Decomposition of the compilation process leads to interfaces specified by abstract data types, and the basic purposes of these interfaces are largely independent of the source language and target machine. Information crossing an interface between major compilation tasks constitutes a representation of the program in an intermediate language. This representation may or may not be embodied in a concrete data structure, depending upon the structure and goals of a particular compiler. Similarly, the characteristics of a particular compiler may make it useful to summarize the properties of objects in tables stored separately from the program text.

The general characteristics of each interface stem from the modular decomposition of the compiler discussed in Chapter 1. In this chapter we consider several important intermediate languages and tables in detail. By determining the content and possible realization of these interfaces, we place more concrete requirements upon the major compilation tasks.

4.1. Intermediate Languages

Our decomposition leads to four intermediate languages: the token sequence, the structure tree, the computation graph and the target tree. A program is transformed from one to the other in the order given, and they will be presented here in that order.

4.1.1. Token Sequence  Chapter 2 pointed out that a source program is composed of a sequence of basic symbols. These basic symbols, rather than the characters from which they are formed, are the relevant units of the source text. We shall use the term symbol to denote the external representa-
tion of a basic symbol (or an encoding thereof); a *token* is the internal representation.

LAX symbols are described in Section A.1. Production A.1.0.1 classifies them as identifiers, denotations and delimiters respectively. Comments are not basic symbols, and therefore do not appear in the token sequence.

We can characterize the information carried by one token in terms of the type declarations shown in Figure 4.1. *Location* encodes the information required to relate an error message to the source language listing. Section 12.1.3 discusses error reporting mechanisms in detail, and hence we leave the specification of the type *coordinates* open until then.

Most syntactic classes (encoded by members of the enumerated type *tokens*) contain only a single symbol. Tokens representing such symbols need specify only the syntactic class. Only identifiers and denotations require additional information.

A LAX identifier has no intrinsic meaning that can be determined from the character string constituting that identifier. As a basic symbol, therefore, the only property distinguishing one identifier from another is its external representation. This property is embodied in the *sym* field of the token. Section 4.2.1 will consider the type *symbol*, and explain how the external representation is encoded.

The field *intv* or *fptv* is a representation of the value denoted by the source language denotation that the token abstracts. There are several possibilities, depending upon the goals of the particular compiler; Section 4.2.2 considers them in detail.

### 4.1.2. Structure Tree

A structure tree is a representation of a compilation unit in terms of source concepts. It is an ordered tree (in the sense of Section B.1) whose structure is that of an abstract syntax of the source

```plaintext
type tokens =
  identifier, (* A.1.0.2 *)
  integer_denotation, (* A.1.0.6 *)
  floating_point_denotation, (* A.1.0.7 *)
  plus, ..., equivalent, (* specials: A.1.0.10 *)
  and_kw, ..., while_kw); (* keywords: A.1.0.11 *)

abstract_token = record
  location: coordinates; (* for error reports *)
  case classification: tokens of
    identifier: (sym: symbol);
    integer_denotation: (intv: integer_value);
    floating_point_denotation: (fptv: real_value);
  end;

Figure 4.1. LAX Abstract Token
```