مجلة جامعة تشرين للبحوث والدراسات العلمية _ سلسلة العلوم الهندسية المجلد (36) العدد (4) 2014
Tishreen University Journal for Research and Scientific Studies - Engineering Sciences Series Vol. (36) No. (4) 2014

تصميم مترجم للغة مصدرية مقترحة باستخدام الأداتين البرمجيتين LEX و BISON

الدكتور جبر حنا*
علي ميا**

(تاريخ الإيداع 2 / 10 / 2013. قُبِل للنشر في 14/ 7 / 2014)

□ ملخّص □

تقدم الدراسة طريقة لتوليد مترجم متكامل للغة مصدرية مقترحة تتضمن جميع العمليات الحسابية والمنطقية والحلقات وبنى التحكم وعمليات التصريح والإسناد. تتضمن الدراسة عدة مراحل بدءاً من مرحلة بناء محلل المفردات (الماسح) اعتماداً على البنية البرمجية LEX، يلي ذلك مرحلة بناء المحلل القواعدي باستخدام الأداة البرمجية BISON وذلك من أجل تحديد قواعد اللغة الناظمة لعمل المترجم، بعد ذلك يتم ترجمة المعرب (Parser) باستخدام لغة Turbo C++ وذلك للحصول على الخرج النهائي. تم اختبار المترجم المقترح على مئة ملف مصدري، وأظهر المترجم قدرة على ترجمة كل الملفات وتحديد مواقع الخطأ والعبارات المسببة للخطأ في كل ملف مصدري.

الكلمات المفتاحية: المترجمات، الماسح، المعرب، قواعد الإعراب، تحليل المفردات، التحليل القواعدي، تحليل المعاني.

* أستاذ – قسم هندسة الحاسبات والتحكم الآلي– كلية الهندسة الميكانيكية والكهربائية– جامعة تشرين– اللاذقية– سورية.
** قائم بالأعمال – قسم هندسة الحاسبات والتحكم الآلي – كلية الهندسة الميكانيكية والكهربائية – جامعة تشرين–اللاذقية – سورية.

مجلة جامعة تشرين للبحوث والدراسات العلمية _ سلسلة العلوم الهندسية المجلد (36) العدد (4) 2014
Tishreen University Journal for Research and Scientific Studies - Engineering Sciences Series Vol. (36) No. (4) 2014

Designing a compiler of proposed source language Using LEX and BISON

Dr. Jabr Hanna[*]
Ali Mia[**]

(Received 2 / 10 / 2013. Accepted 14 / 7 / 2014)

□ ABSTRACT □

This paper proposes an approach for compiler construction of a proposed source language consisting of all computational and logical operations, control statements, loops, declarations and assignment operations. This study is based on many steps starting from lexical analysis (Scanner) depending on LEX environment; after that, the syntax analysis step is done to obtain the syntax analyzer (Parser) which defines the grammar rules. The final step is compiling the parser using Turbo C++ to get the final output. The designed compiler has been tested on 100 source files, and the results show that the designed program compiles all these source files correctly .In addition it defines the error's locations through them.

Keywords: Compilers, Scanner, Parser, LEX, BISON, Grammar rules, Lexical analysis, Syntax analysis, Semantic analysis.

[*] **Professor in Department of computer and automatic control Engineering, Faculty of Mechanical and electrical Engineering, Tishreen University, Lattakia, Syria.**
[**] **Academic Assistant, Department of computer and automatic control Engineering, Faculty of Mechanical and electrical Engineering, Tishreen University, Lattakia, Syria.**

مقدمة:

تعتبر المترجمات برامج تقوم بتحويل البرنامج المصدري المكتوب بلغة عالية المستوى إلى برنامج هدفي بلغة الآلة [2،1]. اللغات عالية المستوى هي لغات أقرب إلى المستخدم الذي يقوم بكتابة البرامج بلغات تحاكي لغته الطبيعية، أما اللغة الهدفية فهي اللغة التي يتعامل معها الحاسب وهي لغة الآلة (أصفار وواحدات) [3].

طورت لغات البرمجة الأولى من لغة الآلة، واعتمدت على كتابة أسماء رمزية بالأصفار والواحدات، وليس كما هو متعارف عليه الآن من خلال تحديد عنوان التعليمة المراد جلبها من الذاكرة وتنفيذها، وسميت هذه اللغات باللغات الإجرائية procedural languages مثل لغات الفورتران FORTRAN، الكوبول COBOL، الباسكال Pascal والـ C[8،7،6،5].

تبع ذلك ظهور اللغات الوظيفية Functional programming والمبنية على نموذج تورينغ في البرمجةTuring Machine، حيث أن أي شيء يمكن أن ينجز من خلال تعابير برمجية مثل الحلقات والتوابع والمصفوفات الخ...

ولاحقاً ظهرت اللغات غرضية التوجه Object Oriented Languages OOP، والتي اعتمدت بشكل أساسي على مفهوم الغرض أو الكائن Object وليس على التوابع، وكانت الميزة الأساسية لها هي إنشاء كائنات تحتفظ بقيمها بشكل خاص private مما يجعله مرئية لأجزاء محددة من البرنامج. قدمت هذه اللغات مفهوم قابلية الوصول من قبل أجزاء خاصة من البرنامج هي الطرائق methods والتي تشبه فكرة التوابع وتسمى بمفهوم التغليف encapsulation. أما الموضوع الأكثر أهمية والذي قدمته هذه اللغات هو الوراثة inheritance والذي جعل الصفوف قادرة على وراثة سلوك وحالة الصفوف الأب Superclass [8،7،6،5].

عموماً، كان الظهور الأول لمترجمات لغات البرمجة في العام 1946 [3] من خلال لغة Short Code، والتي كانت لغة البرمجة الأولى المستخدمة في الأجهزة الاكترونية. على كل الأحوال كانت هذه اللغة مترجمة يدوياً فقط hand-compiled.

في العام 1951، عمل الباحث Grace Hopper على أول مترجم معروف في تلك الفترة والذي سمي مترجم A-0 [3]. لاحقاً قام الباحث Rand inبتوليد لغة من هذا المترجم وسميت MATH-MATIC [3].

صمم الباحث Alick E. Glennie من جامعة مانشستر فيما بعد مترجماً سمي بالشيفرة الفورية AUTOCODE [2].

بعدها بعدة سنوات وتحديداً في العام 1957، ظهرت لغة البرمجة العالمية المشهورة الفورتران FORTRAN [3،2]، حيث قاد المبرمج John Backus تطوير هذه اللغة، حيث ظهرت بعد ذلك نسخة متطورة من اللغة سميت FORTRAN II وذلك في العام 1958، والتي كانت تدعم الإجرائيات الفرعية [3].

في الفترة ذاتها نشأت العديد من اللغات مثل لغة الكوبول المطورة من قبل المعهد العالمي الأمريكي ANSIلمعالجة البيانات المتعلقة بالأمور الاقتصادية، ولغة Lisp لمعالجة الحسابات ذات الرموز، وكان الهدف الأساسي من إنشاء هذه اللغات هو تسهيل تعامل المستخدمين مع الحسابات ذات الأرقام العددية الكبيرة والمعقدة، والتطبيقات الاقتصادية، والبرامج التي تحتوي عدد كبير من الرموز (المعرفات) [2].

