Automatic Scanner Generation*

Suresh Purini, IIIT-H

Modern optimizing compilers are organized in to a sequence of 3 modules – Front End, Optimizer (Middle End) and Back End, as in Figure 1. The Front End of the compiler checks whether an input program is syntactically and semantically well-formed, if not it will generate suitable error messages for the user. Otherwise it will generate an intermediate representation (IR) of the program which the Optimizer component will optimize with respect to parameters like speed, memory footprint, power etc. The optimized IR is in turn translated into target machine code by the Back End of the compiler. It has to be noted Back End also performs certain machine dependent code optimizations on the IR which could be Back End specific.

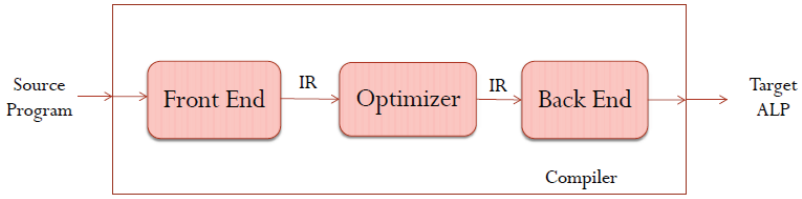

Figure 1: Structure of a Three Phase Compiler

Figure 2 shows the structure of a Front End of the compiler. We can see there are four major components in the front end – Scanner, Parser, Semantic Analyzer and IR Generator. Although it is shown as a sequence of four modules, in compiler implementations, these four modules could be tightly interwoven as a single monolithic component. The Scanner component takes the source program as input and splits it into a sequence of words and their associated parts of speech. A word together with its parts of speech is called is as a Tokens. In the compiler jargon, a word is called a *lexeme* and its parts of speech is called its *token type*. So in general a token is defined as a $\langle lexeme,\ token\ type \rangle$ pair. The parser takes as input a token stream and check if the token stream constitutes a syntactically valid program. In practice, the parser and the scanner works in an interactive fashion as shown in the Figure 3, where the parser keeps asking the scanner for the next token in the token stream untill the token stream gets exhausted.

Given a programming language, what goes into lexical structure (micro-syntax) and what goes in to the higher syntactic structure (macro-syntax) is a design choice. For example consider the following CFG. With respect to this CFG, the possible token types are $\{\ number,\ id,\ op\ \}$.

$$\begin{aligned} Expr &\rightarrow Expr\ op\ Term \mid Term \\ Term &\rightarrow number \mid id \end{aligned}$$

However for the same language the possible token types with respect the following CFG, every

*This document is strictly for IIIT-H internal purpose as I am using certain copyrighted material in this lecture notes.

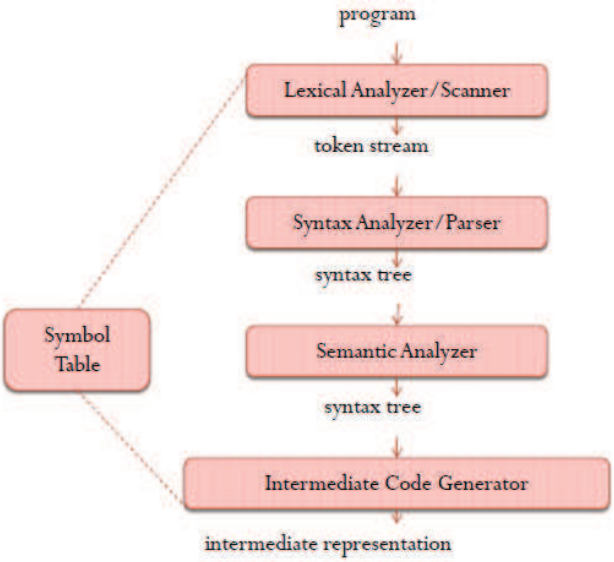

Figure 2: Structure of a Front End

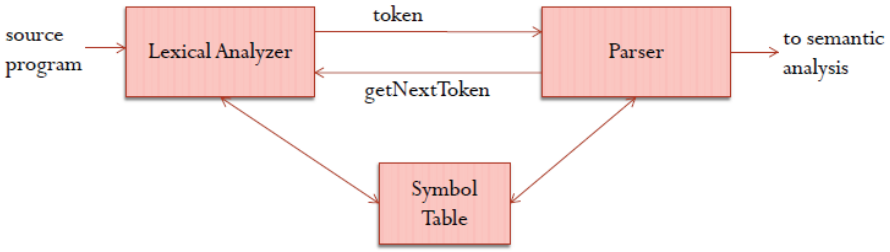

Figure 3: Scanner and the parser works in an interactive fashion

character constitutes a distinct token type.

$$
\begin{aligned}
Expr &\rightarrow Expr\ Op\ Term \mid Term \\
Term &\rightarrow Number \mid Id \\
Number &\rightarrow Digit \mid Digit\ Number \\
Digit &\rightarrow 0 \mid \cdots \mid 9 \\
Id &\rightarrow Alpha \mid AlphaId \\
Alpha &\rightarrow a \mid \cdots \mid z \mid A \mid \cdots \mid Z
\end{aligned}
$$

Question 1. *Between the above possible two CFG options, which option is better? Why?*

1 Logical Design of a Lexical Analyzer

We can follow the following logic steps while designing the Scanner component of a compiler for a programming language.

