الفصل الأول

مقدمة عن المشروع

Introduction

مقدمة

يهتم المشروع ببناء أدوات مساعدة في إنشاء المترجمات أو المصنفات (compilers)، و تطويرها.

أدوات المصنفات تساعد في تصميم و توليد المصنف و ذلك من خلال توليد بعض مراحل الترجمة بشكل آلي. مثل هذه الأدوات تدعى أنظمة كتابة المصنف (translator writing systems) أو مصنف المصنفات (compiler compilers) أو مولد المصنفات (compiler generators).

عادة ما تقدم هذه الأدوات خدمات كبيرة في إنشاء المحللات الخاصة باللغة المراد بناء مترجم لها، و هي بذلك توفر الكثير من الجهد و الزمن في إنشاء تلك الأجزاء. فإذا أردنا مثلاً إنشاء محلل لفظي و تركيبي للغة الباسكال مثلاً، فإن ذلك قد يتطلب ما يقارب الشهر أو أكثر من العمل المتواصل انطلاقاً من قواعد هذه اللغة، بينما و باستخدام الأدوات الخاصة بذلك، فإن العمل سوف ينجز في دقائق معدودة، و ذلك انطلاقاً من القواعد التي تصف اللغة، هذا بالإضافة إلى الأخطاء التي يمكن أن تكتشف مباشرة من خلال هذه الأدوات.

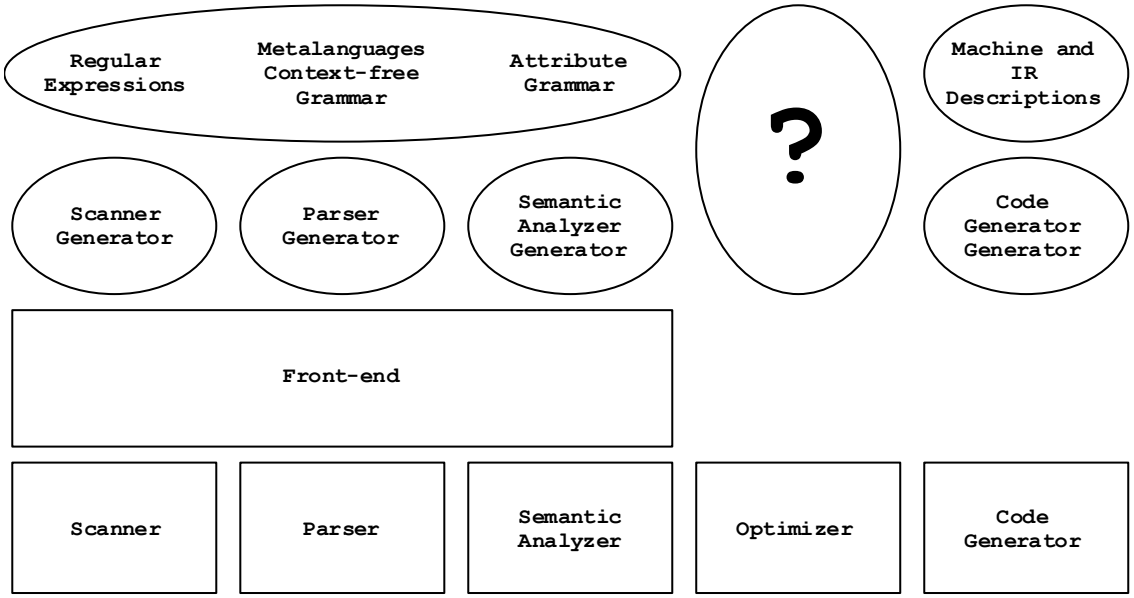
قد تساعد هذه الأدوات في بناء العديد من أجزاء المترجم و لكن حتى هذا الوقت لا توجد أدوات تساعد على بناء جميع أجزاء المترجم بشكل مؤتمت، حيث لا بد من تدخل المصمم في إنشاء بعض مراحل أو أجزاء من المترجم، و ذلك يعود إلى عدة أسباب، منها خصوصيات تعود إلى اللغة المراد إنشاء المترجم لها، و منها ما يعود إلى الغاية أو الآلة الهدف المراد إنشاء مصنف لها.

كما هو الحال بالنسبة لتقسيمات مراحل المصنف، فإن مولدات المصنفات تقسم تبعاً لهذه المراحل، فمن أجل أجزاء النهاية الأمامية للمصنف توجد أدوات التالية :

- **مولد المحلل اللفظي (lexical analyzer generator).**
- **مولد المحلل التركيبي (syntax analyzer generator) أو المُعرب (parser generator).**
- **مولد المحلل الخاص بالمعاني (semantic analyzer generator).**

أما بالنسبة فيما يتعلق بأجزاء النهاية الخلفية للمترجم، و التي تكون متعلقة بشكل كبير بآلة الهدف المراد أن ينشأ المصنف لها، فهي مازالت في موضع البحث في الوقت الحالي، و لا توجد أدوات تساعد في توليد هذه الأقسام بشكل فعال، و خاصة فيما يدعى بمولدات مولدات الشيفرة (code generator generators).

فيما يلي شكل يبين مولد المصنف حيث تظهر المستطيلات أجزاء المصنف، و تظهر الدوائر المولدات الموافقة لهذه الأجزاء. الدخل الخاص بهذه المولدات عبارة عن توصيفات مكتوبة بلغات فوقية أو سامية **(Metalanguages)**، و هي عبارة عن لغات تصف لغات أخرى، من حيث البنية اللغة النصية و التركيبية، و نلاحظ أنه فيما يتعلق بأجزاء النهاية الأمامية، فهي مجمعة في مجموعة واحدة لأنه عادة ما يكون دخل هذه الأدوات تكون على شكل ملف وحيد يصف النهاية الأمامية للمصنف بشكل كامل.



الشكل (**1–1**)

تاريخ أدوات توليد المصنفات

تزايد عدد أجهزة الكمبيوتر و كذلك عدد لغات البرمجة بشكل كبير منذ الخمسينات حتى الآن. من أجل إنشاء مصنفات لـ **N** لغة و **M** آلة هدف يتطلب الأمر إنشاء **N*M** برنامج، و ذلك في حال إنشاءها بشكل منفصل. و لكن إذا بنينا أقسام النهاية الأمامية لـ **N** لغة برمجة بحيث كل واحد منها تولد برامج بلغة وسيطية عامة، و أنشأنا نهايات خلفية من أجل كل آلة هدف، بحيث يكون الدخل النهايات الخلفية هي برامج بتلك اللغة الوسيطية، و خرجها برامج بلغة آلة الهدف. إن هذه التقنية تمكننا من إنشاء فقط **N + M** برنامج، بدلاً من **N * M**. إن اسم المتعارف عليه لهذه لغة الوسيطية هذه هي **UNCOL**، و هي اختصار لـ **UNiversal Computer Oriented Language**، وهي كما و تم ذكره لغة وسيطية و ليست أداة لتوليد المصنفات.

من الصعب جداً وضع مثل هذه اللغة الوسيطية و التي تستطيع أن تشمل في تمثيل وظائف مختلف لغات البرمجة، و تكون ملائمة في مفاهيمها مختلف آلات الهدف. إن هذه الحقيقة أبقت الحاجة الماسة إلى أدوات توليد المصنفات حتى يومنا هذا.

إن أشهر الأدوات المترجمات هي YACC (Yet Another Compiler-Compiler) و التي تم بناءها عام 1975، إن هذه الأداة تعمل وفق النظام UNIX، و وظيفتها توليد المحللات التركيبية للمصنفات، وهي مرتبطة بأداة أخرى تدعى LEX و التي وظيفتها توليد المحللات اللفظية.

اللغات السامية ***Metalanguages***

إن اللغة السامية هي تلك اللغات التي تصف لغة أخرى، وهناك العديد من اللغات السامية هذه معروفة، منها ما يسمى بالتعابير النظامية (Regular Expressions)، و التي تستخدم بشكل أساسي في وصف العناصر النصية. و BNF (Backus-Naur form) و التي تصف البنية التركيبية للغة البرمجة.

ما تم إنجازه من خلال هذا المشروع ***The Project Achievements***

لقد تم بناء أداتي توليد النهاية الأمامية هما LEX و YACC، حيث مهمة الأولى إنشاء المحلل اللفظي انطلاقاً من التعابير النظامية، و الواصفة للعناصر النصية للغة المراد إنشاء مصنف لها. و مهمة الأداة الثانية إنشاء المحلل التركيبي للغة، انطلاقاً من قواعد اللغة الخالية من السياق الواصفة للبنية التركيبية للغة البرمجة.

إن معظم الأدوات المتوافرة في هذا المجال تهتم بتوليد خرج يناسب لغة البرمجة C، و C++. بينما الأدوات التي تم تحقيقها من خلال المشروع، غير متعلقة بلغة معينة، و إنما يمكن الاستفادة منها عند استخدام اللغات الأخرى في كتابة المترجم.

إن الوظيفة الأساسية التي تقوم بها هذه الأدوات هو إنشاء جداول التفكيك من أجل لغة ما، حيث يتم استخدام هذه الجداول ضمن أطوار النهاية الأمامية للمصنف، و التي بشكل عام ذات شكل ثابت بالنسبة لجميع لغات البرمجة، و إنما الاختلاف فقط يكمن في الجداول التي تقود عملياتها. ويشمل المشروع أيضاً على الشكل العام لكل من المحلل اللفظي، و المحلل التركيبي، و قد تم إنشاءهما على أساس غرضي التوجه، ليسهل استخدامها و إعادة تعريف الوظائف الجزئية المراد تعديلها دون الرجوع إلى البرنامج الأصلي لتحقيق ذلك، أي أنها تملك صفة الـ Reusability.

من الوظائف الأخرى لهذه الأدوات هو أنها أدوات مساعدة في إنشاء قواعد لغة البرمجة ما، فهي قادرة على الكشف عن المشاكل و الغموضات التي يمكن أن تحويها اللغة أو القواعد التي يتم تطويرها.

كما و من الممكن أن يستفيد الدارس في حقل المترجمات من هذه الأدوات لبساطتها، و إمكانية الاستفادة منها في تجاربه.

الفصل الثاني

المصنفات

Compilers

مقدمة

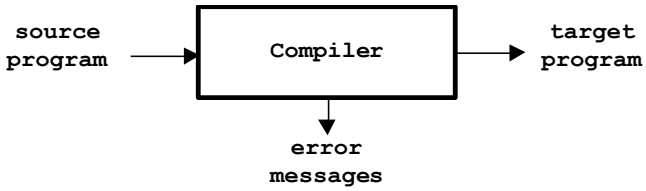
إن علم المترجمات ما هو إلا امتداد لعلوم الحاسب و التي تشمل لغات البرمجة، بنية الحاسب، نظرية اللغة، خوارزميات و هندسة البرمجيات، و قد أصبح هناك العديد من التقنيات في هذا المجال، تسمح لبناء مصنفات لشريحة واسعة من لغات البرمجة، و الآلات.

إن تطور علم المترجمات بدأ اعتباراً من بناء المترجمات الأولى وذلك في بداية الخمسينات، و من الصعب تحديد تاريخ بناء أول مترجم، فهو يعود إلى جهود كبيرة في التجارب و الإنجازات المنفردة و التي تمت بشكل أعمال انفرادية تم إنجازها من قِبل العديد من مجموعات العمل في ذلك الوقت. و معظم هذه الأعمال كانت مركزة على بناء مترجمات خاصة بالمعادلات الرياضية إلى لغة الآلة. وقد كان من المعروف في ذلك الوقت أن المترجمات هي من أعقد البرامج التي يمكن إنشاءها. فعلى سبيل المثال، تطلب بناء المترجم الأول للغة **FORTRAN** ما يقارب \18 مبرمج – السنة*\. و بعد ذلك فقط تم التوصل إلى معرفة التقنيات الأساسية لمعالجة وظائف عملية الترجمة. و منذ ذلك الوقت قد تم إنشاء العديد من اللغات و أدوات برمجة ذات اعتمادية جيدة.

ما هو المصنف أو المترجم؟

بكل بساطة، المصنف هو برنامج يقوم بقراءة برنامج مكتوب بلغة ما (لغة المصدر **source language**) و ترجمته إلى برنامج مكافئ بلغة أخرى (لغة الهدف **target language**). إن مدى الاختلاف بين البنية التركيبية للبرنامج المترجم و البرنامج الناتج من عملية الترجمة يحدد نوع المترجم، فإذا كان الاختلاف يشمل شكل الأوامر فقط دون تغيير في هيكليته، فالمترجم عندئذ يدعى مُجمع **(Assembler)** أما المصنف **(Compiler)** فهو ذاك المترجم الذي يأخذ برنامج المصدر و يعمل على تحليل البنية التركيبية له من أجل توليد برنامج بلغة أخرى تختلف في هيكليتها.

و الشكل **2–1** يبين مهمة المترجم بالشكل العام.



الشكل (2–1) المصنف أو المترجم.

نموذج التحليل–التركيب الخاص بعملية التصنيف *The Analysis-Synthesis Model of Compilation*

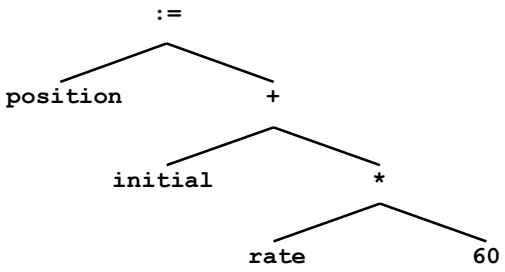
هناك جزءان أساسيان لوظيفة الترجمة هما: التحليل و التركيب.

فمهمة التحليل تقوم بتقسيم البرنامج إلى الأجزاء المكونة له، و بناء تمثيل وسيطي له. أما التركيب فيقوم بإنشاء البرنامج الهدف بالاعتماد على المعلومات الممثلة في التمثيل الوسيطي للبرنامج. إن قسم التركيب في المصنف عادة ما يتطلب تقنيات أكثر تخصصاً مما هو عليه الحال في قسم التحليل.

خلال عملية التحليل، يتم تزويد المترجم بالبرنامج المترجم (برنامج المصدر) و الذي يتم تحويله إلى تمثيل هرمي (شجري) يدعى شجرة البرنامج. غالباً ما يستخدم نوع محدد من الأشجار يدعى شجرة التراكيب **(Syntax Tree)** حيث تعبر كل عقدة فيها عن العملية، أما أبناء هذه العقدة فتعبر عن بارامترات هذه العملية. مثال ذلك، ليكن لدينا التعبير التالي :

```
Position := initial + rate * 60
```

تكون الشجرة التركيبية المعبرة عنه مبينة في الشكل 2-2.

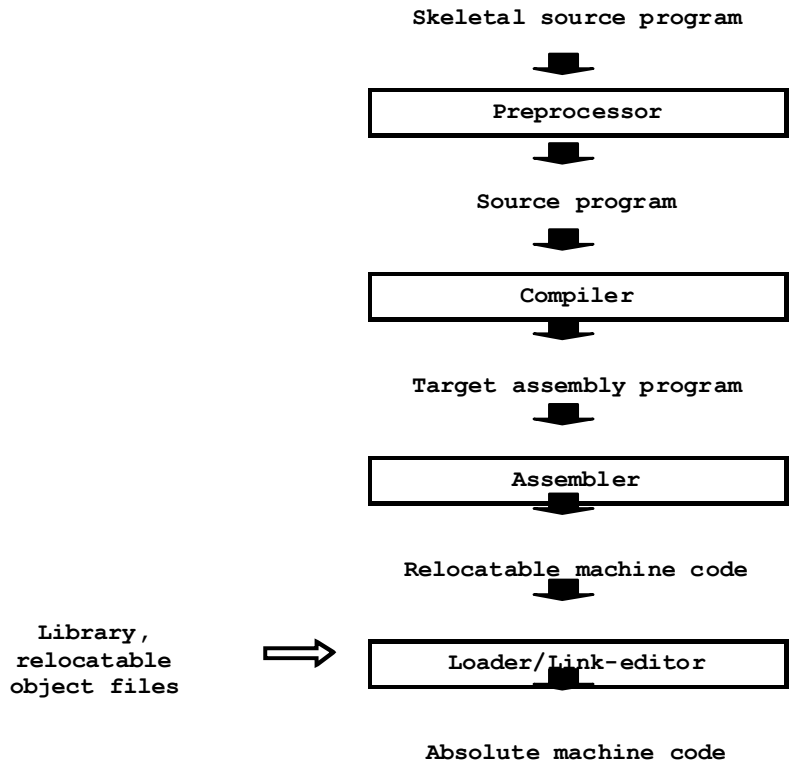


الشكل (2–2) شجرة الإعراب للتعبير الرياضي

سياق المصنف *The Context of a Compiler*

بالإضافة إلى المصنف، قد نحتاج إلى برمجيات أخرى داعمة له. و ذلك من أجل إجراء عملية الترجمة و توليد البرنامج النهائي.

قد يكون برنامج المصدر مقسم إلى موديولات تكون مخزنة في ملفات منفصلة. لهذا و من أجل إتمام عملية الترجمة، لا بد من تجميع هذه الأجزاء. و يتم هذا عادة من خلال برمجية تتفاعل مع المصنف في عملها تدعى المعالج الأولي **preprocessor**. و قد يكون من مهمة المعالج الأولي هذا بعض العمليات الأخرى مثل توسيع بعض الاختصارات (الماكروات) و دمجها في تعليمات البرنامج المصدر المُترجم.



الشكل (2–3) نظام المعالجة اللغوية

الشكل (2–3) يبين عملية التصنيف النموذجية حيث نجد أن نتيجة التصنيف قد تحتاج إلى إجراءات أخرى تطبق على خرج المصنف، حيث نجد أن خرجه هو عبارة عن برنامج بلغة التجميع يتم تجميعه باستخدام المجمع **assembler** و من ثم تتم عملية الربط بواسطة الـ **linker** مع بعض البرامج الفرعية الموجودة في المكتبات، و بالتالي توليد الشيفرة التنفيذية النهائية.

تحليل برنامج المصدر *Analysis of The Source Program*

تتم عملية التحليل وفق ثلاثة أطوار و هي :

التحليل الخطي Linear analysis: و فيه تتم قراءة سلسلة المحارف المكونة للبرنامج المصدر و ذلك وفق تسلسل ورودها من اليسار إلى اليمين (من البداية و باتجاه نهاية الدخل) و يتم تجميعها في مجموعات تدعى العناصر اللفظية tokens. حيث كل عنصر لفظي له معنى جزئي.

التحليل الهرمي Hierarchical analysis: و فيه يتم تجميع العناصر اللفظية بشكل مجموعات هرمية لتشكل معاني تجميعية، و تمثل بذلك هيكلية الدخل (أو هيكل البرنامج).

تحليل المعاني Semantic analysis: و فيه يتم إجراء عمليات اختبار محددة يتم التحقق من خلالها بأن عناصر البرنامج متلائمة مع بعضها البعض، و يتم ذلك بعد الحصول على معلومات كافية عن هيكلية البرنامج.

التحليل اللفظي *Lexical Analysis*

في المصنف، التحليل الخطي يدعى التحليل اللفظي lexical analysis أو المسح scanning. على سبيل المثال فإن المحارف المكونة لأمر الإسناد التالي:

```
position := initial + rate * 60
```

سوف تجمع في العناصر اللفظية التالية :

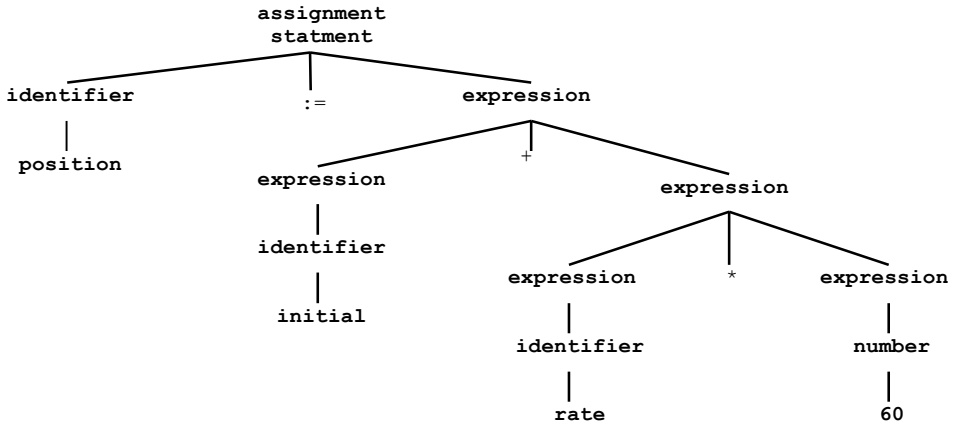
1	المميز	position
2	رمز عملية الإسناد	:=
3	المميز	initial
4	رمز الجمع	+
5	المميز	rate
6	رمز الضرب	*
7	الرقم	60

نلاحظ أن الفراغات الفاصلة بين العناصر اللفظية يتم إهمالها بشكل تلقائي من قِبل المحلل اللفظي.

التحليل التركيبي *Syntax Analysis*

يدعى التحليل الهرمي للبرنامج المصدر بالإعراب parsing أو التحليل التركيبي syntax analysis. و فيه يتم استخدام العناصر اللفظية المكونة للبرنامج بتجميعها في تكوين الجمل القواعدية و التي بدورها تستخدم في إنشاء الخرج.

عادة ما تمثل الجمل القواعدية هذه بما يسمى شجرة الإعراب كالتي مبينة في الشكل 2–4.



الشكل (2–4) شجرة الإعراب للتعبير position := initial +rate*60

نلاحظ في التعبير المذكور آنفاً أنه العبارة rate * 60 هي وحدة منطقية (على مستوى معين من مستويات التعبير الرياضي، أي اصطلاحاً يمكننا القول أنها حد من حدود عملية الجمع) لأنه و بشكل اصطلاحي في التعابير الرياضية تتم عملية الضرب قبل إجراء عملية الجمع. و بسبب مجيء عملية الضرب بعد العبارة initial + rate مباشرة فإن العبارة الأخيرة لن تكون وحدة وحيدة بل تكون على شكل وحدة محتواة في وحدة أوسع.

إن البنية الهرمية أو التركيبية للبرنامج توصف بشكل اصطلاحي من خلال قواعد لها الصفة التعاودية. فعلى سبيل المثال يمكن أن يكون لدينا تعريف جزئي القوانين التالية كجزء من تعريف للتعابير:

1. أي مميز identifier هو تعبير expression.
2. أي رقم number هو تعبير expression.
3. بفرض أن expression1 و expression2 تعبيران فإن كل من

expression1 + expression2

expression1 * expression2

(expression1)

هي عبارة عن تعابير.

القاعدتان (1) و (2) هما قاعدتان غير تعاوديتان. بينما القاعدة (3) فإنها تعرف التعبير على أنه عملية مطبقة على تعابير أخرى. و بهذا نجد أن initial و rate هما تعبيران وفق القاعدة (1)، و 60 هو تعبير وفق القاعدة (2). و بالاعتماد على القاعدة (3) يمكن أن نقول أن 60 * rate هو تعبير، و كذلك initial + rate * 60 هو أيضاً تعبير.

و يمكن أن نجد في بعض اللغات مثل الباسكال بعض التعريفات التالية:

1. **إذا كان identifier1 هو مميز identifier، و expression2 هو تعبير expression، فإن**

identifier1 := expression2

هو أمر.

2. **إذا كان expression1 هو تعبير، و statment2 هو أمر statement، فإن كل من**

while (expression1) do statment2

و

if (expression1) then statment2

أمر.

يعود إنشاء التقسيم في المصنف بين المحلل اللفظي و المحلل التركيبي إلى غاية التبسيط في العملية التحليل ككل. و ليست هناك قاعدة تحكم هذا التقسيم سوى طبيعة الوظيفة المسندة إلى كل جزئ من المصنف. فهي تتحدد فيما إذا كانت عملية التحليل ذات طبيعة تعاودية أم لا، فالتحليل اللفظي ليس له صفة التعاودية في عمله، بينما المحلل التركيبي يتطلب ذلك بشكل أساسي في عمله. فمثلاً نحن لا نحتاج التعاودية من أجل تمييز سلسلة المحارف التي تكون المميز في لغة الباسكال و الذي هو عبارة عن سلسلة أحرف أو أرقام تبدأ بحرف أبجدي، حيث يمكن أن نميز سلسلة المحارف المكونة للمميز بمجرد عملية مسح بسيطة لمحارف البرنامج المحلل إلى أن نصل إلى محرف ليس هو بحرف أبجدي أو رقم و من ثم و بتجميع هذه المحارف من أجل تشكيل العنصر اللفظي (والذي هو هنا مميز أو الاسم) و من ثم يتم تسجيل هذا المميز في جدول الرموز. و بتجاوز المحارف الممسوحة تبدأ عملية التعرف إلى العنصر اللفظي التالي في البرنامج.

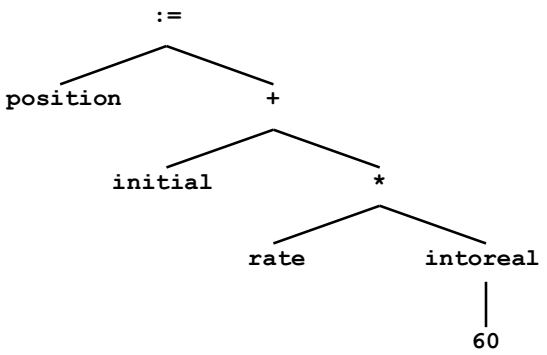
من ناحية أخرى فإن المحلل اللفظي لا يملك الإمكانية الكافية من أجل إجراء تحليل التعبير الرياضي الحاوي على الأقواس أو تراكيب الأوامر المعششة في لغة الباسكال، أي التي لها الصفة التعاودية.

تحليل المعاني Semantic Analysis

يتم في طور تحليل المعاني فحص برنامج المصدر بالبحث عن أخطاء المعاني (كالأخطاء في عدم توافقية الأنواع في عمليات النسب و التعابير الرياضية)، كما و يقوم بجمع المعلومات حول الأنواع من أجل المرحلة توليد الشيفرة اللاحقة لهذه المرحلة. في هذه المرحلة تتم الاستفادة من المعلومات التي تم الحصول عليها من مرحلة التحليل التركيبي السابقة و التي تتضمن مختلف العمليات، و المعاملات المطبقة عليها هذه العمليات.

إن أهم ما يقوم به طور تحليل المعاني هو فحص الأنواع، حيث يتم اختبار أنواع المعاملات و العمليات المطبقة عليها، و فحص التوافق بينها و ذلك بما يتوافق مع قواعد اللغة. فمثلاً في معظم لغات البرمجة يتم

التقرير عن الخطأ الناجم في استخدام متحول أو قيمة حقيقة (ذات فاصلة عائمة) في عملية فهرسة عناصر مصفوفة، أو في بعض الأحيان يتم توليد عمليات التحويل بين الأنواع المعطيات عند الضرورة، كما هو الأمر في التعابير التي تستخدم أنواع مختلفة من المعاملات (كعملية جمع بين متحول حقيقي و صحيح، حيث يتم إدراج عمليات تحويل إلى النوع الصحيح للمعامل الحقيقي، أو تحويل المعامل الصحيح إلى النوع الحقيقي).



الشكل (2–5)

فمثلاً، في ذاكرة الحاسب يتم تمثيل القيمة الحقيقية ثنائياً بشكل يختلف عن القيمة الصحيحة و إن كان لهما القيمة الجبرية ذاتها، و بالتالي، لا يمكن أن تطبق عليهما عملية تتبع لأحد النوعين داخلياً. بفرض أن جميع المميزات المستخدمة في التعبير الخاص بالمثال السابق تتبع إلى النوع الحقيقي، نجد أن القيمة **60** بحد ذاتها تمثل قيمة صحيحة، و التي هي معامل ضمن عملية ضرب مع متحول حقيقي، إن الحل في هذه الحالة هو تحويل القيمة الصحيحة إلى حقيقية (داخلياً)، و يتم ذلك من خلال إدراج عقدة جديدة إلى شجرة التعبير كما هو في الشكل **2–5**، و هذه العقدة تعبر عن عملية تحويل من النوع الصحيح إلى الحقيقي **inttoreal**. مع الملاحظة أن المعامل هنا هو ثابت رقمي صحيح، و تحويله هو ثابت أيضاً، بالتالي يمكن توليد ثابت حقيقي (من قِبل المصنف أو المترجم) ضمن البرنامج دون الحاجة إلى إدراج تعليمات التحويل في البرنامج النهائي.

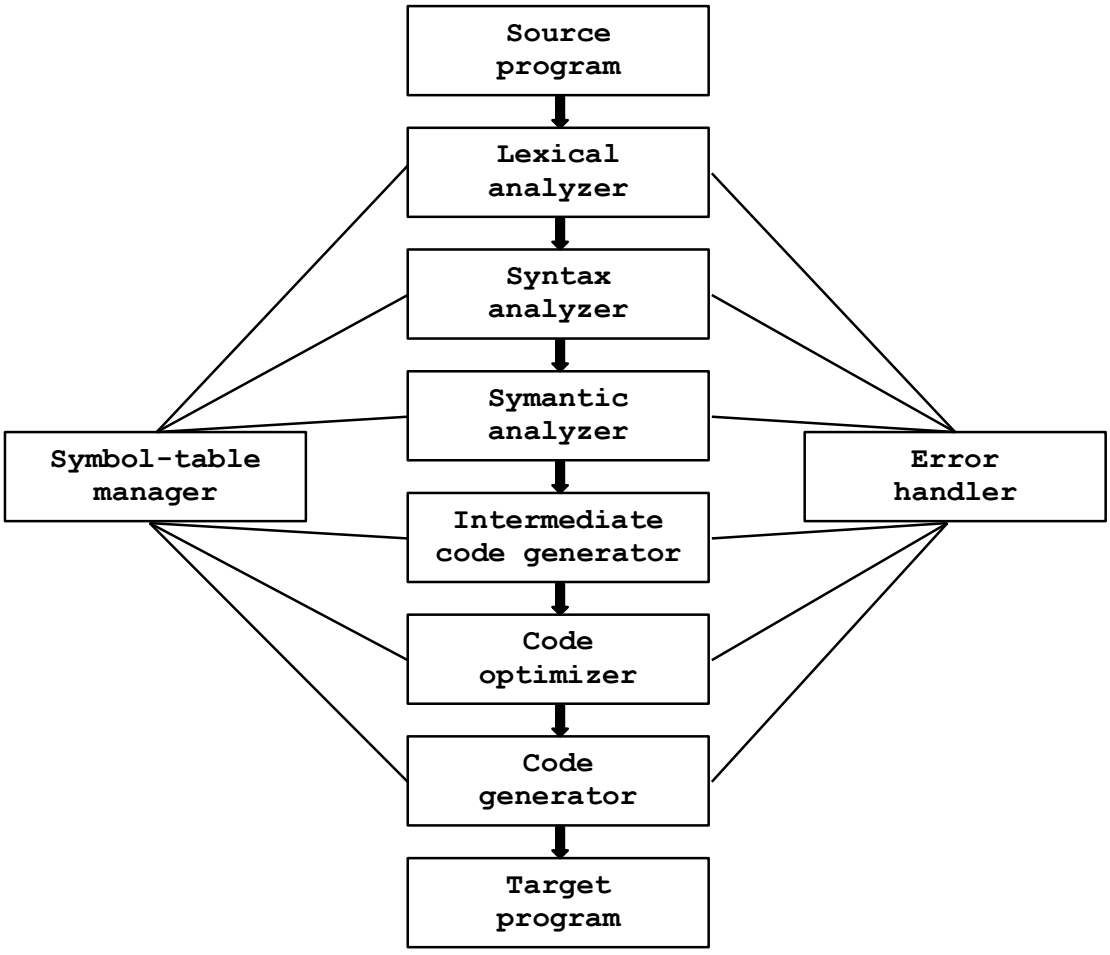
أطوار المصنف ***The Phases of a Compiler***

إن المصنف يعمل ضمن أطوار، كل طور يقوم بنقل البرنامج المترجم من تمثيل إلى آخر. التحليل النموذجي للمصنف مبين في الشكل **2–6**. من الناحية العملية يتم دمج عدة أطوار في طور واحد و بالتالي لن يكون هناك حاجة لتكوين تمثيل مرحلي ما بين الأطوار التي تم دمجها.

