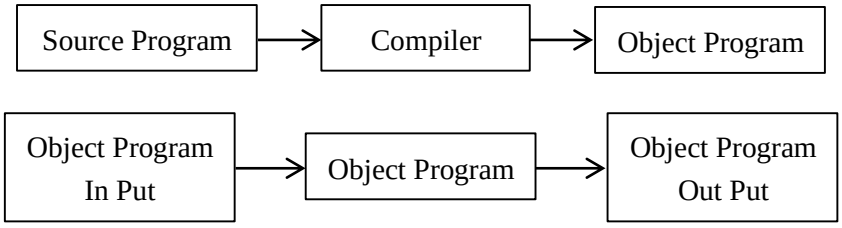
مقدمة عن المترجمات:

من الواضح أنَّه عند تصميم وبناء البرمجيات قد تطرأ عليها بعض التعقيدات، ومن أجل تقليل هذه التعقيدات في تصميم وبناء البرمجيات، تم تصميم هذه البرمجيات من أجل القيام بأوامر بسيطة ولكنها تقوم بها بسرعة عالية، يتم بنائه بكتابة هذه الأوامر بلغة الآلة، ولكنها عملية صعبة وتؤدي إلي حدوث أخطاء كثيرة فتم الاستعاضة عنها بلغات عالية المستوي (High Level Languages) هذه اللغات يمكن أن تكون بعيدة كل البعد عن لغة الآلة والتي لا يفهم الكمبيوتر غيرها، لذا كان لابد أن توجد وسائل لسد هذا الفراغ وهنا يأتي دور المترجم (Compiler).

فالمترجمات هي: جمع مترجم وهو عبارة عن برنامج يأخذ كإدخال نحو برنامج مكتوب بلغة معينة وينتج برنامج آخر مكتوب بلغة أخرى، البرنامج الناتج يسمى (The Object or target Program)، من الواضح أنَّ لغة الآلة (Machine Language) تكون على تماس مباشر مع الحاسب بمسجلاته ومعاملاته المرتبطة بشكل وثيق بالآلة نفسها، وبما أن البرنامج المكتوب بلغة الآلة ليس إلا تسلسل من الواحدات والأصفار، فإنَّ محاولة برمجة خوارزمية معقدة نوعًا ما باستخدام لغة الآلة يعتبر أمراً معقدًا ومرهقا ومفتوحًا على امكانية كبيرة للأخطاء.

ونظرا لصعوبة البرمجة باستخدام لغة الآلة فقد تم التفكير بخلق بيئة برمجية أعلى مستوى تمكِّن المستخدم من كتابة نص برمجي يُعبِّر عن تسلسل أفكاره لحل المشكلة بدلًا من التفكير بطريقة الآلة، وقد كان هذا هو المدخل إلى لغات التجميع الرمزية ومنها إلى اللغات عالية المستوى. إنّ المترجم هو برنامج كبير ومعقد وتحتوي المترجمات غالبًا على مئات الآلاف، إن لم يكن الملايين من أسطر الشيفرة، وتمتلك أجزاؤها العديدة عملًا مشتركاً معقداً. حيث يكون للقرارات التصميميّة المتعلقة بجزءٍ ما تأثيرات متشعبة متعلقة بالأجزاء الأخرى. ولذلك فإنَّ عملية تصميم وبناء مترجم هي تمرين أساسي في هندسة البرمجيات. يحتوي المترجم الجيد على نموذج مصغر لكل ما يشمله علم الحاسب، حيث يحتوي على تطبيق عملي للخوارزميات الكبيرة والتي تتطلب حجز عدد كبير من السجلات وتقنيات البحث التجريبي(جدولة القوائم)،ونظرية البيانات والبرمجة الديناميكية (اختيار التعليمة)، والقواعد المنتهية وقواعد المسح والتحليل وخوارزميات الفاصلة الثابتة (تحليل تدفق البيانات)، يقوم المترجم بالتعامل مع مشاكل مثل الحجز الديناميكي والتزامن والتسمية وادارة هيكلية الذاكرة، ويوفر العمل في تصميم المترجم خبرة عملية في هندسة البرمجيات يصعب الحصول عليها من أنظمة أصغر و أقل تعقيداً.



المترجمات ، وبرامج الترجمة

Compilers and Translators

برامج الترجمة translator Programs: هي برامج تستقبل برنامج الدخل المكتوب بإحدى لغات البرمجة وتولد برنامج الهدف (Source Program) المكتوب بلغة برمجة أخرى.

فإذا كانت لغة المصدر هي لغة عالية المستوى مثل(Pascal, C++ , Java, ...)، ولغة الهدف هي لغة متدنية المستوى كلغة التجميع (Assembly language) أو لغة الآلة (Machine language) فإننا ندعو البرنامج الذي يقوم انطلاقًا من برنامج مكتوب بلغة المصدر بتوليد برنامج مكتوب بلغة الهدف بالمترجم Compiler.

إن تنفيذ البرنامج المكتوب بلغة عالية المستوى يتم على مرحلتين:

1. ترجمة برنامج المصدر إلى برنامج مكتوب بلغة الهدف.
2. تحميل البرنامج الهدف إلى الذاكرة وتنفيذه.

نظراً لتعقيد المترجم compiler وتفاصيله المتشعبة فإنَّ المترجمات كانت تصنف سابقا من البرامج شبه مستحيلة الكتابة، فمثلا أول مترجم للغة Fortran احتاج إلى (18) مبرمج عملوا بجد وبجهد بالغين حتى تمكنوا من تحقيقه.

أمَّا الآن فإنَّ المترجم يمكن بناؤه بجهد أقل وبفاعلية أكبر نظرا لتطور الأدوات المستخدمة في بناء المترجم وعلى أيّ حال فإن خلاصة العمل في هذا المجال للسنوات العشرين الأخيرة قادت إلى الملاحظات التي يجب أخذها بعين الاعتبار عند الشروع ببناء مترجم، وهي:

1. فهم كيفية تنظيم وتمثيل عمليات الترجمة.
2. البحث عن تقنيات وحلول لمعالجة المهام التي ستطرأ خلال الترجمة.
3. تطوير أدوات برمجية تسهل من تحقيق المترجم ومكوناته.

بنية المترجم The Structure Of The Compiler:

إنَّ المترجم يستقبل برنامج المصدر كدخل (input) وبناءً عليه يولد سلسلة من تعليمات الآلة، هذه العملية هي على درجة من التعقيد بحيث أنه من غير المقبول منطقيًا النظر إلى عملية الترجمة كخطوة واحدة، وعلى هذا الأساس وبنظرة متمعنة فإنه يمكن تقسيم عملية الترجمة إلى معالجات فرعية sub processes تدعى كل منها مرحلة (أو طور) Phase.

كل مرحلة من مراحل الترجمة تتلقى مدخلًا واحدًا يعبّر عن ترميز بشكل ما لبرنامج الهدف وتعطي على مخرجها الوحيد ترميزا آخر، وفيما يلي تبيان لأهم المراحل التي تشكل عملية الترجمة، ويتلخص المترجم بمكوناته أو مراحله الستة التالية:

1. المحلل اللفظي (Lexical Analyzer)
2. المحلل القواعدي (Syntax Analyzer)
3. محلل المعاني (Semantic Analyzer)
4. مولد الشيفرة الوسيطة (Intermediate Code Generator)
5. محسن الشيفرة (Code Optimizer)
6. مولد الشيفرة (Code Generator)

أنواع اللغات Types of Programing Language:

1. Machine level Language " لغة الآلة "

هي الصيغة الأدنى للحاسوب، كل ايعاز في البرنامج يمثل ب أرقام وعناوين رقمية تستخدم خلال البرنامج للإشارة إلى مواقع الذاكرة، مثل: ايعاز " Mov, 16".

2. Assembly Languages " لغات التجميع"

في هذه اللغة تستخدم الرموز وتعتبر هي أعلى مستوى من لغة الآلة " Machine Level Language " الأولى وأقل من " High – level Language " وتعطي رمز معين لكل عملية مثل "Add" للجمع ،والـ" Sub" للطرح، والـ "Mult" للضرب، وتكون صيغة الفهم للإنسان.

3. High – level Language " لغة عالية المستوى"

هي لغة مفهومة من قبل الإنسان وتستخدم كلمات وأحرف باللغة الإنكليزية وكذلك رموز العمليات الحسابية والمنطقية مثل لغة C++، ولغة Java، ونستطيع ادخال افكارنا مباشرة إلى البرنامج باستخدام هذه اللغات.

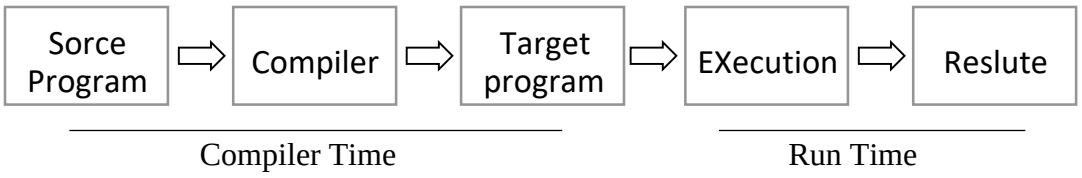
المترجم (Compiler): هو عبارة عن برنامج يقوم بتحويل لغة عالية المستوى مثل C++ إلى لغة الالة أقل مستوى لكي تفهمها الحاسبة .

المفسر (Interpreter) : هو نوع آخر من المترجمات يقوم بتنفيذ البرنامج خطوة خطوة وعندما يظهر خطأ يتوقف ولا يكمل عملية الترجمة إلا بعد تصحيح الخطأ ويكون أبطأ من الـ "Compiler" ولا ينتج برنامج تنفيذي exe.

الفرق بين المترجم والمفسر:

1. من حيث الوظيفة: يقوم المترجم بتحويل برنامج المصدر إلى برنامج تنفيذي حتى يستطيع المعالج التعامل معه، بينما يقوم المفسر بتنفيذ البرامج خطوة خطوة ولا ينتج برنامج تنفيذي.

2. من حيث استغراق الوقت: الوقت الذي يستغرقه المترجم لتحول البرنامج المصدر إلى البرنامج الهدف يسمى وقت الترجمة (Compiler Time)، بينما الوقت الذي يستغرقه المفسر بتنفيذ بقية البرامج يسمى وقت التنفيذ (Run Time).



الأشياء التي الذي ينبغي الحذر منها عندما نريد التصميم على برنامج التحويل البرمجي:

1- Correct ness (التصحيح):

يجب أن يكون المترجم له قابلية على إعطاء تقرير بالأخطاء بصورة صحيحة عندما البرنامج لا يتبع تعابير اللغة.

مثال: int x, iint x.

2- Efficiency (الكفاءة):

يجب الاخذ بنظر الاعتبار السرعة بعملية الترجمة، كلما يكون وقت الترجمة قليل كان تصميم المترجم كفوء (سرعة كتابة البرنامج).

3- Usability (قابلية الاستخدام):

يتميز المترجم بسهولة الاستخدام وان لا يتسم بالصعوبة، فنقصد بسهولة الاستخدام هنا هو ان المترجم يقود المستخدم إلى الخطأ بصورة مباشرة (تحديد الخطأ).

السؤال:

Q\ what are things that should be careful when we want design a compiler?

هناك جزأين من عملية الترجمة:

1. Analysis Part – تحليل.
2. Synthesis Part – تجميع.

المترجم بصورة عامة يعمل بم ارحل، كل مرحلة تحول البرنامج المصدر من تمثيل إلى آخر.

المرحلة الأولى:

1. Analysis Part – تحليل الجزء

في هذا الجزء يقسم البرنامج إلى مجموعة من الأجزاء المتتابعة ويكون تمثيل وسطي للبرنامج ويتضمن أيضًا تكوين Syntax Tree " شجرة الإعراب " من خلال هذه المرحلة يتم تحويل العمليات الموجودة بالبرنامج إلى تركيب هرمي يسمى Tree الشجرة، كل Node عقدة في هذه الشجرة تسمى Operation عملية، والأوارق لهذه العقد تسمى Operand متغيرات.

أسبقيات العمليات:

1. ( ) الأقواس: c*(a+b).
2. ^ الرفع: $(a)^5$+b.
3. *، / الضرب والقسمة.
4. +، – الجمع والطرح.
5. = اليساوي.

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

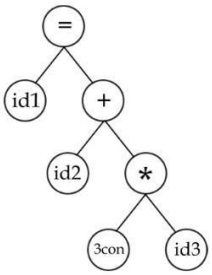
ملاحظة: أثناء رسم شجرة الإعراب للعمليات نبدأ من الأعلى بأقل أسبقية.

Ex: Draws Pars Tree for the following Expression:

1) **a = b+3*d**

Sol:

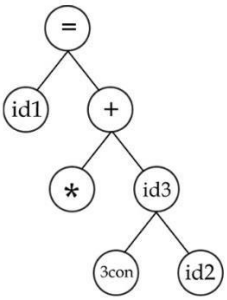
id1 = id2+ Constant * id3



2) **a = 3*d + b**

Sol:

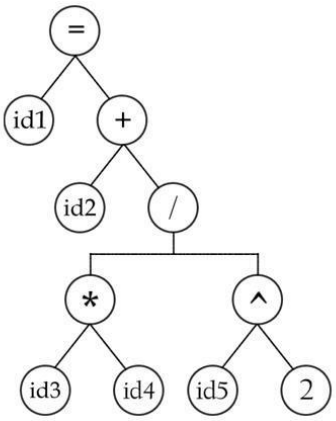
id1 = 3 * id2 + id3



3) c = a + b * c / x^2

Sol:

id1 = id2 + id3 * id4 / id5 ^ 2



. .

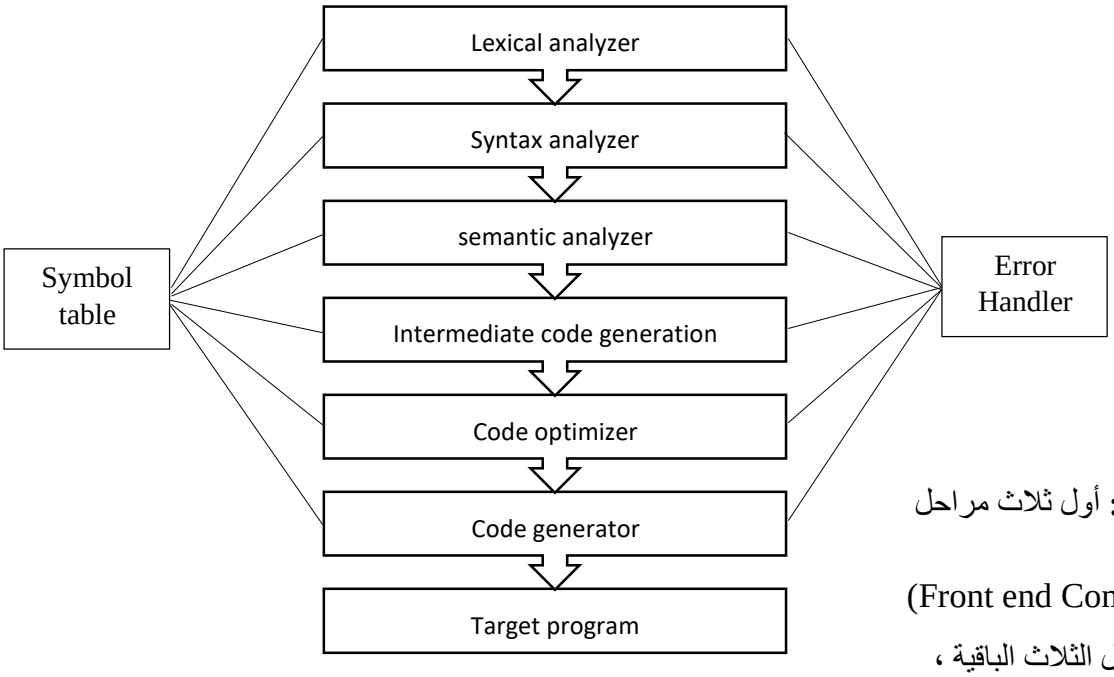
المرحلة الثانية:

2. Synthesis Part – تجميع الجزء

ففي هذا الجزء يتم توليد البرنامج الهدف في التمثيل الوسطي للبرنامج الذي نتج من الجزء الأول.

Compiler Structure(هيكلة المترجم)

المترجم يعمل بمراحل، كل مرحلة تحول البرنامج المصدر من تمثيل إلى آخر، التجزئة المثالية للمترجم موضحة بالشكل (5) وهو اهم شكل.



ملاحظة: أول ثلاث مراحل تسمى:
(Front end Compiler)
والمراحل الثلاث الباقية ، تسمى:
(Back end compiler)

Fig (5) Phases of compiler

الشكل (5) مراحل المترجم

التحليل اللفظي (Lexical analysis):

هو أول مرحلة من مراحل المترجم يتم فيها قراءة نص البرنامج المصدر (Source program) والذي يكون مكتوب بلغة عالية المستوى ويتم قراءة البرنامج من اليسار إلى اليمين ويقطع البرنامج إلى سلسلة من المفردات والتي يستخدمها الـ Parser فيما بعد في مرحلة التحليل القواعدي (Syntax analysis)، وهذه المفردات تسمى: Token.

التحليل القواعدي (Syntax analysis):

مرحلة الاعراب: في هذه المرحلة يتم تحديد فيما إذا كانت سلسلة المفردات Stream of token يمكن توليدها بواسطة قواعد اللغة المعطاة أم لا، إذ أنَّ لكل لغة برمجية قواعد لغة، وتنتج هذه المرحلة شجرة الإعراب (Parse Tree).

المحلل القواعدي له وظيفتان:

1.التحقق من أن الوحدات اللفظية التي تظهر على الـ input (الذي هو خرج المحلل اللفظي) تقع ضمن الصيغ المحددة والمسموح بها في لغة المصدر Source Language.

2.تفرض أو تحدد شجرة بنيوية قواعدية والتي ستستخدم في الأطوار اللاحقة للمترجم.

المرحلة الثالثة

3. Semantic Analysis – التحليل النحوي:

في هذه المرحلة من المترجم تتم عملية مطابقة الأنواع.

```
Int a;
Int b;
Float x;
b= a+x
```

المرحلة ال اربعة

4. Intermediate code generation – مرحلة توليد الكود الوسطي:

يولد تمثيل وسطي يتكون بواسطة الـ Compiler، وليس الـ User، وهذا الكود يجب أن يتصف بصفتين مهمتين:

1. يجب أن يكون سهل التكوين.
2. سهل التحويل إلى البرنامج الهدف – Target Program

المرحلة الخامسة

5. Code optimizer – تحسين الكود:

يحدث تحسين او اختصار للكود الوسطي، لذلك فإن الكود الوسطي الناتج يكون أسرع بالتنفيذ.

المرحلة السادسة

6.Code generation (generator) – مولد الكود:

يتولد الـ Target Code والذي هو عبارة عن Machine code أو Assembly code وكذلك يتم اختصار مواقع الذاكرة للمتغيرات التي تم استخدامها في البرنامج.

