بسم الله الرحمن الرحيم

مفـاهيـــم لـغـــــات البـرمجـة
(Concepts of Programming Language)
C. P.L.

إعداد:

عبد الفتاح عبد الرب المشرقي

Preliminaries أساسيات

Reasons for Studying Concepts of Programming Languages أسباب دراسة مفاهيم لغات البرمجة

سنسرد في البداية بعضاً من أهم الأسباب والفوائد من وراء دراسة مفاهيم لغات البرمجة، وهي على النحو التالي:

- **Increased capacity to express ideas** ♣ **الحاجة المتزايدة للتعبير عن الأفكار**

من المسلم به أن العمق الذي نفكر فيه يتأثر بالقوة التعبيرية للغة التي نتواصل بها مع الآخرين. فأولئك الذين لديهم إدراك محدود للغة الطبيعية يكونون محدودين في تعقيد تفكيرهم. بمعنى آخر، من الصعب على الناس تصور التراكيب التي لا يقدرون على وصفها شفوياً أو كتابياً. والمبرمجين كذلك مقيدين بنفس الطريقة، فاللغة التي يستخدمونها في تطوير البرمجيات، تضع قيوداً على أنواع بنى التحكم، وبنيات البيانات والأفكار التجريدية التي يمكنهم استخدامها؛ وبالتالي فإن الخوارزميات التي يضعونها تكون محدودة أيضاً.

يمكن للغات برمجة معينة لا تدعم بنية برمجية معينة أن تعمل محاكاة لهذه البنية باستخدام مكونات اللغة المتوفرة. مثال على ذلك:

لغة FORTRAN 90: تزود بمجموعة من الدوال لمعالجة السلاسل النصية string مثل دالة البحث في سلسلة نصية جزئية INDEX. بينما لغة Pascal: يمكن أن نبني فيها برامج فرعية للتزويد بمثل هذه العمليات.

- **Improved background for choosing appropriate languages** ♣ **تحسين الخلفية اللازمة لاختيار اللغات المناسبة**

لدى العديد من المبرمجين المحترفين المعرفة الأساسية بعلوم الحاسوب، وذلك لأنهم تعلموا لغات البرمجة بأنفسهم. ومثل هؤلاء المبرمجين يتعلمون فقط لغة أو لغتين لهما علاقة مباشرة بالعمل الخاصة بالمؤسسة التي ينتمون إليها.

وهناك مبرمجين آخرين تلقوا معلوماتهم التدريبية عن لغات البرمجة منذ أمد بعيد، وبالتالي: فإن اللغات التي استخدموها لم تعد مستخدمة الآن، وكذلك ظهرت مفاهيم برمجية لم تكن موجودة على عهدهم. ونتيجة لهذه الخلفية، فإن العديد من هؤلاء المبرمجين عندما يعطوا لغات متعددة كخيارات لإنشاء مشروع ما، فإنهم يختارون اللغة التي اعتادوا على التعامل معها سابقاً، حتى لو كانت هذه اللغة فقيرة بالأدوات التي تمكنهم من تنفيذ هذا المشروع. لو كان هؤلاء معتادون على استخدام لغات أخرى متوفرة، فإنهم سوف يكونون قادرين على تبوئ مكانة أفضل.

- **Increased ability to learn new languages** ♣ **زيادة القدرة على تعلم لغات جديدة**

يعتبر مجال برمجة الحاسوب مجالاً يافعاً؛ فمنهجيات التصميم، وأدوات التطوير البرمجي، ولغات البرمجة ما زالت في تطور مستمر. وهذا يجعل تطوير النظم حرفة مثيرة، ولكن هذا يعني أيضاً أن التعلم المستمر شيء أساسي في مجال الحاسوب.

كلما كان لدى المبرمج فهماً أوسع لمفاهيم اللغات البرمجية المختلفة، أصبح من السهولة بمكان رؤية كيف يمكن تركيب هذه المفاهيم عند تصميم اللغات المختلفة.

مثال: المبرمجون الذين لديهم مفهوم "تجريد البيانات" data abstraction، سيحتاجون وقتاً أقل لتركيب الأنواع البيانية المجردة في لغة JAVA، وبشكل أفضل من أولئك الذين ليس لديهم أدنى رؤية لهذا المفهوم. وهذه الظاهرة تنطبق أيضاً على اللغات الطبيعية. فكلما كان فهمك أكثر لقواعد اللغة الأصلية، كان من الأسهل عليك تعلم لغة طبيعية ثانية.

♣ الفهم الأفضل لأهمية التنفيذ ♣ Better understanding of the significant of implementation

عند تعلم مفاهيم لغات البرمجة، من الممتع والضروري جداً التطرق للقضايا التنفيذية التي تؤثر في هذه المفاهيم. وهذا الفهم للقضايا التنفيذية يقودنا إلى استنباط الفوائد التالية:

1- فهمنا للقضايا التنفيذية يقودنا لفهم لماذا صممت لغات البرمجة بالكيفية التي هي عليها حالياً. وهذا بدوره يقودنا إلى المقدرة على استخدام هذه اللغات بشكل أكثر ذكاءً.
2- هناك بعض الأخطاء البرمجية التي لا يفهمها إلا المبرمج الذي لديه معرفة كافية بتفاصيل التنفيذ.
3- فهمنا للقضايا التنفيذية يسمح لنا أن نتصور كيف ينفذ الحاسوب البنيات البرمجية المختلفة.

مثال: المبرمجون الذين لديهم معرفة قليلة عن كيفية تنفيذ "النداء الذاتي" recursion، غالباً لا يعلمون أن خوارزمية النداء الذاتي أبطأ من الخوارزمية التكرارية.

