بناء المترجم باستخدام مولدات المحلل اللفظي والقواعدي

الباحثة :م. هبه عبدالباسط تدمري

كلية الهندسة المعلوماتية – جامعة البعث

الملخص:

بناء المترجم هو نظام متطور جداً نظراً لوجود تاريخ طويل له مدعوم بنظريات علمية ومجموعة كبيرة من الكتب، حيث يحتوي على مئات الآلاف إن لم يكن الملايين من أسطر الشيفرة، ولذلك فإنّ عملية تصميمه وبناءه هو أمر أساسي في هندسة البرمجيات تمر مراحل دورة بناء المترجم بمحطات عديدة ابتداء من المحلل اللفظي وانتهاء بتوليد البرنامج الهدف حيث يمكن كتابة هذه المراحل يدوياً أو يمكن الاعتماد على بعض الأدوات لكتابتها. نحاول أن نتناول في هذا البحث الأدوات المساعدة لبناء المترجم وخصوصاً مولدات المحلل اللفظي ومولدات المحلل القواعدي، وكما سنوضح البنية العامة لها وكيفية استخدامها ومدى توفيرها للوقت والجهد.

الكلمات المفتاحية :

المترجم – المحلل اللفظي –المحلل القواعدي– مولدات – الوحدة اللفظية – شجرة الأعراب المجردة– التعابير المنتظمة –القواعد.

Build Compiler using Lexical and Parser generators

Abstract :

Building the compiler is a very advanced system due to its long history supported by scientific theories and a large collection of books, as it contains hundreds of thousands if not millions of lines of code, and therefore the process of designing and building it is essential in software engineering. The stages of the compiler building cycle pass through many stations Starting with the lexical analyzer and ending with generating the target program, where these stages can be written manually or some tools can be relied upon to write them. In this research, we try to deal with the auxiliary tools for building the translator, especially the lexical analyzer and Syntax analyzer generators, and we will explain the general structure of them, how to use them, and the extent to which they save time and effort.

Key words:

Compiler – Lexical Analyzer - Syntax analyzer - generators - Token - Abstract syntactic tree (AST)- the regular expressions - the grammar.

1-المُقدّمة:

تعتبر لغة الآلة لغة الحاسب الأساسية فهي تتعامل مع الحاسب مباشرة من مسجلات و معاملات مرتبطة بشكل وثيق بالآلة نفسها، ويعتبر البرنامج المكتوب بلغة الآلة ما هو إلا تسلسل من الواحدات والأصفار، لذا محاولة برمجة خوارزمية معقدة نوعاً ما باستخدام لغة الآلة يعتبر أمراً معقداً مع وجود إمكانية كبيرة للأخطاء. نظراً لصعوبة البرمجة باستخدام لغة الآلة فقد تم التفكير بخلق بيئة برمجية أعلى مستوى، تُمكّن المبرمج من كتابة نص برمجي بشكل مفهوم بالنسبة له بدلاً من التفكير بطريقة الآلة، هذه البيئة البرمجية تسمى عادة بالمترجم، لذا يعتبر بناء المترجم من التطبيقات المستخدمة على نطاق واسع في هندسة البرمجيات.

إنه لمن الصعب تحديد أول مترجم خرج إلى الحياة وذلك بسبب وجود عدد كبير من فرق البحث التي تعمل ضمن هذا المجال إلا أن أول المحاولات كانت لإيجاد مترجم يحول المعادلات الرياضية إلى ترميز الآلة.

في عام(1950) ظهر أول مترجم فورتران(Fortran) واحتاج إلى جهد عدد كبير من الأشخاص ولسنوات عديدة[7]، فيما بعد ظهرت أدوات تساعد المبرمج على بناء المترجم من خلال لغات توصيفية تبعده عن الأخطاء وهدر الوقت في اكتشافها وتصحيحها.

1- هدف البحث ومبرراته:

يعتبر المترجم الجسر الذي يربط بين لغات البرمجة والبنية الصلبة للحاسب بمعنى هو الجسر الذي يربط نظام التشغيل بالتطبيقات ولغات البرمجة المختلفة.

هذا البحث يغطي فعليا:

1) توضيح بنية المترجم العامة وخصوصاً مرحلة المحلل اللفظي والقواعدي.
2) دراسة تفصيلية لأدوات المساعدة لبناء المحلل اللفظي والقواعدي (Flex&Bison) وتوضيح مدى أهميتها في توفير الجهد والوقت عند بناء أي مترجم.
3) دراسة عملية نتناول فيها لغة تجريبية كلغة مصدر ونحاول تحقيق المحلل

اللفظي والقواعدي يدوياً ومقارنتها مع طريقة تحقيقها باستخدام الأدوات المساعدة وإظهار حل لمشكلة حالات الغموض التي يمكن أن نقع فيها أثناء بناء الشجرة.

3-المترجم:

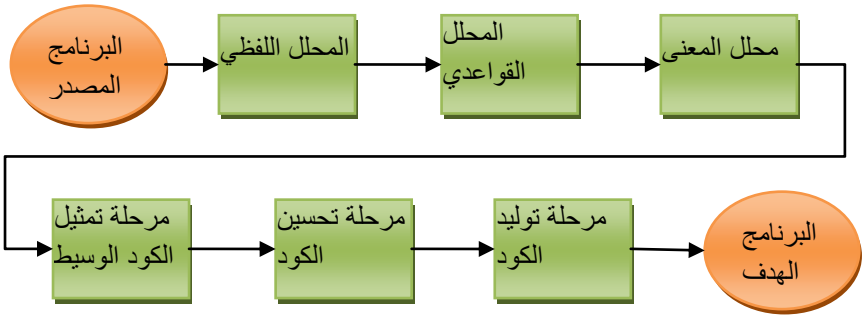
هو برنامج يساعدك على تحويل البرنامج المكتوب بلغة عالية المستوى إلى برنامج مكتوب بلغة منخفضة المستوى مثل لغة الآلة، فهو يترجم الشيفرة بدون أن يغير معنى هذه الشيفرة بحيث نحصل على شيفرة فعالة بأفضل زمن تنفيذ واشغال ذاكرة.

