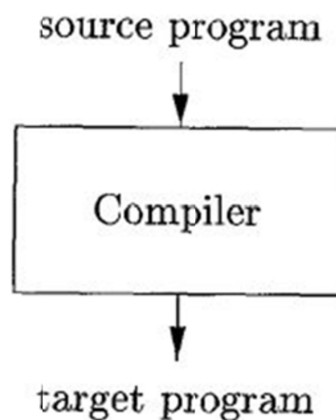
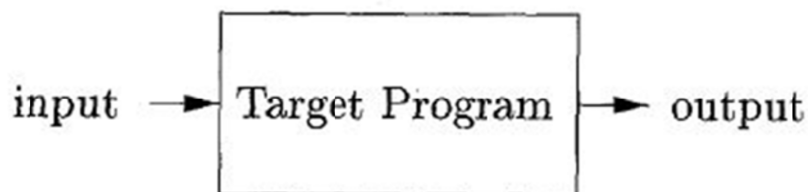
Language Processors

a compiler is a program that can read a program in one language - the *source* language - and translate it into an equivalent program in another language - the *target* language.

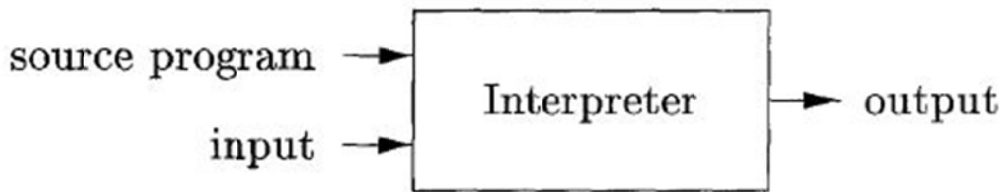
An important role of the compiler is to report any errors in the source program that it detects during the translation process.



If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs

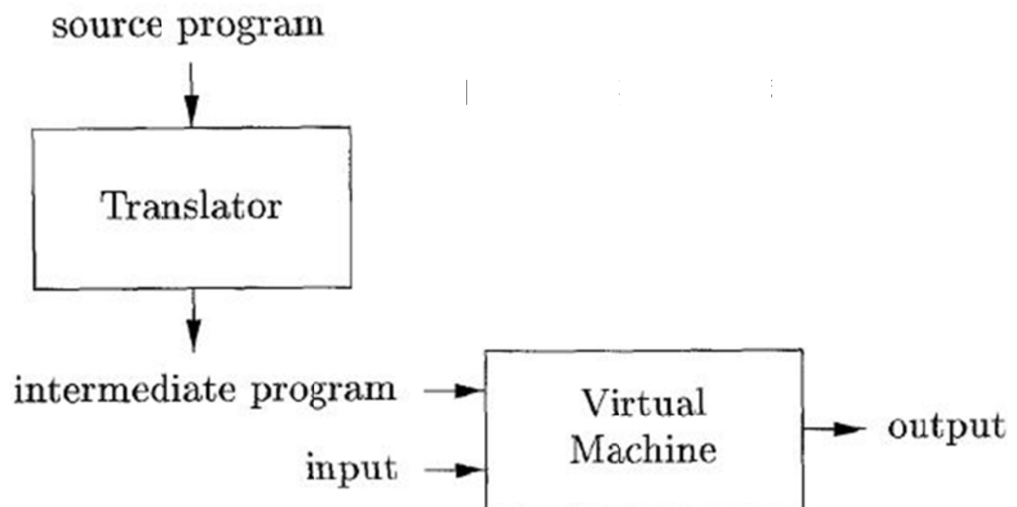


An *interpreter* is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user



The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs . An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

Some languages like Java the language processors combine compilation and interpretation, A Java source program may first be compiled into an intermediate form called ***bytecodes***. The ***bytecodes*** are then interpreted by a virtual machine. A benefit of this arrangement is that ***bytecodes*** compiled on one machine can be interpreted on another machine, perhaps across a network.

