# Chapter 1

# INTRODUCTION

[AU] 1.

## 1.1 Elements of the definition of a language

In describing a language three aspects are traditionally distinguished:

- The study of possible relationships between the symbols of the language (words), and therefore the structure of precisely those sequences of words which do make up a sentence, is termed *syntax*.
- The study of relationships between sentences and their meaning is termed *semantics*.
- Finally, by *pragmatics* is meant the study of the relationships between sentences and the people using them.

Applying this terminology to programming languages is not just a fancyful borrowing: the programs of a programming language can be seen as the sentences belonging to that language, each with a meaning attached to it. The salient difference between natural and programming languages is that programming languages do not just arise, but are designed for their specific purpose.

The **syntax** of a programming language is a description, in some convenient notation, of exactly which sequences of symbols are programs in the language. It allows to decide whether a given text is a program or not. Programming languages will tend to have a simpler syntax than natural languages. Furthermore they are not spoken but exist primarily in a written from.

The **semantics** of a programming language defines the effect which a given program has, e.g. in terms of the actions of a (concrete or hypothetical) computer.

The **pragmatics** of a programming language, the relationship between programs and the people who use them, is a field that has never been consistently studied. In a narrow sense it has to do with e.g. the user-friendliness of programming languages, in a wider sense with the use that is made of the computer as a tool and as a partner in modern society.

## 1.2 Some terminology

In this course we approach programming languages from a different aspect than is usual: we are now concerned with their efficient implementation on concrete machines.

- **programming language**: a language intended for writing executable algorithms.

- **machine-oriented language**: a programming language whose concrete algorithms and objects have a structure similar to the architecture of some class of digital computers.

- **machine language**: a machine-oriented language which closely follows the architecture of a given digital computer.

- **problem-oriented language**: a programming language which serves to write programs of a certain application area, independent of any specific digital computer.

- **machine-sensitive language**: a programming language which allows adaptation and access to the specific architecture of any particular machine but is itself not necessarily machine-oriented.

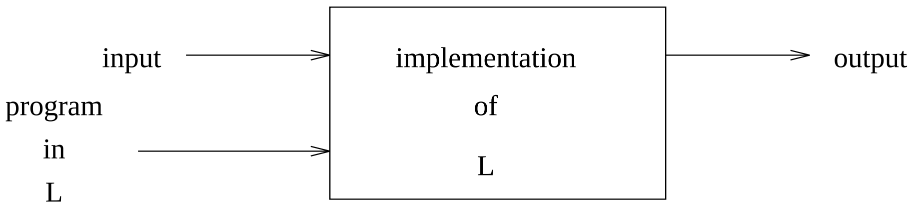
A **program** is a constructive realization of some function $F : A \to$ B written in a programming language O.

A **compiler** is a program whose input- and output languages S and T are programming languages and which translates programs, written in S, into equivalent programs in T.

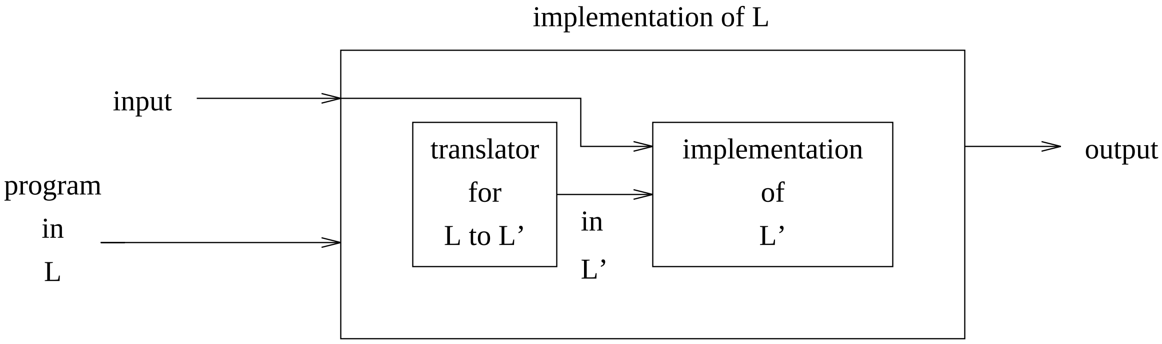
An **implementation** of a programming language O is an (abstract or concrete) machine that can execute programs written in O.

### 1.2.1 Compilers and interpreters

We can consider the *implementation* of a language L as a black box, into which we can put a program written in L, and which then produces an effect or a result.




Looking into the black box, we can distinguish between a compiler and a interpreter: the *compiler* translates the program written in L to a program written in L', which is the input for the interpreter for L', whereas an *interpreter* interprets the program without such a translation step.




A "pure" interpreter doesn't pretranslate and is slow in execution. A "pure" compiler on the other hand pretranslates as much as possible and is therefore fast in execution. In practice any point in between a "pure" interpreter and a "pure" compiler can be taken.

### 1.2.2 Passes and phases

WG 1.2

In a **pass** the compiler reads over the text of the **source program** , or over the output of the previous pass, and produces a transformed image, which may be read by a following pass. The word *pass* finds its origin in the time that computer memories were small and programs were on papertape. The compiler would read the papertape and produce another tape, the input to the next pass.

Now we have greater random access memories and when a program is compiled its image can be kept in memory. Therefore the word pass is an imprecise notion. Passes can be understood to be any component of a translator whose execution complexity is "linear" in program size.

A **phase** is one coherent function or activity, depending on other phases and preparing for other phases. Phases are combined into passes.

In an **one-pass compiler** all phases take place as a part of, and simultaneously with, one pass over the source program.

In a **multipass compiler** the compilation takes place in a number of passes, which communicate by common memory, global tables, physical or virtual IO.

A one pass compiler is definitely easier to comprehend and construct for one person. On the other hand, for a one-pass computer the language must be suited or be modified (e.g. no dependencies on right context allowed).