♣ زيادة القدرة على تصميم لغات جديدة ♣ Increased ability to design new languages

التعلم الأكثر تعمقاً للغات البرمجة يساعد في تصميم أنظمة معقدة، وكذا يساعد المستخدمين على اختبار وتقييم هذه المنتجات.

مثال: أغلب الأنظمة البرمجية تتطلب من المستخدم أن يتفاعل بطريقة ما، قد يكون هذا التفاعل عن طريق إدخال مجموعة من البيانات والأوامر، وفي مثل هذه الأنظمة يكون شكل المدخلات بسيطاً جداً. وفي الجانب الآخر، لدينا أنظمة معقدة يتطلب على المستخدم فيها التنقل بين مستويات عديدة من "القوائم" menus، وعليه كذلك إدخال أوامر مختلفة ومتعددة. كمثال على ذلك "أنظمة معالجة النصوص" word processing systems.

♣ التحسين الكلي للحوسبة ♣ Overall advancement of computing.

بالرغم من أنه من الممكن تحديد سبب شعبية لغة برمجية معينة، فإنه من الواضح دائماً أن لغات البرمجة الأكثر شعبية هي الأكثر توفراً. ولكن في بعض الحالات، يمكن القول أن شهرة بعض اللغات قد يكون بسبب أن هؤلاء الذين يختارون هذه اللغات ليسوا على دراية كافية بالمفاهيم الأساسية للغات البرمجة.

مثال: العديد من الناس يؤمنون أنه كان من الأفضل لو حلت ALGOL 60 محل FORTRAN في بداية الستينات، وذلك لأنها أكثر ذكاءً ولديها بنيات تحكم أفضل من FORTRAN. والحقيقة أن العديد من المبرمجين ومديري تطوير البرمجيات في ذلك الوقت لم يكونوا على فهم واضح للتصميم المفاهيمي لـ ALGOL 60. فقد كان وصف هذا اللغة صعب القراءة والفهم، ولذلك لم يشعروا بقيمة البنيات الهيكلية لهذه اللغة مثل بنيات التحكم المبنية بشكل جيد، لذا فشلوا في رؤية فوائد ALGOL 60 التي تتميز بها عن FORTRAN.

وعموماً: لو أن هؤلاء الذين يختارون اللغات البرمجية لديهم فهم أدق للمفاهيم البرمجية، فإن اللغات البرمجية ستتطور وستتم الاستفادة منها بالشكل الأمثل.

المجالات البرمجية Programming Domains

دخلت الحواسيب في عدد ضخم من المجالات المختلفة، ابتداء من منشآت الطاقة النووية وصولاً إلى خزن السجلات والمعلومات الشخصية. وبسبب هذا التنوع الكبير في استخدام الحاسوب، فقد تم تطوير العديد من لغات البرمجة والتي لها أغراض متعددة. وهنا سنناقش بعضاً من تطبيقات الحاسوب البرمجية واللغات البرمجية المتصلة بها.

التطبيقات العلمية Scientific Applications

تتطلب التطبيقات العلمية بنيات بيانية بسيطة ولكنها تتطلب عدداً كبيراً من العمليات الحسابية الحقيقية. البنيات الأكثر شيوعاً هي: "المصفوفات" arrays، وبنيات التحكم الأكثر شيوعاً هي "التكرار" looping و "الاختيارات" selections. وفي هذه التطبيقات غالباً ما تكون "الفاعلية" efficiency هي الأهم.

أمثلة: لغة FORTRAN ولغة ALGOL 60.

التطبيقات التجارية Business Applications

بدأ استخدام الحواسيب في التطبيقات التجارية في خمسينيات القرن الماضي.

أمثلة: لغة COBOL، وهي أول لغة تجارية عالية المستوى ناجحة وقد ظهرت في الستينات وما تزال اللغة الأكثر استخداماً في هذا المجال. ومن خصائصها ما يلي:

1- وجود تسهيلات لتوليد تقارير مطورة.
2- طرق دقيقة لوصف وخزن الأرقام العشرية والبيانات الحرفية.
3- القدرة على تحديد عمليات رياضية عشرية.

بالإضافة إلي لغات البرمجة ظهرت أدوات برمجية خاصة تستخدم في الحواسيب الصغيرة في المجالات التجارية، مثل "أنظمة الأوراق الانتشارية" spreadsheets systems و "أنظمة قواعد البيانات" database systems.

الذكاء الاصطناعي Artificial Intelligence

تمتاز تطبيقات الذكاء الاصطناعي بما يلي:

1- استخدام الحسابات "الرمزية" symbolic بدلاً من "الرقمية" numeric: وفي العمليات الحسابية الرمزية يتم معالجة رموز مكونة من أسماء وليس من أرقام.
2- العمليات الحسابية الرمزية تتم بشكل أكفأ عن طريق استخدام "القوائم المتصلة" linked lists وليس "المصفوفات" arrays.
3- يتطلب هذا النوع من البرمجة أحياناً مرونة أكبر من المجالات البرمجية الأخرى.

أمثلة:

1- لغة LISP: وهي "لغة وظيفية" functional language، ظهرت هذه اللغة في عام 1959م، وبهذه اللغة كتبت أغلب تطبيقات الذكاء الاصطناعي.
2- لغة Prolog: وهي "لغة منطقية" logic language، ظهرت في بداية السبعينات.