Symbol table – جدول الرموز:

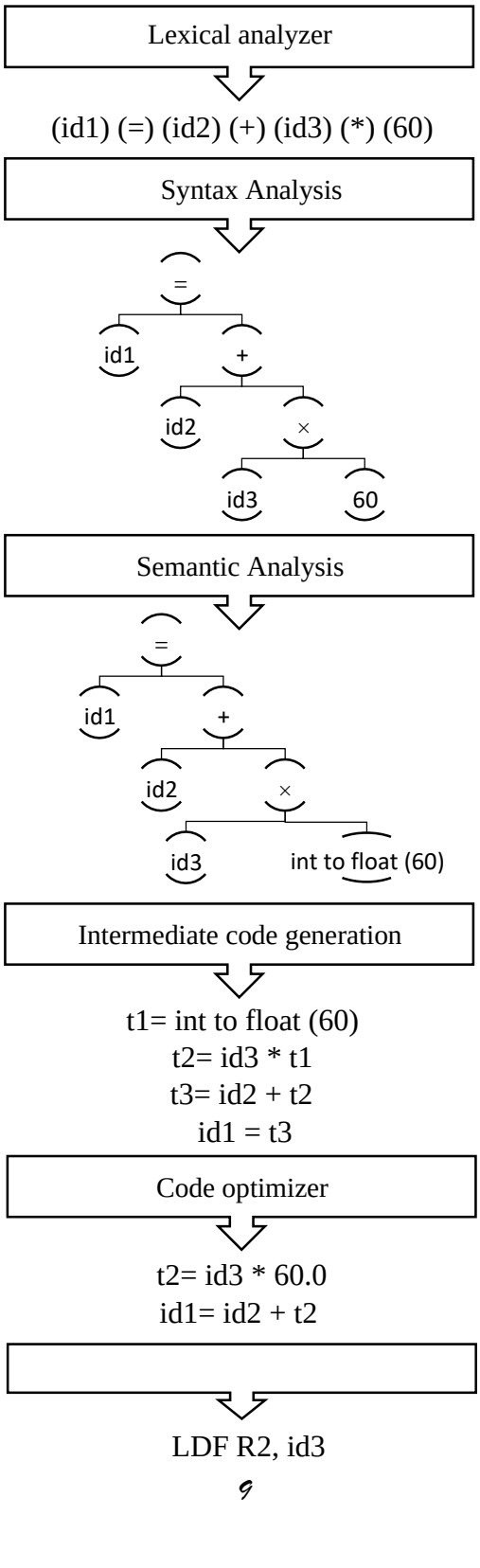
هو عبارة عن جدول يتكون من مرحلة التحليل يخزن فيه المعلومات عن المتغيرات المستخدمة في البرنامج.

Error handler – معالج الخطأ:

برنامج يتم استدعائه عند تحديد أي خطأ في مصدر البرنامج يبعث رسالة تحذيرية للمبرمج بأنه هناك خطأ.

نأخذ مثال، نقوم بحله مرورًا بالمراحل الستة:

EX:
Float a, b, x;
x= a+ b* 60;



MultF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1

Pass:-

One Pass Compiler	Multi Pass Compiler
1. Is a type of compiler that passes through the parts of source file only once.	1. Is a type of compiler that process source program several times.
2. All of the steps happen in one pass read some of source file and analysis it type checked it and optimized it and generate code for it.	2. Separate compilation into multiple pass would continue with the result of previous pass.
3. Called narrow compiler.	3. Called wide compiler.
4. consume less memory because it don't hold intermediate representation of whole program.	4. Consume much memory
5. faster.	5. Slower and more efficient.

(الترجمة)

مترجم ذات المرور الواحد One Pass Compiler	مترجم متعدد المرور Multi Pass Compiler
1. هو المترجم الذي يعالج (يمر) على البرنامج المصدر مرة واحدة فقط.	1. هو المترجم الذي يعالج (يمر) على البرنامج المصدر مرات عديدة.
2. جميع مراحل الترجمة تتم بمرور واحد أي يقرا جزء من الملف الأصلي ويقوم بتحليله واعرابه فحص الأنواع وكذلك تحسن الكود وتوليد الهدف .	2. يفصل مراحل الترجمة في أكثر من مرور على سبيل المثال يقوم بمرحلة التحليل لكل برنامج المصدر بمرور واحد ثم يجري عملية الاعراب لكل البرنامج بمرور آخر وهكذا...
3. يسمى بالمترجم ضيق المدى.	3. يسمى بالمترجم الواسع المدى.
4. يستهلك ذاكرة أقل لأنه لا يحتاج أن يخزن ملفات وسطية.	4. يحتاج إلى ذاكرة أكبر لأنه يحتاج لخزن نتيجة كل مرور بملف وسطي يتم استخدامه بالمرور الآخر.

5. أسرع وأقل كفاءة.	5. أبطأ وأكثر كفاءة.

المترجم ذات المرور الواحد:

يأخذ جزء من البرنامج المصدر أو كله ويقوم بتمريره على جميع المراحل مرة واحدة فقط، بحيث إذا أخذ جزء من البرنامج المصدر لا يتركه حتى يقوم بمروره على جميع المراحل حتى النهاية، ومن ثم ينتقل للجزء التالي، وهكذا، ولهذا لا يستهلك ذاكرة لخزن ملهات وسطية.

المترجم المتعدد المرور:

يأخذ جزء من البرنامج المصدر ويقوم بتمريره على مرحلة أو مرحلتين ،ثم يخزن النتيجة بملف وسطي ،وبعد ذلك يأخذ الجزء المتبقي ويقوم بتمريره بمرحلة أو مرحلتين ومن ثم يخزن النتيجة بالملف الوسطي ،ثم بعد ذلك يأخذ النتيجة من الملف الوسطي ويقوم بتمريره بمرحلة أو مرحلتين متقدمتين، وهكذا.

أنواع الأخطاء Types of Error:

1. أخطاء لفظية (معجمي) – Lexical Error:

عندما تكون هناك أحرف في الإدخال لا تقابل أي مفردة (Token) موجودة باللغة.

مثال:

Int $; ,b= 0..5; , a= 2x → 2*x, 2+x, Fer(.., ..

2. أخطاء قواعدية(إعراب) في بناء الجملة – Syntax Error:

إذا كانت الجملة (عبارة عن سلسلة من المفردات (Tokens))لا تتطابق مع قواعد اللغة.

مثال:

For)int i=0; i<10; i+++), cin<<x, ..

3. أخطاء نحوية – Semantic Error:

إذا كانت الأنواع غير متطابقة (جمع بيانات مختلفة).. ،.

مثال:

عندما أجمع (int) مع (float)، أو عندما أجمع دالة مع مصفوفة، أو اعرف دالة فيها (Parameters 2) ولما استدعي الدالة استدعيها بـ (3 Variables)، ...

4. الخطأ وقت التنفيذ – Run Time Error:

مثال:

القسمة على Zero،

Int x=5, y=0;

x= 1/y;

جدول الرموز – Symbol Table:

يحتاج المترجم إلى معلومات عن الأسماء التي تظهر في برنامج المصدر Source Program، حيث أن هذه المعلومات يتم إدخالها (تخزينها) في بنية معطيات تسمى جدول الرموز Symbol Table. وهو عبارة عن هيكل بيانات يستخدم لتخزين المعلومات التي يتم جمعها خلال مرحلة التحليل، **والمعلومات المخزونة بالجدول تسمى صفات (attributes)، وهي كما يلي:**

1. اسم المتغير – Variable name:

يجب ان يكون موجود دائمًا بالجدول كونه مفتاح البحث عن المتغير وتدخل هذه المعلومة إلى الجدول أثناء مرحلة التحليل اللفظي.

2. عنوان رمز الكائن – object code address:

يحدد الموقع الخاص لقيمة المتغير في وقت التنفيذ.

3. النوع – the type:

هذه المعلومة تستخدم لتحديد مساحة الذاكرة المطلوبة لقيم المتغيرات.

Int = 2, char= 1, float= 4, double= 8, …

4. عدد البعد وعدد المعامل – number of dimension & number of parameter:

متغير بسيط – Simple variable = 0,

مصفوفة أحادية – Vectors = 1,

مصفوفة ثنائية – Matrices = 2,

م/ الدالة يكون بعدها من عدد المعاملات (Parameters) التابعة لها، أو عدد المتغيرات (Variables) التي تستقبلها الدالة.

5. سطر التعريف – line declared:

هو رقم السطر الذي تم تعريف المتغير عنده في البرنامج (سطر واحد)

6. سطر الإشارة – Signal line:

هو رقم السطر أو السطور التي استخدم فيها المتغير عدا السطر الذي عرف به.

م/ إذا ذُكِرت الخاصيتين (6) & (5) في جدول الرموز، فإنه يسمى عبور واشارة (Cross-reference).

الإمكانيات المطلوبة التي يجب أن تتوفر في جدول الرموز (Symbol table):

1. تحديد ما إذا كان اسم معين (اسم عملية، اسم صنف، متغير. ،..) موجود بالجدول أدناه.
2. جدول الرموز يجب أن يكون قادر على خزن واستعادة المفردة (Token).
3. استخدام دوال للجدول:

دالة Insert (s, t): هي دالة إمكانية الإضافة والحذف لبرنامجك لجدول الرموز.

دالة Lookup (s): دالة تقوم بترجيع تسلسل الاسم في الجدول إذا كان موجود (s)، وصفر إذا كان غير موجود.

4. الوصول إلى المعلومات الخاصة باسم معين وامكانية إضافة معلومات جديدة لها.
5. حذف اسم أو مجموعة أسماء من الجدول.

Ex

Draw diagram of across reference Symbol table that would result when compiling the following program Segment?!

```
Void main ()
 {
Int i, j[5];
Char C, index [5][6], block [5];
Float f;
i=0;
i= i+k;
 f= f+i;
C= "x";
Block [4]= C;
}
```

Sol:

Counter	Variable Name	Object Address	Type	Diminution الرتبة	Line declared	Line Reference
1	I	0	int	0	3	6, 7, 8
2	J	2	int	1	3	–
3	C	12	char	0	4	9, 10
4	Index	13	char	2	4	–
5	Block	43	char	1	4	10
6	F	48	float	0	5	8

7	k	–	–	–	–	–

Error Line 7 = K is under fined

Warning Line 3 = J is not used

Warning Line 4 = index is not used

Warning Line 8 = F used before initiation

أنواع جدول الرموز Types of Symbol table:

1. Unordered symbol table: جدول رموز غير مرقم

- تمثيل بسيط.
- السمات (الخصائص) تضاف إلى الجدول حسب تسلسل تعريف المتغير في البرنامج.
- إضافة المتغيرات لا تتطلب عملية مقارنة.
- معدل طول البحث – Average Search Length

$$ASL = n+1/2$$

حيث أن (N) عدد المتغيرات.

2. Ordered symbol table: جدول رموز مرقم

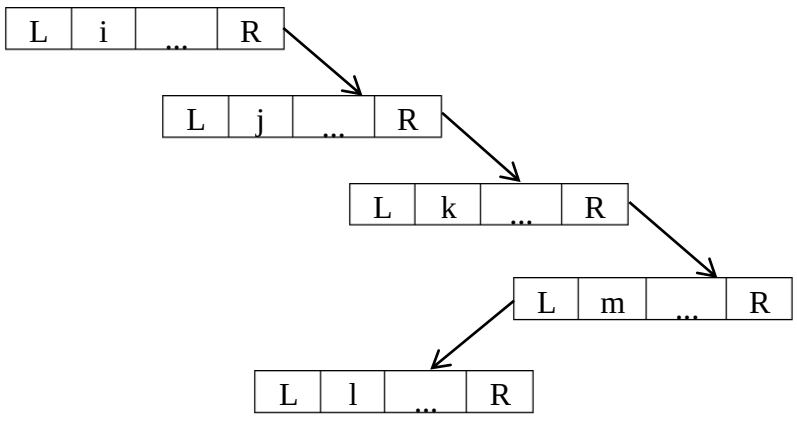
- سجل غير مرتب
- جدول مرتب حسب الهجائية
- عملية ادخال السمات للمتغيرات تتطلب عملية بحث داخل الجدول لمعرفة المكان المناسب للخزن.

3. Tree structured symbol table: تنظيم شجرة جدول الرموز

تتم إضافة حقلين جديدين إلى الهيكل البياني الخاص بكل قيد وهما مؤشر أيمن ومؤشر أيسر

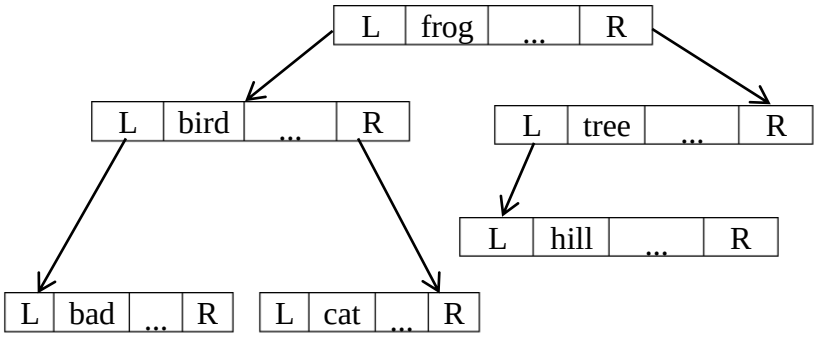
Ex:

i, j, k, m, l



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

Sol:



م/ التعامل مع الحروف الصغيرة والكبيرة لا فرق بينهما.

م/المقارنة تعتمد على الأصل (تبدأ المقارنة من الأعلى نزولاً). .

.

4. Hash table تجزئة الجدول

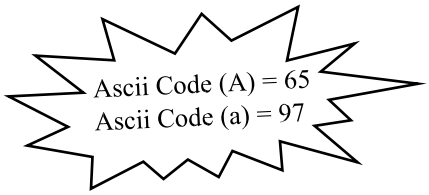
– يعتمد على معادلة معينة لاحتساب المتغير.

hash (اسم المتغير) = (ASCII للحرف الأول من الكلمة + طول الكلمة للمتغير) %Hash max

Ex:

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

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



م/ كيفية حساب باقي القسمة:

الأولى:

106/6 = 17

17 * 6 = 102

106 – 102 = 4

الطريقة الثانية:

106/6 = 17. 6666666667

0.6666666667 * 6 = 4

Sol:

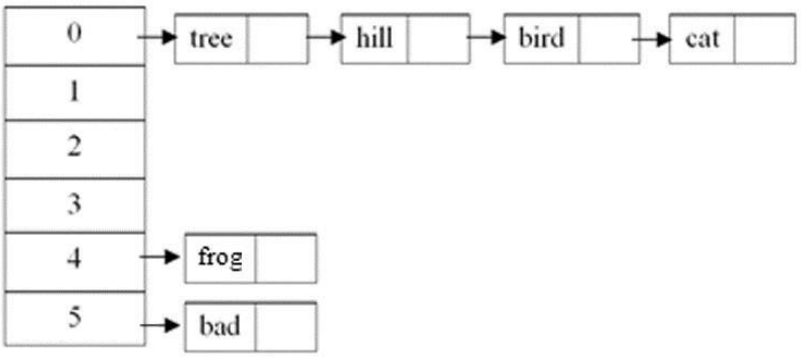
hash (tree) = (4+116)%6= 0

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

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

hash (bad) = (3+98)%6= 4.9 = 5

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

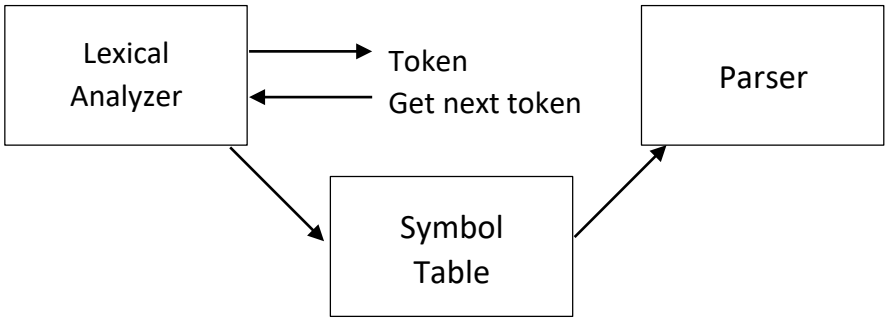


Lexical Analysis:

The role of lexical analysis:-

- is the first phase of compiler, the main task of lexical analyzer is to read. The input characters and produce a sequence of to kens such as names, keyword. The interaction between lexical analyzer and parser implemented by making a lexical analyzer be a subroutine of the parser.

هو المرحلة الأولى من المترجم المهمة الرئيسية للمحلل اللفظي، هي قراءة مجموعة من أحرف وانتاج سلسلة من المفردات (String of tokens). والتعامل بين Lexical وParser تمثل بجعل Lexical جزء من Parser فعندما يتم استلام الأمر الذي هو (Get next token) من Parser فإن Lexical يقرأ مجموعة من الأحرف إلى أن يتم تحديد token جديدة.



The Lexical analyzer is divided in to two parts:

1. Scanning → Scanner.
2. Lexical analysis.

Scanner:

Is responsible for doing a simple task like stripping out from the source program, Comments and whites pace while lexical analysis does the more complex tasks.

الماسح: هو الجزء المسؤول عن المهام السهلة في مرحلة التحليل اللفظي، مثل ازالة التعليقات والفراغات من الـ Source Program بينما الـ Lexical يقوم بمهمة أكثر تعقيدًا منه.

Tokens:

A set of constructs forming the source program language like keyword, identifier constant.

المفردات: هي عبارة عن مجموعة من التراكيب التي تشكل اللغة البرمجية مثل المتغيرات والكلمات المفتاحية وغيرها.

Lexeme:

A sequence of characters in source program that matched by the rule for token.

اللكسيم: هي مجموعة من الحروف التي تنطبق عليها القانون الخاص بـ Token، حيث أن (Lexeme) هي صورة (Token) في البرنامج المصدر.

Pattern: the role that describing the set of lexeme that can represent a particular token.

النمط: هو القانون الذي يصف مجموعة من الـ (Lexemes) والتي تمثل الـ (Token) معينة ويعتبر محور عمل (Lexical analyzer).

Types of token:-

1. Identifiers (Variables) معرفات، متغيرات
2. Constants الثوابت
3. Operators (Operation) +, −, *, /, ^. الرموز المنطقية، العمليات
4. Relational <, >, !=, ≤, ≥, =. الرموز العلائقية
5. Special characters ($, ;, {, }, …) الرموز الخاصة
6. Key words (for, int, if, …) الكلمات المفتاحية

المدخلات التخزين المؤقت

Input Buffering

بما أنه الـ (Lexical Analyzer) يقرأ حرف حرف فمن غير المعقول أن يقوم بعملية الوصول للقرص (Hard) عند قراءة كل حرف، لذلك يقوم بجلب مجموعة من الأحرف(Block – بلوك) أو يخزنها بمخزن (Buffer) وحجم ذلك المخزن قد يكون 100، حسب تصميم الـ Lexical Analyzer، هناك مؤشرين لكل مخزن (Buffer)

1. **b Ptr مؤشر البداية**، يشير إلى بداية اللكسيم الحالي صورة المفردة token في البرنامج المصدر Source Program
2. **f Ptr مؤشر النهاية**، يتحرك إلى الأمام إلى أن تتم مطابقة النمط (Patren) أي عندما تشكل مجموعة من الأحرف التي تمت قراءتها توكن معينة.