جذب بناء المترجمات في نهايات الستينات اهتمام الكثير من المبرمجين، وفي العام 1970 تمكن المبرمج Charles Moore من كتابة أول برامجه باستخدام لغته الخاصة Forth والمبنية بالاعتماد على لغة البرولوج

Prolog[3]، وفي العام 1972 أطلقت لغة الـ C من قبل Dennis Ritchie و Brian Kernighan، ولاحقاً تم تطوير هذه اللغة من قبل Bjarne Stroustrup والذي أضاف الصفوف إليها والتي عرفت بلغة الـ C++. في نهاية 1983 وبدايات 1984 قدمت كل من مايكروسوفت وديجيتال مترجم الـ C الأول للمعالجات الصغرية [3]. ظهرت لاحقاً العديد من لغات البرمجة عالية المستوى مثل الجافا والـ C# والتي تضمنت العديد من التحسينات على اللغات الأم مثل الوراثةinheritance وتعدد الأشكال polymorphism وتعدد المسارات multithreading ومعالجة الاستثناءات exception handlingوالمعالجة التفرعية parallel processing.

تقدم الدراسة المقترحة تصميماً لمترجم متكامل بالاعتماد على الأداتين البرمجيتين LEX و BISON، ثم توليد الشيفرة الوسيطية بلغة التجميع Assembly Language.

أهمية البحث وأهدافه:

نظراً للتعقيد الشديد في توليد المترجمات فقد بقي هذا المجال من أقل المجالات اهتماماً، ونظراً للأهمية الشديدة لمعرفة كيفية عمل لغات البرمجة بدءاً من مرحلة قراءة الكود المصدري وصولاً إلى مرحلة توليد الكود الهدفي النهائي، كان لابد من إبراز أهمية هذه المراحل التي تعتبر عصب لغات البرمجة. يهدف البحث إلى توليد مترجم متكامل وفوري للغة مصدرية مقترحة تتضمن كل الحالات التي يمكن أن تصادف المبرمج في لغات البرمجة (عمليات تصريح- إسناد- حلقات – حلقات متداخلة- عمليات حسابية ومنطقية) مع إمكانية التعرف على أماكن الأخطاء والعبارات المسببة لها وإعلام المستخدم بها.

طرائق البحث ومواده:

تم استخدام عدة أدوات برمجية ولغات برمجة للوصول للمترجم النهائي وهذه الأدوات واللغات هي:

- مولد محلل المفردات LEX:

للحصول على الماسح (محلل المفردات).

- مولد محلل القواعد BISON:

للحصول على المعرب (محلل القواعد).

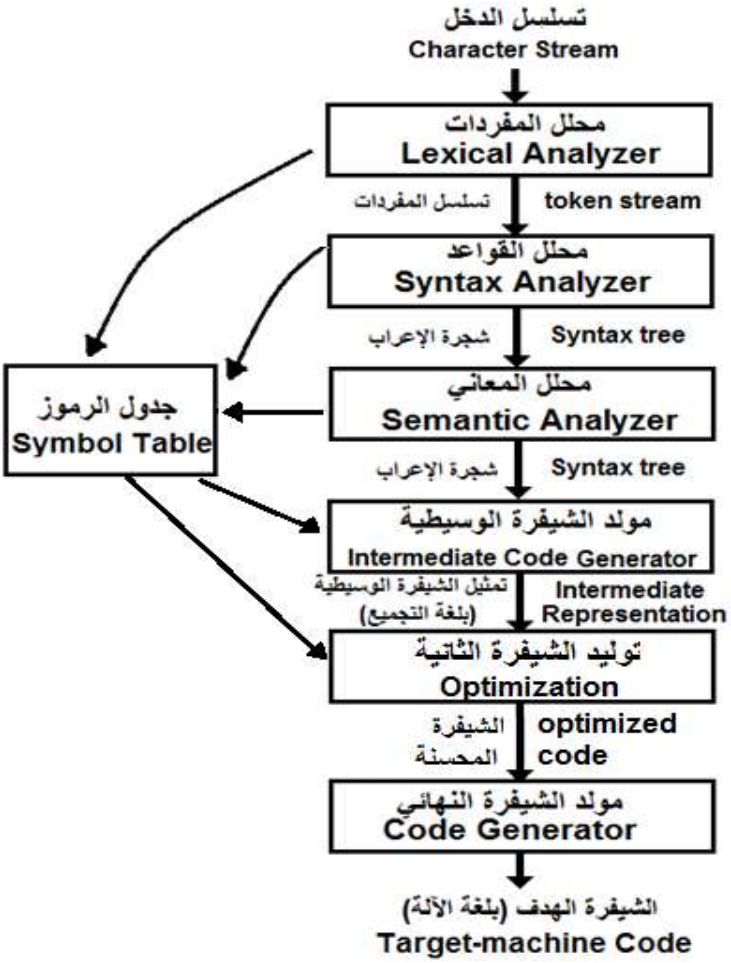
- لغة التربو Turbo C++:

لترجمة ملف المعرب.

1 - مراحل الترجمة:

تتألف عملية الترجمة من عدة مراحل[1،2] مبينة في الشكل (1) وهي المراحل التالية:

1- مرحلة تحليل المفردات Lexical Analysis.
2- مرحلة تحليل القواعد Syntax Analysis.
3- مرحلة تحليل المعاني Semantic Analysis.
4- مرحلة توليد الشيفرة الوسيطية Intermediate Code Generation.
5- مرحلة توليد الشيفرة الثانيةOptimization.
6- مرحلة توليد الشيفرة الهدف Code Generation.



الشكل (1) مراحل الترجمة

➢ **مرحلة تحليل المفردات:** والتي تتضمن تقسيم السلسلة المصدرية إلى عدد من الرموزtokens، لذا فإن دخل هذه المرحلة هو البرنامج المصدري أما خرجها فهو تسلسل الرموزtokens. تستخدم عادة عدة قوالب لتقسيم تسلسل الدخل إلى مجموعة من الرموز، فمثلاً يعبر عن قالب الأعداد بالشكل +[0–9]، وعن قالب الفراغات بالشكل [\t\n] وهكذا...

تستخدم الأداة البرمجية LEX لتوليد الماسح Scannerالذي سيقوم بعملية تحويل سلسلة الدخل إلى تسلسل من الرموز [4].يتم كذلك خلال هذه المرحلة بناء جدول الرموز، والذي يتضمن أسماء الرموز التي يتم مسحها من سلسلة الدخل ونوع كل منها.

➢ **مرحلة تحليل القواعد:** وفيها يتم تحديد قواعد النحو الناظمة لعمل المترجم، والتي تحدد العبارات التي يمكن ترجمتها من قبل المترجم، وهذا يترتب عليه رفض العبارات التي لا تنتمي للنحو. يستقبل المحلل القواعدي الرموز التي استخلصها محلل المفردات كدخل له، ويقوم بإنتاج شجرة الإعراب الموافقة لهذه الرموز كخرج له في حال تمكن من إعرابها وفقاً لقواعد النحو، بينما يعطي خطأً قواعدياً في حال لم تعرب سلسلة الرموز بنجاح. تستخدم الأداة البرمجية BISON لتوليد المعرب parser الذي يقوم بعملية الإعراب.

➢ **مرحلة تحليل المعاني:** قد تكون العبارة صحيحة من حيث المفردات والقواعد، لكنها خاطئة معنوياً، فمثلاً قد يتم إسناد قيمة لمتحول ما لم يتم التصريح عنه مسبقاً، أو يتم إسناد قيمة لمتحول يمتلك نوع بيانات مختلف عن هذه القيمة.

يستخدم جدول الرموز الذي يتم بناؤه في مرحلة تحليل المفردات في هذه المرحلة، لكي يتم معرفة المتحولات المصرح عنها ونوع كل منها، وهذا ما نجده في جدول الرموز.

