Syntax Analysis: Scanning and Parsing

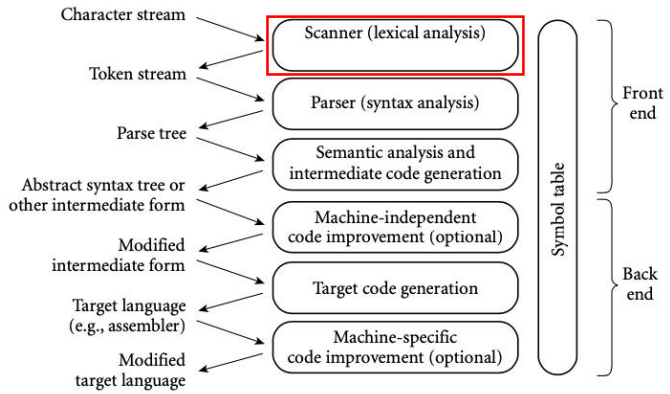
204315: OPL

Outline

- Scanner
  - Regular Expression
  - Deterministic Finite Automaton
- Parser
  - Context-Free Grammar
  - LL Parsing
  - LR Parsing

Scanner

Compilation Process Overview (Recap)



Scott, M. L. (2016). *Programming Language Pragmatics*.

Scanner (Recap)

- **Program tokenization**
- Uses **regular expression**
  - **Regular expressions have the capability to express languages.**

```
int main() {
    int i = getint(), j = getint();
    while (i != j) {
        if (i > j) i = i - j;
        else j = j - i;
    }
    putint(i);
}
```

```
int     main    (       )       {       int     i       =
getint  (       )       ,       j       =       getint  (
)       ;       while   (       i       !=      j       )
{       if      (       i       >       j       )       i
=       i       -       j       ;       else    j       =
j       -       i       ;       }       putint  (       i
)       ;       }
```


Scanner

- responsible for
  - tokenizing source code
  - removing comments
  - (often) dealing with *pragmas* (i.e., compiler directives)
  - saving text of identifiers, numbers, strings
  - saving source locations (file, line, column) for error messages

Scanner/Lexical Analysis

- Unlike natural languages such as English or Thai, **computer languages must be precise**.
  - To provide the precision, **language designers use formal syntactic (syntax) and semantic notation**.
- Different programming languages
  - often provide features with very **similar semantics** but very **different syntax.**
  - It is generally much easier to learn a new language if one is able to identify the common semantic ideas beneath the unfamiliar syntax.

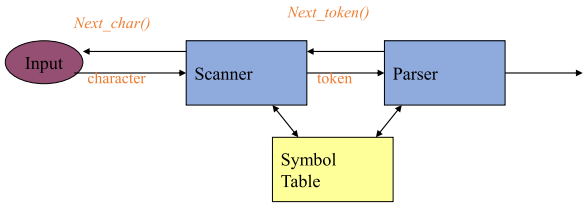
Scanner/Lexical Analysis

- **Lexical analysis** is the first phase of a compilation process.
  - reads/scans character streams from the source code
  - checks for legal tokens
  - passes the data to the syntax analyzer when it demands.
- **Lexical analyzer** needs to scan and identify only a finite set of valid string/token
  - It searches for the pattern defined by the language rules.
- **Regular expressions** have the capability to express finite languages by defining a pattern for finite strings of symbols.
  - The grammar defined by regular expressions is known as **regular grammar**
  - The language defined by regular grammar is known as **regular language**.

Syntax analyzer: checks if the source code follows the grammatical rules of the programming language.

