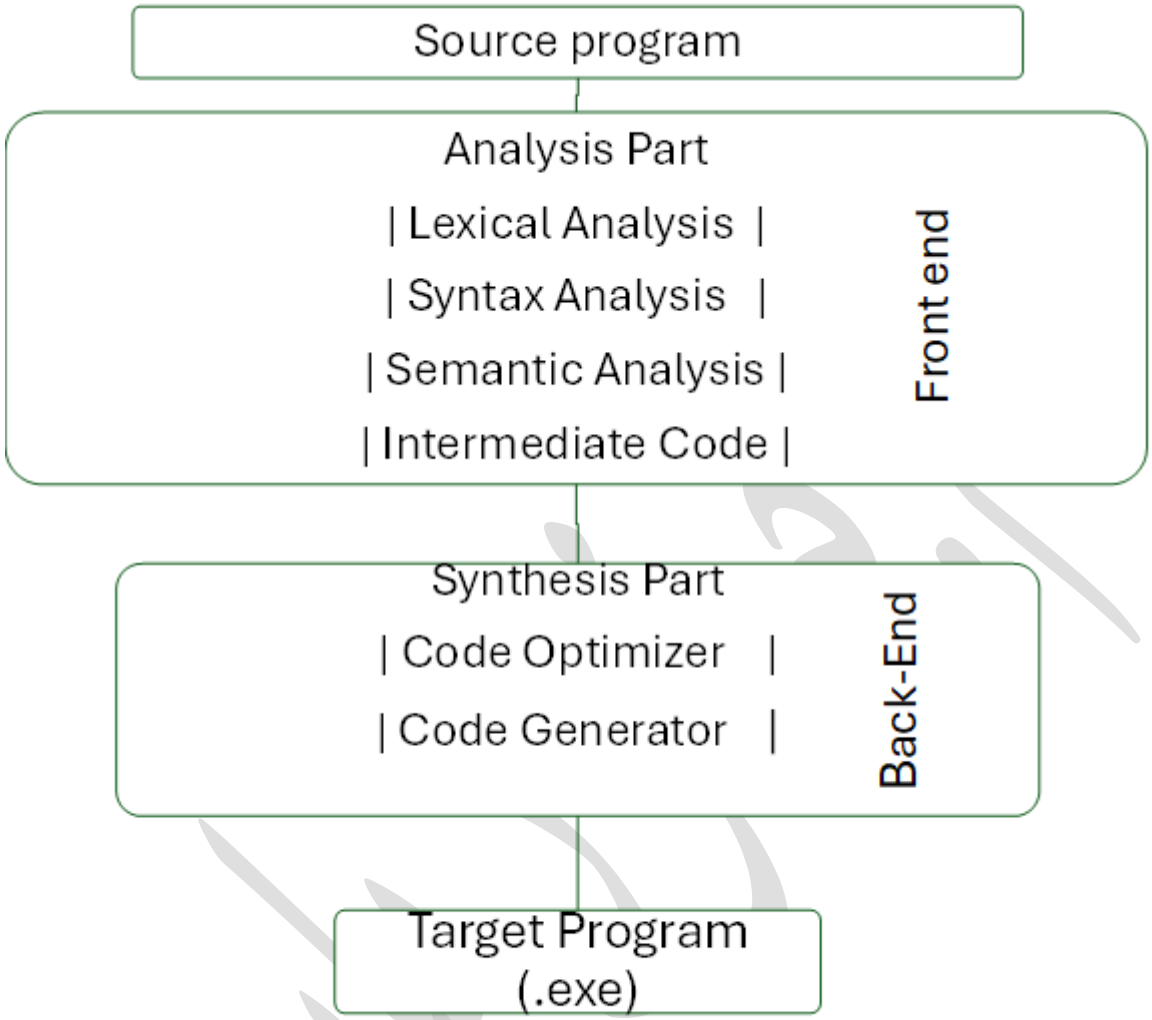
- يمر على البرنامج المصدر مرة واحدة.
- سريع وبسيط.
- مناسب للغات ذات تركيب بسيط.

2. المترجم متعدد الطور**:**(Multi-Pass Compiler)

- يمر على البرنامج في عدة مراحل.
- كل مرحلة تحلل جزءًا محددًا من اللغة.
- أكثر دقة وأفضل للأخطاء.

3. المترجم المكون من طبقات:(Layered Architecture)

- يُبنى على شكل طبقات مستقلة.
- كل طبقة تعتمد فقط على مخرجات الطبقة السابقة.
- يسهل صيانته وتطويره.



مكونات المترجم

7.1 مراحل بناء المترجم(Phases of Compiler Construction) :

1.المحلل اللفظي(Lexical Analyzer)

المسؤولية الأساسية لهذه المرحلة هي تحليل سلسلة الأحرف في برنامج المصدر وتجزئتها إلى وحدات لغوية(Tokens) ، مثل الكلمات المحجوزة(if, while, return...) ، المعرفات، الثوابت، والعلامات الخاصة (مثل +، -، =).

- يقوم بالكشف عن الأخطاء الإملائية.(Lexical Errors)
- يزيل الفراغات والتعليقات.
- يمثل كل وحدة لغوية في شكل مناسب (عادة باستخدام تركيب ثنائي: اسم الوحدة + القيمة).

مثال: int x = 10;

تم تحويله إلى وحدات لغوية مثل

<KEYWORD, int>,<IDENTIFIER, x>,<ASSIGN, =>,<NUMBER, 10>,<SEMICOLON, ;>

- اللكسيم (Lexeme) :هو سلسلة من الحروف في البرنامج المصدر، تطابق القاعدة (النمط) الخاصة بتوكن معين.
مثال : int x = 5; , الرموز ; int, x, =, 5, كل منها هو lexeme لأنه يطابق نمطًا معينًا.
- التوكن(Token):هو التصنيف العام أو النوع المجرد الذي تنتمي إليه مجموعة من اللكسيمات .
في المثال السابق:
  - int → Token type: Keyword
  - x → Token type: Identifier
  - 5 → Token type: Number
  - = → Token type: Assignment Operator

النوع	المثال	الوصف
Keyword	if, while	كلمات محجوزة
Identifier	x, count, avg	أسماء متغيرات أو دوال
Operator	+, -, *	معاملات رياضية أو منطقية
Literal	3, 'A', true	قيم صريحة
Separator	;, (,)	فواصل وعلامات بناء

- النمط Pattern هو القانون أو القاعدة التي تصف الشكل الذي يجب أن يكون عليه اللكسيم لكي يُصنف ضمن token معين. يُكتب غالبًا باستخدام تعبيرات نمطية (Regular Expressions).
مثال : على نمط لمعرف Identifier باستخدام التعبيرات النمطية (Regular Expressions):

Pattern: [a-zA-Z_][a-zA-Z0-9_]*

شرح النمط:
[_a-zA-Z]: تعني أن أول حرف في المعرف يجب أن يكون حرفًا إنجليزيًا صغيرًا أو كبيرًا، أو رمز التسطير السفلي _.
[_a-zA-Z0-9]: تعني أن ما يلي يمكن أن يكون أي عدد (بما في ذلك الصفر) من الأحرف أو الأرقام أو رمز _.

