A small example of language translation

$L(Add) =$

sums of two digits, expressed expressed in the usual (infix) notation

That's not very formal. What do we mean by this?

$$\{0+4, 3+7, \ldots\}$$

```
input(s)
case (s)
  of("0+0") return(OK)
  ...
  of("9+9") return(OK)
  default   return(BAD)
endcase
```

The program shown on the left *recognizes* **the** Add **language. Suppose we want to** *translate* **strings in** Add **into their sum, expressed base-4.**

```
input(s)
case (s)
  of("0+0") return("0")
  ...
  of("5+7") return("30")
  ...
  of("9+9") return("102")
  default   oops(BAD)
endcase
```

A *language* is a *set of strings*. With 100 possibilities, we could easily list all strings in this (small) language. This approach seems like lots of work, especially for languages with infinite numbers of strings, like C. We need a finite specification for such languages.

Grammars

The grammar below *generates* the Add language:

$$
\begin{array}{lcl}
\mathbf{S} & \rightarrow & \mathbf{D + D} \\
\mathbf{D} & \rightarrow & \mathbf{0} \\
 & | & \mathbf{1} \\
 & | & \mathbf{2} \\
 & \vdots & \\
 & | & \mathbf{9}
\end{array}
$$

A grammar is formally

$$G = (V, \Sigma, P, S)$$

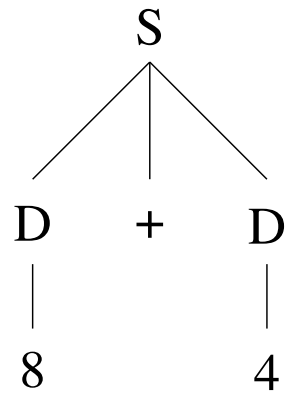
where

- V **is the set of nonterminals. These appear on the left side of rules.**
- Σ **is an alphabet of terminal symbols, that cannot be rewritten.**
- P **is a set of rewrite rules.**
- S **is the *start* or *goal* symbol.**

The process by which a terminal string is created is called a *derivation*.

$$
\begin{array}{lcl}
\mathbf{S} & \Rightarrow & \mathbf{D + D} \\
 & \Rightarrow & \mathbf{8 + D} \\
 & \Rightarrow & \mathbf{8 + 4}
\end{array}
$$

This is a *leftmost* derivation, since a string of nonterminals is rewritten from the left. A tree illustrates how the grammar and the derivation *structure* the string:



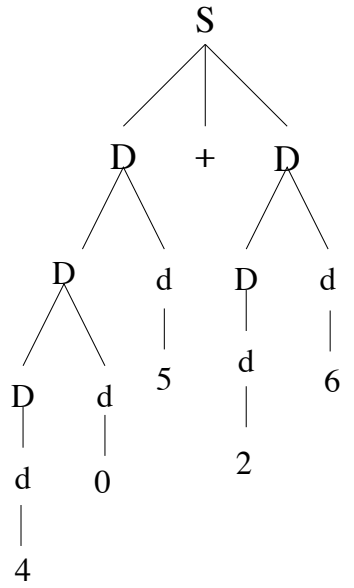
The above could be called a derivation tree, a (concrete) syntax tree, or a parse tree.

Strings in $L(G)$ are constructed by rewriting the symbol S according to the rules of P until a terminal string is derived.

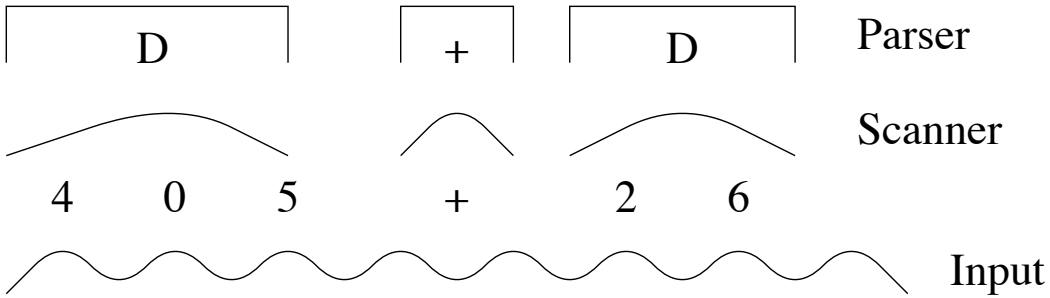
Sums of two numbers

Consider the set of strings that represent the sum of two numbers, such as $405 + 26$**. We could rewrite the grammar, as shown below:**