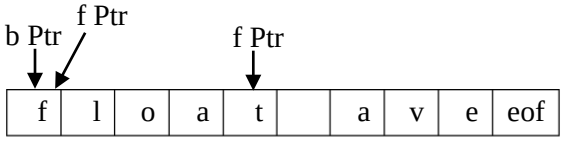
عند الوصول إلى نهاية المخزن(Buffer)، (eof) فإنَّ الـ Lexical analyzer سيقوم بعملية إعادة تحميل للمخزن ،Lexical analyzer إما يستخدم One Buffer، أو Two Buffer لكن عند استخدام One Buffer سنواجه مشكلة إذا جاءت اللكسيم (Lexeme) في نهاية البفر، وامتلئ البفر قبل أن تنتهي اللكسيم.

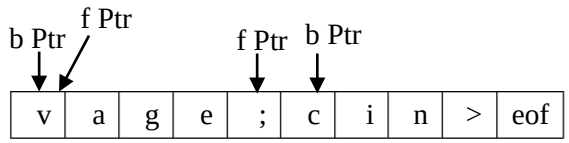
م/ إذا وصل الـ F Ptr إلى نهاية البفر وعملنا إعادة تحميل له، فستكتب بيانات جديدة فوق البيانات القديمة وبذلك سنفقد بداية اللكسيم.

Ex:

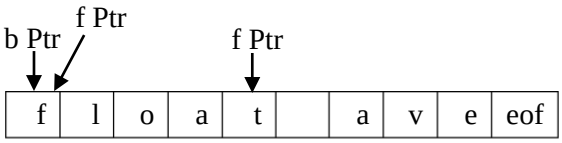
Float average;
Cin>> average;

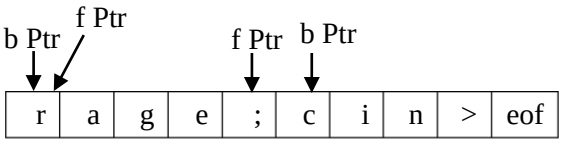
Sol:





*** لحل هذه المشكلة سيتم استخدام Buffer 2:**





* إذا وصلنا إلى نهاية البفر الأول ستجرى عملية إعادة تحميل للبفر الثاني من البرنامج المصدر، واذا وصلنا إلى نهاية البفر الثاني ستجرى عملية إعادة تحميل للبفر الأول.

* عند وصول الـ f ptr إلى نهاية اللكسيم فإن الـ Lexical analyzer سيأخذ الكلمة المحصورة بين الـ b ptr – f ptr.

أمثلة على اللكسيم: (اللكسيم صورة التوكن (المفردة) في البرنامج المصدر)

- Float هو لكسيم للكلمات المفتاحية
- + لكسيم للعمليات الحسابية
- < > لكسيم للعمليات العلائقية

س/ عرف اللكسيم واعطِ مثال؟ .

أسئلة الفصل الأول:

(A)

Q1) what is the Difference between compilation time and runtime, draw a figure to illustrate compilation process.

Q2) what is dimension attribute? and what type of error can be determined based on the dimension attribute?

Q3) draw tree structured symbol table to store the following variables: name, age, degree, average.

Q4) how one pass compiler work?

Q5) what is the input and output for semantic phase in compiler?

(B)

Q1) Where the variables (floor, door, window) will be stored in hashing type of symbol table if the hash table size is (100) record.

Q2) define (Parser, Lexeme, and Machine Level Language).

Q3) What Error Handler? List types of Error and explain it with examples.

(C)

Q1) what is the Difference between Compiler and interpreter?

Q2) what is the problem that accrue when Lexical Analyzer use one buffer for input buffering? explain with examples?

Q3) draw syntax tree for following expression: a=d*c/2-5.

Q4) list symbol table Types.

(D)

Q1) what are compiler phases?

Q2) what is the Difference between ordered symbol table and unordered symbol table.

Q3) How multi-pass compiler work?

Q4) what are variables (dog, cat, bird, door) will be stored in hashing type of symbol table if the hash table size is (100) record.

التعرُّف على المفردات

Recognition of token

في اللغات البرمجية يوجد أنواع من المفردات، ولتمثيل هذه المفردات فإن الـ Lexical analyzer سوف يرسل زوج من القيم للمعرب (Parser) وهي:

< token , token >

Type , value

التوكن Type: يخبرنا عن صنف هذه التوكن هل هي متغير أو عملية حسابية أو ثابت (رقم)

التوكن Value: عبارة عن معلومات عن هذه التوكن وتسمى أيضًا (Token Attributes).

*الـ Lexical analyzer يرسل كود رقمي وليس كلمات.

*إذا كانت التوكن من النوع البسيط مثل (الكلمة المفتاحية (int, float, ..)، {، }، ،، ..(فإن الـ Lexical analyzer سوف يرسل كود رقمي واحد فقط يبين نوع التوكن، أما إذا كانت معقدة (الثوابت والمتغيرات) فإن الـ Lexical analyzer سوف يرسل كود رقمي بنوع التوكن إضافة إلى مؤشر لجدول الرموز يدل على مكان خزن هذا المتغير في جدول الرموز:-

< id, Ptr to Symbol table entry >

س/ ما هي أنواع المفردات التي يعتمد عليها الـ Lexical Analyzer؟

ج/ المفردات البسيطة، والمفردات المعقدة.

م/ وقد تستخدم الـ Token value لتمييز الـ Token مثلا لو عاملنا الـ Operator كأنها token لسوف تحتاج الـ Token Value لتمييز نوع الـ Operator:

< Operator , + − * / >

Ex (للاطلاع) Consider Source encoding of token as follows:

Token	Token Type Code	Value
If	1	–
Else	2	
For	3	
Identifier	4	Pointer to symbol table entry for Identifier
Constant	5	Pointer to symbol table entry for Constant
<	6	1
>	6	2
==	6	3
>=	6	4
<=	6	5
(	7	
)	8	

Location	Category	Value
100	Identifier	x
101	Identifier	a
102	Constant	8
103	Constant	6.7

Write the tokens stream generated by lexical analyzer for following cod:

If (a>8)

Sol

<1,–> <7,–> <4,101> <6,2> <5,102> <8,–>

Ex (للاطلاع) Consider Source encoding of token as follows:

Token	Token Type Code	Value
:		
if	51	–
Identifier	334	Pointer to symbol table entry for Identifier
Constant	9	Pointer to symbol table entry for Constant
int	7	–
float	12	–
+	75	1
–	75	2
*	75	3
/	75	4
;	60	–
!=	23	–
=	20	–
:		

Location counter	Token Type	Value
100	Constant	8
101	Identifier	h
102	Constant	4
103	Identifier	i

Write the tokens stream generated by lexical analyzer for following cod:

int h=4;

h=h+8;

Sol int h=4; ≡ <7,–> <334,101> <20,–> <9,102> <60,–>

h=h+8; ≡ <334,101> <20,–> <334,101> <75,1> <9,100> <60,–>

خطوات تصميم الـ Lexical analysis (Stepped Design Lexical Analysis):

1. كتابة Regular Expression (RE) لكل مفردة (Token).
2. بناء Nor deter monistic Finite Automata (NFA) لكل (RE).
3. تحويل (NFA) إلى (DFA).

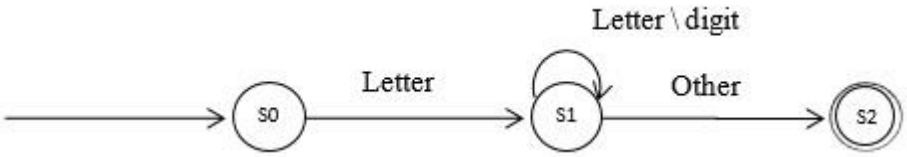
ملاحظة: DFA: تستخدم لقبول أو رفض مجموعة من الأحرف والرموز لذلك تم استخدامها لـ Lexical analyzer لمعرفة هل المفردة تنتمي إلى هذه اللغة ام لا.

Ex:

Write Regular Expression (RE) for following token and then draw transition diagram for each one.

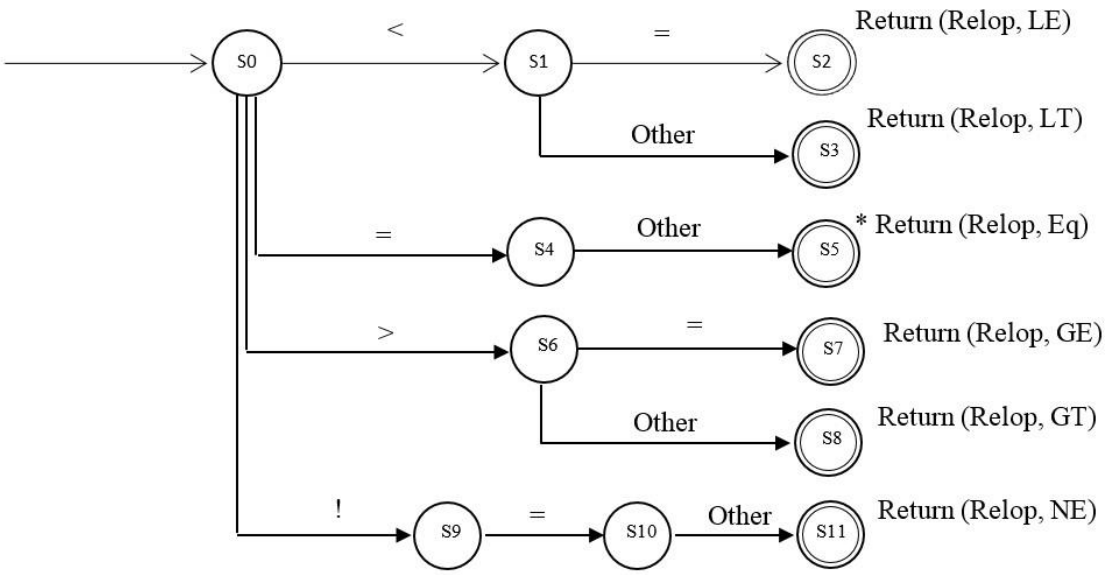
1. Identifier

R = Letter (Letter / digit) *



2. RE for Relop (Relation Operation)

R = < = | = | >= | = | >| !=



Parser:

هو البرنامج المسؤول عن تسلسل المفردات في مرحلة الـ Lexical analysis، ينبغي أن تكون صحيحة.

Lexical analyzer:

هو البرنامج المسؤول عن تسلسل الأحرف داخل المفردة الواحدة في مرحلة التحليل القواعدي (Lexical analysis).

Ex:

int x;

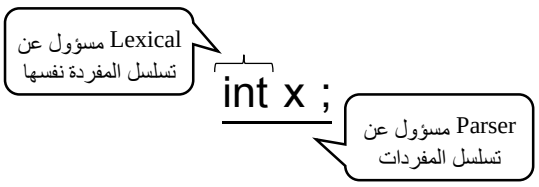
سيمر بمرحلة التحليل القواعدي عبر المحلل اللفظي والمعرب، والاثنين صحيح

nit x;

سيمر بمرحلة التحليل القواعدي ويأتي على أول مفردة (nit) تسلسل الاحرف داخل المفردة الواحدة خطأ ،ولهذا يقف عند الـ Lexical analyzer لأن تسلسل الاحرف داخل المفردة ليست مقبولة قواعديًا.

int ;x

سيمر بمرحلة التحليل القواعدي أول مفردة صحيحة قواعدية (مفردة صحيحة) وأيضًا صحيحة تسلسل المفردة، ويأتي إلى المفردة الثانية (;) صحيحة قواعديًا، لكن تسلسل المفردة غير صحيح توقف عند الـ (Parser) لان (;) تسلسلها يكون بعد المتغير لا قبله.



3. Constant

R =Float تمثل(Recolor)لنفسها

f	l	o	a	t		x	

1 e + 3 = 1000, how do I 1e + 'x' where x is variable:

$1000 = 1 * 10^3 \rightarrow 1\ e + 3 \mid 10\ e + 2 \mid 100\ e + 1$

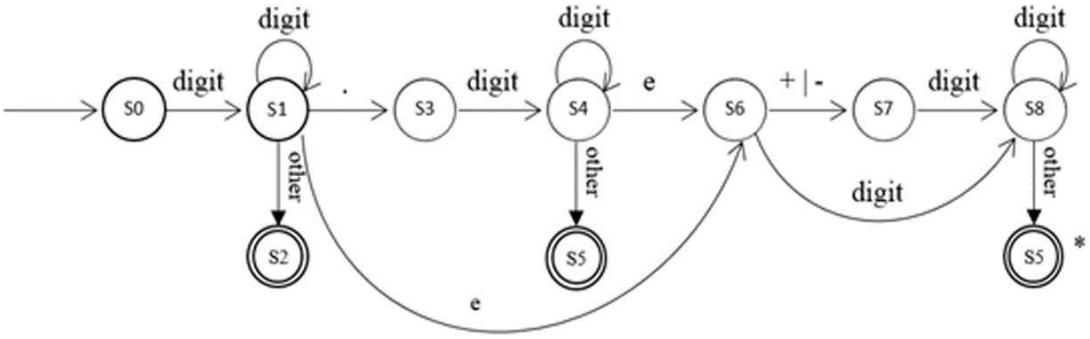
$50\ 000 = 5 * 10^4 \rightarrow 5\ e + 4$

$0.001 = 1 * 10^{-3} \rightarrow 1\ e - 3 \mid 0.1\ e - 2 \mid 0.01\ e - 1$

$0.0008 = 8 * 10^{-4} \rightarrow 8\ e - 4$

R = (digit)$^+$ |(digit)$^+$ (.) (digit)$^+$ | (digit)$^+$ e (+|-) (digit)$^+$ | (digit)$^+$ (.)(digit)$^+$ e (+|-) (digit)$^+$

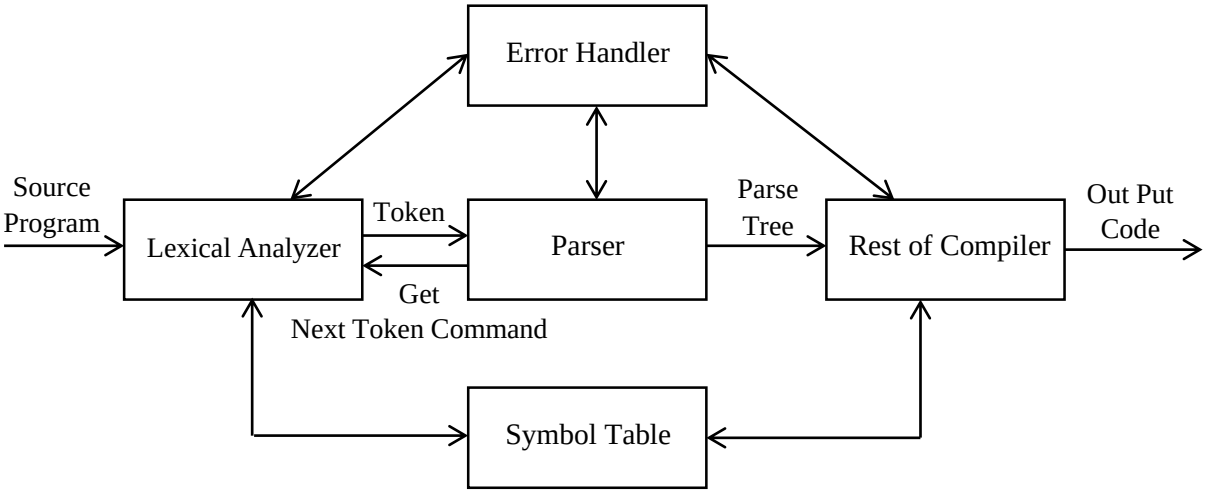
Not: λ (Lambda) = Empty
0 (Zero) = (digit)$^+$



وظيفة المحلل

Role of Parser

هو برنامج يقوم بتنفيذ التحليل القواعدي (Syntax analyzer)، حيث يتم أخذ سلسلة من المفردات (String of token) من المحلل اللفظي (Lexical analyzer)، ويقوم بتدقيق تتابع المفردات بهذه السلسلة فيما إذا كانت مقبولة قواعديًا ذات تركيبة متحققة في لغةٍ ما أي متولدة باعتماد قواعد تلك اللغة، وذلك باشتقاق تلك السلسلة ابتداءً من رمز البداية لتلك القواعد (Grammar) ويبني شجرة اعراب لها وعن طريقها سوف يعرب الـ (Parser) فيما اذا كان هناك أخطاء قواعدية ام لا.



وصف القواعد

Specification of Syntax

وصف قواعد اللغات البرمجية يجب أن يكون دقيق وغير غامض، لذلك سوف نستخدم (Contex Free Grammar) ويمكن تمثيل الـ (CFG) بالشكل التالي:

$$G = \{ V, T, S, P \}$$

V = Set of non-terminal

T = Set of Terminal

S = Start Symbol

P=Production Rule

قواعد الإنتاج (CFG) تكتب بالصيغة التالية:

$$NT \rightarrow \alpha$$

$$\alpha \in \{ V \cup T \cup \lambda \}$$

بمعنى: أنَّ α تكون إما V، أو T، أو λ أو جميعهم.

If we want to define declarative sentence using **CFG** it could be as follow:

- اكتب الجمل الخاصة بالجمل التعريفية:

State → type | List | terminator

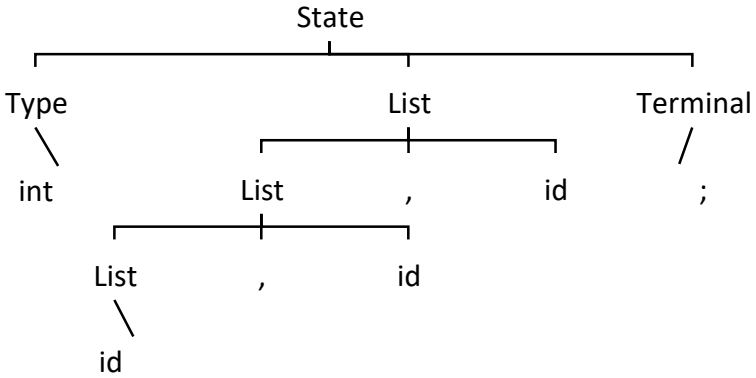
Type → int | float | char

List → List, id | id;

Terminator → ; **Ex:**

W= int id, id, id, id;

Sol:



الاشتقاق وشجرة الاعراب

Derivation and Parse Tree

لمعرفة هل الجملة مقبولة قواعديًا أم لا سوف نستخدم طريقة الاشتقاق أي نبدأ من عنصر البداية من القواعد المعطاة ونعوض عنه بواسطة احلاله بما يقابله في الجهة اليمنى من نفس القافلة ونستمر بعملية الاشتقاق إلى أن نبرهن أن التعويض النهائي بشكل الجملة نهسه وعندئذٍ نستنتج أن الجملة صحيحة قواعديًا

وهناك طريقتين

1. Left Most Derivation (LMD) الاشتقاق أيسر الترتيب

م/ يتم فيها اشتقاق الرموز القابلة للاشتقاق (Non-terminal) الكبيرة التي تقع أقصى اليسار.

