

**References**

"Compilers Principles, techniques, & tools", by Alfred Aho, second edition, Pearson Addison Wesley, 2007.

**Compiler & translator**

A translator is a program that take as input a program written in one programming language (source language ) and produce as output a program in another language ( object ) program or target).

Source Program

Target or object Program

Translator

Example C++ machine (0,1)

**Types of translator**

There are four types of translator according to the source language as well as target language:

Translator

Source Program

Target or object Program

Compiler

High Level Language Low Level Language

 (assembly or machine language)

Interpreter

High Level Language Intermediate code

(directly execution)

assembler

Assembly Language machine Language

Preprocessor

High Level Language **L1** High Level Language **L2**

HW-1/ let G be the following grammar:

 S → ScS / SbS / a

* Is the string "acabaca" in L(G) ? How?
* Is the grammar ambiguous or not? Why?

**Structure of compiler**

The process of compilation is so complex to be achieved in one single step, either from a logical point of view or from an implementation point of view. For this reason it is partition into a series of sub processes called ***phases***. The typical compiler consists of several phases each of which passes its output to the next phase plus symbol table manager and an error handler as shown below:

*Source Language*

Symbol Table management

Error

Handler

*Target Language*

Code generation

Code Optimization

Intermediate code generation

Semantic Analyzer

Syntax Analyzer

Lexical Analyzer

**Lexical Analyzer :**

It is the first phase of the complier and also called scanner, which represent the interface between source program and the compiler.

Lexical read the source program by one letter at a time and decomposed it into group of letters called TOKEN that is logically interrelated between them. This token represent string of characters that can be processed together as an logical input.

Separates characters of the source language program into groups (sets) that logically belong together. These groups are called Tokens. Tokens are Keywords, Identifiers, Operator Symbols, and Punctuation Symbols. The output of the analyzer is stream of tokens, which is passed to the syntax analyzer.

**Tokens:** groups of characters of source language program logically belong together. For example:

Main( )

{

Int a,b,c ;

a=7;

b=8;

c=a+b

If ( c=5 ) cout<< c;

}

Lexical

Analyzer

Source program Stream of token

 Error

**Type of tokens**

There are five types of token as below:

|  |  |
| --- | --- |
| Type of token | Examples |
| Keyword | Main , int , cin , if , for , cout , while , else |
| Identifier | A , B , C , sum , a1 , b45 |
| Constant | 5 , 12.5 , 1.3 , 0.3E5 , 200 |
| Operator | (arithmetic) : + , - , \* , / , ^ |
| (logical) : > , < , = , <= , >=  |
| punctuation | , ; : " ' ( ) $ @ . { }  |

Example of lexical analyzer of the above C++ code will produce the following output on the table symbol as:

Token Type Code Value

main keyword 1 1

{ punctuation 5 12

int keyword 1 2

a Identifier 2 *pointer to memory location1*

b Identifier 2 *pointer to memory location2*

c Identifier 2 *pointer to memory location3*

 ; punctuation 5

7 Constant 3 7

8 Constant 3 8

= Operator (*logic*) 4 8

+ Operator (*arithmetic*) 4 1

Cout<< keyword 1 6

} punctuation 5 12

HW-2/ Let the following Fortran statement:

 IF ( 5.EQ.A) Goto 20

* What the output of lexical of Fortran compiler?

***Syntax Analyzer (Parser):*** Groups tokens together into syntactic structures called ***Expressions***. Expressions might further be combined to form Statements. Often the syntactic structure can be regarded as a ***Tree*** whose leaves are the tokens. The interior nodes of the tree represent strings of tokens that logically belong together.

وهي المرحلة اللاحقة لمرحلة تحليل النص وتسمى احيانا parser او العارب حيث يقوم بالاعراب حيث ياخذ مخرجات المرحلة السابقة (stream of token) سلسلة من الـtoken ويحولها الى شجرة الاعراب (parse tree) لغرض تحليل الجمل من الناحية القواعدية.

***Semantics:*** is a term used to describe "meaning", and so the constraint analyzer is often called the static semantic analyzer, or simply the semantic analyzer. The output of the syntax analyzer and semantic analyzer phases is sometimes expressed in the form of a **Abstract Syntax Tree** (AST). This is a very useful representation, as it can be used in clever ways to optimize code generation at a later phase.