The attitude "all our experience is in one-pass compilers" has led to Lalgol68 1-pass subsets (La68r, La68s) and to the (logically superfluous) forward declarations in Lpascal.

A multi-pass compiler has to convert many intermediate codes or languages, which causes a certain amount of overhead. In particular each pass has to write its produced code in an intermediate file, which has to be read by the next pass. The time needed for the IO of intermediate files may dominate the processing time. Furthermore the sequential character IO may influence the compiler design.

On the other hand small passes are easy to get correct. The original Los Lpl/1 compiler, written at a time when compiler knowledge within Libm was rudimentary, counted 192 passes (involving many magtape units simultaneously). For some programming languages such as Lalgol68, without special restrictions or artifacts like forward declarations a multi-pass compiler is unavoidable.

## 1.3 The structure of a compiler

WG 1.4

A compiler can at a very rough level of granularity be split into two phases, the **analysis phase a**nd the **synthesis phase**, which communicate by various tables as well as by a series of transformed images of the source program. At a somewhat more detailed level we can distinguish several phases, as shown in picture 1.1.

### 1.3.1 The analysis phase

The first phases are dedicated to the analysis of the structure of the source program, the collection and check of its static semantics and the construction of the parse tree. They are:

1. **lexical analysis**.
   This phase transduces the source program into a less redundant text and may collect

further information such as linenumbers for later steps of the compiler. The recognition of the basic building stones (tokens), such as IF, THEN, BEGIN etc., and the resolving of the context dependent meaning of characters occurs also in this phase.

2. **syntactical analysis**.
   The purpose of this phase is the recognition of the structure of the source text, the construction of a parse tree and the collection of type information from defining occurrences.

3. **context sensitive analysis**
   In this phase, the parse tree is decorated with types and other attributes and the relations between defining and applied occurrences are established (**identification**), and errors in the identification are reported.

The transformed image of the source program resulting from this last phase is termed the **attributed parse tree**.

### 1.3.2 The synthesis phase

In the synthesis phase(s) of the compiler a program for some concrete target machine is produced (the **target program**). They comprize one or more of the following:

1. **abstract code generation**.
   Transduction of attributed parse tree to a program for an abstract machine.

2. **global optimization**.
   Optimizations of the abstract program based on its overall structure.

3. **local optimization**. Optimizations of parts of the abstract program based on their local structure.

4. **concrete code generation**.
   Generation of code for a concrete target machine.

5. **post optimization**.
   optimizations on the concrete code, based on the architecture of the target machine.

These phases can be composed into passes in many ways; some of them may be omitted or combined with another phase, leading to a spectrum from one-pass to many pass compilers.

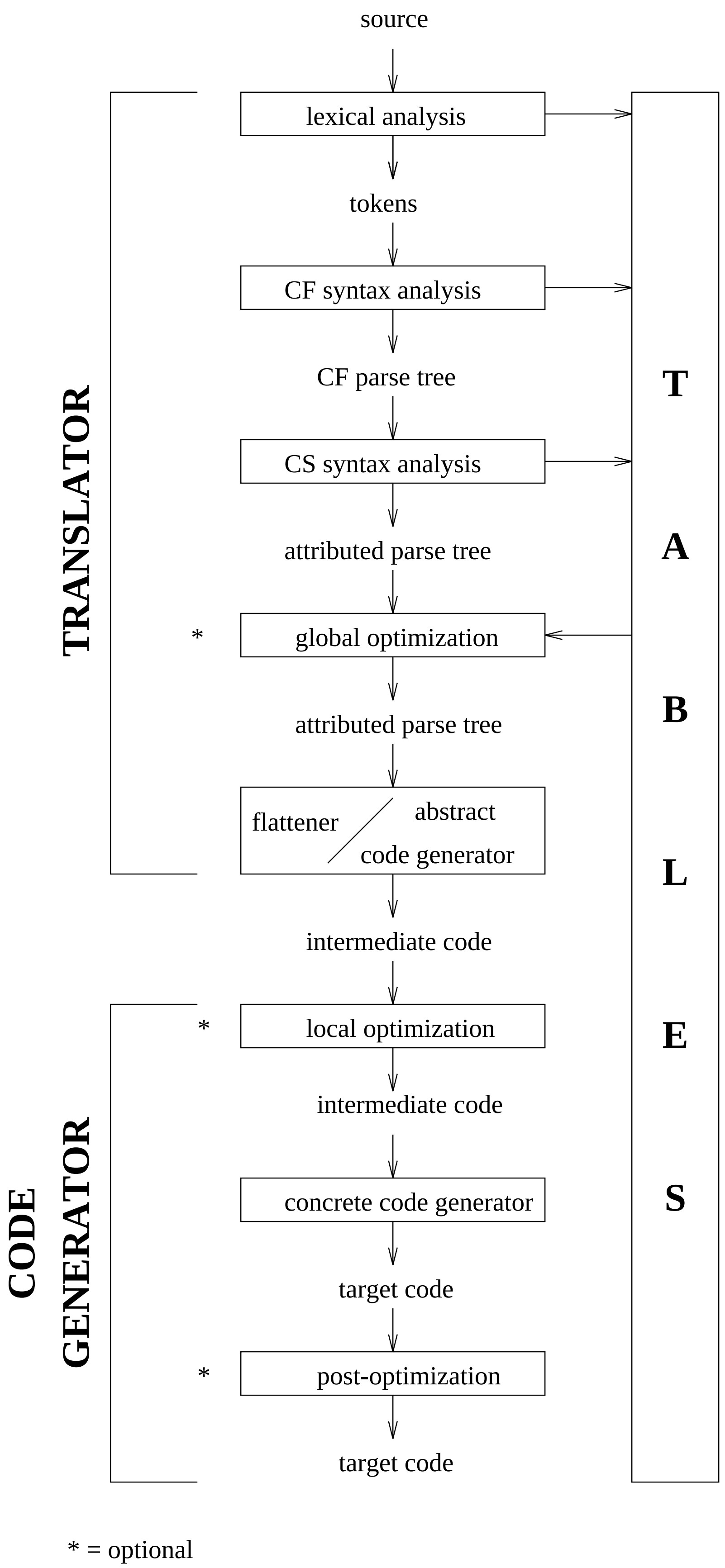



Figure 1.1: Structure of a compiler

## 1.4 T-diagrams

The reason we have computers is the fact that we want to solve problems. We solve a problem by programming the computer to have a specific input/output behaviour.

