Compiler Principle

Prof. Dongming LU

Feb. 26th, 2024

Content

2 Lexical Analysis

Introduction

- **Compiling Process**

  To translate a program from one language into another, a compiler first pull it apart and understand its structure and meaning, then put it together in a different way.

- **The front end** ： performs analysis
- **The back end** ： performs synthesis

Introduction

The analysis is usually broken up into

- **Lexical analysis**

  Breaking the input into individual words or "tokens";

- **Syntax analysis**

  Parsing the phrase structure of the program;

- **Semantic analysis**

  Calculating the program's meaning.

Task of the lexical analyzer

- Taking a stream of characters
- Produces a stream of tokens
  - ✓ *names*, *keywords*, and *punctuation marks*
- Discarding *white space* and *comments*

Why separating lexical analysis from parsing ?

- Would unduly complicate the parser

Why discussing lexical analysis?

- Attacked with high-powered formalisms and tools
- Similar formalisms will be useful in the study of parsing
- Similar tools have many applications in areas other than compilation.

2.1 Lexical Token

A lexical token

- A sequence of characters
- A unit in the grammar of a programming language

Token types

- Classification of lexical tokens: A finite set of token types
- Some of the token types of a typical programming language:

Type	Examples
ID	foo n14 last
NUM	73 0 00 515 082
REAL	66.1 .5 10. 1e67 5.5e-10
IF	if
COMMA	,
NOTEQ	!=
LPAREN	(
RPAREN	)

Reserved words, in most languages, not be used as identifiers

- Punctuation tokens such as IF, VOID, RETURN

Non-Tokens

Examples of non-tokens:

comment	**/* try again */**
preprocessor directive	**#include<stdio.h>**
preprocessor directive	**#define NUMS 5, 6**
macro	**NUMS**
blanks, tabs, and new-lines	

The preprocessor deletes the non-tokens

- Operates on the source character stream
- Producing another character stream to the lexical analyzer

An example

Given a program such as

```
float match0(char *s) /* find a zero */
{ if (!strncmp(s, "0.0", 3))
      return 0.;
}
```

The lexical analyzer will return the stream:

FLOAT ID(match0) LPAREN CHAR STAR ID(s)
RPAREN LBRACE IF LPAREN BANG ID(strncmp)
LPAREN ID(s) COMMA STRING(0.0) COMMA NUM(3)
RPAREN RPAREN RETURN REAL(0.0) SEMI
RBRACE EOF

The token-type of each token is reported

Some of the tokens attached ***semantic values***

Such as identifiers and literals, with auxiliary information

Questions?

- How should the lexical rules of a programming language be described?
- In what language should a lexical analyzer be written?

Describing the lexical tokens of a language in English; An example : *identifiers in C or Java*

1. An identifier is a sequence of letters and digits; the first character must be a letter
2. The underscore _ counts as a letter
3. Upper- and lowercase letters are different
4. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token
5. Blanks, tabs, newlines, and comments are ignored except as they serve to separate tokens
6. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants

An ad hoc lexer

- Any reasonable programming language can be used to implement it

A simpler and more readable lexical analyzers

- ***Regular expressions*** :Specify lexical tokens
- ***Deterministic finite automata*** : Implementing lexers
- **Mathematics**: Connecting the above two

2.2 Regular Expression

Some Concepts

- A *language* is a set of *strings*
- A string is a finite sequence of *symbols*
- A symbol is taken from a finite *alphabet*

One Regular Expression

Notices:

- Not assign any meaning to the strings;
- Classify each string as in the language or not.

The notation of *regular expressions*

Symbol: a

For each symbol **a** in the alphabet of the language

The regular expression **a:** denotes the language containing just the string a.

<u>The notation of *regular expressions*</u>

Alternation: A vertical bar ‖

Given two regular expressions *M* and *N*, the alternation operator makes a new regular expression *M* ‖ *N*.

A string is in the language of *M* ‖ *N* if it is in the language of *M* or in the language of *N*.

Example:
The language of **a** ‖ **b** contains the two strings a and b

The notation of *regular expressions*

Concatenation: operator ·

Given two regular expressions *M* and *N*, the concatenation makes a new regular expression $M \cdot N$.

A string is in the language of $M \cdot N$ if it is the concatenation of any two strings α and β such that α is in the language of *M* and β is in the language of *N*.

Example:
The regular expression (**a** ‖ **b**) · **a** defines the language containing the two strings aa and ba.