لكن هناك مترجمات تقوم بالترجمة بين اللغات عالية المستوى كأن تقوم مثلاً بترجمة شيفرة مكتوب بلغة(C++) إلى أخرى مكتوبة بلغة الجافا.[1,9]

الشكل(1) آلية عمل المترجم

4-البنية الأساسية للمترجم :

يتبع تصميم المترجم طريقة شهيرة في تصميم البرمجيات تعتمد على تقسيم البرمجية إلى مراحل بدلاً من التفكير الغرضي التوجه وهذه المراحل ضعيفة الترابط فيما بينها، وخرج كل مرحلة هو دخل المرحلة التالية، سنوضح في هذا البحث آلية بناء مرحلتي المحلل اللفظي والقواعدي وشرح الأدوات اللازمة لبنائهما.[9]

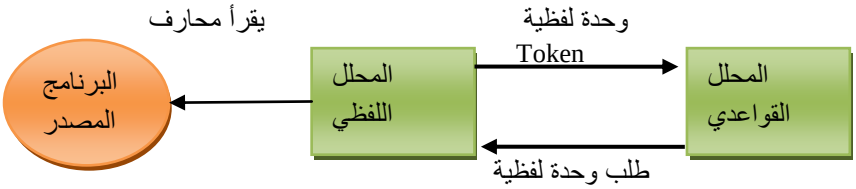


الشكل(2) مراحل المترجم

يتألف المترجم من المراحل التالية:

4-1 المحلل اللفظي (Lexical Analyzer):

هو أول مرحلة من مراحل المترجم حيث يتم فيه قراءة الملف كاملاً وتجميع المحارف مع بعضها لتشكل وحدات لفظية (Token)، مع إهمال الفراغات والتعليقات والأحرف البيضاء، يتمر إرسال هذه الوحدة اللفظية إلى المحلل القواعدي، طبعاً من يقوم بدفع المحلل اللفظي للعمل هو المحلل القواعدي.[6]



الشكل(3) المحلل اللفظي

4-1-2 الوحدة اللفظية :

هي تتالي المحارف، يمكن معاملتها كوحدة متكاملة (أو كلمة)، وكل وحدة لفظية لها زوج من المعلومات (نوع–قيمة (lexem)) ويمكن أن تصنف في إحدى المجموعات التالية:

- الكلمات المفتاحية .
- المتحولات.
- الأرقام.
- الرموز.

4-1-3 آلية عمل المحلل اللفظي :

- تقسيم البرنامج إلى وحدات لفظية.
- حذف الفراغات
- حذف التعليقات.
- يساعد في توليد معلومات عن رسائل الخطأ مثل رقم السطر والعمود

مثال : إذا كان لدينا الدخل التالي : int x=5;

يقوم المحلل اللفظي بتحديد نوع وقيمة كل وحدة لفظية.[5]

نوع الوحدة اللللفظية Token-Type	قيمة الوحدة اللفظية Lexem
Int_Token	Int
Id_Token	X
Assign_token	=
Integer_Token	Num

الجدول (1) الوحدات اللفظية للدخل

4-2 المحلل القواعدي (Syntax Analyzer)

أو يدعى أحياناً (Parser)،هو المرحلة الثانية من مراحل المترجم يقوم بالتحقق من صحة قواعد اللغة المدروسة، بحيث يتم تجميع الوحدات اللفظية ضمن جمل بمعنى مطابقة رموز القواعد القادمة من الدخل مع قواعد اللغة المبنية أي التأكد من أن البرنامج صحيح من الناحية القواعدية، كما يقوم بتشكيل شجرة الأعراب كخرج (AST : Abstract Syntax Tree) عقد هذه الشجرة تمثل رموز القواعد وأوراق الشجرة النهائية هي الوحدات اللفظية والانتقالات في الشجرة هي خطوات الاشتقاق للانتقال بين القواعد.[8]

لماذا نحتاج المحلل القواعدي ؟

- التحقق من أن الشيفرة صحيحة قواعدياً.
- يساعد بمطابقة القواعد مع البرنامج المدخل.
  - يطابق كل قوس مفتوح أن له قوس مغلق.
  - كل عملية تصريح يجب أن يكون لها نمط وهذا النمط معتمد من قبل اللغة.

آلية عمل المحلل القواعدي تعتمد على النموذج القواعدي خارج السياق الذي يعرف

التعابير وترتيب ظهورها.[3]

لبناء الشجرة لدينا حالتين:

1) الإعراب من الأعلى للأسفل: حيث يتم بناء الشجرة من الأعلى للأسفل.

2) الإعراب من الأسفل للأعلى : حيث يبدأ المحلل ببناء الشجرة من الدخل ويحاول بناءها وصولا لرمز البداية أي بشكل بديهي يحاول المحلل القواعدي تحديد العناصر الأساسية ، ثم العناصر التي تحتوي عليها ، وما إلى ذلك. مثال عنها محللات (LR) كما يستخدم مصطلح لمثل هذا النوع من المحللات هو Shift–Reduce .

هذا النوع من المحللات يحتاج إلى (buffer) لتخزين سلسلة الدخل ومكدس لتخزين القاعدة الذي يقوم بمعالجتها مع وجود ثلاث عمليات أساسية يقوم فيها

1) الإزاحة (shift) وهي ازاحة الرمز من (buffer) إلى المكدس.

2) تبديل (Reduce) وهي المطابقة بين حالة المكدس والطرف اليمين للقواعد واستبدالها بالطرف اليساري للقواعد.

3) مقبول(Accept) إذا بقي في المكدس رمز البداية فقط و(Buffer) الإدخال فارغ معناها الدخل مقبول.

4)الخطأ(Error): تحدث هذه الحالة عندما يعجز المحلل عن القيام بعملية الإزاحة أو التبديل. [11]

4-3 محلل المعنى (Semantic Analyzer)

يقوم بفحص مضمون الشجرة القواعدية و يتحقق من صحتها من حيث المضمون، تكشف هذه المرحلة الأخطاء مثل جمع عدد صحيح مع سلسة نصية فكما نلاحظ أن هذا الخطأ لن يتم كشفه في مرحلة التحليل القواعدي، كما تقوم هذه المرحلة بتتبع حالة الرموز المعرفة من قبل المستخدم (من متحولات وتوابع و..) .[4]

4-4 التمثيل الوسيط (Intermediate Code Representation):

يتم في هذه المرحلة توليد الكود المرحلي (IR) من شجرة الإعراب بعدة طرق أشهرها (3 Address Code) أما في اللغات الشهيرة مثل الجافا ولغات إطار (Net.) يتم استخدام كود وسيط لآلة افتراضية تعتمد على مكدس واحد، في الحقيقة الفائدة الأساسية من هذه المرحلة تكمن في تسهيل تعزيز محمولية المترجم.[10]

4-5 تحسين الكود الوسيط:

يتم في هذه المرحلة تحسين الكود الوسيط مثلاً اختصار بعض التعليمات وهو من أصعب المراحل وعلى الكود المحسن تحقيق خاصيتين مهمتين :الأولى أن يكافئ البرنامج الأصلي تماماً، والثانية أن يكون أفضل من البرنامج الأصلي من حيث استهلاك الذاكرة وسرعة التنفيذ.[4]

4-6 توليد الكود الهدف:

يتم في هذه المرحلة توليد الكود النهائي تبعا للمنصة التي نريد للكود العمل عليه .