أمثلة على معرّفات تطابق هذا النمط:
x
total_sum
var123
hidden

أمثلة لا تطابق هذا النمط:
123 var(لأنه يبدأ برقم)
@ name(لأن @ غير مسموح به)

2.المحلل القواعدي(Syntax Analyzer)

يطلق عليه أحيانًا اسم "المحلل النحوي"، ويتعامل مع تسلسل الوحدات اللغوية الناتجة من المحلل اللفظي ليتحقق من صحة تركيبها وفق قواعد اللغة.(Grammar)

بمعنى اخر التحليل القواعدي هو المرحلة التي تأتي بعد **التحليل اللفظي(Lexical Analysis)** ، ويتم فيها فحص سلسلة المفردات (Stream of Tokens) الناتجة من المحلل اللفظي للتأكد مما إذا كانت هذه السلسلة **تتبع قواعد النحو (Syntax Rules)** الخاصة باللغة البرمجية.

- يقوم ببناء شجرة اشتقاق أو شجرة تحليل.(Parse Tree) :حيث يقوم بتمثيل الجملة البرمجية بشكل هرمي يُظهر كيفية اشتقاقها من القواعد النحوية,هذه الشجرة تُستخدم في المراحل اللاحقة مثل التحليل المعنوي وتوليد الشيفرة.
- يكشف عن الأخطاء القواعدية (Syntax Errors) :حيث يتأكد من أن سلسلة الـ Tokens مطابقة للقواعد النحوية الرسمية و في حال وجود خطأ نحوي (مثل نسيان فاصلة منقوطة أو قوس)، يتم إصدار رسالة خطأ مناسبة.

مثال : int = x 10;

فإن هذه الجملة تحتوي على خطأ في الترتيب، وسيرفضها المحلل القواعدي لأنها لا تطابق القواعد النحوية.

أنواع المحللات القواعدية:

النوع	أمثلة	الشرح
Top-Down	LL(1)	يبدأ من القاعدة ويحاول الاشتقاق
Bottom-Up	LR, SLR, LALR	يبدأ من الرموز ويحاول بناء القاعدة

فائدة التحليل القواعدي:

- ضمان كتابة البرامج بصيغة نحوية صحيحة.
- تمهيد الطريق أمام التحليل المعنوي وتوليد الشيفرة

3.محلل المعاني(Semantic Analyzer)

التحليل المعنوي (أو الدلالي) هو المرحلة التي تلي التحليل القواعدي(Syntax Analysis) ، ويهتم بفحص المعنى الصحيح للبرنامج بعد التأكد من سلامته النحوية.

بمعنى آخر: التأكد من أن الجمل البرمجية المكتوبة منطقية وسليمة من حيث الأنواع والتعابير، وليس فقط صحيحة نحويًا

- يكتشف الأخطاء الدلالية مثل: استخدام متغير غير معرف، أو محاولة إسناد عدد عشري إلى متغير صحيح بدون تحويل.
- يبني الجداول الدلالية مثل جدول الرموز.(Symbol Table)
- اكتشاف التصريحات المكررة أو المتضاربة.
- التحقق من صحة المتغيرات المستخدمة هل تم تعريفها؟
- ضمان المعاني الصحيحة للعمليات الحسابية والمنطقية

امثلة :
```
x = x + y;
```
إذا لم يكن المتغير y معرفًا مسبقًا، فهذه تعتبر خطأً دلاليًا.
صحيح نحويًا، لكن خطأ معنويًا:

```
int a;
float b;
a = b + "hello";   // خطأ: لا يمكن جمع float مع سلسلة نصية

int x;
 x = y + 2;        // خطأ: المتغير y لم يتم تعريفه

int a;
int a;             // خطأ: تعريف متكرر لنفس المتغير

int a;          المتغير من نوع integer
int b;          المتغير من نوع integer
 a = b + 3.5;   العملية b + 3.5 تنتج float.
```
إسناد floatإلى intقد يؤدي لفقدان دقة (خطأ أو تحذير حسب اللغة)

العمليات التي يقوم بها المحلل المعنوي:

الشرح	الوظيفة
التأكد من توافق أنواع البيانات	Type Checking
التحقق من أن كل متغير معرف في نطاقه الصحيح	Scope Resolution
حفظ المعلومات عن المتغيرات وأنواعها ومجالاتها	Symbol Table Maintenance
التحقق من وجود التعريفات لجميع الدوال أو العلامات المستخدمة	Label Checking

4.مولد الشيفرة الوسيطة(Intermediate Code Generator)

يقوم بتحويل البرنامج إلى تمثيل وسيط مستقل عن لغة الآلة النهائية، هذا التمثيل لا يُشاهد من قبل المستخدم النهائي، بل يُستخدم داخليًا من قبل المترجم لتسهيل عمليات التحسين والتوليد النهائي للكود الهدف أو نقله إلى معمارية مختلفة.
ويُعد الكود الوسيط بمثابة جسر يربط بين التحليل (اللغوي والنحوي والدلالي) وبين توليد الكود النهائي، وهو يمثل البرنامج بشكل مبسط مجرد من تفاصيل البنية الخاصة بلغة المصدر أو الهدف.
التمثيل الوسيط يكون أبسط من لغة المصدر لكنه غير خاص بأي معالج.

مثال :

```
a = b + c * d;
t1 = c * d
t2 = b + t1
a = t2
```

خصائص الكود الوسيط الجيد:

1. سهولة التكوين:(Ease of Generation)
يجب أن يكون الكود الوسيط سهل التوليد من البنية النحوية أو شجرة التحليل المجرّدة (AST)، مما يعني أنه ينبغي أن يكون قريبًا بدرجة كافية من لغة المصدر.
2. سهولة التحويل إلى الكود الهدف:(Ease of Translation to Target Code)
يجب أن يكون هذا الكود مصممًا بطريقة تجعل تحويله إلى تعليمات لغة الهدف (مثل لغة الآلة) عملية مباشرة ومنظمة، مما يُسهّل على مرحلة توليد الكود النهائي أداء مهمتها بكفاءة.

أنواع التمثيل الوسيط الشائعة:

- الكود الثلاثي:(Three-Address Code - TAC)
حيث يُعبّر عن كل تعليمة باستخدام ثلاثة عناوين مثلاً.(t1 = a + b)
- الكود التجميعي الرمزي:(Postfix / Stack Code)
يستخدم نظام المكدس في العمليات مثلاً(a b +)
- التمثيل الرسومي) مثل:SSA – Static Single Assignment
تمثيل أكثر تقدماً حيث يُعطى كل متغير قيمة واحدة فقط ويُستخدم في التحسينات المتقدمة.