More abstractly, a **program** is a constructive realization of some function

$$F : A \longrightarrow B$$

which maps any argument in a given domain $A$ (the "input") to an element of another domain $B$ (the "output"; we disregard in this section the issue of non-determinism). A program in $\mathcal{L}$ realizing $F$ differs from the mathematical function $F$ by the fact that, given an implementation of $\mathcal{L}$ on some computer, we can physically provide some input $x$ from $A$, run the program for some time (hopefully not too long) and obtain some output $y$ from $B$ such that $y = F(x)$.

By an **implementation** of $\mathcal{L}$ we mean a machine (abstract or concrete) such that it executes $\mathcal{L}$-programs correctly, i.e., given $F$ and $x$ in $\mathcal{L}$ it does compute $y$.

We will at this point introduce a notational system [**?**] suitable for the discussion of programs, as distinct from the discussion of functions. We use a **T-Diagram** (see figure 1.2) to denote a program, written in $\mathcal{L}$, that realises a mapping $F$ from $A$ to $B$.

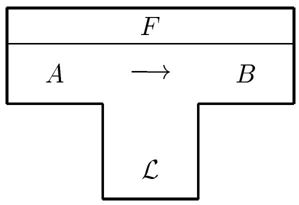



Figure 1.2: T-diagram of a program

We call

$A$ = the input domain
$B$ = the output domain
$\mathcal{L}$ = the object language

A program whose object language is $\mathcal{L}$ we will call an $\mathcal{L}$-program (e.g. a Lfortran program).

An implementation of a language $\mathcal{L}$ can, in its simplest form, be depicted as a black box, let us say an $\mathcal{L}$-machine: $\boxed{\mathcal{L}}$. If we put the previous $\mathcal{L}$-program on top of the $\mathcal{L}$-machine, we obtain an (abstract) $F$-machine (figure 1.3).

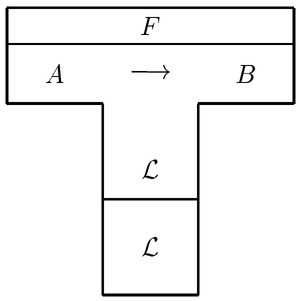



Figure 1.3: An abstract $F$-machine

When we feed it some input $x$ in $A$ we obtain some output in $B$, viz. $F(x)$.

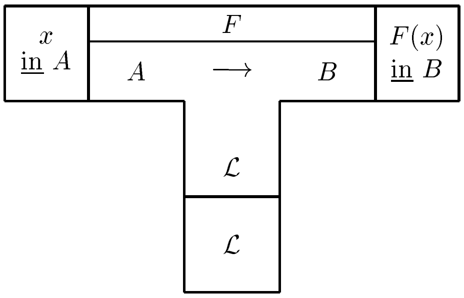



Figure 1.4: Application on the abstract $F$-machine

### 1.4.1 Implementations

Of course, upon closer inspection an implementation of $\mathcal{L}$ will turn out not to be a concrete $\mathcal{L}$-machine, but to consist of a concrete $M$-machine for some language $M$, with an $\mathcal{L}$-interpreter on top of it (figure 1.5).

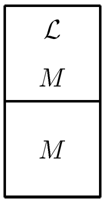
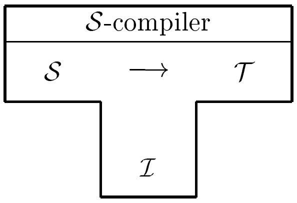



Figure 1.5: An interpreter on top of a machine

Or again, it may contain a **compiler**. A compiler (figure 1.6) is a program with the




$\mathcal{S}$ = source language
$\mathcal{T}$ = target language
$\mathcal{I}$ = implementation language

Figure 1.6: T-diagram for a compiler

special properties that

- its input domain is some programming language $\mathcal{S}$
- its output domain is another programming language $\mathcal{T}$
- it translates $\mathcal{S}$-programs into *equivalent* $\mathcal{T}$-programs.

The object language of a compiler is often called its implementation language.

### 1.4.2 Further terminology

A compiler whose source language is $\mathcal{S}$ is called an *$\mathcal{S}$-compiler* (e.g., a Lcobol-compiler)

A compiler whose target language is $\mathcal{T}$ is called a *compiler towards* $\mathcal{T}$ (e.g., a compiler towards 360 Assembler). By transference, if the language $\mathcal{T}$ is amongst those available on some machine $M$, then a compiler towards $\mathcal{T}$ may also be called a compiler towards $M$ (through $\mathcal{T}$) (e.g., a compiler towards 360 (through Assembler)).

A compiler whose implementation language is available on some machine $M$ is called a *compiler on* $M$ (e.g., a compiler on Ŀcdc Cyber).

Finally, a compiler whose implementation and target languages are each available on different computers may be called a **cross-compiler**.

As an example, the T-diagram in figure 1.7 depicts a Ŀpascal cross-compiler on Ŀvax towards Ŀpdp11 assembler under ĿUnix.

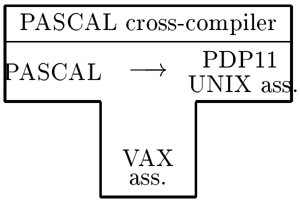



Figure 1.7: A cross-compiler

Using this compiler it is possible to execute Ŀpascal programs, provided the necessary machines are available (we abstract here from the by no means trivial problems in communicating between the different computers involved).