بقية مراحل المترجم تتعلق بنقل اللغة المصدرية إلى تعليمات بلغة التجميع (Assembly) (مرحلة توليد الشيفرة الوسيطية) بغية تحويلها لاحقاً إلى تعليمات بلغة الآلة اعتماداً على جداول الأسمبلي (مرحلة توليد الشيفرة النهائية).

2- جدول الرموز:

يتم بناء جدول الرموز Symbol Tableخلال مرحلة تحليل المفردات، ويتم تحديث هذا الجدول في كل مرة يعثر فيها الماسح على رمز من رموز الدخل، ويتضمن جدول الرموز حقلين يمثل الأول اسم المتحول بينما يتضمن الثاني نوع بيانات المتحول.

تكمن أهمية جدول الرموز في مرحلة تحليل المعاني، حيث يتم استخدامه للتحقق من عدم وجود أخطاء في المعاني، فعندما يتم إسناد قيمة لرمز غير معرف ضمن جدول الرموز يتم إصدار خطأ، وعندما يتم التصريح أكثر من مرة عن نفس المتحول كذلك لابد من إصدار خطأ، أما النوع الأخير من أخطاء المعاني التي يمكن كشفها اعتماداً على جدول الرموز فهي الأخطاء الناتجة عن إسناد قيم من نوع مخالف لنوع بيانات الرموز المخزنة في جدول الرموز.

يوضح الجدول التالي توصيفاً لجدول الرموز الذي يتم بناؤه خلال مرحلة ترجمة الملف المصدري (ملف المدخلات):

جدول (1) مثال عن جدول رموز بعد مصادفة الماسح لثلاثة رموز من أنواع مختلفة ضمن الملف المصدري:

رقم الرمز	اسم الرمز	نوع الرمز
1	X	رقم صحيحInteger
2	Y	رقم بفاصلة عائمة floating-number
3	M	سلسلة string

برمجياً، يمثل جدول الرموز بملف رأسي (h.) يتم تضمينه من قبل ملف المعرب كونه الملف الوحيد الذي ستتم ترجمته، ويتضمن ملف جدول الرموز عدداً من التصريحات والتوابع اللازمة لإنشاء جدول الرموز وتحديثه (إضافة رموز) والبحث عن رموز معينة ضمنه، ويمكن تقسيم بنية هذا الملف إلى الأجزاء التالية:

-قسم التضمين والتصريح:

يتضمن هذا القسم تعليمات بلغة الـ C لتضمين المكتبات اللازمة والضرورية لعمل بعض التوابع التي سيتم استخدامها ضمن جسم هذا الملف البرمجي. كذلك يتضمن التصريح عن المتحولات اللازمة لعمل الملف مثل التصريح عن أنماط البيانات المختلفة التي يمكن مشاهدتها في الملف المصدري. يوضح الجزء التالي من الكود بنية القسم الأول التي تم اقتراحها لبناء ملف جدول الرموز:

```
#include<stdlib.h> // تضمين المكتبة المعيارية
#include<stdio.h> // تضمين مكتبة الدخل الخرج الأساسية
```

```
#include<string.h> // تضمين مكتبة السلاسل
#define _int 1 // التصريح عن المتحولات التي تمثل الأنماط الأربعة
#define _float 2
#define _str 3
#define _chr 4
typedefstruct sym_node// تعريف تركيبة تمثل بنية جدول الرموز
{char name[50];
struct sym_node *next;
}sym_node
```

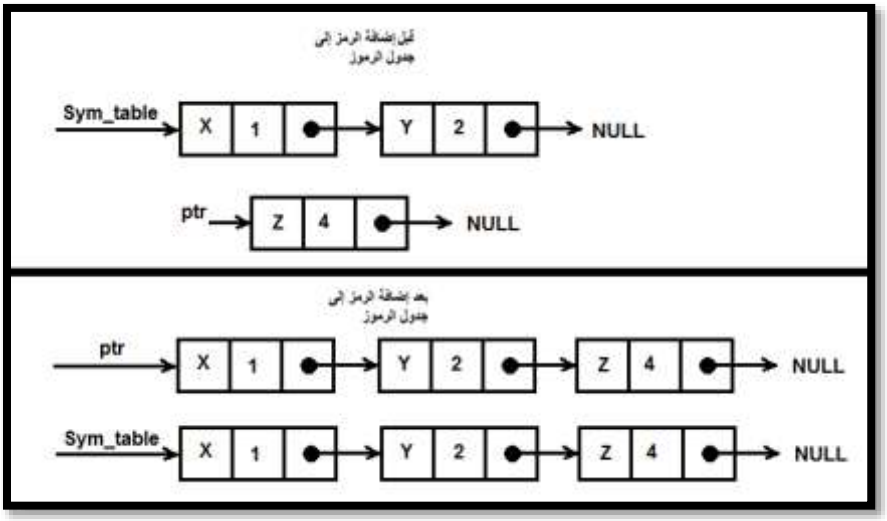
تم تضمين عدد من المكتبات الضرورية لعمل عدد من التوابع التي ستظهر لاحقاً، ثم تم تعريف أنماط البيانات التي تعاملنا معها في برنامجنا وهي النوع الصحيح وله الرقم 1، النوع الحقيقي وله الرقم 2، السلسلة ولها الرقم 3، والنوع المحرفي وله الرقم 4. بعدها تم تعريف تركيبة struct مكونة من سلسلة محرفية بحجم أعظمي 50 محرف اسمها name تمثل هذه السلسلة اسم المتغير ، كما تتضمن التركيبة مؤشر لتركيبة من نفس النوع next، وحقل يمثل نوع الرمز type، ثم تم إنشاء مؤشر sym_table من نوع التركيبة أيضاً وجعلناه يشير إلى الـ NULL، مبدئياً وحقيقة هو القائمة التي تمثل جدول الرموز.

-الجزء الخاص بإضافة رمز لجدول الرموز:

هنا يتم التصريح عن تابع اسمه put_sym يعيد قيمة من النوع sym_node ويأخذ كوسيط البارامتر sym_name ويقوم هذا التابع بتسجيل الرمز sym_name ضمن جدول الرموز ليصبح معلوماً، ويعيد مؤشر إلى جدول الرموز وهو ptr، أي أن وظيفة هذا التابع تسجيل الرمز ضمن جدول الرموز وفقاً للتعليمات البرمجية التالية [4]:

```
sym_node *sym_table=NULL;// إنشاء جدول الرموز
sym_node *put_sym(char *sym_name, int sym_type)// تابع إضافة رمز إلى جدول الرموز
{sym_node *ptr;
ptr=(sym_node*)malloc(sizeof(sym_node));
strcpy(ptr->name,sym_name);
ptr->type=sym_type;
ptr->next=(sym_node*)sym_table;
sym_table=ptr;
returnptr;}
```

يوضح الشكل (2) تمثيلاً مقترحاً لكيفية إضافة الرموز لجدول الرموز:



الشكل (2) تمثيل مقترح لكيفية إضافة الرموز لجدول الرموز

-الجزء الخاص بالبحث عن رمز ضمن جدول الرموز:

هنا يتم التصريح عن تابع اسمه get_sym يعيد قيمة من النوع sym_node ويأخذ كوسيط البارامتر sym_name ويقوم هذا التابع بالحصول على الرمز ذو الاسم sym_name من جدول الرموز، حيث يقوم بالبحث ضمن جدول الرموز الممثل بالمؤشر sym_table حتى الوصول إلى نهايته (أي إلى NULL) وفي حال وجوده يعيد مؤشر لمكان وجوده ptr وإلا فإنه يعيد NULL ليدل على أن الرمز المطلوب غير موجود في جدول الرموز.