خطوات توليد الكود الوسيط:(TAC)

سنقوم بتقسيم التعبير إلى عمليات بسيطة تتكون من ثلاثة عناوين كحد أقصى:

```
a=b+c*d-e
t1 = b + c
t2 = d - e
t3 = t1 * t2
a  = t3
```

- t1 = b + c → حساب الجزء الأول من المعادلة.
- t2 = d - e → حساب الجزء الثاني.
- t3 = t1 * t2 → ضرب النتيجتين.
- a = t3 → تخزين الناتج النهائي في المتغير a

المتغيرات المؤقتة t1, t2, t3 لا يراها المستخدم، بل تُستخدم فقط في الكود الوسيط داخل المترجم, هذا التمثيل يجعل من السهل تحليل الكود لاحقًا، سواءً لتحسين الأداء أو لتوليد الكود الهدف

5.محسن الشيفرة(Code Optimizer)

في هذه المرحلة، يعمل المترجم على تحسين الكود الوسيط الناتج من المرحلة السابقة، بهدف جعله أكثر كفاءة من حيث سرعة التنفيذ أو استهلاك الذاكرة، دون التأثير على صحة البرنامج أو تغيير نتيجته المنطقية.

- يقوم بإزالة العمليات الزائدة.
- يمكنه إعادة ترتيب الأوامر بطريقة أكثر فاعلية.
- يجب ألا يغير معنى البرنامج.

أمثلة على تقنيات تحسين الكود:

1. الحساب الثابت:(Constant Folding)

إذا كان هناك تعبير ثابت مثل x = 3 * 4;، يمكن للمترجم تبسيطه مباشرة إلى x = 12;

2. إزالة التعليمات الميتة:(Dead Code Elimination)

حذف التعليمات التي لا تؤثر على نتيجة البرنامج، مثل:

x = x + 0 ممكن حذف هذه التعليمة لأن ليس لها تأثير

```
int x = 10;
int y = 20;
x = 30;  // هذه التعليمة ميتة لأن القيمة 10 لم تُستخدم أبداً
int z = x + y;
```

بعد التحسين

```
int x = 30;
int y = 20;
int z = x + y;
```

القيمة الأولى المعطاة لـ x (وهي 10) لم تُستخدم قبل أن تُستبدل بالقيمة 30، لذا فإن التعليمة x = 10; تعتبر "ميتة" ويمكن حذفها.

6.مولد الشيفرة النهائية(Code Generator)

يحول الشيفرة الوسيطة (التي قد تكون محسّنة) إلى شيفرة الآلة النهائية المناسبة لمعمارية الجهاز المستهدف مثل x86 أو ARM

- يحدد التعليمات المناسبة لكل عملية.
- يدير السجلات والذاكرة.
- ينتج ملف البرنامج التنفيذي.

```
MOV R1, c
MUL R1, d
ADD R1, b
MOV a, R1
```

أهداف هذه المرحلة:

- توليد تعليمات فعّالة وسليمة التنفيذ.
- تقليل حجم الكود الناتج.
- تحسين استخدام السجلات والذاكرة.
- المحافظة على المنطق البرمجي الصحيح للبرنامج الأصلي

8.1 المفسر (Interpreter) والمترجم(Compiler)

المفسر:(Interpreter)

هو أداة برمجية تقوم بتنفيذ تعليمات البرنامج **سطرًا بسطر**، ولا تقوم بتحويل البرنامج بالكامل إلى ملف تنفيذي.

خصائص المفسر

- يقوم بتنفيذ الأوامر مباشرة عند قراءتها.
- إذا حدث خطأ، يتوقف فورًا عن التنفيذ حتى يتم تصحيح الخطأ.
- أبطأ من المترجم من حيث الأداء.
- لا يُنتج ملفًا تنفيذيًا(exe.) ، وإنما يعتمد على وجود الكود المصدر في كل مرة للتنفيذ

أمثلة على لغات تستخدم المفسر:

- Python,JavaScript, Ruby

المترجم:(Compiler)

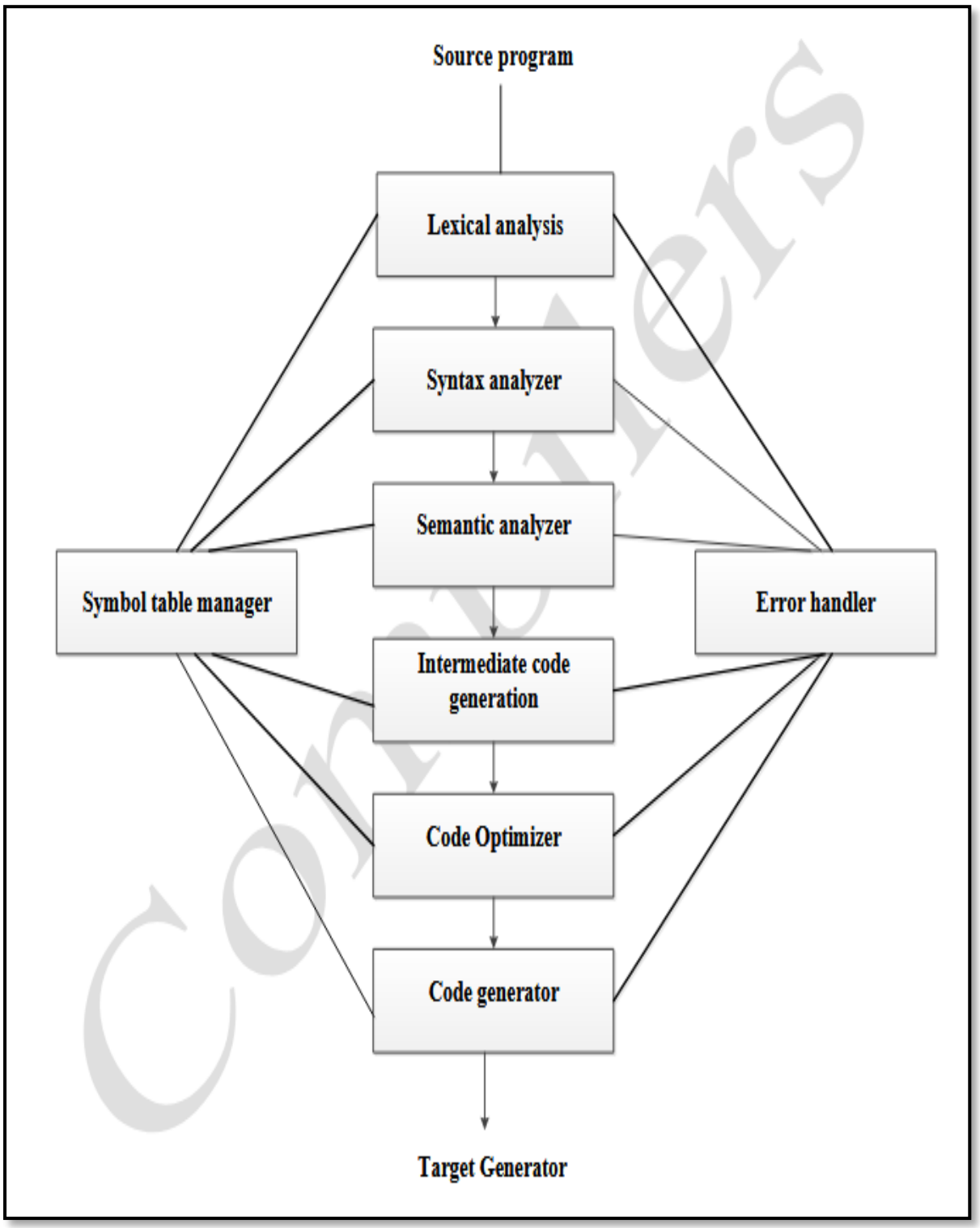
هو برنامج يقوم بتحويل كامل البرنامج المكتوب بلغة عالية المستوى إلى برنامج تنفيذي **قبل** تشغيله.

