Lecture 3: Lexical Analysis

COMP 524 Programming Language Concepts
Stephen Olivier
January 20, 2009

Based on notes by A. Block, N. Fisher, F. Hernandez-Campos, J. Prins and D. Stotts
Goal of Lecture

Scanner (lexical analysis)

Parser (syntax analysis)

Semantic analysis &
intermediate code gen.

Machine-independent
optimization (optional)

Symbol Table

This includes regular expressions.

Target code generation.

Machine-specific
optimization (optional)

Modified target language

Machine language

Modified intermediate form

Parse Tree

Token Stream

Character Stream

This includes regular expressions.
Scanning

- The main task of scanning is to **identify tokens**.
Pseudo-Code Scanner (Fig 2.5)

We skip any initial white spaces
we read the next character
if it is a ( we look at the next character
  if that is a * we have a comment;
    we skip forward through the terminating *)
  otherwise we return a ( and reuse the look-ahead
If it is one of the one-character tokens (\[],\;=+- etc.)
  we return that token
...

We skip any initial white spaces
we read the next character
if it is a ( we look at the next character
if that is a * we have a comment;
we skip forward through the terminating *)
otherwise we return a ) and reuse the look-ahead
If it is one of the one-character tokens ([],;=+- etc.)
we return that token

We could just turn this into real code and
use that as the scanner, and that would be
fine for small programs...
Pseudo-Code Scanner (Fig 2.5)

We skip any initial white spaces
we read the next character
if it is ( we look at the next character
if that is a * we have a comment;
we skip forward through the terminating *)
otherwise we return a ) and reuse the look-ahead
If it is one of the one-character tokens ([],;=+- etc.)
we return that token
...

... However, for larger programs that must be correct a more formal approach is more appropriate.
Regular expressions

\[\begin{align*}
  \text{digit} &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
  \text{non\_zero\_digit} &\rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
  \text{natural\_number} &\rightarrow \text{non\_zero\_digit} \text{ digit}* \\
  \text{non\_neg\_number} &\rightarrow (0 \mid \text{natural\_number}) ((. \text{ digit}* \text{ non\_zero\_digit}) \mid \varepsilon)
\end{align*}\]
Regular expressions

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → non_zero_digit digit*

non_neg_number → (0 | natural_number) ( ( . digit* non_zero_digit ) | ε )

“→” denotes assignment
Regular expressions

\[
digit \rightarrow 0 | 1 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\]

\[
\text{non_zero_digit} \rightarrow 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\]

\[
natural_number \rightarrow \text{non_zero_digit} \text{ digit}^*
\]

\[
\text{non_neg_number} \rightarrow (0 | \text{natural_number}) \text{ ( . digit}^* \text{ non_zero_digit) | } \varepsilon
\]

“|” denotes \textbf{or}
Regular expressions

\[\text{digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\]

\[\text{non_zero_digit} \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\]

\[\text{natural_number} \rightarrow \text{non_zero_digit} \text{ digit}^*\]

Thus, digit equal “0” or “1” or “2” or ....
Regular expressions

"*" denotes **zero or more of this type.**

\[
\text{non_zero_digit} \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]

\[
\text{natural_number} \rightarrow \text{non_zero_digit} \ \text{digit}^*
\]

\[
\text{non_neg_number} \rightarrow (0 \mid \text{natural_number}) ((. \ \text{digit}^* \ \text{non_zero_digit}) \mid \varepsilon)
\]
Regular expressions

Two REs next to each other denotes concatenation.

- `non_zero_digit` → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- `natural_number` → `non_zero_digit` digit*
- `non_neg_number` → (0 | `natural_number`) ( ( . digit* `non_zero_digit`) | ε )
Regular expressions

So natural number equals at least 
“one non-zero digit” followed by 
“zero or more digits”.

\[
\text{non_zero_digit} \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]

\[
\text{natural_number} \rightarrow \text{non_zero_digit} \ \text{digit}^* 
\]

\[
\text{non_neg_number} \rightarrow (0 \mid \text{natural_number}) \ (\ . \ \text{digit}^* \ \text{non_zero_digit}) \mid \varepsilon
\]
Regular expressions

(non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)

natural_number → non_zero_digit digit*

non_neg_number → (0 | natural_number) ( ( . digit* non_zero_digit ) | ε )