$$
\begin{array}{lcl}
\mathbf{S} & \rightarrow & \mathbf{D + D} \\
\mathbf{D} & \rightarrow & \mathbf{D\ d} \\
 & | & \mathbf{d} \\
\mathbf{d} & \rightarrow & \mathbf{0} \\
 & | & \mathbf{1} \\
 & \vdots & \\
 & | & \mathbf{9}
\end{array}
$$



Another solution would be to have a separate *tokenizing* process feed "D"s to the grammar, so that the grammar remains unchanged.



Scanners

Scanners are often the ugliest part of a compiler, but when cleverly designed, they can greatly simplify the design and implementation of a parser.

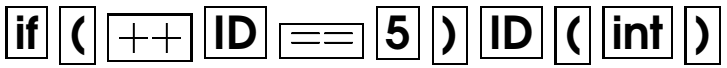
Typical tasks for a scanner:

- **Recognize reserved keywords.**
- **Find integer and floating-point constants.**
- **Ignore comments.**
- **Treat blank space appropriately.**
- **Find string and character constants.**
- **Find identifiers (variables).**

The C **statement**

```
if (++x==5) foo(3);
```

might be tokenized as



Unusual tasks for a scanner:

- **In (older)** FORTRAN**, blanks are optional. Thus, the phrases**

  `DO10I=1,5` **and** `DO10I=1.5`

  are distinguished only by the comma vs. the decimal. The first statement is the start of a DO loop, while the second statement assigns the variable `DO10I`**.**
- **In** C**, variables can be declared by built-in or by user-defined types. Thus, in**

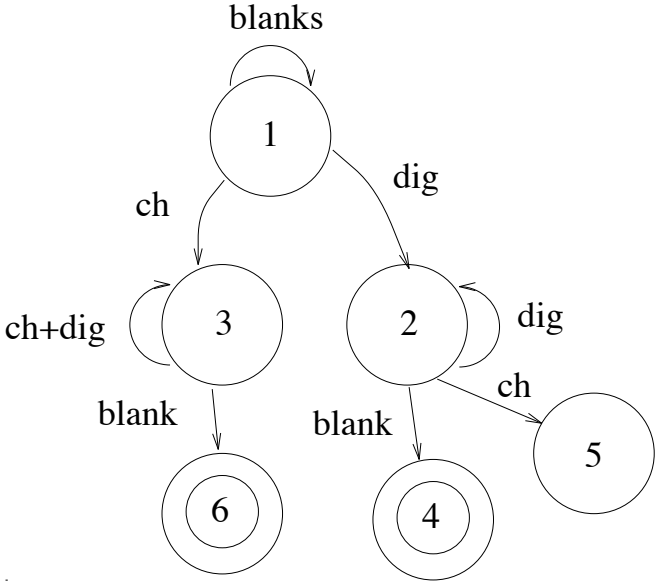
  ```
  foo x,y;
  ```

  the C **grammar needs to know that** `foo` **is a type name, and not a variable name.**

The balance of work between scanner and parser is typically dictated by restrictions of the parsing method and by a desire to make the grammar as simple as possible.

Scanners and Regular Languages

Most scanners are based on a simple computational model called the *finite-state automaton*.



These machines recognize *regular languages*.

To implement a finite-state transducer one begins with a GOTO **table that defines transitions between states:**

	GOTO table		
State	**ch**	**dig**	**blank**
1	**3**	**2**	**1**
2	**5**	**2**	**4**
3	**3**	**3**	**6**
4	**3**	**2**	**4**
5	**5**	**5**	**5**
6	**3**	**2**	**6**

which is processed by the driver

$state \leftarrow 1$
while (**true**) **do**
    $c \leftarrow NextSym()$
    /$\star$ Do action ACTION[$state$][c] $\star$/
    $state \leftarrow$ GOTO[$state$][c]
od

Notice the similarity between states 1, 4, and 6.