إن الأطوار الثلاثة الأولى تمثل قسم التحليل في المصنف، هناك أيضاً فعاليتان إضافيتان هما تنظيم جدول الرموز، و معالجة الأخطاء، و تتفاعلان مع جميع الأطوار في المصنف و التي هي التحليل اللفظي، التحليل التركيبي، التحليل الدلالي، توليد الشيفرة الوسيطية، مؤمثل (أو محسن) الشيفرة و توليد شيفرة الهدف.

إدارة جدول الرموز *Symbol-Table Management*

من الوظائف الأساسية للمصنف هي تسجيل كل المُميزات (المُعرفات أو الأسماء **Identifiers**) التي يتم التصريح عنها ضمن البرنامج و جمع مختلف المعلومات وصفات حول كل مُميز. و هذه المعلومات تصف مقدار و موقع الذاكرة التي يحجزها هذا المميز، بالإضافة إلى نوعه، مجال تعريفه ضمن البرنامج، و بعض المعلومات الخاصة بالمميز، مثلاً في حال كون هذا المميز هو اسم لبرنامج فالمعلومات ستكون وصف لبارامترات الإجراء و أنواعها، و نوع القيمة المعادة في حال كان تابع.



الشكل (**2–6**) مراحل المترجم أو المصنف

إن جدول الرموز هو بنية معطيات تحوي على سجلات من أجل كل مُميز، حيث كل سجل يحوي على حقول تحوي كافة المعلومات و الصفات الخاصة بهذا المميز. و بنية المعطيات هذه تسمح لنا بإيجاد المعلومات حول مميز ما و تعديلها بسرعة كبيرة. خلال عملية التحليل الفظي، و عندما يتم الحصول على المميز من برنامج

المصدر خلال عملية المسح، يتم إدخال هذا المميز إلى جدول الرموز مباشرة من قِبل المحلل الفظي، مع الملاحظة أن صفات هذا المميز لن تكون محددة بعد، حيث أن المحلل اللفظي ليس له القدرة على تحديد صفاته. مثلاً، ليكن لدينا السطر التالي بلغة الباسكال :

```
var position, initial, rate: real;
```

إن النوع **real** سوف لن يكون معروفاً بعد أثناء مرور المحلل اللفظي بالمميزات **position, initial, rate**. إن الأطوار الأخرى هي التي تقوم بإدخال المعلومات عن المميزات التي تم إضافتها في طور التحليل اللفظي، وهي التي تستفيد من هذه المعلومات في عملها. فالمحلل الدلالي و مولد الشيفرة الوسيطية يحتاجان إلى معرفة مختلف المعلومات حول المميزات في البرنامج من أجل إتمام عملهما. فالمحلل الدلالي يستفيد من المعلومات حول أنواع المعطيات الخاصة بالمميزات من أجل إدراج عمليات التحويل المناسبة بين الأنواع المختلفة، و كذلك يقوم مولد الشيفرة الوسيطية بالاستفادة من جدول الرموز في تسجيل مختلف المعلومات المرحلية حول المميزات و مختلف حالات مخزونها في كل نقطة من نقاط تعليمات البرنامج (في حال كانت تعبر عن متحولات).

التقصي عن الأخطاء و إنشاء التقارير عنها ***Error Detection and Reporting***

يمكن أن تحصل أخطاء ترجمة في أي طور من أطوارها. و في جميع الأحوال فإن أي خطأ يتم اكتشافه و في أي طور يجب أن يتم معالجتها بطريقة أو بأخرى. هناك مصنفات تستمر في عملية الترجمة حتى في حال ظهور بعض الأخطاء في البرنامج المترجم حيث تقوم بالتقرير عن الأخطاء المكتشفة بعد أن تمر على كامل البرنامج، بينما في البعض الآخر يتم التوقف بمجرد الوصول إلى أول خطأ في البرنامج المُترجم.

إن العدد الأكبر من الأخطاء التي يكتشفها المصنف تكون عادة متركزة في طوري المحلل التركيبي و المحلل الدلالي. فالمحلل اللفظي يكتشف الأخطاء الناجمة عن ورود محارف دخل لا تكوِّن في تشكيلها أي عنصر لفظي في اللغة. أما الأخطاء التي تنتهك القواعد التركيبية للغة يتم اكتشافها من خلال المحلل التركيبي. خلال التحليل الدلالي (أو تحليل المعاني) تظهر أخطاء التي تكون فيها البنية التركيبية لسلسة الأوامر سليمة و لكن ليس لها أي معنى دلالي في اللغة، مثال ذلك : إذا كان لدينا عملية جمع لمميزين الأول يعبر عن المصفوفة و الثاني هو اسم إجراء ...!!

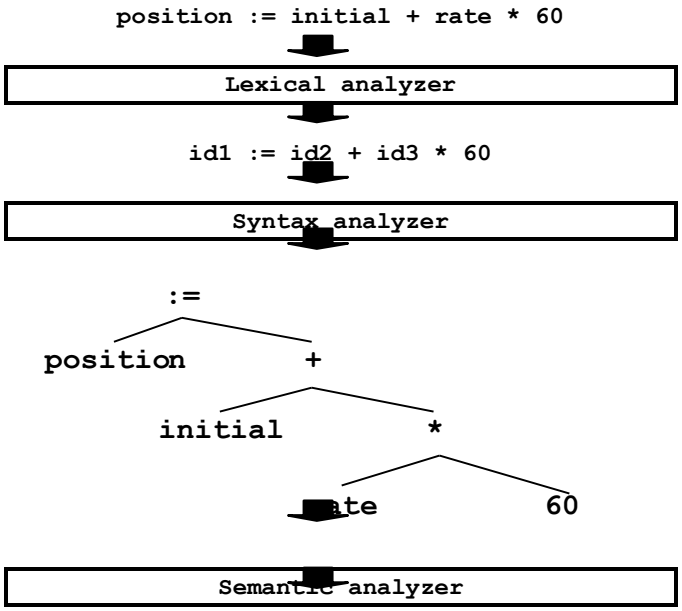
أطوار التحليل ***The Analysis Phases***

خلال تقدم عملية التصنيف فإن التمثيل الداخلي للبرنامج المُترجم يتغير باستمرار تبعاً لطور الترجمة. و فيما يلي مثال يستعرض أشكال التمثيلات الناجمة عن ترجمة التعبير الرياضي التالي :

```
position := initial + rate * 60
```

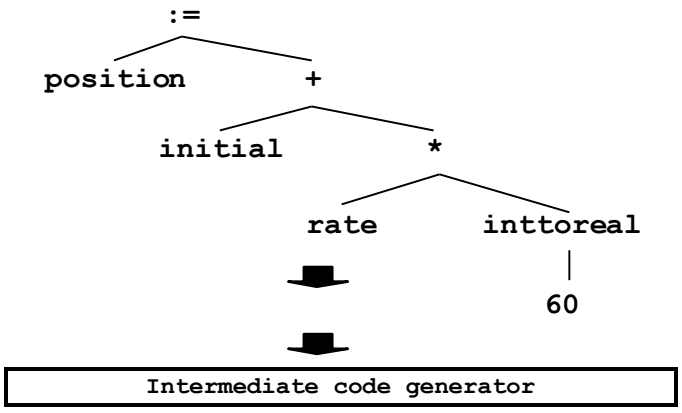
```
position := initial + rate *
60
```
(2.1)

الشكل 2–7 يبين التمثيل الناتج عن كل طور من أطوار التصنيف.



Symbol Table

1	position	…
2	initial	…
3	rate	…
4		




```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

Code optimizer

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

Code generator

```
MOVEF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

الشكل (2–7) مراحل ترجمة الأمر position := initial + rate * 60

يقرأ المحلل اللفظي سلسلة المحارف المكونة لبرنامج المصدر و يجمعها في مجموعات مكوناً بذلك الأجزاء النصية **(Tokens)** و يتعرف على طبيعة هذه العناصر النصية هل هي كلمة محجوزة أم مميز أم علامة ترقيم أو هل هي عملية مكونة من عدة محارف مثل عملية إسناد في الباسكال =:.

إن سلسلة المحارف المكونة للعنصر النصي تدعى اصطلاحاً **Lexeme** الخاص بالعنصر اللفظي **Token**.

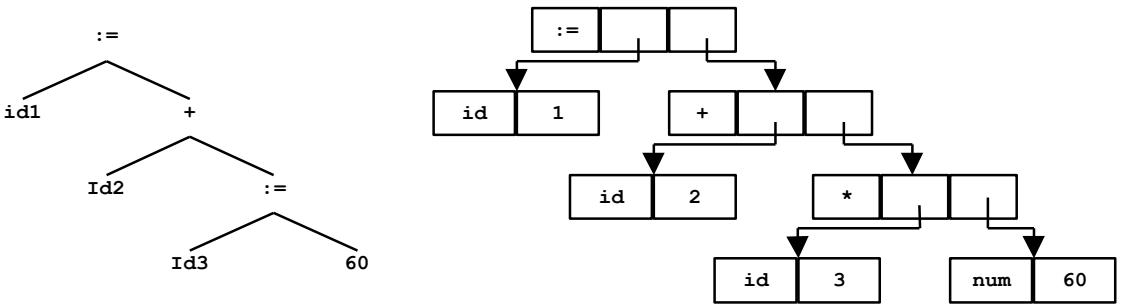
عندما يتعرف المحلل اللفظي على العنصر النصي يقوم بتوسيعه من خلال إرفاقه بما يسمى القيمة اللفظية **(Lexical value)** فمثلاً عندما يتعرف المحلل على المميز **rate** مثلاً فإنه لا يقتصر على توليد ما يدل على أن العنصر المقروء هو مميز **(id)** بل يقوم بإدخال هذا المميز في جدول الرموز في حال عدم وجوده فيه، و من ثم يرفق العنصر النصي هذا بمؤشر يدل عليه ضمن الجدول. في الفقرات التالية سيتم تمثيل المؤشر ضمن جدول الرموز بـ **id1**، **id2**، و **id3** وذلك من أجل كل من **position**، **initial**، و **rate** على التوالي، و ذلك من أجل التأكيد على أن التمثيل الداخلي يختلف عن سلسلة الرموز المكونة للمميز.

نتيجة لذلك فإن خرج المحلل اللفظي سيعبر عنه بالشكل التالي :

```
id1 := id2 + id3 * 60                                    (2.2)
```

طبعاً إن العناصر النصية الأخرى لن تبقى على حالها بل سيتم تمرير ما يعبر عنها من التمثيل الداخلي للمصنف.

لقد تم استعراض المحلل التركيبي و المحلل الدلالي في فقرات سابقة. تكون نتيجة التحليل التركيبي و التحليل الدلالي تمثيل شجري مكون من العناصر اللفظية، و يدعى هذا التمثيل بالشجرة التركيبية كما هو مبين في الشكل **2–8**.



الشكل (2–8) بنية المعطيات الداخلية الممثلة لشجرة الإعراب

نلاحظ أن كل عقدة من عقدها الداخلية عبارة عن سجل يتكون من حقل يحوي على قيمة تعبر عن عملية،

و حقلان آخران يحويان مؤشرين إلى المعاملين الأيمن و الأيسر لهذه العملية. أما أوراق الشجرة فهي عبارة عن سجل مكون من حقلين أو أكثر، الأول يميز طبيعة العنصر اللفظي و الثاني يشير إلى المعلومات عن هذا العنصر. بالطبع فإن التراكيب اللغوية الأخرى قد تتطلب حقول إضافية في السجلات المعبرة عن عقد شجرة التراكيب.

تولید الشیفرة الوسیطیة *Intermediate Code Generation*

بعد الانتهاء من التحلیل التركیبي و تحلیل المعاني، فإن بعض المصنفات تولد ما یدعى التمثیل الوسیطي لبرنامج المصدر. و یمكن أن نشبه التمثیل الوسیطي هذا بأنه برنامج من أجل آلة مجردة **(program for abstract machine)**. یجب أن یتمیز التمثیل الوسیطي بسهولة التولید و سهولة الترجمة إلى البرنامج الهدف **(target program)**. و ما هو جدیر بالذكر بأنه هناك العدید من أشكال التمثیل الوسیطي. من هذه التمثیلات ما یدعى بالتعلیمات ثلاثیة العنوان **(three-address code)**، و هي تشبه إلى حد ما لغة التجمیع حیث یكون فیها كل عنوان ذاكري (متحول) یتم التعامل معه كما لو كان مسجلاً. إن الشیفرة الثلاثیة العنوان تتألف من تسلسل من أوامر بسیطة، وكل أمر من هذه الأوامر یملك ثلاثة معاملات على الأكثر.

نلاحظ أن تمثیل التعبیر الریاضي **2.1** بالشیفرة الوسیطیة له الشكل التالي :

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

یتمیز التمثیل الوسیطي بعدة صفات :

- كل أمر ثلاثي العنوان یملك على الأكثر عملیة واحدة فقط بالإضافة إلى عملیة إسناد، و بهذا نجد أنه عند قیام المصنف بتولید هذه الشیفرة فإنه یجب أن یقرر تسلسل العملیات التي یجب أن تنفذ من أجل أن تكافئ برنامج المصدر.
- یجب أن یولد المصنف أسماء لمتحولات مؤقتة، و التي ستحوي القیم المرحلیة للعملیات الحسابیة المنفذة في كل أمر وسیطي من أجل استخدام القیمة المحسوبة كمعامل في المراحل التالیة من تقدم عملیة حساب التعابیر.
- بعض الأوامر ثلاثیة العنوان قد تحوي على أقل من ثلاثة معاملات كما هو الحال في الأمر الأول و أخیر في الأوامر **2.1** و هذا یعتمد بشكل أساسي على العملیة المطبقة في الأمر.

أَمثلة الشیفرة *Code Optimization*

إن أمثلة الشیفرة تهدف إلى تحسین الشیفرة الوسیطیة، حیث تكون نتیجة هذه العملیة شیفرة أسرع تنفیذاً و أصغر حجماً، مع الملاحظة أن بعض أنواع عملیات الأمثلة لیست ضروریة.

وكمثال عن عملیة أمثلة، نرى أن الخوارزمیة الطبیعیة الخاصة بتولید الشیفرة الوسیطیة تولد سلسلة الأوامر الموضحة في الفقرة السابقة و ذلك من أجل كل عملیة في الشجرة الممثلة للتعبیر الریاضي و الناتجة عن عملیة تحلیل المعاني.

و مع ذلك نجد أن نفس العمليات الحسابية يمكن أن تنفذ بصورة أكبر فعالية و ذلك من خلال عمليتين فقط كما هو موضح بالشكل التالي :

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```
(2.4)

و بهذا نجد أنه بأنه ليست هناك أية مشكلة في ضعف خوارزمية توليد الشيفرة الوسيطية طالما هناك طور للأمثلة يمكن من خلاله التغلب على هذا الضعف.

نلاحظ من المثال السابق بأن المؤمثل قام بحذف العملية **inttoreal** من خلال محاكاتها على الثابت الصحيح **60** و حساب قيمته الحقيقية **60.0** أثناء طور الأمثلة و لمرة واحدة و بهذا تم الاستغناء عن توليد الأمر الخاص بذلك.

بالإضافة لذلك، يمكن أن نلاحظ بأن المتحول المؤقت **temp3** مستخدم لمرة واحدة فقط و ذلك من أجل نقل القيمة المحتواة فيه إلى **id1**، و بذلك فلا توجد مشكلة في استبداله بـ **id1** في الأمر الثالث و حذف الأمر الأخير.

هناك اختلافات كبيرة في المهام التي تقوم به المؤمثلات الخاصة بمختلف المصنفات، و تدعى المصنفات التي تحوي طور الأمثلة بالمصنفات المؤمثلة **(optimizing compilers)**. مع الملاحظة أن الحصة الأكبر من زمن التصنيف يصرف في طور الأمثلة. يوجد الكثير من المصنفات التي تستخدم عمليات أمثلة بسيطة لا تسبب تباطؤ في عملية التصنيف، في المقابل تولد شيفرة جيدة إلى حد ما.

توليد شيفرة الهدف Code Generation

إن طور توليد الشيفرة الهدف هو الأخير في عملية التصنيف، و شيفرة الهدف عادة تكون عبارة عن شيفرة لغة الآلة ذات العنونة قابلة للتغيير **(relocatable machine code)**، أو يمكن أن تكون برنامج بلغة التجميع.

يتم في هذا الطور مجموعة من العمليات و هي :

- يتم حجز أماكن ذاكرة من أجل جميع المتحولات ضمن البرنامج.
- يتم ترجمة كل أمر من أوامر الشيفرة الوسيطية إلى سلسلة أوامر الآلة الهدف المكافئة.
- ومن العمليات الحاسمة في توليد شيفرة الهدف عملية إسناد المسجلات الخاصة بالمعالج الهدف إلى التحولات المؤقتة.

فيما يلي مثال يبين نتيجة الترجمة للأوامر الوسيطة المبينة في **2.4** إلى تعليمات الآلة مع الملاحظة أنه تم استخدام المسجلات **R1** و **R2** كأماكن مؤقتة للقيم المحسوبة.

```
MOVF id3, R2
```
(2.5)

```
MULF #60.0, R2
MOVF id2, R1
ADDF R2,R1
MOVF R1, id1
```

نلاحظ أن كل من المعامل الأول و الثاني في كل أمر يحددان المصدر و الهدف من أجل العملية المطبقة، و الحرف اللاحق F في كل أمر يبين لنا أن العملية المطبقة تتعامل مع أعداد بالفاصلة العائمة.

يقوم الأمر الأول المبين في 2.5 بنقل محتويات الذاكرة ذات العنوان id3 إلى المسجل الثاني R2، و من ثم يتم ضرب القيمة الحقيقية 60.0 بمحتويات المسجل R2 و وضع النتيجة في نفس المسجل. إن الرمز # يشير إلى أنه قيمة ستعالج كثابت عددي لا كعنوان ذاكرة. يقوم الأمر الثالث بنقل محتويات المتحول id2 إلى المسجل الأول R1 و من ثم يتم جمع القيم المخزنة في المسجلات R1 و R2 و وضع النتيجة في R1، و من ثم يتم حفظ قيمة المسجل R1 في المتحول id1.

الفصل الثالث

التحليل اللفظي
Lexical Analysis

مقدمة

إن أبسط طريقة لإنشاء المحلل اللفظي هو بناء مخطط يصف بنية العناصر النصية للغة المصدر و من ثم ترجمة هذه المخططات إلى برنامج بشكل يدوي حيث يقوم هذا البرنامج بالتعرف على هذه العناصر، يمكن استخدام هذه الطريقة بناء محللات لفظية فعالة إلى حد ما.

إن التقنيات المستخدمة في بناء المحللات اللفظية من الممكن تطبيقها في مجالات أخرى غير المترجمات مثل أنظمة استرجاع المعلومات. إن الفكر الأساسية للتحليل اللفظي هو إنشاء برنامج يقوم بتنفيذ مجموعة من الفعاليات يتم تفعيلها من خلال نماذج محددة من سلاسل المحارف **(String patterns)**. و عادة ما يتم بناء هذه البرامج أو أجزاءها بما يسمى باللغات الموجهة نحو النماذج أو لغات (نموذج – فعالية) **(pattern-action languages)**، و من هذه اللغات لغة **LEX** و التي تستخدم في بناء المحلل اللفظي للمترجمات. و يتم فيها وصف النماذج النصية من خلال ما يسمى بالتعابير النظامية **(Regular expressions)**، و من ثم و بترجمة هذه التعابير يتولد أوتومات محدد يقوم بالتعرف على النماذج النصية **(Finite-automate recognizer)**.

و من الأدوات المُستخدمة للتعابير النظامية في وصف سلاسل الرموز بعض أوامر النظام **MS-DOS** و التي تتقبل بارا مترات تصف أسماء الملفات مثل الأمر :

```
DEL *.TXT
```

و الذي يقوم بحذف جميع الملفات التي لها الامتداد **TXT** مهما كان اسمها.

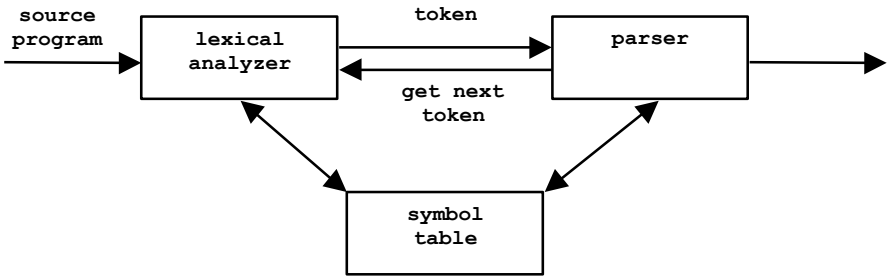
وظيفة المحلل اللفظي ***The Role Of The Lexical Analyzer***

إن المحلل اللفظي هو الطور الأول من المترجم. الوظيفة الأساسية له هو قراءة محارف الدخل و توليد خرج مكون من لسلسلة من العناصر النصية **(tokens)** و التي من السهل أن يتعامل معها المحلل التركيبي. إن هذا التفاعل ملخص في الشكل **3–1**.

عادة ما يتم تنظيم المحلل الفظي كروتين فرعي، يتم استدعاءه من قِبل المحلل التركيبي في كل مرة يريد فيها الحصول على عنصر النص جديد.

من الوظائف الإضافي التي يمكن أن يقوم بها المحلل اللفظي، التخلص من الملاحظات في برنامج المصدر،

و محارف الفراغات و فواصل الأسطر، و محارف الجدولة. كما و من الممكن أن يقوم بحساب الأسطر في البرنامج المصدر أثناء الترجمة لتحديد أماكن الأخطاء إن وجدت و ذلك لتضمين مكانها في رسائل الأخطاء.



الشكل (3–1) التفاعل بين المحلل الفظي و المعرب

في بعض الأحيان يتم تقسيم التحليل اللفظي إلى طورين متسلسلتين في العمل، الأول يدعى المسح **(scanning)** و الثاني هو التحليل اللفظي، حيث يقوم الماسح بالعمليات البسيطة على نص البرنامج، بينما المحلل الفظي يقوم بالعمليات المعقدة.

الرموز و النماذج و المفردات ***Tokens, Patterns And Lexemes***

في الحقيقة هناك عادة يكون هناك مجموعة السلاسل المحارف التي ينتج عنها ذات الرمز **(Token)**. إن مثل هذه المجموعة من السلاسل تكون مرتبطة مع الرمز من خلال ما يسمى ارتباط النماذج بالرموز. و المفردات **(lexemes)** هي سلاسل المحارف التي قد تعرف عليها المحلل اللفظي على أنها رموز **(tokens)**.

مثال

بفرض أنه لدينا أمر الباسكال التالي :

```
const pi = 3.1416;
```

إن السلسلة **pi** هي الـ **lexeme** الخاصة بالـ **token "identifier"**.

و الشكل **3–2** يبين أمثلة حول المفردات و النماذج و الرموز.

Token	*Sample lexemes*	*Information description of pattern*
const	const	Const
if	if	If
relation	<,<=,=,<>,>,>=	< or <= or = or <> or >= or >
id	pi,count,D2	Letter followed by letters and digits
num	3.1416,0,602E23	Any numeric constant

literal	"core dumped"	Any character between " and " except "

الشكل (2-3) أمثلة عن الـ Tokens

يتم التعامل مع الرموز (tokens) على أنها الرموز المحددة في قواعد اللغة و التي يتم استخلاصها من برنامج المصدر. و المفردات هي سلاسل المحارف التي يتم تجميعها ليتشكل منها نموذج الفظي (pattern).

إن النموذج (pattern) هو القاعدة التي تصف تركيب مجموعة المفردات و التي يمكن أن تمثل الرمز بشكل جزئي في برنامج المصدر.

صفات الرموز Attributes for Tokens

عندما يتعرف المحلل الفظي على عناصر النص، يقوم بجمع كافة المعلومات حول هذا العنصر النصي، و من هذه المعلومات سلسلة المحارف المكونة له (lexeme)، من أجل الأعداد الرقمية، يمكن أن يقوم باستخراج القيمة العددية الذي يمثله هذا العنصر النصي.

من الممكن أن تكون صفات الرموز عبارة عن مؤشر ضمن جدول الرموز و الذي من خلاله يمكن أن نحصل على كافة المعلومات حول ذلك الرمز.

الأخطاء اللفظية Lexeical Error

إن الأخطاء التي من الممكن أن تظهر في مرحلة التحليل اللفظي محدودة إلى حد كبير، و تنتج عادة من التشكيلات الخاطئة للأحرف و التي لا يمكن أن تشكل أي من نماذج المعروفة لديه. أما الأخطاء الأخرى فتترك للمراحل التالية من عملية الترجمة.

مثال

لو كان لدينا التعبير الشرطي بلغة الـ C المغلوط التالي:

```
fi (a==f(x)) …
```

نجد أن fi هي ليست الكلمة المفتاحية if و لكنها مميز صحيح، و بالتالي لا يوجد أي خطأ بالنسبة إلى المحلل اللفظي، و هنا يأتي دور المراحل التالية من عملية الترجمة في كشف الخطأ.

وصف رموز اللغة Specification of Tokens

إن التعابير النظامي هي تنويت هام لوصف النماذج النصية. و من خلالها يمكن وصف النماذج المختلفة لرموز اللغة.

السلاسل و اللغات ***String and Languages***

إن الأبجدية أو مجموعة المحارف تعبر عن مجموعة محددة من الحروف أو الرموز. إن كل من الـ ASCII و الـ EBCDIC هما مثالات لأبجدية الكمبيوتر. و السلسلة على أبجدية ما هي تتالي من رموز (أحرف) تلك الأبجدية.

يرمز للسلسلة بـ s و لطول السلسلة بـ |s| و الذي هو عدد الرموز أو أحرف الأبجدية المكونة للسلسلة s.

اللغة هي أي مجموعة من السلاسل فوق أبجدية ما.

إن اللغة المجردة المرمزة بـ∅ هي عبارة عن مجموعة فارغة و هي مجموعة تحوي سلسلة فارغة فقط.

إذا كان لدينا x و y سلسلتين يكون حاصل جمع x مع y و الذي يعبر عنه بـ xy هي السلسلة المكونة من ضم y إلى x.

إن السلسلة الفارغة حيادية بالنسبة لعملية جمع السلاسل $s\in = \in s = s$.

من الممكن التعبير عن ضم السلاسل إلى نفسها من خلال القوة :

$$S^0 = \in, S^1 = S, S^2 = SS, S^3 = SSS$$

تعريفات أخرى

- **بادئة السلسلة s (prefix of s) : و هي السلسلة الجزئية من s و التي من الممكن أن نحصل عليها بإزالة صفر رمز أو أكثر من نهاية السلسلة.**
- **لاحقة السلسلة s (suffix of s) : و هي السلسلة الجزئية من s و التي من الممكن أن نحصل عليها بإزالة صفر رمز أو أكثر من بداية السلسلة.**
- **السلسلة الجزئية من السلسلة s (substring of s) : و هي السلسلة الجزئية من s و التي من الممكن أن نحصل عليها صفر رمز أو أكثر من بداية السلسلة و صفر أو أكثر من بداية السلسلة s.**
- **البادئة المناسبة و اللاحقة المناسبة و السلسلة الجزئية المناسبة من s (proper prefix, suffix or substring of s) : و هي أي سلسلة غير فارغة و التي من الممكن أن تكون بادئة أو لاحقة أو سلسلة جزئية من s، بحيث $s \neq x$.**
- **التتالي الجزئي من s (subsequence of s) : و هي أي سلسلة تنتج عن إزالة صفر رمز أو أكثر من s، دون أن يكون الحذف استمراري.**

العمليات على اللغات *Operations on Languages*

هناك العديد من العمليات التي يمكن أن تطبق على اللغات. و من أجل موضوع التحليل اللفظي سوف نهتم بعملية الاتحاد، و الوصل أو الضم، و الإنغلاق و المعرفة كما يلي :

- الاتحاد بين L و M و يرمز بـ L∪M :

$$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ in } M \}$$

- وصل بين L و M و يرمز بـ LM :

$$LM = \{ st \mid s \text{ in } L \text{ and } t \text{ in } M \}$$

- الإنغلاق السلبي لـ L :

$$L^* = \cup_{i=0..\infty} L^i$$

- الإنغلاق الإيجابي لـ L :

$$L+ = \cup_{i=1..\infty} L^i$$

التعابير النظامية *Regular Expressions*

وهي عبارة عن تنويت يمكن من خلاله وصف العناصر النصية المكونة من سلاسل من المحارف و التي بدورها تكون أبجدية الحاسب.