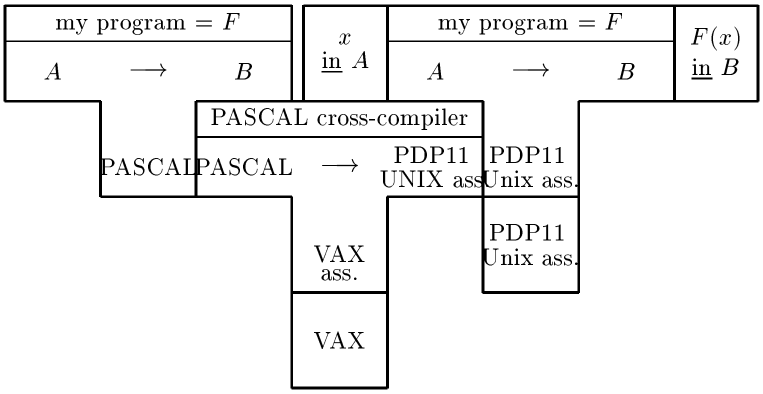



Figure 1.8: Using a cross-compiler

The T-diagram notation by itself does not actually enlarge our understanding of programs, but it serves to represent the obvious in a picturesque and helpful fashion.

## 1.5 Bootstrapping

The use of a formalized notation for translation processes like the T-diagram becomes essential if we want to discuss *bootstrapping*. In informatics this word is bandied about in various places, but we will mean specifically: the translation of compilers.

Let us put a Ŀpascal program, which has the function of translating Ŀbasic-programs into Ŀunivac assembler (a Ŀbasic compiler in Ŀpascal), through a Ŀpascal compiler on Ŀunivac (figure 1.9)

Since the Ŀpascal compiler preserves the semantics of the programs it translates, the result is another Ŀbasic compiler, but this time in the target language of the Ŀpascal compiler, viz. Ŀunivac assembler, and therefore executable on a Ŀunivac machine.

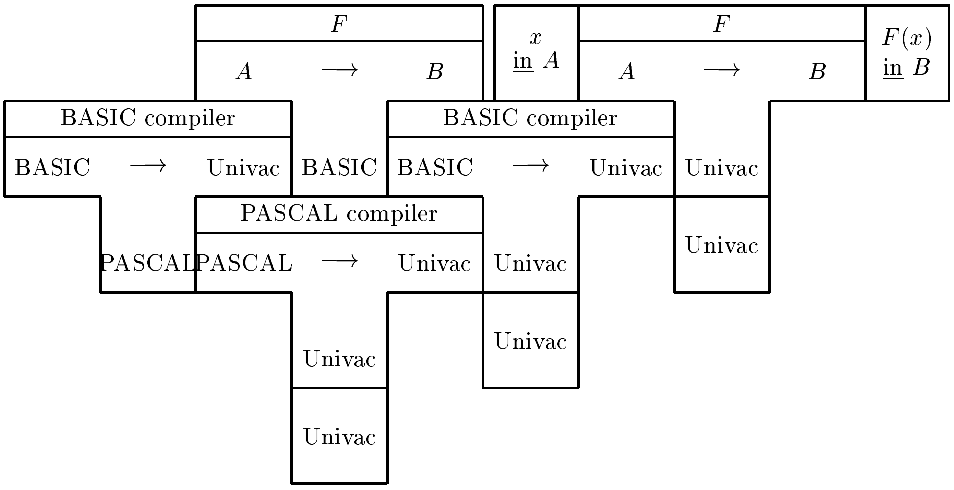



Figure 1.9: A bootstrap

We have compiled a compiler by means of the Lpascal compiler. The original Lbasic compiler, written in Lpascal, may well have been a beautiful piece of craftsmanship, but it was not directly executable, due to the absence of a concrete Lpascal machine. By contrast, the result of putting it through the Lpascal compiler is an ordinary compiler, executable on the Lunivac machine.

There is some wonderful magic about getting an executable compiler from a non-executable one, which has led people to the name **bootstrapping**: pulling yourself up by the straps of your own boots. Somehow a whole compiler was pulled out of a hat.

The situation becomes even more interesting if we put the same Lbasic-compiler in Lpascal through a Lpascal compiler on another machine, say the Lvax.

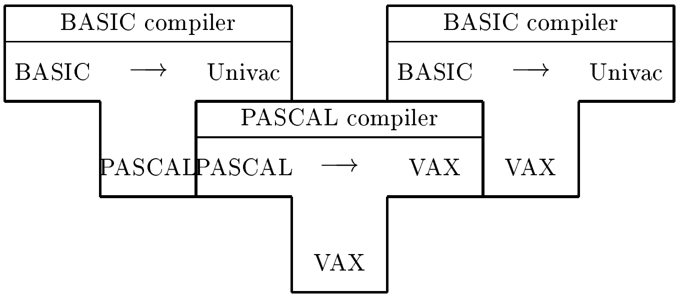



Figure 1.10: Generating a cross-compiler

The result is a Lbasic-compiler running on the Lvax but producing code for a Lunivac machine: a Lbasic-cross-compiler on Lvax to Lunivac. If we wish instead to get Ldec20 assembler as a target language, we will have to modify either the Lpascal or the Lvax version of the Lbasic compiler. Writing a compiler in a high-level language by itself is not enough to make it portable.

A compiler, in order to be truly portable, has to be a **retargetable compiler**, i.e. it must be so designed that it can easily be adapted to different target languages, e.g. by a process of parametrization or by the adaptation of a rather small part of it, the concrete code-generator. The Lcdl2 compiler is an example of a retargetable compiler.

