Lexical and Syntax Analysis

Lexical structure

Lexical structure: the structure of the tokens, or words, of a language

Related to, but different than, the syntactic structure

Scanning phase: the phase in which a translator collects sequences of characters from the input program and forms them into tokens

Parsing phase: the phase in which the translator processes the tokens, determining the program's syntactic structure

Lexical Structure

Tokens generally fall into several categories:

- Reserved words (or keywords)
- Literals or constants
- Special symbols, such as “;”, “<=“, or “+”
- Identifiers

Predefined identifiers: identifiers that have been given an initial meaning for all programs in the language but are capable of redirection

Principle of longest substring: process of collecting the longest possible string of nonblank characters

Lexical Structure

Token delimiters (or **white space**): formatting that affects the way tokens are recognized

Indentation can be used to determine structure

Free-format language: one in which format has no effect on program structure other than satisfying the principle of longest substring

Fixed format language: one in which all tokens must occur in pre-specified locations on the page

Tokens can be formally described by regular expressions

Scanning and Parsing

The **scanner** (or **lexical analyzer**) processes the source program, recognizing tokens and returns them one at a time.

- Scanning is essentially pattern matching – each token is described by some pattern for its lexeme

The **parser** (or **syntactic analyzer**) organizes the stream of tokens (from the scanner) into parse trees according to the grammar for the source languages.

Separating the scanner and parser into different stages of the compiler:

- makes implementing the parser easier
- increases portability of the compiler
- permits parallel development
- eases tool support

Lexical analysis

Tokens are usually represented by integer values (constants) in the lexical analysis program

```
#define INT_LIT 10
#define IDENT 11
#define ASSIGN_OP 20
#define ADD_OP 21
#define SUB_OP 22
#define MULT_OP 23
```

It is generally simpler and faster to have the lexical analyzer recognize reserved words as identifiers and then identify them as reserved words later

The lexical analyzer is often responsible for the initial construction of the symbol table. The symbol table may store information about the identifier – attributes, address, score…

Regular languages

The **language** specified by a regular expression is the set of all strings that match the pattern given by the regular expression.

- The language specified by `id` include `foo`, `bar`, `x264`, `tax_Rate`, …

A **regular language** is any language that can be specified by a regular expression. Every regular languages is context free (can be specified by a BNF definition), but not vice-versa.

Regular Expressions

Regular expressions are a notation for defining patterns.

notation	name	meaning	example	matches
[...]	character class	any of a set	`[a-zA-Z]`	any letter
*	Kleene *	0 or more occurrences	`a*`	0 or more a's
+	Kleene +	1 or more occurrences	`a+`	1 or more a's
?	optional		`a?`	a or $\in$
|	alternation		`a | b`	a or b
(…)	grouping		`a(b|c)`	ab or ab a followed by b or c

Regular definitions

A **regular definition** is a named regular expression. Such names can be used in place of the regular expression in following definitions.

- The name of a regular definition is put in { } on the RHS. of another regular definition.
- This is to distinguish to use of the named definition, for example, {digit}, from the simple pattern given by the letters in the name

```
letter = [a-zA-Z]
digit = [0-9]
id = {letter}({letter} | {digit} | _)*
num = (+ | -)?{digit}+.{digit}+
```

Parsing

Two goals of the parser:

- The syntax analyzer check the program to determine if it is syntactically correct
  - Displays an error when found
  - Attempts to recover from error and look for additional errors
- Produce a complete parse tree
  - The parse tree is used for translation

Parsing

A **recognizer** accepts or rejects strings based on whether they are legal strings in the language

A **bottom-up parser** constructs derivations and parse trees from the leaves to the roots

- Matches an input with right side of a rule and reduces it to the nonterminal on the left
- Bottom-up parsers are also called shift-reduce parsers
- Tokens are shifted onto a stack prior to reducing strings to nonterminals

A **top-down parser** expands nonterminals to match incoming tokens and directly construct a derivation

A **parser generator** is a program that automates top-down or bottom-up parsing

Bottom-up parsing is the preferred method for parser generators (also called **compiler compilers**)

Top Down Parsers

A top down parse builds a parse tree in preorder

A preorder traversal of a parse tree begins with the root

Choosing the correct RHS is not easy

- Some of the RHSs of the leftmost terminal may being with a nonterminal

Most common top-down parsing algorithms are closely related

- Recursive descent parsers are coded versions of a syntax analyzer based directly on the BNF of a languages syntax

Parsing Techniques

Left recursion can present problems

- For example: `expr → expr + term | term`
- Infinite loops
- No way to decide which of the two choices is correct until a + is found

EBNF expresses the recursion as a loop

expr → term { + term }

- The curly brackets in EBNF effectively remove left recursion