وهنا يتم التأكد من أن الرمز موجود فعلاً أو غير موجود ضمن جدول الرموز[4].

```
sym_node *get_sym(char *sym_name)// تابع الحصول على رمز من جدول الرموز
{sym_node *ptr;
for(ptr=sym_table;ptr!=NULL;ptr=(sym_node*)ptr->next)
if(!strcmp(ptr->name,sym_name))return ptr;
return NULL;}
```

-الجزء الخاص بالتحقق من نوع بيانات الرمز:

نحتاج هنا لتابع يقوم لدى استدعائه بإعادة نوع المتغير (الرمز) الذي نريد معرفة نوعه، وهذا التابع هو get_sym_type ويعرَّف كمايلي:

```
int get_sym_type(char *sym_name)// تابع الحصول على نوع رمز من جدول الرموز
{sym_node *ptr;
for(ptr=sym_table;ptr!=NULL;ptr=(sym_node*)ptr->next)
if(!strcmp(ptr->name,sym_name)) return ptr->type;
return 0;}
```

يقوم هذا التابع (الإجرائية) بأخذ كوسيط السلسلة sym_name وهي تمثل اسم الرمز المطلوب إعادة نوعه، وفي جسم التابع يتم تعريف مؤشر ptr من النوع sym_node ويتم بعدها البحث ضمن جدول الرموز عن الرمز ذو الاسم sym_name وعند العثور عليه يتم إعادة نوع هذا الرمز ptr->type وفي حال عدم العثور عليه يتم إعادة الـ 0.

تكتب كل هذه الأجزاء ضمن ملف واحد يحفظ بامتداد h. ويتم تضمينه لاحقاً في جسم ملف المعرب.

3- ملف وصف الماسح Scanner file:

تكتب تعليمات وصف الماسح كبرنامج واحد في ملف بامتداد (l.) لنتمكن من تنفيذها على بيئة الـ LEX، وبعد تنفيذها نحصل على ملف الماسح النهائي. يتألف برنامج وصف الماسح من مراحل التضمين والتصريح وتحديد القوالب والتوابع.

3-1 مرحلة التضمين والتصريح:

تكتب هذه المرحلة بين قوسين على الشكل التالي:{% }% للدلالة على أنها تعليمات بلغة C وتتضمن التصريح عن المتحولات في حال وجودها بالإضافة لعمليات التضمين، مثلاً يتم تضمين المكتبة <stdlib.h> نظراً لاستخدام التابع atoi ضمن ملف الماسح والذي ينتمي للمكتبة المذكورة. عموماً، يجب تضمين الملف الرأسي الذي ينتج عن المعرب ويحتوي مجموعة تعريفات الرموزtokens التي يستخدمها المعرب لبناء القواعد والتي يجب على الماسح scanner اسخدامها لمطابقة محارف الدخل وفقاً لها، كما يتضمن التصريح الكامل للمتحول yylval الذي يعتبر صلة الوصل بين الماسح والمعرب والتي يقوم من خلالها الماسح بتمرير قيم الرموز إلى المعرب. يمثل المقطع التالي مثالاً عن كيفية كتابة قسم التضمين [4]:

```
%{  #include<stdlib.h>
#include"C:\Lex_Yacc\examples\example1\expy2.h" %}
```

3-2 تحديد القوالب:

بعد مرحلة التضمين، تبدأ مرحلة بناء القوالب وتتضمن التصريح عن كل القوالب التي سيستخدمها الماسح لمطابقة رموز سلسلة الدخل وتحويلها إلى tokens. فيمايلي مثال عن تعريف ثلاثة قوالب (قالب الفراغات، الأرقام والأعداد).

```
blank       [ \t]+
number      [0-9]
entire      {number}+
%%
```

توضع إشارة %% للدلالة على انتهاء قسم القوالب، وبداية قسم جديد.

3-3 مرحلة القواعد أو تحديد استجابة الـ Scanner عند اكتشاف الـ Tokens:

يتم في هذه المرحلة تحديد استجابة الماسح وفق قواعد معينة، فإذا صادف الماسح مثلاً فراغ أو أكثر ضمن رموز الدخل فإنه لن يقوم بأي شيء (أي سيتجاهله) وفقاً للقاعدة التالية: {blank}

بينما سيقوم الماسح بإعادة المفردة المسماة NUMBER إلى المعرب، كما سيعيد معها قيمة هذا الرقم عبر المتحول yylval.

```
{entire}      {
               yylval.Tint=atoi(yytext);
              return(NUMBER);
              }
```

توضح التعليمات السابقة أن المتحول yylval ليس متحولاً عادياً بل هو عبارة عن سجل struct يحتوي عدة حقول وكل حقل من هذه الحقول له نوع بيانات مختلف حسب الرمز الذي يتم مسحه من رموز الدخل فلو كان الرمز من النوع الحقيقي لأصبحت التعليمات السابقة على الشكل:

```
{real}        {
              yylval.Treal=atof(yytext);
               return(REAL);
               }
```

وهذا يتطلب أن يكون المتحول yylval معرفاً كسجل على الشكل الآتي ضمن ملف المعرب:

```
%union{
charTstr[10];
int Tint;
floatTreal;}
```

أي أن المتحول yylval أصبح لديه ثلاثة حقول الأول من النمط char وهو مصفوفة مؤلفة من 10 عناصر، والثاني من النمط الصحيح integer، والثالث من النمط الحقيقي float.

3-4 مرحلة بناء ملف الماسح فعلياً:

بعد الانتهاء من كتابة ملف وصف الـ scanner باستخدام برنامج المفكرة، يتم تخزين الملف في مسار معين ثم باستعمال الأمر FLEXيولد برنامج الـ LEX شيفرة C للماسح scanner، أي أنه سنحصل بالنتيجة على ملف بامتداد C. يمثل الماسح يوضحه الشكل (3).

lex.yy.c

الشكل (3) ملف الماسح النهائي

4 - ملف وصف المعربParser file:

4-1 مرحلة التضمين والتصريح:

كما في ملف الماسح، فإن الجزء الأول من ملف الماسح مخصص لتضمين المكتبات والتصريح عن المتحولات اللازمة لعمل البرنامج.لكن المهم في هذه المرحلة هو تضمين ملف الـ C الذي ينتج من بناء ملف وصف الماسح ضمن بيئة الـ LEX، وهذا الملف يمثل الماسح والذي يجب تضمينه ضمن المعرب ليقوم المعرب باستخدامه لمسح رموز سلسلة الدخل أثناء بناء المترجم.

4-2 تحديد الأولويات وسلاسل الرموز:

يتم في هذه المرحلة تحديد أولويات العمليات الحسابية والمنطقية، والتعريف عن الرموز التي سيتم استخدامها في قواعد الإعراب لاحقاً. تمثل التعليمات التالية التصريح عن ثلاث مفردات خاصة بالجمع والطرح والأعداد.

%token NUMBER PLUS MINUS

بينما تمثل التعليمات التالية تحديد أولويات العمليات الحسابية بعد الكلمتين المفتاحيتين left و right:

%left PLUS MINUS
%right POWER

4-3 كتابة قواعد اللغة وأفعال المعاني Semantic Actions المرافقة لها:

تعتبر المرحلة الأهم في بناء ملف المعرب حيث تتضمن تحديد قواعد النحو التي سيتم على أساسها قبول عبارة ما أو رفضها.

ويوضح المثال التالي نموذجاً من قواعد اللغة المعمول بها في المترجم المقترح:

```
Exp|Exp PLUS Exp            {$$=$1+$3;}
```

تمثل هذه القاعدة عملية الجمع، في هذه القاعدة هناك نصف أيمن ونصف أيسر ولإسناد قيم أو قراءة قيم الرموز المكونة للقاعدة فإننا نستعمل $x:

```
EXP -> EXP   PLUS  EXP
  $$     $1    $2    $3
```

أي يتم ترميز EXP النتيجة بـ $$ أما EXP على الطرف الأيمن للقاعدة ترمز بـ $1 أما الـ EXP الثالثة ترمز $3 أما الـ TOKEN التي تتوسط القاعدة والممثلة بـ PLUS فإنها ترمز بـ $2.