1. Identify all the possible Tokens that can occur in any valid program. We need to consult the grammar for the programming language to identify the Tokens.

2. Specify the tokens using regular expressions.

3. Either hand-write the lexical analyzer or use a lexical analyzer generator tool like flex or jflex to translate the lexical structure specification into a C, C++ or a Java scanner.

We shall understand the first two steps using an example. Refer Figure 4 for an example lexical structure specification for a C-like programming language. If you notice there are couple of

Lexeme	Token Type	Comment
int	INT	
float	FLOAT	
while	WHILE	
for	FOR	
.....		More keywords
[a-zA-Z][a-zA-Z]*	IDENTIFIER	Any ambiguity in the token specification here?
[0-9][0-9]*	INTLIT	
....	FLOATLIT	
>	GT	Alternatives (>, RELOP), (>, OP)
>=	GE	Any ambiguity in the token specification here?
==	EQ	
....		More operators
(	LPAR	
)	RPAR	
....	...	.....

Figure 4: Lexical structure specification for a C-like programming language

ambiguities in the lexical structure specification. For example if the current lexeme is *int*, then it could be classified into either the token type INT or as an identifier. Another ambiguity is if the curent input pointer is pointing to '>' character in the input string "foo = fun >= fan", and if the parser asks for the next token, the lexical analyzer can either return GT or GE as the next token. These ambiguities can be resolved by using the following two simple rules.

1. Always prefer a longer prefix to a shorter prefix.

2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the lexical specification.

Although these two simple rules resolve most ambiguities in the lexical structure, certain programming languages have very bad syntactic traits which the above two rules may not be able to resolve.

Question 2. *Are you aware of any programming languages whose lexical structure couldn't be specified using the strategy we discussed in this lecture notes?*

Now given a lexical structure specification, we can either hand code the Scanner[1] or use a Scanner generator like Flex. A scanner generator like Flex (refer Figure 5) would take the lexical

[1]Perhaps it is not even necessary to write a clear lexical structure specification if we want to handcode a Scanner, nevertheless it is a very good software engineering practice

structure specification in a certain format and generates C/C++ scanner function named **yylex()** which we can be called by a parser whenver it requires a token. In the rest of the discussion, we

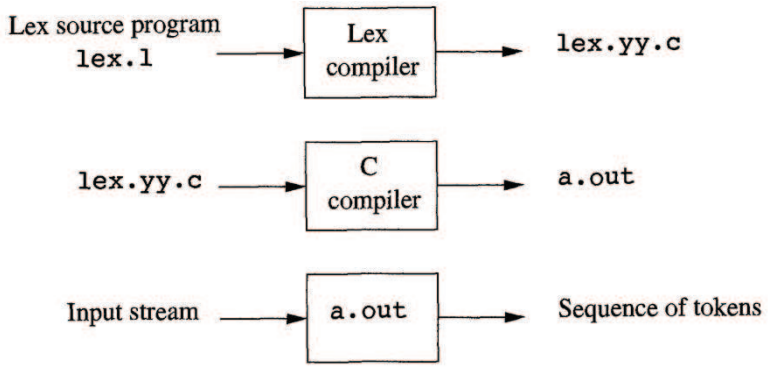

Figure 5: How Flex Works?

shall see how a scanner generator converts a lexical structure specification given as a sequence of regular expressions (REs) in to a scanner[2]. We shall first start with a simpler formulation of the problem and keep refining it to move towards a full fledged scanner generator.

Problem 1: Given a regular expression **R**, generate a C program **IsR(char *str)** which checks whether an input string is described by the RE **R**.
Approach:

1. Convert the RE **R** into an NFA.
2. Convert NFA to DFA.
3. Minimize the DFA.
4. Translate the DFA into an equivalent program function.

All the above steps can be carried out algorithmically. The time complexity of the resulting function is $O(t)$ where t is the length of the input string.

There are two approaches to generate a C-function simulating a given DFA.

1. Table Driven Scanners
2. Direct Coded Approach

In a Table-Driven scanner, we use an explicit data structure like a 2D-array to store the transistion function of a DFA, whereas in the Direct-Coded approach, the transistion function is encoded in the scanner function itself using a set of **goto** statements. These two approaches can be understood by observing the respective scanners (refer Algorithms 1 and 2) for the DFA corresponding to a floating point literal in the Figure 6. One can easily envisage that for an arbitrary RE, we can generate a suitable scanner function by appropriately replacing the state transistion function in the table driven scanner approach. However extra effort needs to be made to emit suitable code in a direct-coded scanner.

[2]An interesting observation here is the Flex program takes as input a token specification and generates C code. It is an example of a Compiler/Translator which translates a source program, here a flex program, in to another program in high level language. We have also translators from C++ to C, PHP to C etc.