“ε” means empty.
Regular expressions

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → non_zero_digit digit*

non_neg_number → (0 | natural_number) ( (. digit* non_zero_digit) | ε )

So, what does this mean?
Regular expressions

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → non_zero_digit digit*

non_neg_number → (0 | natural_number) ( ( . digit* non_zero_digit) | ε )

It means “0 or a natural number” followed by “nothing” or by “. and zero or more digits concluded by a non_zero number”
Regular Expression Rules

• A RE consist of:
  • A character (e.g., “0”, “1”, ...)
  • The empty string (i.e., “ε”)
  • Two REs next to each other (e.g., “non_negative_digit digit”) to denote concatenation.
  • Two REs separated by “|” next to each other (e.g., “non_negative_digit | digit”) to denote one RE or the other.
  • An RE followed by “*” (called the Kleene star) to denote zero or more iterations of the RE.
  • Parentheses (in order to avoid ambiguity).
Regular Expression Rules

• A RE consist of:
  • A character (e.g., “0”, “1”, ...)
  • The empty string (i.e., “ε”)
  • Two REs next to each other (e.g., “non_negative_digit digit”) to denote concatenation.
  • Two REs separated by “|” next to each other (e.g., “non_negative_digit | digit”) to denote one RE or the other.
  • An RE followed by “*” (called the Kleene star) to denote zero or more iterations of the RE.
  • Parentheses (in order to avoid ambiguity).

A RE is NEVER defined in terms of itself! Thus, REs cannot define recursive statements.
Regular Expression Rules

- A RE consist of:
  - A character (e.g., “0”, “1”, ...)
  - The empty string (i.e., “ε”)
  - Two REs next to each other (e.g., “non_negative_digit digit”) to denote concatenation.
  - Two REs separated by “|” next to each other (e.g., “non_negative_digit | digit”) to denote one RE or the other.
  - An RE followed by “*” (called the Kleene star) to denote zero or more iterations of the RE.
  - Parentheses (in order to avoid ambiguity).

The set of tokens that can be recognized by regular expressions is called a **regular set**.
Deterministic finite automaton (DFA)

• Every regular set can be defined by using deterministic finite automaton (DFA).
  
  • DFAs are turing machines that have a finite number of states and deterministically move between states.
Deterministic finite automaton (DFA)

- Every regular set can be defined by using **deterministic finite automaton**.
- DFAs are turning machines that have a finite number of states and deterministically move between states.
Deterministic finite automaton (DFA)

• Every regular set can be defined by using **deterministic finite automaton** (DFA).

• DFAs are turning machines that have a finite number of states and deterministically move between states.
Deterministic finite automaton (DFA)

• Every regular set can be defined by using **deterministic finite automaton** (DFA).

• DFAs are turning machines that have a finite number of states and deterministically move between states.

End State (double circle)
Deterministic finite automaton (DFA)

- Every regular set can be defined by using **deterministic finite automaton** (DFA).

- DFAs are turning machines that have a finite number of states and deterministically move between states.

This stands for: $0^*10^*1(0|1)^*$
Constructing DFAs

A DFA can be constructed from a RE via two steps.

1. Construct a nondeterministic finite automaton (NFA) from the RE.
2. Construct a DFA from the NFA.
3. Minimize the DFA
What is an NFA?

• An NFA is similar to a DFA, except that state transitions are nondeterministic.

• This nondeterminism is encapsulated via the \textit{epsilon transition} (written as $\varepsilon$).
What is an NFA?

- An NFA is similar to a DFA, except that state transitions are nondeterministic.
- This nondeterminism is encapsulated via the epsilon transition (written as $\varepsilon$).

This stands for: $0|1$
What is an NFA?

- An NFA is similar to a DFA, except that state transitions are nondeterministic.
- This nondeterminism is encapsulated via the epsilon transition (written as $\varepsilon$).

The $\varepsilon$ transitions imply that either transition can be taken with any (or no) input.
The four RE rules and NFA

Rule 1---**Base case:** “a”
The four RE rules and NFA

Rule 2---**Concatenation**: “AB”
The four RE rules and NFA

Rule 2 -- Concatenation: “AB”

Stands for Some NFA called “A”
The four RE rules and NFA

Rule 3--**Alternation**: “A|B”
The four RE rules and NFA

Rule 3: **Alternation**: “A|B”

Notice the epsilon transitions.
The four RE rules and NFA

Rule 4--**Kleene Closure**: “A*”

S

A

empty or repeated

S

ε

A

ε

A

ε

A*
The four RE rules and NFA

Rule 4: Kleene Closure: “^*”

Notice the epsilon transitions.