خصائص المترجم:

- يحول البرنامج المصدر إلى ملف تنفيذي مستقل مثل .(exe.)
- يقوم بفحص البرنامج بالكامل ثم يبدأ في الترجمة.
- إذا وُجدت أخطاء، يتم الإبلاغ عنها كلها دفعة واحدة أثناء وقت الترجمة.
- أسرع في وقت التشغيل، لكنه يحتاج إلى وقت مسبق للترجمة.

أمثلة على لغات تستخدم المترجم:

- C / C++,Java,Fortran



مخطط يوضح مراحل المترجم

الفرق بين المترجم والمفسر:

وجه المقارنة	المفسر(Interpreter)	المترجم(Compiler)
الوظيفة	تنفيذ البرنامج سطرًا بسطر	تحويل البرنامج بالكامل إلى ملف تنفيذي
التعامل مع الخطأ	يتوقف عند أول خطأ يظهر	يعرض جميع الأخطاء بعد الترجمة
ملف الإخراج	لا يتم إنتاج ملف تنفيذي	ملف تنفيذي مثلexe
السرعة	بطيء نسبيًا لأن التنفيذ يتم لحظيًا	سريع في التشغيل بعد الترجمة
زمن التنفيذ	Run Time يسمى بـ (وقت التنفيذ)	Compiler Time يسمى بـ (وقت الترجمة)

9.1 أمور التي ينبغي الحذر منها عند تصميم برنامج تحويل برمجي (Compiler) Key Considerations and Pitfalls to Avoid When Designing a Compiler

1. الصحة أو التصحيح (Correctness):
   - يجب أن يكون المترجم قادرًا على كشف الأخطاء في الكود المصدر بدقة.
   - ينبغي عليه أن يعطي رسائل توضيحية ومفهومة عند وجود خطأ.

   أمثلة على الأخطاء: تعريف خاطئ للمتغيرات مثل: int x, iint x ;

2. الكفاءة:(Efficiency)
   - الكفاءة تعني سرعة المترجم في تحويل الكود المصدر إلى برنامج هدف.
   - كلما قل وقت الترجمة، دلّ ذلك على أن تصميم المترجم أكثر كفاءة.
   - الكفاءة تؤثر على:
     - زمن التطوير.
     - تجربة المبرمج.

3. قابلية الاستخدام:(Usability)
   - المترجم يجب أن يكون سهل الاستخدام، حتى للمبتدئين.
   - يُفضل أن يكون قادرًا على:
     - تحديد مكان الخطأ بالضبط.
     - تقديم شرح أو اقتراح للحل إن أمكن.

مثلا إذا حدث خطأ في السطر 12، يُظهر المترجم رسالة"*خطأ نحوي*: متغير غير معرف في السطر 12."

10.1 أسبقيات العمليات (Operator Precedence)

تنتمي هذه الفقرة إلى مرحلة التحليل القواعديSyntax Analysis فعند تحليل العبارات الرياضية أو المنطقية في لغات البرمجة، من المهم أن يعرف المترجم ترتيب تنفيذ العمليات، خصوصًا عند وجود أكثر من عملية في تعبير واحد. هذا الترتيب يُعرف باسم أسبقية العمليات.(Operator Precedence)

ترتيب أسبقيات العمليات (من الأعلى إلى الأدنى):

الرقم	العملية	الرمز	الملاحظات
1	الأقواس	()	دائمًا لها الأولوية الأعلى.
2	الأس (الرفع للقوة)	^أو **	تنفذ بعد الأقواس.
3	الضرب والقسمة	*, /	تُنفذ من **اليسار إلى اليمين.**
4	الجمع والطرح	+, -	تُنفذ بعد الضرب والقسمة بنفس الاتجاه.
5	الإسناد (المساواة)	=	أقل أولوية.

ملاحظة: إذا جاءت اسبقيتان متساويتان مثل الضرب والقسمة، تكون الاسبقية التي من جهة اليسار أعلى.

حيث ان معرفة أسبقية العمليات مهمة جدًا لعدة أسباب في تصميم وتنفيذ المترجمات، وخاصة أثناء مرحلة التحليل القواعدي(Syntax Analysis) ، وها هي أهم الفوائد:

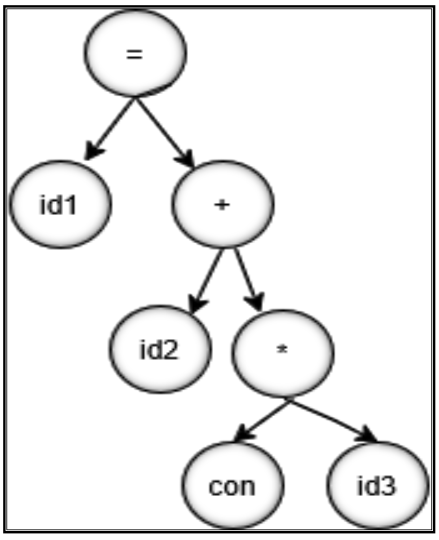
- بناء شجرة الإعراب (Parse Tree) بشكل صحيح
- تحويل التعبيرات إلى تمثيل وسيط(Intermediate Code)
- اكتشاف الأخطاء المنطقية والنحوية
- تحقيق الدقة في تنفيذ البرنامج

ملاحظة: أثناء رسم شجرة الإعراب للعمليات نبدأ من الأعلى بأقل أسبقية.