Scanner input/output

- INPUT: sequence of characters
- OUTPUT: sequence of tokens



Some Definitions

- **token** – set of strings that is meaningful in source language
- **pattern** – a rule describing a set of string
- **lexeme** – a sequence of characters that matches the patterns of tokens

Some Examples

Token	Pattern	Sample Lexeme
while	while	while
relation_op	= | != | < | >	<
integer	(0-9)*	42
string	Characters between " "	"hello"

Regular Expression (RE)

- A way to describe pattern that specifies how tokens are generated
- Examples of RE for : *number token, integer token, real token, …*

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

RE and their operations

- RE is one of the followings
  - A character
  - The empty string, denoted by ε
  - Two regular expressions concatenated
  - Two regular expressions separated by | (i.e., or)
  - A regular expression followed by the Kleene star * (concatenation of zero or more strings)

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

Check your understanding

- Are these lexemes generated by these REs?
  - 3.1415
  - 1.1.0
  - 15e-5
  - 3E++
  - .357

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

Recognizing Regular Expression

- A mechanical approach to check if a string is generated by a particular RE is by constructing and using a Deterministic Finite Automaton (DFA)
- A DFA is a finite-state machine that accepts or rejects a given string of symbols, by running through a state sequence uniquely determined by the string
- Multiple version of DFAs can be constructed from a single RE

Deterministic Finite Automaton

- A deterministic finite automaton M is a 5-tuple, (Q , Σ , δ , q0 , F) consisting of
  - a finite set of states Q
  - a finite set of input symbols called the alphabet Σ
  - a transition function $\delta : Q \times \Sigma \rightarrow Q$
  - an initial or start state $q0 \in Q$
  - a set of accept states $F \subseteq Q$

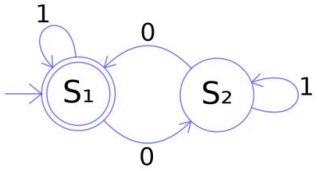
DFA Example

- Given RE = (1*) (0 (1*) 0 (1*))*
  - Possible valid strings
    - 10101
    - 0101
    - 0000
  - Invalid string
    - 000
    - 011
- The corresponding DFA is

$M = (Q, \Sigma, \delta, q_0, F)$ where

- $Q = \{S_1, S_2\}$
- $\Sigma = \{0, 1\}$
- $q_0 = S_1$
- $F = \{S_1\}$ and
- δ is defined by the following state transition table:

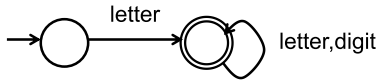
	0	1
S_1	S_2	S_1
S_2	S_1	S_2



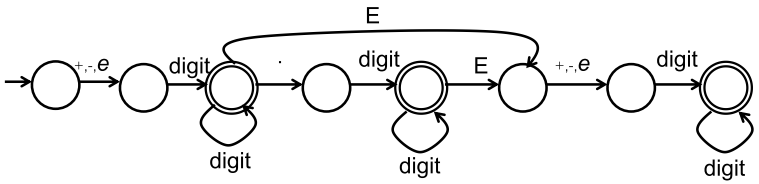
https://en.wikipedia.org/wiki/Deterministic_finite_automaton

More DFA examples

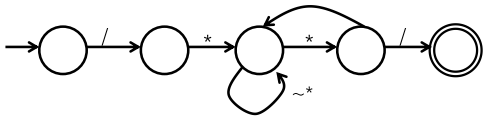
- Identifier



- Numeric



- C comment



Ref: 2301373 Introduction to Compilers

Remarks

- We run the machine (DFA) over and over to get one token after another
  - always take the longest possible token from the input
    thus foobar is foobar and never f or foo or foob
  - more to the point, 3.14159 is a real const and never 3, ., and 14159
- Regular expressions "generate" a regular language; DFAs "recognize" it

How to construct a Scanner?

- Manual
  - Write down regular expressions for your language syntax
  - Construct a DFA for the regular expressions
  - Write a program (as a set of nested IFs) which walks the DFA
- Semi-automatic
  - Write down regular expressions for your language syntax
  - Use automatic softwares such as *lex, flex, jflex* that reads REs and source code and produce the tokenization result

References

- PPT of Lecture 2: Lexical Analysis, CS 540, George Mason University
- PPT of 2301373 Introduction to Compilers
- Scott, M. L. (2016). Programming Language Pragmatics.

What's next?

- Parsing
  - LL Parsing
  - LR Parsing