S

A

empty or repeated

A

ε

ε

ε

ε

ε

ε

Notice the epsilon transitions.
Some examples:

- $0|1^*$
- $AB^*$
- $F|(GH^*)$
- $Z^*|\varepsilon|Y^*X$
Constructing DFAs

- A DFA can be constructed from a RE via two steps.
  1. Construct a **nondeterministic finite automaton (NFA)** from the RE.
  2. Construct a DFA from the NFA.
  3. Minimize the DFA.
Constructing DFAs

- A DFA can be constructed from a RE via two steps.
  1. Construct a **nondeterministic finite automaton (NFA)** from the RE.

```
FOR_KEYWORD → for
IDENTIFIER → Alph ( Alph | Dig )*  
REAL → Dig Dig* . Dig*  
INT → Dig Dig*
```
Constructing DFAs

• A DFA can be constructed from a RE via two steps.
  
  1. Construct a **nondeterministic finite automaton (NFA)** from the RE.
  
  2. Construct a DFA from the NFA.
  
  3. Minimize the DFA
Constructing a DFA from an NFA.

• Construct the DFA by “collapsing” the states of an NFA.

• Three steps

  1. Identify set of states that can be reached from the start state via epsilon-transitions and make this one state.

  2. For a given DFA state (which is a set of NFA states) consider each possible input and combine the resulting NFA states into one DFA state.

  3. Repeat Step 2 until all states have been added.
An example

NFA

DFA

Start

\(S, A, B, F\)

\(C, G, E, F\)

\(B, D, F\)

\(G, E, F\)
All the states that we can reach via $\varepsilon$ are in this state.
All the states that we can reach via 0 or ε from SABF

Start

NFA

DFA

S,A,B,F

C,G,E,F

B,D,F

G,E,F
All the states that we can reach via 1 or $\varepsilon$ from SABF
All the states that we can reach via 0 or $\varepsilon$ from B,D,F or C,G,F
An example

NFA

DFA

Start

Self Loop.
An example

\[ 0|1^*00^* \]

NFA

DFA

Start
Minimize via partitioning

• First, partition states into final and non-final
• Second, determine the effect of the state transition based on what partition the transition goes to.
• Third, Create new partition for those states that have different transitions.
• Fourth, repeat.
An example

Start

DFA

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S,A,B,F</td>
<td>X-2</td>
<td>X-1</td>
</tr>
<tr>
<td>C,G,E,F</td>
<td>X-2</td>
<td>X-1</td>
</tr>
<tr>
<td>B,D,F</td>
<td>X-2</td>
<td>N/A</td>
</tr>
<tr>
<td>G,E,F</td>
<td>X-2</td>
<td>N/A</td>
</tr>
</tbody>
</table>
An example

Start

\[
\begin{align*}
&X-1 \\
\rightarrow & \quad X-2 \\
\end{align*}
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{State} & 0 & 1 \\
\hline
\text{SABF} & X-2 & X-1 \\
\hline
\text{BDF} & X-2 & X-1 \\
\hline
\text{CGEF} & X-2 & \text{N/A} \\
\hline
\text{GEF} & X-2 & \text{N/A} \\
\hline
\end{array}
\]
Scanner Code

- Can create Scanner from the DFA one of two ways:
  - Nested case statements (Handwritten)
  - Tables (easy to generate from code, hard to write by hand)
Two complications--Nested Case

• **Keywords**
  - It is possible to maintain a DFA for keywords, but the number of states would be even larger! So, they are handled as exceptions to the rule.

• **“Dot-Dot”**
  - Pascal uses “..” to denote a range of numbers; however, to determine the meaning of the “..” we need to “look ahead” after reading the first “.” to determine if “.” denotes the end of a token or a beginning of a new token.
    - “3.14” one token
    - “2 .. 5” three tokens
This code specifies a two-dimensional transition table, which tells “whether to move, return token, or announce error”
A second table tells when we might have hit the end of a token (for backing up)
Pragmas

- **Pragmas** are “comments” that provide direction for the compiler.
  - For example, “Variable x is used a lot, keep it in memory if possible.”
- These are often handled by the parser since this makes the grammar much simpler.