و الآلية التي تعمل وفقها التعابير النظامي هي أنها تصف لغات فوق أبجدية ما، و من ثم تصف لغات أخرى معتمدة على اللغات المعرفة و ذلك من خلال العمليات تطبقها عليها.

و فيما يلي القوانين التي تعرف التعابير النظامية فوق أبجدية ما ∑، و كل قانون مرفق باللغة التي يعرفها.

1. ∈ هو تعبير نظامي يشير إلى الـ {∈}، و التي هي مجموعة تحوي على سلسلة فارغة.
2. في حال كانت a رمز من الأبجدية ∑، فإن a هو تعبير نظامي يشير إلى {a}. مع الملاحظة أن التعبير النظامي a هو ليس بالسلسلة المكونة من a أو الرمز a.
3. بفرض أن r و s هما تعبيران نظاميان يصفان لغتان L(r) و L(s) فإن :
   - (r)|(s) هي تعبير نظامي يشير إلى اللغة L(r)∪L(s).
   - (r)(s) هي تعبير نظامي يشير إلى اللغة L(r)L(s).
   - *(r) هي تعبير نظامي يشير إلى اللغة *(L(r)).
   - (r) هي تعبير نظامي يشير إلى اللغة L(r).

إن اللغة المشار إليها من خلال تعبير نظامي ما تدعى بمجموعة نظامية (regular set).

إن قولنا الرمز الأساسي يشير إلى ∈ أو إلى رمز من الأبجدية ∑.

إن المعامل * يملك الأولوية العليا في التعابير النظامية، تأتي بعده معامل الوصل (المذكور في البند الثاني)، و المعامل | له الأولوية الدنيا. فمثلاً ((a)|((b)*(c))) تكافئ a|b*c.

أمثلة

بفرض لدينا الأبجدية {a, b} = ∑ فإن :

- **التعبير a|b يشير إلى المجموعة {a, b}.**
- **التعبير (a|b)(a|b) يشير إلى المجموعة {aa, ab, ba, bb}. و أيضاً فإن aa|ab|ba|bb تشير إلى نفس المجموعة.**
- **التعبير a* يشير إلى المجموعة {∈,a,aa,aaa,aaaa,...}.**
- **التعبير (a|b)* يشير إلى مجموعة من جميع السلاسل ذات الطول الصفري أو أكثر و المكونة من a و b. و التعبير المكافئ أيضاً (a*b*)*.**
- **التعبير a|a*b يشير إلى مجموعة مكونة من السلسلة a و جميع السلاسل الكون من تكرار صفري أو أكثر لـ a متبوعة بـ b.**

في حال كان لدينا التعبيران r و s يشيران إلى نفس اللغة، فإننا نقول أن r و s متكافئتان، و يعبر عن ذلك بـ r = s، فمثلاً (a|b) = (b|a).

و فيما يلي جدول بالصفات الجبرية للتعابير النظامية :

حقائق	الوصف
r|s = s|r	| عملية تبديلية
r|(s|t) = (r|s)|t	| عملية تجميعية
(rs)t = r(st)	الوصل هو عملية تجميعية
r(s|t) = rs|rt (s|t)r = sr|tr	عملية الوصل توزيعية على |
∈r = r r∈ = r	هو عنصر حيادي بالنسبة لعملية الوصل∈
r* = (r|∈)*	* و ∈و هي العلاقة بين
r** = r*	* هي عملية مكررة لنفسها

التعريفات النظامية ***Regular Definition***

في بعض الأحيان نحتاج إلى إعطاء أسماء للتعابير النظامية، و من ثم استخدام هذه الأسماء من أجل تعريف تعابير نظامية أخرى. بفرض لدينا الأبجدية $\sum$، إن شكل التعريفات (التصريحات) النظامية تأخذ الشكل التالي:

$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
...
$d_n \rightarrow r_n$

حيث أن كل d_i هو اسم مميز، و التعبير النظامي r_i يكون مطبقاً فوق $\sum \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

مثال

فيما يلي التصريح عن المميز في لغة الباسكال و الذي يتكون بدوره من أرقام و حروف، و التي هي رموز جزئية من أبجدية الحاسب:

```
letter → A|B|…|Z|a|b|…|z
digit → 0|1|…|9
id → letter(letter|digit)*
```

مثال

فيما يلي التعريفات النظامية التي تصف التنويت الأعداد الحقيقية ذات الفاصلة العائمة في لغة الباسكال:

```
digit → 0|1|…|9
digits → digit digit*
optional_fraction → . digits | ∈
optional_exponent → (E(+|-|∈)digits)| ∈
num → digits optional_fraction optional_exponent
```

الاختصارات في التعابير النظامية

فيما يلي بعض الاختصارات المستخدمة و المعروفة في التعابير النظامية :

1. الوجود لمرة واحدة أو أكثر و يعبر عنه من خلال اللاحقة +. بفرض أن r تعبير نظامي يعبر عن اللغة L(r) فإن +(r) يشير إلى +(L(r))، و يكون لدينا r+ = rr*، و r* = r+|∈.
2. الوجود لمرة واحدة أو عدم وجود و يعبر عنه باستخدام ? كلاحقة، حيث يشير ?(r) إلى اللغة ?(L(r)) و هي L(r)∪{∈}.
3. أصناف المحارف، في المحارف التي تتبع لترتيب معروف من الممكن أن نستخدم مفهوم مجال من المحارف، فمثلاً [a-z] يكافئ التعبير a|b|…|c. و التعبير [A-Za-z] يكافئ A|B|…|Z|a|b|…|z.

تمييز العناصر النصية ***Recognition of Tokens***

في الفقرة السابقة قد تم ذكر الطريقة التي يمكن وصف العناصر النصية، أما هنا فسيتم استعراض الطريقة التي سيتم التعرف عليها في برنامج المصدر.

تكون العناصر النصية في برنامج المصدر عادة موجودة بشكل متسلسل، و مفصول فيما بينها بفراغات أو وجودها خلف بعضها البعض، و لكون خصائصها تميزها عن بعضها البعض فمن الممكن تمييزها و استخلاصها من نص البرنامج. عند استعراض آلية التعرف عليها سوف يتم ذكرها من أجل كل عنصر أو مجموعة معينة من عناصر النص و كأنها مفصولة عن بقية العناصر النصية، و بعدها سيتم تجميع هذه الآلية الخاصة بكل مجموعة جزئية من أجل التعرف على العناصر الموجودة في البرنامج بشكل عام.

مخططات الانتقال ***Transition Diagrams***

إن مخططات الانتقال تصف بشكل عام الفعاليات التي يقوم بها المحلل اللفظي عندما يطلب منه عنصر نص تالي ليجلبه من نص البرنامج.

في مخطط الانتقال، الدوائر تشير إلى الحالات **(states)** التي يمكن أن يكون فيها المحلل اللفظي أثناء عمله، و هذه الحالات موصولة فيما بينها بأقواس تدعى الحدود **(edges)** و تعبر عن مسار تغير الحالات، و كل حد أو قوس يغادر الحالة يملك لافتة تشير إلى الحرف التالي الذي يمكن أن يظهر في الدخل التالي بعد الوصول إلى هذه الحالة. و اللافتة **other** تشير إلى جميع الرموز التي لم تذكر في اللوافت الخاصة بالحدود المغادرة الأخرى لتلك الحالة.

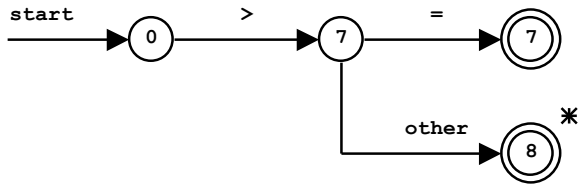
يكون مخطط الانتقال محدداً، في حال عدم وجود رموز مكرر من أجل جميع اللوافت المغادرة للحالة الواحدة، أي أننا نستطيع أن نحدد تماماً الحالة التي سيتم الانتقال إليها من أجل رمز دخل ما. كما و لا يستخدم الرمز ∈ كلافتة لحد ما، أي أنه لا يتم الانتقال من حالة إلى أخرى دون أن يكون هناك رمز قد تم قراءته.

دوماً هناك حالة بدائية و يشار إليها بـ **start**، و هي الحالة التي يكون فيها المحلل عند بداية عملية تمييز العنصر النصي.

هناك حالات يقوم فيها المحلل بتنفيذ فعالية ما و ذلك عندما يكون فيها و يقرأ رمز الدخل التالي، و يشار إليها بدائرة مزدوجة، و عادة ما تدعى مثل هذه الحالة بالنهائية، حيث يكون المحلل قد ميز عنصر نصي ما.

مثال

فيما يلي الشكل **3-3** يبين مخطط انتقال الخاص بتمييز النموذجان النصيان< و =<.

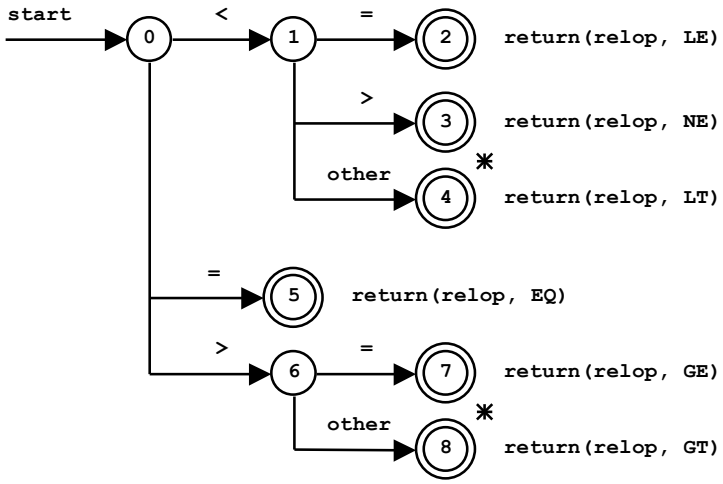


الشكل (3–3) مخطط الانتقال الخاص بـ =< و <

تشير النجمة بأن الحرف المقروء في آخر مرة (الحرف الذي سبب الانتقال إلى هذه الحالة) لا يشكل جزئ من النموذج الذي تعرف عليه في هذه الحالة. و بالتالي يجب أن يعاد الرمز الذي تم قراءته إلى الدخل (الرجوع إلى الخلف بمقدار رمز واحد) من أجل تمييز العنصر النصي التالي.

مثال

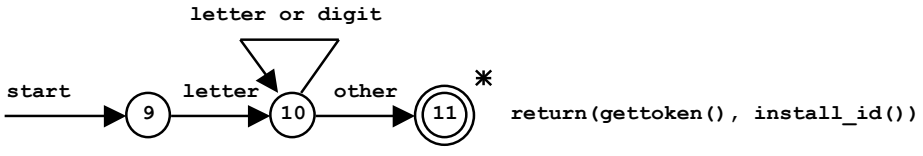
فيما يلي مخطط الانتقال الخاص بجميع النماذج النصية و التي تعبر عن العلاقات في المتراجحات الخاصة بلغة الباسكال، وهي مبينة في الشكل 3–4.



الشكل (3–4) مخطط الانتقال الخاص بالعلاقات

مثال

فيما يلي الشكل 3–5 يبين مخطط الانتقال الخاص بالمميز في لغة الباسكال.



الشكل (3–5) مخطط الانتقال الخاص بالمميز (identifier)

الأوتومات المحدود Finite Automata

المميز (recognizer) الخاص بلغة ما هو برنامج الذي يأخذ سلسلة الدخل s و يجيبنا بنعم (إذا كانت السلسلة تشكل جملة صحيحة لغوياً) أو لا.

يتم ترجمة التعابير النظامية إلى مخططات الانتقال عامة و هذه المخططات تدعى بالأوتومات المحدود. و الأوتومات يمكن أن يكون محدداً (deterministic) أو غير محدد (nondeterministic)، و الأوتومات غير المحدد هو ذاك الذي من الممكن يكون فيه أكثر من انتقال من حالة ما فيه من أجل رمز دخل واحد. و من خلال الأوتومات المحدد أو غير المحدد يكننا أن نعبر عن الآلية التي يتم من خلالها تمييز مجموعة نظامية ما، تلك المجموعة التي يمكن وصفها من خلال التعابير النظامية.

إن الأوتومات المحدد عادة يكون أكبر بكثير من الأوتومات غير المحدد المكافئ.

الأوتومات المحدود خير المحدد Nondeterministic Finite Automata

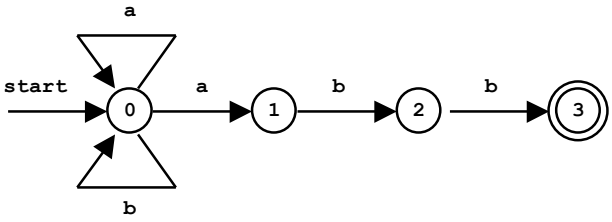
و يرمز يشار عليه عادة بـ NFA، و النموذج الرياضي الخاص به يتألف من :

1. مجموعة من الحالات S.
2. مجموعة رموز الدخل Σ، أو أبجدية الدخل.
3. تابع الانتقال move، و الذي يأخذ بارامتراته هو ثنائية حالة–رمز (state-symbol) و قيمته مجموعة حالات.
4. الحالة البدائية له s_0.
5. مجموعة الحالات النهائية F.

و يتم وصف الـ NFA من خلال مخطط الانتقال. و هو مخطط موجه، العقد هي الحالات، و الحدود المسماة (labled edges) تعبر عن تابع الانتقال.

مثال

فيما يلي أوتومات محدود غير محدد NFA، و الذي يميز اللغة (a|b)*abb، وهو مبين في الشكل 3–6



الشكل (3–6) أوتومات محدود غير محدد NFA

إن حالات الأوتومات هي {0,1,2,3}، أبجدية الدخل {a,b}، حالة البداية 0 state، مجموعة الحالات النهائية {3}.

في الحالة العادية نستعمل مخطط الانتقال لوصف تابع الانتقال، أما في الحاسب فيستخدم عادة جدول الانتقال، و هو مبين في الشكل 3–7.

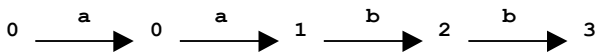
State	Input symbols	
	A	b
0	{0,1}	{0}
1	-	{2}
2	-	{3}

الشكل (3–7) جدول الانتقال للأوتومات المحدود

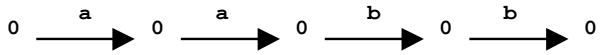
إن الـ NFA يقبل سلسلة الدخل x، فقط عندما وفقط وجد مسار من الحالة البدائية إلى إحدى الحالات النهائية فيه بحيث تكون الأحرف الدخل للانتقالات تهجئ السلسلة x.

مثال

إن الأوتومات السابق يقبل سلاسل الدخل abb، aabb، babb، aaabb. بأخذ السلسلة aabb، نجد أن الانتقالات التي يأخذها الأوتومات هي:



في الحقيقة فإن هذا الأوتومات من الممكن أن يأخذ أكثر من مسار من أجل نفس سلسلة الدخل فمثلاً يمكن أن يكون المسار له الشكل التالي:



الأوتومات المحدود المحدد *Deterministic Finite Automata*

و يشار إليه عادة بـ DFA، و يتميز بما يلي:

1. **لا توجد فيه انتقالات من أجل ∈.**
2. **من أجل أي حالة s و رمز دخل ما a، لا يوجد أكثر من انتقال واحد يغادر به هذه الحالة من أجل هذا الرمز.**

بكلام آخر يمكننا القول بأنه في الأوتومات المحدد ينفذ انتقال واحد فقط من حالة ما إلى أخرى من أجل رمز دخل ما غير صفري، و هذا الانتقال هو محدد، ويأخذ مسار ثابت و محدد من أجل سلسلة دخل ما.

و فيما يلي خوارزمية محاكاة (التنفيذ البرمجي) الـ DFA:

Algorithm. Simulating a DFA

Input. An input string x terminated by an end-of-file character *eof*. A DFA D with start state s_0 and set of accepting states F.

Output. The answer "yes" if D accept x, "no" otherwise.

Method. Apply the algorithm in next figure to the input string x. The function move(s,c) gives the state to which there is a transition from state s on input character c. The function next character of the input string x.

```
s := s0;
c := nextchar;
while c ≠ eof do
      s := move(s,c);
      c := nextchar
end;
if s is in F then
      return "yes"
else return "no";
```

تحويل الأوتومات غير المحدد إلى محدد *Conversion of an NFA into a DFA*

إن الفكرة الأساسية و الكامنة في تحويل NFA إلى DFA هو أن كل حالة من حالات الـ DFA توافق مجموعة من حالات الـ NFA. حيث تكون كل حالة من حالات الأوتومات المحدد حاوية على جميع الحالات التي يمكن أن ينتقل إليها الأوتومات غير المحدد من أجل سلسلة دخل ما. أي أنه و من أجل سلسلة دخل و المكونة من الرموز $a_1a_2...a_n$، فإن الـ DFA سوف ينتقل إلى حالة ما فيه، و هذه الحالة هي مجموعة من جميع الحالات التي يمكن أن يصل إليها الـ NFA من أجل هذه السلسلة.

و فيما يلي خوارزمية التحويل :

Algorithm. Construction a DFA from an NFA.

Input. An NFA N.

Output. A DFA D accepting the same language.

Method. This algorithm constructs a transition table Dtran for D. Each DFA state is a set of NFA states and we construct Dtran so that D will simulate "in parallel" all possible moves N can make on given input string.
The operations in next figure will be used to keep track of sets of NFA states (s represent an NFA state and T a set of NFA states).

Operation	*Description*
∈-closure(s)	Set of NFA states reachable from NFA state s on ∈-transitions alone.
∈-closure(T)	Set of NFA states reachable from some NFA state s in T on ∈-transitions alone.
move(T,a)	Set of NFA states to which there is a transition on input symbol a from some NFA state s in T.

Before it sees the first input symbol , N can be in any of the set

∈-closure(s_0), where s_0 is the start on N. Suppose that exactly

the states in set T are reachable from s_0 on given sequence of input symbols, and let a be the next input symbol. On seeing a, N can move to any of the states in the set move(T,a). When we allow for

∈-transitions, N can be in any of the states in ∈-closure(move(T,a)), after seeing the a.

```
initially, ∈-closure(s0) is the only state in Dstates and is is
unmarked.
while there is an unmarked state T  in Dstates do begin
      mark T;
      for each input symbol a do begin
            U := ∈-closure(move(T,a));
            if U is not in Dstates then
                  add U as an unmarked state to Dstates;
            Dtran[T,a] := U
      end
end
```

1. The start state of D is ∈-closure(s0).
2. A state of D is an accepting state if it is a set of NFA states containing at least one accepting state of N.

Computation of ∈-closure.

```
push all states in T onto stack;
initialize ∈-closure(T) to T;
```

```
while stack is not empty do begin
      pop t, the top element, off of stack;
      for each state u with an edge from t to u labeled ∈ do
            if u is not in ∈-closure(T) do begin
                  add u to ∈-closure(T);
                  push u onto stack
            end
end
```

Construction of an NFA from *بناء الأوتومات غير المحدد اعتباراً من التعابير النظامية* *Regular Expressions*

تعتمد عملية بناء الـ NFA اعتباراً من التعابير النظامية على تفكيك التعابير النظامية إلى أخرى جزئية أبسط، و ذلك حتى نصل إلى أدنى مستوى، ومن ثم يتم تكوين مخططات انتقال خاصة بالتعابير النظامية الجزئية هذه، و بتجميع هذه المخططات في مخططات أكبر فأكبر للحصول على مخططات NFA تقبل اللغة الموصوفة بالتعابير النظامية.

و فيما يلي الخوارزمية الخاصة بهذه العملية :

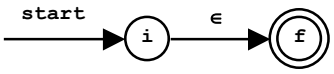
Algorithm. Construction NFA from regular expression.

Input. A regular expression r over an alphabet Σ.

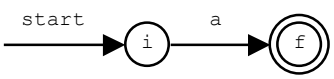
Output. An NFA N accepting L(r)

Method. We first parse r into constituents subexpressions. Then, using rule (1) and (2) below, we construct NFA's for each basic symbols in r. The basic symbols correspond to parts (1) and (2) in the definition of a regular expression. It is important to understand that is a symbol a occurs several times in r, a separate NFA is constructed for each occurrence.
Then, guided by the syntactic structure of the regular expression r, we combine these NFA's inductively using rule (3) below until we obtain the NFA for the entire expression. Each intermediate NFA produced during the course of the construction corresponds to subexpression or r and has several important proprieties: it has exactly one final state, no edge enters the start state, and no edge leaves the final state.

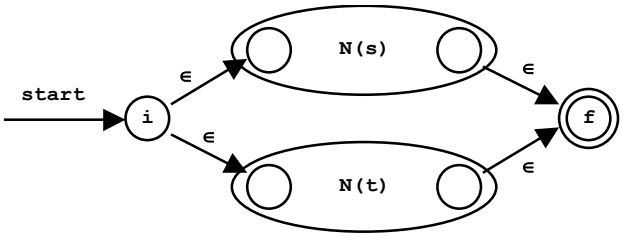
1. For ∈, construct the NFA



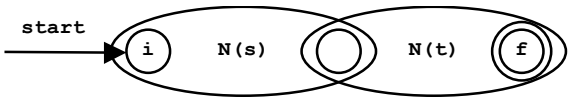
2. For a in Σ, construct the NFA



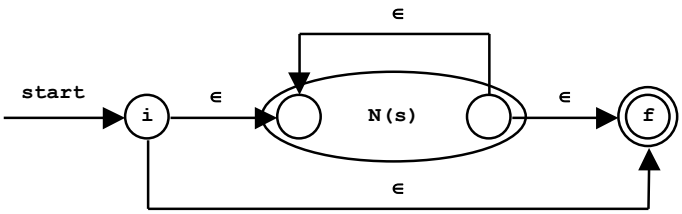
3. Suppose N(s) and (N(t) are NFA'a for regular expressions s and t
a) For regular expression s|t, construct the following composite NFA N(s|t):



b) For regular expression st, constuct the composite NFA N(st):



c) For regular expression s*, construct the composite NFA N(s*):

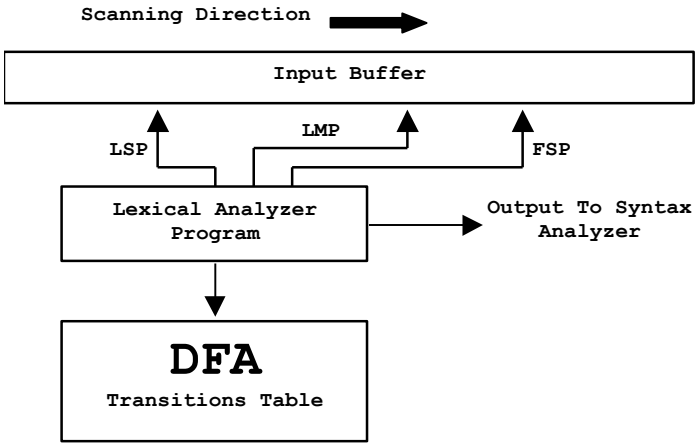


d) For the parenthesized regular expression (s), use N(s) itself as the NFA.

النموذج البرمجي للمحلل اللفظي

يتكون نموذج المحلل اللفظي هذا من دارئ الإدخال، و الذي بدوره يحوي لسلسلة الدخل، و البرنامج الذي ينفذ عملية التحليل اللفظي، و جدول الانتقالات الخاصة بالأوتومات المحدد الذي يقود المحلل التركيبي.

إن جدول الانتقالات هو مصفوفة ثنائية البعد، الأسطر توافق حالة الأوتومات، و الأعمدة تقابل رموز الدخل، و يتم تحديد الحالة 's التالية من أجل كل رمز يتم إدخاله و ليكن a، وذلك عندما يكون في الحالة s من خلال قراءة محتوى المركبة DFATable[s,a]، أو بكلام آخر DFATable[s,a] = 's. مع الملاحظة أن قيمة الحالة الجديدة من الممكن أن تكون ERROR، كما و يرافق الجدول الانتقال، جدول آخر ببعد وحيد، خلاياه تقابل حالات الأوتومات، حيث قيم مركباته تشير إلى العناصر النصية التي تقابل كل حالة نهائية للأوتومات، حيث يمكن أن تكون قيمة الخلية إما nul، و ترمز إلى أن الحالة الموافق هي حالة غير نهائية في الأوتومات، أو قيمة تشير إلى أن الحالة هي حالة نهائية، و القيمة ذاتها ترمز إلى العنصر النصي الموافق لتلك الحالة.



الشكل (3–9) النموذج المحلل اللفظي الذي يستخدم جدول الانتقالات DFA

إن هذا النموذج، يعمل وفق مبدأ استخلاص أطول عنصر نصي ممكن في كل مرة يطلب منه ذلك، و يتحقق ذلك من خلال الآلية التالية :

يملك المحلل ثلاث مؤشرات ضمن سلسلة الدخل، الأول يشير إلى الحرف الأول للعنصر الذي يتم تحليله الآن (lexeme start pointer LSP)، و الثاني يشير إلى نهاية آخر العنصر نصي محقق (last matched lexeme pointer LMP)، و المؤشر الثالث يشير إلى الحرف الذي يتم معالجته الآن (forward scan pointer FSP).

في البداية يقوم البرنامج بوضع نفسه في الحالة البدائية s := start state، و يجعل المؤشرات الثلاثة تشير إلى بداية السلسلة المراد استخلاص العناصر النصية منها. ثم يبدأ بتحريك المؤشر FSP، إلى الحرف التالي، و في كل مرة يتم تجاوز محرف دخل a يقوم بتعديل متحول حالة الأوتومات إلى القيمة الجديدة s := DFATable[s,a]، و يقوم بفحص الحالة الجديدة هل هي حالة نهائية أم لا، في كل مرة لا تكون فيها الحالة نهائية، يقوم بتحريك المؤشر FSP إلى الأمام، في حال صادف حالة نهائية يقوم بضبط المؤشر LMP، إلى مكان المؤشر FSP، و يقوم بالاحتفاظ بقيمة الحالة النهائية الموافقة لموقع المؤشر LMP، و من ثم يتابع في تحريك FSP، و تغيير الحالة s إلى أن يصل إلى حالة يكون فيها DFATable[s,a] = ERROR، يقوم بالعودة المؤشر LMP، و يفحصه، فإذا كان LMP = LSP (أي لم يتم المرور بأية حالة نهائية خلال تقدم المؤشر FSP انطلاقاً من LSP) و هذا يعني أن الدخل خاطئ، و يتوقف المحلل اللفظي عن العمل، معلناً عن وجود خطأ في الدخل. و في حال كان LMP ≠ LSP، فهذا يعني يقوم المحلل باستخلاص السلسلة الموجودة بين LSP و LMP و يتم تحديد نوع العنصر الذي تمثله السلسلة من خلال قيمة الحالة الموافقة للموقع LMP، من ثم يهيئ المحلل اللفظي نفسه من أجل البحث عن العنصر النصي التالي، حيث يجعل LSP := LMP، و FSP := LMP، و s := start state، و يبدأ بتحريك FSPمن جديد ...

الفصل الرابع

التحليل التركيبي

Syntax Analysis

مقدمة

إن كل لغة برمجة تمتلك قوانين خاصة بها تصف البنية التركيبية للبرامج الصحيحة بهذه اللغة. ففي الباسكال مثلاً، البرنامج يتكون من كتل **blocks** و الكتلة تتكون من أوامر، و الأمر يتكون من التعابير، و التعبير يتكون من عناصر لفظية **tokens** و هكذا ...

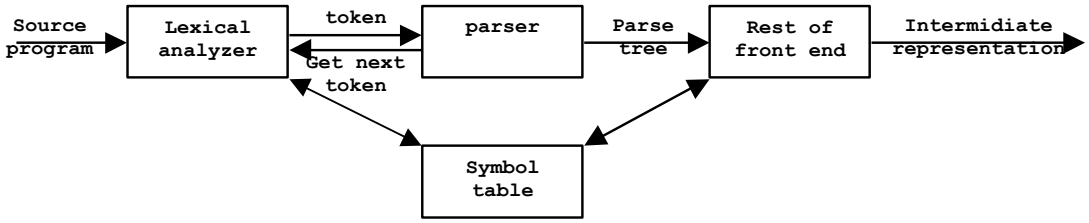
إن البنية التركيبية للغة برمجة ما يمكن وصفها من خلال قواعد خالية من السياق **Context free grammar** أو باستخدام التعابير **BNF**.

إن وصف اللغة من خلال القواعد تقدم مزايا كبيرة لكل من مصممي اللغة و مصممي المترجمات، منها:

- إن القواعد تعطي فهم سهل و دقيق للمواصفات التركيبية للغة برمجة ما.
- من أجل بعض الأصناف من القواعد من الممكن بناء محلل تركيبي (معرب) **Parser or Syntax Analyzer** فعال و الذي بدوره يمكن استخدامه في تحديد فيما إذا كان برنامج مصدر ما قد تم بناءه تركيبياً بشكل سليم تبعاً للغة ما تصفها هذه القواعد. و من المزايا الأخرى لعملية إنشاء المعرب هو أنه و من خلال عملية إنشاء المعرب يمكن كشف الغموضات و الصعوبات في عملية الإعراب أو التحليل التركيبي من أجل قواعد قد تم وضعها لهذه اللغة، و هذه المشاكل لا تظهر في الحالة العادية في المراحل الأولية من تصميم اللغة.
- القواعد التي تم تصميمها بشكل جيد تقدم بنية تركيبية للغة البرمجة و التي تكون مفيدة لعملية الترجمة لبرنامج المصدر إلى شيفرة هدف صحيحة و كذلك لكشف الأخطاء في البرنامج المُترجم. و هناك أدوات متاحة من أجل تحويل الترجمة الموصوفة من خلال القواعد اللغوية إلى برامج تعمل كمترجمات.
- إن اللغة تتغير عبر الزمن، و ذلك من خلال إضافة تراكيب جديدة إليها أو إضافة مهام جديدة لمترجماتها، و هذه الإضافات يمكن تحقيقها بسهولة أكبر عند وجود أدوات خاصة قد تم إنشاءها لتعمل معتمدة على أساس وصف اللغة باستخدام القواعد.