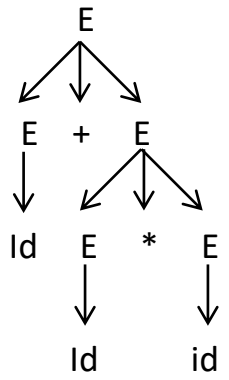
Ex:

$E \rightarrow E + E \mid E * E \mid id$

W= id + id *id $

نهاية الـ String

E
E + E
id + E
id + E * E
id + id * E
id + id * id



2. Right Most Derivation (RMD) الاشتقاق أيمن الترتيب

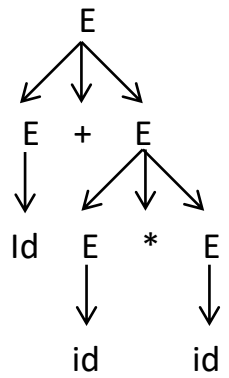
هي طريقة يتم فيها اشتقاق الرموز القابلة للاشتقاق (non–terminal) التي تقع أقصى اليمين.

Ex:

$E \rightarrow E + E \mid E * E \mid id$

W = id + id * id $

E
E + E
E + E * E
E + E * id
E + id * id
id + id * id



اذا جاء في السؤال id+* id لايمكن الاشتقاق ولا يمكن تكوين Parser Tree
ولا تطابق ال grammar

Context Free Grammar (CFG)

Ex:

Driver (LMD)

W= a*(b+a) $

S → E

E → T | E + T

T → T * F | F

F → P

P → (S) | a | b

* لا يمكن التحويل إلى الحد الثاني إلا بعد تحويل كل ما قبله إلى terminal.
* (LMD) هو الاشتقاق من أو الحرف من اليسار إلى اليمين (باتجاه اليمين)
* (RMD) هو الاشتقاق من نهاية الحرف إلى اليسار (باتجاه اليسار)

LMD

S
E
T
T* F
F * F
P * F
a * F
a * P
a * (S)
a * (E)
a * (E + T)
a * (T + T)
a * (F + T)
a * (P + T)
a * (b + T)
a * (b + F)
a * (b + P)
a * (b + a)

H.W: حل نفس السؤال بطريقة (RMD).

Ex:

W = aaabbabbba$

S → aB | bA

A → a | aS | bAA

B → b | bS | aBB

Sol:

S **RMD**

aB

aaBB

aaBaBB

aaBaBbS

aaBaBbbA

aaBaBbba

aaBabbba

aaaBBabbba

aaaBbabbba

aaabbabbba

H.W: حل نفس السؤال بطريقة (LMD).

تحليل التقنيات

Parsing techniques

1. top down Parsing من الأعلى إلى الأسفل
2. bottom Up Parsing من الأسفل إلى الأعلى

top down Parsing:

فيه يتم الاشتقاق ابتداءً من رمز البداية لتلك القواعد إلى أن نصل إلى السلسلة المطلوبة.

bottom Up Parsing:

فيه يتم الاشتقاق ابتداءً من الجملة (سلسلة المفردات) إلى أن نصل إلى رمز البداية (Start Symbol) لتلك القواعد.

*** كل Token داخل الـ Source Program هو terminal.**

Ex: Consider the given grammar:

S → aB | bA

A → a | aS | bAA

B → b | bS | aBB

Obtain left most derivation for the string "aababbabb".

Sol

S

aB

aaBB

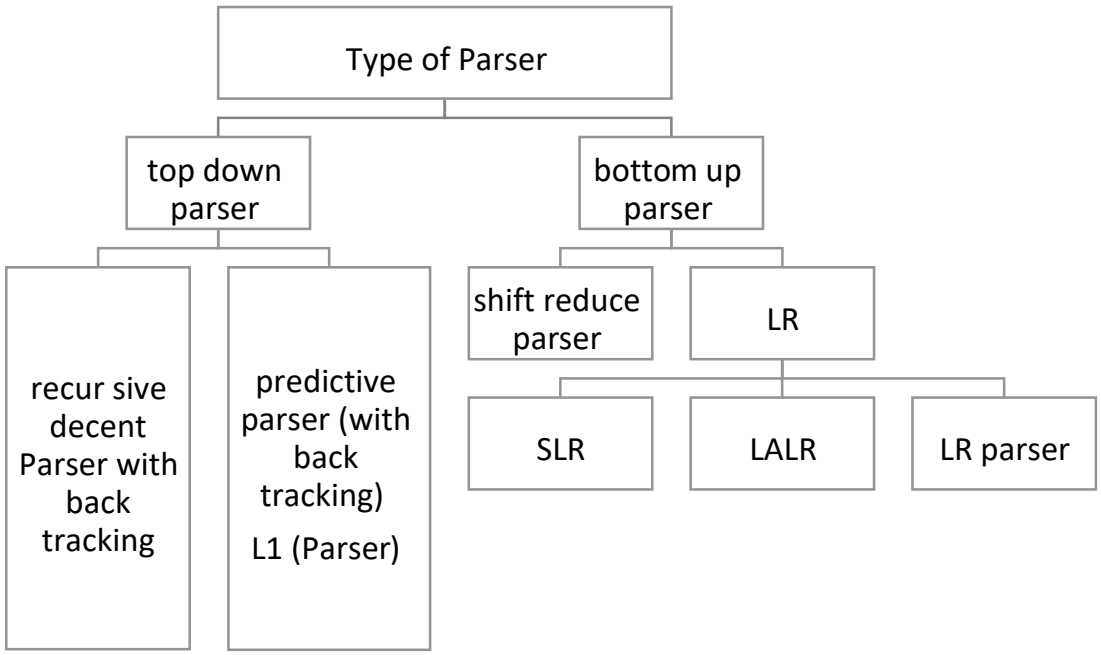
aabSB

aabaBB

aababSB

aababaBB

aababab



Brain Type of Parser

. .

مشاكل الـ CFG

Problem With top down Parser

هناك مشاكل معينة قد تواجه عملية الاعراب من الأعلى إلى الأسفل لذلك يجب الغاء هذه المشاكل لكي نتمكن من تنفيذ الاعراب التنبؤي.

1. التراجع Back Tracking:

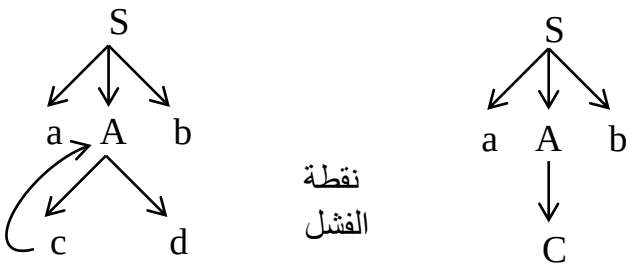
عندما يقوم المعرب بإعراب جملة فإنه ينتقل إلى جهة اليمين من قاعدة الإنتاج المقصودة في حالة عدم التطابق مع الجملة المراد اعرابها يراجع إلى اليسار ليتخذ طريقا آخر في الاعراب والمثال التالي يوضح ذلك..

EX:

S → aAb

A → cd | c

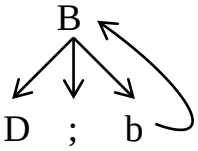
W = acb $



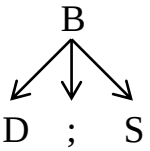
إن عملية الفشل والرجوع إلى جهة اليسار من قاعدة الإنتاج تسمى **Backtracking**.

EX:

$B \rightarrow D\ ;\ b \mid D\ ;\ S$ $W = D\ ;\ S\ \$$



نقطة الفشل



ويمكن تعريف التراجع بأنه: حالة انتقال معرب من اليسار إلى اليمين لقاعدة الإنتاج وفي حالة حصول تطابق بين الجملتين (الجملة المعربة والجملة المراد الوصول إليها) يتراجع المعرب إلى اليسار ليتخذ طريق آخر في الإعراب كما في المثال السابق.

. .

2. التكرار الذاتي Left Recursion:

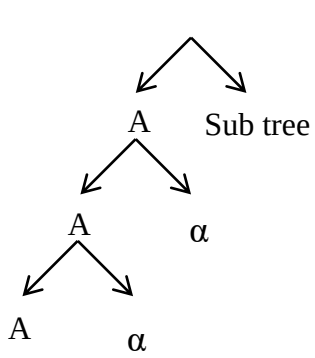
- ينقسم إلى قسمين:

1(التكرار الذاتي المباشر - (Direct) immediate Left Recursion

تكون صيغة القواعد الذي يحتوي على تكرار ذاتي أيسر، كالتالي:

$A \rightarrow A\ \alpha \mid \beta$

أي ظهور non terminal في اليسار في أقصى يسار واليمين من نفس قاعدة الإنتاج، فعند وجود التكرار الذاتي الأيسر فإنَّ المعرب قد يدخل في Lop غير منتهي كما موضح في الشكل:-



لإلغاء التكرار الذاتي الايسر نحتاج أن نحدد على الـ Grammar، فنعيد صياغة الـ Grammar بالخطوات: قاعدة:

$A \rightarrow A\ \alpha \mid \beta$

$A \rightarrow \beta\ A'$

$A' \rightarrow \alpha\ A' \mid \lambda$

م /β هي كل الحدود الناتجة عد التي تحتوي على ال non terminal المطلوب.

م /A' هو الذي يحتوي على non terminal بالإضافة الى λ

Ex:

$E \rightarrow E + T \mid T$

Sol:

$E \rightarrow T\ E'$

$E' \rightarrow + T\ E' \mid \lambda$

Ex:

$T \rightarrow T * F \mid F$

Sol:

$T \rightarrow F\ T'$

$T' \rightarrow * F\ T' \mid \lambda$

Ex:

exp → exp or term | term

term → term and factor | factor

factor → not factor | (exp) | tree | false

factor لا يوجد فيه Left Recursion لأنه لم يظهر factor في أقصى اليسار (بداية) الجهة الثانية (تنزل كما هي).

SOL\

exp → term exp' exp'

→ or term exp' | λ

---

term → factor term' term'

→ and factor term' | λ

---

factor → not factor | (exp) | tree | false

Ex:

$A \rightarrow A\ \alpha1 \mid A\ \alpha2 \ldots \mid A\ \alpha n \mid \beta1 \mid \beta2 \ldots \mid \beta n$

إذا كان أكثر من β نكتب أكثر من β وإذا λنكتب λ واحد

$A \rightarrow \beta1\ A' \mid \beta2\ A' \ldots \mid \beta n\ A'$

$A' \rightarrow \alpha1\ A' \mid \alpha2\ A' \ldots \mid \alpha n\ A' \mid \lambda$

* β هو الحد الذي لا يحتوي على Left Recursion

Q\ Eliminate Left Recursion from follow grammar:

1.$E \rightarrow E + T \mid E - T \mid T$

2.$T \rightarrow T * F \mid T / F \mid F$

3..$F \rightarrow (E) \mid id$

Sol

1.

$E \rightarrow T\ E'$

$E' \rightarrow + T\ E' \mid - T\ E' \mid \lambda$

2.

$T \rightarrow F\ T'$

$T' \rightarrow * F\ T' \mid F\ T' \mid \lambda$

3.

$F \rightarrow (E) \mid id$ (تنزل كما هي)

Q\ Eliminate immediate Left recursion Form given Grammar:

1. $X \rightarrow Xa \mid X+b \mid Ba$

2. $B \rightarrow b \mid Ba$

Sol

1.

$X \rightarrow BaX'$

$X' \rightarrow aX' \mid +bX' \mid \lambda$

2.

$B \rightarrow bB'$

$B' \rightarrow aB' \mid \lambda$

2) التكرار الذاتي الغير المباشر Not immediate Left Recursion (indirect)

Ex:

S → Aa | b

A → Ac | Sd | f

Sol:

S → Aa | b

A→Ac |Aad|bd|f

Ex:

1. S → AB | BA | a
2. A → Sa | b
3. B → bB | AB | a

Sol:

* لأنها لا تحتوي على A فهي β شرط A في البداية

1. S → AB | BA | a
2. A → ABa | BAa | aa | b
3. A → BAaA' | aaA' | bA'

* نطبق القاعدة كي نتخلص من الـ Left Recursion

A' → BaA' | λ

* نعوض من A الناتجة وليس من الـ A الموجودة بالسؤال:

B → bB | BAaA'B | aaA'B | bA'B | a

B → bBB' | aaA'BB' | bA'BB' | aB'

B' → AaA'BB' | λ

Ex:

1. S → SX | SSb | XS | a
2. X → Sa | Xb | b

*** نلغي S الأولى فقط بطريقة .. Left**

S → XSS' | aS'

S' → XS' | SbS' | λ
X → XSS'a | aS'a | Xb | b

X → aSaX' | bX' | λ

X' → SS'aX' | bX' | λ

*** عندما تطلب غير مباشر يعني تعويض**

*** هي من تحدد الطريقة بالسؤال..**

3. التحليل من اليسار Left Factoring:

هو عملية تحليل قاعدة الإنتاج لإيجاد المشترك، الصيغة العامة هي:

A → α β1 | α β2

من الصعب اختيار أي Rule أو قاعدة، بهذا سوف نأخذ لحل هذه المشكلة (سوف نكتب grammar) بصيغة أخرى، وهي:

A → α A'

A' → β1 | β2

Ex:

S → iEts | iEtses | a

E → b

Make the following grammar a left factored grammar

S → iE t SS' | a

S' → eS | λ

E → b (تكتب كما هي)

* من يبقى أي شيء نكتب λ ولا يوجد داعي لكتابتها أكثر من مرة في السطر الواحد..

Q\ Make following Grammar a Left Factored Grammar:

Q → aQB | aE | q

E → beQ | b | bB

B → b | λ

Sol:

Q → aQ' | q

Q' → QB | E

E → bE'

E' → eQ | B | λ

B → b | λ

Ex

A → aAB | aA | a

Sol

A → aA'

A' → AB | A | λ

Ex

A→ aAB | aA

A→ bB | b

Sol

A → aAA'

A' → B | λ

B → b B'

B' → B | λ

4. الغموض Ambiguity:

الغموض: هو تعدد الاشتقاق لـ Grammar معين، والـ grammar الغامض غير مرغوب به في تقنية الاعراب من الأعلى إلى الأسفل، لذلك يجب ان يلغى الغموض إن وجد.

القاعدة: –

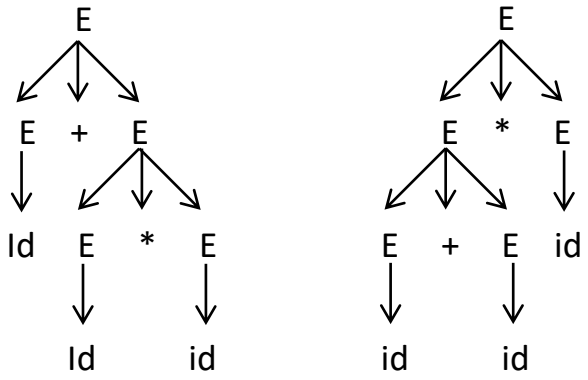
$$\boxed{NT \rightarrow NT\ \alpha\ NT}\ .$$

Ex:

$E \rightarrow E + E \mid E * E \mid id$

Sol:

id + id * id



تكونت شجرتي اعراب لنفس الجملة أي إن الـ grammar غامض ويمكن معرفة هل الـ grammar غامض أم لا بدون رسم شجرة الاعراب بواسطة بعض الصيغ:

1. $\mathbf{NT \rightarrow NT\ \alpha\ NT}$

$S \rightarrow S + S$

$S \rightarrow 2$

بمعنى آخر ظهور الـ non terminal في جهة اليمين من قاعدة الإنتاج في أقصى اليمين وفي أقصى اليسار، حيث أن الـ non terminal تمثل الرموز القابلة للاشتقاق.

$\mathbf{NT \rightarrow NT\ \alpha \mid \beta\ NT}$

$E \rightarrow E\ a \mid b\ E$

بمعنى ظهور الـ non terminal في جهة اليمين من قاعدة الإنتاج مرة في أقصى اليمين ومرة في أقصى اليسار.

Left associative	Right associative
=	+ , −
↑	* , /
not	OR
	and

هناك قاعدتان للتخلص من الغموض,وهما:

1. Operator Precedence – أسبقية العمليات
2. Associativity of Operator – تجميع العمليات

قبل التحويل من الـ ambiguous gram إلى الـ un ambiguous gram يجب أن نعرف هل الـ α الموجودة في القاعدة: **NT → NT α NT** هل هي Left associative أو Right associative من اليسار إلى اليمين، أو من اليمين إلى اليسار.

* العمليات التي تكون Left associative نحولها Left recursion، والعمليات التي تكون Right associative نحولها Right recursion.

Ex:

$E \rightarrow E + E \mid E * E \mid a \mid b$

Sol:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow a \mid b$

ما تبقى من الـ Grammar

Ex:

$E \rightarrow E + E \mid E - E \mid a \mid b$

Sol

$E \rightarrow E + F \mid E - F \mid F$

$F \rightarrow a \mid b$

Ex:

E → E + E | E - E | E * E | a | b

Sol:

E → E + F | E – F | F

F → F * R | R

R → a | b

Ex:

R → R + R | id | R - R | a | r z

Sol:

R → R + T | R – T | T

T → id | a | r z

Ex:

B → B ↑ B | a | b

Sol:

B → T ↑ B | T

T → a | b

Ex:

A → A * A | A ↑ A | A – A | A = A | a | id

Sol:

A → T = A | T

T → T – F | F

F → F * Z | Z

Z → R ↑ Z | R

R → a | id

Ex:

E → E + E | E * E | E – E | E / E | a | b

Sol:

E → E + T | E – T | T

T → T * F | T / F | F

F → a | b

H.W:

bExp → bExp OR bExp | bExp and bExp | not bExp | True | False

Sol:

bExp → bExp OR F | F

F → R and G | G

G → not bExp | True | False

ملاحظات:

1. الصيغة الثانية لمعرفة الـ Grammar أنه غامض هي: **NT → NT α | β NT**

لا بدَّ أن تتحقق الاثنين، مثل: E → E not | not E

2. أسبقيات العمليات المنطقية تكون بالشكل التالي (من الأعلى إلى الأسفل)

1. not
2. and
3. OR

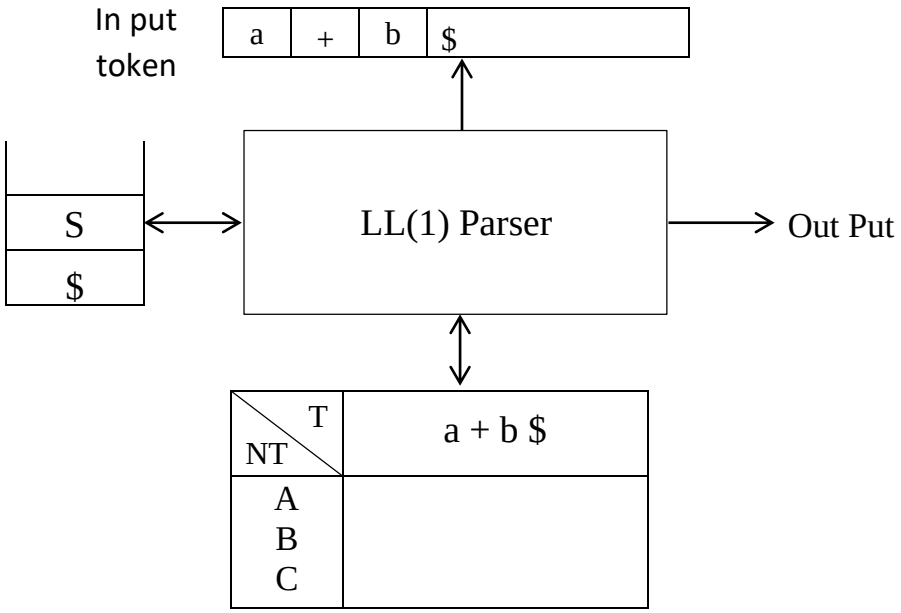
3. لا يمكن(**بمعنى غير منطقي**)أن يأتي Grammar حاوياً على عمليات حسابية وعمليات منطقية في آنٍ واحد (نفس الـ Grammar)

LL(1) التنبؤي

Predictive LL1

في هذا النوع يكون الإعراب (non-Recursive) أي بدون رجوع، ونحتاج لبناء جدول اعراب.