الهدف الأساسي من هذه العملية هو إنجاز عملية الجمع بعد التحقق من صحة توافق تسلسل الرموز القادمة من الماسح مع هذه القاعدة، وتعتبر هذه العملية من أفعال المعاني Semantic Actions[4] ولا علاقة لها بمرحلة تحليل القواعد وهذه الميزة تسمح بها أداة BISON التي يتم استخدامها لبناء المعرب ويمكن أن نصف المترجم في هذه الحالة بالمترجم الفوري الذي سيتحقق من صحة العبارات ثم يقوم بحساب ناتج العمليات الحسابية والمنطقية.

4-4 التصريح عن التوابع اللازمة لعمل المعرب:

بقي لنا إضافة التوابع اللازمة لعمل المعرب والتي ستقود عمليتي تحليل المفردات (المسح) والقواعدي (الإعراب):

تابع الخطأ yyerror:

يقوم هذا التابع بطباعة رسالة الخطأ، ويتم التصريح عنه كمايلي:

```
int yyerror (char *s)
{printf("%s\n",s);}
```

تابع إنهاء المسح:

يستدعى هذا التابع من قبل الماسح عند نهاية تسلسل الدخل ليبلغ المعرب نهاية رموز سلسلة الدخل.

```
int yywrap(){return 1;}
```

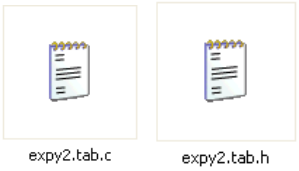
التابع الرئيسي:

يتم ترجمة المعرب من قبل لغة turbo c ولذلك سيحتاج المعرب إلى تابع رئيسي لتبدأ منه عملية الترجمة، والتعليمة التي نشاهدها دوماً في جسم التابع الرئيسي هي استدعاء المعرب ()yyparse، كذلك يتضمن التابع الرئيسي فتح ملف المدخلات input.txtمن خلال إعادة مؤشر إلى موقع هذا الملف وإسناده للمتحول yyin، وفي حال عدم وجوده تتم طباعة رسالة خطأ.

```
main()
{if((yyin=fopen("c:\\lex_yacc\\examples\\example1\\input.txt","r"))==NULL)
//الحصول على مؤشر لموقع ملف المدخلات في ذاكرة الحاسب
{printf("input.txt not found !\n");
return;}
yyparse();}
```

4-5 مرحلة بناء ملف المعرب فعلياً:

بعد الانتهاء من كتابة ملف وصف الـ Parser باستخدام برنامج المفكرة، يتم تخزين الملف في مسار معين ثم باستعمال الأمر d– Bison يولد برنامج الـ BISON شيفرة C للمعرب فنحصل على ملفين الأول بامتداد C. يمثل المعرب، أما الثاني بامتداد h. ويتضمن تعريفات الرموز tokens والتصريح الكامل للمتحول yylval.



الشكل (4) ملف المعرب والملف الرأسي الحاوي على تعريف الرموز

5- مرحلة تحليل المعاني:

يتم في هذه المرحلة التحقق من عدم وجود أخطاء المعاني وذلك بالاعتماد على جدول الرموز الذي جرى بناؤه خلال مرحلة تحليل المفردات،وتنجز هذه المرحلة فعلياً باستخدام ملف جدول الرموز مع ملف المعرب، وهذا يتطلب تعديلات جوهرية في ملف المعرب لتجعله قادراً على إنجاز عملية تحليل المعاني، وهذه التعديلات هي التالية:

تعديل ملف المعرب:

يجب تعديل ملف المعرب ليتلاءم مع التغييرات الجديدة والتي تتطلب إضافة الرموز إلى جدول الرموز أو التحقق من وجود رمز ضمن جدول الرموز لتجنب إجراء عملية إسناد خاطئة أو عملية تصريح خاطئة.

هناك العديد من أخطاء المعاني التي يمكن أن نصادفها وهي:

التصريح المكرر عن الرموز، إسناد قيمة لرمز غير معرف ضمن جدول الرموز، أو إسناد قيمة من نوع مخالف لنوع الرمز المعرف ضمن جدول الرموز.

هذا يتطلب إضافة التعديلات التالية إلى ملف المعرب:

5-1التأكد من عدم تكرار التصريح عن نفس الرمز:

في حالة التصريح عن رمز جديد يجب ضمان أن الرمز غير مصرح عنه قبل إضافته لجدول الرموز. هنا يتم تعريف تابع للتحقق من وجود الرمز ضمن جدول الرموز وفي حال عدم وجوده يتم إدخاله إلى جدول الرموز، أما في حال وجوده يتم استدعاء تابع الخطأ.

ويعرف هذا التابع ضمن قسم التصريح في ملف المعرب أي قبل البدء بكتابة قواعد الإعراب:

```
voidsetup_sym(char* sym_name)
{sym_node *sym;
sym=get_sym(sym_name);
if(sym==NULL) put_sym(sym_name);
else
{
errors++;
printf("Error %d :: %s Identifier is defined previously: line%d.\n",errors,sym_name,line);
}}
```

تابع setup_sym يقوم بنداء التابع get_sym (المعرف في جسم ملف جدول الرموز) ليتحقق فيما إذا كان الرمز sym_name موجود ضمن جدول الرموز فإذا كانت القيمة المعادة منه NULL فهذا يعني أن الرمز غير موجود عندها تتم إضافته باستخدام التابع put–sym(المعرف في جسم ملف جدول الرموز)، أما إذا لم تكن القيمة المعادة NULL فهذا يعني أن الرمز موجود لذا يطبع التابع رسالة خطأ تفيد بأن الرمز معرف وموجود مسبقاً ولا يجوز التصريح عنه من جديد.

5–2التأكد من عدم إسناد قيمة لرمز غير معرف:

عند إسناد قيمة لرمز ما، يجب التأكد من أن الرمز موجود بالفعل ضمن جدول الرموز، حيث يتم تعريف تابع للتحقق من وجود الرمز ضمن جدول الرموز وفي حال عدم وجوده يعطي رسالة خطأ، وفي حال وجوده يتم إسناد القيمة إلى هذا الرمز.

```
intsym_check(char* sym_name)
{if(get_sym(sym_name)==NULL)
{errors++;
printf("Error %d :: %s Identifier is not known: line %d.\n",errors,sym_name,line);
return 0;}
return 1;}
%}
```

يقوم تابع sym_check بالتحقق من أن الرمز sym_name الممرر كوسيط له معرف أو لا ضمن جدول الرموز، يتم ذلك من خلال مناداة التابع get_sym باستخدام الرمز sym_name كوسيط لهذا التابع فإذا كانت القيمة المعادة من التابع هي NULL فهذا يعني أن الرمز غير معرف وستتم طباعة رسالة تفيد بأن الرمز غير معرف مسبقاً، ولا يجوز إسناد قيمة لمتغير غير معرف.