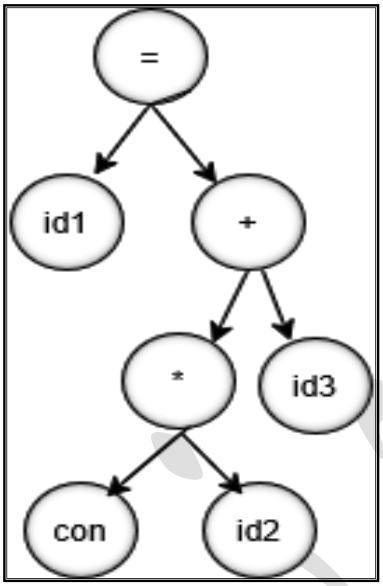
"مثال: ارسم شجرة التحليل النحوي (Parse Tree) للتعبير التالي:

1. a= b+3*d

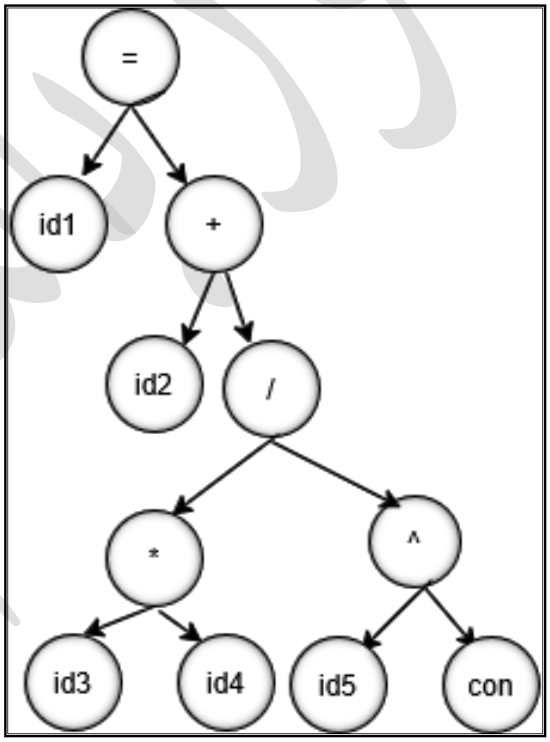
id1 = id2+ Constant * id3



2. a = 3*d + b
id1 = con * id2 + id3

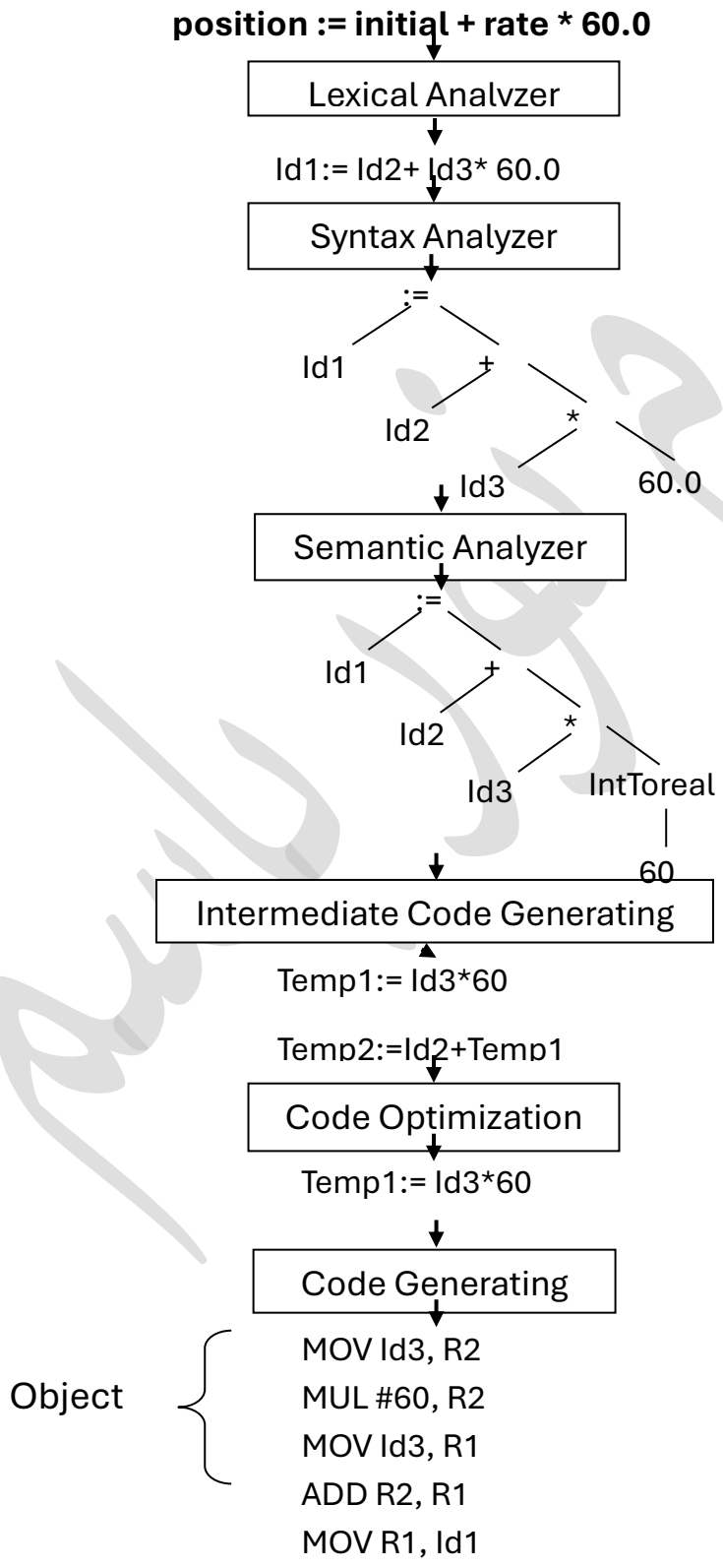


3. z= a + b * c / x^2
id1 = id2 + id3 * id4 / id5 ^ con



مثال: ترجم التعليمة التالية مروراً بالمراحل الستة للمترجم

position := initial + rate * 60.0



11.1جدول الرموز(Symbol Table)

يُعد جدول الرموز (Symbol Table) أحد الهياكل الأساسية التي يستخدمها المترجم أثناء عملية الترجمة. يتم إنشاؤه وتحديثه غالبًا خلال مرحلة التحليل اللغوي (Syntax Analysis) والتحليل الدلالي(Semantic Analysis) ، ويحتوي على معلومات مفصلة عن الرموز المستخدمة في البرنامج، مثل المتغيرات، الثوابت، الدوال، المعاملات، الكائنات، وغيرها.