في الحقيقة إن المراحل 4 و 5 قد لا تكون موجودة في بعض المترجمات، كما أن بعض الكتب تفضل تقسيم المترجم إلى طرف أمامي (front-end) و طرف خلفي (back-end)

والسبب وراء هذا التقسيم أن مراحل الطرف الأمامي غير مرتبطة بمنصة عمل أو نظام تشغيل معين فإذا أردت مثلاً كتابة مترجم يعمل على عدة أنظمة تشغيل يكفي أن تقوم بتغيير الطرف الخلفي في كل نظام[9].

4-7 جدول الرموز :

يحتاج المترجم إلى معلومات عن الأسماء التي تظهر في برنامج المصدر، حيث أنّ هذه المعلومات يتم تخزينها في بنية معطيات تسمّى جدول الرموز Symbol Table، إنّ كل سطر في جدول الرموز هو عبارة عن ثنائيّة (اسم،معلومات)، و في كل مرّة يتم فيها إدخال اسم فإنّه يجب إجراء بحث في جدول الرموز لمعرفة فيما إذا كان الاسم المُدخل

موجود سابقاً أم لا، فإذا كان الاسم جديداً يتم إدخاله إلى الجدول، و يتم إدخال المعلومات المتعلّقة بهذا الاسم إلى الجدول خلال فترة التحليلين اللّفظي و القواعدي.[10]

إنّ المعلومات المجمّعة في جدول الرموز يمكن أن تستخدم خلال مراحل متعدّدة من العمل، فمثلاً يتم استخدام هذه المعلومات في مرحلة تحليل المعنى و ذلك للتأكّد من التوافق بين التصريح عن الاسم و نمط هذا الاسم.

كما يتم استخدام المعلومات الموجودة في جدول الرموز أيضاً في مرحلة توليد الشيفرة Code Generation لمعرفة كميّة و نوعيّة الذاكرة التي يجب أن تخصّص للاسم.[5]

عادة في المترجم، تكون الأسماء الموجودة في جدول الرموز تشير إلى أغراض Objects من أنواع مختلفة ، و من الممكن أن توجد جداول منفصلة لكل من :

أسماء المتحوّلات ، أسماء الاجراءات ، الثوابت ، ... و أنماط أخرى من الأسماء تبعاً للّغة.

إن تشكيل جدول الرموز ممكن و ذلك اعتماداً على إحدى بنى المعطيات :

1) اللوائح الخطيّة :وهي بطيئة في الولوج لكنّها سهلة التحقيق و التطبيق.
2) البنية الشجريّة: تعطي استجابة متوسّطة.[9]

ولا ننسى أنه في كل مرحلة من مراحل المترجم سينتج عنه أخطاء لذا لدينا ما يسمى (Error Handling) لكل مترجم وعملية تحديد موقع الخطأ تقع على عاتق المحلل اللفظي فهو من يقوم بتحديد رقم السطر والعمود، لكن لن نغطي في هذا البحث معالجة الأخطاء وإنما تركزينا على بناء المترجم باستخدام الأدوات المساعدة.[10],[8]

5-أدوات بناء المترجم:

تم تقديم أدوات بناء المترجم مع انتشار التقنيات البرمجية عالمياً وهي تعرف أيضا باسم (مترجم-المترجمات) (مولدات المترجم)، تستخدم لغات أو خوارزمية محددة لتحديد وتنفيذ مكونات المترجم فيما يلي أمثلة عن أدوات بناء المترجم:

- المولدات اللفظية (Scanner generators) : يكون مدخلات هذه الأدوات عينات تعتمد على التعابير المنتظمة (regular expressions)، على سبيل المثال LEX لنظام التشغيل Unix .
- مولدات المحلل اللغوي: يكون دخلها عبارة عن قواعد وتقوم اتوماتيكياً بتوليد الكود البرمجي لهذه القواعد يمكنها تحليل الوحدات اللفظية بمساعدة هذه القواعد.
- مولدات الكود الاتوماتيكية تؤخذ الكود (الشيفرة) الوسطى وتحولها الى لغة الآلة.

كما ذكرنا سابقاً سنركز في بحثنا على مولدات المحلل اللفظي والقواعدي. [8]

1-5 مولدات المحلل اللفظي :

وهي برامج مصممة لتوليد المحللات اللفظية(Scanner)، تقوم بتمييز العينات اللفظية في البرنامج وهي مصمم بالبداية للأنظمة المعتمدة على اليونكس طوّر شفرته كلاً من (Eric Schmidtو Mike Lesk).[7,2]

إن الغرض الأساسي منها هو تسهيل عملية التحليل اللفظي التي تعتمد على مبدأ معالجة تتالي المحارف من البرنامج المصدر لإنتاج سلسلة موافقة من الرموز تدعى الوحدات اللفظية التي تستخدم بدورها كدخل لبرامج أخرى مثل (Parser).[7]

تعتبر برامج (lex & yacc) برامج لتوليد برامج المحللات اللفظية و القواعدية مكتوبة بلغة (C) هناك نسخة مطورة عنهم هم (Flex &Bison) سنلقي الضوء عليهما في هذا البحث.

- **أداة توليد المحلل اللفظي السريع**

Flex(Fast Lexical Analyzer Generator)

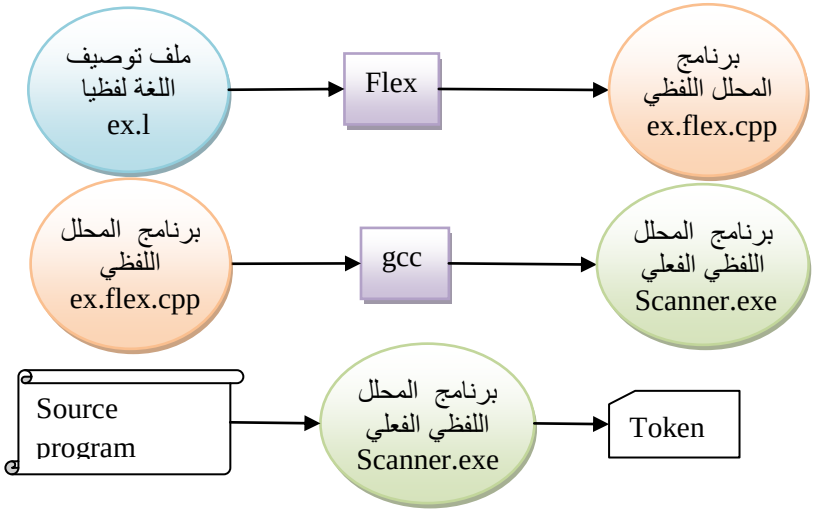
مولد المحلل اللفظي السريع هي أداة لتوليد المحلل اللفظي صممت من قبل (Vern Paxson) بلغة (C) في عام1987 يمكن استخدامها مع مولدات المحلل القواعدي (Yacc, Bison) لكن سنوضح في بحثنا هذا الأدوات (Flex & Bison) لأنها أحدث من النسخة (Lex &Yacc).[7]

- **آلية عمل Flex:**

الخطوة الأولى : ملف دخل يوصف المحلل اللفظي للغة المصدر مكتب بلغة (flex) أي تكون لاحقته(l.*) يُدخل إلى أداة (Flex) فتقوم بتحويل هذه الملف إلى برنامج المحلل اللفظي مكتوب بلغة (C\C++) يدعى عادة (flex.cpp.*).