الدور الذي يلعبه المعرب (المحلل التركيبي)

في نموذج المترجم الذي سيتم الاعتماد عليه، المعرب أو المحلل التركيبي يقوم باستقبال سلسلة من العناصر النصية **tokens** من المحلل اللفظي كما هو مبين في الشكل **4–1** و يقوم بالاختبار فيما إذا كان بالإمكان توليد هذه السلسلة اعتماداً على قواعد اللغة الخاصة ببرنامج المصدر. كما و نفترض أن المعرب يقو بالتقرير عن الأخطاء بشكل مفهوم. كما أنه من الواجب أن يتم اكتشاف الأخطاء الشائعة في البرنامج المصدر و متابعة عملية التحليل لباقي سلسلة الدخل.



الشكل (4–1) موقع المحلل التركيبي في نموذج المترجم

هناك ثلاثة أنواع عامة للمحللات التركيبية المعتمدة على أساس القواعد. إن الطرق العامة في عملية الإعراب مثل خوارزمية **Cocke-Younger-Kasami** و الخوارزميات البدائية تستطيع إجراء التحليل التركيبي (الإعراب) من أجل أي نوع من القواعد. و لكن هذه الطرق ذات كفاءة متدنية عند إنشاء مترجمات بالاعتماد عليها.

الطرق الشائعة و المستخدمة في إنشاء المترجمات تصنف إلى نوعين هما التحليل من الأعلى إلى الأسفل **top-down** أو من الأسفل إلى الأعلى **bottom-up**. و كما هو ظاهر من تسمية هذه الطرق فإن المعرب من الأعلى إلى الأسفل يقوم بإنشاء شجرة الترجمة ابتداءً من الأعلى (الجذر) و باتجاه الأسفل (الأوراق). بينما المعربات التي تعتمد مبدأ من الأسفل إلى الأعلى فإنها تبدأ بتكوين الأوراق أولاً و من ثم الأفرع العليا حتى تصل إلى الجذر. و في كلا النوعين فإن الدخل يقرأ باتجاه واحد من اليسار إلى اليمين رمز تلو الآخر.

إن الطرق الأكثر فاعلية في كلا النوعين تعمل على مجموعة جزئية من القواعد اللغوية، و لكن معظم هذه مجموعات الجزئية و التي منها القواعد **LL** و **LR** قادرة على وصف و شمول البنية التركيبية لمعظم لغات البرمجة.

المعربات التي يتم بناءها بشكل يدوي تتبع عادة إلى الصنف **LL**. بينما المعربات التي تشمل الأصناف أوسع من قواعد **RL** عادة يتم إنشاءها باستخدام أدوات برمجية خاصة بذلك.

معالجة الأخطاء التركيبية ***Syntax error handling***

في حال كان المترجم يقوم بمعالجة البرامج الصحيحة فقط، فإن تصميمه سيكون بسيطاً إلى حد كبير. لكن

المبرمجين يقومون بارتكاب الأخطاء بشكل متكرر و بالتالي فإن المترجم يجب أن يساعدهم في تحديد الأخطاء و أماكنها. ما يثير الدهشة هو أن الأخطاء هو شيء مألوف جداً، و هناك العديد من لغات البرمجة تم وضعها مع الأخذ بعين الاعتبار أمر معالجة الأخطاء.

إن معظم الخواص الواصفة للغات البرمجة لا تصف الطريقة التي سيتم الاستجابة بها إلى الأخطاء، و بالتالي فإن الطريقة المستخدمة يتم تركها على عاتق مصمم المترجم لهذه اللغة. كما أن التخطيط المبكر لطريقة معالجة الأخطاء يؤدي إلى ميزتين هما بساطة بنية المترجم و تقدم تحسينات في الاستجابة للأخطاء.

كما نعلم أن البرامج يمكن أن تحتوي على أخطاء تنتمي إلى مستويات مختلفة، فمثلاً يمكن أن تكون:

- أخطاء لفظية، كخطأ في تهجئة المميزات (الأسماء) و الكلمات المفتاحية أو العمليات (العوامل).
- الأخطاء التركيبية، مثل الأخطاء في التعابير الرياضية ذات الأقواس غير المتوازنة.
- أخطاء المعاني، كما هو الأمر تطبيق عمليات على أنواع غير متوافقة من المعاملات فيما بينها.
- الأخطاء المنطقية، مثل الاستدعاء الذاتي اللانهائي.

عادة يتم التركيز في التقصي عن الأخطاء و تصحيحها في مرحلة التحليل التركيبي، و من أحد أسباب هذا الأمر هو أن معظم الأخطاء تنجم عن سلسلة الدخل من العناصر النصية التي فيها عدم التزام بقوانين اللغة. و السبب الآخر هو وجود طرق تحليل تركيبي متطورة تستطيع و بفاعلية كبيرة اكتشاف الأخطاء التركيبية. إن اكتشاف الأخطاء المنطقية أثناء زمن الترجمة هو عمل معقد جداً. في هذا الفصل سيتم استعراض بعض التقنيات الأساسية في تصحيح الأخطاء التركيبية. و سيتم استعراض هذه الطرق بشكل متلازم مع أساليب عملية الإعراب. إن معالج الأخطاء في المترجم يجب أن يحقق الأهداف التالية :

- يجب أن يقرر عن الأخطاء بشكل واضح و دقيق.
- يجب أن يقرر عن الأخطاء بشكل سريع كفاية ليستطيع استعراض الأخطاء اللاحقة.
- يجب ألا يبطئ عملية ترجمة البرامج السليمة.

إن تحقيق هذه الأهداف لهو تحدٍ كبير، و لحسن الحظ إن الأخطاء الشائعة هي بسيطة و بالتالي فإن آلية الكشف البسيطة تكون كافية عادة، و مع ذلك يمكن أن تكون الأخطاء قد حصلت قبل المكان الذي تم الكشف فيه عن الخطأ بوقت طويل، و هذا ما يجعل عملية التخلص من الخطأ صعبة. في بعض الحالات الصعبة فإن معالج الأخطاء يجب أن يقوم بتخمين ما كان يفكر به المبرمج عند كتابة البرنامج.

العديد من طرق الإعراب مثل **LL** و **LR** تكتشف الأخطاء بشكل مبكر قدر الإمكان عند وروده. فهي تمتلك ميزة البادئة الممكنة و تدعى **viable-prefix property**، و يعني هذا أنها تكشف الخطأ عند ورود بادئة لسلسلة دخل غير صحيحة لا تتبع لا أي من الاحتمالات الصحيحة لها في مكان ما من البرنامج.

مثال

من أجل الوصول إلى الإدراك عن مختلف أنواع الأخطاء التي قد تحصل عملياً، سوف نستعرض بعض الأخطاء التي وجدها كل من **Druseikis** و **Ripley** في عام **1978** في برامج معدة من قبل الطلاب.

لقد اكتشفوا أن الأخطاء لا تحدث بشكل متكرر إلى حد كبير، فقد وجد أن **60%** من البرامج التي تم ترجمتها صحيحة قواعدياً و معنوياً. هذا و حتى إن وجدت أخطاء في البرامج فهي متفرقة، فقد وجدوا أن **80%** من الأوامر تحوي فقط خطأ وحيد، بينما **13%** تحوي خطأين فقط هذا و معظم الأخطاء هي بسيطة للغاية حيث أن **90%** من الأخطاء عبارة عن أخطاء في كتابة العناصر النصية.

كما و من الممكن تصنيف الأخطاء و بشكل بسيط :

- **60%** من الأخطاء هي أخطاء في إشارات الترقيم.
- **20%** من الأخطاء هي أخطاء في العمليات و المعاملات.
- **15%** من الأخطاء في كتابة كلمات المفتاحية اللغة.
- باقي الأخطاء هي أخطاء في استخدام الفاصلة المنقوطة.

و كمثال على ذلك لنأخذ البرنامج التالي :

```
(1)       program prmax(input, output);
(2)       var
(3)       x,y: integer;
(4)       function max(i: integer; j: integer): integer;
(5)       { return maximum of integers i and j }
(6)       begin
(7)       if i > j then max := I
(8)       else max := j
(9)       end;
(10)    begin
(11)    readln(x,y);
(12)    writeln(max(x,y));
(13)    end.
```

الشكل (**2–4**)

إن أشهر الأخطاء في علامات الترقيم هو استخدام الفاصلة مكان الفاصلة المنقوطة في قائمة البارامترات في قسم التصريح الخاص بالتابع (مثل ورود الفاصلة العادية مكان الفاصلة المنقوطة في السطر **4**). و من الأخطاء الأخرى عدم كتابة الفاصلة في الأماكن الإجبارية لها (مثل عدم وجود الفاصلة المنقوطة في نهاية السطر **4**). و أيضاً ورود فاصلة منقوطة زائدة في أماكن يجب ألا ترد فيها (مثل ورودها قبل **else** في بداية السطر **8** أو نهاية السطر **7**).

من الممكن كون شيوع الأخطاء المتعلقة بالفاصلة المنقوطة يعود إلى اختلاف مفهوم استخدامها من لغة إلى أخرى. ففي الباسكال، الفاصلة المنقوطة هي الفاصل بين الأوامر. أما في لغة **PL/1** و لغة الـ **C** تستخدم لإنهاء الأوامر. بعض الدراسات بينت أن الاستخدام الثاني لها ينتج عنها أخطاء أقل.

من الأخطاء الشائعة المتعلقة في كتابة العوامل هي عدم كتابة النقطتان في عملية النسب **=:**.

التهجئة الخاطئة للكلمات المفتاحية من الأخطاء النادرة نسبياً، و مثال عن مثل هذا الخطأ نسيان الحرف **i** في الكلمة **Writeln**.

العديد من المترجمات الخاصة بالباسكال ليس فيها أية صعوبة في معالجة الأخطاء الناجمة عن عمليات إدخال و الإلغاء و التحويل. فهناك العديد من مصنفات الباسكال تستطيع ترجمة البرامج التي قد تحوي الأخطاء الشائعة في علامات الترقيم و المعاملات، و هي تقوم فقط بالتقرير عن الأخطاء المكتشفة مع الإشارة إلى موقعها بشكل دقيق.

هناك أخطاء من الصعب معالجتها بشكل صحيح، مثل الأخطاء المتعلقة بـ **begin** و **end** (مثال ذلك نسيانها في السطر **9**). فمعظم المصنفات لا تحاول إصلاح مثل هذه الأخطاء.

ولكن، كيف يجب على المترجم أن يقرر عن الأخطاء الموجودة في البرنامج ؟ على الأقل يجب على المترجم أن يقر عن مكان الخطأ الذي تم الكشف عنه، لأنه هناك احتمال كبير عن وقوع الخطأ الحقيقي في الأجزاء النصية **tokens** السابقة لمكان المؤشر إليه عن وجود الخطأ فيه. كما و من الأفضل أن يكون التقرير عن الخطأ بشكل إعلامي صريح (مثال ذلك الرسالة "**semicolon missing at this position**").

ما إن يتم الكشف عن وجود الخطأ ماذا يجب أن يفعل المعرب **parser** ؟ كما سوف نرى هناك عدة استراتيجيات العامة، و لكن لا توجد واحدة هي المسيطرة. في معظم الحالات ليس من الملائم إيقاف عملية التحليل التركيبي (الإعراب) عند ورود الخطأ الأول في البرنامج المترجم، لأن المعالجة التالية لمكان الخطأ قد تكشف أخطاء أخرى إضافية في البرنامج. في العادة هناك أشكال من الأخطاء يقوم المعرب فيها إعادة ضبط نفسه إلى وضع الذي يكون فيه معالجة الدخل التالي لمكان الخطأ أملاً في أن يستطيع فيه أن يتابع معالجة الدخل السليم بشكل صحيح.

إن التصرف غير الملائم يمكن أن يؤدي إلى إرباك مزعج يؤدي إلى الكشف عن أخطاء كاذبة و التي قد لا يكون المبرمج قد ارتكبها، وإنما قد تولدت نتيجة لدخول المعرب إلى حالة ما تؤدي إلى هذا الشيء و ذلك عند معالجة خطأ ما سابق. و ما يشابه ذلك، فإن معالجة الأخطاء التركيبية قد تؤدي إلى توليد أخطاء معاني كاذبة أيضاً و التي سيكشف عنها نتيجة التحليل اللاحق للمعاني أو في طور توليد الشيفرة الوسيطة.

كمثال على هذا كله، يمكن أن يقوم المعرب بإهمال التصريحات عن بعض المتحولات نتيجة لمعالجة أمر خطأ تركيبي ما مما يؤدي في مكان لاحق للخطأ و عند استخدام هذا المتحول (الذي تم إهمال التصريح عنه)

في تعبير لاحق فإنه لا توجد معلومات عنه في جدول الرموز مما يؤدي إلى صدور رسالة تبين عدم التصريح عن مثل هذا المتحول.

الاستراتيجية الحذرة **conservative** التي يمكن أن يعتمدها المترجم تقوم بكبح مجموعة رسائل الخطأ المتولدة في مكان القريب من الخطأ في سلسة الدخل، حيث يوجب المعرب معالجة بعض العناصر النصية الصحيحة التالية لمكان الخطأ قبل أن يبدأ في توليد رسائل أخطاء جديدة. في بعض الحالات يمكن أن يكون هناك العديد من الأخطاء التي يمر عليها المترجم قبل أن يبدأ بعملية المعالجة المحسوسة للدخل.كما هو واضح أن استراتيجية اكتشاف الأخطاء يجب أن يتم معالجتها بشكل دقيق مع الأخذ بعين الاعتبار أنواع الأخطاء التي من الممكن أن تحصل و التي من المعقول معالجتها.

كما ذكرنا آنفاً فإن بعض المترجمات تقوم بتصحيح الأخطاء و التي تحاول التخمين مالذي كان المبرمج يريد أن يكتبه أو ماذا كان يقصد. كمثال على مثل هذه المترجمات مترجم لـ **PL/C** من (**Conway and Wilox [1973]**).

من الممكن استثناء تلك بيئات البرمجة الموجهة نحو البريمجات الصغيرة المكتوبة من قبل الطلاب و التي تكون ذات كثافة أخطاء كبيرة، بحيث ليس من المجدي فعلياً إضافة هذه المزايا إليها، و حيث يكون التوجه في هذه الحالة نحو بناء بيئات برمجة متفاعلة، حيث فيها من السهل إيجاد الأخطاء و معالجتها تفاعلياً.

استراتيجيات معالجة الأخطاء

هناك العديد من الاستراتيجيات العامة المختلفة و التي يكون المعرب فيها موظفاً لاكتشاف الأخطاء التركيبية. كما و لم تثبت أي استراتيجية منها لتكون هي المعتمدة بشكل عام. و فيما يلي مجموعة من هذه الاستراتيجيات :

- النمط المسعور **panic mode**.
- مستوى العبارة **phrase level**.
- طريقة إنتاج (اشتقاق) الأخطاء **Error productions**.
- التصحيح الشامل **global correction**.

طريقة النمط المسعور **panic mode**: و هي أبسط الطرق التي يمكن تطبيقها، كما و يمكن أن تستخدم من أجل أية طريقة إعراب. ففي حال اكتشاف الخطاء، يقوم المعرب بإهمال العناصر اللفظية **tokens** واحد تلو الآخر إلى أن يجد عنصر لفظي تزامني أو دوري. من هذه العناصر اللفظية المحددات مثل الفواصل المنقوطة أو **end** و التي لها دور واضح في البرنامج المصدر. و هنا يكون دور مصممي المصنف في أن يختاروا مثل هذه العناصر اللفظية و التي تكون خاصة بلغة المصدر. مع أن هذه الطريقة في معالجة الأخطاء تقوم بإهمال جزء من

الدخل و التي يمكن أن تحوي أخطاء أخرى، فإنها سهلة التطبيق و تضمن عدم دخول المترجم في حلقات لا نهائية كما هو الحال في الطرق الأخرى من معالجة الأخطاء، كما و تناسب تلك الحالات التي لا يكون فيها عدد الأخطاء كبيراً في الأمر الواحد.

طريقة مستوى العبارة **phrase level**: و تعتمد على أنه في حال اكتشاف الأخطاء فإن المعرب يقوم بتصحيح موضعي للدخل التالي، و ذلك من خلال إجراء عملية استبدال لبادئة سلسلة الدخل بأجزاء لفظية أخرى تسبب الاستمرار في عملية الإعراب مثل استبدال الفاصلة بالفاصلة المنقوطة أو إدخال فاصلة منقوطة إلى السلسلة قد تحتاجها. القرار في تحديد ما يمكن إضافته متروك للمصمم. كما و يجب أن نختار ما يمكن إضافته بعناية و ذلك بما يكفل عدم الدخول في حلقات لا نهائية من عمليات الإضافة هذه. مثال ذلك إدخال عنصر هو في الأصل موجود في بداية سلسلة الدخل.

إن هذا الاستبدال يقوم بتصحيح أي سلسلة دخل و هذا مستخدم في العديد من المترجمات التي تقوم بتصحيح الأخطاء تلقائياً. إن هذه الطريقة قد تم استخدامها في المعربات التي تعتمد التحليل التركيبي من الأعلى إلى الأسفل. من نقاط الضعف في مثل هذه الطريقة عدم قدرتها على تصحيح الأخطاء التي حصلت قبل المكان الذي تم اكتشاف الخطأ فيه.

طريقة إنتاج (اشتقاق) الأخطاء **Error productions**: في حال كانت لدينا فكرة جيدة عن الأخطاء الشائعة التي يمكن أن تكون موجودة، فإنه من الممكن أن نضع قواعد اللغة مع تضمين الأخطاء فيها بحيث يمكن أن نولد من هذه القواعد سلاسل التي تحوي أخطاء لغوية فيها. و من ثم بناء محلل تركيبي لهذه اللغة و معالجة أماكن الأخطاء التركيبية التي تم التعرف عليها في الدخل.

التصحيح الشامل **Global correction**: في الحالة المثالية نحن نريد من المترجم أن يقوم ببعض التعديلات في سلسلة الدخل من أجل معالجة الدخل غير الصحيح. و هناك خوارزميات خاصة في تحديد العدد الأصغري لعمليات التعديل في سلسلة الدخل من أجل تصحيحها.

فمثلاً من أجل سلسلة الدخل **x** و القواعد **G** فإن هذه الخوارزميات تقو بتوليد شجرة الإعراب للسلسة **y** وذلك بحيث يكون عمليات التعديل على **y** من حذف و إضافة أصغرياً. لسوء الحظ فإن هذه الطرق مكلفة زمنياً و من حيث سعة الذاكرة و بالتالي فهي تهم عمليات البحث النظري فقط.

كما ويجب أن نبين أن الاستبدال الأصغري ليس دائماً يتطابق مع ما يدور في خلد المبرمج، و مع ذلك فإن فكرة التعديل الأصغري هي المعيار في تقييم مختلف طرق معالجة الأخطاء و هي مستخدمة في تحديد السلاسل البديلة في طريقة مستوى العبارة **phrase level**.

القواعد الخالية من السياق context-free grammar

العديد من لغات البرمجة لها بنية تركيبية ذات صفة تعاودية و التي يمكن تعريفها من خلال القواعد الخالية

من السياق، مثال ذلك يمكن أن نعرف الأمر الشرطي من خلال القاعدة التالية :

إذا كان **S1** و **S2** أوامر و **E** هو تعبير فإن

if E then S1 else S2

هو أمر.

إن مثل هذا الشكل من الأوامر لا يمكن وصفه باستخدام التعابير النظامية و التي تستخدم عادة في وصف العناصر النصية للغة،

بفرض المتحول التركيبي **stmt** يصف الصنف من النوع (أمر) و المتحول **expr** يصف الصنف (تعبير) يمكننا كتابة التعبير القواعدي السابق بالشكل التالي :

```
stmt → if expr then stmt else stmt
```

تتألف القواعد بشكل عام من الرموز المحددة، الرموز غير المحدد، رمز البداية، و الاشتقاق.

1 – الرموز المحددة terminals هي الرموز الأساسية و التي منها تتكون السلاسل، فإن العنصر اللفظي token هو المرادف للرمز المحدد terminal و ذلك عندما نتحدث عن القواعد من أجل لغة برمجة ما. و في التعبير السابق فإن كل من if، then، و else هي عبارة عن رموز محددة.

2 – الرموز غير المحددة nonterminals هي عبارة عن متحولات تركيبية، و هي تشير إلى مجموعات من السلاسل. من المثال السابق فإن stmt و expr هي عبارة عن رموز غير محددة. إن الرموز غير المحددة تعرف مجموعات السلاسل و التي تساعد في تعريف اللغة المولدة من القواعد. كما و هي أيضاً تفرض أو تقدم البنية التركيبية للغة و التي تكون مفيدة لكل من التحليل التركيبي و التحليل اللفظي و عملية الترجمة.

3 – في القواعد هناك رمز غير محدد واحد و الذي يدعى برمز البداية، و هذا الرمز يعرف مجموعة من السلاسل و التي هي اللغة المعرفة من خلال القواعد.

4 – الإشتقاقات في القواعد تصف الطريقة التي يمكن ضم الرموز المحددة و غير المحددة من أجل تكوين السلاسل. كل اشتقاق يتكون من رمز غير محدد ملحوقاً بسهم (أو الرمز =:: في بعض الأحيان بدلاً منه) ملحوقاً بسلسلة من الرموز المحددة و غير المحددة.

مثال

القواعد التالية تعرف التعبير الرياضي البسيط :

```
expr → expr op expr
expr → (expr)
expr → - expr
expr → id
op → +
```

```
op → -
op → *
op → /
op → ^
```

في هذه القواعد :

الرموز المحددة هي	**+,-,*,/,(,),id**
الرموز غير المحددة	**op, expr**
الرمز البدائي	**expr**

ملاحظات حول الاصطلاحات

من أجل تجنب ذكر العبارات (هذه هي الرموز المحددة) و (هذه هي الرموز غير المحددة) سوف نستخدم الاصطلاحات التالية في هذا الكتاب :

- **الرموز المحددة :**

  الأحرف الصغيرة الأولى مثل a,b,c,d ...

  رموز العمليات +,− ...

  علامات الترقيم مثل الفواصل و الأقواس ...

  الخانات الرقمية مثل 0,1,...,9.

  سلاسل الأحرف السميكة.

- **الرموز غير المحددة :**

  الأحرف الكبير الأولى مثل A,B,C ...

  الرمز S هو عبارة عن رمز البداية.

  الأسماء بالأحرف الصغيرة المائلة

- **الأحرف الكبيرة الأخيرة في الأبجدية متثل X,Y,Z تمثل رموز القواعد و التي يمكن أن تكون محددة أو غير محددة .**

- **الأحرف الصغيرة الأخيرة في الأبجدية مثل u,v,...,z تمثل سلاسل من الرموز المحددة.**

- **الأحرف الإغريقية مثل α,β,γ سلسلة من رموز القواعد المتنوعة. مثلاً يمكن كتابة عملية الاشتقاق بشكل عام وفق التعبير A→α و التي تظهر أن هناك رمز غير محدد على يسار السهم هو A وسلسلة من رموز القواعد α في الطرف اليميني من السهم.**

- إذا كان لدينا الاشتقاقات التالية :

$A \rightarrow \alpha_1$
$A \rightarrow \alpha_2$
...
$A \rightarrow \alpha_k$

بحيث يكون لدينا الرمز **A** موجود على الطرف اليساري منها. فإنه من الممكن التعبير عن جميع هذه الاشتقاقات بالشكل التالي :

$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$

و ندعو كل من $\alpha_1,\alpha_2,\ldots,\alpha_k$ الخيارات من أجل الرمز **A**.

- في باقي الحالات يكون الطرف اليساري في أول قانون (الاشتقاق) هو رمز البداية.

مثال

باستخدام هذه الاختصارات، يمكننا كتابة القوانين التالية :

```
expr → expr op expr
expr → (expr)
expr → - expr
expr → id
op → +
op → -
op → *
op → /
op → ^
```

بالشكل التالي :

```
E → E A E | (E) | -E | id
A → +|-|*|/|^
```

و من خلال الاصطلاحات الخاصة بنا نجد أن **E** و **A** هي رموز غير محددة، و **E** هو رمز البداية، أما باقي الرموز هي رموز محددة.

الاشتقاقات:

هناك عدة طرق لإيضاح الآلية التي تعرف بها القواعد اللغة الخاصة بها. في الفصل الأول تم استعراض هذه الآلية منن خلال بناء شجرة الإعراب، و هناك أيضاً طريقة مفيدة أخرى تظهر لنا آلية الاشتقاق و التي كثيراً ما تكون مفيدة، هذه الطريقة وصف دقيق لعملية بناء شجرة الإعراب من الأعلى إلى الأسفل. و الفكرة الأساسية فيها هو أن عملية الاشتقاق تظهر من خلال إعادة كتابة الرمز غير المحدد بالطرف الأيمن للقانون الذي تم استخدامه في الاشتقاق.

كمثال لنفرض أنه لدينا القواعد التالية الخاصة بالتعابير الرياضية و التي فيها الرمز **E** يعبر عن التعبير :

```
E → E + E | E * E | (E) | -E | id
```

إن الاشتقاق E - → E و يعني أن التعبير المسبوق برمز السالب هو تعبير رياضي، و هنا نجد أن هذا الاشتقاق يسمح لنا بناء تعابير أعقد من خلال استبدال E بـ "E-".

و بالتالي فإن هذا الاستبدال يمكن التعبير عنه بالشكل E ⇒ - E

و التي تعني E تشتق –E، كما أن القانون E→(E) يعن أننا نستطيع استبدال E بـ "(E)". مثال : E * E ⇒ (E) * E or, E * E ⇒ E * (E).

من الممكن أخذ الرمز E و تطبيق الاشتقاق بالترتيب الذي نريد و ذلك من أجل الحصول على تتالي من عمليات الاستبدال مثال : E ⇒ - E ⇒ -(E) ⇒ -(id).

ندعو هذا التتالي من عمليات الاستبدال المتكرر باشتقاق السلسة –(id) إعتباراً من E، و هذه الاشتقاقات المتتالية تقدم برهان أن السلسلة –(id) هو إحدى أشكال التعبير الرياضي الموصوف بالقواعد السابقة.

و بشكل مجرد أكثر، يمكننا القول أن αAβ ⇒ αγβ في حال لدينا A → γ و كان كل من α و β سلاسل ما مؤلفة من رموز القواعد. و في حال كان لدينا $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ فإننا نقول أن α_1 تشتق α_n، و الرمز ⇒ تعني الاشتقاق لمرحلة واحدة، كثيراً ما نحتاج القول "الاشتقاق لصفر مرة أو أكثر" ومن أجل هذا نستخدم الرمز ⇒* و بالتالي يكون لدينا:

1. a *⇒ a for any string a
2. if a *⇒ b and b ⇒ g, then a *⇒ g

و بنفس الطريقة نستخدم الرمز ⇒+ من أجل الدلالة على "الاشتقاق لمرة واحدة على الأقل".

من أجل قواعد ما G ذات الرمز البدائي S، يمكننا استخدام العلاقة ⇒+ من أجل تعريف L(G) و التي هي اللغة المولدة بواسطة G، مع العلم أن جميع الرموز في مجموعة السلاسل L(G) هي رموز محددة من القواعد G. و نقول أن السلسلة ω تنتمي إلى L(G) إذا و فقط S ⇒+ ω.

و ندعو السلسلة w بأنها جملة من G. و اللغة التي يمكن توليدها من القواعد G تدعى لغة خالية من السياق. في حال كان لدينا مجموعتين مختلفتين من القواعد، و كلاهما تولدان نفس اللغة، فإننا نقول أن هاتان المجموعتان من القواعد متكافئتان.

إذا كان لدينا α ⇒* S و كانت α تحوي على رموز غير محددة فإننا ندعوها شكل جملي من G، و الجملة في كل الأحوال هي شكل جملي خالٍ من الرموز غير المحددة.

مثال

السلسلة (id+id)– هي جملة تنتمي للغة المولدة بالقواعد الذكورة سابقاً لأنه يمكن إجراء الاشتقاق التالي :

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$$

إن السلاسل السابقة هي جميعها أشكال جملية من القواعد G، و يمكننا كتابة :

$$E \overset{*}{\Rightarrow} -(id+id)$$

و ذلك من أجل الدلالة على أن (id+id)– يمكن اشتقاقها من E.

لقد بينا و خلال شرح الاشتقاق أن جميع الجمل المولدة بالقواعد السابقة هي تعبير رياضي بسيط يحوي العمليات الثنائية +،–،*،\ و العملية الأحادية –. و بالتالي يمكن القول أن القواعد هذه قواعد كتابة التعبير الرياضي.

في كل مرحلة من مراحل الاشتقاق يوجد قراران من الواجب اتخاذهما في كل عملية اشتقاق، و هما أي رمز سنشتقه من السلسلة، و أي اشتقاق للرمز سوف نستخدم. كمثال، من الممكن أن نشتق إعتباراً من (E+E)– بالشكل التالي :

$$(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$$
or
$$(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$$

في كل من الطريقتين قد تم استبدال نفس الرمز و بنفس الاشتقاق و لكن الترتيب مختلف في كل مرة.

من أجل الفهم الدقيق لعمل المعربات، من الواجب الافتراض أن الاشتقاق يطبق في كل مرة على الرمز غير المحدد الأيسر من السلسلة دوماً. و مثل هذا الاشتقاق يدعى الاشتقاق اليساري و يرمز بـ $\overset{lm}{\Rightarrow}$

$$E \overset{lm}{\Rightarrow} -E \overset{lm}{\Rightarrow} -(E) \overset{lm}{\Rightarrow} -(E+E) \overset{lm}{\Rightarrow} -(id+E) \overset{lm}{\Rightarrow} -(id+id)$$

و باستخدام مصطلحاتنا فإن المرحلة اليسارية من الممكن أن نعبر عنها بالشكل $\omega A\gamma \overset{lm}{\Rightarrow} \omega\delta\gamma$.

حيث لدينا ω تتكون من الرموز المحددة فقط، و γ تتكون من رموز القواعد و من أجل التأكيد على أن α تشتق β يسارياً نكتب $\alpha \overset{lm}{\Rightarrow}{}^{*} \beta$، و إذا كان لدينا $S \overset{lm}{\Rightarrow}{}^{*} \alpha$، فإننا نقول أن α هي شكل جملي يساري للقواعد.

و بالشكل المشابه يمكننا وصف الاشتقاق اليميني، والذي يعني اشتقاق الرمز اليميني غير المحدد من السلسلة في كل مرحلة، و في بعض الأحيان يدعى هذا الاشتقاق بالاشتقاق القانوني.