برمجة الأنظمة Systems Programming

"برمجيات النظم" systems software: هي عبارة عن "نظام التشغيل" operating system وكل أدوات الدعم البرمجي في النظام الحاسوب. ولها الخصائص التالية:

1- في الأغلب تستخدم باستمرار.
2- يجب أن تكون كفاءتها التنفيذية محسنة.

مثال: "نظام تشغيل يونكس" Unix O.S.، مكتوب بلغة C مما جعله نظام قابلاً للحمل في الآلات المختلفة، وذلك لأن لغة C تمتاز بالخصائص التالية:

1- مستوى قريب من المتدني.
2- كفاءة تنفيذية عالية.
3- لا ترهق المستخدم بقيود الحماية الزائدة.

اللغات النصية Scripting Languages

تستخدم مثل هذه اللغات عن طريق وضع مجموعة من الأوامر في ملف ثم تنفيذها.

أمثلة: لغة sh أو shell: وهي أول هذه اللغات، وقد بدأت كمجموعة صغيرة من الأوامر التي تفسر بعد ذلك كنداءات للبرامج الفرعية للنظام والتي تؤدي وظائف مفيدة مثل إدارة الملفات وتنقيتها.

لغات الأغراض الخاصة Special-Purpose Langusges

ظهرت هذه اللغات منذ أكثر من 40 سنة. وهي تخدم مجالات متعددة من مجالات الحياة، ومنها ما يلي:

1- لغات الـ RPG: وتستخدم لتوليد تقارير تجارية.
2- لغات الـ APT: لتوليد أدوات ميكانيكية قابلة للبرمجة.
3- لغات GPSS: وتستخدم في أنظمة المحاكاة.

معايير تقييم اللغة Language Evaluation Criteria

لعل الغرض الأهم من دراستنا لهذه المادة هو أن نفحص بعناية المفاهيم الأساسية للبنيات المختلفة ومقدرات لغات البرمجة، كذلك سنقوم بتقييم هذه المزايا مركزين على تأثيرها على عملية التطوير (بما في ذلك الصيانة).

سوف نوضح في هذا الجزء أهم المعايير المستخدمة لتقييم لغات البرمجة، وهي "قابلية القراءة" readability و"قابلية الكتابة" writability و"الاعتمادية" reliability.

الخصائص Characteristic	المعيار Criteria		
	قابلية القراءة Readability	قابلية الكتابة Writability	الاعتمادية Reliability
البساطة الكلية Overall Simplicity	●	●	●
قواعد التركيب Orthogonality	●	●	●
بنيات التحكم Control Structures	●	●	●
أنواع وبنيات البيانات Data types & Structures	●	●	●
تصميم الصيغة Syntax Design	●	●	●
دعم التجريد Support for Abstraction		●	●
إمكانية التعبير Expressivity		●	●
فحص البيانات Type Checking			●
التعامل مع الاستثناءات Exception Handling			●
استعارة التسمية المقيدة Restricted Aliasing			●

قابلية القراءة Readability

من أهم المعايير في تقييم اللغات والحكم عليها هو مدى السهولة التي يمكن بها قراءة البرنامج وفهمه. وتعتبر قابلية القراءة مقياساً مهماً لجودة البرامج ولغات البرمجة. ومن المهم أن نقول أن قابلية القراءة يجب أن تأخذ بعين الاعتبار ضمن نطاق أو مجال المشكلة، وفيما يلي مثال توضيحي لهذه النقطة:

مثال: إذا كان البرنامج الذي يقوم بحل المشكلة الحاسوبية مكتوباً بلغة غير مصممة لهذا الغرض الحسابي، فقد يكون هذا البرنامج غير طبيعي أو مربك، مما يجعل من الصعوبة بمكان قراءته بشكل سهل.

البساطة الكلية Overall Simplicity

تؤثر البساطة الكلية للغات البرمجة بشكل قوي على قابلية القراءة. وسنوضح ذلك من خلال النقاط التالية:

أولاً: اللغة التي لديها عدد كبير من المكونات الأساسية أصعب من حيث التعلم من تلك التي لديها عدد صغير من المكونات الأساسية. بمعنى أنه كلما زادت مزايا اللغة كان ذلك مؤثراً سلباً على قابلية القراءة.

ثانياً: "تضاعف المزايا" feature multiplicity، يعتبر خاصية تعقيدية أخرى تعيق قابلية القراءة. ونقصد بتضاعف المزايا أن يكون لدينا أكثر من طريقة لأداء المهمة الواحدة.

مثال: في لغة C++ و Java توجد أربع طرق مختلفة لزيادة قيمة متغير عددي بمقدار واحد صحيح، فلو كان لدينا متغير count وأردنا إضافة 1 إليه، فسنختار واحدة من الجمل التالية:

```
count = count + 1;

count += 1;

count++;

++count;
```

الاستقلالية Orthogonality

جاءت كلمة orthogonal من المفهوم الرياضي المسمى orthogonal vectors أي المتجهات المتعامدة، بمعنى أيضاً المستقلة عن بعضها البعض. وهذا المفهوم في لغات البرمجة معناه: يمكن تركيب مجموعة بنيات ابتدائية صغيرة نسبياً بعدد صغير نسبياً من الطرق لبناء بنيات التحكم وبنيات البيانات الخاصة باللغة. ونقص خاصية الاستقلالية يقود إلى أخطاء أو استثناءات في قواعد اللغة، وبعكس ذلك كلما زادت استقلالية اللغة زاد ذلك من القدرة على تعلمها وكذلك القابلية لقراءتها. أي تركيب ممكن للأنواع البيانية الابتدائية هو شرعي وله معنى كذلك.