1) الـ (L) الأولى: تعني قراءة الـ input من اليسار إلى اليمين Left to Right.
2) الـ (L) الثانية: تعني الاشتقاق من اليسار Left most derivation.
3) الـ (1): تعني اعراب مقطع مفردة Token واحدة في كل مرة.



Model for LL1 Parser

المعرب LL1 يستخدم هياكل البيانات التالية:

1. in put buffer
2. Stack
3. Parsing table

بناء معرب تنبؤي من نوع LL(1)

Construction of Predictive LL1 Parser

لبناء معرب LL1 يجب أن نتبع الخطوات التالية:

1. Computation of first and follow function

(نحسب دالتي الـ first والـ follow)

2. Construct the predictive parsing table using first and follow functions.

(نبني جدول الاعراب –نملئ الجدول– باستخدام دالتي الـ first والـ follow).

3. Parse the input string with the help of predictive parsing table

(نعرب الجملة المطلوبة بمساعدة جدول الاعراب التنبؤي)

First function :

First(α):

هي مجموعة الرموز الغير قابلة للاشتقاق والتي تكون أول الرموز التي تظهر على جهة اليمين في اشتقاق الـ α.

القوانين المستخدمة لحساب دالة الـ First:

(The Rules are used to compute first function)

1. إذا كانت x رمز غير قابل للاشتقاق (terminal)، فإن:

First (X) = {x}

EX:

$E \rightarrow a$

Sol:

First (E) = {a}.

---

2. إذا كانت [$X \rightarrow \alpha \mid \lambda$] نفس القاعدة الأولى مضاف λ إلى الـ First(X):

EX:

$E \rightarrow a \mid \lambda$

Sol:

First (E) = {a, λ}.

3. إذا كانت [$X \rightarrow \alpha_1 \mid \alpha_2 \mid .. \mid \alpha_n$] فنأخذ أول ظهور للـ terminal في كل حد:

First (A) = First(α_1) ∪ First(α_2) ∪ ... First(α_n)

EX:

$S \rightarrow iEt \mid \lambda$

Sol:

First (S) = {i, λ}.

---

4. إذا كانت $\alpha = XYZ$ فإنَّ: First (α) يحسب بالشكل التالي:

- إذا كانت X لا تؤدي إلى λ بمعنى آخر First (X) لا يحتوي على λ فإنَّ:

First (α) = First (XYZ) ≈ First (X).

EX:

$S \rightarrow XYZ$

$X \rightarrow x$

$Y \rightarrow y$

$Z \rightarrow z$

Sol:

First (S) = {x}

- أما إذا كانت First (X) تحتوي على λ فإنَّ:

First (α) = First (XYZ) ≈ First (X) - {λ} ∪ First {XZ}

EX:

$S \rightarrow XYZ$

$X \rightarrow x \mid \lambda$

$Y \rightarrow y$

$Z \rightarrow z$

Sol:

First (S) = {x, y}

Ex1:

S → XYZ

X→ x | λ

Y→ y | λ

Z→ z

Sol:

First (S) = {x, y, z}

Ex2:

S → XYZ

X → x | λ

Y → y | λ

Z → z | λ

Sol:

First (S)= {x, y, z, λ}

Ex3: **Sol:**

S → iEtSS1 | a ⇒ First (S) = {i,a}

S1 → eS | λ ⇒ First (S1) = {e, λ }

E → b ⇒ First (E) = {b}

Ex4:

S → XYZ

X→ Y

Y→ y | λ

Z→ z | λ **Sol:**

First (S) = {y, z, λ}

First (X) = {y, λ}

First (Y) = {y, λ}

First (Z) = {z, λ}

Ex5:

S → XYZa

X→ Y

Y→ y | λ

Z → z | λ

First (S) = {y, z, a}

Ex6:

S → XaYZ

X→ Y

Y→ y | λ

Z→ z | λ

Sol:

First (S) = {y, a}

Ex7:

Find First Function for the following grammar:

G= ({E, E', T, T', F}, {+, *, (,), id}, E, P)

E → TE'

E' → +TE' | λ

T → FT'

T' → *FT' | λ

F → (E) | id

Sol:

Non−Terminal	First
E	{ (, id }
E'	{ +, λ }
T	{ (, id }
T'	{ *, λ }
F	{ (, id }

Follow Function:-

Follow (A):

هي مجموعة الرموز الغير قابلة للاشتقاق والتي تظهر مباشرة بعد A أي الـ NT في جهة اليمين من قاعدة الإنتاج.

القوانين المستخدمة لحساب دالة Follow:

(The Rules of computing follow function are as given below)

1. تضاف علامة ($) إلى Follow (S) حيث أن S هي رمز بداية القواعد.

Ex1:

$A \rightarrow aBz$

Sol:

Follow (A) = {$}

2. إذا كانت لدينا قاعدة انتاج على الشكل: $A \rightarrow \alpha B\beta$ فسوف تضاف الـ First(β) عدا الـ λ إلى الـ Follow(B).

Ex2:

$A \rightarrow aBz$

Sol:

Follow (B) = {z}

3. إذا كانت لدينا قاعدة انتاج بالشكل: $A \rightarrow \alpha B\beta$ وكان First(β) يحتوي على λ فإنَّ Follow(A) تضاف إلى Follow(B)

Ex3:

$A \rightarrow aBC$

$C \rightarrow d \mid \lambda$

Sol:

Follow (B) = {d, $}

ملاحظة: دالة الـ Follow لا يمكن ان تحتوي على λ

4. إذا كانت لدينا قاعدة انتاج بالشكل: $A \rightarrow \alpha B$ فإ ن: Follow(A) تضاف إلى Follow(B) حيث أ ن A و B هي رموز غير قابلة للاشتقاق.

Ex4:

$A \rightarrow aB$

Sol:

Follow (B) = {$}

Ex5:

Sol:

$S \rightarrow iEtSS1 \mid a \quad \Rightarrow$ Follow (S) = {$, e}

$S1 \rightarrow eS \mid \lambda \quad \Rightarrow$ Follow (S1) = {$, e}

$E \rightarrow b \quad \Rightarrow$ Follow (E) = {t}

Ex6:

Find Follow Function for the following grammar:

G= ({E, E', T, T', F}, {+, *, (,), id}, E, P)

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \lambda$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \lambda$

$F \rightarrow (E) \mid id$

Sol:

First	Follow
First(E) = { (, id }	Follow(E) = {$,)}
First(E') = { +, λ }	Follow(E') = {$,)}
First(T) = { (, id }	Follow(T) = {$, +,)}

First(T') = { *, λ }	Follow(T') = {$, +,)}
First(F) = { (, id }	Follow(F) = {*, +, $,)}

Ex7:

A → e | BCD

B → bEc | Cb | Dc | λ

C → c | g

D → a | Cb | λ

E → a | bEAe | λ

Sol:

First	Follow
First(A) = { e, b, c, g, a}	Follow(A) = {$, e}
First(B) = { b, c, g, a, λ }	Follow(B) = {c, g}
First(C) = { c, g }	Follow(C) = {a, c, g, b, $, e}
First(D) = { a, c, g, λ }	Follow(D) = {$, e, c}
First(E) = { a, b, λ }	Follow(E) = {c, e, b, g, a}

Q\ Find first and follow functions for following Grammar

S → A | BCD

A→ Be | EB

A→ bEc | Cb | Dc | λ

A→ c | λ D → a | Cb

E → a | bESe | λ

Sol:

First	Follow
First(S) = { b, c, a, e, λ }	Follow(S) = {$, e}
First(A) = { b, c, a, e, λ }	Follow(A) = {$, e}
First(B) = { b, c, a, λ }	Follow(B) = {c, a, b, e, $}
First(C) = { c, λ }	Follow(C) = {a, c, b}

First(D) = { a, c, b }	Follow(D) = {$, e, c}
First(E) = { a, b, λ }	Follow(E) = {b, c, a, e, $}

جدول الإعراب التنبؤي

Productive Parser table LL(1)

Algorithm (خوارزميات) for construct productive parser table:

1. **For each Productive A → ∝ of grammar do steps 2 and 3.**

كل قاعدة بالـ Grammar نطبق عليها الخطوة (2) والخطوة(3).

2. **For each terminal (a) first (∝) add**

A → ∝ to M [A, a].

كل حرف صغير First نضيف له (في الخلية التي في الجدول M التابعة أو التي تحت الحرف الصغير) القاعدة التالية: في .

$$A \rightarrow \propto$$

يعني الحرف الصغير من أي قاعدة ظهر بالـ First مثلاً من القاعدة (A→ a)، فنضيف القاعدة (A→ a) إلى الجدول M في الخلية التابعة للـ a.

3. If λ is in First (∝) add A → ∝ to M [A, b], For each terminal (b) in follow (A), If λ is First (∝) and $ in Follow (A), add A → ∝ to M[A, $].

إذا كان الـ First يحتوي على (λ) نضيف القاعدة التي نتجت منها الـ (λ) إلى الحرف التابع للـ First (مع القاعدة التي نتج منها الحرف الصغير في الـ First عدا الـ (λ)(ونضيف القاعدة أيضًا التي ظهر فيها الـ λ الى الـ Follow الخاص بالرمز الكبير.

EX: Construct (ابن) Productive Parsing table For Following grammar:

S → iEtSS1 | a

S1 → eS1 | λ

E → b

Sol:

1. نجد الـ First والـ Follow:

الرمز	First	Follow
S	i, a	$, e
S1	e, λ	$, e
E	b	t

2 . نكون جدول الإعراب:

NT \ T	i	t	a	e	b	$
S	$S \rightarrow iESS_1$		$S \rightarrow a$			
S1				$S1 \rightarrow eS1$ $S1 \rightarrow \lambda$		$S1 \rightarrow \lambda$
E					$E \rightarrow b$	

∴ Not LL(1) Grammar لوجود أكثر من قاعدة انتاج في خلية الجدول

ملاحظات:

* (λ) لا تهمل، وانما تكتب تحت الـ follow التابع لـ S1

* عندما تكون الخلية تحتوي على أكثر من قاعدة، فهذا يعني أن القواعد ليست من نوع (LL(1))، وذلك لأنها تحتوي على غموض وفي حال وجود قاعدة واحدة يعني أنها من نوع ((LL(1)).

Ex:

Construct (ابن) productive parsing table For Following grammar:

E → AB | e

A → a | λ

B → b | λ

Sol:

1. نجد الـ First والـ Follow:

الرمز	First	Follow
E	a, b, λ, e	$
A	a, λ	b, $
B	b, λ	$

2 . نكون جدول الإعراب:

NT \ T	e	a	b	$
E	E → e	E → AB	E → AB	E → AB
A		A → a	A → λ	A → λ
B			B → b	B → λ

∴ LL(1) Grammar

ملاحظات:

* نكرر القاعدة على حسب عدد عناصر الـ Follow.

* في حال وجود λ في القاعدة تكتب في مكان الـ $.

LL1 Grammar

تعتبر هذه القواعد جزء من قواعد السياق الحر (Context free Grammar)، ومعناها اعراب الجملة من اليسار إلى اليمين باستخدام الاشتقاق أقصى اليسار (Left most derivation) والرقم واحد يعني اعراب token مفردة واحدة في كل قاعدة ويتميز هذا النوع من القواعد بأنه كل خلية من خلايا الجدول Table cells المتولد من القواعد فيها قاعدة انتاج واحدة فقط، وبشرط في هذا النوع من القواعد أن يكون:

1. **Un ambiguous – غير غامض**
2. **No left recursion – لا يحتوي على تكرار ذاتي أيسر**
3. **left factoring – لا يحتوي على عامل مشترك**

ولمعرفة ذلك هناك شرطين إذا تحققا ذلك يعني أن القواعد Grammar هي من نوع LL1 وانها لا تعاني من الغموض والتكرار الذاتي الأيسر ومن العامل المشترك.

الشرط الأول: إذا كان في القواعد Grammar قاعدة انتاج نأخذ الشكل $A \rightarrow \alpha \mid \beta$:

ف إنَّ: $First(\alpha) \cap First(\beta) = \emptyset$

الشرط الثاني:

إذا كان $First(\beta)$ يحتوي على λ وكان $First(\alpha)$ لا يحتوي على λ سيكون الشرط:

فإنَّ: $First(\alpha) \cap Follow(A) = \emptyset$

Ex: $\overbrace{\alpha}$ $\overbrace{\beta}$

S → iEtSS1 | a

S1 → eS1 | λ

E → b

Sol:

1. نجد الـ First والـ Follow:

الرمز	First	Follow
S	i, a	$, e
S1	e, λ	$, e
E	b	t

1. $\{i\} \cap \{a\} = \emptyset$

2. $\{e\} \cap \{\$, e\} = \{e\} \neq \emptyset$

.Not LL1 Grammar إذًا

Ex:

Construct Parsing table for the following grammar:

A → cdB | λ

B → b | Ca

C → c

Sol:

1. نجد الـ First والـ Follow:

الرمز	First	Follow
A	c, λ	$
B	e, c	$
C	c	a

2. نكون جدول الإعراب:

NT \ T	c	d	b	a	$
A	A → cdB				A → λ
B	B → Ca		B → b		
C	C → c				

∴ LL(1) Grammar

نأخذ قواعد الـ T الموجودة بالـ First الـ grammar من النوع LL1 لأن كل خلية من خلايا الإنتاج لها قاعدة واحدة.

Ex:

Construct Productive Parsing table For Following grammar and show if this grammar is LL1 or not?

Note: إذا كان المطلوب "Show if LL1 or not" يكون بناء الجدول اختياري.!

E → TE'

E' → +TE' | λ

T → FT'

T' → *FT' | λ

F → (E) | id

Sol:

1. نجد الـ First والـ Follow:

الرمز	First	Follow
E	(, id	$,)
E'	+ , λ	$,)
T	(, id	+, $,)
T'	*, λ	+, $,)
F	(, id	*, +, $,)

2. نكون جدول الإعراب:

NT \ T	+	*	(	)	id	$
E			E → TE'	Synch	E → TE'	Synch
E'	E' → +TE'			E' → λ		E' → λ
T	Synch		T → FT'	Synch	T → FT'	Synch
T'	T' → λ	T' → *FT'		T' → λ		T' → λ
F	Synch	Synch	F → (E)	Synch	F → id	Synch

∴ **LL(1) Grammar**

ملاحظات:

* عندما لا توجد λ لا نأخذ الـ Follow، وعندما توجد λ نأخذ T(Terminal) الثانية لنفس الـ NT(Non Terminal) ونضع القاعدة فيهم.

* وجود أكثر من قاعدة إنتاج في خلية جدول الاعراب تسمى" **حالة تصادم**" Conflict.

* في حالة عدم وجود λ في الـ First نضع كلمة (Synch) تحت عناصر الـ Follow.

إعراب الجملة باستخدام الـ LL1:

Parse The Following String { (id) } using LL(1) Parser?

// تكملة الحل بالاعتماد على السؤال السابق:

Parse the string w=(id)$

Stack	Input buffer	Notes
$E	(id)$	
$E'T	(id)$	
$E'T	(id)$	
$E'T'F	(id)$	
$E'T')E(	(id)$	نلغي المتشابهات من آخر (Stack) وأول (input buffer)
$E'T')E	id)$	
$E'T')E'T	id)$	
$E'T')E'T'F	id)$	
$E'T')E'T'id	id)$	
$E'T')E'T'	)$	بما أنه {T' مع)} = ($T' \rightarrow \lambda$) نضع فراغ، ونأخذ الحرف الذي بعده.
$E'T')E'	)$	
$E'T'	$	POP
$E'	$	POP
$	$	Accepted

* عندما نصل إلى الـ ($) يكون قد انتهى.
* إذا كان تقاطع الرمز داخل الـ (Stack) مع الرمز داخل الـ (Input buffer) فراغ (يعني لا يوجد معادلة) فالجملة تكون خطأ، لا تنتمي للقاعدة، أما إذا كان التقاطع ناتجة منه (معادلة تؤدي إلى لمدا) نعمل Pop الكبير ومن ثم ننتقل للحرف الذي يليه.

تجاوز الخطأ في الاعراب التنبؤي

Error Recovery in LL1 Predictive Parsing

1.عندما يكون هناك خطأ عند الاعراب التنبؤي عندما يكون الـ Terminal الموجود أعلى المكدس Stack لا يتطابق مع رمز الادخال التالي.

2.عندما يكون الـ Non terminal الموجود أعلى المكدس Stack هو (A) ورمز الإدخال التالي هو (a) وكان ادخل الجدول للخلية M[A, a] تكون فارغة (يعني التقاطع ما بين المكدس والادخال فراغ

لتجاوز هذه الأخطاء يتم استخدام استراتيجية Panic mode والتي تستند على فكرة تجاوز الادخالات حيث يتم اختيار مجموعة عناصر (حروف) التزامن Synchronization symbol ويتم تحديد هذه المجموعة بوضع كلمة Synch تحت عناصر الـ Follow لـ (Non Terminal) NT الذي يكون الـ First التابعة له لا يحتوي على لمدا(λ).

1. إذا كان ناتج التقاطع ما بين رمز المكدس ورمز الادخال كلمة (Synch) نعمل POP للمكدس Stack ما عدا إذا كان هناك Start symbol.

2. إذا كان ناتج التقاطع ما بين رمز المكدس ورمز الادخال فراغ (Empty) نعمل تزحيف Shift Operation للإدخال Input buffer (بمعنى نترك الحرف وننتقل للحرف الذي يليه.)

3. إذا كان هناك T في أعلى المكدس stack لا يتطابق مع Terminal الموجود في الادخال input buffer نعمل POP للمكدس Stack.

