PART I: LEXICAL ANALYSIS

- The first phase of compilation.
- Takes raw input, which is a stream of characters, and converts it into a stream of *tokens,* which are logical units, each representing one or more characters that "belong together."

Typically,

1. Each keyword is a token, e.g, `then`, `begin`, `integer`.
2. Each identifier is a token, e.g., `a`, `zap`.
3. Each constant is a token, e.g., `123`, `123.45`, `1.2E3`.
4. Each sign is a token, e.g., `(`, `<`, `<=`, `+`.

Example:

```
if foo = bar then ···
```

consists of tokens:

IF, <ID,1>, EQSIGN, <ID,2>, THEN ···

Note that we have to distinguish among different occurrences of the token ID.

- LA is usually called by the parser as a subroutine, each time a new token is needed.

Approaches to Building Lexical Analyzers

The lexical analyzer is the only phase that processes input character by character, so speed is critical.

Either

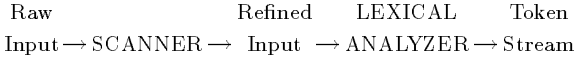
1. Write it yourself; control your own input buffering, or
2. Use a tool that takes specifications of tokens, often in the regular expression notation, and produces for you a table-driven LA.

(1) can be 2–3 times faster than (2), but (2) requires far less effort and is easier to modify.

Hand-Written LA

Often composed of a cascade of two processes, connected by a buffer:

1. *Scanning:* The raw input is processed in certain character-by-character ways, such as
   a) Remove comments.
   b) Replace strings of tabs, blanks, and newlines by single blanks.
2. *Lexical Analysis:* Group the stream of refined input characters into tokens.



The Token Output Stream

Before passing the tokens further on in the compiler chain, it is useful to represent tokens as pairs, consisting of a

1. *Token class* (just "token" when there is no ambiguity), and a
2. Token value.

The reason is that the parser will normally use only the token class, while later phases of the compiler, such as the code generator, will need more information, which is found in the token value.

Example:

Group all identifiers into one token class, <identifier>, and let the value associated with the identifier be a pointer to the symbol table entry for that identifier.

(<identifier>, <ptr to symbol table>)

Likewise, constants can be grouped into token class <constant>:

(<constant>, <ptr to value>)

Keywords should form token classes by themselves, since each plays a unique syntactic role. Value can be null, since keywords play no role beyond the syntax analysis phase, e.g.:

(<then>, –)

Operators may form classes by themselves. Or, group operators with the same syntactic role and use the token value to distinguish among them.

For example, + and (binary) − are usually operators with the same precedence, so we might have a token class <additive operator>, and use value = 0 for + and value = 1 for −.

State-Oriented Lexical Analyzers

The typical LA has many *states,* or points in the code representing some knowledge about the part of the token seen so far.

Even when scanning, there may be several states, e.g., "regular" and "inside a comment." However, the token gathering phase is distinguished by very large numbers of states, perhaps several hundred.

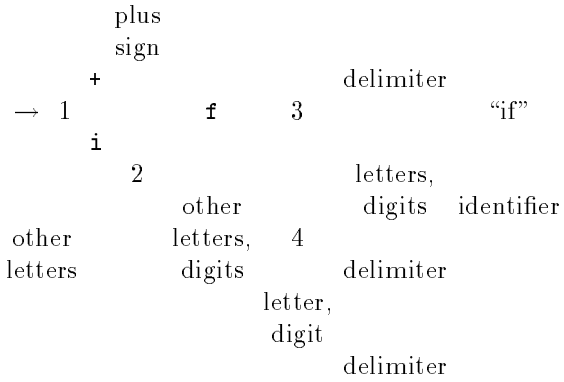
Keywords in Symbol Table

To reduce the number of states, enter keywords into the symbol table as if they were identifiers.

When the LA consults the symbol table to find the correct lexical value to return, it discovers that this "identifier" is really a keyword, and the symbol table entry has the proper token code to return.

Example:

A tiny LA has the tokens "+", "if", and <identifier>. The states are suggested by:



For example, state 1, the initial state, has code that checks for input character `+` or `i`, going to states "plus sign" and 2, respectively.

On any other letter, state 1 goes to state 4, and any other character is an error, indicated by the absence of any transition.

Table-Driven Lexical Analyzers

Represent decisions made by the lexical analyzer in a two-dimensional ACTION table. Row = state; column = current input seen by the lookahead pointer.

The LA itself is then a driver that records a state i, looks at the current input, X, and examines the array ACTION$[i, X]$ to see what to do next.

The entries in the table are actions to take, e.g.,

1. Advance input and go to state j.
2. Go to state j with no input advance. (Only needed if state transitions are "compressed" — see next slide.)
3. The token "if" has been recognized and ends one position past the beginning-of-token marker. Advance the beginning-of-token marker to the position following the `f` and go to the start state. Wait for a request for another token.
4. Error. Report what possible tokens were expected. Attempt to recover the error by moving the beginning-of-token pointer to the next sign (e.g., +, <) or past the next blank. This change may cause the parser to think there is a deletion error of one or more tokens, but the parser's error recovery mechanism can handle such problems.

Example:

The following suggests the table for our simple state diagram above.

State	`i`	`f`	`+`	`a`
1	2*	4*	A*	4*
2	4*	3*	B	4*
3	4*	4*	C	4*
4	4*	4*	B	4*

Key:

* = Advance input
A = Recognize token "plus sign"
B = Recognize token "identifier"
C = Recognize token "if"

- `+` stands for itself, and also represents a typical delimiter.
- `a` represents a typical "other letter or digit."