The notation of *regular expressions*

Epsilon: ϵ

The regular expression ϵ represents a language whose only string is the empty string.

Example: ($a \cdot b$) ‖ ϵ represents the language {"", "ab"}.

Repetition:

Given a regular expression *M*, its Kleene closure is *M**.

A string is in *M** if it is the concatenation of zero or more strings, all of which are in *M*.

Example: ((**a** ‖ **b**) · **a**)* represents the infinite set {"", "aa", "ba", "aaaa", "baaa", "aaba", "baba", "aaaaaa", …}.

The notation of *regular expressions*

Symbols, **Alternation**, **Concatenation**, **Epsilon**, and **Kleene closure**:

- Specify the set of ASCII characters corresponding to the lexical tokens of a programming language

The notation of *regular expressions*

- **Example 1**

$(\mathbf{0} \mid \mathbf{1})^* \cdot \mathbf{0}$

Binary numbers that are multiples of two.

The notation of *regular expressions*

- **Example 2**

b*(**abb***)*(**a**|∈)

Strings of a's and b's with no consecutive a's.

The notation of *regular expressions*

- **Example 3**

Strings of a's and b's containing consecutive a’s.

(**a**|**b**)***aa**(**a**|**b**)*

The notation of *regular expressions*

In writing regular expressions, sometimes the **concatenation** symbol or **the epsilon** will be omitted

Assuming that

- Kleene closure "binds tighter" than concatenation
- Concatenation binds tighter than alternation;

so that

- **ab** | **c** means (**a** · **b**) | **c**, and (**a** |) means (**a** | ∊)

The notation of *regular expressions*

Introducing some more abbreviations:

[**abcd**] means (**a** | **b** | **c** | **d**),

[**b-g**] means [**bcdefg**],

[**b-gM-Qkr**] means [**bcdefgMNOPQkr**],

M? means (*M* | ∊), and *M*+ means (*M*·*M**).

These extensions are **convenient**

None extend the **descriptive power** of regular expressions

Any set of strings that can be described with these abbreviations could also be described by just the basic set of operators.

a	An ordinary character stands for itself.
∊	The empty string. Another way to write the empty string.
M | *N*	Alternation, choosing from *M* or *N*.
M · *N*	Concatenation, an *M* followed by an *N*.
MN	Another way to write concatenation.
*M**	Repetition (zero or more times).
M^+	Repetition, one or more times.
M?	Optional, zero or one occurrence of *M*.
[a − zA − Z]	Character set alternation.
.	A period stands for any single character except newline.
"a.+*"	Quotation, a string in quotes stands for itself literally.

Figure 2.1: **Regular expression notation.**

Regular expressions for some tokens

```
if                      {return IF;}
[a-z][a-z0-9]*                  {return ID;}
[0-9]+                          {return NUM;}
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)    {return
REAL;}
 ("--"[a-z]*"\n")|(" "|"\n"|"\t")+      {/*do nothing*/}
                        { error();}
```

Figure 2.2: Regular expressions for some tokens

The fifth line of the description recognizes comments or white space but does not report back to the parser.

Instead:

- The white space is discarded and the lexer resumed
- The comments begin with two dashes, contain only alphabetic characters, and end with new-line.

Regular expressions for some tokens

```
if                                    {return IF;}
[a-z][a-z0-9]*                        {return ID;}
[0-9]+                                {return NUM;}
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)   {return REAL;}
("--"[a-z]*"\n")|(" "|"\n"|"\t")+     {/*do nothing*/}
.                                     { error();}
```

Figure 2.2: Regular expressions for some tokens.

A lexical specification should be *complete*

- A rule that matches any single character

(and in this case, prints an "illegal character" error message and continues).

Two important disambiguation rules

These rules are a bit ambiguous.

For example,

- Does if8 match as a single identifier or as the two tokens if and 8?
- Does the string if 89 begin with an identifier or a reserved word?

Two important disambiguation rules used by Lex, JavaCC, SableCC, and other similar lexical-analyzer generators:

Two important disambiguation rules

Longest match:

The longest initial substring of the input that can match any regular expression is taken as the next token.

Rule priority:

- For a *particular* longest initial substring, the first regular expression that can match determines its token-type.
- This means that the order of writing down the regular-expression rules has significance.

Thus,

if8 matches as an identifier by the longest-match rule

if matches as a reserved word by rule-priority.

The end of Chapter 2(1)