As an example, let us discuss one possible technique for implementing some high-level language Llan. Being high-level, Llan should contain some small subset Llan* suitable for writing Llan compilers. Therefore we will write a retargetable Llan compiler in this

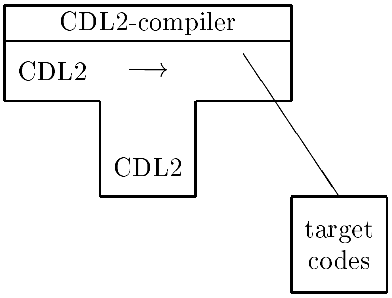



Figure 1.11: A retargetable compiler

subset Llan*. We assume some low-level target language Lasm to be available on our target machine.

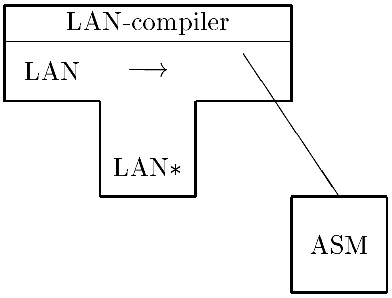



Figure 1.12: A retargetable LAN compiler

We will describe two techniques to port this Llan compiler to some specific machine, both based on retargeting.

### 1.5.1 Pulling bootstrap

Assume we have produced a retargetable Llan compiler in Llan*. As a next step, we make in Lasm, the assembly language of the machine to which the compiler is to be ported, a quick-and-dirty implementation of Llan* on the target machine, which is relatively easy for a number of reasons:

- It can assume that it will never be confronted with incorrect programs. In fact it may even assume it will never be confronted with any other program than the Llan-compiler.
- It can restrict itself to Llan* and need not implement any feature of Llan which is not absolutely essential. Since we have the extent of Llan* completely in our hands, we can stick to things which are easy to implement.
- It will have to run successfully only once, as we will see. Therefore it need not at all be efficient.

This gives a quick-and-dirty (interpreted) Llan compiler, which will probably be slow and inefficient.

We then compile the compiler with itself (figure 1.14).

The result is a version of the Llan compiler, running on our machine which (if we have done our work well) will be much more efficient than the initial quick-and-dirty version.

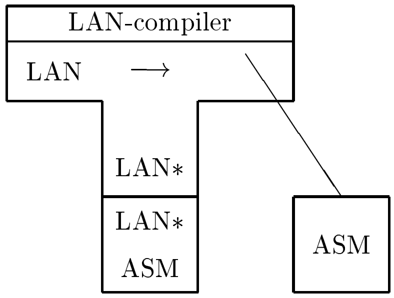



Figure 1.13: Interpreting the LAN-compiler

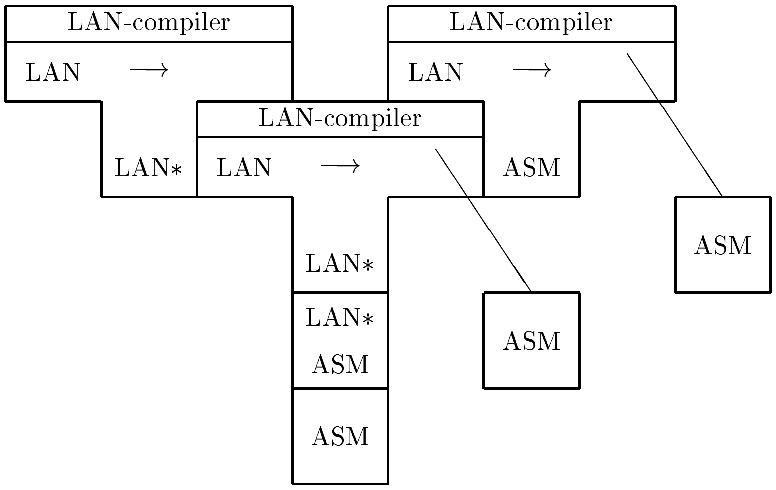



Figure 1.14: A pulling bootstrap

Through bootstrapping we have obtained an efficient compiler. Notice that in this **pulling bootstrap** all the work is done on the target machine.

In academic circles, pulling bootstrap is the preferred technique, because all the work (and all the responsibility) can then be left to the recipient of the software, working on the machine best known to him.

### 1.5.2 Pushing bootstrap

An alternative technique is to first make a Llan to Lasm compiler on the object machine $X$. We do this by suitable parametrization of a retargetable compiler on $X$.

Next, this resulting compiler is carried bodily to the target machine in the form of an Lasm program, which is then run as a retargetable compiler on the target machine, with as target Lasm.

In this **pushing bootstrap** technique, most of the work is done on the object machine. Pushing bootstrap may be advantageous over the more popular pulling bootstrap in case the target computer is small and uncomfortable or difficult to operate (like most computers).

For software producers, producing various versions from one same portable compiler, pushing bootstrap is advantageous for security reasons (the source text of the original compiler is never stored in the target computer) and for maintenance (all versions can be maintained on one same object machine).

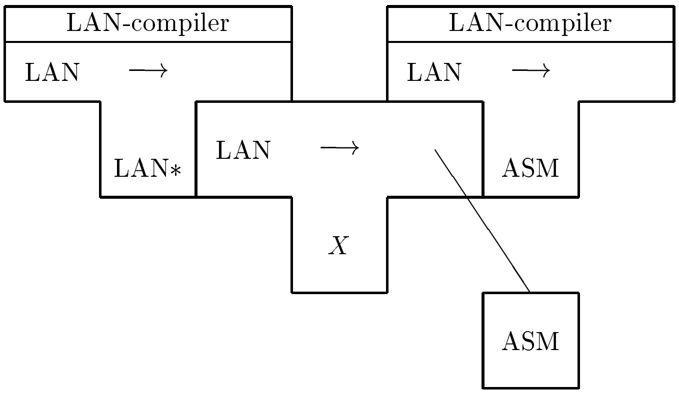



Figure 1.15: A pushing bootstrap

# Chapter 2

# LEXICAL ANALYSIS

WG 4.1.1, 4.2.1, 5.2, 6
ASU 2.7, 7.6, 3.6, 3.7

## 2.1 Introduction

A program in a certain programming language can be considered on three levels of abstraction:

1. **constructs**, i.e. terminal productions of nonterminals of the grammar of the language
2. **symbols**, the terminal symbols of that grammar, and
3. **characters** from e.g. the ASCII characterset, by means of which those symbols are represented.