- من السؤال السابق:

Parse the string w=)id*+id$

Stack	Input buffer	Notes
$E	)id*+id$	Error, Skip)
$E	id*+id$	نلغي المتشابهات من آخر (Stack) وأول (input buffer)
$E'T	id*+id$	
$E'T'F	id*+id$	
$E'T'id	id*+id$	
$E'T'	*+id$	
$E'T'F*	*+id$	
$E'T'F	+id$	Error, POP F
$E'T'	+id$	POP T'
$E'	+id$	
$E'T+	+id$	
$E'T	id$	POP T'
$E'TF'	id$	POP E'
$E'T id	id$	Accepted
$E'T	$	
$E'	$	
$	$	

H.W: id + id * id

Stack	Input buffer	Notes	Stack	Input buffer	Notes
$E	**id+id*id$**		**$E'T'id**	**id*id$**	
$E'T	**id+id*id$**		**$E'T'**	***id$**	
$E'T'F	**id+id*id$**		**$E'T'F***	***id$**	
$E'T'id	**id+id*id$**		**$E'T'F**	**id$**	
$E'T'	**+id*id$**	**POP**	**$E'T'id**	**id$**	

$E'	**+id*id$**		**$E'T'**	**$**	**POP**
$E'T+	**+id*id$**		**$E'**	**$**	**POP**
$E'T	**id*id$**		**$**	**$**	**Accepted**
$E'T'F	**id*id$**				

التحليل من الأسفل

Bottom up Parsing

الاعراب من الأسفل أو التحليل من الأسفل: يعرف على أنه محاولة لتقليص الجملة المدخلة (w) للوصول إلى زمن البداية للقواعد باستخدام الاشتقاق اقصى اليمين (RMD) للجملة وهذا مكافئ لإنشاء شجرة الاعراب للجملة المدخلة وذلك بالبدء من أوارق الشجرة والتقدم باتجاه قمة الشجرة. – ويتضمن البحث عن جزء من سلسلة الادخال يطابق جهة اليمين من قاعدة الإنتاج ثم يستبدل هذا الجزء من السلسلة بما هو موجود بجهة اليسار من قاعدة الإنتاج والذي يكون عبارة عن (NT) والذي هو عبارة عن عملية اشتقاق بالعكس.

Bottom up Parser Start from the sequence of terminal symbol and work their way back up to the start symbol by repeatedly replacing grammar rules right hand sides by the cores ponding Non−terminal in left hand side.

Ex:

E → E*E | E+E | id

w= id+id*id Sol:

E → E+E → E+E*E → E+E*id → E+id*id → id+id*id

Handle: هو المقطع الذي يطابق جهة اليمين من قاعدة الإنتاج في القواعد Grammar والذي يستبدل بالرمز المقابل للاشتقاق (NT) الموجود في جهة اليسار من نفس قاعدة الإنتاج.

Handle: is substring that matches a right hand side of production rule in the grammar and which reduction to the non-terminal on the Left hand side.

Notes:-

-There is a general style of bottom up Parsing known as shift Reduce Parsing.

-An easy to implement form of this Parsing Called Operator Precedence Parsing.

-A much more general method of shift-Reduce Parsing called LR Parsing.

الإعراب باستخدام

:Shift-Reduce

Shift-Reduce Method

هناك مشكلتين يجب أن تحل إذا أردنا أن نعرب جملة باستخدام التشذيب Handle Pruning:

1. تحديد الـ Handle
2. أي قاعدة انتاج سوف نختار إذا كان لدينا أكثر من قاعدة انتاج لها نفس الـ Handle، الطريقة المناسبة لتمثيل الـ Shift-Reduce Parsing هي استخدام Stack يوضع بداخله رموز القواعد وكذلك استخدام input buffer توضع بداخله الجملة المراد اعرابها وتستخدم علامة الـ $ لتؤشر على أسهل المكدس Stack وكذلك إلى نهاية الجملة المدخلة نضيف علامة $.

Shift Operation: عملية التزحيف

The next input symbol is a shifted into the top of stack.

Reduce Operation: عملية التقليص

Replace a set of grammar symbols to the top of stack with the LHS left hand side of Production Rule.

Ex)

Parse the input any (id+id*id) for grammar **using shift reduce method**?!

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

Sol:

Stack	Input buffer	Action
$	id + id * id $	Shift
$id	+ id * id $	Reduce by E → id
$E	+ id * id $	Shift
$E+	id * id $	Shift
$E+id	* id $	Reduce by E → id
$E+E	* id $	Shift
$E+E*	id $	Shift
$E+E*id	$	Reduce by E → id
$E+E*E	$	Reduce by E → E * E
$E+E	$	Reduce by E → E + E
$E	$	Accepted

ملاحظة:

المعرب يقوم بفحص المؤشر (Top) آخر عنصر بالمكدس (Stack) إذا كان يشبه أي Handle بجهة اليمين، سيعمل (Reduce) له (أي Pop لـ Handle) ثم (Push) لـ NT الموجود بجهة اليسار، أما إذا كان المؤشر (Top) يشبه آخر حرف من يمين الـ (Handle) (أي النهاية اليمنى لـ Handle) فسوف يقوم المعرب بإيجاد النهاية اليسرى لـ (Handle))أي عندما يكون (Handle) أكثر من رمز واحد.)

Q\ What is the Bottom Up Parsing used in the expression process?

Sol\ Uses Shift and Reduce

محلل أسبقية المشغل

Operator Precedence Parser

هو معرب من نوع (Shift–Reduce) يستخدم تقنية الاعراب من الأسفل إلى الأعلى ويستخدم مع فئة صغيرة من القواعد والتي تسمى قواعد المعاملات (Cooperator Grammar).

مميزاته:

- Using Bottom up Parsing Technique
- Parser is the Cooperator Grammar
- Is a shift reduction

*** لا تأخذ قواعده الشكلين التاليين:- -**

1.(A → λ) لا يحتوي على لمدا

2. A → BC∝ (قاعدة انتاج لا تؤدي إلى أكثر من حرف كبير متجاور)

Ex:

A→ Ba | a

B→ b

C→ c

D→ BA | d

Sol:

∴ Not Operator Grammar

Ex:

Show if the Following Grammar or an operator Grammar:

E→ EAE | (E) | -E | id

A → + | - | * | /

Sol:

∴ Not Operator Grammar

ليست من نوع قواعد المعاملات، لأنه يوجد أكثر من NT متجاورين، لكن إذا قمنا بتعويض عن الـ NT الذي هو (A) بما يقابله تصبح القواعد بالشكل التالي:

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid (E) \mid -E \mid id$

نلاحظ بعد التعويض لا يوجد أكثر من NT متجاورين، بهذه الحالة تصبح من نوع قواعد المعاملات.

ملاحظة:

يحتاج هذا المعرب إلى التعامل مع الأسبقيات بين المعاملات، وهناك ثلاثة أنواع من الأسبقيات:

1) a>.b 2) a<.b 3) a=.b

على أساس هذه الأسبقيات يقوم المعرب (Parser) بإعراب الجملة، وايضًا يقوم ببناء جدول الاعراب على أساس الاسبقيات، وليس على أساس الـ First والـ Follow.

المعرب من اليسار إلى اليمين

LR Parsing

أي الاعراب من اليسار إلى اليمين باستخدام اشتقاق اقصى اليمين يحتاج هذا المعرب في عمله إلى اوتوميتا محددة (DFA) ولبناء هذه الـ (DFA) نحتاج إلى نوع آخر من القواعد (Augmented Grammar)، وهي إذا كانت لدينا القواعد:

$$G = \{V, T, P, S\}$$

$$G = \{V \cup \{S\}, T, P \cup \{S1 \rightarrow S\}, S1\}$$

Define Augmented Grammar:-

هو نوع آخر من القواعد يستخدم في معرب LR لبناء الـ DFA، وهي إذا كانت لدينا القواعد: G = {V, T, P, S}، تكون بالشكل التالي: G = {V ∪ {S}, T, P ∪ {S1 → S}, S1}

Ex:

E → E + E

E → id

Sol:

E' → E

E → E + E

E → id

أي تم إضافة قاعدة انتاج جديدة، واعتبار (E') رمز البداية.

Q\ What is the use (ما الغاية من استخدام) of Augmented Grammar?

Sol\ In building (لبناء) DFA

هناك ثلاث تقنيات لـ LR Parsing:

LALR	CLR(1) = LR(1)	SLR(1)
1. مختصر لـ: (Took Ahead LR(1))	1. مختصر لـ: (Canonical LR(1))	1. مختصر لـ: (Simple LR(1))
2. يعمل على حجم متوسط من القواعد.	2. يعمل على مجموعة كاملة من القواعد.	2. يعمل على أصناف صغيرة من القواعد.
3. يولد نفس عدد الحالات (States) في المعرب ((LR(1)).	3. يولد عدد كثير من الحالات (States)، لهذا جدول الاعراب يكون كبير ومعقد.	3. عدد الحالات (States) يكون قليل، لهذا جدول الاعراب يكون صغير.
4. بناءه متوسط التعقيد.	4. بناءه معقد وبسيط.	4. بناءه بسيط وسريع.
5. متوسط الفعالية، والكلفة له تكون أقل من الـ ((LR(1))، والـ ((SLR(1)).	5. أعلى كلفة، وأكثر فعالية.	5. أقل كلفة وأقل فعالية.
6. يستخدم LR(1) Item لبناء الـ DFA.	6. يستخدم LR(1) Item لبناء الـ DFA.	6. يستخدم LR(0) Item لبناء الـ DFA.

قلنا سابقا أنَّ هذا النوع (LR Parsing) يعتمد على Automata (DFA) وهذه الـ DFA تتكون من عدد محدود من الحالات ولإيجاد تلك الحالات نتبع ما يلي:

1(نحول القواعد إلى (add mended grammar)

2(نوجد LR(0) Item والتي تسمى أيضًا بعملية Closure (1) Operation

لإيجاد LR(0) Item يكون ذلك من خلال إضافة نقطة إلى الـ Grammar وبعدها نقوم بتزحيف هذه النقطة من اقصى اليسار إلى اليمين، إنَّ هذا التأشير بواسطة النقطة يعطي تسمية مختلفة لذلك العنصر (Item) ومنها:

1. Kernel Item: all items whose dots are not at the beginning of RHS (Right Hand Side) plus the augment initial item S' → .S

جميع العناصر التي تكون نقاطها في بداية جهة اليمين من قاعدة الإنتاج إضافة إلى initial item، مثل:

S → A.a

S → Aa.

2. Non kernel: all items with dots at the beginning of RHS except S' → .S

جميع العناصر التي تقع النقطة فيها بعد السهم مباشرةً أو بداية الطرف الأيمن، ما عدا S' → S. (النقطة في بداية لكن ما تعتبر Kemal) initial item

3. Completed item: all items with dots at the end of RHS, X → ab.

هي كل العناصر التي تقع النقاط في أقصى اليمين من الطرف الأيمن أو نهاية الجملة القواعدية.

LL(0)

Ex: Find LR(0) item for following grammar:

$S \rightarrow AA$ ……. (1)

$A \rightarrow aA$ …….. (2)

$A \rightarrow b$ …….. (3)

Sol:

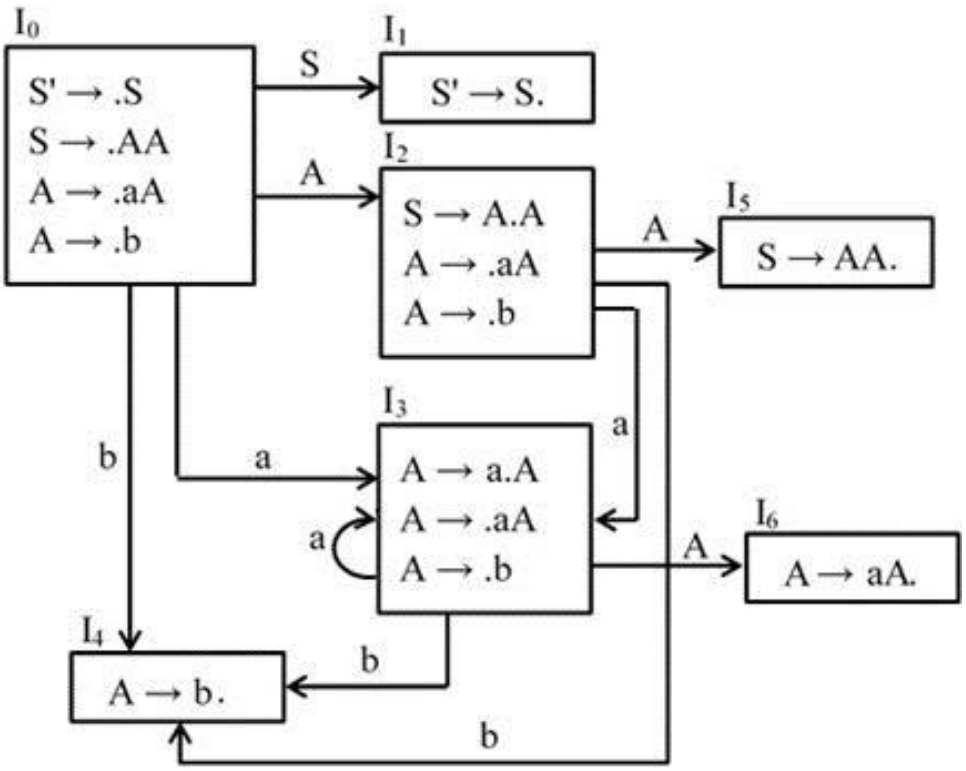
$S' \rightarrow S$

$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

ملاحظة: عندما تأتي Non-terminal بعد النقطة نستمر بإضافة القواعد الخاصة لها (نستمر بالاشتقاق بالاعتماد على القواعد الأصلية الثانية بعد الترقيم(واذا كان Terminal نتوقف، وتكون إضافة القواعد من الـ grammar وليس من التكرار السابق، لأن التكرار لا يعتمد على التكرار الذي قبله أبدًا أبدًا.



Note: هناك طريقة أخرى لإيجاد LR(0) item وهي استخدام دالة Go to Function(للاطلاع.)

I_0:-

S' → .S

S → .AA

A → .aA

A → .b

I_1:-

Go to (I0, S)

S' → S

I_2:-

Go to (I0, A)

S → .AA

A → .aA

A → .b

I_3:-

Go to (I0, a)

A → a.A

A → .aA

A → .b

I_4:-

Go to (I0, b)

A → b.

I_5:-

Go to (I2, A)

S → AA.

I_3: Go to (I2, a)

I_4:-

Go to (I2, b)

State عدد حالات الـ DFA	Action table			Goto table	
	a	b	$	S	A

0	S3	S4		1	2
1	.		Accept		
2	S3	S4			5
3	S3	S4	r3		6
4	r3	r3	r1		
5	r1	r1	r2		
6	r2	r2			

بناء جدول اعراب للمثال السابق

ملاحظة:

دائمًا عندما تكون هناك هذه الحالة المنتهية .S'→S، تكتب Accept تحت الـ $، ضمن التكرار الأول ($I_1$) والحالة التي تكون بلا أسهم تكتب Reduce (r) تحت حقل الـ Terminal جميعها، ونأخذ رقم الـ (r) من ترقيم القواعد الأصلية بداية السؤال.

حالات التصادم في الـ LR Parsing:

Confliction in LR Parsing

1.Shift-reduce conflict: occurs when the grammar indicates that different successful parses might either occur with a shift or reduce of a given point during parsing.

(يكون هناك Shift (S) و Reduce (r) في نفس الخلية)المربع ضمن الجدول.)

2.Reduce-reduce conflict: occurs when the parser has tow or more handles at the same time on top of stack.

(يكون هناك Reduce (r) و Reduce (r) في نفس الخلية)المربع ضمن الجدول)

س/ كيف يمكن معرفة هل أنَّ الـ Grammar هو LR(0) أم لا؟

ج/ بعد اشتقاق القواعد للـ Item وايجاد الحالات الخاصة بالـ DFA إذا كان هناك أكثر من Item منتهية (Complete) يكون هناك تصادم من نوع reduce–reduce، مثل:

	a	b
3	r1 r2	r1 r2

A' → b.

A → aA.

وإذا كان هناك أكثر من item احدها منتهية Complete، والأخرى لا..

LR(0)Parser

LR(1) Parser, كلاهما يستخدمان نفس ال Item

سوف يكون هناك تصادم من نوع Shift- reduce , مثال

x → a.

y → .b

a	b
r1	r1 S4

Ex: Do the following rules of type LR(0) or not?

E → T + E ……. (1)

E → T …….…… (2)

T → T * F ……. (3)

T → F …… (4)

F → id ……... (5)

E' → E

E → T + E

E → T

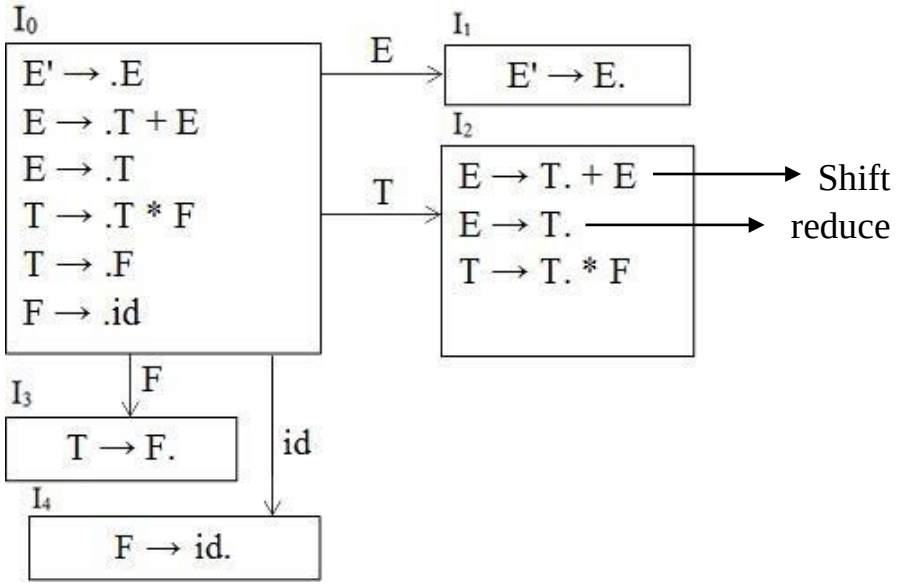
T → T * F

T → F

F → id

Sol:

نقوم بتحويل القواعد إلى (Augmented):



∴ Not LR(0) Grammar.

SLR(1) Parser (Simple LR Parser)

هناك أربع خطوات لبناء معرب من نوع SLR(1):

1. Find first and follow function
2. Find set of item (closure I)
3. Construct Parsing
4. Parse the input

ملاحظة: الـ item الـ SLR(1) هي نفسها المستخدمة في LR(0) ولكن الفرق يكون عند بناء الجدول لـ LR(0)، نضع الـ reduce (r) لجميع الـ terminal الموجودة في الجدول، أما عند بناء

الجدول لـ SLR(1) نضع الـ reduce (r) تحت العناصر الموجودة في الـ follow الخاصة بالـ non terminal الموجودة أقصى يسار قاعدة الإنتاج.

Ex: Consider the following Grammar, find SLR and Draw a DFA that represent transition diagram.