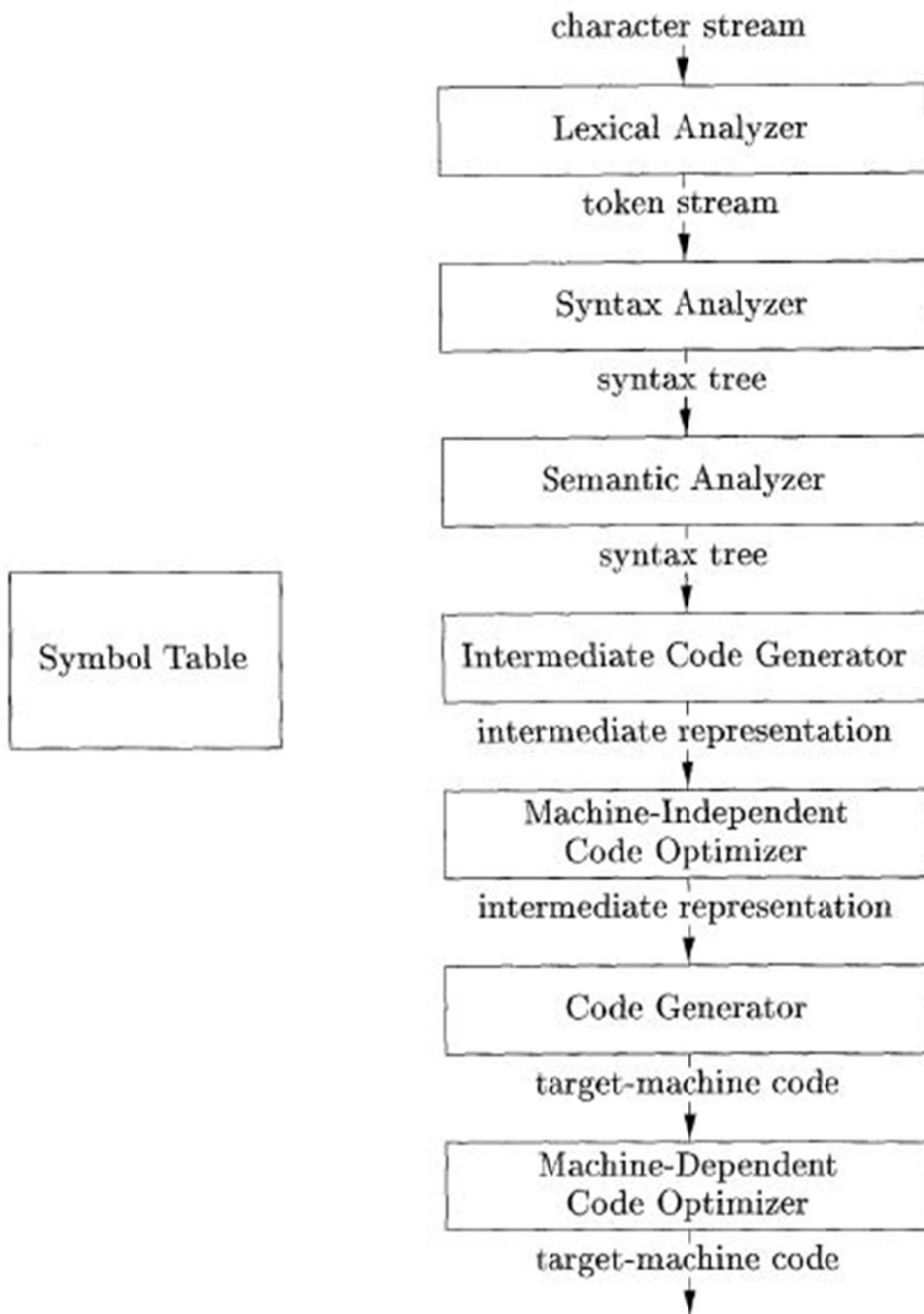


In order to achieve faster processing of inputs to outputs, some Java compilers, called *just-in-time* compilers, translate the ***bytecodes*** into machine language immediately before they run the intermediate program to process the input.