مثال: افترض أن لغة برمجية ما لها أربعة أنواع بيانية ابتدائية، هي: integer و float و double و character، بالإضافة إلى نوعين آخرين هما arrays و pointers. إذا كان بالإمكان تطبيق الـ arrays و الـ pointers على نفسيهما والأربعة الأنواع السابقة، فإن عدداً كبيراً من البنيات البيانية يمكن تعريفه. مع ذلك، إذا افترضنا أن الـ pointers لا تشير إلى الـ arrays، فإن العديد من هذه الاحتمالات ستختفي.

ومفهوم الاستقلالية تفتقر إليه بعض اللغات التي تنتمي إلى اللغات عالية المستوى، كما هو الحال في لغة C في المثال التالي.

مثال: بالرغم من أن لغة C لديها نوعين من الأنواع البيانية الهيكلية، وهي "المصفوفات" arrays و "السجلات" records أو structs، فإن هناك بعض النقاط التي تتعلق بهذين النوعين من البيانات:

- يمكن إرجاع السجلات من داخل الدوال، بينما لا يمكن ذلك مع المصفوفات.
- العنصر داخل السجل يمكن أن يكون من أي نوع ما عدا void أو سجل من نفس النوع، أما بالنسبة للمصفوفات فإن أي عنصر داخلها يمكن أن يكون من أي نوع ما عدا void أو دالة.
- القيم تمرر بالقيمة pass by value إلا إذا كانت مصفوفات فإنها تمرر بالمرجع pass by reference.

من أكثر اللغات التي تتصف بالاستقلالية لغة ALGOL 68. مثلاً، لكل بنية في اللغة نوع خاص بها، وليس هناك أي قيود على هذه الأنواع.

نقص الاستقلالية أو فرطها يمكن أن يسبب المشاكل.

البعض يعتقد أن "اللغات الوظيفية" functional languages تقدم مزجاً جيداً بين "البساطة" simplicity و "الاستقلالية" orthogonality. واللغة الوظيفية، مثل لغة LISP، هي اللغة التي تحتاج بشكل أساسي لتطبيق الدوال على وسائط معطاة. وعلى النقيض من ذلك، فإن "اللغات الأمرية" imperative languages، مثل لغة C ولغة C++ و Java، تتم فيها العمليات الحسابية عن طريق المتغيرات وعبارات الإسناد.

تعتبر اللغات الوظيفية من حيث البساطة الكلية من أبسط أنواع اللغات لأنها تنجز كافة المهام عن طريق بنية واحدة عبر نداء الدوال، والتي يمكن بدورها أن تركب مع نداءات أخرى بطرق بسيطة.

جمل التحكم Control Statements

ثورة البرمجة الهيكلية في سبعينيات القرن الماضي كانت نتاجاً لنقص قابلية القراءة الذي كانت تسببه جمل التحكم المحدودة لبعض اللغات في خمسينيات وستينيات القرن الماضي.

مثال: بالتحديد، كان من المعلوم أن الاستخدام الغير المقيد و المفرط لجمل goto يقلل وبشكل كبير من قابلية البرنامج للقراءة، وذلك لأن البرنامج الذي يُقرأ من أعلى إلى أسفل أسهل من حيث الفهم من ذلك البرنامج الذي يتطلب من المستخدم عند قراءته القفز من جملة معينة إلى جملة أخرى غير مجاورة وذلك بغرض تتبع كيفية سير البرنامج.

أنواع البيانات وبنياتها Data Types and Structures

إن وجود تسهيلات كافية لتعريف الأنواع البيانية والبنيات البيانية في اللغة البرمجية يعد عاملاً مساعداً آخر مهم بالنسبة لقابلية القراءة.

مثال: افرض إننا استخدمنا نوع رقمي كراية مؤشرة وذلك لأنه لا يوجد أنواع بوليانية في اللغة التي نستخدمها. سيكون شكل الجملة لتعريف هذه الراية المؤشرة كما يلي:

```
sum_is_too_big = 1
```

ومعنى هذا العبارة غير واضح بما فيه الكفاية، بينما في اللغات التي تحتوي على أنواع بيانية بوليانية، نكتب التعبير المكافئ للتعبير السابق كما يلي:

```
sum_is_too_big = true
```

وهذه الأخيرة معناها أكثر وضوحاً.

اعتبارات صياغية Syntax Considerations

للصيغة أو لشكل العناصر في لغة البرمجة تأثير مهم على قابلية البرامج للقراءة. وفيما يلي ثلاث أمثلة لخيارات تصميم صياغي لها تأثير على قابلية القراءة.

- "شكل المعرف" identifier form:

  تقييد المعرفات بطول قصير جداً يؤثر سلباً على قابلية القراءة.

  وكمثال على ذلك: في FORTRAN 77 أقصى طول للمعرفات هو 6 حروف فقط، وهذا يجعل من الصعب استخدام أسماء ذات معنى ودلالة للمتغيرات وعناصر البرنامج الأخرى.

  وهناك مثال آخر أكثر تطرفاً: في لغة ANSI BASIC يتكون المعرف من حرف واحد فقط أو حرف واحد متبوع برقم، ولك أن تتخيل مدى صعوبة فهم معنى متغيرات البرنامج في هذه اللغة.

- "الكلمات الخاصة special words:

  يتأثر مظهر البرنامج وبالتالي قابليته للقراءة بشكل كبير جداً بأشكال الكلمات الخاصة للغات البرمجة. (ومن الأمثلة للكلمات الخاصة، begin، end، for). والمهم هنا هو طريقة صياغة جمل مركبة، أو مجموعات جملية، خصوصاً في بنيات التحكم.