Transduction

While the finite-state mechanism *recognizes* appropriate strings, action must now be taken to construct and supply tokens to the parser. Between states, actions are performed as prescribed by the ACTION table shown below.

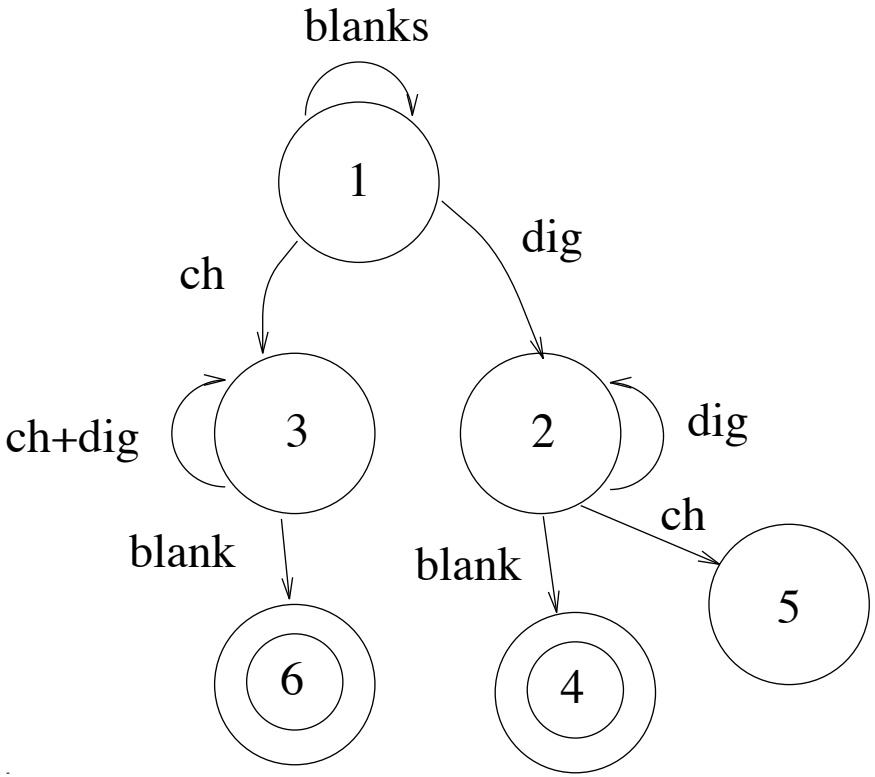
	ACTION table		
State	**ch**	**dig**	**blank**
1	**1**	**2**	**3**
2	**4**	**5**	**6**
3	**7**	**7**	**8**
4	**1**	**2**	**3**
5	**4**	**4**	**4**
6	**1**	**2**	**3**

Actions

1. $ID = ch$
2. $Num = dig$
3. **Do nothing**
4. **Error**
5. $Num = 10 \times Num + dig$
6. **return NUM**
7. $ID = ID||ch$
8. **return ID**

Technically, the ability to perform arbitrary actions makes our tokenizer more powerful than a finite-state automaton. Nonetheless, the underlying mechanism is quite simple, and can in fact be automatically generated. . . .

Regular grammars



$$
\begin{aligned}
\boxed{1} &\rightarrow \textbf{blank}\ \boxed{1} \\
&\mid \textbf{ch}\ \boxed{3} \\
&\mid \textbf{dig}\ \boxed{2} \\
\boxed{2} &\rightarrow \textbf{dig}\ \boxed{2} \\
&\mid \textbf{ch}\ \boxed{5} \\
&\mid \textbf{blank}\ \boxed{4} \\
\boxed{3} &\rightarrow \textbf{ch}\ \boxed{3} \\
&\mid \textbf{dig}\ \boxed{3} \\
&\mid \textbf{blank}\ \boxed{6} \\
\boxed{4} &\rightarrow \lambda \\
\boxed{6} &\rightarrow \lambda
\end{aligned}
$$

In a regular grammar, each rule is of the form

$$
\begin{aligned}
A &\rightarrow a\,A \\
A &\rightarrow a
\end{aligned}
$$

where $A \in V$ and $a \in (\Sigma \cup \{\lambda\})$.