الخطوة الثانية : نقوم بإدخال الملف الناتج عن الخطوة السابقة إلى مترجم لغة (C\C++) لينتج عنه ملف تنفيذي.

الخطوة الثالثة : هذه الملف التنفيذي هو المحلل اللفظي الفعلي حيث يأخذ كدخل سلسلة محارف وخرجه الوحدات اللفظية (Token).[5]



الشكل(4) آلية عمل Flex

- **بنية ملف(l.*):**

يتألف ملف(flex) من ثلاث أقسام رئيسية يتم فصل كل قسم عن الآخر بالإشارة (%%) بحيث يجب وضعها على سطر منفرد.[3]

```
{Definitions}
%%
{Rules}
%%
{User subroutines}
```

1. قسم التصريحات :

وهي قسمان :

a. تصريحات لغة C يتم احاطتحها بالرمز (%{)

b. تصريحات Flex.

```
%{
//c++ decleration
%}
Flex decleration
%%
```

2. قسم القواعد:

القسم الأساسي يحوي سلسلة من القواعد بحيث كل قاعدة تتألف من عينة(Pattern) يجب مطابقتها مع وحدة لفظية تكتب بالاعتماد على التعابير النظامية، وحدث موافق لهذه العينة عبارة عن شيفرة بلغة(C++) كما يلي:

```
%%
pattern1 { action1 }
pattern2 { action2 }
…
pattern_i  { actionn }
%%
```

مع ملاحظة أنه يجب وضع كل عينة على سطر، ويجب فصل بين العينة وحدثها بفراغ واحد على الأقل، ويجب أن لانضع فراغ قبل العينة إطلاقاً،عند مطابقة الدخل يتم المقارنة بين المحارف المدخلة مع العينات المكتوبة في ملف التوصيف وتحقيق الحدث المقابل للعينة المطابقة، يوضح الجدول التالي بعض العينات التي يمكن مطابقتها.[7]

العينة	يمكن مطابقتها
[0-9]	كل الأرقام بين العددين صفر وتسعة
[0+9]	إما 9 أو اشارة + أو العدد0
[0, 9]	إما 9 أو إشارة الفاصلة أو العدد0
[0 9]	إما 9 فراغ أو العدد0
[-09]	أما إشارة الطرح- او 9 او 0

أما إشارة الطرح- أو الأرقام بين الصفر والتسعة	[-0-9]
عدد صحيح من أكثر بخانة وحدة أو أكثر	[0-9]+
كل المحارف ماعدا المحرفa	[^a]
كل المحارف ما عدا الأحرف الأبجدية بحالة الأحرف الكبيرة	[^A-Z]
أي محرف	.
تكرار حرف a أو عدم تكراره	a*
تكرار الحرف a مرة وحدة على الأقل	a+
كل الأحرف الأبجدية بحالة الحرف الصغير	[a-z]
أي حرف أبجدي	[a-zA-Z]
السلسلتين wxzأوwyz	w(x | y)z

الجدول(2) بعض التعابير المنتظمة

3. قسم التوابع المعرفة من قبل المستخدم: يحوي هذا المقطع على جمل برمجية بلغة (C\C++) وتوابع إضافية .[1].

مثال :

كتابة ملف (flex) للتعرف على عبارة التصريح عن متحول التالية بلغة++C :

```
int x=10;
```

أولا يجب تمييز أنواع الوحدات اللفظية :

- كلمة محجوزة (int) .
- متحول.
- واشارتي (= و ;) .
- العدد الصحيح 10.

```
%{
#include <iostream>
Using namespace std;
%}
```

```
%%
int          { cout <<"this is int token";}
[a-z]+       { cout<<"this is id token";}
[=]          { cout<<"this is = equal token" ;}
[;]          { cout<< "this is semicolon token" ;}
.            {cout<<" Lexical error not valid charchter";}
%%
Void main()
{
yylex();
}
```

ملف (a.l)

مع ملاحظة أن الحدث هنا هو طباعة نوع الوحدة اللفظية ولكن عند ربط المحلل اللفظي مع القواعدي يجب إضافة تعليمة(return) لنعود بنوع الوحدة اللفظية الموافق للعينة.
يتم توليد التابع(yylex) اتوماتيكاً من قبل الأداة(flex) داخل ملف (a.flex.cpp)،حيث يعتبر قلب المحلل اللفظي يقوم بتحويل قسم القواعد إلى شيفرة موافقة لها مبينة على التعابير النظامية.[5,6]

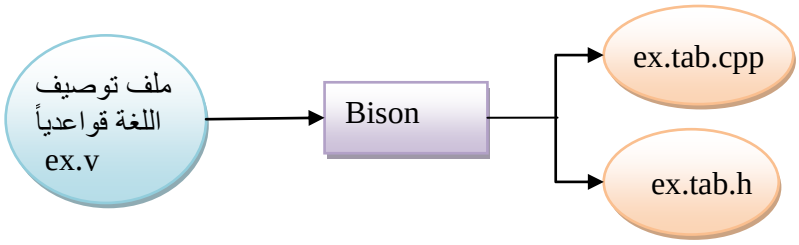
5-2 مولدات المحلل القواعدي :

وهي برامج مصممة لتوليد برامج المحللات القواعدية (Parser). يعتبر(Yacc) أولى المولدات القواعدية تعتمد على ما ينتجه المحلل اللفظي(lex) حيث يأخذ كدخل توصيف القواعد فيقوم بإنشاء محلل((LALR(1) للتعرف على الجمل في تلك القواعد وهو اختصار لـ(yet another compiler compiler)، سنوضح في بحثنا هذه نسخة محدثة عنه وقريبة له تدعى (Bison)[9]

- **أداة توليد المحلل القواعدي (BISON)**

هي أداة لتوليد المحلل القواعدي صممت من قبل(GNU Project)مفتوحة المصدر، دخلها عبارة عن ملف توصيف اللغة قواعدياً (*.y) مبني بالاعتماد على النموذج

القواعدي خارج السياق، [1]أماخرجها ملف شيفرة برنامج المحلل القواعدي (*.tab.cpp) مع ملف ترويسة (*.tab.h) يحوي تعريف أسماء الوحدات اللفظية يجب تضمينه في ملف توصيف اللغة (a.l) حتى تكون تعليمة (return) مع نوع الوحدة اللفظية صحيحة.[3].