أشجار الإعراب و الاشتقاقات

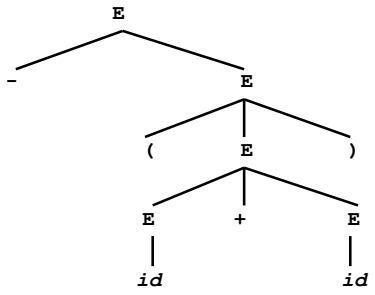
إن شجرة الإعراب من الممكن أن تظهر كتمثيل بياني للاشتقاق و لكن بإخفاء إظهار ترتيب الاشتقاق. إن كل عقدة من الشجرة تسمى برمز غير محدد، و أفرع كل عقدة تسمى بالرموز الواقعة في الطرف اليميني من قانون الاشتقاق للرمز غير المحدد، و أوراق الشجرة تسمى بالرموز المحددة من القواعد:

مثال

شجرة الإعراب للسلسلة (id+id)– و التي توافق الاشتقاق التالي :

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$

مبينة في الشكل 4–3.



الشكل (4–3) شجرة الإعراب للسلسلة (id+id)–

و من أجل تبيان العلاقة بين الاشتقاق و شجرة الإعراب، نفرض أنه لدينا سلسلة الاشتقاقات $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$، حيث أن α_1 هو سلسلة مكونة من رمز غير محدد وحيد A، و من أجل كل شكل جملي α_i من سلسلة الاشتقاق ، نقوم بتشكيل شجرة الإعراب و التي تعبر عن α_i. في البداية نفرض α_1=A و تكون الشجرة الموافقة لها هي عقدة وحيدة تسمى بـ A، و من أجل الاستمرار، نفرض أنه تم تكوين الشجرة من أجل $\alpha_{I-1} = X_1X_2\ldots X_k$، و مع فرض أن α_I مشتقة من α_{I-1} و ذلك من خلال استبدال الرمز غير المحدد X_j بـ $\beta= Y_1Y_2\ldots Y_k$ و بهذا نكون قد أنجزنا الاشتقاق ذو الترتيب i للسلسة بتطبيق القانون $A\rightarrow\beta$ على السلسلة α_{i-1} لنحصل على السلسلة $\alpha_i = X_1X_2\ldots X_{j-1}\beta X_{j+1}\ldots X_k$.

و هذا يكافئ بإعطاء الورقة X_j من الشجرة مجموعة أبناء هي $Y_1,Y_2,\ldots,Y_r$، في حال كانت β = e يتم إضافة ورقة و تسميتها بـ e، حيث أن e هو رمز السلسلة الفارغة أو الصفرية.

مثال

مع الأخذ سلسلة الاشتقاق التالي :

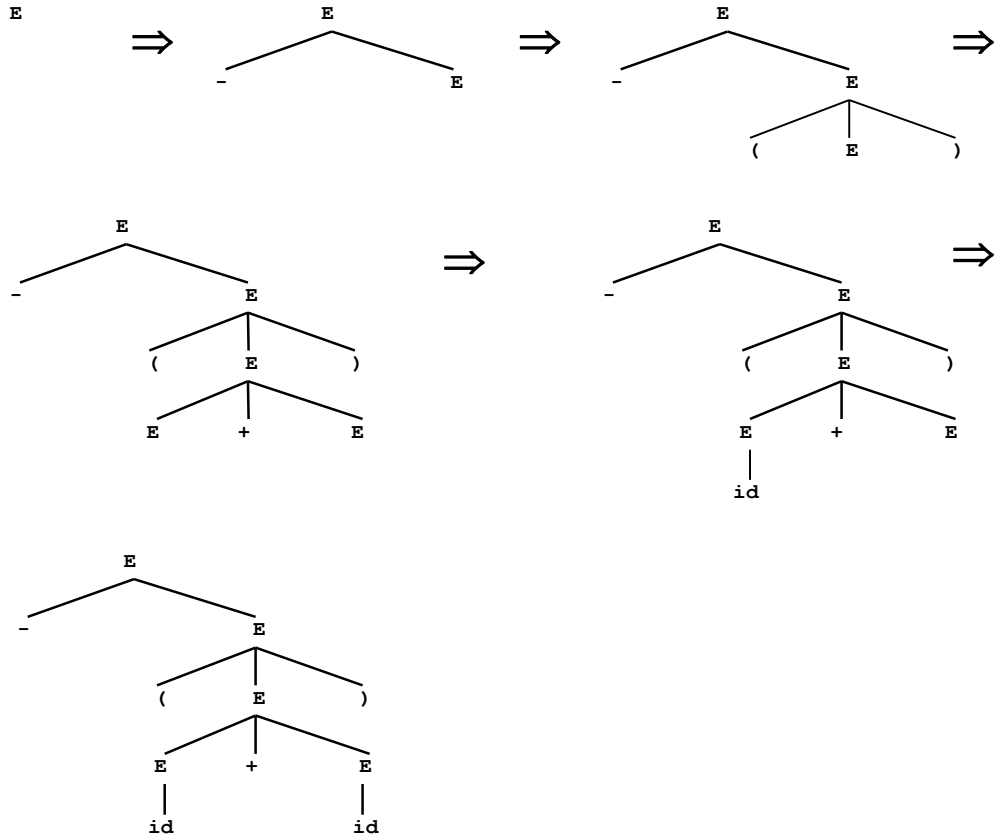
$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$

تكون مراحل تكوين شجرة الإعراب الخاصة بهذه السلسلة من الاشتقاق مبين في الشكل 4–4.

في الخطوة الأولى من عملية الاشتقاق $E \Rightarrow -E$. يتم تمثيل هذه الخطوة يتم إضافة ابنان للجذر E، و يتم تسميتهما بـ – و E و بذلك تتكون بذلك الشجرة الثانية.

في المرحلة الثانية من عملية الاشتقاق $E \Rightarrow -(E)$، يتم إضافة الأبناء (، E،)، للعقدة اليسرى E من الشجرة الثانية، لتنتج شجرة المعبرة عن (E)–. و بالاستمرار بهذه الطريقة تنتج شجرة الإعراب الكاملة و التي

هي الشجرة الأخيرة.



الشكل (4–4) مراحل بناء شجرة الاشتقاق للسلسلة (id+id)–

كما وذكرنا سابقاً فإن شجرة الإعراب تهمل ترتيب العمليات التي تمت عند بنائها أو بعبارة أخرى الرموز في الشكل الجملي التي تم استبدالها في كل مرحلة. كمثال، لو أن عملية الاشتقاق استمرت كما في الشكل (id+id)- ⇒ (E+id)- ⇒ (E+E)-، فإن شجرة الإعراب الناتجة ستكون كما هي في الحالة السابقة. و بهذا نجد أنه من الممكن أن يتم تطبيق عمليات التغيير على الشجرة بما يتوافق مع الاشتقاق اليساري أو اليميني. و كما هو واضح فإن كل شجرة توافق شكل و حيد لعملية الاشتقاق اليميني أو الاشتقاق اليساري. في الحالات القادمة سوف نجري عملية الإعراب اليميني أو اليساري. و بحكم كوننا فهمنا هذا الأمر فإننا سوف نستعرض شجرة الإعراب فقط. كما و علينا أن نعلم أنه ليس من الشرط وجود شجرة إعراب وحيدة من أجل كل جملة (سلسلة) أو عملية اشتقاق يميني أو يساري وحيد.

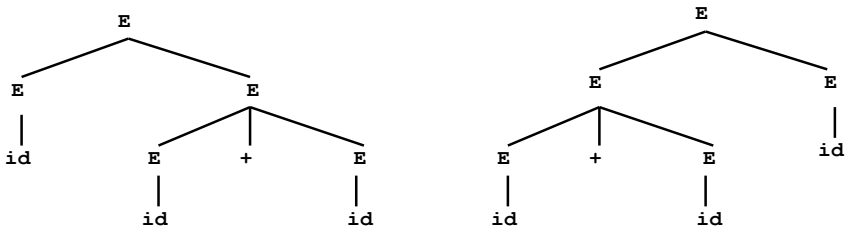
مثال

لنأخذ قواعد لغة التعابير الرياضية السابقة، نجد أنه و من أجل السلسلة id+id*id تملك اشتقاقين يساريين

مختلفين و هما:

1. E ⇒ E+E ⇒ *id*+E ⇒ *id*+E*E ⇒ *id*+*id**E ⇒ *id*+*id***id*
2. E ⇒ E*E ⇒ E+E*E ⇒ *id*+E*E ⇒ *id*+*id**E ⇒ *id*+*id***id*

و الذان لهما شجرتي إعراب موافقتين لهما مبينتان في الشكل التالي:



الشكل (4–5) شجرتا اشتقاق للسلسلة id+id*id

لاحظ أن شجرة الإعراب اليسرى تعكس كون عملية + ذات أفضلية أعلى من العملة * في السلسلة، بينما في الشجرة الأخرى لا تأخذ ذلك بعين الاعتبار. كما هو معروف رياضياً فإننا نتعامل على أساس أن عملية الضرب تملك أولوية على الجمع، و هذا يوافق أننا نحسب التعبير a+b*c بالشكل a+(b*c) بدلاً من (a+b)*c.

الغموض *Ambiguity*

إن القواعد التي تنتج أكثر من شجرة إعراب واحدة من أجل جملة ما يقال أنها قواعد غامضة **ambiguous**. بتعبير آخر، فإن القواعد الغامضة هي تلك التي تنتج أكثر من اشتقاق يميني أو اشتقاق يساري من أجل السلسلة نفسها. من أجل أصناف محددة من المُعربات، من المفضل أن يتم وضع القواعد لتكون غير غامضة، لأنه لو لم يتم ذلك لن يكون هناك أي إمكانية لتحديد أي شجرة إعراب يجب اعتمادها من أجل سلسلة دخل ما بشكل قطعي من أجل سلسلة ما. كما و سوف نلاحظ أنه و في بعض التطبيقات من القواعد الغامضة بوجه التحديد، و ذلك مع استخدام قواعد التخلص من الغموض و التي تتخلص من الأشجار التي لا نحتاجها، مما يبقي شجرة فريدة من أجل كل جملة لغوية.

وضع قواعد اللغة

إن القواعد قادرة على وصف معظم و لكن ليس مجمل الهيكلية للغة البرمجة. فهناك قدر محدود من التحليل التركيبي يقوم به المحلل اللفظي و الذي يقوم بتكوين العناصر اللفظية من المحارف الدخل. كما و هناك قيود محددة على الدخل مثل أن المميزات (الأسماء) يجب أن تكون مصرح عنها قبل أي استخدام لها، لا يمكن وصفها من خلال قواعد اللغة الخالية من السياق **Context-Free Grammar**، و لذلك فإن العناصر

النصية التي يقبلها المحلل التركيبي تكون تابعة لمجموعة عامة، و بالتالي من الواجب أن تقوم المراحل التالية من عملية الترجمة بأن تقوم بعمليات الاختبار على خرج المحلل التركيبي (المُعرب) و ذلك للتأكد من توافق الدخل مع قواعد اللغة و التي لم يتم بعد التأكد من تحققها.

سوف نبدأ هذا الفصل مع الأخذ بعين الاعتبار توضيح تقسيم العمل ما بين المحلل اللفظي و المحلل التركيبي. إن كل طريقة في إجراء عملية الإعراب تتطلب قواعد ذات شكل خاص، فإن القواعد التي يتم وضعها، قد يكون من الواجب أن تجرى عليها بعض التعديلات لأجل أن توافق الطريقة التي ستكون مختارة لعملية الإعراب. فالقواعد الخاصة بالتعابير الرياضية من الممكن أن تكون مبنية بالاعتماد على معلومات حول الترافق و الأسبقية. في هذا القسم سوف نستعرض بعض التحويلات على القواعد لكي تلائم الإعراب من الأعلى إلى الأسفل (**top-down parsing**). و سوف نستعرض بعض اللغات التي لا يمكن وصفها من خلال أية القواعد.

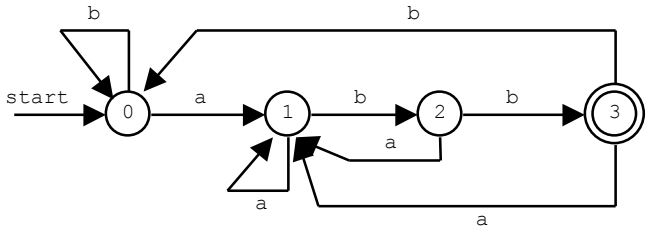
الموازنة بين التعابير النظامية و القواعد الخالية من السياق

إن أي تركيب يمكن وصفه من خلال التعابير النظامية، من الممكن و صفه من خلال القواعد الخالية من السياق، فمثلاً فإن التعبير النظامي **(a|b)*abb** و القواعد الخالية من السياق التالية :

$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$
$A_1 \rightarrow bA_2$
$A_2 \rightarrow bA_3$
$A_3 \rightarrow e$

تصف اللغة ذاتها، و التي هي المجموعة من السلاسل المؤلفة من الأحرف **a** و **b** و المنتهية بـ **abb**.

و من الممكن و بشكل ميكانيكي تحويل الأوتومات غير المحدد (**NFA**) إلى القواعد التي تولد نفس اللغة التي يميزها الـ **NFA**. إن القواعد السابقة قد تم تركيبها بالاعتماد على الـ **NFA** الموضح بالشكل **4–6**.



الشكل (**4–6**) الـ **DFA** الذي يتقبل **(a|b)*abb**

و ذلك بالاعتماد على الطريقة التالية :

من أجل أي حالة **i** من الـ **NFA** يتم تكوين رمز غير محدد، A_i. فإذا كانت الحالة **I** تملك انتقال إلى الحالة **j** من خلال رمز ما **a** مثلاً، يتم إدراج الاشتقاق $A_I \rightarrow aA_j$. و إذا كان هناك انتقال من الحالة **I** إلى

الحالة **j** برمز دخل هو **e**، يتم إضافة القاعدة $A_i \rightarrow A_j$. إذا كانت الحالة **i** هي حالة قبول، يتم إضافة القاعدة $A_i \rightarrow e$. و إذا كانت **i** هي حالة بدائية، يتم جعل A_i رمز بداية للقواعد.

بما أن كل مجموعة التعابير النظامية هي لغة خالية من السياق، فمن الممكن أن نطرح السؤال: "لماذا نستخدم التعابير النظامية لوصف البنية اللفظية للغة؟".

في الحقيقة هناك العديد من الأسباب في ذلك، هي:

- إن القواعد اللفظية للغة ما هي غالباً ما تكون بسيطة، و بالتالي فليس هناك من حاجة لاستخدام أداة قوية مثل القواعد الغالية من السياق في توصيفها.
- إن التعابير النظامية تقدم تنويت (تدوين) أكثر اختصاراً، و أكبر سهولة في الفهم العناصر النصية مما هو الحال في القواعد.
- من الممكن بناء محلل لفظي ذو فعالية أكبر – و بشكل مؤتمت و ذلك اعتباراً من التعابير النظامية – مما هو الحال في القواعد التحكمية.
- إن تقسيم البنية التركيبية للغة إلى قسم تركيبي، و قسم غير تركيبي، يقدم طريقة مناسبة لتقسيم النهاية الجبهية للمترجم إلى موديولين مُحجمين و طيعين.

ليس هناك من خطوط رئيسية توضح مالذي يجب وضعه في القواعد اللفظية على عكس ما هو الحال في القواعد التركيبية. إن التعابير النظامية مفيدة في وصف هيكلية التراكيب اللفظية مثل المميزات (الأسماء)، الثوابت و الكلمات المفتاحية ...إلخ. بينما ومن جهة أخرى، القواعد أكثر فائدة في وصف التراكيب المعششة أو التعاودية، مثال الأقواس المتوازنة، أو كشف الـ **begin(s)** و الـ **end(S)** المتلازمة، و كذلك الأمر بالنسبة إلى الـ **if-then-else** ... الخ. و كما سبق و ذكرنا، فإن هذه التراكيب لا يمكن وصفها من خلال التعابير النظامية.

فحص اللغة المتولدة عن القواعد

مع أن مصممي المترجمات نادراً ما يفعلون ذلك من أجل مجمل قواعد لغة البرمجة، و لكن من الممكن فعل ذلك على التراكيب المعقدة، و ذلك من خلال كتابة قواعد مجردة بشكل جزئي من قواعد اللغة، و إجراء دراسة عليها و على اللغة المولدة بها. و سوف نقوم بتكوين مثل هذه القواعد.

البرهان على أن القواعد **G** تولد اللغة **L** يتكون من قسمين :

1. علينا أن نبرهن أن أي سلسلة مولدة بواسطة **G** تنتمي إلى اللغة **L**.
2. و أن كل سلسلة من **L** من الممكن أن تكون بالإعتماد على **G**.

مثال

ليكن لدينا القواعد التالية:

```
S → (S)S | e
```

من الممكن أن لا يكون ظاهراً بشكل مبدئي، ولكن هذه القواعد البسيطة تولد فقط سلاسل من الأقواس المتوازنة و لا شيء آخر. و من أجل رؤية ذلك، سوف نعرض عليكم أن كل جملة قابلة للاشتقاق من S هي جملة ذات أقواس متوازنة. و أن كل جملة مكونة من أقواس متوازنة هي جملة يمكن اشتقاقها من S، من اجل إظهار أن كل جملة مشتقة من S هي متوازنة، سوف يتم الاعتماد على برهان افتتاحي على عدد من الخطوات في عملية الاشتقاق. من أجل الخطوة الأساسية في الاشتقاق و من أجل تلك الحالة التي تكون الجملة فيها مكونة من الرموز المحددة هي سلسلة فارغة ، و التي هي سلسلة موازنة الأقواس.

و الآن بفرض أن جميع الاشتقاقات و المكونة من عدد خطوات أقل من n تنتج جملة متوازنة، و مع اعتبار أن الاشتقاق هو يساري من أجل n خطوة، تكون شكل هذه السلسلة من الاشتقاقات له الشكل التالي:

```
S ⇒ (S)S *⇒ (x)S *⇒ (x)y
```

إن عدد مراحل اشتقاق كل من x و y اعتباراً من S هي أقل من n، و مع الفرض أن كل من x و y متوازنة الأقواس. نتيجة لذلك فإن السلسلة (x)y يجب أن تكون موازنة.

و الآن و قد بيننا أن أي سلسلة التي يمكن اشتقاقها من S هي متوازنة. يجب البرهان أن أي سلسلة متوازنة الأقواس يمكن اشتقاقها من S. و من أجل هذه الغاية، سوف نأخذ بعين الاعتبار طول السلسلة. و كخطوة أساسية في البرهان نبدأ بالقول أن السلسلة فارغة هي سلسلة قابلة للاشتقاق من S.

و الآن و بفرض أن كل سلسلة عدد رموزها أقل من 2n يمكن اشتقاقها من S، و بأخذ السلسلة المتوازنة ω ذات طول 2n، حيث n >= 1. بالتأكيد أن السلسلة ω تبدأ بقوس يساري، و لتكن (x) هي البادئة الأقصر من ω و المكونة من أقواس يسارية و يمينية متوازنة، عندئذ فإن السلسة ω يمكن كتابتها بالشكل (x)y حيث كل من x و y سلاسل متوازنة. و بما أن طول كل من x و y أقل من 2n فإنه و بالفرض الجدلي فإن x و y من الممكن اشتقاقها اعتباراً من S، و لذلك نجد أننا نستطيع أن نوجد اشتقاق له الشكل التالي:

```
S ⇒ (S)S *⇒ (x)S *⇒ (x)y
```

و بهذا نجد أن ω = (x)y من الممكن اشتقاقها اعتباراً من S.

التخلص من الغموض في القواعد

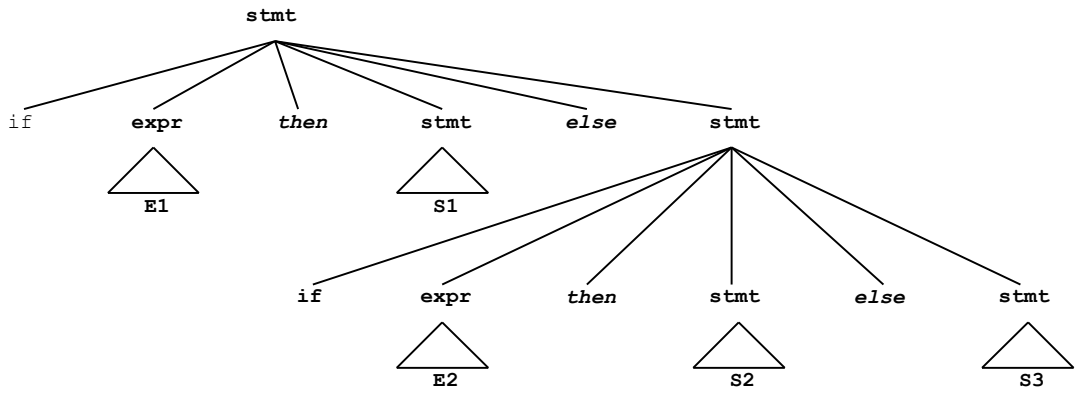
في بعض الأحيان من الممكن أن يعاد صياغة القواعد الغامضة من أجل التخلص من الغموض. و كمثال على ذلك سوف نقوم بحذف الغموض من القواعد التالية الخاصة بالـ dangling-else أي بالـ else المتدلي الخاص بالأمر if.

```
Stmt -> if expr then stmt
       |if expr then stmt else stmt
       |other
```

مع الفرض أن **other** هو أي أمر آخر. و بناء على هذه القواعد فإن الأمر الشرطي المركب التالي:

```
if E1 then S1 else if E2 then S2 else S3
```

يملك شجرة إعراب مبينة في الشكل **4–7**.



الشكل (**4–7**) شجرة الإعراب الخاصة بالأمر الشرطي

إن القواعد السابقة هي غامضة لأن السلسة التالية

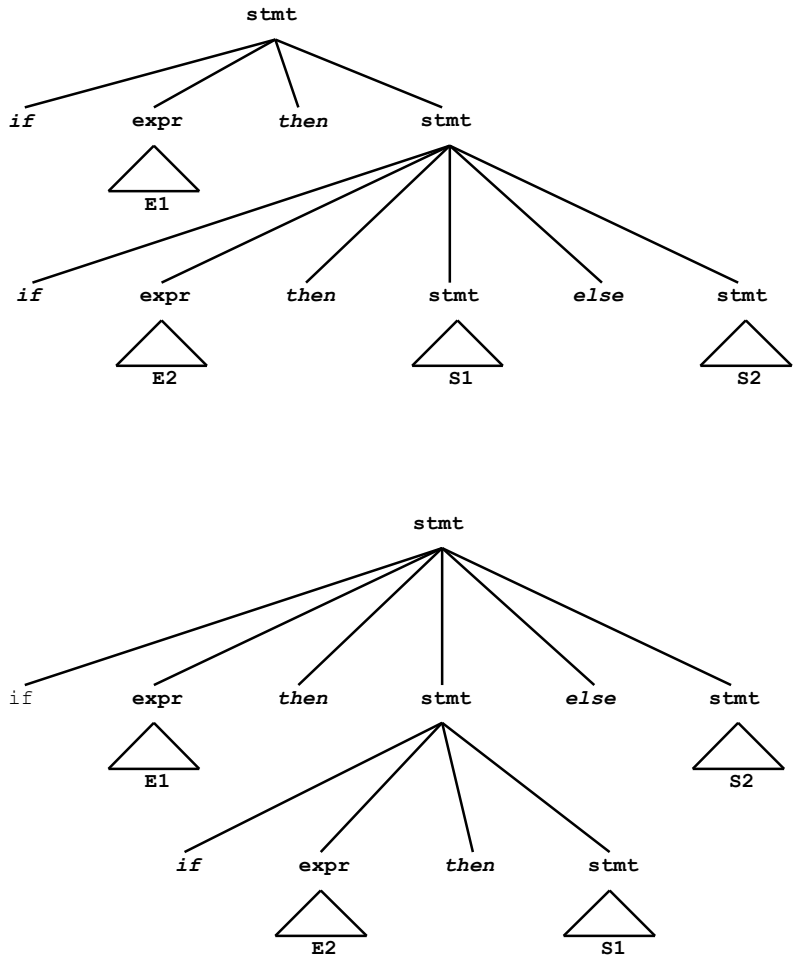
```
if E1 then if E2 then S1 else S2
```

مثلاً لها شجرتي إعراب كما هو مبين في الشكل **4–8**.

في جميع لغات البرمجة الحاوية على التعابير الشرطية المماثلة للشكل السابق، يكون فيها شجرة الإعراب الأولى هي المفضلة، و تكون بذلك القاعدة العامة في الأمر الشرطي هذا و بشكل عام هي " أن يتم ربط كل عبارة **else** مع عبارة **if** الأقرب إليها و التي لم يتم ربطها مع عبارة **else** خاصة بها. إن هذه القاعدة المستخدمة في التخلص من الغموض تترجم ليعبر عنها عن طريق القواعد لتكون بالشكل التالي :

```
stmt → matched_stmt
      | unmatched_stmt
matched_stmt → if expr then matched_stmt else matched_stmt
      | other
unmatched_stmt → if expr then stmt
      | if expr then matched_stmt else unmatched_stmt
```

نلاحظ أن القواعد هذه تنتج نفس مجموعة السلاسل الخاصة بالقواعد السابقة و لكنها تولد شجرة إعراب وحيدة من أجل السلسة في المثال السابق.



الشكل (8–4) شجرتي إعراب للسلسلة الغامضة $if\ E_1\ then\ if\ E_2\ then\ S_1\ else\ S_2$

Non-Context free Language Cunstructs *اللغات ذات السياق*

هناك بعض اللغات التي لا يمكن أن تتولد عن القواعد أي لا يمكن وصفها بقواعد خاصة لها. فهناك عدد من البنى التركيبية للعديد من لغات البرمجة لا يمكن وصفها من خلال قواعد اللغة فقط.

مثال

بفرض أنه لدينا اللغة L_1 بحيث تعرف كما يلي:

$$L_1 = \{\ \omega c \omega \mid \omega \text{ is in } (a|b)^*\ \}$$

أي أن L1 هي اللغو المكونة من جميع الكلمات و التي تتكون من سلسلة مكررة من الحرف a و b و بينها الحرف c، مثال ذلك السلسلة aabcaab.

من الممكن برهان أن مثل هذه اللغة هي ليست لغة خالية من السياق.

إن هذه اللغة تظهر و بنوع من التجريدي مشكلة فحص وجوج لتصريح عن مميزات أو أسماء قبل استخدامها في جسم البرنامج. حيث أن السلسلة الأولى ω تمثل التصريح عن المميز و السلسلة الثانية ω تعبر عن عن استخدام هذا المميز ضمن البرنامج. إن مثل هذه اللاحرية في السياق لـ L_1 تذكرنا باللا حرية في لغات البرمجة مثل الـ **Algol** و الـ **Pascal** و التي تتطلب التصريح عن الأسماء قبل استخدامها، أو القيود التي يمكن أن تفرض على الأسماء مثل تقييد طولها. و لهذا السبب فإن القواعد الواصفة للتركيب اللغة في كل من **Algol** و **Pascal** لا تحدد صفات الأسماء، و بالتالي فإن كل الأسماء تعامل كعنصر لفظي واحد ضمن القواعد يرمز بـ **id**. و في المترجم فإن مهمة فحص وجود التصريح عن الأسماء يترك لطور تحليل المعاني.

الإعراب من الأعلى إلى الأسفل ***Top-Down Parsing***

الإعراب التعاودي من الأعلى إلى الأسفل ***Recursive-Descent Parsing***

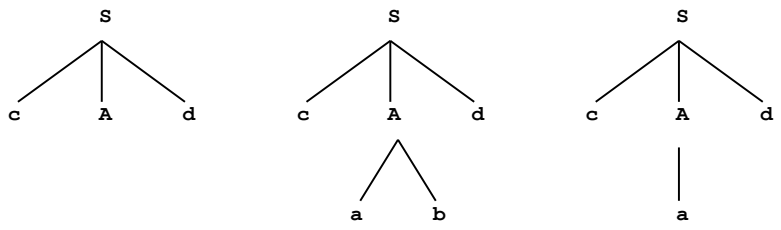
إن الإعراب (أو التحليل التركيبي) من الأعلى إلى الأسفل هو عملية يتم من خلالها البحث عن الاشتقاق اليساري لسلسلة الدخل. أو بصورة أخرى، من الممكن رؤيتة و كأنه عبارة عن محاولة بناء شجرة الإعراب من أجل سلسلة دخل ابتداءاً من الجذر و بناء عقد هذه الشجرة في ترتيب بادئي **preorder**.

هناك نوعان من الإعراب التعاودي من الأعلى إلى الأسفل، الأول هو الإعراب التوقعي، حيث لا يوجد فيه الحاجة إلى العودة إلى الخلف أثناء عملية التحليل التركيبي للسلسلة، و لكن الشكل العام من التحليل هو الذي قد يكون بحاجة إلة العودة إلى الخلف من أجل أخذ مسار تحليل تركيبي قد يكون الأول لم يصل إلى النجاح في تحليل لسلسلة الدخل. إن المعربات التي تنجز العودة إلى الخلف خلال عملها لا تستخدم كثيراً و السبب في ذلك هو أنه نادراً ما تحتاج لغة البرمجة ما إلى مثل هذا النوع من المعربات، و في حالات مثل معالجة اللغات الطبيعية، فإن الإعراب مع العودة إلى الخلف يعد ذو فعالية متدنية كما و هناك العديد من الطرق الأخرى مثل خوارزميات البرمجة الديناميكية و التي هي المفضلة.