LEX as a scanner

First, define character classes:

```
ucase    [A-Z]
lcase    [a-z]
letter   ({ucase}|{lcase})
zero     0
nonzero  [1-9]
sign     [+-]
digit    ({zero}|{nonzero})
blanks   [ \t\f]
newline  \n
```

Next, specify patterns and actions:

{L}({L}|{D})*	{ String(yytext);
	  return(ID);
	}

''++''	{ return(IncOP); }

In selecting which pattern to apply, LEX uses the following rules:

1. **LEX always tries for the longest match. If any pattern can "keep going" then LEX will keep consuming input until that pattern finishes or "gives up". This property frequently results in buffer overflow for improperly specified patterns.**
2. **LEX will choose the pattern and action that succeeds by consuming the most input.**
3. **If there are ties, then the pattern specified earliest to LEX wins.**

The notation used above is *regular expression* notation, which allows for choice, catenation, and repeats. One can show by construction that any language accepted by a finite-state automaton has an equivalent regular expression.

A comment

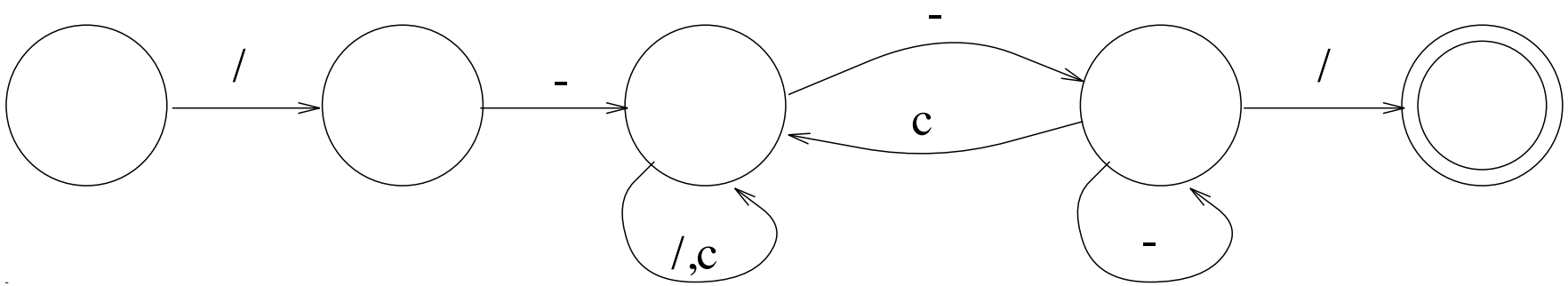
An interesting example is the C-like *comment* specification, which might be tempting to specify as:

```
"/-"  .*  "-/"
```

But in a longest match, this pattern will match the beginning of the first comment to the end of the last comment, and everything in between. If LEX's buffers don't overflow, most of the input program will be ignored by this faulty specification.

A better specification can be determined as follows:

1. **Start with the wrong specification.**
2. **Construct the associated deterministic FSA.**
3. **Edit the FSA to cause acceptance at the end of the first comment (shown below).**
4. **Construct the regular expression associated with the resulting FSA.**



with the corresponding regular expression

```
/- [ (/|c)* -(-)* c ]* (/|c)* -(-)* /
```

Teaching regular languages and scanners

Classroom

1. **Motivate the study with examples from programming languages and puzzles (THINK-A-DOT, etc.).**
2. **Present deterministic FSA (DFA).**
3. **Present nondeterministic FSA (NFA).**
4. **Show how to construct NFAs from regular expressions.**
5. **Show good use of the empty string (λ or ϵ).**
6. **Eliminate the empty string.**
7. **Eliminate nondeterminism.**
8. **Minimize any DFA.**
9. **Construction of regular expressions from DFA.**
10. **Show the correspondence between regular grammars and FSAs.**
11. **The pumping lemma and nonregular languages.**