5–3 التأكد من تطابق نوع البيانات خلال عمليات الإسناد:

في حالة إسناد قيمة لرمز ما، يجب التأكد من أن نوع البيانات لهما متطابق، وهذا ما يتم تأمينه عن طريق التصريح عن متحول يحمل رقم النوع، يصرح عن هذا المتحول ضمن قسم التصريحات في ملف المعرب كمايلي:

```
int current_type;
```

وتتم تعديل قيمة هذا المتحول كلما صادف الماسح تصريحاً عن رمز جديد ضمن رموز الدخل، حيث أن لكل رمز نوع خاص به. توضح التعليمات التالية كيفية تعديل قيمة المتحول current_type:

```
decl_type:DEC_INT {current_type = _int;}
|DEC_REAL {current_type = _float;}
|DEC_CHAR {current_type = _chr;}
|DEC_CHAIN {current_type = _str;};
```

تمثل التعليمات السابقة قواعد إعراب ضمن جسم المعرب، وبجانب كل قاعدة هناك تعليمة بلغة الCفمثلاً لدى مصادفة المعرب للقاعدة decl_type:DEC_INT سيتم تغيير قيمة المتحول current_type لتصبح int_ أو الرقم 1، ولدى مصادفة المعرب للقاعدة decl_type:DEC_CHAIN سيتم تغيير قيمة المتحول current_type لتصبح str_ أو 4 وهكذا..

5-4 تعديل قواعد الإعراب لإنجاز مرحلة تحليل المعاني:

بعد التصريح عن كل التوابع والمتغيرات اللازمة لاستخلاص أخطاء المعاني بقيت مرحلة أخيرة تتضمن تعديل قواعد الإعراب بحيث نتمكن من كشف هذه الأخطاء اعتماداً على التوابع التي قمنا ببنائها في جسم المعرب وجدول الرموز.

1- يتم بداية تعديل القواعد الخاصة بمرحلة التصريح كمايلي:

```
ident:ID {setup_sym($1,current_type);}
|ID ident {setup_sym($1,current_type);};
```

هنا وعند كل ظهور للرمز ID ضمن مرحلة التصريح عن المتحولات، يتم تسجيل الرمز ضمن جدول الرموز فقط في حال كونه لم يكون معرف مسبقاً. يتم ذلك من خلال استدعاء التابع setup_sym باستخدام $1 والتي تمثل الرمز IDوذلك من أجل تثبيت هذا الرمز ضمن جدول الرموز، فإن كان موجود بالفعل ستطبع رسالة خطأ، وإن كان غير موجود ستتم إضافته.

2- ثانياً، يتم تعديل القواعد الخاصة بمرحلة الإسناد كمايلي:

تمثل القاعدة exp:ID EQ ID SEMICOLON حالة إسناد رمز إلى رمز آخر مثل a=b وفي هذه الحالة لابد من التحقق من أن الرمزين موجودين بالأصل ضمن جدول الرموز، ثم يجب التأكد من أن الرمز a من نوع بيانات مطابق للرمز b. وهذا يعني استدعاء التابع sym_check باستخدام كلا الرمزين مرتين متتاليتين، ثم استخدام التابع get_sym_type للتأكد أن نوعي البيانات لهما متطابقان، وفي حال عدم التطابق سيتم طباعة رسالة خطأ.

في حالة عمليات الإسناد من الشكل a=100، فإنه يجب تعديل القاعدة الخاصة بالأرقام الصحيحة كمايلي:

```
ID EQ NUM SEMICOLON{
sym_check($1);
if(get_sym_type($1)!=1)
{errors++;
printf("Error %d :: %s is not declared as integer type:line %d.\n",errors,$1,line);}}
```

هنا يتم التأكد من أن الرمز $1 معرف ضمن جدول الرموز من خلال sym_check($1)، يتم التحقق من أن الرمز ID له نفس نوع بيانات الرقم NUM أي النوع الصحيح من خلال تعليمة الشرط if(get_sym_type($1)!=1)، حيث تتم طباعة رسالة خطأ في حال عدم التطابق.

تكرر هذه العملية مع بقية الأنواع (الحقيقي والسلسلة والمحرفي) لضمان صحة عملية الإسناد في بقية أنواع البيانات.

يتضمن الملف المصدري عبارات مثل a=b+c; أو a=b/c أو a=b and c وغيرها الكثير من العبارات المماثلة، وللتعامل مع هذه العبارات يجب تعديل القواعد لتتلاءم مع هذه العمليات الحسابية والمنطقية، والقواعد الواجب إضافتها لملف المعرب:

```
exp:ID EQ ID PLUS ID SEMICOLON {
sym_check($1);//التحقق من وجود الرمز الأول ضمن جدول الرموز
sym_check($3);//التحقق من وجود الرمز الثاني
sym_check($5);//التحقق من الرمز الثالث
```

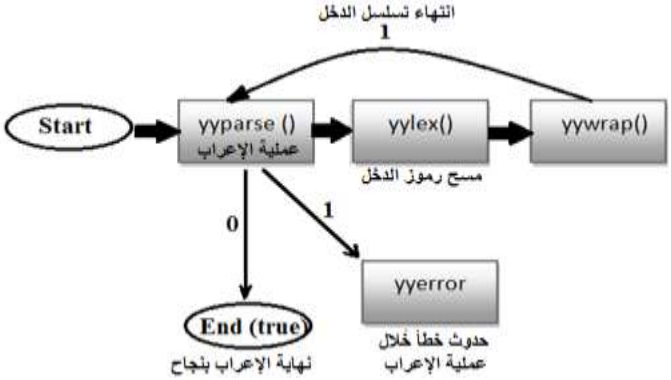
```
if(get_sym_type($1)!=get_sym_type($3)  ||get_sym_type($3)!=get_sym_type($5)  ||
get_sym_type($1)!=get_sym_type($5)// التحقق من أن الرموز الثلاثة لها نفس نوع البيانات
{// طباعة خطأ في حال عدم تطابق أنواع البيانات
errors++;
printf("Error %d :: %s and %s have different types :line %d.\n",errors,$1,$3,line);}
else if(get_sym_type($1)==3 || get_sym_type($1)==4)
{ // طباعة خطأ في حال كان الرمز المصدر محرفاً أو سلسلة حيث لا يمكن جمع هذه الأنماط من البيانات
errors++;
printf("Error %d :: can't add string or char variables! :line %d.\n",errors,line);
}};
```

6 – توليد المترجم:

بعد الحصول على ملفي الماسح والمعرب تبدأ مرحلة توليد المترجم النهائي من خلال ترجمة ملف المعرب باستخدام لغة turbo C++ وبما أن ملف الماسح مضمن في ملف المعرب لذلك ليس هناك داعي لترجمة ملف الماسح.

لدى ترجمة ملف المعرب تبدأ عملية الترجمة من التابع الرئيسي الذي يستدعي المعرب والذي بدوره يستدعي الماسح ليمسح رموز الدخل، وعند انتهائه من مسح سلسلة الدخل يستدعي الماسح التابع yywrap() الذي يعيد قيمة 1 للمعرب، وهنا تبدأ مرحلة التحليل القواعدي (الإعراب) وفي حال انتهت بنجاح يعيد المعرب قيمة الصفر للتابع الرئيسي، بينما يعيد قيمة 1 ليشير إلى وجود خطأ، وفي هذه الحالة يقوم التابع الرئيسي باستدعاء تابع الخطأ.

يوضح الشكل (5) تسلسل عملية استدعاء التوابع وصولاً إلى مرحلة المترجم النهائي الموضح في الشكل (6).



