# 2 Analysis and the symbol table

The first phases of a compiler are concerned with analysis of a source program. The intermediate products of the analysis become increasingly abstract. They are either transient, forming the intermediate products between phases, or persistent, becoming increasingly refined and complete, as each phase contributes more information.

The temporary products have various forms, the most widely known being the parse tree. They move towards more concise and machine processable forms as analysis progresses.

The resident product is generally known as a symbol table. The amount of information to be held depends on the nature of the later synthesising stages. In particular, the requirements of optimisation and diagnostics increase the amount of such information.

### Lexical vs syntax analysis

Much of the work of a lexical analyser may be left to the syntactic or even the semantic analysis phases. The choice of an appropriate division of work between the lexical and other phases is a pragmatic one and depends on considerations of efficiency.

There are two ways of approaching lexical analysis in common use, the ad hoc and the regular expression based. Both allow optimisation of time consuming parts of program analysis.

As an example, it is generally possible to extend the syntactic analysis phase in a syntax driven compiler to cope with identifiers. The syntax of identifiers is usually expressable as a context free grammar, suitable for top down or bottom up analysis. In a recursive descent compiler this would require a procedure call for each character in the identifier.

Typically

$$identifier \rightarrow letter \ \ letter\text{-}or\text{-}digit*$$

which would lead to procedures `letter`, `digit`, `letterordigit` and `identifier`. The procedure `letterordigit` could either call itself recursively or be called from a while loop (tail recursion).

It is usually possible to implement the scanning of a string more efficiently. This is, therefore, usually left to the lexical analyser. (Another name for this phase is the *scanner*).

In general strings are very inefficient to manipulate and we wish to substitute more compact and efficient representations as early as possible. Typically, we produce a sequence of integer values, each standing for a symbol in the source or an attribute of that symbol.

### Simple approaches to lexical analysis

In simple "scanners", lexical analysis merely replaces the atoms of the language with more convenient internal representations. Thus, an integer value may be provided for each keyword, operator and basic symbol in a language. This takes care of the predefined symbols.

At the same time, *white space* may be discarded. This includes any parts of a program with no semantic contribution. This term was originally coined to mean blank spaces, such as space characters and tabs. In addition, comments, which are syntactically ignored or treated as spaces in most languages, may be skipped. Newlines are only significant in certain languages, such as Fortran, where they delimit certain constructs. They may be ignored or treated as spaces in others.

N.b. *the description "left to right" in describing the traversal of a source string usually assumes newlines to have been treated as spaces.*

It may also be possible to ignore certain keywords, which are really aids to readability and have no syntactic importance. The word "program" and the terminating full stop of Pascal might be seen in this way, although they must be checked to be present before being ignored. Less obviously, perhaps, the opening parenthesis before the expression following "if" may be discarded in C.

### Identifiers

Unfortunately, programs do not just consist of predefined symbols. Users are free, in all practical languages, to define their own. Neither the precise form, nor the exact meaning of such symbols is fixed. In Pascal, "Val" might be a variable of any predefined or user defined type, a user defined constant, a procedure, function or program name, a user defined type etc. What is worse, in any given program it might not exist, exist only with one meaning or exist with several meanings.

Simple minded scanners simply substitute a new unique integer value for each new identifier. This leaves the syntax analyser to assign meaning and to distinguish occurrences.

In general, the lexical analyser must pass on the fact that an identifier has been found and must show where an identifier is the same as others (*i.e.* lexically the same). It builds a table of identifiers which it has found, known as the *symbol table*, which enables it to check for earlier occurrences. As we have seen, this will usually be used by later phases to accumulate information about each identifier. In its output, the lexical analyser typically uses pointers into the symbol table to distinguish identifiers.

Most programs contain literals. The set of possible literals in any programming language, all of which have a predefined meaning, is so large as to be beyond the mechanisms defined so far. In practice we want to pass on the value of the literal, having, possibly, converted it to some machine dependent form. This produces a conflict, as:

- some valid integer values must be reserved for our other tokens.
- some literals cannot be represented as single integers.