  مثال: تستخدم العديد من لغات البرمجة زوج خاص من الكلمات أو الرموز لتكوين المجموعات الجملية أو ما يسمى بالـ blocks. فمثلاً لغة Pascal تتطلب استخدام الكلمتين begin و end لصياغة المجموعات في كل بنيات التحكم ما عدا بنية repeat-until حيث يمكن الاستغناء عن هاتين الكلمتين هنا (وهذا يعد مثالاً على نقص الاستقلالية). أما في لغة C فنستخدم الأقواس المعكوفة لنفس الغرض. وفي كلا هاتين

اللغتين هناك مشكلة، وهي أن كل الـ blocks تبدأ بنفس الرمز أو الكلمة وتنتهي كذلك بنفس الرمز أو الكلمة، وهذا يجعل من الصعب تحديد ما هي الـ block التي انتهت عند مصادفتنا لـ end او لـ {.
أما لغتا FORTRAN 90 ولغة Ada فقد تلافتا هذه المشكلة وجعلتا عملية بدء الـ block وإغلاقها أوضح عن طريق استخدام صيغة إغلاق مختلفة لكل نوع من أنواع الـ blocks.
مثال: تستخدم لغة Ada العبارة end if لإنهاء بنيات الاختيار، والعبارة end loop لإنهاء بنيات الدوران.
وهذا يعتبر مثالاً على التضارب بين البساطة الناتجة من استخدام كلمات محجوزة قليلة في Pascal، وقابلية القراءة الكبيرة جداً الناتجة من استخدام المزيد من الكلمات المستخدمة كما في لغة Ada.
وهناك نقطة مهمة أخرى وهي: هل يمكن استخدام هذه الكلمات الخاصة كأسماء للمتغيرات في البرنامج. إذا كان ذلك ممكناً، فإن البرامج الناتجة يمكن أن تكون مربكة جداً من حيث قابليتها للقراءة.
مثال: في لغة FORTRAN 90، يمكن استخدام الكلمات الخاصة مثل DO و END كأسماء شرعية للمتغيرات، ولذلك فإن ظهور هذه الكلمات في البرنامج يحتمل أن تكون كلمة خاصة أو اسماً لمتغير.

- ♣ "الشكل والمعنى" form and meaning:

يعني هذا، تصميم الجمل البرمجية بحيث أن ظهورها على الأقل يشير جزئياً إلى غرضها. وهذا يعد عاملاً مساعداً لقابلية القراءة. بمعنى أن المعنى أو الدلالة يجب أن يستنبطا مباشرة من الصيغة أو الشكل البرمجي للجملة.
ولكن هذا المبدأ قد ينتهك كما في اللغات التي فيها بنيتان متطابقتان في المظهر ولكن لهما معنيان مختلفان، وهذا ربما يعتمد على السياق.
مثال: في لغة C، معنى الكلمة المحجوزة static يعتمد على سياق ظهورها. فإذا استخدمت في تعريف متغير داخل دالة، فهذا يعني أن المتغير يتولد في زمن الترجمة. أما إذا استخدمت في تعريف متغير خارج نطاق كل الدوال، فهذا يعني أن المتغير مرئي فقط في الملف الذي ظهر فيه تعريفها.

قابلية الكتابة Writability

تعد قابلية الكتابة مقياساً لمدى سهولة استخدام اللغة لنوع مشكلة بعينها. كل ما يؤثر في قابلية القراءة يؤثر كذلك في قابلية الكتابة. وهذا مستنبط مباشرة من الحقيقة القائلة بأن عملية كتابة البرنامج تتطلب من المبرمج إعادة القراءة المتكررة لجزء مكتوب مسبق من البرنامج.

كما هو الحال مع قابلية القراءة، فقابلية الكتابة يجب أن تأخذ في سياق المشكلة الهدفية للغة. ببساطة، من غير المبرر مقارنة قابلية الكتابة للغتين في نطاق تطبيق معين عندما تكون إحدى هاتين اللغتين مصممة لذلك التطبيق والأخرى غير مصممة لذلك.

مثال: تختلف قابلية القراءة للغة ANSI COBOL و APL بشكل كبير جداً عند توليد برنامج للتعامل مع بنيات البيانات ثنائية البعد، حيث تعتبر COBOL غير جيدة في التعامل مع المصفوفات ثنائية البعد بخلاف APL. من جهة أخرى، تعتبر COBOL أقوى من APL فيما يخص توليد التقارير المالية لأن COBOL مصممة في الأساس لهذه المهمة.

البساطة والاستقلالية Simplicity and Orthogonality

إذا كان للغة عدد كبير من البنيات المختلفة، فقد لا يكون بعض المبرمجين معتادين عليها كلها. وهذا يمكن أن يؤدي إلى استخدام خاطئ لبعض مزايا اللغة أو عدم استعمال للبعض الآخر، وقد تكون هذه البنيات المهملة أكثر ذكاءً وفاعلية من تلك المستخدمة. كذلك، قد يكون من الممكن استخدام مزايا غير معروفة بشكل خاطئ، مما يؤدي إلى نتائج غريبة. لذلك، يعد العدد الصغير من البنيات الابتدائية والمجموعة الثابتة من قواعد تركيبها أفضل من الحصول على عدد كبير من البنيات الابتدائية.

دعم التجريد Support for Abstraction

نقصد بـ "التجريد" abstraction القدرة على تعريف ثم استخدام هياكل أو عمليات معقدة بطريقة تسمح بجعل الكثير من التفاصيل مهملة. والتجريد واحد من المفاهيم الأساسية في تصميم لغات البرمجة الحديثة. وبالتالي فإن درجة التجريد التي تسمح بها لغة البرمجة أمر مهم فيما يتعلق بقابلية الكتابة. ويمكن للغات البرمجة أن تدعم نوعين من التجريد، هما: "تجريد العمليات" process abstraction، و "تجريد البيانات" data abstraction.