الشكل (5) تسلسل استدعاء التوابع خلال مرحلة الإعراب

الشكل (6) المترجم النهائي

النتائج والمناقشة:

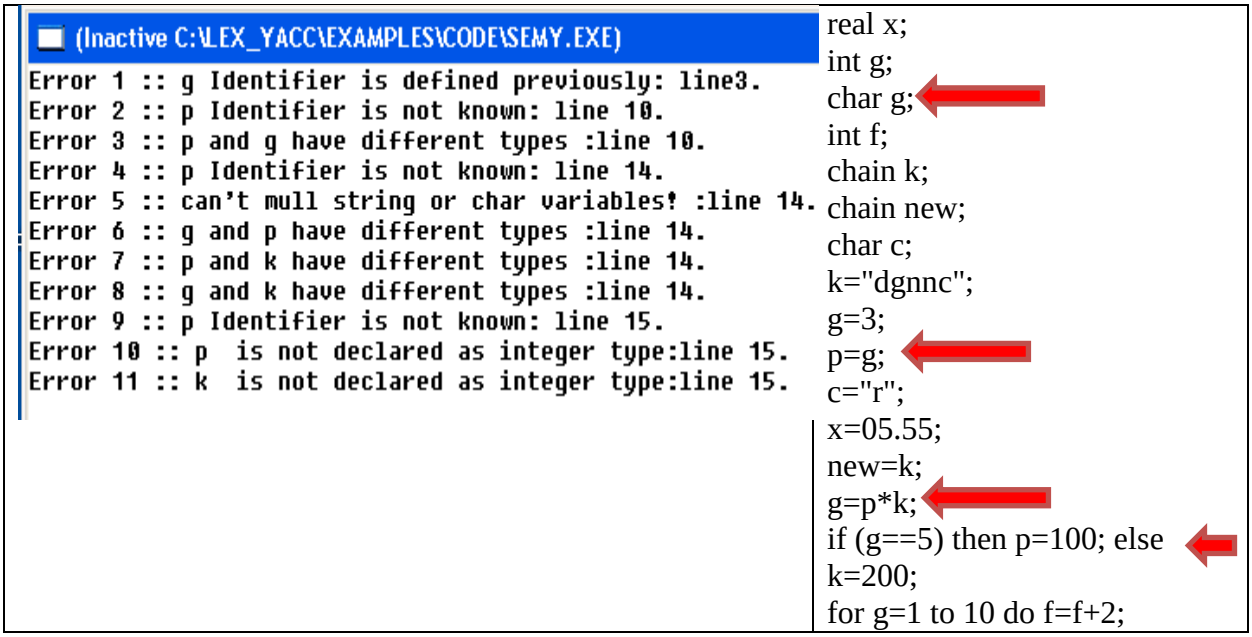
تم استخدام الأدوات البرمجية LEX, BISON ولغة الـ C++ لبناء المترجم المقترح برمجياً، ثم اختُبِر المترجم المصمم على 100 ملف مصدري مختلف. تضمنت هذه الملفات المصدرية المقترحة تعليمات التصريح والإسناد

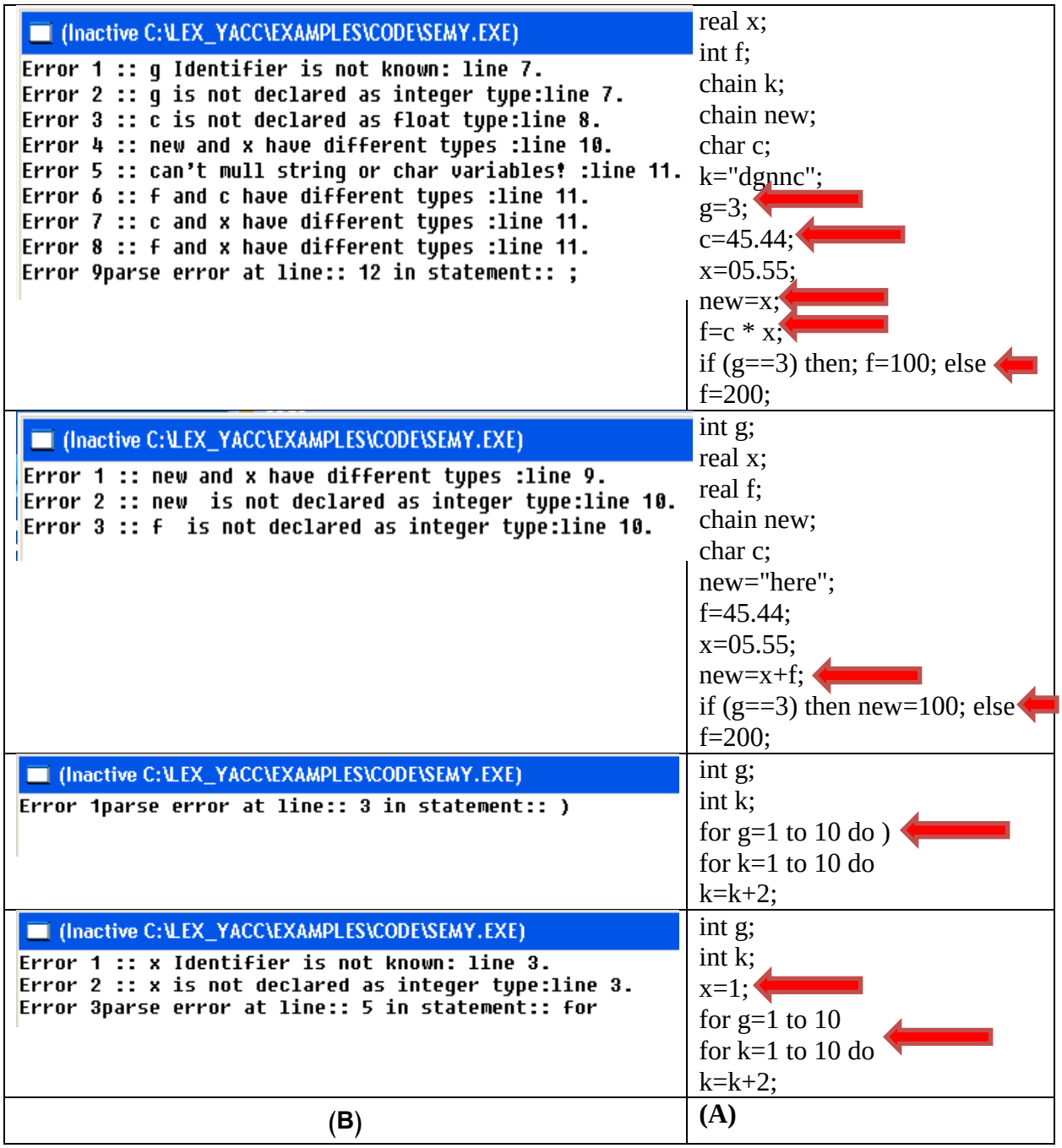
بمختلف أنواعها وأشكالها، إضافة إلى الحلقات وبنى التحكم، وقد تمكن المترجم المصمم من ترجمة جميع هذه الملفات بنجاح، كما تم اختباره على عدد من الملفات التي تحوي أخطاءً لفظية وقواعدية وأخطاء معاني، وقد تمكن المترجم من كشف جميع هذه الأخطاء. يوضح الشكل (7) نماذجاً من الملفات المصدرية التي لا تحوي أخطاء والتي تمت ترجمتها بنجاح.

int g;int p;
int f;chain k;
chain new;char c;
k="dgnnc";
g=3;
p=g;
c="r";
new=k;
g=g*p;
if (g==5) then p=100; else
p=200;
for g=1 to 10 do f=f+2;
	real x; int d;
	char g;char f;
	chain k;
	k="HERE";
	g='a';
	f='b';
	x=05.55;
	x=x*100;
	for d=1 to 10 do x=x*2.2;
	if(g=='a') then f='b' else
	f='c';
	while(d<10) {x=x/10;
	d=d+1;}
		real x; int g;
		int p;chain k;
		chain new;
		k="computer";
		g=3;
		p=g;
		x=05.55;
		new=k;
		g=g*p;
		if (g==5) then p=100; else
		p=200;

الشكل (7) نماذج من الملفات المصدرية المستخدمة في الاختبار

في حين يوضح الشكل (8) عدداً من الملفات المصدرية الحاوية على أخطاء قواعدية وأخطاء معاني وكيفية اقتناص المترجم لهذه الأخطاء.