الشكل(4) آلية عمل Bison

يتم ترجمة هذه الملفات مع الملف الناتج عن أداة ال(Flex) لنحصل على المحلل القواعدي، ولا يمكن تنفيذ المحلل القواعدي لوحده بل يعمل بالتوازي مع المحلل اللفظي لان دخله هو الوحدات اللفظية .[5].

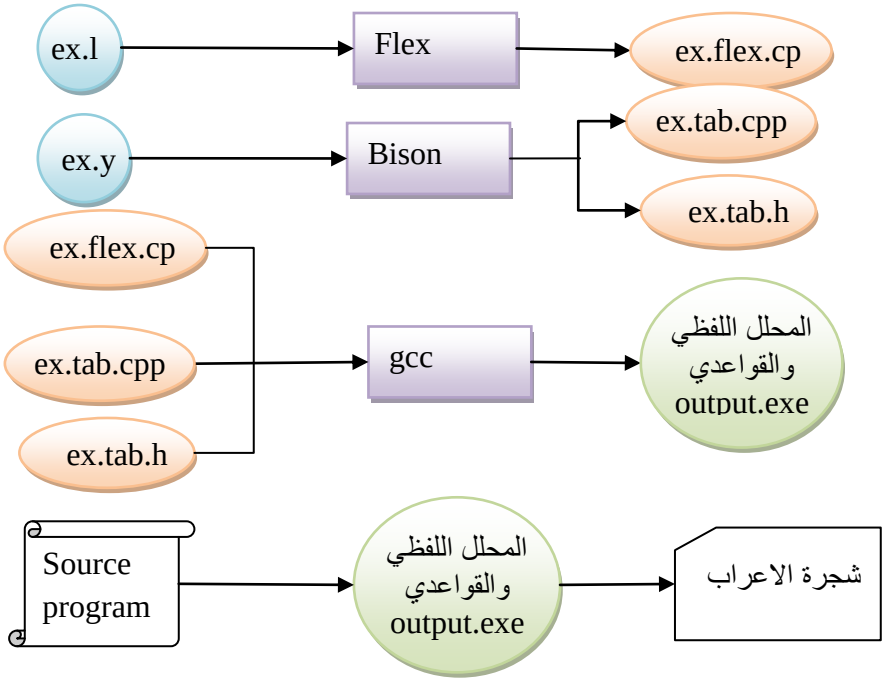
- **بنية ملف(y.*):**

يتألف من ثلاث أقسام رئيسية كل قسم مفصول عن الآخر بالإشارة (%%) بحيث يتم وضعها على سطر منفرد (كما في ملف flex)، [1] لكن هنا شكل القواعد والتصريحات مختلف.

```
{Definitions}
%%
{Rules}
%%
{User subroutines}
```

1. قسم التصريحات :

```
%{
//c++ decleration
%}
bison decleration
%%
```



الشكل(4) آلية عمل Flex&Bison

وهي قسمان :

a. تصريحات لغة C يتم حصرها بالرمز (%{)

b. تصريحات Bison:

- التصريح عن أسماء الوحدات اللفظية التي سوف تأتي من المحلل اللفظي(%toke)
- التصريح عن أنماطها(%type).
- التصريح عن الأنماط (%union).
- التصريح عن رمز البداية(%start)

2. قسم القواعد وهو أساسي يحوي سلسلة من القواعد بحيث كل قاعدة تتألف من (Production) وحدث موافق لها عبارة عن شيفرة بلغة (C++) كما يلي:

%%

production$_1$ { *action*$_1$ }

production$_2$ { *action*$_2$ }

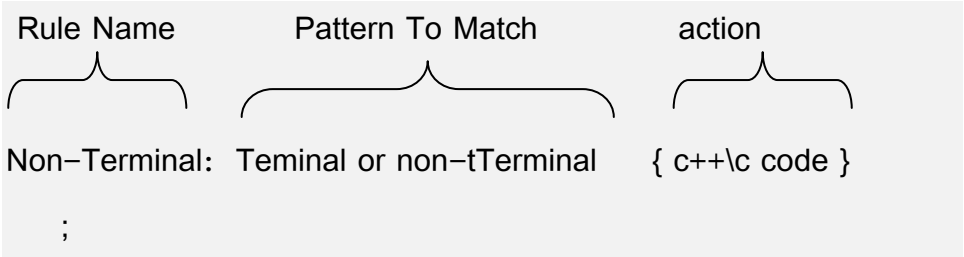
…

%%

القاعدة :

هي مجموعة من القواعد المهيكلة توصف اللغة وتطابق جملة ما، لتشكيل أي قاعدة نحتاج إلى مجموعة رموز لانهائية تظهر على الطرف اليسار للقاعدة، ومجموعة من الرموز النهائية (الوحدات اللفظية) لا تظهر أبداً على يسار القاعدة،بالتالي تعتمد على نموذج قواعدي خارج السياق .[5][1]

تكون في (Bison) من الشكل التالي:



3. قسم التوابع المعرفة من قبل المستخدم: يحوي هذا المقطع على جمل برمجية بلغة (C\C++) وتوابع إضافية.

يتم توليد التابع (yyparse) اتوماتيكا من قبل الأداة (Bison) عند إعطاءها ملف (a.y) كدخل كما أن هذا التابع هو قلب المحلل القواعدي يقوم بتحويل قسم القواعد إلى شيفرة موافقة لها.[5]

<u>مثال</u> :

كتابة ملف (bison) للتحقق من عبارة التصريح عن متحول التالية بلغة++C بالشكل التالي :

int x=10;

أولا: نجد أن قاعدة التصريح تتألف من نمط ثم اسم المتحول ومن ثم إشارة يساوي ومن

ثم القيمة وأخيرا الفاصلة المنقوطة.