مثال

بفرض لدينا القواعد التالية:

```
S → cAd
A → ab | a
```



الشكل (4–9) مراحل الإعراب من الأعلى إلى الأسفل

و بفرض أنه لدينا سلسلة الدخل **cad** = ω. من أجل بناء شجرة إعراب لهذه السلسلة و فق طريقة من الأعلى إلى الأسفل فإننا في البداية ننشئ العقدة **S** و نضع المؤشر على الحرف الأول من سلسلة الدخل و الذي هو **c** لدينا. نبدأ باستخدام الاشتقاق الأول للرمز **S** لتنشأ لدينا الشجرة الأولى/ نجد أن الورقة اليسرى الأولى تطابق الحرف الأول من السلسلة ω، و نتيجة لذلك نحرك المؤشر إلى الحرف الثاني في سلسلة الدخل ω، و الآن لننظر إلى الورقة التالية من الشجرة و التي هي **A**، و نلاحظ و بما أنها تمثل رمز غير محدد فإننا نستطيع أن نوسعها باستخدام الاشتقاق الأول لـ **A**، وبهذا نحصل على الشجرة الثانية في الشكل السابق. نلاحظ أنه حصل تتطابق في الرمز الثاني لسلسلة الدخل و الذي هو **a** مع الورقة المسماة بـ **a**. و نتيجة لذلك نحرك المؤشر إلى الحرف **d** ضمن السلسلة ω و بمقارنة الحرف الجديد من ω مع الورقة التالية و المسماة بـ **b** نجد عدم تطابق، و بهذا يتم التقرير عن الفشل و نعود إلى الوراء لنأخذ الاحتمالات الأخرى لعملية اشتقاق الرمز **A** و التي من الممكن أن تولد تطابق. و بهذا و بالرجوع إلى **A** علينا أن نعيد المؤشر إلى الحرف الثاني من سلسلة الدخل و هذا الموقع هو ذلك الذي كنا فيه عند وصولنا لمعالجة العقدة **A** في الشجرة، و هذا يعني أن الإجراء المسؤول عن توسيع **A** يجب أن يحتفظ بقيمة مؤشر سلسلة الدخل القديم. و الآن يتم استخدام الاحتمال الثاني لاشتقاق **A** و بذلك تنتج الشجرة الثالثة، عندها نجد أن الرمز **a** يطابق الرمز المشار إليه في سلسلة الدخل ω، و نجد أن الورقة الثالثة في الشجرة تطابق الحرف الأخير من السلسلة، و بهذا نوقف عملية الإعراب و نصدر إشارة النجاح في بناء شجرة الإعراب للسلسة ω.

نلاحظ أن الطريقة التعاودية لا تصل مع القواعد الحاوية على التعاودية اليسارية في قوانينها إلى نهاية عملية الإعراب نتيجة الاستمرار في استخدام الاشتقاق للرمز غير المحدد و الذي نقف عنده في الشجرة مع حصول توقف في قراءة سلسلة الدخل.

المعربات التوقعية ***Predicative Parsers***

أن الإنشاء المتقن لقواعد اللغة، و مع حذف التعاودية اليسارية منها، و بتحويلها لتكون ذات توليد يساري، كل ذلك يمكننا من الحصول على قواعد من الممكن بناء ما يسمى بالمعربات التوقعية لها و التي لا تتطلب تنفيذ حالات رجوع في عملها. إن الأساس الذي تعتمد عليه ممثل هذه المعربات بأنها تقاد خلال مراحل الإعراب من خلال معرفة رمز الدخل الحالي و ليكن **a** و الرمز غير المحدد **A** الذي نريد أن نستخدم أحد اشتقاقاته الممكنة $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ و من الممكن تحديد أي اشتقاق يجب الأخذ به و بشكل محدد تماماً بناء على الرمز **a** و الذي هو في البداية الحالية لسلسلة الدخل.

مثال

stmt → ***if*** **expr** ***then*** **stmt** ***else*** **stmt**

```
| while expr do stmt
| begin stmt_list end
```

نجد أن الكلمات المفتاحية **if**، **while**، و **begin** تخبرنا أي اشتقاق يمكننا أن نأخذه للرمز غير المحدد **stmt** و ذلك عند ورودها في بداية سلسلة الدخل.

الإعراب التوقعي لا تعاودي ***Nonrecursive Predictive Parsing***

إن المعربات التعاودية تعتمد على بناء إجراء من أجل كل رمز غير محدد و يتم استدعاء هذا الإجراء عند ورود الرمز غير المحدد الخاص به في شجرة الإعراب عند إجراء التوسع لعقدته (و الشجرة عادة ما تكون ممثلة بشكل ضمني و ليس بشكل صريح كبنية معطيات شجرية، حيث يكون عادة تسلسل استدعاء مجموعة من الإجراءات التعاودية و الخاصة بالرموز غير المحددة في القواعد، مكافئ لعملية عبور هذه الشجرة و فق ترتيب ما – **prefix** عادة – و معالجة كل عقدة من عقد الشجرة).

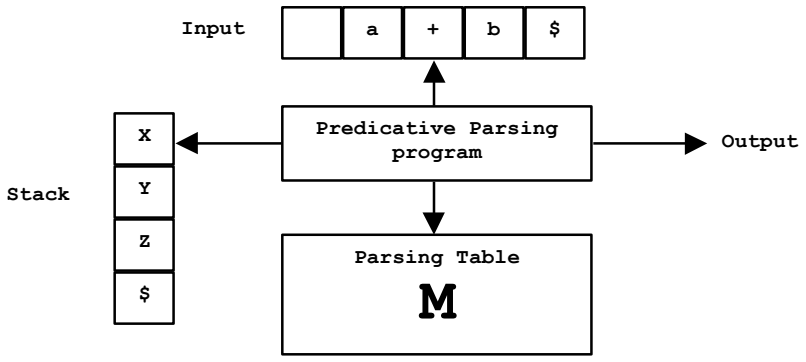
إن جسم الإجراء الخاص برمز ما **A** مثلاً يمثل من خلال العمليات التي يقوم بها الطرف اليميني من القانون و ذلك بالشكل التالي :

1. الرمز المحدد يكافئ عملية فحص وجوده في بداية سلسلة الدخل و التقدم بخطوة واحدة إلى الرمز التالي فيها.
2. الرمز غير المحدد يكافئ استدعاء الإجراء الخاص به.

إن الإعراب التوقعي لا تعاودي يعتمد على إنشاء مكدس صريح بدلاً من استخدام المكدس الخاص بالآلة الافتراضية للغة التي كتب بها المترجم (أو المصنف) و ذلك عند استخدام النداءات التعاودية للإجراءات الخاصة بالرموز غير المحددة و ذلك أثناء إجراء عملية الإعراب (أو التحليل التركيبي).

إن المسألة الأكثر جوهرية في الإعراب التوقعي هو تحديد الاشتقاق الخاص بالرمز غير المحدد الذي يجب تطبيقه في كل مرة يرد في شجرة الإعراب – الضمنية.

إن الشكل **4–10** يبين لنا النموذج البرمجي للمعرب التوقعي لا تعاودي و الذي يعتمد على جدول التفكيك في البحث عن الاشتقاق الذي يجب تطبيقه عند كل توسيع للرمز غير المحدد في الشجرة.



الشكل (4–10) نموذج المعرب التوقعي لا تعاودي

إن الإعراب أو التحليل التركيبي التوقعي المقاد بالجدول يتكون من دارئ الإدخال **Input buffer** و مكدس و جدول الإعراب (أو جدول التفكيك) **Pasring table**، و سلسلة الخرج **Output stream**.

إن دارئ الإدخال يحتوي على السلسلة المراد إعرابها متبوعة برمز نهاية الدخل $. المكدس يحوي على متتالية من رموز القواعد بالإضافة إلى الرمز $ الذي يشير إلى قعر المكدس. في الحالة الابتدائية من عملية الإعراب فإن المكدس يحتوي على رمز البداية S متوضعاً فوق $. جدول الإعراب عبارة عن مصفوفة ثنائية البعد **M[A,a]** حيث **A** هي الرمز غير المحدد و **a** عبارة عن رمز محدد أو $ رمز نهاية الدخل.

إن عملية التحليل التركيبي يتم تنفيذها من قبل برنامج يقوم بما يلي:

بفرض أن **X** هو الرمز في قمة الكدس و **a** رمز الدخ الحالي و من خلال هذين الرمزين يتم تحديد الفعالية التي سيقوم بها في كل مرة و التي يمكن أن تكون إحدى الحالات التالية :

1. في حال كانت **X = a = $** تتوقف عملية الإعراب مشيرة إلى نجاحها.
2. في حال كانت **X = a ≠ $** فإن المعرب يسحب **X** من قمة المكدس و يتقدم في سلسلة الدخل إلى رمز الدخل التالي.
3. في حال كان **X** هو رمز غير محدد فإن المعرب يجلب قيمة المركبة **M[X,a]** من جدول التفكيك **M** و قيمة هذه المركبة إما أن تكون اشتقاق خاص بـ **X** أو مؤشر إلى خطأ في الدخل. بفرض أن **M[X,a] = { X→ UVW }** فإن المعرب يقوم باستبدال **X** في قمة المكدس بـ **UVW** (**U** تكون في القمة).

و كخرج لعملية الإعراب فإن المعرب يقوم بطباعة الاشتقاق الذي تم استخدامه أو أن ينفذ عمل آخر يعبر عن خرج التحليل التركيبي.

إذا كانت المركبة **M[X,a]** فإن المعرب ينادي إجراء التصحيح أو يقوم بالتقرير عن الخطأ.

فيما يلي خوارزمية الإعراب التوقعي لا تعاودي:

Algorithm. Nonrecursive predictive parsing.

Input. A string ω and a parsing table M for grammar G

Output. If ω is in L(G), a leftmost derivation of ω; other wise, an error indication.

Method. Initial, the parser is in a configuration in witch it has $S on the stack with S, the start symbol of G on top, and ω$ in the input buffer.

```
1. Set ip to point to the first symbol of ω$;
2. Repeat
3.     Let X be the top stack symbol and a the symbol pointed to by ip;
4.     If X is a terminal or $ then
5.           If X = a then
6.                 Pop X from the stack and advance ip
7.           Else error()
8.     Else          /* X is a nonterminal */
9.           If M[X,a] = X → Y1Y2…Yk then begin
10.                Pop X from the stack;
11.                Push Yk,Yk-1,…,Y1 onto the stack, with Y1 on top;
12.                Output the production X → Y1Y2…Yk
13.          End
14.          Else error()
15. Until X = $  /* stack is empty */
```

مثال

بفرض لدينا القواعد لتالية:

```
E  →  TE′
E′ →  +TE′ | e
T  →  FT′
T′ →  *FT′ | e
F  →  (E) | id
```

يكون جدول التفكيك الخاص بها له الشكل 4–11.

Nonterminal	Input Symbol					
	$	)	(	*	+	Id
E			E → TE′			E → TE′
E′	E′ → e	E′ → e			E′ → +T E′	
T			T → FT′			T → FT′
T′	T′ → e	T′ → e		T′ → *F T′	T′ → e	
F			F → (E)			F → id

الشكل (4–11) جدول التفكيك في الإعراب التوقعي لا تعاودي

إن الخلايا الفارغة تعبر عن حالات الخطأ. الخلايا غير الفارغة تحوي الاشتقاقات التي يجب تطبيقها على الرمز غير المحدد في الظاهر في قمة المكدس.

بفرض لدينا سلسلة الدخل **id + id * id**، تكون الفعاليات التي يقوم المحلل التركيبي مبينة في الشكل **4–12**.

إذا راقبنا عملية الإعراب بدقة، فإننا سنلاحظ أنه تجرى عمليات الاشتقاق اليميني خلال عملية التحليل و الاشتقاقات المستخدمة تظهر كخرج لعملية التحليل.

Stack	*Input*	*Output*
$E	id + id * id $	
$E'T	id + id * id $	E→TE'
$E'T'F	id + id * id $	T→FT'
$E'T'id	id + id * id $	F→id
$E'T'	+ id * id $	
$E'	+ id * id $	T'→e
$E'T+	+ id * id $	E'→+TE'
$E'T	id * id $	
$E'T'F	id * id $	T→FT'
$E'T'id	id * id $	F→id
$E'T'	* id $	
$E'T'F*	* id $	T→*FT'
$E'T'F	id $	
$E'T'id	id $	F→id
$E'T'	$	
$E'	$	T'→e
$	$	E'→e

الشكل (**4–12**) الفعاليات التي تنفذ من قبل المعرب التوقعي لا تعاودي من أجل الدخل **id+id*id**

مفهوم *FIRST & FOLLOW*

إن بناء جدول التفكيك في الإعراب التوقعي يعتمد على التابعين المرتبطين بقواعد اللغة **G** و هما **FIRST** و **FOLLOW** و الذان يستخدمان في ملئ خلايا الجدول.

في حال كانت α سلسلة ما مكونة من رموز القواعد **G** فإن **FIRST(α)** هو مجموعة من الرموز المحددة التي تبدأ بها جميع السلاسل المشتقة من α، في حال كانت $\alpha \Rightarrow^* e$ تكون **e** هي أيضاً عنصر في المجموعة **FIRST(α)**.

إن **FOLLOW(A)** هي جميع الرموز المحددة α و التي تظهر مباشرة بعد الرمز غير المحدد **A** في بعض الأشكال الجملية الحاوية على **A** و التي تكون مشتقة اعتباراً من **S**، أو بشكل آخر : **FOLLOW(A) = { a : $S \Rightarrow^* \alpha A a \beta$, $\forall \alpha,\beta$}**.

في بعض الأحيان تكون هناك أشكال جملية (و التي هي بدورها سلاسل من رموز القواعد) واقعة بين **A** و **a**، و لكنها تشتق **e** و لذلك تكون غير ظاهرة.

في حال كون الرمز غير المحدد **A** الرمز الأخير في بعض الأشكال الجملية المشتقة من **S** يكون الرمز **$** عنصراً من **FOLLOW(A)**.

طريقة حساب التابع **FIRST(X)** :

1. إذا كان **X** هو رمز محدد، عندها يكون **FIRST(X) = {X}**.
2. في حال كانت **$X \rightarrow e$**، عندها يتم إضافة **e** إلى المجموعة **FIRST(X)**.
3. في حال **X** رمز غير محدد، وكان **$X \rightarrow Y_1Y_2...Y_k$** فإن **a** يكون عنصر في **FIRST(X)** من أجل بعض قيم لـ **I** حيث أن **a** هي عناصر **$FIRST(Y_I)$** و **e** هي عناصر في جميع **$FIRST(Y_1)...FIRST(Y_{I-1})$**، أي **$Y_1...Y_{I-1} \Rightarrow^* e$**. و في حال كانت **e** تنتمي إلى **$FIRST(Y_j)$** من أجل **j = 1,..,k** عندها تكون **e** هي عنصر من **FIRST(X)**، و يمكن القول بأن **$FIRST(Y_1) \subseteq FIRST(X)$**، و في حال كانت **$Y_1 \Rightarrow^* e$** عندئذ نضيف **$FIRST(Y_2)$** إلى **FIRST(X)** و هكذا ...

و الآن لحساب قيمة التابع **FIRST** من أجل السلسلة **$X_1X_2...X_n$**، نقوم بإضافة عناصر **$FIRST(X_1)$** دون **e** إن وجد، و إضافة رموز **$FIRST(X_2)$** دون الرمز **e** إن وجد و ذلك عندما **e** عنصر من **$FIRST(X_1)$**، و إضافة **$FIRST(X_3)$** عندما يكون **e** عنصر من كل من **$FIRST(X_1)$** و **$FIRST(X_2)$** و هكذا ... ، و أخيراً يتم إضافة الرمز **e** إلى **$FIRST(X_1X_2...X_n)$** عندما **$e \in FIRST(X_i)$** من لأجل **i=1,...,n**.

من أجل حساب التابع **FOLLOW(A)** للرمز غير المحدد **A**، يتم تطبيق الخطوات التالية:

1. إضافة الرمز **$** إلى **FOLLOW(S)**، و ذلك من أجل **S** هو رمز البداية و **$** هي رمز نهاية الدخل.
2. في حال كان **$A \rightarrow \alpha B\beta$**، فإن جميع عناصر **$FIRST(\beta)$** ما عدا **e** تضم إلى **FOLLOW(B)**.
3. في حال وجد اشتقاق **$A \rightarrow \alpha B$** أو **$A \rightarrow \alpha B\beta$** و **$FIRST(\beta)$** يحوي على **e**، عندئذ يكون **FOLLOW(A)** يكون ضمن **FOLLOW(B)**.

مثال

بفرض لدينا القواعد التالية :

```
E  →  TE'
E' →  +TE' | e
T  →  FT'
T' →  *FT' | e
F  →  (E) | id
```

يكون :

```
FIRST(E) = FIRST(T) = FIRST(F) = {(,id}
FIRST(E') = {+,e}
FIRST(T') = {*,e}
FOLLOW(E) = FOLLOW(E') = {),$}
FOLLOW(T) = FOLLOW(T') = {+,),$}
FOLLOW(F) = {+.*.),$}
```

بناء جدول التفكيك الخاص بالإعراب التوقعي

فيما يلي خوارزمية يمكن استخدامها في بناء جدول التفكيك من أجل القواعد G. إن فكرة الخوارزمية تعتمد على ما يلي :

بفرض أن $A\rightarrow\alpha$ قانون من G و كان $a \in FIRST(\alpha)$، فإن المعرب سوف يقو بتوسيع A بـ α عندما يكون a هو رمز الدخل الحالي. و التعقيد الوحيد يكون عندما $\alpha=e$ أو $\alpha\Rightarrow^* e$. في هذه الحالة علينا توسيع A بـ α عندما يكون رمز الدخل هو عنصر من FOLLOW(A)، أو في حال كان وصلنا إلى $ في الدخل و $\in$ FOLLOW(A) $.

Algorithm. Construction of predictive parsing table.

Input. Grammar G.

Output. Parsing table M.

Method.

1. *For each production A→α of the grammar, do steps 2 and 3.*
2. *For each terminal a in FIRST(α), add A→α to M[A,].*
3. *If e in FIRST(α) add A→α to M[A,b] for each terminal b in FOLLOW(A). if e is in FIRST(α) and $ is in FOLLOW(A), add A→α to M[A,$].*
4. *Make each undefined entry of M be error.*

مثال

بتطبيق الخوارزمية السابقة على القواعد التالية:

```
E  →  TE'
E' →  +TE' | e
T  →  FT'
T' →  *FT' | e
F  →  (E) | id
```

لدينا FIRST(TE') = FIRST(T) = {(,id}، لذلك يتم إضافة القانون E→TE' إلى المركبة M[E,(] و المركبة E[E,id] في الجدول.

لدينا القانون E'→+TE' يسبب إضافة القانون هذا إلى المركبة M[E',+] في الجدول.

القانون E'→e يسبب إضافته إلى M[E',)] و M[E',$] لأن FOLLOW(E') = {),$}.

قواعد من النوع LL(1)

إن الخوارزمية السابقة من الممكن تطبيقها على أي قواعد G من أجل إنشاء الجدول M، ولكن و من أجل عدد من القواعد من الممكن أن تحوي بعض مركبات M على عدة عناصر بدلاً من عنصر واحد. ففي حال كانت G قواعد ذات تعاودية يسارية أو حاوية على غموض فإن الجدول سوف تحوي على مركبة واحدة على الأقل فيها أكثر من قانون.

مثال

بفرض لدينا القواعد التالية:

```
S  → iEtSS′ | a
S′ → eS | e
E  → b
```

يكون جدول التفكيك الخاص بها هو :

Nonterminal	*Input Symbol*					
	a	b	e	i	t	$
S	S → a			S → iEtSS′		
S′			S′ → eS′ S′ → eS			S′ → e
E		E → b				

الشكل (4–13)

و هنا يمكن القول بأن القواعد G و التي فيها جدول التفكيك لا يحوي على مركبات بقيم متعددة تدعى قواعد LL(1).

إن الـ L الأولى في الـ LL(1) تعني أن عملية مسح الدخل يبدأ من اليسار و باتجاه اليمين، و حرف الـ L الثاني يعني أن الاشتقاق المتكرر هو يساري و الرقم 1 يدل على أنه سيستخدم رمز واحد فقط من بداية السلسلة في اتخاذ القرار المناسب في اختيار الاشتقاق المناسب في كل مر يتم فيها توسيع رمز غير محدد الواقع في قمة المكدس أثناء عملية الإعراب.

من خصائص قواعد **LL(1)** أنها لا تحوي غموض أو تعاودية يسارية فيها و إذا كان لدينا $A \rightarrow \alpha|\beta$ فإنه :

1. لا يوجد رمز ما **a** يمكن أن تبدأ به سلاسل تشتق من α و β معاً.
2. إذا كان $\beta \Rightarrow^* e$ عندئذ فإن لا يمكن أن يكون هناك رمز ما **a** من الممكن أن يكون مشتركاً في بدايات السلاسل المشتقة من **FOLLOW(A)** مع السلاسل المشتقة من α.

بكلام آخر يمكننا القول بأنه لو كان لدينا $A \rightarrow \alpha|\beta$ فإنه:

$$FIRST(\alpha FOLLOW(A)) \cap FIRST(\beta FOLLOW(A)) = \varnothing$$

الإعراب من الأسفل إلى الأعلى ***Bottom-Up Parsing***

إن هذا النوع من التحليل التركيبي يدعى أيضاً بالإعراب القائم على الإزاحة–إرجاع و يدعى أيضاً بالإعراب **LR**، و يعتمد بشكل أساسي في بناء محللاته بشكل مؤتمت اعتماداً على أدوات برمجية خاصة بذلك.

إن الإعراب **LR** أو الإزاحة–إرجاع يحاول بناء شجرة الإعراب من أجل سلسلة دخل بدءاً من الأوراق (أدنى شجرة الإعراب) و باتجاه الجذر (أعلى الشجرة). و هذه العملية بمجملها تعتمد على إرجاع سلسلة الدخل إلى رمز البداية **S** للقواعد **G**.

في كل مرحلة و عند إيجاد الطرف اليميني لقانون ما من القواعد يتم استبدال السلسلة المكونة له بالرمز الواقع في الطرق اليساري من القانون. و ما يجب أن يؤخذ بعين الاعتبار هو اختيار السلسلة الجزئية المناسبة ليتم إرجاعها وفق قانون ما من القواعد **G**.

مثال

بفرض لدينا القواعد التالية :

```
S → aABf
A → Abc|b
B → d
```

و بفرض لدينا سلسلة الدخل ω=**abbcdf**.

من الممكن إرجاع هذه السلسلة إلى **S** وفق الخطوات التالية :

```
abbcdf
aAbcdf
aAdf
aABf
S
```

إننا نقوم بمسح السلسلة **abbcdf** من أجل إيجاد سلسلة جزئية منها تطابق الطرف اليميني لأحد القوانين. نجد أن الرمزان **b** و **d** تتحقق فيها الصفة السابقة، لنختار الرمز **b** الواقع في اليسار و لنستبدله بـ **A**، و ذلك بالاعتماد على الاشتقاق $A \rightarrow b$ و بهذا نحصل على السلسلة **aAbcdf**. نلاحظ أن السلاسل الجزئية **Abc** و

b و d توافق الطرف اليميني لبعض القوانين، ولكن لنختار السلسلة الجزئية Abc و نستبدلها بـ A و ذلك وفق القانون A→Abc و الآن و قد حصلنا على aAdf و باستبدال d بـ B وفق B→d نحصل على السلسلة aABf و باستبدال السلسلة الأخيرة وفق القانون الأول S→aABf نحصل على السلسلة S و بهذا قد تمكنا من إرجاع السلسلة abbcdf إلى رمز البداية S.

نلاحظ أن الإرجاعات السابقة تمت باستخدام قوانين، لو تم استخدامها و بترتيب معكوس، يمكن أن نحصل على نفس السلسلة و ذلك بالاشتقاق اعتباراً من S:

$$S \underset{rm}{\Rightarrow} aABf \underset{rm}{\Rightarrow} aAdf \underset{rm}{\Rightarrow} aAbcdf \underset{rm}{\Rightarrow} abbcdf$$

استخدام المكدس في إنجاز الإعراب إزاحة–إرجاع

إن الطريقة المناسبة لإنجاز التحليل إزاحة–إرجاع هي استخدام المكدس من أجل الاحتفاظ برموز القواعد و دارئ Buffer من أجل الاحتفاظ بسلسلة الدخل ω. سوف نستخدم الرمز $ للإشارة إلى قعر المكدس وكذلك إلى نهاية سلسلة الدخل في أقصى اليمين. في البداية يكون المكدس فارغاً و السلسلة ω كاملة في الدخل كما في الشكل :

Stack	Input
$	ω$

إن المعرب يعمل على إزاحة رمز دخل أو أكثر إلى المكدس إلى أن يتم الحصول على طرف يميني لقانون في قمة المكدس و بالتالي يقوم بإرجاع β إلى الرمز اليساري للقانون و يستمر المحلل بعمليات الإزاحة و الإرجاع إلى أن يكتشف خطأ ما أو أن يحصل في المكدس على رمز البداية S مع سلسلة دخل فارغة معلناً بذلك نجاح عملية الإعراب للسلسة ω.

Stack	Input
$S	$

مثال

لنطبق التحليل إزاحة–إرجاع على السلسلة $id_1+id_2*id_3$ و ذلك بالاعتماد على القواعد التالية:

```
E → E + E
E → E * E
E → ( E )
E → id
```

إن المراحل التحليل مبينة في الشكل 4–14.

نلاحظ بأنه توجد خيارات أخرى من أجل إجراء هذا الإعراب و ذلك في اختيار السلاسل الجزئية ليتم إرجاعها.

أثناء عملية التحليل إزاحة–إرجاع هذا توجد ثلاثة فعاليات يقوم بها المعرب هي :

1. عملية إزاحة **shift** : و فيها يتم أخذ الحرف التالي من سلسلة الدخل و دفعه إلى قمة المكدس.
2. عملية الإرجاع **reduce** : و فيه يتم سحب رموز الطرف اليميني لقانون من قمة المكدس و دفع الرمز الواقع في الطرف اليساري إلى قمة المكدس، حيث عليه أن يقرر متى و أي رموز يجب أن يسحبها ليرجعها.
3. القبول **accept** : و فيها يعلن المعرب عن نجاح عملية الإعراب للسلسلة.
4. الخطأ **error** :و فيها يحدد المعرب وجود خطأ تركيبي في الدخل حيث ينادي روتين معالجة الخطأ، أو يعلن عنه.

هناك أمر هام عند استخدام المكدس أثناء عملية التحليل التركيبي إزاحة–إرجاع، و هو أن السلسلة الرموز الجزئية التي يتم إرجاعها في كل مرة و الموجودة في المكدس، يجب أن تكون واقعة في قمة المكدس لا في باطنه مغطاة برموز أخرى.

Stack	*Input*	*Action*
$	$id_1+id_2*id_3$	Shift
$id_1	$+id_2*id_3$	Reduce by E → id
$E	$+id_2*id_3$	Shift
$E+	id_2*id_3	Shift
$E+id_2	$*id_3$	Reduce by E → id
$E+E	$*id_3$	Shift
$E+E*	id_3	Shift
$E+E*id_3	$	Reduce by E → id
$E+E*E	$	Reduce by E → E * E
$E+E	$	Reduce by E → E + E
$E	$	Accept

الشكل (**4–14**) مراحل تفكيك السلسلة $id_1+id_2*id_3$ وفق إزاحة–إرجاع

إن هذه العملية تصبح إلى حد ما غامضة عند الاهتمام بشكلين جمليين ناتجين عن مرحلتين متتاليتين في الاشتقاق اليميني الأقصى و هاتان الحالتان من الممكن أن تأخذان الشكلين التاليين :

1) $S \ *_{rm}\Rightarrow \ \alpha Az \ *_{rm}\Rightarrow \ \alpha\beta Byz \ *_{rm}\Rightarrow \ \alpha\beta\gamma yz$
2) $S \ *_{rm}\Rightarrow \ \alpha BxAz \ *_{rm}\Rightarrow \ \alpha Bxyz \ *_{rm}\Rightarrow \ \alpha\beta\gamma xyz$

في الحالة الأولى فإن **A** يجب أن تحل محل **βBy** و من ثم تحل **B** محل **γ**.

في الحالة الثانية فإن **A** يجب أن يتم إرجاعها أولاً و لكن في هذه الحالة بالسلسلة **y** المكونة من رمز محدد وحيد. و في المرحلة الثانية يتم إرجاعها إلى **B** و الذي سيكون موقعه على يسار **y**.

لنأخذ الحالة **1** و ذلك بفرض وصول المحلل إلى الحالة التالية:

Stack	Input
$αβγ	yz$

و الآن سيقوم المحلل بإرجاع السلسلة γ إلى **B**.

Stack	Input
$αβB	yz$

و بما أن **B** هو الرمز غير المحدد الواقع في أقصى اليمين من السلسلة αβByz، و مع الملاحظة أن السلسلة التي يجب أن ترجع لن تقع داخل المكدس، فإن المحلل سوف يقوم بإزاحة **y** إلى المكدس ليصل إلى الحالة :

Stack	Input
$αβBy	z$

لتصبح السلسلة βBy هي التي سترجع إلى **A**.

من أجل الحالة **2** و عندما يكون المحلل في الوضع التالي :

Stack	Input
$αγ	xyz$

نجد أن السلسلة المرجعة ستكون هي γ و بعد أن يتم إرجاعها إلى **B**، يقوم المحلل بإزاحة **xy** للحصول على السلسلة للإرجاع و هي **y** في قمة المكدس :

Stack	Input
$αBxy	z$

و الآن فإن المحلل سيرجع **y** إلى **A**.

نجد في الحالتين بأن المحلل يجب أن يجري إزاحة صفر مرة أو أكثر من أجل الحصول على السلسلة الواجب إرجاعها في قمة المكدس، و يجب أن تكون هذه السلسلة في قمة المكدس حصراً لا داخله، لأنه يتنافى مع مفهوم المكدس، و الذي هو بنية معطيات لا نرى منه سوى قمته.

البادئات القابلة للنمو ***Viable Prefixes***

و هي مجموعة البادئات في الأشكال الجملية اليسرى و التي من الممكن أن تظهر في قمة المكدس في

التحليل إزاحة–إرجاع تدعى بالبادئات القابلة للنمو.

أو بشكل مكافئ يمكننا القول بأنها تلك البوادئ من الشكل الجملي اليميني و الذي لا تنتهي بها أي طرف اليميني سلسلة قابلة للإرجاع في ذلك الشكل الجملي. و من هذا التعريف يمكننا القول بأنه دائماً من الممكن أن نضيف رمز محدد إلى نهاية البادئة القابلة للنمو من أجل الحصول على سلسلة جزئية من الممكن إرجاعها.

الصدامات في الإعراب إزاحة–إرجاع ***Conflicts During Shift-Reduce Parsing***

توجد قواعد خالية من السياق لا يمكن أن نستخدم معها الإعراب إزاحة–إرجاع. في مثل هذه القواعد و في حالات معينة يصل فيها المحلل إلى إمكانية إرجاع و إزاحة في الوقت نفسه و هذه الحالة تدعى بالصدام إزاحة–إرجاع، و فيها يكون لدى المحلل كافة المعلومات عن محتويات المكدس و معلومات عن الدخل القادم، و لا يمكنه اتخاذ القرار المناسب فيما سيقوم به. هناك أيضاً حالات حيرة يمكن أن يقع فيها المحلل بحيث لا يمكنه تحديد الإرجاع المناسب الذي يجب أن يجريه، و هذه الحالة من الصدامات تدعى صدام إرجاع–إرجاع، إن مثل هذه القواعد هي ليست قواعد LR(k) (non-LR(k) grammar)، حيث k هو عدد الرموز التي يجب أن يعرفها المحلل في بداية سلسلة الدخل من أجل اتخاذ القرارات المناسبة في عملياته.