The Structure of a Compiler

The compiler has two parts : analysis and synthesis. The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the front end of the compiler; the synthesis part is the back end


character stream
Lexical Analyzer
token stream
Syntax Analyzer
syntax tree
Semantic Analyzer
syntax tree
Intermediate Code Generator
intermediate representation
Machine-Independent Code Optimizer
intermediate representation
Code Generator
target-machine code
Machine-Dependent Code Optimizer
target-machine code
Symbol Table

1- Lexical Analysis

The first phase of a compiler is called *lexical analysis* or *scanning*. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form

<token-name, attribute-value>

token-name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token. Information from the symbol-table entry 'is needed
for semantic analysis and code generation.
For example, suppose a source program contains the assignment statement

position = initial + rate * 60

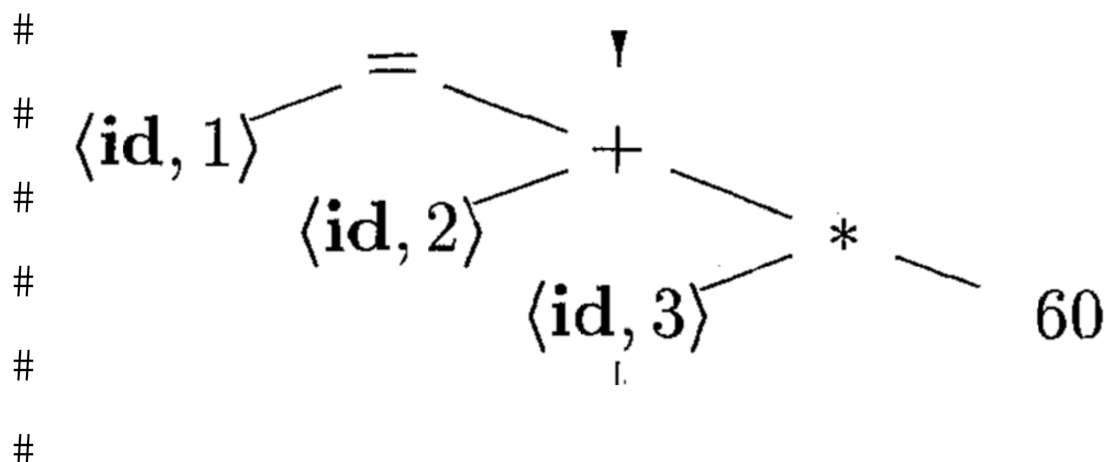
Blanks separating the lexemes would be discarded by the lexical analyzer.
The outputs of this phase for the above example is

<id,1> <=> <id,2> <+> <id,3> <*> <60>

2- Syntax Analysis

The second phase of the compiler is syntax analysis or parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create
a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.

–ª ˇ#‡°-ø°-#‡-#ª Ł‡#þª °¹ˇ#‡‡#ª łÖ ı#ı#ª ˇ#ł ºłÖ ŁıÆ#ŁÆ°ßˇ#



3- Semantic Analysis

The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.
The language specification may permit some type conversions called coercions.
For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

4- Intermediate Code Generation

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine.

This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.

we consider an intermediate form called three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register. The output of the intermediate
code generator for our example consists of the three-address code sequence

```
tl = inttofloat (60)
t2 = id3 * tl
t3 = id2 + t2
id1 = t 3
```

#

#

#

5- Code Optimization

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.

so the inttofloat operation can be eliminated by replacing the integer 60 by the floating-point number 60.0. Moreover, t3 is used only once to transmit its value to id1 so the optimizer can transform the above code into the shorter sequence :

t1 = id3 * 60.0
id1 = id2 + t1

6- Code Generation

The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the judicious assignment of registers to hold variables.

For example, using registers R1 and **R2,** the intermediate code in (1.4) might get translated into the machine code

```
LDF R2, id3
MULF R2 , R2 ,#60.0
LDF Rl , id2
ADDF Rl , Rl , R2
STF idl , Rl
```

#

The first operand of each instruction specifies a destination. The F in each instruction tells us that it deals with floating-point numbers. The code in loads the contents of address id3 into register R2, then multiplies it with floating-point constant 60.0. The # signifies that 60.0 is to be treated as an immediate constant. The third instruction moves id2 into register R1 and the fourth adds to it the value previously computed in register R2. Finally, the value in register **R1** is stored into the address of i d l , so the code correctly implements the assignment statement.

Symbol-Table Management

An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name. These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.