ثانيا: يجب التصريح عن الوحدات اللفظية في ملف الـ(y.*) واستخدامها في ملف الـ(l.*) كما يلي:

Example.l	Example.y

%{
#include <iostream>
using namespace std;
#include "example.tab.h"
int yyerror (char *);
%}
%option noyywrap
%%
int { return Tint;}
[a-zA-Z]+ { return id;}
[0-9]+ { return Tintnum;}
[+;] { return *yytext;}
. {yyerror("lex");}
%%
	%{
	#include <iostream>
	using namespace std;
	int yylex();
	int yyerror (char *);
	%}
	%token Tint
	%token Tintnum
	%token id
	%start decl
	%%
	decl : Tint id '=' Tintnum ';'
	;
	%%
	void main(){yyparse();}
	int yyerror (char *c)
	{cout <<c;
	return 1;}

8-دراسة عملية:

تهدف هذه الدراسة إلى ببناء مترجم للغة بسيطة (مرحلة المحلل اللفظي والقواعدي فقط) يدوياً ومن ثم باستخدام مولدات المحلل اللفظي والقواعدي(Flex&Bison) فيما يلي سوف نقوم بشرح كل من الأداتين بما يغطي التفاصيل المشروحة سابقا بالاعتماد على قواعد اللغة التالية:

- يجب مراعات تسمية المتحولات الشروط نفسها كما في اللغات.
- أنماط المعطيات المدعومة : Boolean – Double – Int

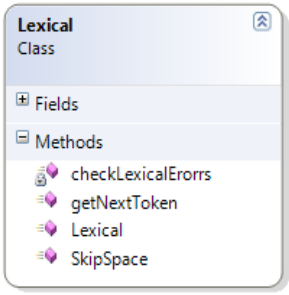
- تعليمة الأسناد .
- تعليمة القراء و الكتابة .
- الحلقات والشروط .

الشكل العام للبرنامج	التعليمات	بعض التوابع

Program programName;
Var
X: Data_type;
z,y: data_type;
Begin
..
instruction
..
End.
	IF
	If (condition) then
	One statement ;
	If (condition) then
	Statement ;
	If (condition) then
	One statement ;
	Else
	One statement ;
	: While
	While (condition) do
	Begin
	One statement ;
	End;
	: For
	For counter = val1 to val2 do
	One statement ;
		- **: Read**
		Read (varaible);
		Readline (varabile);
		- **: Write**
		Write (Exp);
		Writeline (Exp);

جدول (3) قواعد اللغة المدروسة

8-1 مقارنة بين كتابة المحلل الفظي يدويا وباستخدام الأداة (flex):



نجد عند كتابة المحلل اللفظي يدوياً يجب تعريف الكلمات المفتاحية في ملف، ليتم قراءته عند تحميل المحلل اللفظي ومن ثم قمنا بإنشاء الصف (Lexical)الذي يحقق الدوال التالية:

- الدالة(SkipSpace): هذه الدالة تسمح لنا بتخطي المحارف الغير ضرورية مثل (الفراغ السطور التعليقات ..) وتتوقف عند وجود محرف مفيد أو عند الوصول إلى نهاية الملف يمكن اعتماد الخوارزمية التالية:

```
While(not end of file)
Switch current_char
  Case space: col++;
  Case tab : col+=7;
  Case new_line : line++ ; col=1;
  Case comment:
          if (comment line) go to end line ;Line++ ;col=1;
        else if(comment multi line) search for endof comment symbol
   default:  return there is character
```

الشكل(5) خوارزمية تجاوز الفراغات والأحرف البيضاء

الدالة (getToken): تسمح بالتعرف على الوحدات اللفظية وفق الخوارزمية:

```
While (Skip space)
Switch (current_Char)
 Case letter:
     While(current_char is letter)
        Temp=temp+ current_char
     Token T_type= search(Tem);
     if (T!= NULL) return T_type;
     else return T_id;
case digit:
     check_intnumber();
     check_doublenumber();
default:
   if (current_char = '+') return T_plus;
    if (current_char = '*') return T_multiply;
 // for all symbol (+-* < > <= >= …) we discuss this
```

الشكل(6) خوارزمية الحصول على الوحدة اللفظية

أما في حال استخدام (Flex) يكفي أن نعرف التعابير المنتظمة الموافقة لهذه الوحدات

اللفظية في ملف توصيف اللغة(l.*)، وبالمقارنة نجد سهولة توصيف اللغة لفظياً باستخدام التعابير المنتظمة من كتابة شيفرة برمجية لكل الحالات (الفراغ– مسافة الجدولة والسطر الجديد).

```
boolean SkipSpace()
      {
while (currentIndex <fileContent.Length)
{
switch currentcharachter
{
case ' ':
      currentIndex++;
        column++;
        break;
case '\n':
      currentIndex++;
       line++;
      column = 1;
        break;
case for tab charchter and comment…
  increase current index
…
…
default:  case there is charchter….    }
```

```
%%

[ \t]     {col += yyleng;}
\n        {line++; col = 1;}
...
```

جدول(4) مقارنة كتابة المحلل اللفظي حالة الأحرف البيضاء

أما من أجل مطابقة الوحدات اللفظية، حالة كلمة نصية:

– في الحالة اليدوية عند تجميع الكلمة يتم استدعاء تابع يقوم بالبحث عنها في جدول الكلمات المفتاحية إذا وجدها يعود بنوع الوحدة اللفظية وإلا يعود أنها متحول.

– في حالة استخدام أداة(flex) يكفي كتابة التعبير المنتظم لكل وحدة لفظية "وهي نفس الكلمة " في حال كانت كلمة مفتاحية وإلا يقوم بمطابقة التعبير المنتظم للمتحول، من هنا نستنتج أن ترتيب الوحدات اللفظية في ملف (flex) له أهمية كبيرة حيث يتم المطابقة من الأعلى للأسفل الأول فالأطول.

for (int i = 0; i < CCompiler.keyWords.Count; i++)
{ if (Buffstr == ((Symbol)CCompiler.keyWords[i]).symbol)
{tknTemp = ((Symbol)CCompiler.keyWords[i]).type;
break; }
else tknTemp = token.U_ID;
	%%
	program {col += yyleng;
	return PROGRAM}
	begin {col += yyleng;
	return Begin}
	end {col += yyleng;
	return End;
	…….
	[a-zA-Z_][A-Za-z0-9_]*
	{ col += yyleng;
	Return IDENT;}

جدول(5) مقارنة كتابة المحلل اللفظي حالة الأحرف النصية

كذلك الحال عند مناقشة الرموز الأخرى.

public token getNextToken()
{ if (!SkipSpace())
goto test_U_EOF;
switch (fileContent[currentIndex++])
{ case '=':
{
column++;
if(currentIndex<fileContent.Length)
{
if(fileContent[currentIndex+1]=='=')
{ column++;
currentIndex++;
tknTemp = token.U_EQUAL;
break; }
}
tknTemp = token. U_ASSIGN;
break; }
	%%
	[\t] {col += yyleng;}
	\n {line++; col= 1;}
	[;,():=+*/-] {col += yyleng;
	return*yytext;}
	"==" {col += yyleng;
	return EQUAL;}

جدول(6) مقارنة كتابة المحلل اللفظي حالة الرموز

نلاحظ من خلال هذا البحث توفير الأداة (Flex) لآلية مريحة لكتابة أي محلل لفظي دون الدخول بتفاصيل قد تستهلك وقت أكثر فهي تقوم بتوليد الدالة (yylex()) التي يكون مضمونها موافق بالعمل للدالة (getNextToken).