In natural language description, a similar distinction is made between *phrases*, *lexemes* and letters or *phonemes*.

The distinction into three levels allows us to deal with multiple representations for one symbol, and allows the application of simpler and more efficient recognition techniques for the symbols, which usually have a regular structure.

In this chapter we consider the **lexical analysis** phase of compilers. During this phase the symbols are recognised from the input character stream, the classes to which the symbols belong are determined, and if needed the symbols are stored in a symbol table. The source program is transformed to a sequence of symbols.

The lexical analysis may be a subroutine or coroutine of the syntax analysis, or it may be a separate pass. In either case, factoring the recognition of symbols out of the parsing of constructs leads to a separation of concerns which simplifies design.

### 2.1.1 Symbol classes

By an **closed class** of symbols we mean a fixed (and small) set of symbols, defined by enumeration. By an **open class** we a set of symbols which is either large or open ended, defined by some construction principle.

Symbols in a programming language typically belong to:

- open classes of:
  - identifiers, e.g. `mar1lyn`
  - bold names, e.g. `INCR`
  - denoters e.g. `3.14`

- closed classes of:
  - delimiters, e.g. `BEGIN` or `(`
  - special names e.g. `+`

Furthermore there are (irrelevant) comments and layout, which the lexical analyser should ignore.

## 2.2 Lexical analysis by finite state automata

Since the symbols of a programming language usually possess a right-regular grammar, their recognition can be done by a **Finite State automaton**.

Some examples:

```
IDENT: letter , LETGITS.

LETGITS: letter , LETGITS;
         digit , LETGITS; .
```

For the purpose of this discussion we assume `letter` and `digit` to be defined as characters (or rather, classes of characters).

A Finite State Automaton (**FSA**) recognising the language described by `IDENT` is depicted in figure 2.1.

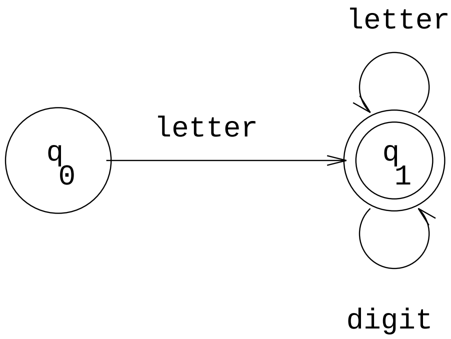



Figure 2.1: FSA for `IDENT`

Similarly, we may define other classes of symbols:

```
NUMBER: digit, NUMBERTAIL.
NUMBERTAIL: NUMBER; .

STRING: "'", STRINGTAIL.
STRINGTAIL: "'"; any, STRINGTAIL.
```

were `any` is defined to be the set of all characters except '.

### 2.2.1 Realization of FSA

It is extremely simple to realize an FSA in any programming language which includes jumps and labels, by making the correspondence

| | |
|---|---|
| state | label |
| transition | conditional jump |

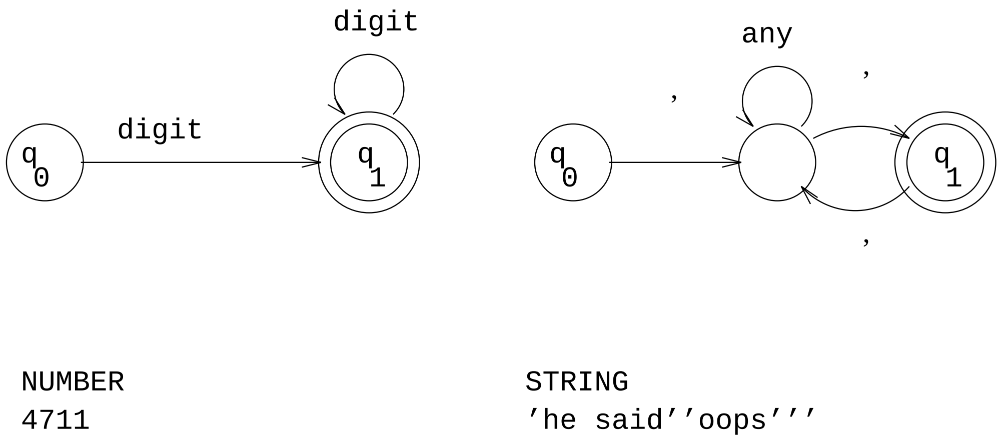



Figure 2.2: FSA for `NUMBER` and `STRING`

```
ident:
   IF ahead (letter) THEN next char; GOTO letgits ELSE error FI;

letgits:
   IF ahead (letter) THEN next char; GOTO letgits FI;
   IF ahead (digit) THEN next char; GOTO letgits ELSE error FI;
```

Obviously this realization can be somewhat improved upon, but on conventional machines even this crude realization should be quite efficient.

### 2.2.2 Composing FSA's

As a next step, we will combine the automata of all lexical units into one automaton. We obtain the composed automaton in figure 2.3.

Sometimes combining two automata introduces **nondeterminism**, in the sense some of the states in the composed FSA have outgoing arrows marked with the same symbol or symbol class. As an example, consider the denotations for binary numbers (bitstrings) and character strings in Lpl/1. Both are defined by a regular expression

$$bitstring = \ ' \ (0|1)^* \ ' \ B$$

$$charstring = \ ' \ any^* \ ' \ \{ \ ' \ any^* \ '\}^*$$

Combining the two DFA's, a nondeterministic DFA is obtained (fig. 2.5).