Projects and Homework

1. **Implement THINK-A-DOT.**
2. **Check if a YACC grammar is regular. If so, then emit the GOTO table for a finite-state driver.**
3. **Augment the above with ACTION clauses.**
4. **Process a YACC file for reserved keyword specifications:**

   ```
   %token <rk> then
   ```

   and generate the appropriate pattern and action for recognizing these:

   ```
   "then" { return(THEN); }
   ```
5. **Show that regular expression notation is itself not regular.**

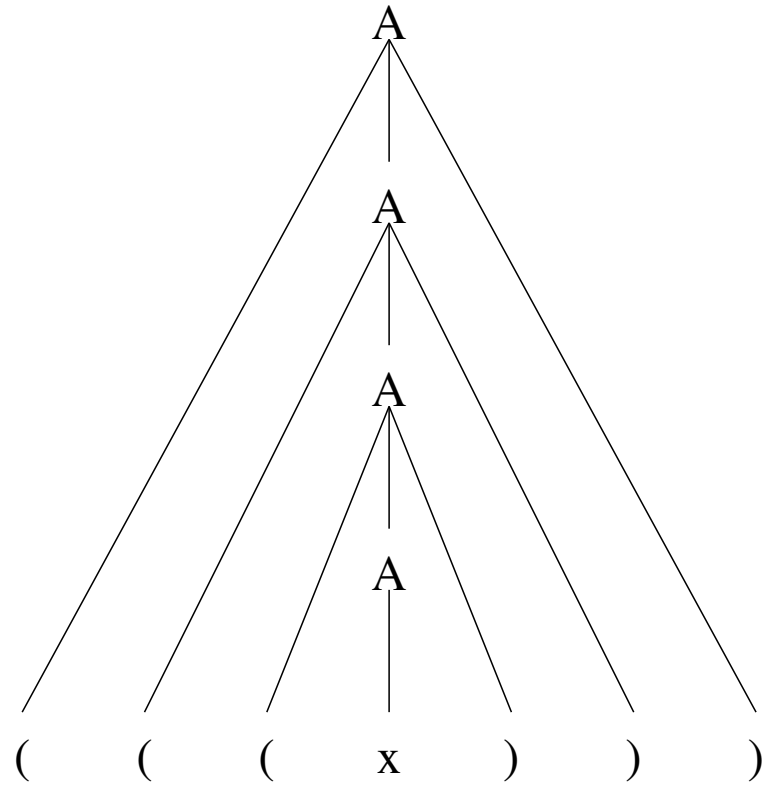
Some useful resources: (24, 28, 16, 2, 26).

Nonregular languages

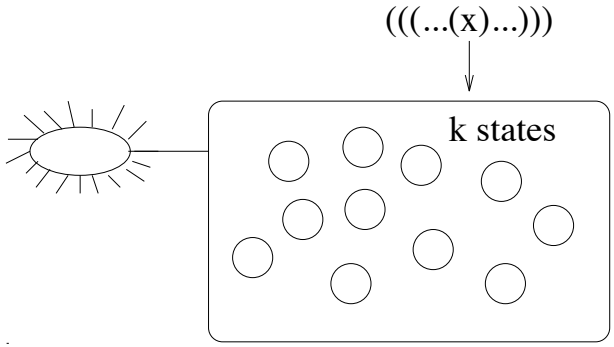
To grow beyond regular languages, we now allow a rule's right-hand side to contain any string of terminals or nonterminals.

A → (A)
| x

describes the language $({}^{n}x)^{n}$.



Suppose that some finite-state machine M of k states can recognize $\{ ({}^{n}x)^{n} \}$.



Consider the input string $z = ({}^{k}x)^{k}$. After processing the k^{th} '(', some state must have been visited twice. By repeating the portion of z causing this loop, we obtain a string

$$({}^{k}({}^{j}x)^{k}, k \geq 0, j > 0$$

which is not in the language, but is accepted by M.

Since the proof did not depend on any particular k, we have shown that no finite-state machine can accept exactly this language.