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Sol:

1)) $E \rightarrow E + T$

2)) $E \rightarrow T$

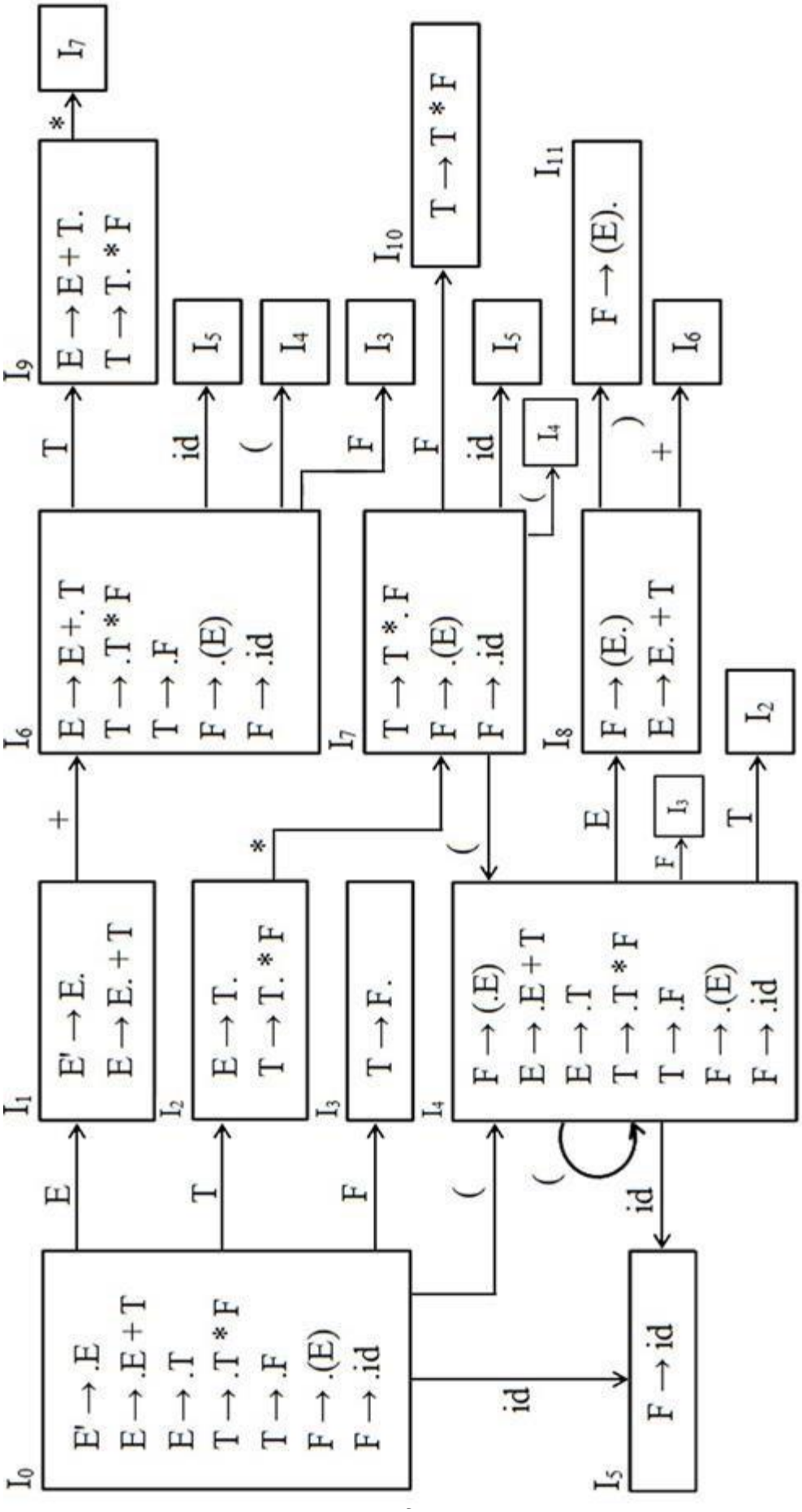
3)) $T \rightarrow T * F$

4)) $T \rightarrow F$

5)) $F \rightarrow (E)$

6)) $F \rightarrow id$

Non−Terminal	First	Follow
E	(, id	$, + ,)
T	(, id	$, + ,) , *
F	(, id	$, + ,) , *


I0
E' → .E
E → .E + T
E → .T
T → .T * F
T → .F
F → .(E)
F → .id
E
T
F
(
id
I1
E' → E.
E → E. + T
+
I2
E → T.
T → T. * F
*
I3
T → F.
I4
F → (.E)
E → .E + T
E → .T
T → .T * F
T → .F
F → .(E)
F → .id
(
id
I5
F → id
I6
E → E + .T
T → .T * F
T → .F
F → .(E)
F → .id
I7
T → T * .F
F → .(E)
F → .id
I8
F → (E.)
E → E. + T
I9
E → E + T.
T → T. * F
*
I10
T → T * F
I11
F → (E).
T
id
(
F
+
)
I7

- بناء جدول الإعراب للمثال السابق:

نبني جدول اعراب، وذلك لأن المعرب يعرب على أساس الجدول.

State	Action table						Goto table		
	id	+	*	(	)	**$**	**E**	**T**	**F**
0	S_5			S_4			1	2	3
1		S_6				Accept			
2		r_2	S_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		r_1	S_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

س/ كيف يمكن معرفة هل القواعد من نوع ((SLR(1)) أم لا؟!

ج/ الـ Item لـ SLR(1) نجدها بنفس طريقة LR(0) لكن إذا كانت القواعد ليست LR(0) فهذا لا يعني أنها ليست SLR(1)، ولمعرفة أن القواعد من نوع SLR(1) يجب أن يتحقق الشرطان الآتيان:

1) Follow (A) ∩ Follow (B) = ∅

إذا كان هناك أكثر من Item منتهية فالـ State أي تصادم من نوع (reduce – reduce) نطبق الشرط.

$A \rightarrow x.$

$B \rightarrow y.$ Follow (A) ∩ Follow (B) = ∅

2) Follow (A) ∩ First ($\alpha 2$) = ∅

إذا كان هناك Item منتهية وأخرى غير منتهية، أي تصادم من نوع (shift – reduce) نطبق الشرط.

$A \rightarrow x.$

$B \rightarrow \alpha 1.\ \alpha 2$ Follow (A) ∩ First ($\alpha 2$) = ∅

ملاحظة1: القواعد من نوع SLR(1) نضع رمز (r) تحت عناصر الـ Follow لذلك يجب أن لا يكون هناك عملية Shift لأي رمز من رموز الـ Follow لـ A.

ملاحظة2: إذا لم يتحقق الشرطان أعلاه فهذا يعني أنَّ القواعد ليست من نوع SLR(1).

ملاحظة3: دائمًا الـ 'S نكتب داخل الجدول (Accept) بحقل الـ $.

Show if the following Grammar are SLR(1) grammar or not?!

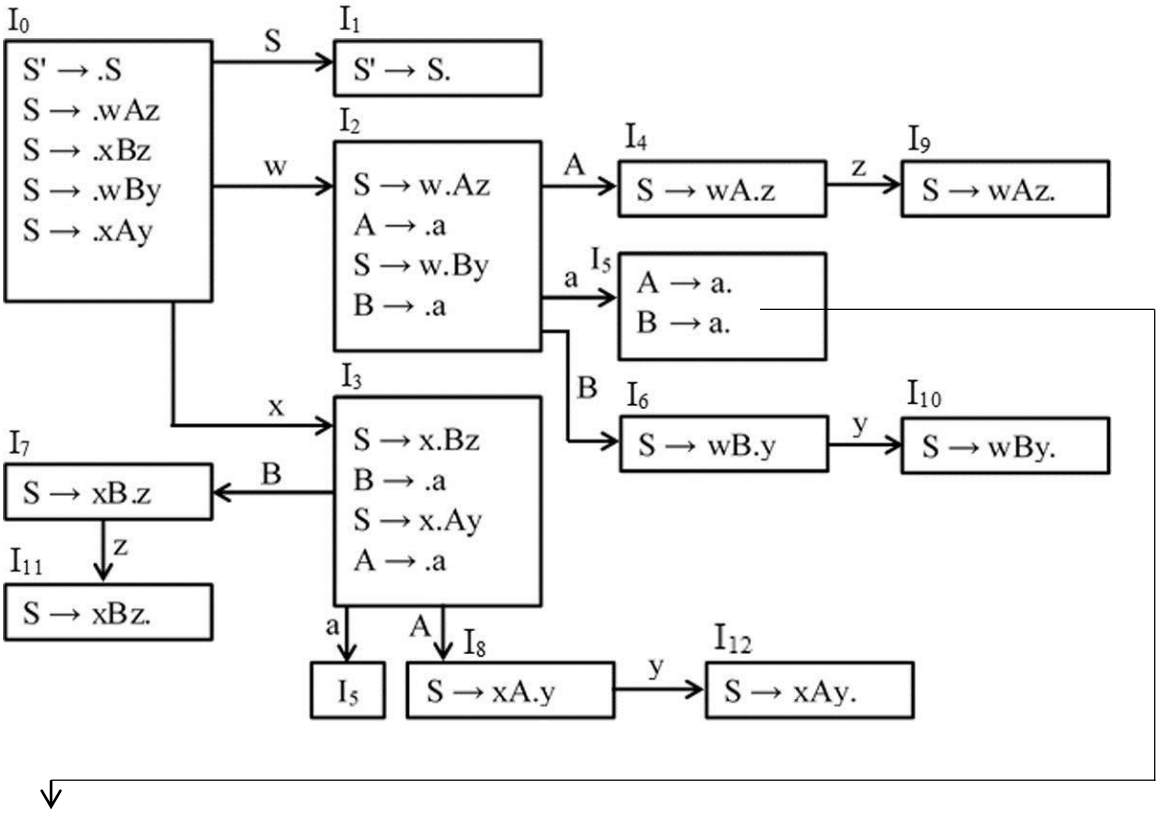
S → wAz A → a

S → xBz B → a

S → wBy

S → xAy

Sol:



Follow(A) ∩ Follow(B) = [y, z] ∩ [z, y] ≠ ∅

∴ not SLR(1)

Construct DFA for following Grammar, and Show if it is a LR(0) grammar or not?!

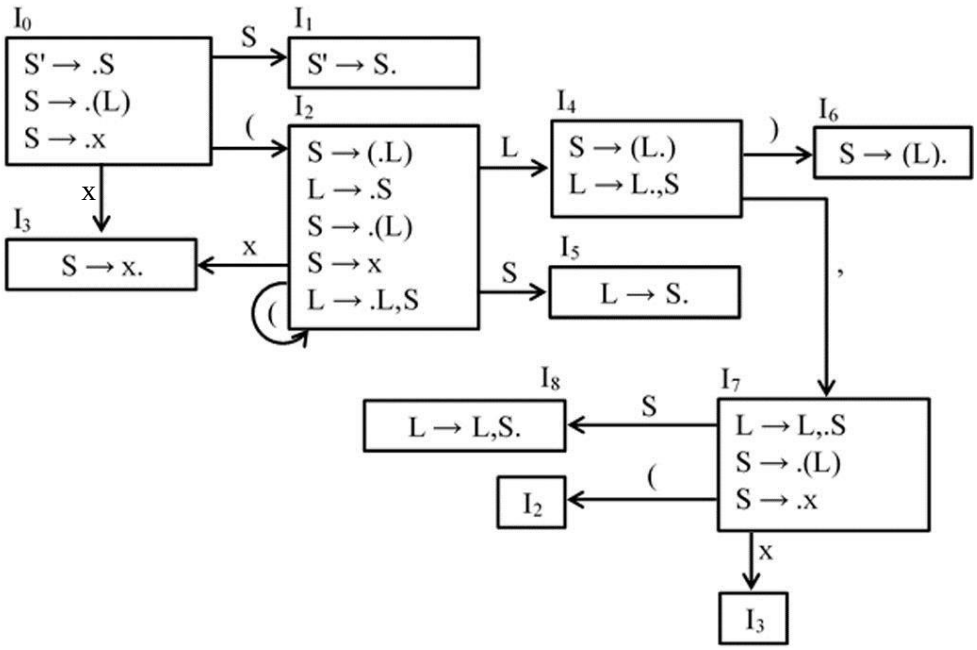
$S \rightarrow (L)$

$S \rightarrow x$

$L \rightarrow S$

$L \rightarrow L,S$

Sol:



State	Action table					Goto table	
	(	)	,	x	$	S	L
0	S_2			S_3		1	
1					Accept		
2	S_2			S_3		5	4
3	r_2	r_2	r_2	r_2	r_2		
4		S_6	S_7				
5	r_3	r_3	r_3	r_3	r_3		
6	r_1	r_1	r_1	r_1	r_1		
7	S_2			S_3		8	
8	r_4	r_4	r_4	r_4	r_4		

Construct DFA for following Grammar, and Show if it is a SLR(1) grammar or not?!

(1) $E \rightarrow T + E$

(2) $E \rightarrow T$

(3) $T \rightarrow int * T$

(4) $T \rightarrow int$

(5) $T \rightarrow (E)$

$E' \rightarrow E$

$E \rightarrow T + E$

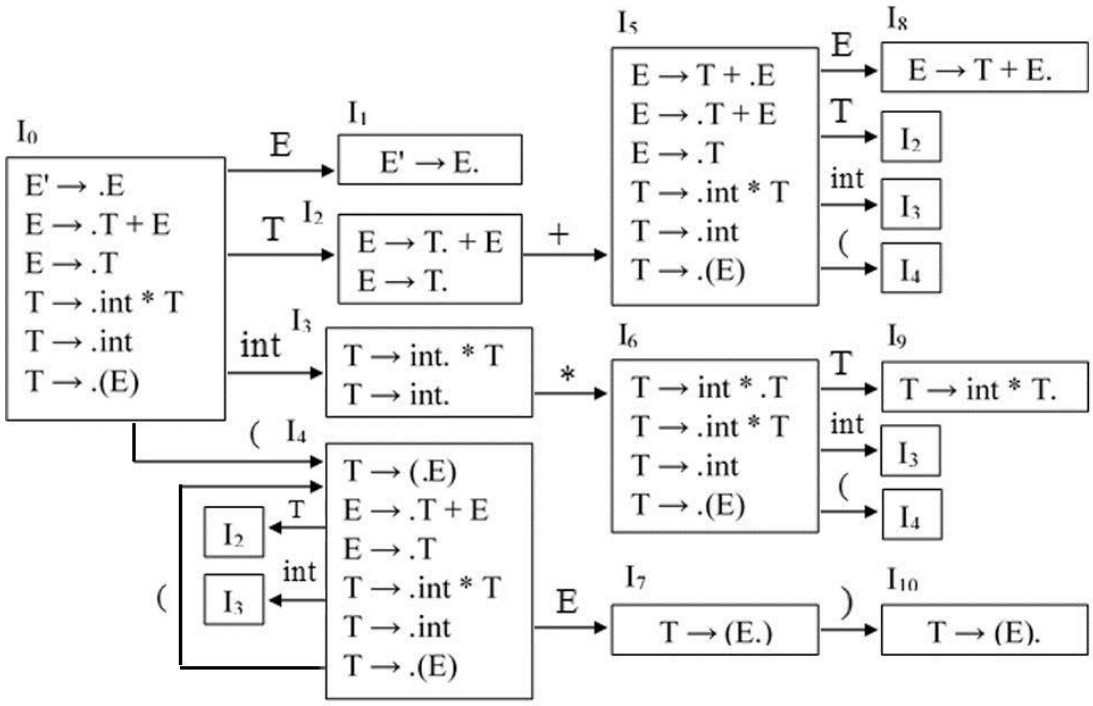
$E \rightarrow T$

$T \rightarrow int * T$

$T \rightarrow int$

$T \rightarrow (E)$

Sol:



Follow(E) = { $,) }

Follow(T) = { +, $,)}

I2 = Follow(E) ∩ First(∝2) = { $,) } ∩ { + } = Ø .. إذًا نُكمل الحل

I3 = First(∝2) ∩ Follow(T) = { * } ∩ { +, $,) } = Ø .. إذًا نُكمل الحل

ملاحظة: عندما يكون هناك تصادم من نوع (Shift-Reduce) نأخذ عناصر الـ Follow للـ Reduce، ونقاطعها مع عناصر الـ First للـ Shift دائمًا، حسب القاعدة المذكورة سابقا، وتكون عناصر الـ Follow للـ Reduce للـ non-terminal الموجود أقصى اليسار، أما عناصر الـ First للـ Shift تكون ما بعد النقطة ضمن قاعدة الإنتاج نفسها، إذا كان حرف صغير نأخذه وحده مباشرةً، أما إذا كان حرف كبير نأخذ عناصر الـ First جميعها لذلك الحرف الكبير.

الآن نبني جدول إعراب:

State	Action table						Goto table	
	+	int	*	(	(	$	E	T
0		S3		S4			1	2
1						Accept		
2	S5				r2	r2		
3	r4		S6		r4	r4		
4		S3		S4			7	2
5		S3		S4			8	2
6		S3		S4				9
7					S10			
8					r1	r1		
9	r3				r3	r3		
10	r5				r5	r5		

Q\ Draw DFA for the following Grammar, and Construct crossword table type of SLR(1) for the first five States:

1))

E → Aa | b

A → BzBe

B → eT

T → t

2))

E → F * B

F → T + a

T → t

B → E – b | b

LR(1) Parser (CLR(1))

في هذا النوع من الاعراب تكون هناك معلومات أكثر من الحالات (States)، الـ Item المتكونة تتكون من جزء إضافي عبارة عن Terminal يسمى (Look ahead) لتصبح صيغة الـ Item، كالآتي:

$$A \rightarrow \alpha . B\beta , x$$

وعند حدوث عملية Reduce، نقوم بوضع (r) تحت عناصر الـ Look ahead التابع لذلك الـ Item، وكيفية كتابة الـ Look ahead تكون بالشكل التالي:

طريقة إيجاد الـ Look ahead:

أولاً: نحول القواعد إلى Augment

ثانيًا: أول قاعدة يكون الـ Look ahead هو علامة ($).