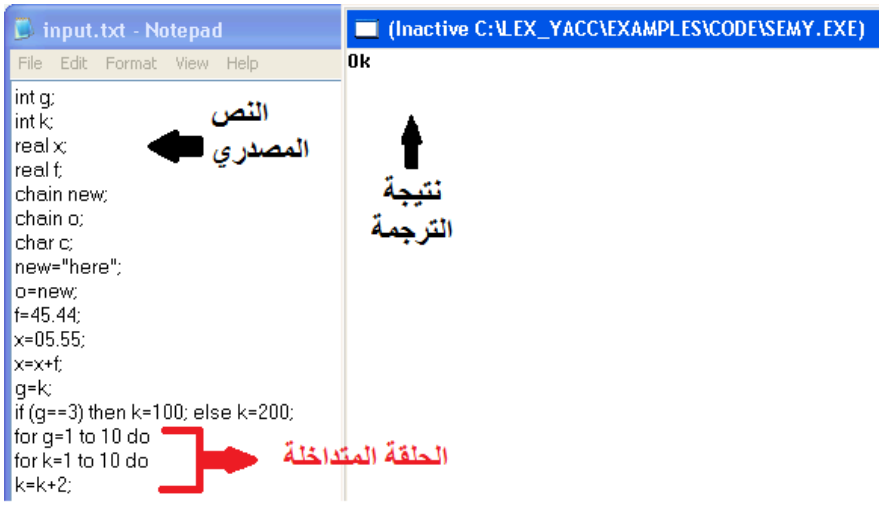




الشكل (8) نماذج من الملفات المصدرية المستخدمة في الاختبار والحاوية

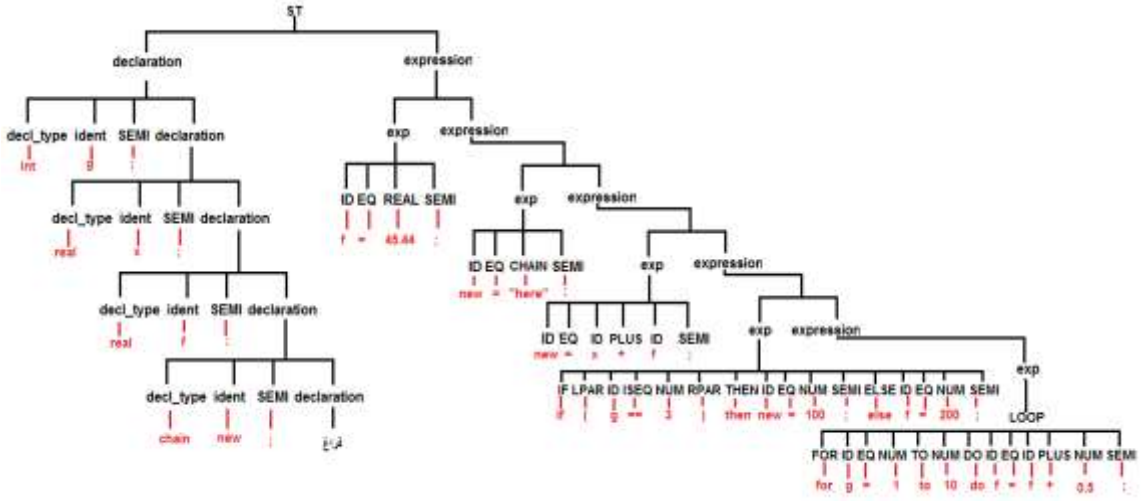
على أخطاء قواعدية وأخطاء معاني: (A): الملف المصدري، (B): نتيجة تنفيذ المترجم

بينما يوضح الشكل التالي اختبار المترجم على الحلقات المتداخلة حيث يظهر تماماً كيفية تمكن المترجم من إعراب الحلقات المتداخلة بنجاح:



الشكل (9) كيفية استجابة المترجم للحلقات المتداخلة

يوضح الشكل (9) شجرة الإعراب الخاصة بأحد الملفات المصدرية التي تم استخدامها في الاختبار:



الشكل (9) شجرة الإعراب الخاصة بأحد الملفات المصدرية التي تم استخدامها في الاختبار

حيث أن شجرة الإعراب السابقة موافقة للملف المصدري التالي:

```
int g;real x; real f;chain new;
new="here";
new=x+f;
f=45.44;
if (g==3) then new=100; else f=200;
for g=1 to 10 do f=f+0.5;
```

يتضمن الجدول (2) عدداً من الملفات المصدرية وعدد الأخطاء ضمنها ونسبة كشف الخطأ من قبل المترجم، حيث أن الأخطاء التي لم يكشفها المترجم كانت أخطاء في شكل البرنامج مثل إضافة نقطة لنهاية التعليمة، ويمكن التخلص منها بتعديل مرحلة تحليل المفردات، أما الأخطاء القواعدية والمعنوية فكشفت جميعاً.

جدول (2) نسبة كشف الخطأ في عدد من الملفات المصدرية:

الملف	عدد الأخطاء الكلية	عدد الأخطاء المكتشفة	نسبة كشف الخطأ %
1	4	4	%100
2	5	5	%100
3	2	2	%100
4	10	10	%100
5	15	14	%93.3
6	20	19	%95
7	25	25	%100
8	27	26	%96.2
9	34	32	%94.11

الاستنتاجات والتوصيات:

تم في هذه الدراسة تصميم مترجم للغة مصدرية تتضمن جميع العمليات الحسابية والمنطقية وبنى التحكم والحلقات وعمليات الإسناد والتصريح. جرى بناء هذا المترجم بدءاً من مرحلة تحليل المفردات إلى التحليل القواعدي وصولاً لمرحلة تحليل المعاني وقد تمكن المترجم المصمم من ترجمة أشكال مختلفة من الملفات المصدرية وكشف مواقع الأخطاء فيها بنجاح.

المراجع:

[1] A.V. AHO, R. SETHI, J.D. ULLMAN,"Compilers: Principles, Techniques and Tools", Addison-Wesley, 1986.

[2] ANTHONY A. AABY, *"Compiler Construction using Flex and Bison"*, Walla Walla College, February 25, 2004, pp:1-18.

[3] F.J.F. BENDERSET AL., *"Compiler Construction A Practical Approach"*, Macmillan Technical Publishing, January 29, 2003, pp:21-16.

[4] J. Levine, *"Lex and Yacc"*, O'Reilly & sons, 2000.

[5] P.H. SALUS,*"Handbook of Programming Languages, Volume I: Object oriented Programming Languages"*, Macmillan Technical Publishing, 1998.

[6] P.H. SALUS,*"Handbook of Programming Languages, Volume II: Imperative Programming Languages"*, Macmillan Technical Publishing, 1998.

[7] P.H. SALUS,*"Handbook of Programming Languages, Volume III: Little Languages and Tools"*, Macmillan Technical Publishing, 1998.

[8] P.H. SALUS,*"Handbook of Programming Languages, Volume IV: Functional and Logic Programming Languages"*, Macmillan Technical Publishing, 1998.

[9] Cohen, Albert, "Compiler Construction", Grenoble, France, April 5-13, 2014.

[10] ياسين الجزائري، "كتاب كيف تبني مترجماً، فهم برمجة الكومبايلر خطوة بخطوة"، دار القاهرة للنشر، 2010.