Lexical Analysis

The first phase of the compiler is the *lexical analyzer*, also known as the *scanner*, which recognizes the basic language units, called *tokens*.

- The exact characters in a token is called its *lexeme*.
- Tokens are classified by *token types*, e.g. identifiers, constant literals, strings, operators, punctuation marks, and key words.
- Different types of tokens may have their own semantic attributes (or *values*) which must be extracted and stored in the symbol table.

- The lexical analyzer may perform *semantic actions* to extract such values and insert them in the symbol table.
- How to classify token types? It mainly depends on what form of input is needed by the next compiler phase, the *parser*. (The parser takes a sequence of tokens as its input.)

After we decide how to classify token types, we can use one of several ways to precisely express the classification.

- A common method is to use a *finite automaton* to define all character sequences (i.e. strings) which belong to a particular token type.

3

- We shall look at several examples (e.g. in Figure 2.3) of token types and their corresponding finite automata.
- *The states, the starting state, the accepting states* of a finite automaton. An accepting state is also called a *final state*.

Given the definitions of different token types, it is possible for a string to belong to more than one type. Such ambiguity is resolved by assigning priorities to token types. For example: Key words have a higher priority over identifiers.

4

- Finite automata for different token types are combined into a transition diagram for the lexical analyzer.
- Following the "longest match" rule – keep scanning the next character until there is no corresponding transition. The longest string which matches an acceptance state during the scanning is the recognized token.
- Go back to the starting state of the transition diagram, ready to recognize the next token in the program.

- Semantic actions can be specified in the transition diagram.
- (The lexical analyzer can also be used to remove comments from the program.)

Merging several transition diagrams into one may create the problem of $nondeterminism$.

A nondeterministic finite automaton (NFA) $accepts$ an input string x if and only if there exists some path from the start state to some accepting state, such that the edge labels along the path spell out x.

Let us look at examples of NFAs accepting and rejecting strings. Pay attention to the treatment of ϵ in spelling x: $\epsilon y = y$ and $y\epsilon = y$.

Scanners based on NFAs can be inefficient due to the possibility of backtracking.

We study an algorithm which transform an NFA into a DFA (deterministic finite automaton). This algorithm can be found on P. 27.

The intuition behind the algorithm which transforms an NFA to a DFA is $factoring$. Let us look at an extremely simple example first, to see the idea of factoring.

The idea is formalized by identifying $a\ set\ of\ states$ which can be reached after scanning a substring.

For an NFA which contains ϵ edges, we also need to define the ϵ-closure of a state s, which the set of states reachable from s by taking ϵ transitions. The ϵ-closure of s of course includes s itself.

Regular Expressions

A DFA can be easily implemented based on a table look-up.

For a programming language which has a big alphabet and many token types, it would be desirable to *automatically* construct such a table.

9

Such a *lexical-analyzer generator* takes a definition of token types as input and generate the DFA table. The graphical form of DFA is not suitable as an input to the lexical-analyzer generator. Some textual representation is needed.

Regular expressions are such a textual representation.

Regular expressions are equivalent to DFAs in that:

1) For any given regular expression, there exist a DFA which accepts the same set of strings represented by the regular expression.

10

2) For a given DFA, there exist a regular expression which represents the same set of strings accepted by the DFA.

Regular expressions are composed by following a set of syntax rules:

- Given an input alphabet Σ, a regular expression is a string of symbols from the union of the set Σ and the set { (,), *, |, ϵ }
- The regular expression ϵ defines a language which contains the null string. What is the DFA to recognize it?
- The regular expression a defines the language $\{ a \}$.
- If regular expressions R_A and R_B define languages A and B, respectively, then the regular expression $(R_A) \mid (R_B)$ defines the language $A \cup B$, the regular expression $(R_A)(R_B)$ defines the language AB ($concatenation$), and the regular expression $(R_A)^*$ defines the language A^* ($Kleene\ closure$).
- The parentheses define the order of the construction operators (|, * and concatenation) in regular expressions. Within the same pair of parentheses (or among the operators not in any parentheses), * takes the highest precedence, concatenation comes next, | comes last.
- When a pair of parentheses are unnecessary for defining precedences, they can be omitted.

Let us look at a number of examples of regular expressions, including those in Figure 2.2.

Two regular expressions are equivalent if they represent the exactly the same set of strings.

13 There exist algorithms which, for any regular expression E, can directly construct a DFA to recognize the set of strings represented by E. We shall not study such a direct algorithm. Instead, we study an algorithm which construct an NFA for the given regular expression (see P. 25).

We already know how to convert NFA to DFA.

There also exist algorithms which, for any DFA M, can construct a regular expression which represents the set of strings recognized by M. (Unfortunately, sometimes the regular expression generated by such
14 algorithms can be difficult to read.) We do not discuss such algorithms in this course.

The compiler-generator tool JavaCC contains a lexical-analyzer generator. In our project, we will apply the tool to a simple language called miniJava. This will be discussed in our PSOs.