ثالثًا: بعد كتابة القواعد الأخرى ضمن الحالة، يكون الـ Look ahead لكل قاعدة هو الرمز الموجود ما بعد الحرف الذي نشأت منه القاعدة السابقة في تلك الحالة **حصرًا**، وهذا يكون على ثلاث حالات:

a) حرف صغير نأخذه مباشرة

b) حرف كبير نأخذ الـ First له

c) فراغ، نأخذ الـ Look ahead

نأخذ أمثلة لتوضيح ما قلنا:

Ex1:

S → AA

A → aA | b

Sol:

S' → S

S → AA

A → aA

A → b

I_0:

S' → .S, $

S → .AA, $

A → .aA, a | b

A → .b, a | b

Ex2:

S → V = E

E → F | E + F

F → V | int | (E)

V → id

Sol:

S' → S

S → V = E

E → F | E + F

F → V | int | (E)

V → id

I_0:

S' → .S, $

S → .V = E, $

V → .id, =

Ex3:

S → XX

X → aX | b

Sol:

S' → S

S → XX

X → aX

X → b

I_0:

S' → .S, $

S → .XX, $

X → .aX, a | b

X → .b, a | b

Ex: Find LR(1) item for following grammar:

S → CC

C → cC | d

Sol:

1. Find First Function:

- First (S) = [c, d]
  First (C) =[c,d]

2. Find augmented grammar:

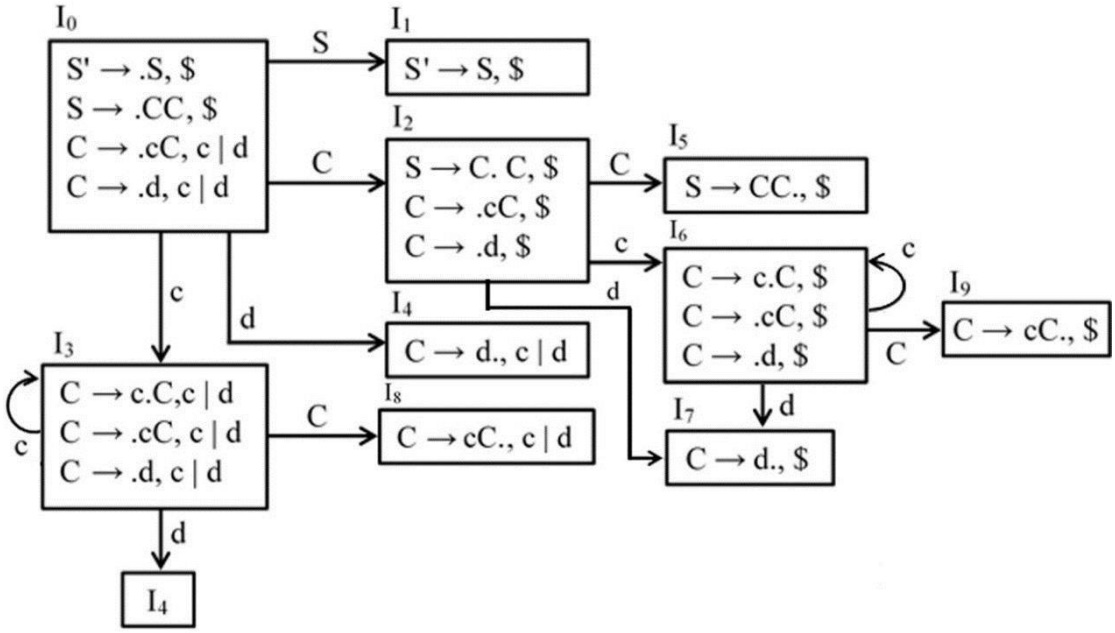
S' → S

S → CC

C → cC

C → d

3. Find LR(1) Items:



4. Construct DFA:

State	Action table			Goto table	
	c	d	$	S	C
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

Ex: Find LR(1) item for following grammar:

$E \rightarrow T + E \mid T$

$T \rightarrow id$

Sol:

1. Find First Function

- First (E) = [id]
- First (T) = [id]

2. Find augmented grammar:

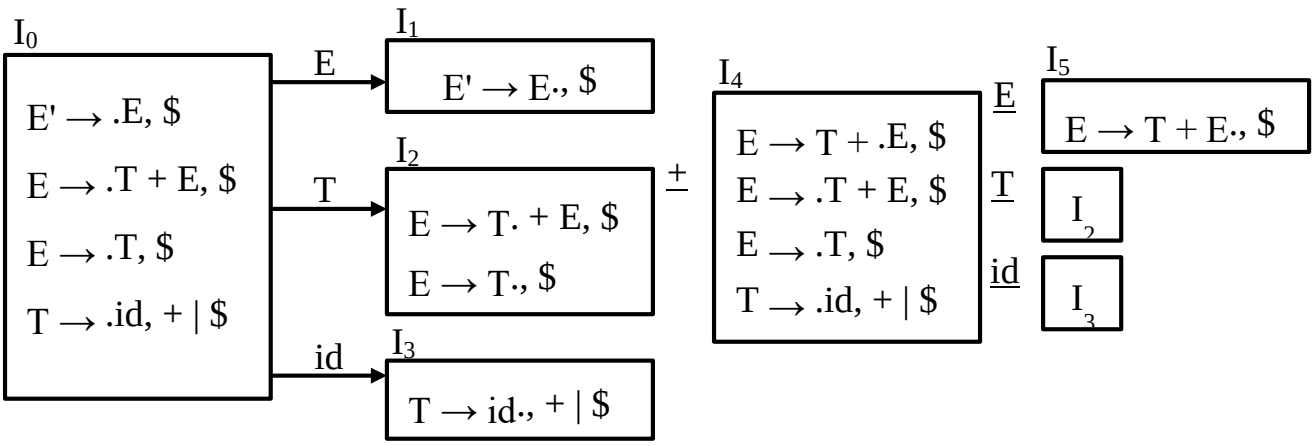
$E' \rightarrow E$

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow id$

3. LR(1) Items:



4. Construct DFA:

State	Action table			Goto table	
	+	id	$	E	T
0		S3		1	2
1			Accept		
2	S4		r2		
3	r3		r3		
4		S3		5	2
5			r1		

Ex: Find LR(1) item for following grammar:

S → AA

A → aA |b

Sol:

1. Find First Function

- First (S) = [a, b]
- First (A) = [a, b]

2. Find augmented grammar:

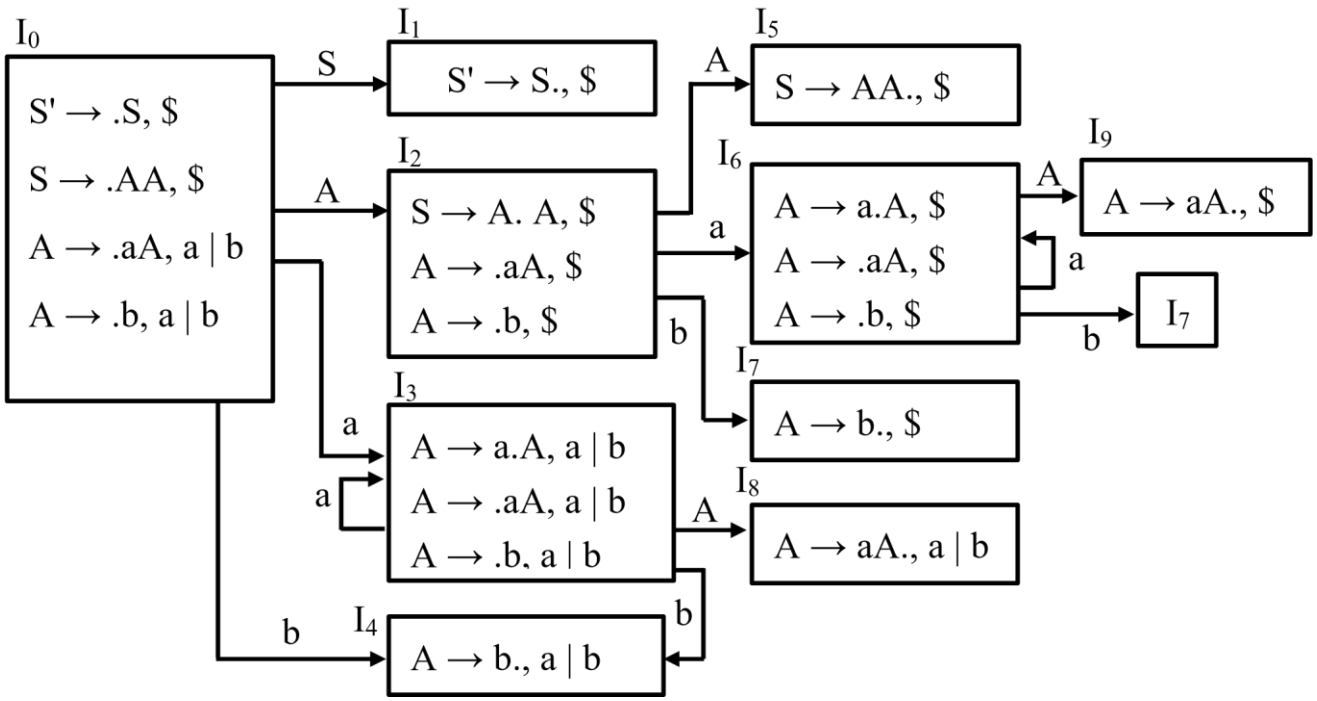
S' → S

S → AA

A → aA

A → b

3. LR(1) Items:

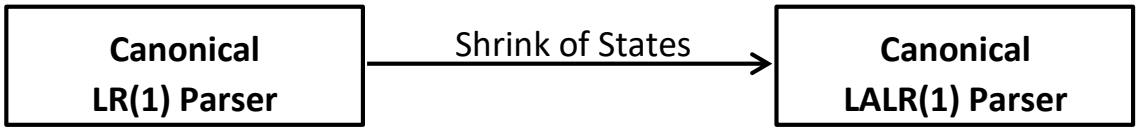


4.Construct DFA:

State	Action table			Goto table	
	a	b	$	S	A
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

LALR(1) Parser

هو بناء جدول اعراب من نوع LALR(1) ويكون بناءه بنفس خطوات بناء جدول LR(1)، أي باستخدام الـ Look ahead، ولكن طريقة بناء الـ LALR(1) ستكون باختصار عدد من الحالات (States) وتقوم بدمجها مع الحالات المشابهة لها، وتصبح بذلك كأنها حالة واحدة، وتهمل الحالة الثانية، مع مراعاة جمع الـ Look ahead لكلا الحالتين مع بعض، المثال التالي يوضح كيفية اختصار الحالات ودمجها:



I_1: L → id., =
I_2: L → id., $

A New State

I_{12}: L → id., =
L → id., $

OR:

I_{12}: L → id., = | $

ولبناء جدول اعراب من نوع LALR(1):

1. نقوم بإيجاد الـ First & Follow
2. نقوم بتحويل القواعد إلى Augmented
3. نقوم بإيجاد حالات الـ LR(1)
4. نقوم باختصار عدد من الحالات ونقوم بدمجها مع الحالات المشابهة لها، مع مراعاة جمع الـ Look ahead، واهمال الحالة الثانية.
5. بناء جدول من نوع LALR(1) بالاعتماد على الحالات بعد الاختصار والدمج.

Ex: Find LALR(1) item for following grammar:

S → CC

C → cC | d

Sol:

1. Find First Function:

- First (S) = [c, d]
- First (C) = [c, d]

2. Find augmented grammar:

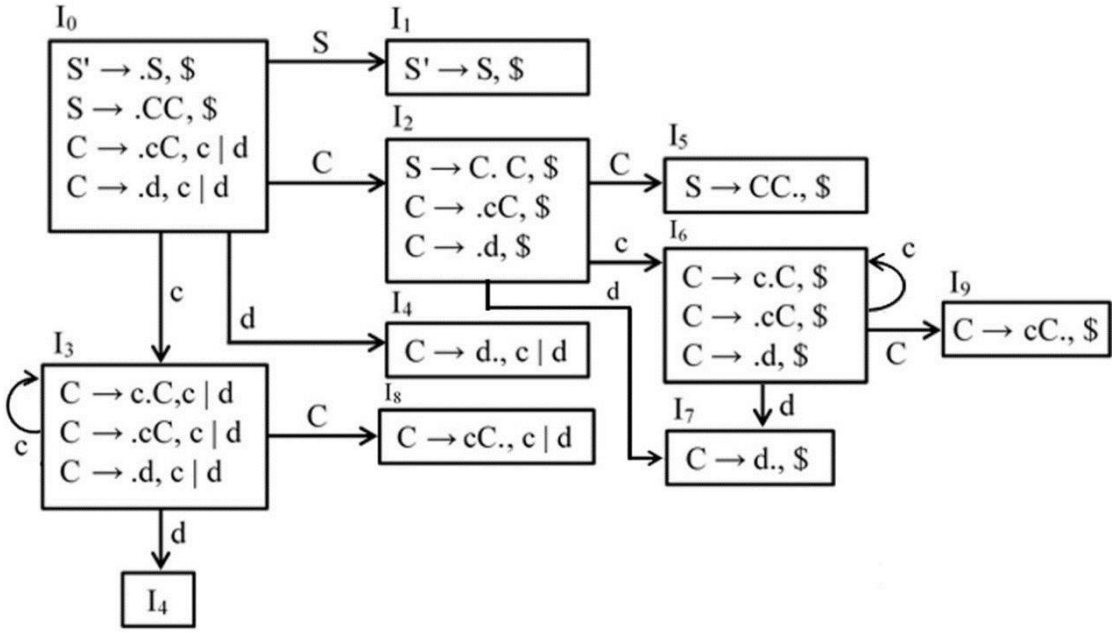
S' → S

S → CC

C → cC

C → d

3. Find LR(1) Items:



4. Shorten a number of cases and merge them with similar situations:

$I_3 + I_6 \equiv I_{36}$ = C → c.C, c | d | $
C → .cC, c | d | $
| C → .d, c | d | $

$I_4 + I_7 \equiv I_{47}$ = C → d., c | d | $

$I_8 + I_9 \equiv I_{89}$ = C → cC., c | d | $

ملاحظة:

عند بناء جدول DFA التابع لـ LALR(1)، نقوم بإهمال الحالات (6)، (7)، (9)، فيكون الترقيم كما موضَّح بالجدول أدناه:

5. Construct DFA:

State	Action table			Goto table	
	c	d	$	S	C
0	S36	S47		1	2
1			Accept		
2	S36	S47			5
36	S36	S47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		9

Ex: Find LALR(1) item for following grammar:

S → L = R | R

L → * R | id

R → L

Sol:

1. Find First Function:

- First (S) = [*, id]
- First (L) = [*, id]
- First (R) = [*, id]

2. Find augmented grammar:

S' → S

S → L = R

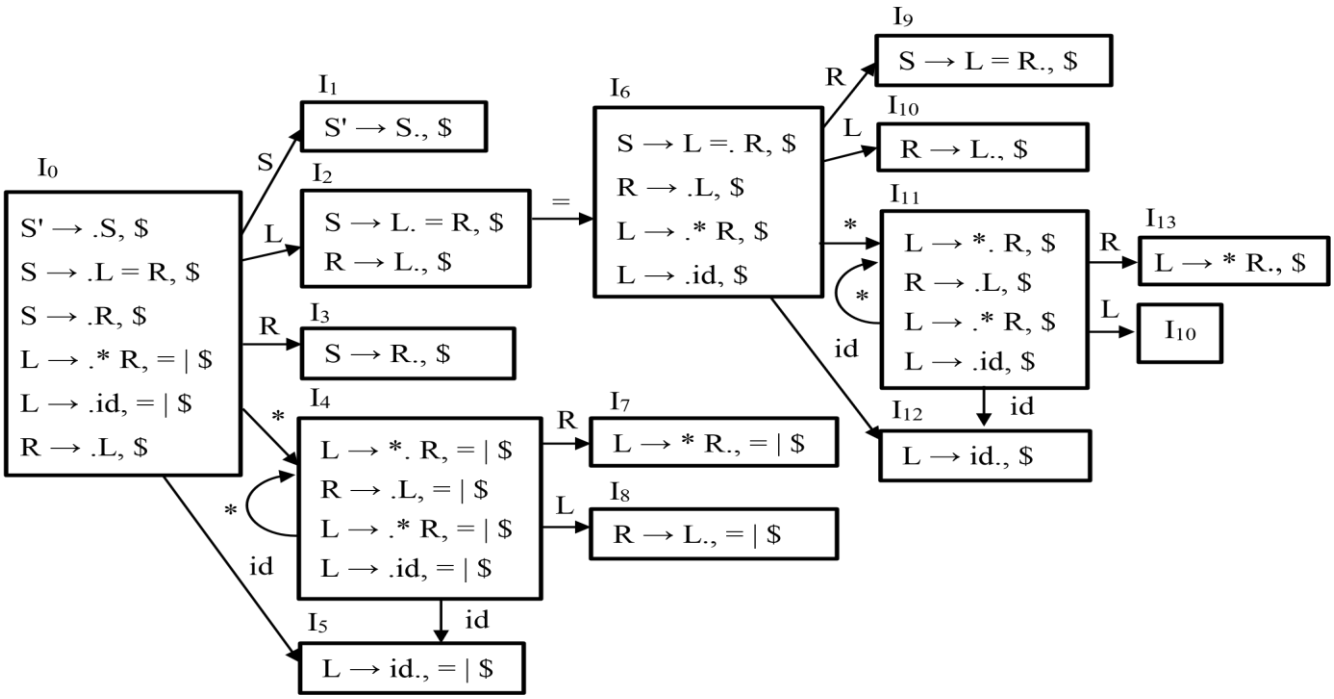
S → R

L → * R

L → id

R → L

3. Find LR(1) Items:



4. Shorten a number of cases and merge them with similar situations:

$I_4 + I_{11} \equiv \mathbf{I_{411}} = L \rightarrow *.\ R, = \mid \$$

$R \rightarrow .L, = \mid \$$

$L \rightarrow .*\ R, = \mid \$$

$L \rightarrow .id, = \mid \$$

$I_5 + I_{12} \equiv \mathbf{I_{512}} = L \rightarrow id., = \mid \$$

$I_7 + I_{13} \equiv \mathbf{I_{713}} = L \rightarrow *\ R., = \mid \$$

$I_8 + I_{10} \equiv \mathbf{I_{810}} = R \rightarrow L., = \mid \$$ **ملاحظة:**

عند بناء جدول DFA التابع لـ LALR(1)، نقوم بإهمال الحالات (11)، (12)، (13)، (10) فيكون الترقيم كما موضَّح بالجدول أدناه:

5. Construct DFA:

State	Action table				oto table		
	id	*	=	$	S	L	R
0	S512	S411			1	2	3
1				Accept			
2			S6	r5			
3				r2			
411	S512	S411				8 10	7 13
512			r4	r4			
6	S512	S411				8 10	9
713			r3	r3			
810			r5	r5			
9				r1			

أخطاء الاستعادة في LR

Error Recovering in LR Parsing

غالبًا ما يطرأ على جدول الإعراب أخطاء الاستعادة (الاستيراد)، ولتجاوز أخطاء الإعراب LR من اليسار إلى اليمين، فإذا كان هناك حقل فارغ في جدول الاعراب فسيقوم معرب LR بإعلان وجود خطأ، ولحل هذه المشكلة يتم ملئ كل حقل فارغ بمؤشر يستدعي "Error Routine" والذي سوف يتخذ الاجراء المناسب، وهذا الاجراء قد يكون ادراج رموز أو حذف رموز من المكدس (Stack) أو الـ (Input Buffer) أو من كليهما حسب تصميم المترجم.

Ex:

W= (id + id)$

E → E + T | T

T → T * F | F

F → (E) | id

Sol:

E → E + T
E → T
T → T * F
T → F
F → (E)
F → id

ملاحظات:

1. Shift

لما يكون عندي Shift:

أعمل Push للـ Terminal الموجود بالـ Input buffer، مع Push لرقم الحالة.

2. Reduce

لما يكون عندي Reduce:

أعمل POP بطول جهة اليمين* 2، ثم أعمل Push لجهة اليسار من القاعدة نفسها، ثم أقارن هذا الحرف الذي عملت له Push مع الرقم الذي قبله، يظهر رقم الحالة.

1)) E → E + T

2)) E → T

3)) T → T * F

4)) T → F

5)) F → (E)

6)) F → id

State	Action table						Goto table		
	id	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				Accept			
2		r_2	S_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		r_1	S_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

Line	Stack	Input Buffer	Action
1	0	(id + id)$	Shift
2	0 (4	id + id)$	Shift
3	0 (4 id 5	+ id)$	Reduce by F → id
4	0(4 F3	+ id)$	Reduce by T → F
5	0(4 T2	+ id)$	Reduce by E → T
6	0(4 E8	+ id)$	Shift
7	0(4 E8+6	id)$	
8	0(4 E8+6 id5	)$	Reduce by F → id
9	0(4 E8+6 F3	)$	Reduce by T → F
10	0(4 E8+6 T9	)$	Reduce by E → E+T
11	0(4 E8	)$	Shift
12	0(4 E8) 11	$	Reduce by F → (E)
13	0F3	$	Reduce by T → F
14	0T2	$	Reduce by E → T
15	0E1	$	Accept

بما أنه ظهر (Accept)، هذا يعني أنَّ الجملة مقبولة ضمن جدول الاعراب