بالمقارنة نجد أنه احتجنا تشفير برنامج مكون من 613 تقريباً بالمقابل عند توصيف اللغة باستخدام (Flex) تم بحوالي 55 سطر.

استخدام أداة flex	البرمجة اليدوية لمرحلة المحلل اللفظي	
489	890	عدد الأسطر

جدول(7) مقارنة عدد الأسطر لتوليد المحلل اللفظي

8-2 مقارنة بين كتابة المحلل القواعدي يدويا وباستخدام الأداة(Bison)

عندما نتحدث عن تحليل القواعد أي كل قاعدة من ماذا تتكون وهل هي مطابقة لشروط القاعدة، مثلا: بعد أن تأتي كلمة Program سوف نناقش الكلمة التي تأتي بعدها وتكون اسم البرنامج، وبالنسبة للمتحولات من أي نمط وطريقة كتابتها والتعليمات(الإسناد والحلقات ..) كيف تكتب.

عند تحقيق المحلل القواعدي يدوياً سنحصل على الصف (Syntax) هذا الصف يقوم بأعمال كثيرة، من خلاله نستطيع قراءة المتحولات وقراءة التعابير الرياضية وقراءة التعليمات، وأخيرا تحليل اللغة قواعدياً من خلال الدالة (parser) التي تعتبر قلب المحلل القواعدي.

الدالة (Parser) تقوم بالتأكد من تتالي كلمة (program) وبعدها اسم البرنامج ومن ثم قسم التصريح عن المتحولات ومن ثم قسم التعليمات.

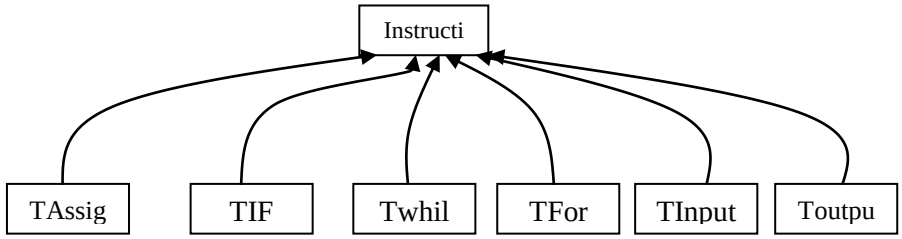
```
parser()
{
        syntaxToken = lex.getNextToken();
        if (syntaxToken != token.U_PROGRAM)   // Error
        syntaxToken = lex.getNextToken();
        if (syntaxToken != token.U_ID) // Error
        syntaxToken = lex.getNextToken();
        if (syntaxToken != token.U_SEMICOLON)// Error
        syntaxToken = lex.getNextToken();
        if (syntaxToken == token.U_VAR)
        {
            readBlock ofVarDeclare();
        }

        if (syntaxToken != token.U_BEGIN)// Error
       InstructionsList=readBlockInstructions();
      if (InstructionsList == null)          // Error
      syntaxToken = lex.getNextToken();
       if (syntaxToken != token.U_EOF)      // Error
```

```
}
```

الشكل(7) خوارزمية التحقق من الشكل العام للبرنامج

بما انه لدينا مجموعة من التعليمات (تعليمة شرطية ، تكرارية، اسناد ...)سنقوم بإنشاء صف عام اسمه(Instruction) ونشتق منه صفوف موافقة لهذه التعليمات(صف لكل تعليمة) وذلك من أجل بناء الشجرة.



الشكل(8) صفوف التعليمات

مثلاً من أجل تعليمة التصريح عن متحول:

X : int

VarName Type

قسم مطابقة قاعدة التصريح عن المتحول:

```
Var readDec()
    {
       var = readIDs();
      if (syntaxToken != token.U_COLON)
      {
         //Error
        // breakParsing
         return null;
      }
      syntaxToken = lex.getNextToken();
      type = readType();
       if (type == token.U_UNKNOWN)
         return null;
      setType(var, type);
       return var;}
```

الشكل(9) خوارزمية التحقق من التصريح عن متحول

أما عند استخدام الأداة (bison) نجد الجزء الخاص بتعريف البرنامج الاساسي:

```
%token PROGRAM IDENT Begin End BOOLT  INTT  |DOUBLET
%start prog
%%
prog:       PROGRAM  IDENT  Begin  members   End
          ;
members:          /* Empty */
                  | members member
;
member:           function
                  |global
;
function:         Type IDENT '('arg_s')' '{'Statements'}'
;
global :          Type Vars ';'
;
Statements:        /*Empty*/
                  | Statements STMT
;
global :    Type Vars ';'
;
Vars:    Var
          | Vars ',' Var
;
Var:     IDENT
        | IDENT '=' expression
;
Type:  BOOLT
        |INTT
        |DOUBLET
;
```

الشكل(10) قواعد التحقق من البرنامج باستخدام(Bison)

ولبناء الشجرة نحتاج إلى إنشاء مجموعة من الصفوف الموافقة للقواعد. ويتم وضع الشيفرة الخاصة بالبناء ضمن الحدث المرافق لكل قاعدة .

كما في المثال:

```
prog:PROGRAM IDENT Begin members  End { $$=new Prog
($2,$4,line,col);}
 ;
```

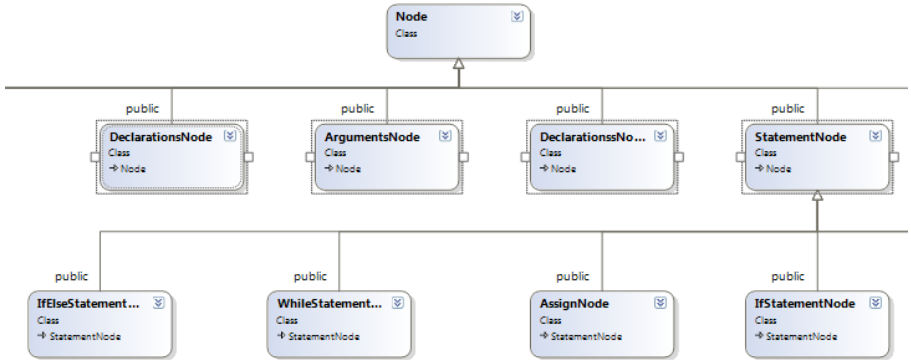
حيث ($$) تحمل قيمة الطرف اليساري من القاعدة ، ($2) من النمط متحول تحمل

قيمة الرمز الثاني للقاعدة ($4) من النمط عناصر البرنامج تحمل قيمة الرمز الرابع من القاعدة.