مثال: استخدام "البرامج الفرعية" subprograms يعتبر مثالاً واضحاً على النوع الأول من التجريد والذي هو تجريد العمليات، مثلاً استخدام برنامج فرعي لتنفيذ خوارزمية الترتيب التي نحتاجها لمرات عديدة في برنامج ما. فبدون البرنامج الفرعي، سنحتاج إلى تكرير كتابة كود الترتيب في كل الأماكن التي نحتاج فيها إلى هذه العملية، وهذا يسبب في جعل البرنامج أكثر طولاً ويجعل من كتابته أمراً مملاً ومضجراً. والشيء الأكثر أهمية، إذا لم نستخدم برنامج فرعي، فإن كود خوارزمية الترتيب سيضطرب مع تفاصيل خوارزمية الترتيب.

مثال: استخدام "الشجرة الثنائية" binary tree يعتبر مثالاً واضحاً على النوع الثاني من التجريد والذي هو تجريد البيانات، لو أخذنا بعين الاعتبار شجرة ثنائية تخزن في "عقدها" nodes بيانات صحيحة. مثل هذه الشجرة تنفذ عادة في لغة FORTRAN 77 كثلاث مصفوفات صحيحة متوازية بحيث أن اثنين من الأعداد الصحيحة تستخدم كـ "فهارس" subscripts لتحديد العقد الطرفية. أما في لغة C++ فيمكن تنفيذ هذه الشجرة عن طريق استخدام تجريد لعقدة الشجرة في شكل class بسيط له مؤشرين وعدد صحيح، وهذا يسهل أكثر من كتابة مثل هذه البرامج.

درجة التعبير Expressivity

درجة التعبير في لغة ما يمكن أن يقصد بها عدة خصائص. فمثلاً في لغة مثل لغة APL، نعني بدرجة التعبير أن هناك العديد من "العلامات القوية" powerful operator، وهذا أدى بدوره إلى تعامل رائع مع العمليات الحسابية. وأيضاً قد تعني أن لغة ما لديها طرق ملائمة غير معقدة للقيام بالعمليات الحسابية.

مثال: في لغة C، يمكننا استخدام الجملة count++ لزيادة قيمة count بمقدار واحد بدلاً من استخدام الجملة count = count + 1، والسبب في ذلك أن الأولى ملائمة أكثر ومختصرة أكثر.

الاعتمادية Reliability

يقال أن البرنامج "معتمد عليه" reliable إذا كان يقوم بأداء وظيفته تحت ظل أي ظروف. وسنناقش في الأقسام التالية مجموعة من المزايا التي لها أثر مهم على اعتمادية البرامج في لغة معينة.

فحص الأنواع Type Checking

نقصد بـ "فحص الأنواع" type checking اختبار الأخطاء المتعلقة بالأنواع البيانية في برنامج ما. وهو عامل مهم من العوامل المؤثرة على الاعتمادية. ومن المرغوب به دائماً فحص أخطاء الأنواع في "زمن الترجمة" compile-time وليس في "زمن التنفيذ" run-time وذلك لأن الأخير مكلف أكثر. وسنناقش هذا المفهوم باستفاضة أكثر في محاضرات لاحقة.

مثال: في لغة C الأصلية، لم يكن يتم فحص "الوسطاء الفعلية " actual parameters عند نداء دالة ما، من أجل التأكد من موافقته لنوع "الوسيط الشكلي" formal parameters الموجود في ترويسة الدالة. فيمكن إرسال قيمة int إلى دالة تستقبل float دون أن يحدث أخطاء. وهذا يقود إلى مشاكل عديدة في الغالب تكون هناك صعوبة في تحديدها، وذلك لأن مثل هذه الأخطاء لا تكتشف لا في زمن الترجمة ولا في زمن التنفيذ.

التعامل مع الاستثناءات **Exception Handling**

ونقصد بهذه الميزة قدرة البرنامج على إيقاف الأخطاء (وكذلك الحالات الغير اعتيادية) ومن ثم اتخاذ الإجراءات اللازمة ومن ثم الاستمرار. وهذا الأمر كذلك داعم كبير للاعتمادية. وهناك لغات برمجية عديدة تزود بتسهيلات رائعة للتعامل مع الاستثناءات، مثل لغة Ada ولغة C++ ولغة Java.

استعارة التسميات **Aliasing**

ونقصد بذلك أن يكون لدينا مرجعين أو اسمين مختلفين لنفس المكان في الذاكرة. ومن المجمع عليه أن هذه الميزة خطرة جداً على اعتمادية لغة البرمجة.

مثال: "المؤشرات" pointers في لغة C ولغة C++ يعتبر مثالاً واضحاً على استعارة التسمية، فبإمكانك تعريف مؤشرين مختلفين يشيران إلى نفس المكان في الذاكرة.

```
int   x;    int  *xPtr, *yPtr;

xPtr = &x;

yPtr = &x;
```

قابلية القراءة وقابلية الكتابة **Readability and Writability**

تؤثر كل من قابلية القراءة وقابلية الكتابة على الاعتمادية. فالبرنامج المكتوب بلغة لا تدعم طرق طبيعية للتعبير عن الخوارزميات المطلوبة سيستخدم بالضرورة طرق غير طبيعية. وهذا يؤثر على صحة البرامج، حيث أنه كلما كانت كتابة البرنامج أسهل كلما كان تصحيحه أسهل.