Algorithm 1 Table Driven Scanner

IsFloatingPoint(String str)

```
1:  state = s_0
2:  for i = 1 to str.length do
3:      state = nextState(state, str[i])
4:      if (state == s_e) then
5:          return reject
6:      end if
7:  end for
8:  if (state ∈ F) then
9:      return accept
10: else
11:     return reject
12: end if
```

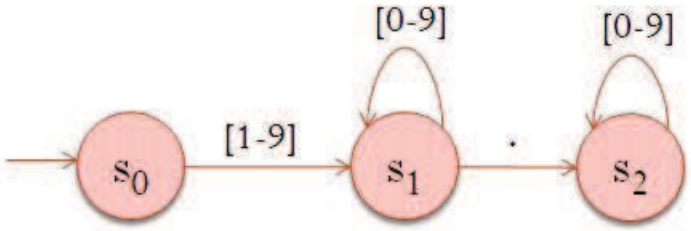

Figure 6: DFA recognizing a floating point literal

Question 3. *What are the pros and cons of using table-driven scanners against direct-coded scanners? You may want to think along the following directions.*

1. *Computations per character.*
2. *Memory hierarchy utilization (both instruction and data).*
3. *Effectivness of the processor's branch prediction strategy.*
4. *Any other parameters I missed?*

Question 4. *On Time Complexity.*

1. *What is the time complexity of the Scanner Generator?*
2. *What is the time complexity of the Generated Scanner?*

Question 5. *Given an sequence of REs $R_1, \cdots, R_k$, design a scanner generator which emits a function* **checkString(String str)** *which takes a string as input and outputs the index i of the first RE R_i describing the input string. The time complexity of the emitted* **checkString(String str)** *function should be at most $O(kt)$ where k is the number of REs and t is the length of the input string.*

The challenge now is to come up with a scanner generator which can emit a **checkString(String str)** function whose time complexity is $O(t)$. To solve this problem let us take a brief detour. Recall

Algorithm 2 Direct Coded Scanner

```
IsFloatingPoint(String str)
 1: i = 1
 2: s0:
 3: if (i > str.length) then
 4:     return reject
 5: end if
 6: if ('1' ≤ str[i] ≤ '9' ) then
 7:     goto s1
 8: else
 9:     return reject
10: end if
11: s1:
12: i = i + 1
13: if (i > str.length) then
14:     return reject
15: end if
16: if ('0' ≤ str[i] ≤ '9' ) then
17:     goto s1
18: end if
19: if ( str[i] = '.' ) then
20:     goto s2
21: else
22:     return reject
23: end if
24: s2:
25: i = i + 1
26: if ( i > str.length ) then
27:     return accept
28: end if
29: if ( '0' ≤ str[i] ≤ '9' ) then
30:     goto s2
31: else
32:     return reject
33: end if
```

that a DFA M is defined by a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where Q is the set of states, Σ is the input alphabet, $\delta : Q \times \Sigma \to Q$ is a state transistion function, $q_0 \in Q$ is the initial state and $F \subseteq Q$ is the set of final states. We can define a function $\hat{\delta} : Q \times \Sigma^* \to Q$ as follows: $\hat{\delta}(q, \epsilon) = q$ and $\hat{\delta}(q, aw) = \hat{\delta}(\delta(q, a), w)$ where $a \in \Sigma$, $w \in \Sigma^*$. We say that a DFA accepts a string $w \in \Sigma^*$ if and only if $\hat{\delta}(q_0, w) \in F$. Given two DFAs $M_1 = (Q_1, \Sigma, \delta_1, q_0, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, p_0, F_2)$, we can construct a product DFA $M_{12} = (Q_1 \times Q_2, \Sigma, \delta, [p_0, q_0], F)$ where $\delta([p, q], a) = (\delta_1(p, a), \delta_2(q, a))$. In general it can be seen that $\hat{\delta}([p, q], w) = (\hat{\delta}_1(p, w), \hat{\delta}_2(q, w))$. The set of final states F depends on what would like the product DFA to recognize.

Question 6. *1. How should we define the set of states F of the DFA M_{12} if we would like $L(M_{12}) = L(M_1) \cap L(M_2)$?*

2. *How should we define the set of states F of the DFA M_{12} if we would like $L(M_{12}) = L(M_1) \cap \overline{L(M_2)}$?*

You could now take some time and think how to use product DFAs to come up with a **checkString(String str)** function with time complexity $O(t)$. The following are the steps in constructing such a function.

1. Corresponding to each of the REs $R_1, \cdots, R_k$ construct respective DFAs $M_1, \cdots, M_k$.
2. Now construct a product DFA $M = M_1 \times \cdots \times M_k$.
3. Apply the $\hat{\delta}$ function on the input string and let the resultant state in the product DFA M be $[p_1, \cdots, p_k]$. Return the first index i such that $p_i \in F_i$.

Having come up with a $O(t)$-algoritm for **checkString**, our problem in the context of the lexical analysis still remains only partially solved.

Question 7.

What is problem that is yet not solved? How do you propose to solve it? What is the time complexity of the final scanner that is generated?