جدول الرموز: هو هيكل بيانات عادة ما يكون جدولًا أو خريطة (Hash Table) يُستخدم لتخزين وتتبع معلومات كل عنصر رمزي (Identifier) يظهر في شفرة البرنامج المصدر.

المعلومات المخزنة في جدول الرموز: لكل رمز (مثل متغير أو دالة) يتم تخزين مجموعة من المعلومات، أهمها:

1. اسم الرمز(Name)
2. نوع البيانات – (Data Type) مثل int, float, bool
3. نوع الكيان – (Kind) هل هو متغير، دالة، ثابت، كائن...
4. النطاق – (Scope) هل هو محلي أم عام
5. الموقع في الذاكرة(Memory Location or Offset)
6. عدد المعاملات (إذا كانت دالة)
7. قيمته الابتدائية (إذا وُجدت)

وظائف جدول الرموز:

- التحقق من إعادة التصريح الخاطئ لنفس المتغير.
- التأكد من تطابق الأنواع في العمليات.(Type Checking)
- توفير المعلومات اللازمة خلال توليد الكود.
- تتبع النطاقات (Scopes) عند وجود متغيرات محلية وعالمية.
- المساعدة في توليد تقارير الأخطاء الدلالية.

مثال : ارسم مخططًا لجدول الرموز المرجعي المتقاطع **across reference** الذي سينتج عند ترجمة مقطع البرنامج التالي؟"

```
void main() {
   int i, j[5];
   char c, index[5][6], block[5];
   float f;
   i = 0;
   i = i + k;
   f = f + i;
   c = 'x';
  block[4] = c;
}
```

الاسم (Name)	النوع (Type)	الشكل (Form)	المجال (Scope)	سطر التعريف	أسطر الاستخدام
i	int	متغير (Scalar)	main	2	5, 6
j	int	مصفوفة بحجم 5	main	2	-
c	char	متغير (Scalar)	main	3	8, 9
index	char	مصفوفة D [5][6]2	main	3	-
block	char	مصفوفة بحجم 5	main	3	9
f	float	متغير (Scalar)	main	4	7
k	غير معروف (✗)	غير معرّف في البرنامج	—	✗	6 (✗ خطأ)

ملخص الأخطاء والتحذيرات:

الحالة	الرسالة/الوصف	السطر	نوع
✗ خطأ ترجمة	استخدام kغير المعرف	6	خطأ
✗ خطأ نوع	"x"سلسلة وليس حرف char	8	خطأ
⚠ تحذير	mainيجب أن يُرجع int	1	تحذير
⚠ تحذير	المتغيران j وindexغير مستخدمَين	2–3	تحذير

1.11.1 الإمكانيات المطلوبة في جدول الرموز: (Symbol Table Requirements)

يجب أن يتمتع جدول الرموز بعدة خصائص تضمن فعاليته وسرعته ودقته أثناء عمليات الترجمة المختلفة:

1. **سهولة الإدراج: (Efficient Insertion)**
   - يجب أن يتمكن المترجم من إدراج رمز جديد بسرعة.
   - مثلًا عند إعلان متغير أو دالة، يجب إضافته فورًا إلى الجدول.
2. **سرعة البحث: (Fast Lookup)**
   - أهم عملية يقوم بها جدول الرموز هي **البحث** عن رمز معين (مثل متغير أو دالة).
   - يجب أن يكون الوصول إلى معلومات أي رمز يتم في زمن قريب من الثابت. O(1)
3. **الدعم للنطاقات المختلفة: (Support for Scopes)**
   - يجب أن يُميز بين الرموز المحلية (Local) والعامة (Global).
   - كما يجب أن يدعم النطاقات المتداخلة (Nested Scopes) في اللغات الحديثة.
4. **التحقق من التكرار: (Detection of Redeclarations)**
   - ينبغي أن يتحقق إذا ما تم إعلان نفس المتغير أو الدالة مرتين في نفس النطاق.
5. **التحديث: (Modification)**
   - يجب أن يُسمح بتعديل بعض السمات، مثل تحديث نوع البيانات أو قيمة ابتدائية إذا كانت معلومة لاحقًا.

6. **أدارة أنواع البيانات:(Type Information)**
   - يجب أن يحتفظ بمعلومات نوع كل رمز ...int, float, char, struct :إلخ.
   - هذا مهم للتحقق الدلالي.(Semantic Checking)
7. **أمكانية تخزين سمات إضافية:(Attribute Handling)**
   - مثل: عدد المعاملات للدوال، ما إذا كانت دالة أو متغير، القيم الابتدائية، حجم التخزين، إلخ.
8. **دعم الهياكل المعقدة:(Composite Types)**
   - مثل الكائنات(Objects) ، الصفوف(Classes) ، الهياكل.(Structures)
   - يجب أن يكون قادرًا على تتبع السمات داخل هذه الكيانات.
9. **تكامل مع مراحل الترجمة:(Integration with Compiler Phases)**
   - يجب أن يكون متاحًا ومشتركًا مع:
     - محلل المعجم(Lexical Analyzer)
     - محلل النحو(Syntax Analyzer)
     - محلل المعاني(Semantic Analyzer)
     - مولد الكود(Code Generator)
10. **كفاءة الذاكرة:(Memory Efficiency)**
    - يجب أن يستخدم الذاكرة بشكل مناسب، خصوصًا في البرامج الكبيرة التي تحتوي على آلاف الرموز.

2.11.1 أنواع جداول الرموز

1.جدول الرموز غير المرتب Unordered symbol table: هو جدول رموز يتم فيه تخزين الرموز بدون أي ترتيب معين حيث يتم إدخال الرموز بشكل تسلسلي حسب ظهورها أثناء التحليل وعمليات البحث فيه قد تكون أقل كفاءة مقارنة بالجداول المرتبة، لأنه قد يتطلب فحص كل العناصر للعثور على رمز معين.

الخصائص:

- **الإدخال** :سريع، حيث تتم الإضافة في نهاية القائمة.
- **البحث** :قد يكون بطيئاً (في أسوأ الحالات، قد يحتاج لفحص كل الرموز).
- **البنية** :عادة ما تكون قائمة أو مصفوفة غير مرتبة.
- **الاستخدام** :يستخدم في الحالات التي يكون فيها حجم جدول الرموز صغير أو حيث لا تحتاج سرعة بحث عالية.

2.جدول الرموز المرتب / المرقم (Ordered Symbol Table) :هو أحد أنواع جداول الرموز المستخدمة في بناء المترجمات، ويُعنى بتخزين الرموز (Identifiers) التي تظهر في البرنامج بطريقة مرتبة وفق معيار معين.
تتمثل أهمية هذا النوع من الجداول في **تسريع عمليات البحث**، بالمقارنة مع الجداول غير المرتبة (Unordered Symbol Tables)، وذلك بفضل المحافظة على ترتيب الرموز عند إدخالها.