One solution is to enter each string which represents a literal into a table and to pass the index to this into the stream of lexical tokens. This means reprocessing the string later to extract its value, in the case of arithmetic literals. This may be inefficient, although it may help to reduce duplicate storage of literals during code generation. It is also useful when cross-compiling between machines with different internal representations.

A more flexible mechanism is to use tuples instead of simple integer tokens. Such a tuple for an integer might consist of a token meaning "integer literal", followed by its value.

In general, it is sensible to use tuples to implement much more powerful lexical analysers than is possible with simple integer tokens.

### Hashing

The ability to find and match symbols in such a table is a major requirement and early compilers produced efficient algorithms, notably those based on "hashing". Work on improving these techniques continues.

In general, hashing transforms the internal values of the individual characters in an identifier into a numeric index into the symbol table. This provides an approximate index to the entry for that identifier, if it already exists. Searching is thus localised.

In many languages, the syntax of keywords is the same as that of identifiers. In addition, many languages have predefined identifiers, such as the standard Pascal types. A convenient way of identifying these is to include them in the symbol table before processing of the source begins. This may not always be the solution chosen, however. The inclusion of all symbols in the table provides problems in distinguishing identifiers from other symbols. Where it is done, the predefined strings must be stored using the same hashing function as the scanner uses.

**A simple hashing algorithm** For the sake of an example, here is a simple algorithm provided in the lexical analysis phase of some real compilers.

*Multiply the binary values of the characters together and, at each step, take the remainder* wrt *the maximum value to be returned.*

A crude implementation of this in C would be:

```
#include <stdio.h>
int hash(char* s)
{
    int i,v;
    v=1;
    i=0;
    While(s[i])
{
    v=(v*s[i++])%256;
}
    return v;
}
```

### Literals

The simple minded approach of merely substituting an integer token for each atom is attractive for its simplicity. A certain range can be reserved for basic symbols and all others regarded as indices into the symbol table, indicating an identifier. Unfortunately, this leaves a class of atoms, i.e. literals, unclassified.

This argues more strongly for the use of a tuple as the form of lexical tokens. Treating a literal as an identifier may allow a portable compiler to operate with string to arithmetic value conversion being performed after intermediate code generation.

On the other hand, recognising literals at scanning time and storing scalars as internal values is more efficient in terms of space and may also allow constant expression evaluation at this early stage. (This is sometimes called "constant folding".)

In practice different compilers use different approaches.

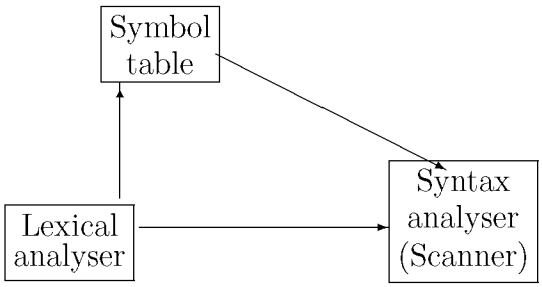
### Organisation and control

In theory the lexical analyser can run in parallel with the syntactic analyser and is free to process ahead. On a single processor based sysem, this is clearly not possible. The lexical analyser is thus a co-routine to the syntactic analyser. Each takes its turn as the analysis proceeds.

In effect, they form a producer/consumer pair.

The buffer is usually of size 1 and is often implemented as a function.

Figure 1: Organisation of front end of a compiler




### Grammars and parsing

There exists a large number of techniques for parsing context free grammars. Most programming languages are deliberately designed to be describable by such grammars to take advantage of these techniques. This raises two questions.

1. What is a context free grammar?

2. Why do we want to parse programs, i.e. identify the syntactic components of them?

We concentrate on considering the answer to the former question in some detail, but a short reply to the latter will help to motivate our studies.

We parse programs because the semantics (sequences of actions or definitions) of programming languages are attached to syntactic structures. In other words, a certain grammatically defined sequence of strings may map on to a certain activity or sequence of activities. The semantics of a language can be viewed as attributes of its syntactic structures.