It is Recently added a stage to analyze in terms of sentence meaning, in other words, will reject the sentence in the case of the discovery of the meaning of error (Note: The sentence can be true in terms of the syntax but have no meaning).

For example the English sentence : "The man eat the house " This is in the natural language, put in programming languages ​​can either write inter correct the hand, but wrong in the sense rules such as the BASIC program:

وهي مرحلة اضيفت حديثاً لتحليل الجملة من ناحية المعنى، أي بمعنى اخر الجملة سوف ترفض في حالة اكتشاف معنى خطأ ( ملاحظة : يمكن ان تكون الجملة صحيحة من ناحية القواعد لكن ليس لها المعنى ).

The man eat the house"" هذا في اللغات الطبيعية اما اللغات البرمجية يمكن نكتب جملة صحيحة من ناحية القواعد لكن خطأ بالمعنى مثل برنامج البيسك :

10 a=30

20 goto a

30 print a \* a

40 end

ناتج هذه المرحلة هي نفس شجرة الاعراب لكن خالية من الاخطاء.

***Intermediate Code Generator:*** Create a stream of simple instructions. Many style of intermediate code are possible. One common style uses instructions with one operator and a small number of operands.

This phase is used structures built in the previous stages and convert them into intermediate codes by using a most famous meta language such as:

- Postfix notation

- Polish notation

 In some versions be enforceable as is the case in the interpreter for the BASIC language and commands interpreter of the DOS operating system.

تستخدم هذه المرحلة الهياكل المبنية في المراحل السابقة وتحويلها الى رموز وسطية باستخدام لغة افتراضية اشهرها

* Postfix notation
* Polish notation

 في بعض الاصدارات تكون قابلة للتنفيذ كما هي الحال في المفسر interpreter للغة بيسك ومفسر ايعازات نظام التشغيل DOS .

***Code Optimization***: Is an optional phase designed to improve the intermediate code so that the ultimate object program runs faster and/or takes less space.

وهي المرحلة التي تلي مرحلة توليد الرموز الوسطية، وتكون اختيارية في بعض نسخ المترجمات، وظيفتها الاساسية هي تبسيط الرموز الناتجة في المرحلة السابقة الى الحد الادنى وبذلك سيكون البرنامج اسرع ويأخذ مساحة ذاكرة اقل، مخرجات هذه المرحلة هي ايضا بالرموز الوسطية لكنها مبسطة وتشبه عمل الرموز الواردة في المراحل السابقة.

***Code Generation:*** In a real compiler this phase takes the output from the previous phase and produces the object code, by deciding on the memory locations for data, generating code to access such locations, selecting registers for intermediate calculations and indexing, and so on.

وهي المرحلة الاخيرة من مراحل عمل المترجم التي يقوم بها انتاج الرموز الهدف object code وتقرير مساحة الذاكرة المطلوبة ومواقع المتغيرات في الذاكرة، وتحديد المسجلات register اللازمة لاجراء كل عملية ، ان تصميم هذه المرحلة لتعمل بشكل كفوء تعتبر عملية صعبة للغاية من الناحية العلمية والنظرية عند تصميم المترجم.

***Table Management:*** Portion of the compiler keeps tracks of the name used by the program and records essential information about each, such as its type integer, real, … etc. the data structure used to record this information is called a ***Symbolic Table***.

وهي روتينات فرعية مساعدة تستخدم لإدارة عملية الادخال الى جدول الرموز وكذلك عمليات المقارنة مع الرموز المخزونة في الجدول. حيث يحتوي الجدول على اسماء المتغيرات ونوعيتها مثل (integer , real , . . . ) وكذلك الحجز في الذاكرة (memory allocation) التي تحتاجها كل المراحل الاخرى.

***Error Handler:*** One of the most important functions of a compiler is the detection and reporting of errors in the source program. The error messages should allow the programmer to determine exactly where the errors have occurred. Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic message.

تستخدم هذه الروتينات لتشخيص الاخطاء التي تصاحب عملية الترجمة ، وهذه المرحلة تعطي رسائل تحذيرية الى المبرمج عن نوعية وطريقة الخطأ. كما تقوم بتصحيح بعض الاخطاء التي تتناقل بين مراحل المترجم، يفضل ان نجعل عملية متابعة تشخيص الاخطاء خلال مرحلة syntax لذلك فان اكثر الاخطاء يتم اكتشافها في مرحلة ترجمة واحدة.

