Lexical Analysis

Principles of Programming Languages

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Department of Computer Science

Faculty of Computer Science and Engineering

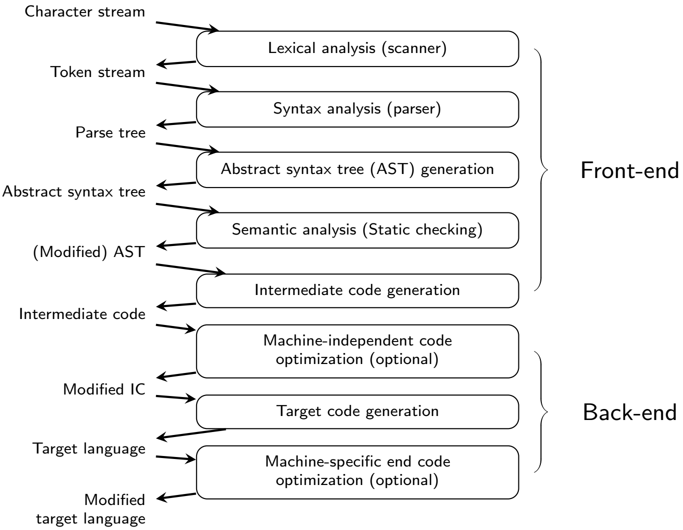
Ho Chi Minh University of Technology, VNU-HCM

Overview

1. **Token and Regular expression**
2. **How to recognize**
   - Ad hoc
   - Finite automaton
3. **ANTLR**

TOKEN AND REGULAR EXPRESSION

An overview of compilation


Character stream
Lexical analysis (scanner)
Token stream
Syntax analysis (parser)
Parse tree
Abstract syntax tree (AST) generation
Abstract syntax tree
Semantic analysis (Static checking)
(Modified) AST
Intermediate code generation
Intermediate code
Machine-independent code optimization (optional)
Modified IC
Target code generation
Target language
Machine-specific end code optimization (optional)
Modified target language
Front-end
Back-end

Tokens

Definition

Tokens are the basic building blocks of programs — the shortest strings of characters with individual meaning, informally to refer to:

- the generic kind
- the specific string (lexeme)

Tokens

Definition

Tokens are the basic building blocks of programs — the shortest strings of characters with individual meaning, informally to refer to:

- the generic kind
- the specific string (lexeme)

Example

C has more than 100 kinds of tokens:

- 44 keywords (`double`, `if`, `return`, `struct`, ...)
- identifiers (`my_variable`, `your_type`, `sizeof`, `printf`, ...)
- integer
- floating-point
- character
- ...

Regular expressions

To specify tokens, we use the notation of regular expressions. A regular expression is one of the following:

1. A character.
2. The empty string, denoted ϵ.
3. Two regular expressions next to each other, meaning any string generated by the first one followed by (concatenated with) any string generated by the second one.
4. Two regular expressions separated by a vertical bar ($|$), meaning any string generated by the first one or any string generated by the second one.
5. A regular expression followed by a Kleene star (*), meaning the concatenation of zero or more strings generated by the expression in front of the star.

Parentheses are used to avoid ambiguity about where the various subexpressions start and end.

Regular expressions: Example

For example, the representation of numeric constants accepted by a simple hand-held calculator:

$$\begin{aligned}
&\mathit{number} \longrightarrow \mathit{integer} \mid \mathit{real} \\
&\mathit{integer} \longrightarrow \mathit{digit}\ \mathit{digit}\,^{*} \\
&\mathit{real} \longrightarrow \mathit{integer}\ \mathit{exponent} \mid \mathit{decimal}\ (\,\mathit{exponent} \mid \epsilon\,) \\
&\mathit{decimal} \longrightarrow \mathit{digit}\,^{*}\ (\ \texttt{.}\ \ \mathit{digit} \mid \mathit{digit}\ \ \texttt{.}\ )\ \mathit{digit}\,^{*} \\
&\mathit{exponent} \longrightarrow (\,\texttt{e} \mid \texttt{E}\,)\,(\,\texttt{+} \mid \texttt{-} \mid \epsilon\,)\ \mathit{integer} \\
&\mathit{digit} \longrightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}
\end{aligned}$$

The symbols to the left of the $\longrightarrow$ signs provide names for the regular expressions.

Regular expressions: Convenience notations

1. $\alpha^{+} = \alpha\alpha*$
2. $\alpha? = \epsilon|\alpha$
3. [xyz] = x | y | z
4. [x-y] = α with α is one of characters from x to y in ASCII digits
5. [^x-y] = α with α is one of characters other than [x-y] in ASCII digits
6. . matches any character

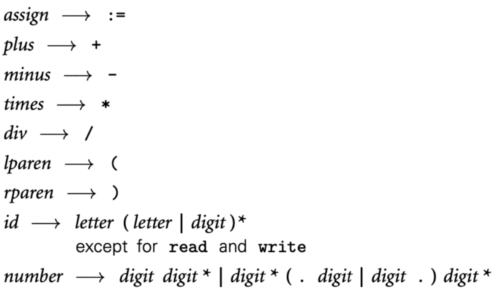
Lexer roles

- Identify **lexemes**
- Return **tokens**
- Ignore **spaces** such as blank, newline, tab
- Record the **position** of tokens that are used in next phases

LEXER:
HOW TO RECOGNIZE?