إن القواعد التي تستخدم في المصنفات (المترجمات) عادة تندرج في مجموعة LR(1)، و التي تحتاج معرفة رمز واحد في بداية سلسلة الدخل.

مثال

إن القواعد الحاوية على الغموض لا يمكن أن تكون من النوع LR مثال ذلك القواعد التالية :

```
stmt → if expr then stmt
     | if expr then stmt else stmt
     | other
```

في حال كان لدينا الحالة التالية :

Stack	Input
$ … if expr then stmt	Else … $

هنا لدينا حالة صدام، حيث نجد أن if expr then stmt هو stmt و أيضاً يمكن إجراء الإزاحة لنحصل على if expr then stmt else stmt و الذي يمكن إرجاعه إلى stmt.

بالإعتماد على ما يلي else في الدخل/ من الممكن أن يكون صحيحاً بأن يرجع if expr then stmt إلى stmt أو من الممكن أن تجرى الإزاحة لـ else و البحث عن الـ stmt الخاصة بها من أجل الحصول على if expr then stmt else stmt، فهنا لا يمكننا اتخاذ القرار فيما إذا يجب إجراء الإزاحة أم إرجاع.

إن مثل هذه القواعد ليست **LR(1)**، و لكن و بشكل عام من الممكن أن تكون **LR(k)** من أجل قيم غير محددة لـ **k**، و بالتالي هي ليست **LR**.

في بعض الحالات يتم تعديل المحلل ليعمل على مثل هذه القواعد بأن نزيل الصدام من خلال إضافة شروط جديدة على عمله، كأن نجعله يفضل الإزاحة على الإرجاع، و في هذه الحالة سيعمل المحلل و من أجل نفس القواعد السابقة بشكل مألوف.

المعربات ***LR***

إن الإعراب **LR(k)** هو عبارة عن تقنية في آلية التحليل التركيبي من الأسفل إلى الأعلى **(bottom-up syntax analyzing technique)** و التي من الممكن استخدامها من أجل تنفيذ الإعراب لطيف واسع من القواعد الخالية من السياق **(context-free grammars)**. إن الـ **L** تعني أن اتجاه مسح الدخل هو من اليسار إلى اليمين، و الـ **R** تعني أنه يتم بناء الإشتقاقات اليمينية القصوى و لكن بشكل معكوس، و **k** هو عدد الرموز من سلسلة الدخل و التي ستكون رموز المنظورة أمامياً **(Lookahead symbols)** من أجل أن يتخذ المحلل التركيبي القرارات أثناء عمله. عند إهمال ذكر **k** فإن ذلك يعني أن قيمتها **1**.

و يمتاز الإعراب **LR** بما يلي :

- إن المعربات **LR** من الممكن نظرياً أن تُنشأ من أجل جميع لغات البرمجة و التي يمكن أن تكتب لها قواعد خالية من السياق.
- إن طريقة الإعراب **LR** هي الأعم من بين طرق الإعراب المعروفة المعتمدة على الإزاحة–إرجاع و التي لا تحتاج إلى العودة إلى الخلف أثناء إجراء التحليل.
- إن أصناف القواعد التي من الممكن أن تحلل من قبل المعرب **LR** هي مجموعة شاملة لتلك التي من الممكن أن تعرب من قبل محللات التوقعية الأخرى.
- من الممكن أن يكشف المعرب **LR** عن الخطأ التركيبي في الدخل في أقرب فرصة ممكنة عند إجراء المسح من اليسار إلى اليمين.

من سيئات الأساسية في معربات **LR** هو أنه من الصعب جداً بناء معرب **LR** بشكل يدوي من أجل لغة برمجة نموذجية لما يتطلب من جهد كبير. و بالتالي فهناك حاجة ماسة إلى وجود أدوات خاصة لذلك، و التي تدعى مولدات المعرب **LR (an LR parser generator)**.

إن مثل هذه الأدوات تأخذ القواعد الخالية من السياق لتنتج و بشكل أوتوماتيكي المحلل التركيبي لها. و إذا كانت القواعد حاوية على غموض أو تراكيب من الصعب إجراء الإعراب لها، فإن الأداة تحدد مكانها و تقرر عن وجودها للمصمم.

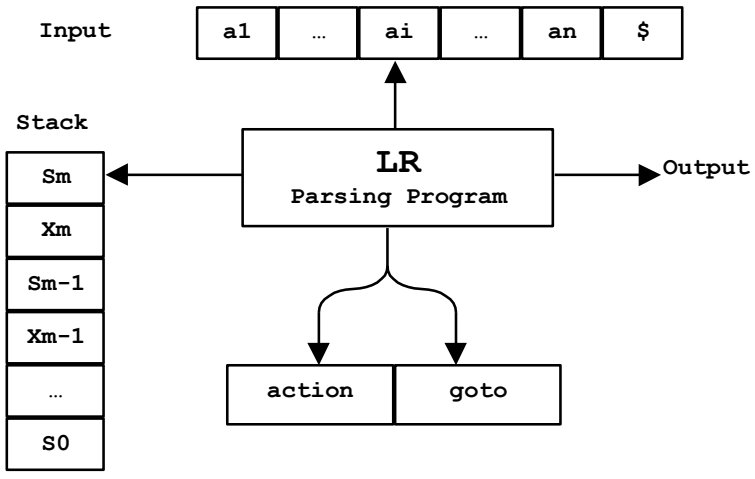
هناك ثلاث تقنيات لبناء جدول الإعراب LR من أجل قواعد ما، الطريقة الأولى تدعى الـ LR البسيطة (Simple LR or SLR)، وهي الأبسط تطبيقاً و لكنها الأضعف ضمن من بين الطرق الثلاث. الطريقة الأخرى هي الـ LR القانونية (Canonical LR) و هي الأقوى من بين الثلاثة و لكنها الأكثر تكلفة في التطبيق. الطريقة الثالثة هي متوسطة القوة و متوسطة الكلفة أيضاً و تدعى (Lookahead LR or LALR). إن الـ LALR تعمل مع معظم لغات البرمجة و مع قليل من الجهد يمكن جعلها لتعمل بشكل فعال.

خوارزمية الـ LR في الإعراب *LR Parsing Algorithm*

إن مخطط المحلل التركيبي LR مبين في الشكل 4-15.

يتألف المحلل التركيبي من الدخل، الخرج، المكدس، برنامج القيادة، و جدول الإعراب الذي يتكون من قسمين جدول الفعالية (Action Table) و جدول الإزاحة أو الانتقال (Goto Table). إن برنامج القيادة هو نفسه من أجل جميع المحللات LR، فقط جدول التحليل هو الذي يتغير من معرب إلى آخر.

يقوم البرنامج بقراءة الدخل من دارئ الدخل رمز تلو الآخر في كل مرة، و يستخدم المكدس لحفظ سلسلة من الشكل $s_0X_1s_2X_2\ldots X_ms_m$، حيث s_m تقع في القمة، إن X_i هو رمز قواعدي و s_i هو رمز يعبر عن الحالة يدعى حالة المحلل. إن كل رمز حالة الواقع في قمة المكدس يلخص المعلومات المحتواة ضمن المكدس. إن رمز الحالة في قمة المكدس، مع رمز الدخل الحالي في مقدمة سلسلة الدخل يستخدمان معاً في فهرسة جدول التحليل من أجل اتخاذ القرار في فعاليات الإزاحة-إرجاع. إن رموز القواعد لا تظهر في قمة المكدس مطلقاً و إنما الغاية منها هو الإيضاح فقط لعملية التحليل التركيبي LR.



الشكل (4-15) المحلل التركيبي LR

كما و ذكرنا فإن جدول التحليل يتكون من قسمين و الذان يعبران عن تابعان هام تابع الإزاحة goto و تابع

الفعالية **action**. إن برنامج القيادة يقوم بتحديد رمز الحالة من قمة المكدس s_m و رمز الدخل الحالي a_i و تستخدمهما في حساب **action**$[s_m,a_i]$ و التي يمكن أن تأخذ إحدى القيم الأربعة التالية :

1. الإزاحة إلى الحالة الجديدة **s**.
2. الإرجاع وفق الاشتقاق $A \rightarrow \beta$.
3. القبول.
4. الخطأ.

التابع **goto** يأخذ كبارامترات له من رمز القواعد، و رمز الحالة ليرجع الحالة الجديدة. إن تنظيم المحلل **LR** يتكون من جزأين هما المكدس و سلسلة الدخل غير الممسوحة بعد، و بذلك يتكون لدينا ثنائية لها الشكل التالي :

$$(s_0X_1s_1X_2s_2\dots X_ms_m,\ a_ia_{i+1}\dots a_n\$)$$

و هذه الثنائية تعبر عن الشكل الجملي اليميني و الذي يظهر بالشكل التالي :

$$X_1X_2\dots X_ma_ia_{i+1}\dots a_n$$

كما و ذكرنا أن الفعالية التالية التي سيقوم بها المحلل التركيبي تتحدد من قراءة الرمز a_i و أخذ s_m، رمز الحالة من قمة المكدس، و بالتالي جلب المركبة **action**$[s_m,\ s_i]$، و بالتالي تحديد الفعالية و تنفيذها و التي يمكن أن تكون آخذة إحدى الحالات التالية :

1. إذا كان **action**$[s_m,\ a_i]$ = **shift s**، يقوم المحلل بتنفيذ عملية الإزاحة ليصل إلى الحالة التالية :

$$(s_0X_1s_1X_1\dots X_ms_ma_is,\ a_{i+1}\dots a_n\$)$$

هنا قد قام المحلل بإزاحة الرمز الدخل الحالي a_i و الحالة الجديدة **s** و التي تتحدد من **action**$[s_m,a_i]$ إلى المكدس ليصبح a_{i+1} هو رمز الدخل الجديد.

2. إذا كانت **action**$[sm,\ ai]$ = **reduce** $A \rightarrow \beta$، عندئذ يقوم المحلل بتنفيذ عملية إرجاع لتصبح الحالة الجديدة كما يلي :

$$(s_0X_1s_1X_2s_2\dots X_{m-r}s_{m-r}As,\ a_ia_{i+1}\dots a_n\$)$$

حيث **s = goto**$[s_{m-r},\ A]$، و **r** طول السلسلة β و هي الطرف اليميني من القانون. في هذه العملية يقوم المعرب بسحب **2r** عنصر من المكدس (**r** رمز حالة و **r** رمز قواعد) لتظهر لدينا الحالة s_{m-r} على قمة المكدس، من ثم يقوم المعرب بدفع الرمز **A** والذي هو الطرف اليساري للقانون المرجع وفقه إلى المكدس و يقوم بدفع رمز الحالة الجديدة **s** و التي هي المركبة **goto**$[s_{m-r},\ A]$. إن رمز الدخل الحالي لم يتغير خلال

عملية الإرجاع هذه. إن مجموعة الرموز القواعد $X_{m-r+1}...X_m$ و المسحوبة من المكدس تطابق دوماً السلسلة β و التي هي الطرف اليميني من القانون المرجع وفقه.

إن خرج المحلل التركيبي **LR** يتولد أثناء عمليات الإرجاع التي ينفذها و ذلك من خلال تنفيذ فعاليات المعاني الموافقة لكل قانون يتم الإرجاع بالنسبة له.

3. في حال كانت **action[sm, ai] = accept**، فإن عملية الإعراب تتوقف.
4. في حال كانت **action[sm, ai] = error**، يكون المحلل قد كشف خطأ قواعدي في سلسلة الدخل، و عادة ما يستدعي روتين تنفيذ التصحيح في هذه الحالة.

ففيما يلي خوارزمية التحليل التركيبي **LR**، و المستخدمة في جميع المحللات **LR**. و الإختلاف الوحيد بين المحللات هو المعلومات الموجودة في جدول الإعراب.

Algorithm. LR Parsing algorithm.

Input. An input string ω and LR parsing table with functions action and goto fror a grammar G.

Output. If ω is in L(G), a bottom-up parse for ω; otherwise an error indiccation.

Method.
Initially, the parser has s0 on its stack, where s0 is the initial state, and ω$ in the input buffer. The parser the executes the program that apears in the next figure until an accept or error action is encountered.

```
Set ip to point to the first symbol of ω$
Repeat forever begin
       Let s be the state on top of the stack  and a the symbol pointed
to by ip
       If action[s, a] = shift s' then begin
              Push a then s' on top of the stack;
              Advance ip to the next input symbol
       End
       Else if action[s, a] = reduce A→β then begin
              Pop 2*|β| symbols off the stack;
              Let s' be the state now on top of the stack;
              Push A then goto[s', A] on top of the stack;
              Output the production A→β
       End
       Else if action[s, a] = accept then
              Return
       Else error()
End
```

مثال

فيما يلي شكل **4–16** يبين جدول الإزاحة و جدول الفعالية الخاصان بالقواعد الخاصة بالتعبير

الرياضي التالية :

```
(1) E → E+T
(2) E → T
(3) T → T*F
(4) T → F
(5) F → (E)
(6) F → id
```

1. **s*i*** تعني إزاحة إلى الحالة ***i***.
2. **r*j*** تعني الإرجاع وفق القانون ***j***.
3. **acc** تعني القبول.
4. الفراغ يعني خطأ.

لاحظ أن فهرسة الأعمدة الخاصة بجدول الفعالية يتم من خلال رموز محددة، أما في جدول الإزاحة فيتم من خلال الرموز غير المحددة.

و فيما يلي شكل **4–17** يبين تفكيك السلسلة **id * id + id**.

state	*action*						*Goto*		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

الشكل (**4–16**) جدول الإعراب **LR** الخاص بقواعد التعبير الرياضي

بناء جدول الإعراب لمعربات ***SLR***

إن الفكرة الأساسية لبناء جدول التفكيك لـ SLR هو إيجاد أوتومات يتم من خلاله تمييز البادئات القابلة للتوسيع. إن بناء هذا الأوتومات يعتمد على إيجاد ما يسمى مجموعة المجموعات الخاصة بالقواعد، و بالاعتماد على مجموعة المجموعات هذه يتم ملئ جدول الإزاحة و جدول الفعالية الخاصة بالقواعد.

ما يجب التنويه إليه بأنه يجب توسيع القواعد G ذات العنصر البدائي S إلى قواعد G' ذات العنصر البدائي S' و ذلك بإضافة القانون S' → S إليها. إن الغاية من هذا التوسيع هو إيجاد قواعد يمكن بناء جدول الإعراب لها بحالة واحدة للقبول، تلك الحالة التي يتم فيها تنفيذ الإرجاع S'→S، و ذلك لضمان تنفيذ الإرجاع من أجل القوانين التي يقع فيها S في الطرف اليساري.

stack	input	Action
0	id*id+id$	Shift
0 id 5	*id+id$	reduce by F → id
0 F 3	*id+id$	reduce by T → F
0 T 2	*id+id$	Shift
0 T 2 * 7	id+id$	Shift
0 T 2 * 7 id 5	+id$	reduce by F → id
0 T 2 * 7 F 10	+id$	reduce by T → T*F
0 T 2	+id$	reduce by E → T
0 E 1	+id$	shift
0 E 1 + 6	id$	shift
0 E 1 + 6 id 5	$	reduce by F → id
0 E 1 + 6 F 3	$	reduce by T → F
0 E 1 + 6 T 9	$	reduce by E → E+T
0 E 1	$	Accept

الشكل (4–17) خطوات تحليل السلسلة id*id+id وفق LR

بناء مجموعة المجموعات من أجل SLR

بفرض أنه لدينا القواعد الموسعة عن القواعد الخاصة بالتعبير الرياضي و التي لها الشكل التالي :

```
E' → E
E  → E + T
E  → T
T  → T * F
T  → F
F  → (E)
F  → id
```

من أجل إيجاد مجموعة المجموعات لهذه القواعد نتبع ما يلي :

الخطوة 1 :يتم أخذ القانون الأول E'→E و بوضع نقطة قبل E (وحيث هذه النقطة تعبر عن المرحلة التي وصل إليها الأوتومات من التحليل بالنسبة للقانون E'→E). و بذلك يتكون لدينا ما يسمى بنواة المجموعة الأولى من مجموعة المجموعات، و من ثم يتم توسيع هذه المجموعة اعتباراً من النواة بإضافة عناصر جديدة من قوانين القواعد و ذلك وفق الطريقة التالية :

من أجل أي قانون من الشكل A → α.Bβ و الموجود في المجموعة يتم إضافة جميع القوانين من الشكل B → γ مع إضافة النقطة إلى الطرف اليميني من القانون و ذلك في بداية السلسلة γ، ليكون له الشكل التالي B → .γ. تستمر هذه العملية إلى أن نحصل على مجموعة لا يمكن إضافة عناصر جديدة.

مثال

لنأخذ القانون الأول للقواعد السابقة **E → E'** و نعتمد عليه في إيجاد النواة و توسيعها لنحصل على المجموعة الأولى من مجموعة المجموعات :

```
{     E'→ .E                    ← نواة المجموعة الأولى
      E → .E+T
      E → .T
      T → .F
      F → .(E)
      F → .id     }
```

الخطوة **2** : من أجل إيجاد المجموعات التالية يتم النظر إلى كل مجموعة موجودة لدينا و أخذ العناصر التي تقع فيها النقطة قبل نفس الرمز و التي لها الشكل **A → α.Xβ** ليتكون و بإزاحة النقطة في هذه العناصر إلى الرمز التالي في هذه العناصر لتكون لها الشكل التالي **A → αX.β** لينتج لدينا نوى لمجموعات جديد. و نوسع المجموعات الجديدة كما هو الحال في الخطوة **1** من اجل المجموعات المكونة. و تستمر هذه العملية إلى أن نعجز عن إضافة مجموعات جديدة.

من أجل القواعد السابقة تكون مجموعة المجموعات الخاصة بها لها الشكل التالي :

```
I     E'→ .E
0     E → .E+T
:     E → .T
      E → .T*F
      T → .F
      F → .(E)
      F → id

      E'→ E.
      E → E.+T
I
1
:     E → T.
      T → T.*F

I     T → F.
2
:     F → (.E)
      E → .E+T
      E → .T
I     T → .T*F
3     F → .(E)
:     F → .id

I
4
:
```

```
I     F → id.
5
:
      E → E+.T
      T → .T*F
I     T → .F
6     F → .(E)
:     F → .id

      T → T*.F
      F → .(E)
      F → .id
I
7     F → (E.)
:     E → E.+T

      E → E+T.
      T → T.*F
I
8     T → T*F.
:

      F → (E).
I
9
:


I
1
0
:

I
1
1
```

:

الشكل (4–18) مجموعة المجموعات الـ LR(0) القانونية من أجل قواعد التعبير الرياضي

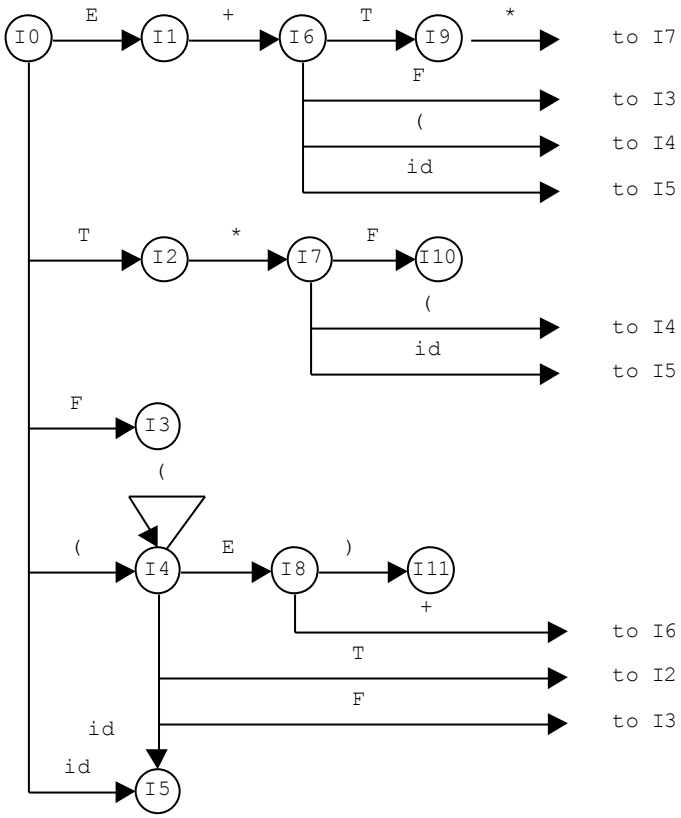
إن كل مجموعة من مجموعة المجموعات السابقة تعبر عن حالة من حالات أوتومات الترجمة LR(0).

إن الحالة التالية من أجل أي حالة من الحالات (مجموعة) و من أجل رمز ما هي تلك التي نواة مجموعتها تنتج عن هذه المجموعة بإزاحة النقطة عبر هذا الرمز.

أي أن الحالات التالية للحالة الأولى I0 من أجل الرمز E مثلاً هي I1 من أجل الرمز E، I2 من أجل الرمز T، I3 من أجل الرمز F، I4 من أجل الرمز (، I5 من أجل الرمز id. الحالات التالية للحالة I7 هي الحالة I10 من أجل الرمز F، و الحالة I4 من أجل الرمز I4، الحالة I5 من أجل الرمز id. و هكذا ...

الحالات التي تقع نقطة في عناصرها في أقصى اليمين من الطرف اليميني من القانون هي حالات إرجاع لتلك القوانين و ذلك من أجل أي رمز غير رمز إزاحة.

فيما يلي الشكل 4–19 يبين مخطط الانتقال الخاص بالأوتومات الذي تعبر عنه مجموعة المجموعات السابقة.



الشكل (4–19): مخطط انتقال في الأوتومات المحدد (DFA D) من أجل البادئات القابلة للتوسيع

فيما يلي خوارزمية تشكيل جدول الإعراب الخاص بالإعراب **SLR** :

Algorithm. Construction an SLR parsing table

Input. An augmented grammar G′.

Output. The SLR parsing table functions *action* and *goto* for G′.

Method.

1. Construct C ={I0,I1,…,In}, The collection of sets of LR(0) items for G′.
2. State i is constructed from Ii. the parsing actions for state i are determined as follows:

a) If [A → α.aβ] is in Ii and goto(Ii,a) = Ij, then set action[i,a] to "shift j." Heare a must be a terminal

b) If [A → α.] is in Ii, then set action[i,a] to "reduce A → α" for all a in FOLLOW(A); here A may not be S′.

c) If [S′→S.] is in Ii, then set action[i,$] to "accept."

If any conflicting actions are generated by the above rules, we say the grammar is not SLR(1). The algorithm fails to produce a parser in this case.

3. The goto transitions for state I are constructed for all nonterminals A using the rule : If goto(Ii, A) = Ij, then goto[i,A] = j.
4. All entries not defined by rules (2) and (3) are made "error.".
5. The initial state of the parser is the one constructed from the set of items containing [S′→.S].

بناء جدول الإعراب للمعربات ***LR*** *القانونية*

بالرجوع إلى طريقة إنشاء الجدول في النوع **SLR**، نجد أننا عندما عالجنا حالات الإرجاع وفق **A→α** في حال وجد العنصر **[A→α.]** ضمن المجموعة الموافقة للحالة، قد أخذنا رموز الدخل **a** من **FOLLOW(A)** هي التي عندها يحصل الإرجاع. في بعض الحالات عندما تكون الحالة **i** في قمة المكدس، فإن البادئة القابلة للتوسيع **βα** و الموجودة في المكدس و التي ستأخذ الشكل **βA** في الحقيقة قد لا يمكن أن تكون متبوعة بـ **a** في الشكل الجملي اليميني للقواعد **G**. و بالتالي فإن الإرجاع **A→α** قد لا يكون صحيحاً في هذه الحالة.

من الممكن أن نُحمل رمز الحالة المزيد من المعلومات و التي تسمح لنا بالتخلص من العناصر **[A→α.]** في بعض الحالات و التي تتسبب في إرجاعات خاطئة. إن ذلك يتم من خلال تقسيم الحالات لكي نصل إلى تلك الحالات الخاصة و التي تكون السلسلة **α** فيها من الممكن إرجاعها إلى **A**.

إن المعلومات الإضافية التي يمكن أن نمنحها للحالة يتم من خلال إعادة تعريف عناصر المجموعات و ذلك بإضافة رمز محدد إليها ليكون المركبة الأخرى في عنصر المجموعة. الشكل العام للعنصر سيصبح **[A → α.β, a]**، حيث **A → αβ** هو قانون و **a** هو رمز محدد أو قد يكون رمز نهاية الدخل **$**. إن مثل هذا العنصر يدعى عنصر **LR(1)**. إن **1** تشير إلى طول المركبة الجديدة في العنصر و هذه المركبة تدعى بـ **lookahead of the item**. إن المركبة الجديدة لا تؤثر على العنصر **[A → α.β, a]** عندما لا تكون **β** هي **e**. و لكن في حال كان

لدينا المركبة [A → α., a] و التي تعني الإرجاع وفق A→α فقط عندما يكون رمز الدخل التالي هو a. و بهذا نكون قد حصرنا الحالات التي يجب أن يتم الإرجاع وفق A →α فقط من أجل الحالات التي تكون في قمة المكدس و الحاوية على عناصر الـ LR(1) من الشكل [A→α,a] و ذلك من أجل رمز الدخل الحالي a. في الحالة العامة فإن مجموعة الرموز a هي مجموعة جزئية من FOLLOW(A).

إن طريق استخراج مجموعة المجموعات من النوع LR(1) مشابهة لتلك الخاصة بـ LR(0) مع بعض التعديلات البسيطة، و الخوارزمية الخاصة بذلك موضحة فيما يلي:

Algorithm. Construction of the sets of LR(1) items.

Input. An augmented grammar G′.

Output. The sets of LR(1) items that are the set of items valid for one or more viable prefixes of G′.

Method. The procedures *closure* and *goto* and the main routine *items* for constructing the sets of items are shown in the following figure.

```
function closure(I);
begin
      repeat
      for each item [A→α.Bβ,a] in I,
            each production B →γ in G′,
            and each terminal b in FIRST(βa)
            such that [B→γ,b] is not in I do
                  add[B→γ,b] to I;
      until no more items can be added to I
      return I
end;

function goto(I,X)
begin
      let J the first set of items [A→αX.β,a] such that
            [A→α.Xβ,a] is in I;
      return closure(J)
end;

procedure items(G′)
begin
      C := {closure({[S′→.S,$]})};
      repeat
      for each set of items I in C and each grammar symbol X
            such that goto(I,X) is not empty and not in C do
                  add goto(I,X) to C
      until no more sets of Items can be added to C
end
```

مثال

بفرض لدينا القواعد الموسعة التالية :

```
S′→ S
S → CC
C → cC | d
```

فيما يلي الشكل **4–20** يبين مجموعة المجموعات **LR(1)** لهذه القواعد.

```
I0:
    S′→.S,$
    S →.CC,$
    C →.cC,c/d
    C →.d,c/d

I1:
    S′→S.,$

I2:
    S →C.C,$
    C →.cC, $
    C →.d,$

I3:
    C → c.C, c/d
    C → .cC, c/d
    C → .d, c/d

I4:
    C → d., c/d

I5:
    S → CC., $

I6:
    C → c.C, $
    C → .cC, $
    C → .d, $

I7:
    C → d., $

I8:
    C → cC., c/d

I9:
    C → cC., $
```

الشكل (4–20) مجموعة المجموعات LR(1)

فيما يلي خوارزمية إنشاء جدول الإعراب الخاص بـ **LR(1)**، إن الاختلاف في طريقة إنشاء جدولي **action** و **goto** عن الحالة السابقة يكمن فقط في طريقة ملئ مركبات الجدولين.

Algorithm. Construction of the canonical LR parsing table.

Input. An augmented grammar G′.

Output. The canonical LR parsing table functions *action* and *goto* for G′.

Method.

1. Construct C ={I0,I1,…,In}, the collection of sets of LR(1) items for G′.
2. State i is constructed from Ii. The parsing actions for state i are determined as follows:

a) If [A → α.aβ,b] is in Ii and goto(Ii,a) = Ij, then set action[i,a] to "shift j." Heare, a is required to be a terminal.

b) If [A → α.,a] is in Ii, A ≠ S′ then set action[i,a] to "reduce A→α.".

c) If [S′→S.,$] is in Ii, then set action[i,$] to "accept."

If conflict results from above rules, the grammar is said not to be LR(1), and the algorithm is said to fail.

3. The goto transitions for state i are determined as follows: If

goto(Ii,A)=Ij, then goto[i,A]=j.

4. All entries not defined by rules (2) and (3) are made "error.".

5. The initial state of the parser is the one constructed from the set of items containing [S'→.S,$].

فيما يلي الشكل **4–21** يمثل جدول الإعراب القانوني للقواعد السابقة.

بناء جدول الإعراب ***LALR***

إن هذه الطريقة هي الأكثر شيوعاً في بناء جداول الإعراب، لأنها أصغر إلى حد كبير مما هو الحال بالنسبة لـ **canonical LR**، كما و أن معظم لغات البرمجة من الممكن تمثيلها من خلال قواعد **LALR**.

بمقارنة أحجم الجداول في كل من **canonical LR** و **LALR** نجد أنه من أجل لغة مثل الـ **PASCAL** يمكن أن تتولد عدة مئات من الحالات من أجل **LALR**، بالمقابل يمكن أن تصل في حال استخدام **canonical LR** إلى عدة آلاف من أجل نفس اللغة.

state	*action*			*goto*	
	c	d	$	S	C
0	s3	s4		1	2
1			ac c		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

الشكل (4–21) جدول التفكيك LR(1)

إن الاختلاف الوحيد في بناء جداول الإعراب LALR عما هو الحال في canonical LR يكمن في بناء مجموعة المجموعات فقط و التي من الممكن إن تنتج عن مجموعة المجموعات القانونية بإجراء عمليات الدمج في النوى بعض مجموعاتها. و فيما يلي الخوارزمية الخاصة بذلك :

Algorithm. An easy, but space-consuming LALR table construction.

Input. An augmented grammar G′.

Output. The LALR parsing table functions *action* and *goto* for G′.

Method.