In general we wish to first parse all or part of a program, then perform certain transformations, such as converting infix to postfix expressions, and finally derive the corresponding actions, as an intermediate or machine code representation.

With our motives clearer, we begin with the definition and analysis of context free grammars. We assume that lexical analysis delivers abstract tokens for our use.

### Context free grammars

Chomsky recognised a hierarchy of grammars and numbered them 0, 1, 2 and 3. This moves from grammars defined very freely (type 0) to those defined in very restrictive ways. A means of describing the differences is in terms of their production rules.

**A type 0 grammar** imposes no restrictions on left hand sides of productions, except that no terminal symbols should appear there. In particular more than one non-terminal symbol may be given.

**A type 1, context dependent, grammar** imposes the restriction that each production replaces only one of its left hand side non-terminals in its right hand side. Thus that (and

only that) production for that non-terminal depends on its being in the context provided by the accompanying left hand side non-terminal symbols.

**A type 2, context free, grammar** allows only one non-terminal symbol on the left hand side of each production. This gives the familiar Backus Normal or Backus Naur Form notation. Context free grammars allow much simpler parsing, as the identification of a single non-terminal allows us to recognise a syntactic construction.

**A type 3, regular definition, grammar** describes languages composed of sets of strings of the form

$$S \rightarrow a\ B$$

where $a$ is a basic symbol and $B$ is a regular expression.

They cannot describe balanced nesting or embedding of constructs, such as parenthesised expressions.

Note that several grammars may define the same language, each supporting a particular parsing scheme.

### Notations and BNF

There are several formats or notational conventions associated with BNF. They are all equivalent and are a product of improvements in the character sets and text processing facilities available to computer users, combined with a search for concise expression of ideas. They all seek to present four fundamental components in terms of a *meta language*. These are

**The production operator** , usually one of ::=, $\Rightarrow$ and $\rightarrow$. I shall use $\rightarrow$.

**Non-terminal symbols** , usually `<name>` or (italicised) *name*. I shall use *name*. When writing by hand these can be underlined instead.

**Terminal symbols** , *i.e.* those identified by the lexical analyser usually one of `<id>` or **id**. I shall use **id**. When writing by hand these are written normally.

**Literal strings** , *i.e.* characters and sequences of characters, usually **begin** or "begin". I shall use **begin**. When writing by hand these are written normally.

In fact the lexical analyser will normally have substituted a lexical value for both literal strings and terminal symbols and the sequence **begin** is used as a mnemonic for the corresponding lexical token within the syntax analyser.

Productions define the set of strings which can replace a non-terminal symbol. In a complete language description there must be a sequence of productions ($\Rightarrow$) leading to literal strings for every non-terminal. As we have seen, this may be unnecessary for the syntax analysis phase, since some productions are regular expressions suitable for lexical analysis. Thus all terminals in the syntactic phase are really lexical tokens.

A concatenation operation is implied by the order of symbols on the right hand side of a production. Note that this allows us to treat the terminal **begin** as either a single value, representing an identifier/keyword, or as five values, representing the characters, **b**, **e**, **g**, **i** and **n**. The interpretation depends on our lexical analysis strategy.

### Meta-symbols

The repetition operators *, + and ? may be used in the same way in all grammars. Operators and parentheses in the meta language will be written in teletype font, *e.g.* ( and ), where this helps to avoid confusion with the corresponding terminal symbols, ( and ).

The *alternative* operator is also the same in all grammars, *i.e.* |, and the same precedence is assumed.

Any language definition which is useful has a single start non-terminal, from which any string in the language may be derived as part of a legal set generated by a sequence of productions.

### Views of productions

There are two ways of interpreting a production rule. One is to use it to derive a string, which is an instance of the left hand side's non-terminal symbol, by successively replacing the non-terminals on the right hand side according to the production rules of those non-terminals. This is equivalent to top down parsing.

The other view is to use the production rules to collapse an existing string back to a single non-terminal, which, for a complete set of strings, *e.g.* a complete program, would leave just the start symbol of the language. This is equivalent to bottom up parsing.