قابلية القراءة تؤثر أيضاً على الاعتمادية سواء خلال طور الكتابة أو الصيانة، فكلما كان البرنامج صعب القراءة، كلما زادت صعوبة كتابته وتعديله وبالتالي الاعتماد عليه.

التكلفة **Cost**

تعتبر التكلفة كذلك معياراً من معايير تقييم لغات البرمجة، ويتم حساب التكلفة النهائية لأي لغة برمجة اعتماداً على عدة خصائص موضحة في الأسطر القادمة.

- أولاً: تكلفة تدريب المبرمجين على استخدام اللغة، وهذه لها علاقة بخاصيتي البساطة والاستقلالية بالإضافة إلى خبرة المبرمجين.
- ثانياً: تكلفة كتابة البرنامج، وهذه لها علاقة بخاصية قابلية الكتابة والتي تعتمد على قربها من تطبيق معين ذو غرض محدد.
  ويمكن تقليل التكلفة في النقطتين السابقتين عن طريق توفير "بيئة برمجة" programming environment جيدة.
- ثالثاً: تكلفة ترجمة البرنامج، وهذه التكلفة تقل كلما ظهرت مترجمات ذات كفاءة عالية.
- رابعاً: تكلفة تنفيذ البرنامج، وهذه التكلفة تتأثر بتصميم لغة البرمجة، فاللغة التي تقوم بعمليات فحص الأنواع سوف تمنع التنفيذ السريع للبرنامج، بغض النظر عن جودة المترجم.
- خامساً: تكلفة نظام تنفيذ اللغة، إذا كانت نظام تنفيذ اللغة مكلفاً أو لا يعمل إلا على هاردوير مكلف فإن قابلية انتشارها ستكون أقل. وهذا يقودنا إلى أهمية توفير مترجمات مجانية.
- سادساً: تكلفة ضعف الاعتمادية، فإن ضعف الاعتمادية يؤدي إلى تكاليف إضافية كان بالإمكان توفيرها في ظل وجود نظام معتمد عليه.
- سابعاً: تكلفة صيانة البرنامج، وتشمل التصحيحات لتلافي مشاكل سابقة، أو التعديلات لإضافة مزايا جديدة للغة.

قابلية الحمل Portability

وهي سهولة نقل البرامج من بيئة تنفيذ إلى أخرى، وتتأثر بدرجة معيرة اللغة. فهناك لغات مثل الـ BASIC مثلاً ليس لها معيارية نهائياً، وهذا يجعل من الصعب نقل برامج هذه اللغة من بيئة تنفيذ إلى أخرى. وعملية المعيرة عملية صعبة ومستهلكة للزمن، فمثلاً، بدأت جمعية معيرة اللغات بمعيرة لغة C++ في عام 1989م ولم تنته من ذلك إلا في نهاية القرن العشرين. ومن اللغات التي لديها قابلية للحمل لغة JAVA.

العمومية Generality

ونقصد بذلك إمكانية تطبيق اللغة على مدى واسع من التطبيقات.

التعريف الجيد Well-Definedness

ونقصد بذلك كمال ودقة وثيقة التعريف الرسمية للغة.

المؤثرات على تصميم اللغة Influences on Language Design

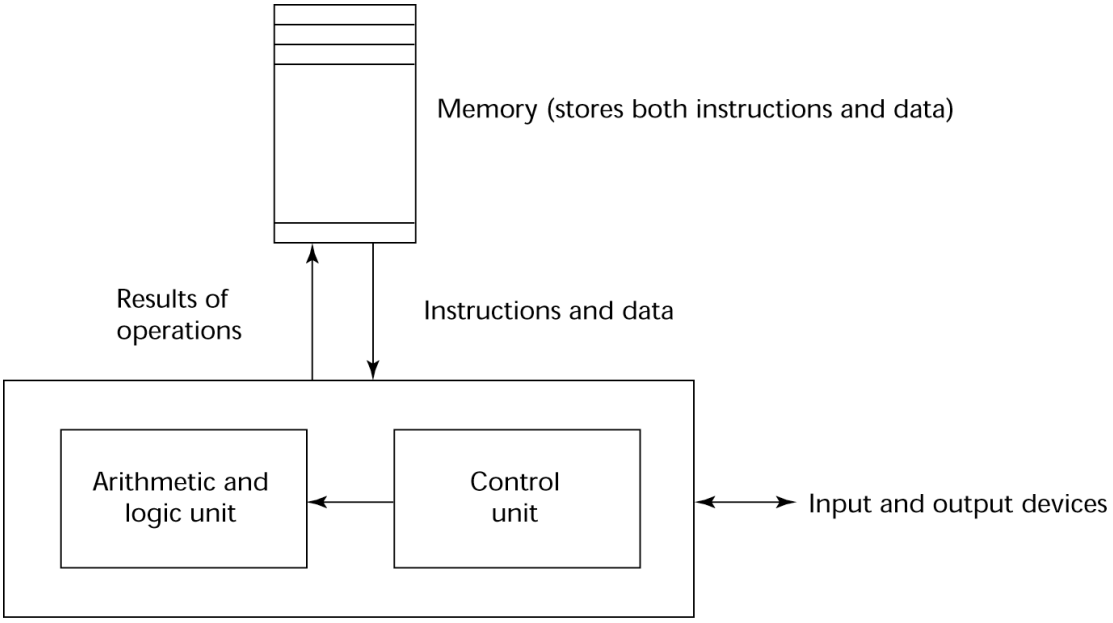
بالإضافة إلى تلك العوامل التي ناقشناها في القسم السابق، هناك أيضاً عوامل أخرى تؤثر على التصميم الأساسي للغات البرمجة. وأهم هذه العوامل معمارية الحاسوب ومنهجيات تصميم البرنامج.