1. Construct C ={I0,I1,…,In}, the collection of sets of LR(1) items for G′.
2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.
3. Let C′ = {J0, J1,…, Jm} be the resulting sets of LR(1) items. The parsing actions for state i are constructed from Ji in the same manner as in algorithm of LR(1). If there a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).
4. The goto table is cunstructed as follows. if J is the union of one or more sets or LR(1) items, that is, J=I1∪I2∪…∪Ik, then the cores of goto (I1,X), goto(I2,X),…,goto(Ik,X) are the same , since I1,I2,…,Ik all have the same core. let K be the union of all sets of items having the same cores as goto(I1,X). Then goto(J,X)=K.

مثال

بأخذ القواعد في المثال السابق، و أخذ مجموعة المجموعات الخاصة بها، نجد أنه هناك ثلاثة أزواج من المجموعات و التي من الممكن أن يتم دمجها. يمكن دمج I3 مع I6 و استعاضتها بمجموعة أخرى نامجمة

عن الاتحاد و هي :

```
I36:  C → c.C, c/d/$
      C → .cC, c/d/$
      C → .d, c/d/$
```

و بدمج I4 مع I7 ينتج لدينا :

```
I47:  C → d., c/d/$
```

و بدمج I8 مع I9 ينتج :

```
I89:  C → cC., c/d/$
```

و فيما يلي الشكل 4–22، يبين جدول الإعراب LALR من أجل تلك القواعد.

state	Action			goto	
	c	d	$	S	C
0	s3 6	s4 7		1	2
1			ac c		
2	s3 6	s4 7			5
36	s3 6	s4 7			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		9

الشكل (4–22) جدول التفكيك LALR

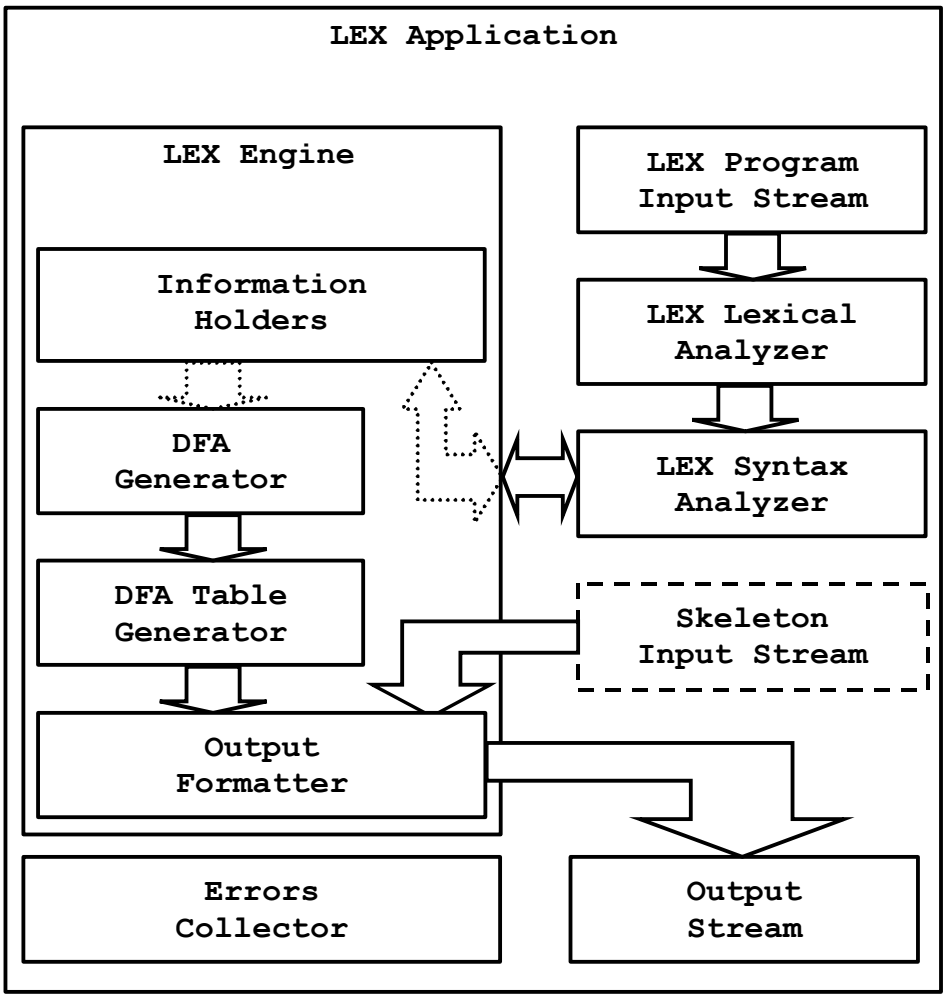
الفصل الخامس

مولد المحلل اللفظي

Lexical Analyzer Generator

بنية البرنامج LEX

فيما يلي الشكل العام لأجزاء البرنامج LEX المنفذ



الشكل (5-1) المخطط العام لعناصر البرنامج LEX

يتكون البرنامج بشكل عام من عدة أجزاء وظيفية و هي :

- دفق الدخل للبرنامج (LEX Program Input Stream) : وهو الكائن الذي يعبر عن ملف الدخل الحاوي على برنامج الـ LEX.

- **المحلل اللفظي (Syntax Analyzer) : و مهمته إنجاز التحليل اللفظي لبرنامج المصدر.**
- **المحلل التركيبي (Syntax Analyzer) : و مهمته إنجاز التحليل التركيبي لبرنامج المصدر، و هذا الجزء هو المسؤول عن استخلاص كافة المعلومات و التصريحات و تحليل التعابير النظامية التي تصف البنية اللفظية للغة المراد إنشاء المحلل اللفظي لها، و يقوم المحلل هذا بإرسال كافة المعلومات التي استخلصها من ملف إلى جزئ قاعدة البيانات من الأداة.**
- **دفق الدخل للهيكل (Skeleton Input stream): و هو كائن الذي يعبر عن ملف الدخل الحاوي على هيكل النتيجة.**
- **دفق الخرج (Output Stream) : و هو الكائن المعبر عن خرج النهائي للبرنامج LEX.**
- **محرك الـ LEX (LEX Engine) : و هو الجزء الأساسي من البرنامج، و فيه يتم معالجة كافة المعلومات المستخلصة من الدخل و توليد الـ DFA الخاص بالمحلل الفظي، و من ثم تشكيل الخرج النهائي ليتم إرساله إلى دفق الخرج. و يتكون كائن قاعدة البيانات من :**

(a) **كائنات مسك المعلومات (Information Holders) : و هي عبارة عن مجموعة جداول تحوي كافة المعلومات من أشجار التعابير النظامية التي بناها المحلل التركيبي و مختلف التصريحات التي استخلصها من الدخل.**

(b) **مولد أوتومات المحلل اللفظي (DFA Generator) : و هو الكائن المسؤول عن بناء الأوتومات غير المحدد NFA اعتباراً من أشجار التعابير النظامية و تحويله إلى أوتومات المحدد DFA، و إجراء عمليات الأمثلة عليه من لتوليد الـ DFA النهائي.**

(c) **مولد جدول الـ DFA (DFA Table Generator) : يقوم هذا الكائن بتحويل التمثيل الشبكي للأوتومات النهائي إلى شكل جدولي ليتم فيما بعد توليد الخرج بالاعتماد عليه.**

(d) **مشكل الخرج (Output Formatter) : حيث يقوم هذا الكائن بتشكيل الخرج النهائي تحضيراً لإرساله إلى دفق الخرج.**

- **جامع الأخطاء (Error Collector) : و مهمته تجميع كافة رسائل الأخطاء المتولدة في مختلف مراحل العمل، ليتم في النهاية إظهارها للمستخدم.**

ما هو جدير بالذكر بأن جداول القيادة الخاصة بالمحلل اللفظي و المحلل التركيبي قد تم إنشاءها باستخدام نفس أجزاء البرنامج الخاص بالأدوات المنفذة في المشروع و ذلك خلال مختلف مراحل إنجاز.

إن الخوارزميات المستخدمة في عمل مختلف أجزاء البرنامج مذكورة في الفصلين الثاني و الثالث، بما فيها الخوارزميات المتعلقة بعمل المحلل اللفظي و التركيبي LR.

يتألف البرنامج LEX من ثلاثة أجزاء، هما قسم التصريحات، و قوانين الترجمة، وقسم هيكل المحلل اللفظي، و الهيكل العام له هو كما يلي :

```
declarations

%%

translation rules

%%
```

نجد أن الرمز %% يفصل ينهي قسما البرنامج.

في قسم التصريحات يتم ذكر التعريفات النظامية، و ذلك ليتم استخدامها في قسم قواعد الترجمة. و الشكل العام للتعريف التعبير النظامي يأخذ الشكل التالي :

```
regular expression name = regular expression;
```

يجب أن لا يحتوي الاسم على فراغات، كما و يتم تمييز بين الأحرف الكبيرة و الصغيرة.

و فيما يلي بعض أمثلة عن بعض التعريفات:

```
PERCENT    = \%;
LETTER     = [\A-\Z,\_,\a-\z];
DIGIT      = [\0-\9];
IDENTIFIER = LETTER(LETTER|DIGIT)*;
COMMAND    = PERCENT IDENTIFIER;
INTEGER    = (\-|\+)?DIGIT+;
CAPITAL    = [65-90];
IF         = "if";
THEN       = "then";
COMMENT    = "/*"([1-\),\+-255]|(\*[1-\.,\0-255]))*"*/" ;
BLANK      = [1-32]+;
```

إن الشكل العام للتعبير النظامي قد تم ذكره في الفصل الثاني، و لكن توجد بعض الاختلافات فعند التعبير عن رمز محرفي، يجب أن يسبق الحرف في التعبير بـ \ ، كما عند التعبير عن مجموعة من المحارف يتم الفصل بين المجالات و الرموز المختلفة في المجموعة بفواصل. كما هو الحال في التصريح الثاني عن **LETTER**. كما و يمكن التعبير عن المحرف عن طريق رقم يمثل رمزه الـ **ASCII**، كما في التصريح عن **CAPITAL**. في التعبير عن سلسلة محارف، يجب أن تحاط بـ "".

في قسم قواعد الترجمة يتم ذكر النماذج النصية، يأخذ الأمر الشكل التالي :

```
regular expression : #{ action #}
```

إن التعبير النظامي في قاعدة الترجمة يصف النموذج **(pattern)** الذي عنده ستنفذ الفعالية **(action)** عندما يصادفه المحلل اللفظي.

و فيما يلي أمثلة عن بعض قواعد الترجمة :

```
IDENTIFIER          : #{ IDENTIFIER acttion code … #}
BLANK               : #{ BLANK acttion code … #}
```

```
COMMENT             : #{ COMMENT acttion code … #}
COMMAND             : #{ COMMAND acttion code … #}
\,                  : #{ COMA action code … #}
\|                  : #{ BAR action code … #}
\;                  : #{ SEMICOLON action code … #}
\:                  : #{ COLON action code … #}
```

كما و من الممكن ذكر الملاحظات و ذلك بوضعها ضمن /* */.

فيما يلي برنامجان لـ **LEX** الأول يصف العناصر النصية الخاصة بالمحلل اللفظي لـ **YACC**، و الثاني يصف العناصر النصية الخاصة بالمحلل اللفظي لـ **LEX**.

```
/*****************************************************
*                                                    *
*   Lexical analyzer for YACC tool definition file   *
*   Date: 25-6-99                                    *
*                                                    *
*****************************************************/

/* Definitions */
LETTER     = [\A-\Z,\_,\a-\z];
DIGIT      = [\0-\9];
IDENTIFIER = LETTER(LETTER|DIGIT)*;
COMMAND    = \%IDENTIFIER;
/* Macros body lexeme define */
BODY       = "#{"([1-\",\$-255]|(\#[1-\|,\~-255]))*"#}" ;
/* Macros comments lexeme define */
COMMENT    = "/*"([1-\),\+-255]|(\*[1-\.,\0-255]))*"*/" ;
/* String constatnt lexeme */
STRING     = \"[1-\!,\#-255]*\";
/* Skipped characters */
BLANK      = [1-32]+;
%%
/* Lexical patterns */
IDENTIFIER          : #{… #}
BLANK               : #{… #}
COMMENT             : #{… #}
BODY                : #{… #}
COMMAND             : #{… #}
\,                  : #{… #}
\|                  : #{… #}
\;                  : #{… #}
\:                  : #{… #}
"%%"                : #{… #}

%%
```

```
/*****************************************************
*                                                    *
*   Lexical analyzer for LEX tool definition file    *
*   Date: 25-6-99                                    *
*                                                    *
*****************************************************/

/* Definitions */

LETTER     = [\A-\Z,\_,\a-\z];
DIGIT      = [\0-\9];
INTEGER    = DIGIT+;
CHAR       = (INTEGER)|(\\[32-127]);
```

```
ID          = LETTER(LETTER|DIGIT)*;
BODY        = "#{"([1-\",\$-255]|(\#[1-\|,\~-255]))*"#}" ; /* Macros
body lexeme define */
COMMENT     = "/*"([1-\),\+-255]|(\*[1-\.,\0-255]))*"*/" ; /* Macros
comments lexeme define */
STRING      = \"[1-\!,\#-255]*\";                          /* String
constatnt lexeme */
BLANK       = [1-32]+;                                     /*
Skipped characters */

%%

/* Lexical patterns */

CHAR                            : #{LEX_Char#}
ID                              : #{LEX_Identifier#}
BODY                            : #{LEX_Body#}
STRING                          : #{LEX_String#}
BLANK                           : #{LEX_Skip#}
COMMENT                         : #{LEX_Comment#}
\:                              : #{LEX_Colen#}
\;                              : #{LEX_Semi#}
\,                              : #{LEX_Coma#}
\(                              : #{LEX_LeftB#}
\)                              : #{LEX_RightB#}
\[                              : #{LEX_LeftMB#}
\]                              : #{LEX_RightMB#}
\?                              : #{LEX_QMark#}
\|                              : #{LEX_Bar#}
\*                              : #{LEX_Start#}
\+                              : #{LEX_Plus#}
\-                              : #{LEX_Minus#}
\=                              : #{LEX_Equal#}
"%%"                            : #{LEX_DPercent#}

%%
```

بالنسبة للخرج، من الممكن الحصول على محلل لفظي كامل من خلال تقديم ما يسمى هيكل المحلل الفظي **(lexical analyzer skeleton)**، و هو عبارة عن شيفرة مصدر، للمحلل اللفظي بشكله العام، و تقوم الأداة بإضافة الجداول المكملة له، بالإضافة إلى إضافة الإجراءات التي تم ذكرها في قواعد الترجمة، و توليد الشكل النهائي للمحلل الفظي من أجل أن يتم ربطه فيما بعد مع المحلل التركيبي.

إن هيكل المحلل اللفظي، هو شيفرة هدف، قد تم تحديد نقاط فيه، ليتم إضافة جداول القيادة عندها، و يتم ذلك من خلال أوامر استدعاء الخرج، و الشكل العام لها هو **%%COMMAND** حيث **COMMAND** تعبر عن الجدول المراد دمجه في هذه النقطة من البرنامج. فيما يلي جدول بأوامر استدعاء الخرج، و الخاصة بالبرنامج **LEX**.

الأمر	الجدول الموافق للأمر
%%PAS_ALPHABET	قيمة، كل جدول ترميز الخاص بأبجدية الأوتومات، و هو جدول يتكون من 255 ، و هذا الجدول سيستخدم كفلتر لمحارف المقروءة ASCIIقيمة تقابل محرف و صيغة الجدول على شكل مصفوفة بلغة الباسكال. من ملف المصدر.
%%PAS_DFATABLE	، و الذي سيقود المحلل اللفظي. وهو مصفوفة DFAجدول الأوتومات المحدد ببعدين، البعد الأول يمثل حالات الأوتومات، و الثاني يمثل رموز أبجدية الدخل (و التي يجب أن تستخلص من الجدول السابق من أجل كل محرف سيتم قراءته من الدخل)، و قيم مركباته تمثل الانتقالات لحالة الأوتومات، اعتباراً من رقم الحالة الموافقة للسطر، من أجل حرف الدخل الموافق للعمود، إلى رقم الحالة الجديدة، والتي هي قيمة المركبة.
%%PAS_ACTIONSPROCEDURES	وهي كتلة الإجراءات الممثلة للفعاليات من أجل كل عنصر لفظي.
%%PAS_ACTIONTABLE	، DFAمصفوفة مؤشرات إلى الإجراءات، حيث كل مركبة توافق حالة من حالات الـ ، من أجل الحالات غير النهائية للأوتومات، أو قيمة nilو يمكن أن تكون إما تؤشر إلى إجراء الخاص بالحالة النهائية.
%%PAS_DEFINES	مجموعة من التصريحات عن الثوابت و التي تمثل أحجام الجداول المولدة.

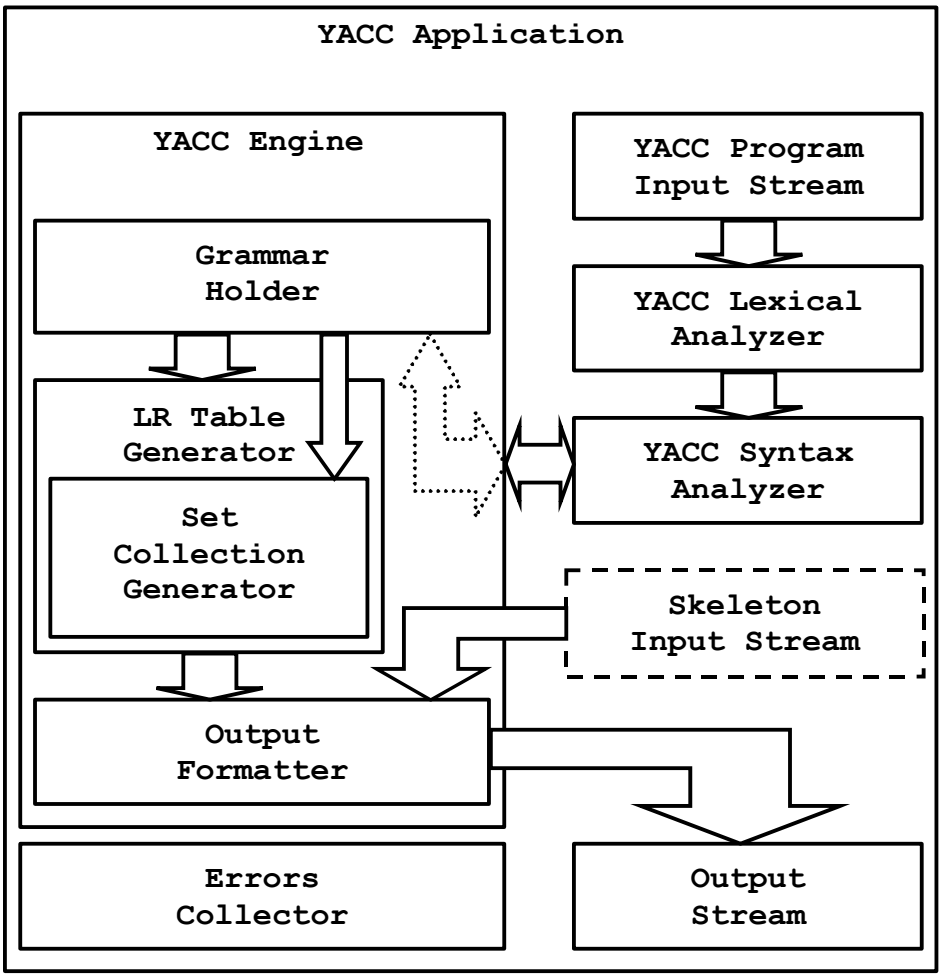
الفصل السادس

مولد المحلل التركيبي

Syntax Analyzer Generator

بنية البرنامج YACC

فيما يلي الشكل العام لأجزاء البرنامج **YACC** المنفذ



الشكل (**6–1**) المخطط العام لعناصر البرنامج **YACC**

يتكون البرنامج بشكل عام من عدة أجزاء وظيفية و هي :

- دفق الدخل للبرنامج **(YACC Program Input Stream)** : وهو الكائن الذي يعبر عن ملف الدخل الحاوي على برنامج الـ **YACC**.

- **المحلل اللفظي (Syntax Analyzer) : و مهمته إنجاز التحليل اللفظي لبرنامج المصدر.**
- **المحلل التركيبي (Syntax Analyzer) : و مهمته إنجاز التحليل التركيبي لبرنامج المصدر، و هذا الجزء هو المسؤول عن استخلاص التصريحات و القواعد، و قوانين المعاني مرتبطة مع القواعد من الدخل.**
- **دفق الدخل لهيكل المحلل اللفظي (Skeleton Input Stream) : و هو الكائن الذي يعبر عن ملف الدخل الحاوي على هيكل النتيجة.**
- **دفق الخرج (Output Stream) : و هو الكائن المعبر عن خرج النهائي للبرنامج YACC.**

محرك الـ YACC (YACC Engine) : و هو الجزء الأساسي من البرنامج، و فيه يتم معالجة كافة المعلومات المستخلصة من الدخل و تشكيل مجموعة المجموعات، و من ثم تشكيل الخرج النهائي ليتم إرساله إلى دفق الخرج. و يتكون كائن قاعدة البيانات من :

(e) **كائن مسك القواعد (Grammar Holder) : و هي عبارة عن كائن يحوي كافة المعلومات المتعلقة بالقواعد المراد إنشاء مترجم لها، بالإضافة لذلك يحتوي هذا الكائن كافة المعلومات المستنتجة من القواعد، كتصنيفات رموز القواعد.**

(f) **مولد جدول التفكيك LR (LR Table Generator) : و هو الكائن المسؤول عن بناء جدول الإعراب اعتباراً من القواعد، حيث يتم في هذا الكائن بناء مجموعة المجموعات، و ذلك من خلال الكائن Set Collection Generator، و استخلاص المعلومات منه في ملئ الجدول.**

(g) **مشكل الخرج (Output Formatter) : حيث يقوم هذا الكائن بتشكيل الخرج النهائي تحضيراً لإرساله إلى دفق الخرج.**

- **جامع الأخطاء (Error Collector) : و مهمته تجميع كافة رسائل الأخطاء المتولدة في مختلف مراحل العمل، ليتم في النهاية إظهارها للمستخدم.**

إن الخوارزميات المستخدمة في عمل مختلف أجزاء البرنامج مذكورة في الفصلين الثاني و الثالث، بما فيها الخوارزميات المتعلقة بعمل المحلل اللفظي و التركيبي LR. كما أن سرد للكائنات المكونة للبرنامج موجودة في الملحق B.

بنية البرنامج المصدر الخاص بالأداة YACC

يتألف البرنامج YACC من ثلاثة أجزاء، هما قسم التصريحات، و قوانين الترجمة، وقسم هيكل المحلل اللفظي، حيث ينهي الرمز %% كل قسم، و الهيكل العام له هو كما يلي :

```
declarations

%%

translation rules

%%
```

في قسم التصريح، يتم ذكر الرموز القواعد المحددة، و ذلك من خلال استخدام الأمر **token%**، يأخذ الشكل التالي:

```
%token SYMBOL1 SYMBOL2 SYMBOL3 …
```

كما و من الممكن استخدام الأمر **token%** عدة مرات:

```
%token SYMBOL1
%token SYMBOL2 SYMBOL3
%token SYMBOL4
      …
```

إن ترتيب التصريحات عن الرموز المحددة مهم جداً، لأنه يحدد ترتيب أعمدة جدول الفعالية **(action table)** الذي سيتم توليده. إن الأداة تقوم بمطابقة الرموز المحددة التي تم التصريح عنها مع التي تستخلصها من القواعد، من أجل التأكد من تطابقها، و في حال عدم تحقق التطابق، تقوم بإصدار رسائل الأخطاء للمستخدم حول ذلك.

و يتم أيضاً في قسم التصريح تحديد رمز البداية للقواعد و ذلك من خلال الأمر **start%** كما في الشكل التالي:

```
%start S
```

إن الأداة **YACC** المنجزة تولد نوعان من جداول الإعراب **LR**، هما جداول الإعراب **LALR**، و جداول الإعراب **Canonical LR**، و ذلك على خلاف معظم أدوات المماثلة، و التي تولد فقط جداول **LALR**. و يتم تحديد نوع الجداول المراد إنشاءها من خلال الأمر **type%** و ذلك باستخدامه أحد الشكلين التاليين:

```
%type canonical
```

أو

```
%type lalr
```

في القسم الثاني من البرنامج يتم ذكر قواعد اللغة المراد إنشاء مترجم لها مع قواني المعاني الخاصة بكل قاعدة، و الشكل العام للقاعدة هو كالتالي:

```
<left side> : <alt 1> #{ semantic action #}
            | <alt 2> #{ semantic action #}

            . . .

            | <alt n> #{ semantic action #}
            ;
```

حيث **<left side>** يمثل الطرف اليساري من القانون، و الذي يجب أن يكون رمز غير محدد الواقع في

الطرف اليساري من القانون. أما <alt i> فتمثل الأطراف اليمنى للاشتقاقات الممكنة للرمز غير المحدد الموجود في الطرف اليساري من القانون، أما **semantic action** فتمثل قوانين المعاني الخاصة بالقوانين، و تكون مكونة من تعليمات تنفذ فعاليات عملية الترجمة من أجل كل إرجاع سينفذ من قبل المحلل التركيبي المولد.

فيما يلي برنامج الـ **YACC** و الذي يصرح عن قواني للغة الخاصة بالأداة المنفذة نفسها:

```
%type lalr
%token block dp id declcommand colen sc bar
%start S

%%
S        : YACC                                  #{#}
         ;
YACC     : DECLS dp RULES dp                     #{#}
         ;
DECLS    : DECLS DECL                            #{#}
         | DECL                                  #{#}
         ;
DECL     : declcommand IDLIST                    #{#}
         ;
IDLIST   : IDLIST id                             #{#}
         | id                                    #{#}
         ;
RULES    : RULES RULE                            #{#}
         | RULE                                  #{#}
         ;
RULE     : LSIDE colen RSIDES sc                 #{#}
         ;
LSIDE    : id                                    #{#}
         ;
RSIDES   : RSIDES bar RSIDE                      #{#}
         | RSIDE                                 #{#}
         ;
RSIDE    : SYMBOLS block                         #{#}
         ;
SYMBOLS  : SYMBOLS id                            #{#}
         | id                                    #{#}
         ;
%%
```

لقد تم استخدام الأحرف الصغيرة في تسمية الرموز المحددة، و الكبيرة من أجل الرموز غير المحددة. إن القواعد المكافئة لتلك الذكورة في البرنامج، تكافئ تماماً القواعد التالية:

```
S → YACC
YACC → DEC %% RULES %%
DECLS → DECLS DECL | DECL
DECL → dec_command IDLIST
IDLIST → IDLIST id | id
RULES → RULES RULE | RULE
RULE → LSIDE : RSIDES ;
LSIDE → id
RSIDES → RSIDES bar RSIDE | RSIDE
RSIDE → SYMBOLS block
SYMBOLS → SYMBOLS id | id
```

```
bar ≡ "|"
```

فيما يلي برنامج يصف القواعد الخاصة بالمحلل التركيبي المستخدم في الأداة **LEX**:

```
%type lalr

%token dp id equ sc colen block bar star plus qmark char
%token lmb rmb str lb rb lmb rmb str lb rb coma minus char

%start S

%%

S          : LEX                                          #{#}
           ;
LEX        : DECLS dp RULES dp                            #{#}
           ;
DECLS      : DECLS DECL                                   #{#}
           | DECL                                         #{#}
           ;
DECL       : id equ RE sc                                 #{#}
           ;
RULES      : RULES RULE                                   #{#}
           | RULE                                         #{#}
           ;
RULE       : RE colen block                               #{#}
           ;
RE         : RE SR                                        #{#}
           | RE bar SR                                    #{#}
           | SR                                           #{#}
           ;
SR         : SR star                                      #{#}
           | SR plus                                      #{#}
           | SR qmark                                     #{#}
           | char                                         #{#}
           | lmb SETCHARS rmb                             #{#}
           | id                                           #{#}
           | str                                          #{#}
           | lb RE rb                                     #{#}
           ;
SETCHARS   : SETCHARS coma SSETCHARS                      #{#}
           | SSETCHARS                                    #{#}
           ;
SSETCHARS  : char                                         #{#}
           | char minus char                              #{#}
           ;
%%
```

فيما يلي التصريح عن قواعد تعبير رياضي، يستخدم العمليات الأربعة، مع تحقيق الأفضليات:

```
%type lalr
%token plus minus mul div lb rb number
%start EXPR
%%
EXPR     : EXPR plus TERM                                 #{#}
         | EXPR minus TERM                                #{#}
         | TERM                                           #{#}
         ;
TERM     : TERM mul FACT                                  #{#}
         | TERM div FACT                                  #{#}
```

```
        | FACT                                          #{#}
        ;
FACT    : lb EXPR rb                                    #{#}
        | number                                        #{#}
        ;
%%
```

بالنسبة للخرج، من الممكن الحصول على محلل تركيبي كامل من خلال تقديم ما يسمى هيكل المحلل التركيبي أو المعرب **(lexical analyzer skeleton)**، و هو عبارة عن شيفرة مصدر، للمحلل تركيبي بشكله العام، و تقوم الأداة بإضافة الجداول المكملة له، بالإضافة إلى إضافة الإجراءات التي تم ذكرها في قواعد الترجمة، و توليد الشكل النهائي للمحلل الفظي من أجل أن يتم ربطه فيما بعد مع المحلل التركيبي.

إن هيكل المحلل التركيبي، هو شيفرة هدف، قد تم تحديد نقاط فيه، ليتم إضافة جداول القيادة عندها، و يتم ذلك من خلال أوامر استدعاء الخرج، و الشكل العام لها هو **COMMAND%%** حيث **COMMAND** تعبر عن الجدول المراد دمجه في هذه النقطة من البرنامج. فيما يلي جدول بأوامر استدعاء الخرج، و الخاصة بالبرنامج **YACC**

PRINCIPLES OF COMPILER DESIGN
ALFRED V. AHO & JEFFRY D. ULLMAN
ADDISON-WESLY PUBLISHING COMPANY
1975

COMPILERS
PRINCIPLES, TECHNIQUES, AND TOOLS
ALFRED V. AHO, RAVI SETHI & JEFFRY D. ULLMAN
ADDISON-WESLY PUBLISHING COMPANY
1985

المترجمات

الدكتور المهندس محمد سعيد كريم

جامعة حلب

1993