حيث القاعد العامة للوصول إلى رمز القاعدة حيث (i) هو رقم ترتيب الرمز بالقاعدة:

Left : M1 M2…Mn

$$ $1 $2…$n



الشكل (11) جزء من الصفوف اللازمة لبناء الشجرة .

حالة الغموض في المحلل القواعدي هي الحالة التي يحدث فيها تعارض في آلية عمله، يوجد نوعين من التعارض (Shift\Reduce) أو تعارض (Reduce\Reduce).

إذا كانت لدينا القاعدة الخاصة بالتعليمة الشرطية :

```
stmt:   expr ';'
      | Type expr ';'
      |IF '(' expr ')' stmt
      | IF '(' expr ')' stmt ELSE stmt
      ;
```

عند قراءة الكلمة (Else) ستكون في قمة المكدس كل شيء قبلها موافق لقاعدة التعليمة الشرطية وهي صحيحة ولكن وجود else يعني أن هناك دخل قادم لإكمال القاعدة الثانية للتعليمة الشرطية هذه الحالة من التعارضات تسمى تعارض (shift\Reduce)هل يقوم المحلل بتبديل القاعدة الأولى للتعليمة الشرطية أم يتابع في قراءة الدخل ويقوم بتنفيذ ازاحة للكلمات.

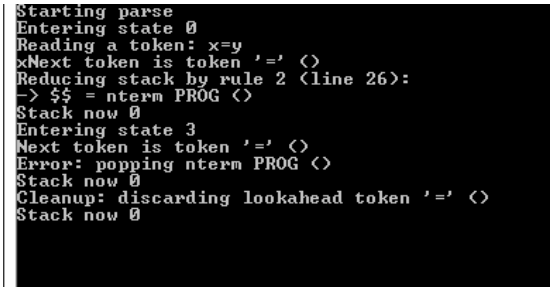
أما في حالة تعارض (Reduce\Reduce) عندما يكوم هناك قاعدتين يمكن أن نقوم بعملية التبديل معهما في نفس الوقت.

نجد هنا أهمية الـ(Bison) فهو مصمم لحل مثل هذه التعارضات من خلال معامل الأولوية الذي يتم تعريفه في قسم التصريحات[12]

```
%prec IF_PREC
%%
….
 stmt:      expr ';'
          | Type expr ';'
          |IF '(' expr ')' stmt  %prec IF_PREC
          | IF '(' expr ')' stmt ELSE stmt
          ;
```

أو يعطينا الخيار لإعادة النظر في القواعد وكتابتها بشكل يجنبنا هذا الغموض، لكن في حالة البرمجة اليديوية لن يكون هناك مؤشر فعلي يوضح مثل هذه الحالات.

نضيف إلى ذلك إمكانية تشغيل (Debuger) أثناء تنفيذ (Bison) ليرينا حالة المكدس وعمليات (Shift –Reduce) التي يقوم فيها فعليا وفق ارقام القواعد المعطاة.



الشكل (12) تشغيل (Bison) في حالة (debugger)

بالإضافة للميزات السابقة وبالمقارنة نجد أنه في حال البرمجة اليدوية للمحلل القواعدي احتاج ما يقارب (860) سطر بينما عند استخدام ملف توصيف اللغة قواعدياً باستخدام الأداة (bison) احتجنا ما يقارب 190 سطر لكتابة القواعد فقط وعند إضافة بناء الشجرة في الحدث المقابل لكل قاعدة ازداد عدد الأسطر إلى حوالي 489 سطر تقريباً مع ملاحظة سهولة تحديد الأخطاء ومكان وقوعها لعدم ضياع ترتيب الوحدات االلفظية وضمان وجود آلية لتجنب حدوث غموض في كتابة القواعد.

	البرمجة اليدوية لمرحلة المحلل القواعدي	استخدام أداةbison
عدد الأسطر	890	489

جدول(8) مقارنة عدد الأسطر لتوليد المحلل القواعدي

9- الاستنتاجات والمقترحات:

من خلال دراسة أدوات توليد المترجم التي قمنا بها، توصلنا في هذا البحث إلى أن استخدام مولدات برامج المحللات اللفظية والقواعدية أقوى من البرمجة اليدوية لبناء المترجم من حيث كشف الأخطاء والغموض، ويمكن استخدامها ضمن شريحة واسعة من التطبيقات ومن قبل أغلب المبرمجين، لتقديمها مرونة كبيرة في كتابة المحلل القواعدي دون الدخول في التفاصيل البرمجية التي يمكن أن تحصل في حال ضياع وحدة لفظية.

وبناء عليه وجدنا أن جاذبية أدوات توليد المحلل اللفظي والقواعدي تتمثل في امكانية توصيف لغة المصدر بطريقة مجردة وبعيدة عن العلاقات الرياضية والحسابات هي التعابير المنتظمة والنموذج القواعدي خارج السياق. كما يمكن اعتبار هذا البحث أساس لدراسات مستقبلية تساعد الباحثين في هذا المجال لبناء مترجم لأي لغة وبداية تشجيعية لدخول عالم المترجمات .

المراجع

[1] Zhang, Y., Lu, Y., & Yang, B,2017, June- Parsing Statement List Program Using Flex And Bison, In 2017 First International Conference on Electronics Instrumentation & Information Systems (EIIS) (pp. 1-4). IEEE.

[2] Farhanaaz and V. Sanju, 2016 ,An exploration on lexical analysis, International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), Chennai

[3] Levine, J.,2009, Flex & Bison: Text Processing Tools. O'Reilly Media, Inc.

[4] Appel, A. W. ,2004, Modern compiler implementation in C, Cambridge university press

[5] Aaby, A. A. ,2003, Compiler construction using flex and bison. Walla Walla College.

[6] Debray, S ,2002, Making compiler design relevant for students who will (most likely) never design a compiler. ACM SIGCSE Bulletin, 34(1), 341-345.

[7] Lesk, M. E., & Schmidt, E. 1975, Lex: A lexical analyzer generator .Computer Science Technical Report No. 39, Bell Laboratories, Murray Hill New Jersey.

[8] Alekhya, A. and Someswar, G.M., 2014, on Development of Compiler Design Techniques for Effective Code Optimization and Code Generation. IJETAE, 4(3).

[9] Maggie Johnson and revised by Julie Zelenski,2015, Introduction to yacc and bison

[10] Louden, K. C. ,1997, Compiler construction. *Cengage Learning*.

[11] 11 Jain, M., Sehrawat, N., & Munsi, N. ,2014,Compiler Basic Design and Construction. *Int. J. Comput. Sci. Mob. Comput*, *3*(10), 850-852.

[12] https://www.gnu.org/software/bison/manual/