Fortunately this is no problem, because every non-deterministic Finite State automaton can be transformed into a deterministic one which recognises the same language, by combining arcs with the same mark into transitions to a new state, representing the unresolved ambiguity, as in fig. 2.6.

The general construction for transforming a nondeterministic finite state automaton into an equivalent deterministic one is given in ASU 3.6 and WG 5.2.

### 2.2.3 Recognizing the end

4711 is a number, but so are 471 and 711. How do we determine what are the boundaries of a symbol? An obvious approach is to start with the next character of the input and accept all subsequent characters that may part of the symbol, thus obtaining the longest symbol that can start at the current position of the input.

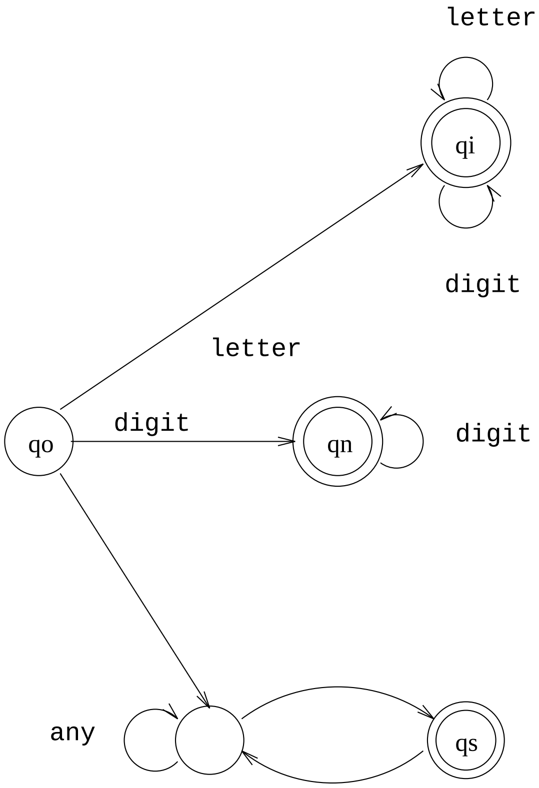



Figure 2.3: Composed FSA

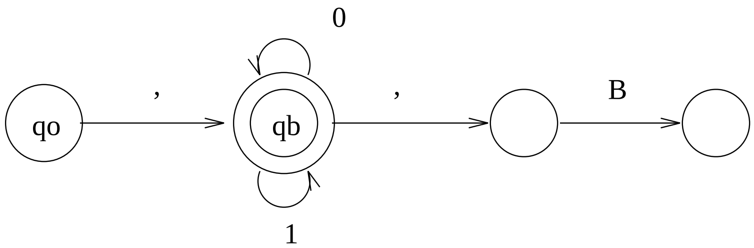



Figure 2.4: bitstrings in PL/1

If the automaton now is in a final state (in the example before the above, qi, qn or qs) then a correct symbol (an IDENT, NUMBER, or STRING) has been recognised. If, on the other hand, the automaton is not in a final state, then a lexical error is detected.

After recognising a correct symbol, the automaton returns to the initial state qo and recognition of the next symbol may commence.

Trying to find the longest symbols does however not always work.
fortran:

- ```
  IF(4.E3+6.E5)1001,1002
       ↕
  IF(4.EQ.A)THENGOTO99
  ```

- `10FORMAT(X5H)=G(I)`

  either assignment statement

  ```
  10FORMAT(X5H)=G(I)
  ```
  ```

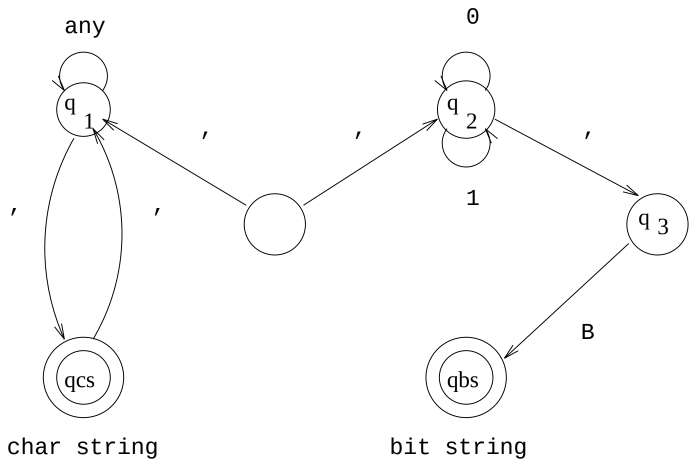



Figure 2.5: Nondeterministic FSA for PL/1 strings

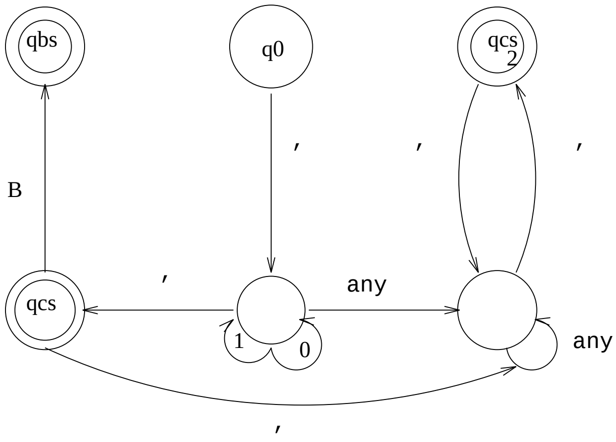



Figure 2.6: Equivalent deterministic FSA

or format statement (including a Hollerith constant of 5 characters)

```
10FORMAT(X5H)=G(I)
```

In decent languages, any program can be uniquely split into symbols, according to the principle of extending a symbol with the next character as long as this is possible.

A counterexample from a modern language:

```
x:=a.          x:=a.b;          x:=3.0E4
b:...
```

### 2.2.4 Error treatment

We now discuss what may happen if a **lexical error** is detected. Consider an automaton to recognise PL1 bit strings.

E.g. '001'B is a legal bit string, but '00C'B isn't. There are roughly two solutions for dealing with such incorrect input.

1. upon detection of the first incorrect character the automaton reports an error and then returns to the initial state. It will ignore the incorrect input and tries to recognise the next symbol. In the above example C will then be recognised as an identifier.

2. the incorrect character is ignored but the automaton stays in the same state. Extraneous characters will thus be skipped. The above example is then recognised as '00'B.

Neither of the two solutions is obviously the "best".

### 2.2.5 Reserved-identifier approach

In some languages, certain identifiers are used as delimiters, or rather some delimiters are written in the same way as the identifiers. Use of such **reserved identifiers** as names for objects may be forbidden, or allowed only if no confusion arises, the interpretation as reserved identifier being preferred. This leads to endless confusion, perfectly innocent looking identifiers being unavailable due of some technicality, and to genuine ambiguities.

Examples:

| | | |
|---|---|---|
| PL1 | `DECLARE DECLARE BIN FIXED (32);` | context necessary |
| PL1 | `IF (EXPR) THEN RETURN;`<br>`IF (EXPR) = 9;` | lexical type of IF can only be<br>determined by (long) lookahead. |
| FORTRAN | `DO9I=1,9`<br>`DO9I=1.9` | <br>may be 1 or 3 lexical units! |

To see to what extent the presence of reserved identifiers leads to problems in a given language, consider a Finite State automaton recognising reserved indentifiers and also non-reserved identifiers.

It is good practice in programming languages to use the distinction between lowercase and uppercase characters to minimise the reserved identifier problem.

## 2.3 Output from the lexical analysis

As mentioned before, the final state the lexical analyser is in when no more transition is defined determines the class that the recognised symbol belongs to. The symbol consists of the part of the input from the position where the automaton was last reset to the initial state upto the current input position.

For delimiters and special names the symbol may be encoded e.g. by a unique number.

For open classes like identifiers, the automaton has to construct, as a side effect, an entry in the symbol table, provided of course that an entry for the symbol is not already present. The symbol may now be represented by a pointer to the proper entry in the symbol table.

### 2.3.1 Symbol table algorithms

The **symbol table** ("namelist", "representation table") is that datastructure of the compiler that contains the representation of symbols and that associates properties with the symbols in the table.

A typical interface for a symbol table may be

- `ACTION enter (>symbol,key>):`
  Given a string S, return the key K which uniquely identifies that string.

- `ACTION put property (>key, >prop):`
  `FUNCTION get property (>key, prop>):`
  There is a changeable mapping:

$$prop : key \rightarrow property$$

- `FUNCTION repr of (>key, symbol>):`
  for e.g. printing the symbol with which the key is associated.

$$repr : key \rightarrow symbol$$

Some requirements:

- Preferably no limit on symbol length, or a liberal one e.g. line length.
- Must hold many different symbols (e.g. 1000-5000). As a rule of thumb, the number of different symbols is proportional to the square rest of the program length.
- The total amount of space is of course bounded.
- Enter/retrieve must be efficient. Delete not necessary.
- Average 10 symbols/line, each symbol on the average applied 4 times.
- Alfabetic listing useful.
- Algorithm must be robust against "pre-sorted" input.

When we have an efficient symbol table for open classes, we may just as well use it to store the representations for members of a closed class, e.g. the delimiters (cf. reserved identifier approach).

### 2.3.2 Choice of data structure

Dick Grune [ALEPH] describes an experiment with different symbol table algorithms, which he presented with a random permutation of 4000 identifiers, 1000 of which are different, corresponding to every identifier being applied 3 times. After entering the identifiers, an alphabetic listing was to be produced.

The results of the experiment are summarized in the following table. The numbered columns mean:

1. number of comparisons (x 1000)
2. degeneration factor: how much worse for ordered input?
3. memory use, number of cells for pointers + 1 per symbol
4. specific overheads for this algorithm.

| name | 1 | 2 | 3 | 4 | |
|---|---|---|---|---|---|
| linear search, unordered table | 2000 | 1 | 1000 | + | 1000 for sort |
| linear search, ordered table | 1800 | 1 | 1000 | + | 250000 moves! |
| binary search, ordered table | 7 | 1 | 1000 | + | 250000 moves! |
| binary search, diluted table | 37 | 1 | 1100 | | |
| ordered linked list | 1800 | 1 | 2000 | | |
| | | | | | |
| binary tree search | 47 | 40 | 3000 | | |
| randomized binary tree | 51 | 1 | 3000 | | |
| | | | | | |
| broad-leaf tree | 47 | | 1500 | | |
| closed hash (T=256), list | 26 | 1 | 2300 | + | 1000 for sort |
| closed hash (T=1024), list | 21 | 1 | 3000 | + | 1000 for sort |
| | | | | | |
| Thumb indices, one lett., list | 150 | 1 | 2000 | | |
| Thumb indices, two lett., list | 7 | 1 | 2700 | | |
| Thumb indices, one lett., tree | 30 | 6 | 3000 | | |
| Thumb indices, two lett., tree | 19 | 3 | 3700 | | |
| Thumb indices, one lett., random | 34 | 1 | 3000 | | |
| Thumb indices, two lett., random | 23 | | 3700 | | |
| Open extending hashing | 22 | 1 | 3000 | | |

Although they much more straightforward in their realization, the tree-based algorithms do not appear much worse than hashing. There is a clear **time/space tradeoff** which may help in choosing a specific datastructure for a particular application of a symbol table.

## 2.4 Exercises

1. Design an algorithm to make a given FSA deterministic.
2. Which representation would you choose for a symbol table, in case you expect at most 1000, 100000 or 1000000 different symbols, respectively?