An esay example

Consider the following set of tokens:



$\textit{assign} \longrightarrow$ `:=`

$\textit{plus} \longrightarrow$ `+`

$\textit{minus} \longrightarrow$ `-`

$\textit{times} \longrightarrow$ `*`

$\textit{div} \longrightarrow$ `/`

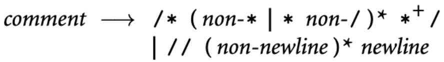
$\textit{lparen} \longrightarrow$ `(`

$\textit{rparen} \longrightarrow$ `)`

$\textit{id} \longrightarrow \textit{letter}\ (\,\textit{letter} \mid \textit{digit}\,)^*$
except for **read** and **write**

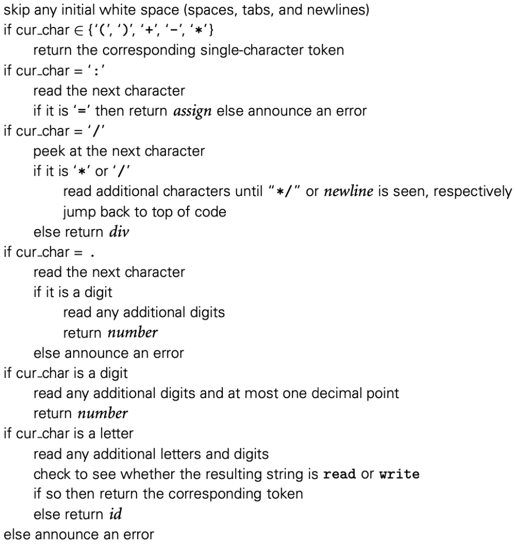
$\textit{number} \longrightarrow \textit{digit}\ \textit{digit}^* \mid \textit{digit}^*\ (\ .\ \textit{digit} \mid \textit{digit}\ .\ )\ \textit{digit}^*$

To make the task of the scanner a little more realistic, we borrow the two styles of comment from C:



$\textit{comment} \longrightarrow$ `/*` $(\,\textit{non-}{*} \mid {*}\ \textit{non-}/\,)^*\ {*}^+$ `/`
$\mid$ `//` $(\,\textit{non-newline}\,)^*\ \textit{newline}$

An ad hoc scanner



```
skip any initial white space (spaces, tabs, and newlines)
if cur_char ∈ {'(', ')', '+', '-', '*'}
    return the corresponding single-character token
if cur_char = ':'
    read the next character
    if it is '=' then return assign else announce an error
if cur_char = '/'
    peek at the next character
    if it is '*' or '/'
        read additional characters until "*/" or newline is seen, respectively
        jump back to top of code
    else return div
if cur_char = .
    read the next character
    if it is a digit
        read any additional digits
        return number
    else announce an error
if cur_char is a digit
    read any additional digits and at most one decimal point
    return number
if cur_char is a letter
    read any additional letters and digits
    check to see whether the resulting string is read or write
    if so then return the corresponding token
    else return id
else announce an error
```

Figure: Outline of an ad hoc scanner for tokens

An ad hoc scanner

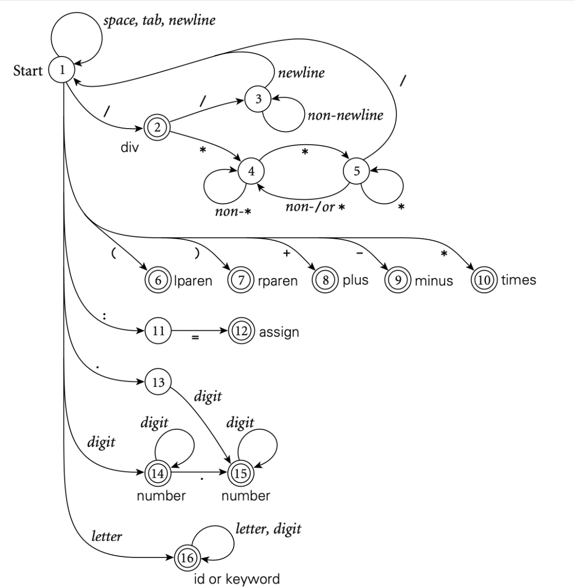
- Reasonable to check the simpler and more common cases first
- Unstructured way to build a scanner

Finite automaton: Structured ways to build scanner

It is usually preferable to build a scanner in a more structured way, as an explicit representation of a *finite automaton*.