The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly

#

#

المترجم Compiler عبارة عن برنامج خاص يقوم بتحويل البرنامج المكتوب بلغة عليا (البرنامج المصدر) الى شفرة قابلة للتنفيذ (البرنامج الهدف) حيث يقوم بإعطاء تقارير بالأخطاء ان وجدت في البرنامج .

المفسر Interpreter هو نوع اخر من معالج اللغات الا انه لا ينتج شفرة هدف و انما يقوم بتطبيق العمليات الموجودة في تعليمات البرنامج المصدري على المدخلات مباشرة للحصول على المخرجات.

The Structure of Compiler هيكل المترجم

يتألف المترجم من مجموعة من الاطوار و هي :

1- طور تحليل المفردات Lexical Analyzer

و يدعى ايضا بالماسح Scanner وظيفته قراءة سيل الرموز المكونة للبرنامج المصدري و تجميعها بسلاسل ذات معنى تدعى المفردات lexemes و انتاج مخرجات تدعى الفقرات Tokens و التي تكون بالشكل التالي :

< Token name , Attribute Value >

Token name عبارة عن رمز مجرد يستخدم في طور تحليل القواعد اللغوية.

Attribute value يشير الى مدخل الفقرة في جدول الرموز و يستخدم اثناء طور تحليل المعاني و طور توليد الشفرة.

2- طور تحليل القواعد اللغوية Syntax Analyzer

و يدعى ايضا بالمعرب Parser حيث يقوم باستخدام الجزء الاول من الفقرات لإنشاء تمثيل وسطي بشكل شجرة و التي تصف التركيب القواعدي للفقرات . ان التمثيل النموذجي هو شجرة الاعراب Syntax tree حيث كل عقدة تمثل عملية operation و الاطفال تمثل معاملات operands هذه العملية .

3- طور تحليل المعاني Semantic Analyzer

هذا الطور يستخدم شجرة القواعد و معلومات جدول الرموز لفحص توافق المعاني لبرنامج المصدر مع تعاريف اللغة و يقوم ايضا بجمع المعلومات و خزنها في جدول الرموز او شجرة القواعد نفسها .

اهم ما في هذا الطور هي عملية فحص الانواع Type Checking حيث يقوم بفحص توافق العمليات operator مع معاملاتها operands فمثلا دليل المصفوفة يجب ان يكون من النوع الصحيح اما اذا كان من النوع الحقيقي يجب اصدار رسالة خطأ.

بعض اللغات تقوم بتحويل الانواع بصورة اوتوماتيكية و تدعى هذه العملية بالإجبار او الاكراه Coercion .

4- طور توليد الشفرة الوسطية Intermediate Code Generation

في هذا الطور يتم توليد شفرة ثلاثية العناوين Three Address Code كما سيتم توضيحها لاحقا حيث تتألف من تعاقب من التعليمات الشبيهة بتعليمات لغة التجميع حيث كل تعليمة لها ثلاث معاملات كل معامل يمكن ان يمثل بسجل .

مواصفات شفرة ثلاثية العناوين

- كل تعليمة اسناد لها عملية واحدة على الاكثر في جهة اليمين .
- المترجم يجب ان يولد متغيرات مؤقتة لخزن القيم المحسوبة .
- بعض التعليمات تحتوي على معاملات اقل من ثلاث كما في التعليمتين الاولى و الاخيرة .

5- امثلية الشفرة الوسطية Code Optimizing

في هذا الطور يتم تحسين الشفرة الوسطية للحصول على شفرة هدف افضل (تقليل عدد العمليات و المتغيرات للسرعة).

6- طور توليد الشفرة Code Generation

في هذا الطور يتم توليد الشفرة الهدف Target Code حيث يتم استخدام السجلات و مواقع الذاكرة لخزن المتغيرات الموجودة في البرنامج لان لغة الهدف هي لغة الالة .

جدول الرموز Symbol Table

عبارة عن هيكل بياني يحتوي على قيود لكل المتغيرات و حقول تمثل مواصفات ذلك المتغير , حيث يتم تصميمه لإيجاد القيد الخاص بمتغير معين و خزن و استرجاع البيانات من القيد بسرعة.