***Passes of Compiler***

The previous phases showing the main parts of the compiler theoretically, either in the case of the application must merge combine more than one phase in round one and this is called one pass. Each pass will read the output of the previous pass ( except for the first pass ) that reads the source program.

In the case of integrating more than one stage in one round will lead to interference in the work of these stages through the control structure between these stages.

The influential factor in the number of stages per pass depends on the structure of the source language , the translator , who owns multi-pass uses less memory space of a translator who is in round one single pass. On the other hand , the multi-pass type slower than the another because each stage reads and writes in the intermediate file leads to the slow work of the translator . Must be a balance between the number of stages in a single pass and between the speed of the compiler. And on this basis can be lexical in one pass with syntax or separated .

When a lexical phase are separate from syntax pass there are four pros :

1 - the design of each model separately and thus achieved simplicity.

2 - be efficiency translator largest so that you can manipulate each part in order to achieve the desired efficiency .

3 - including that the lexical is the interface for that is change when you change the language and vocabulary development process.

4 – can make symbol table more structure in which we can use it as input and output in the same time.

**Notice**: If phase lexical built with syntax In this case, the Lexical in the form of a subroutine assistant called from the phase of syntax that followed , and if their separation will store the results of intermediate file, this file is called symbol table.

lexical

Syntax

Semantic

Intermediate code generation

Code Optimization

Code Generation

*Intermediate File*

*Intermediate File*

ان المراحل السابقة تبين الاجزاء الرئيسية للمترجم من الناحية النظرية ، اما في حالة التطبيق فيجب دمج combine اكثر من مرحلة في جولة واحدة وهذا يسمى one pass. كل جولة سوف تقرأ مخرجات الجولة السابقة لها ( عدا الجولة الاولى ) التي تقرأ البرنامج المصدر source program.

 وهذا سيؤدي الى ان التناقل بين الجولات عن طريق ملف وسطي intermediate file يكتب فيه ويقرأ منه (Read from , Write in .

في حالة دمج اكثر من مرحلة في جولة واحدة سيؤدي الى تداخل في عمل هذه المراحل عن طريق هيكل سيطرة بين هذه المراحل.

ان العامل المؤثر في عدد المراحل في الجولة الواحدة يعتمد على هيكل اللغة المصدر، المترجم الذي يمتلك جولات متعددة multi-pass يستخدم مساحة ذاكرة اقل من المترجم الذي يكون بشكل جولة واحدة single pass. ومن ناحية اخرى فان النوع multi-pass ابطء من الثاني لانه كل مرحلة تقرأ وتكتب في ملف وسطي يؤدي الى بطيء عمل المترجم. فيجب الموازنة بين عدد المراحل في الجولة الواحدة وبين سرعة المترجم. وعلى هذا الاساس يمكن ان يكون lexical في جولة واحدة مع syntax او ان يفصل بينهما.

عند فصل مرحلة lexical عن مرحلة syntax هناك اربع محاسن:

1. تصميم كل نموذج على حدة وبذلك تحققت البساطة.
2. تكون كفاءة المترجم اكبر بحيث تستطيع التلاعب بكل جزء حتى تحقق الكفاءة المرجوة.
3. بما ان الـ lexical هو الواجهة لذلك يتم التغيير فيه عند تغير مفردات اللغة وعملية التطوير.
4. نجعل جدول الرموز symbol table اكثر هيكلية structure بحيث تستخدمه ادخال واخراج في نفس الوقت (input & output).

ملاحظة: اذا كانت مرحلة lexical مدمجة مع syntax ففي هذه الحالة يكون الـ Lexical على شكل روتين فرعي مساعد يستدعى من مرحلة syntax التي تليها ، واذا تم الفصل بينهما سيتم خزن النتائج الوسطية على ملف وسطي يسمى intermediate file يسمى هذا الملف symbol table (جدول الرموز).

General Example of compilation phases

Consider the following sentence is segment from source program:

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

Lexical Analysis

**id1 := id2 + id3 \* 60**

Syntax Analysis

**Id3**

**Id2**

**+**

**\***

**60**

**Id1**

**:=**

Semantic Analysis

**Id3**

**Id2**

**+**

**\***

**60**

**Id1**

**:=**

intermediate Analysis

**temp1:=integer to real (60)**

**temp2:=id3 \* temp1**

**temp3:=id2 + temp2**

**id1:= temp3**

Code optimization

**temp1:= id3 \* 60.0**

**id1:= id2+ temp1**

Code generation

**movf id3 ,R2**

**mulf 60.0 ,R2**

**movf id2 ,R1**

**addf R1,R2**

**movf R2,id1**

HW/ convert the following equation over all compiler phases:

C:=A\*B-D\*5

***How Lexical Work***

**Input Buffering:**

The lexical analyzer scans the characters of the source program one at a time to discover tokens; it is desirable for the lexical analyzer to read its input from an input buffer.