معمارية الحاسوب Computer Architecture

أغلب اللغات الشعبية خلال الأربعة العقود الماضية صممت تبعاً لمعمارية حاسوبية تدعى "معمارية فون نويمان" von Neuman architecture. وهذه اللغات تسمى بـ "اللغات الأمرية" imperative language. والجدير بالذكر أن كل الحواسيب الرقمية تقريباً التي بنيت منذ أربعينيات القرن الماضي اعتمدت على معمارية "فون نويمان". وفي هذه المعمارية يتم ما يلي:

- يتم تخزين "البيانات" data و "البرامج" programs في نفس الذاكرة.
- "وحدة المعالجة المركزية" Central Processing Unit (C.P.U.) منفصلة تماماً عن "الذاكرة" memory.
- يتم نقل "التعليمات"instructions و "البيانات" data من الذاكرة إلى وحدة المعالجة المركزية.
- نتائج العمليات التي تمت في وحدة المعالجة المركزية يتم إرجاعها مرة أخرى إلى الذاكرة بغرض تخزينها.

ويمكن تصور البنية الكلية لهذه المعمارية من خلال الشكل التالي (1.1):



منهجيات البرمجة Programming Methodologies

- في خمسينات القرن الماضي وبداية الستينات: ظهرت تطبيقات بسيطة، وكان هناك قلق شديد فيما يخص "كفاءة الآلة" machine efficiency.
- في نهاية ستينات القرن الماضي: كفاءة الناس أصبحت ذات أهمية، ,أصبحت قابلية القراءة أكثر تعزيزاً من خلال ظهور بنيات تحكم أفضل. وتسمى طريقة البرمجة التي ظهرت في هذه الفترة بـ "البرمجة الهيكلية" structure programming.
- في نهاية السبعينات: ظهرت تقنية برمجة تهتم بالعمليات process-oriented و البيانات data-oriented، وهذا ما نطلق عليه "تجريد البيانات" data abstraction.
- في منتصف ثمانينات القرن الماضي: ظهرت "البرمجة كائنية التوجه" object-oriented programming، أو ما نطلق عليه اختصاراً (.O.O.P). وظهر هنا إلى جانب مفهوم تجريد البيانات مفهومين آخرين، هما: مفهوم "التوريث" inheritance ومفهوم "تعدد الأشكال" polymorphism.

تصنيف اللغات Language Categories

- **"اللغات الأمرية" imperative languages:**

المزية الرئيسية لهذه اللغات هي استخدام "المتغيرات" variables والتي تستخدم لتمثيل خلايا الذاكرة. وكذلك مما يميزها استخدام "عبارات الإسناد" assignment operators، و "الحلقات" iteration، وكأمثلة على هذا النوع نأخذ لغتي C و Pascal.

- **"اللغات الوظيفية" functional languages:**

وهذه اللغات تعتمد بشكل أساسي عند معالجتها للعمليات الحسابية على تطبيق "دوال" functions تستقبل "وسطاء" parameters وتقوم بإنجاز مهام معينة. وكأمثلة على ذلك نأخذ لغتي LISP و Scheme.

- **"اللغات المنطقية" logic languages:**

وهي لغات تعتمد على "القواعد" rules، والقواعد ليس لها ترتيب معين. ومن أشهر اللغات التي تمثل هذا التوجه البرمجي لغة "برولوج" Prolog.

- **"اللغات كائنية التوجه" object oriented programming languages:**

كما ذكرنا سابقاً، هذه اللغات تعتمد عدة مفاهيم برمجية أهمها: "تجريد البيانات" data abstraction و "الوراثة" inheritance" و "تعدد الأشكال" polymorphism. وأشهر اللغات التي تمثل هذا النوع لغتي C++ و JAVA.

- **"اللغات الوسمية" markup languages:**

وهي لغات حديثة، ظهرت مع ظهور تقنية الانترنت والصفحات الالكترونية. وهي ليست لغات برمجة بالمعنى الكامل. وتستخدم في تحديد المخطط العام للمعلومات في صفحات الوب. ومن الأمثلة عليها لغتي HTML و XML.

تباينات في تصميم اللغات Language Design Trade-Offs

- **"الاعتمادية" reliability مقابل "تكلفة التنفيذ" cost of execution:**

وهما معياران متضاربان، وكمثال على ذلك: تتطلب لغة JAVA فحص جميع مراجع عناصر المصفوفة من أجل ضمان فهرسة جيدة ولكن هذا يؤدي إلى زيادة تكلفة التنفيذ.

- **"قابلية القراءة" readability مقابل "قابلية الكتابة" writability:**

كذلك يوجد هنا تضارب بين هذين المعيارين. ويمكن توضيح ذلك من خلال هذا المثال. لغة APL تزود بمجموعة قوية من "العلامات" operators (وكذلك عدد كبير من الرموز الجديدة)، وهذا يسهل كتابة البرامج المعقدة في برنامج صغير نسبياً، ولكن قابلية قراءة هذه البرامج تقل.

- **"قابلية القراءة (المرونة)" writability (flexability) مقابل "الاعتمادية" reliability:**

كذلك يوجد هنا تضارب. المثال التالي يوضح ذلك، استخدام "المؤشرات" pointers في لغة C++ يعد من المزايا القوية والمرنة في التعامل مع الذاكرة، ولكنه مع ذلك يقلل من اعتمادية البرنامج.