الخصائص:

- **سرعة البحث** :بسبب وجود ترتيب، يمكن استخدام تقنيات مثل **البحث الثنائي (Binary Search)**.
- **سهولة التنظيم والتحليل** :الترتيب يسهل تحليل الرموز حسب الاسم أو النوع أو السياق.
- **إدارة مرنة للمجال (Scope Management)** عند وجود ترتيب زمني أو مكاني.

طرق الترتيب الممكنة:

1. **الترتيب حسب الاسم:(Alphabetical Order)**
   تُرتب الرموز أبجديًا من A إلىZ ، مما يُسهّل عملية البحث بناءً على الاسم.
2. **الترتيب حسب نوع البيانات:(By Data Type)**
   تُجمع الرموز التي لها نفس نوع البيانات) مثلint ، float، charلتبسيط عمليات مطابقة الأنواع والتحقق من صحة العمليات.
3. **الترتيب حسب موقع التعريف في الشيفرة:(By Declaration Order)**
   تُدرج الرموز في الجدول حسب ترتيب ظهورها في الكود المصدر، مما يُفيد في تتبع السياق الزمني لتعريف الرموز، ويُساعد في الكشف عن استخدام الرموز قبل تعريفه

3.جدول الرموز المنظم على شكل شجرة(Tree Structured Symbol Table):هو نوع من جداول الرموز يُبنى باستخدام بنية **شجرة ثنائية(Binary Tree)** ، وغالبًا ما تكون **شجرة بحث ثنائية.(BST - Binary Search Tree)**
تُخزّن الرموز داخل العقد (Nodes) بطريقة تُحافظ على ترتيب أبجدي، مما يتيح البحث السريع من خلال التنقل عبر العقد اليسرى واليمنى.

هيكل كل عقدة في الشجرة يتضمن:

- اسم الرمز(Identifier Name)
- نوع الرمز(Data Type)
- المجال(Scope)
- العنوان أو المرجع(Address / Reference)
- المؤشر الأيسر :(Left Pointer) يشير إلى عقدة تحتوي رمزًا أصغر أبجديًا.
- المؤشر الأيمن :(Right Pointer) يشير إلى عقدة تحتوي رمزًا أكبر أبجديًا.

الية الترتيب في الشجرة:

- إذا كان الرمز الجديد **أصغر أبجديًا** من الرمز في العقدة الحالية يُدرج في **الفرع الأيسر** إذا كان **أكبر أبجديًا** يُدرج في **الفرع الأيمن** تُستمر العملية حتى الوصول إلى موضع فارغ مناسب.

آلية البحث في الشجرة:

- يبدأ البحث من **العقدة الجذر.(Root)**
- تتم مقارنة اسم الرمز بالرمز الموجود في العقدة:
  - إذا تطابق، يتم إيجاد الرمز.
  - إذا كان أصغر، ينتقل إلى الفرع الأيسر.
  - إذا كان أكبر، ينتقل إلى الفرع الأيمن.
- تُستمر العملية حتى العثور على الرمز أو الوصول إلى فرع فارغ (رمز غير موجود).

الخصائص :

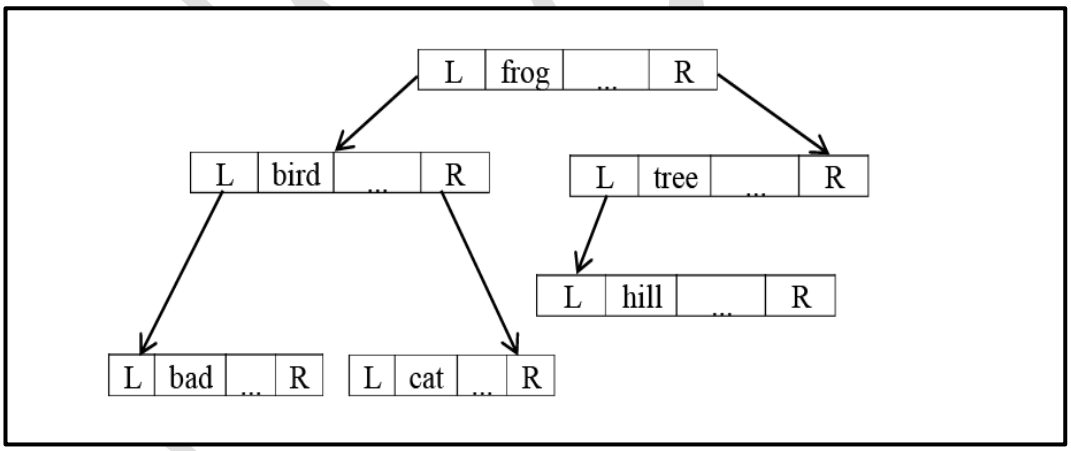
- **أداء محسن للبحث والإدراج**)في حال كانت الشجرة متوازنة.(
- **يُحافظ على ترتيب الرموز** أبجديًا.
- **مرونة** في التوسعة والتعديل دون الحاجة إلى إعادة ترتيب.

ملاحظة : التعامل مع الحروف الصغيرة والكبيرة لا فرق بينهما.

ملاحظة :المقارنة تعتمد على الأصل تبدأ المقارنة من الأعلى نزولا .

مثال : اخزن الكلمات التالية في جدول الرموز المنظم على شكل شجره

frog, tree, hill, bird, bad, cat.



4. جدول التجزئة (Hash Table) من أكثر الهياكل استخدامًا في تصميم جداول الرموز ضمن أنظمة بناء المترجمات، لما يوفره من سرعة كبيرة في الوصول والإدراج والحذف مقارنة بالأنواع الأخرى، خصوصًا عند التعامل مع عدد كبير من الرموز وهو بنية بيانات تعتمد على **دالة تجزئة (Hash Function)** لتحويل أسماء الرموز (Identifiers) إلى مؤشرات في جدول ثابت أو ديناميكي. ويتم تخزين الرمز في الموقع الناتج عن دالة التجزئة، مما يتيح الوصول إليه مباشرة دون الحاجة إلى البحث الخطي أو الثنائي.