We have two pointers one marks to the beginning of the token begin discovered. A look-ahead pointer scans a head of the beginning point, until the token is discovered.

يقوم محلل النص بقراءة النص المدخل بطريقة حرف واحد في الوقت الواحد لذلك فانه يحتاج الى مؤشرين الاول يسمى مؤشر البداية start pointer والاخر يسمى مؤشر التقدم للأمام look ahead pointer كما في الشكل الاتي:

Input Buffer

Main( )

**Start Pointer**

**Look ahead Pointer**

وعلى سبيل المثال اذا كان النص المدخل برنامج بلغة Fortran كما يأتي:

IF (5.EQ.sum) Goto 100

**Start**

**Look ahead**

***الية عمل محلل النص***

يقوم محلل النص بوضع المؤشر الاول على الحرف (I) ويبدأ المؤشر الثاني بالتحرك الى امام حتى يقف عند القوس ) حيث ان المحلل سوف يكتشف انها كلمة IF لذلك فان الحروف الفاصلة بين المؤشرين تمثل الtoken فيتم تحديد نوعه ويتم ادخاله الى جدول الرموز symbol table ثم يتم نقل مؤشر البداية الى القوس المفتوح ) ويبدأ المؤشر الاخر بفحص بقية الحروف وصولاً الى الرقم 5 ثم الى E والى هذه المرحلة لا زال محلل النص يتوقع بانه يقرأ رقم مكتوب بصيغة Floating point format او صيغة النقطة العائمة .

النقطة العائمة floating point :

**1.73 × 103 ≡ 1.73E3**

**0.173 × 104 ≡ .173E4**

**0.0173 × 105 ≡ 0.0173E5**

وعندما يقرأ Q هنا يتوقف المحلل عن البحث وسوف يرجع الى بداية الـtoken ليبدأ القراءة من جديد على انها 5 وهكذا الى نهاية النص او العبارة البرمجية كما في الشكل

token type

IF keyword

( punctuation

5 Constant

.EQ. Logical Operator

Sum Identifier

) punctuation

Goto keyword

1000 Constant

Input Buffer وهي مساحة تخصص من الذ اكرة لوضع block من النص المدخل فيه ومن ثم التعامل معه. لانه سيكون اسرع من ما يتم قراءة حرف بعد حرف من الملف. ان حجم هذا المخزن الوقتي يؤثر بشكل كبير على سرعة المحلل كما ياتي:

1. اذا اخذنا حجم صغير قد تكون كمية المعلومات اكبر من حجمه لذلك لا نستطيع تمييزه.
2. اذا اخترنا مساحة اكبر من اللازم تؤدي الى هدر بمساحة الذاكرة لذلك يجب الموازنة بين الحالتين.

الوظائف الاساسية لمحلل النص lexical

1. تحليل النص الداخل الى مجموعة من ال tokens.
2. اكتشاف بعض الاخطاء.
3. يحافظ على ارقام الاسطر عندما نحتاج الى line number كما في لغة بيسك.
4. يقوم في بعض اللغات بالغاء جميع الفراغات spaces او ال Tab.
5. في معظم اللغات التي تسمح باستخدام الملاحظاتcomment يوم محلل النص بحذفها من البرنامج.
6. عملية ارجاع محتويات الماكرو macro اي يعوض عن الدالة مباشرة (التعويض expansion).
7. بعض المخرجات output تظهر من قبل محلل النص مثل بعض رسائل الاخطاء.

مراحل تصميم محلل النص

 لبناء محلل نص جيد يجب توفر المعلومات الاتية:

1. طريقة لتوصيف الtoken وغالباً ما نستخدم regular expression او التعابير المنظمة.
2. طريقة لتمييز الtoken لذلك نسنخدم ميكانيكية خاصة تسمى (FA) Fint Automata.
3. طريقة أو الية لادخال المعلومات الى جدول الرموز (symbol table).