- It can be generated automatically from a set of regular expressions, making it easy to regenerate a scanner when token definitions change.
- **How**: The automaton starts in a distinguished initial state. It then moves from state to state based on the next available character of input. When it reaches one of a designated set of final states it recognizes the token associated with that state.
- The "longest possible token" rule means that the scanner returns to the parser only when the next character cannot be used to continue the current token.

Finite automaton: Example



Generating a Finite Automaton

1. Change from a Regular Expression to an NFA
2. Change from an NFA to a DFA
3. Minimizing the DFA

Lexical errors

In some cases the next character of input may be neither an acceptable continuation of the current token nor the start of another token, called **lexical errors**.

- Unclosed string
- Illegal escape in string
- Error token

In such cases the scanner must print an error message.

ANTLR:
LANGUAGE RECOGNITION TOOL

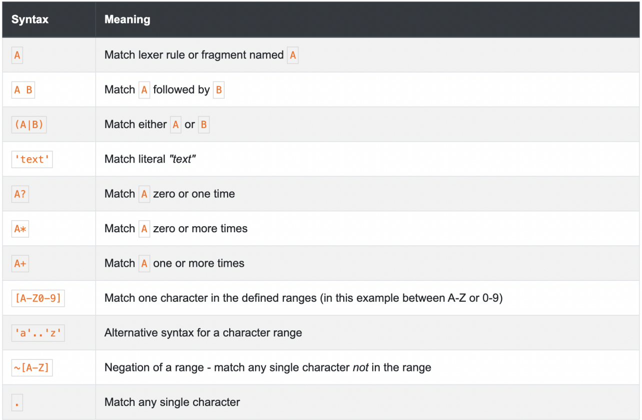
ANTLR: Introduction

ANTLR (ANother Tool for Language Recognition) is a powerful lexer and parser generator by only writing grammar (by regular expressions) of a language.

- Author: Terence Parr, Professor of CS at the University of San Francisco, USA.
- Current version: v4 - ANTLRv4, has some important new capabilities that reduce the learning curve and make developing grammars and language applications much easier.

ANTLRv4: Lexer rules

Token names must begin with an uppercase letter, which distinguishes them from parser rule names.



Syntax	Meaning
`A`	Match lexer rule or fragment named `A`
`A B`	Match `A` followed by `B`
`(A|B)`	Match either `A` or `B`
`'text'`	Match literal *"text"*
`A?`	Match `A` zero or one time
`A*`	Match `A` zero or more times
`A+`	Match `A` one or more times
`[A-Z0-9]`	Match one character in the defined ranges (in this example between A-Z or 0-9)
`'a'..'z'`	Alternative syntax for a character range
`~[A-Z]`	Negation of a range - match any single character *not* in the range
`.`	Match any single character

Figure: How to write regular expression in ANTLR

ANTLRv4: Lexer rules

Several lexer rules can match the same input text. In that case, the token type will be chosen as follows:

- First, select the lexer rule which matches the **longest** input
- If the text matches an implicitly defined token, use the implicit rule
- If several lexer rules match the same input length, choose the first one, based on definition order

ANTLRv4 Lexer: A small example

```
grammar Hello.g4;

// match any integer literals
INTEGER: [0-9]+;

// match any identifiers
IDENTIFIER: [a-zA-Z_] [a-zA-Z_0-9]*;

// match opening and closing parentheses
OPEN_PAREN: ’(’;
CLOSE_PAREN: ’)’;
```

ANTLRv4: Action blocks

A lexer action is a block of arbitrary code in the target language surrounded by {...}, which is executed during matching:

```
grammar Hello.g4;

// match any integer literals
INTEGER: [0-9]+ {print(self.text)};

// match any identifiers
IDENTIFIER: [a-zA-Z_] [a-zA-Z_0-9]*;

// match opening and closing parentheses
OPEN_PAREN: '(';
CLOSE_PAREN: ')';
```

ANTLRv4: Fragments

Fragments are reusable parts of lexer rules which cannot match on their own - they need to be referenced from a lexer rule.

```
grammar Hello.g4;

INTEGER: DIGIT+
       | '0' [Xx] HEX_DIGIT+
       ;

fragment DIGIT: [0-9];
fragment HEX_DIGIT: [0-9A-Fa-f];
```

ANTLRv4: Fragments

A lexer rule can have associated commands:

```
grammar Hello.g4;

WHITESPACE: [ \r\n] -> skip;
```

Commands are defined after a -> at the end of the rule.

- `skip`: Skips the matched text, no token will be emited
- `type(n)`: Changes the emitted token type

ANTLRv4: Reference

- https://riptutorial.com/antlr/topic/2856/introduction-to-antlr-v4
- https://github.com/antlr/antlr4/blob/master/doc/index.md
- Book: The Definitive ANTLR 4 Reference, T. Parr. Pragmatic Bookshelf, Raleigh, NC, 2 edition, (2013)

THANK YOU.