آلية العمل:

1. **يتم تمرير اسم الرمز (Identifier Name)** إلى دالة التجزئة.
2. **تحسب دالة التجزئة قيمة عددية (Index)** تمثل موقع الإدخال في الجدول.
3. **يُخزن الرمز في هذا الموقع** مع باقي معلوماته (النوع، المجال، العنوان...إلخ).

h(s)=(ASCII(s[0])+|s|) mod Hash_max

ASCII(s[0] القيمة الرقمية (ASCII) للحرف الأول في اسم المتغير .s

|s|: عدد الأحرف في اسم المتغير s

mod عملية باقي القسمة لتحديد موقع الإدخال داخل حدود جدول التجزئة

ملاحظة : يوجد فرق في التعامل مع الحروف الصغيرة والكبيرة حيث لكل حرف . ASCII Code

مثال : frog,tree,hill,bird,bad,cat

Hash(frog)=(4+102)%6= 4

Hash (tree)=(4+116)%6=0

Hash(hill)=(4+104)%6=0

Hash (bird)=(4+98)%6=0

Hash (bad)=(3+98)%6=5

Hash(cat)=(3+99)%6=0

كيفية حساب باقي القسمة

الأولى : 6/106 =17

102=6*17

4=102-106

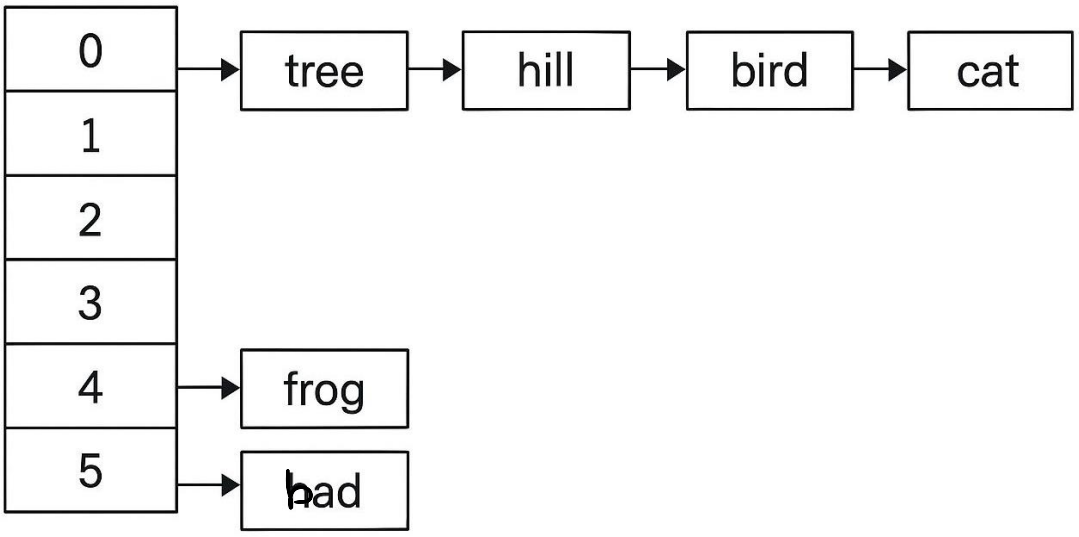
الثانية :

17=6/106.66666667

17=6/106.66666667*6=4

Ascii Code (A)=65

Ascii Code (a)=97



12.1 معالج الأخطاءError Handler

يُعد **معالج الأخطاء** أحد المكونات الحيوية في تصميم المترجم، ويُستخدم للكشف عن الأخطاء في البرنامج المصدر أثناء مراحل الترجمة المختلفة، ثم التعامل معها بطريقة مناسبة، سواء بإصدار رسائل تحذيرية أو تصحيح تلقائي (إن أمكن)، أو إنهاء عملية الترجمة عند الحاجة.

معالج الأخطاء:(Error Handler)

هو جزء من المترجم يتم استدعاؤه عند اكتشاف أي خطأ في البرنامج المصدر، حيث يقوم بتوليد رسالة توضح نوع الخطأ، موقعه في الكود، وأحيانًا يقدم اقتراحًا لكيفية تصحيحه، مما يُساعد المبرمج في تتبع الأخطاء وإصلاحها بسهولة.

أنواع الأخطاء التي يتعامل معها معالج الأخطاء

1. أخطاء لغوية:(Lexical Errors)
   مثل استخدام رمز غير معرف في اللغة.(.. ,#, @)
2. أخطاء نحوية:(Syntax Errors)
   مثل نسيان فاصلة منقوطة أو قوس مغلق:
3. خطاء دلالية:(Semantic Errors)
   مثل محاولة استخدام متغير غير معلن/ ;y = 10 :خطأ y :غير معرّفة
4. أخطاء وقت الترجمة الأخرى:(Compile-Time Errors)
   كالتعارض في الأنواع أو عدد الوسائط في الدوال.

وظائف معالج الأخطاء

1. تحديد الخطأ بدقة: تحديد نوع وموقع الخطأ في الكود.
2. إظهار رسائل مفهومة: توفير رسالة تحذير أو خطأ واضحة للمبرمج.

3. محاولة الاستمرار بالترجمة: في بعض الحالات، يُكمل المترجم العمل لتحديد جميع الأخطاء دفعة واحدة.
4. اقتراح حلول ممكنة: في بيئات التطوير المتقدمة، قد يقترح المترجم طريقة تصحيح الخطأ

13.1 تخزين المؤقت للمدخلات – (Input Buffering)

في مرحلة **التحليل المعجمي**، يقوم المترجم بقراءة البرنامج المصدر **حرفًا بحرف** لتكوين tokens لكن قراءة كل حرف مباشرة من **القرص الصلب** (Hard Disk)تعتبر عملية **بطيئة جدًا**، لأنها تتطلب وقتًا كبيرًا لكل عملية قراءة.

ولحل هذه المشكلة، يتم استخدام **مخزن مؤقت (Buffer)** لتخزين مجموعة من الأحرف دفعة واحدة في الذاكرة، بدلًا من قراءة كل حرف لوحده. هذا يجعل التحليل أسرع بكثير. حيث نستخدم **مخزن (Buffer)**حجمه مثلاً 100 حرف، ونخزن فيه جزءًا من البرنامج. أثناء التحليل، نستخدم **مؤشرين:**

1. **rbPt بداية اللكسيم** : يشير إلى أول حرف من token الذي نحاول تجميعه.
2. **rfPtنهاية اللكسيم :** يتحرك إلى الأمام حرفًا حرفًا حتى نجد token الكامل.

ولكن اذا تم استخدام مخزن واحد(One Buffer) سيكون هناك مشكلة اذا كان token طويلًا وامتد حتى نهاية المخزن، فعند إعادة تحميل المخزن من الملف، قد تُفقد الأحرف السابقة التي كنا نحتاجها لإكمال .token بمعنى اخر لو كنا ما زلنا نقرأ token وجاء حرف جديد، فقد يُكتب فوق الحروف التي لم نستخدمها بعد وحلا لهذ المشكلة تم استخدام مخزنين:(Two Buffer Scheme)

يتم تقسيم المخزن إلى جزأين:

- عند امتلاء أحد الأجزاء، يتم تحميل الجزء الثاني دون حذف الأول.
- وبهذا، لو امتد token بين الجزأين، لن نفقد أي حرف منه.