Compaction of the table is a possibility, since often, most of the entries for a state are the same.

Regular Expressions

- Useful notation for describing the token classes.
- ACTION tables can be generated from them directly, although the construction is somewhat complicated: see the "Dragon book."

Operands:

1. Single characters, e.g, `+`, `A`.
2. Character classes, e.g.,

   letter = `[A-Za-z]`
   digit = `[0-9]`
   addop = `[+-]`

- Expressions built from operands and operators denote sets of strings.

Operators

1. *Concatenation.* $(R)(S)$ denotes the set of all strings formed by taking a string in the set denoted by R and following it by a string in the set denoted by S. For example, the keyword "then" can be expressed as a regular expression by `t h e n`.
2. The operator + stands for union or "or." $(R) + (S)$ denotes the set of strings that are in the set denoted by S union those in the set denoted by R. For example, *letter*+*digit* denotes any character that is either a letter or a digit.
3. The postfix unary operator *, the *Kleene closure*, indicates "any number of," including zero. Thus, (*letter*)* denotes the set of strings consisting of zero or more letters.

- The usual order of precedence is * highest, then concatenation, then +. Thus, $a + bc^*$ is

  $a + (b(c^*))$

Shorthands

4. R^+ stands for "one or more occurrences of," i.e., $R^+ = RR^*$. Do not confuse infix +, meaning "or," with the superscript +.
5. $R?$ stands for "an optional R," i.e., $R? = R + \epsilon$.

- Note ϵ stands for the *empty string*, which is the string of length 0.

Some Examples:

$identifier = letter\,(letter + digit)^*$
$float = digit^+\ .\ digit^* + .\ digit^+$

Finite Automata

- A collection of states and rules for transferring among states in response to inputs.
- They look like state diagrams, and they can be represented by ACTION tables.

We can convert any collection of regular expressions to a FA; the details are in the text. Intuitively, we begin by writing the "or" of the regular expressions for each of our tokens.

Example:

In our simple example of plus signs, identifiers, and "if," we would write:

$$\texttt{+} + \texttt{i}\ \texttt{f} + letter\,(letter + digit)^*$$

States correspond to *sets of* positions in this expression. For example,

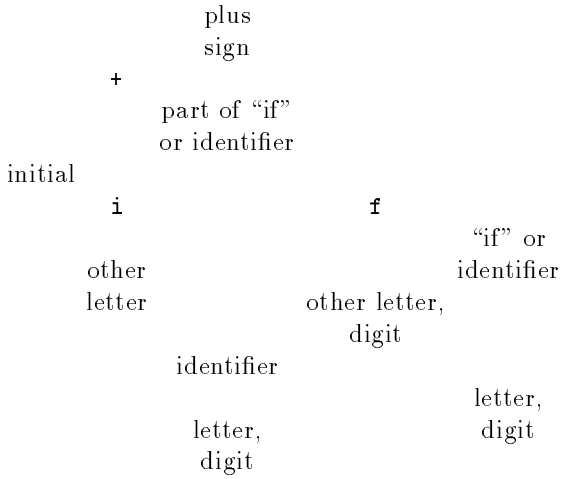
- The "initial state" corresponds to the positions where we are ready to begin recognizing any token, i.e., just before the `+`, just before the `i`, and just before the first *letter*.
- If we see `i`, we go to a state consisting of the positions where we are ready to see the `f` or the second *letter*, or the *digit*, or we have completed the token.

Positions following the expression for one of the tokens are special; they indicate that the token has been seen.

However, just because a token has been seen does not mean that the lexical analyzer is done. We could see a longer instance of the same token (e.g., identifier) or we could see a longer string that is an instance of another token.

Example

The FA from the above RE is:



- Double circles indicate states that are the end of a token.

Note the similarities and differences between this FA and the hand-designed state diagram introduced earlier.

The structures are almost the same, but the FA does not show explicit transitions on "delimiters." In effect, the FA takes any character that does not extend a token as a "delimiter."

Using the Finite Automaton

To disambiguate situations where more than one candidate token exists:

1. Prefer longer strings to shorter. Thus, we do not take `a` or `ab` to be the token when confronted with identifier `abc`.
2. Let the designer of the lexical analyzer pick an order for the tokens. If the longest possible string matches two or more tokens, pick the first in the order. Thus, we might list keyword tokens before the token "identifier," and confronted with string

   `i f +` $\cdots$

   would determine that we had the keyword "if," not an identifier `if`.

We use these rules in the following algorithm for simulating a FA.

1. Starting from the initial state, scan the input, determining the state after each character. Record the state for each input position.
2. Eventually, we reach a point where there is no next state for the given character. Scan the states backward, until the first time we come to a state that includes a token end.
3. If there is only one token end in that state, the lexical analyzer returns that token. If more than one, return the token of highest priority.

Example:

The above FA, given input

```
i f a +
```

would enter the following sequence of states:

	`i`	`f`	`a`	`+`
initial	part of "if" or identifier	"if" or identifier	identifier	none

The last state representing a token end is after 'a', so we correctly determine that the identifier *ifa* has been seen.

Example Where Backtracking is Necessary

In Pascal, when we see `1.`, we cannot tell whether `1` is the token (if part of `[1..10]`) or whether the token is a real number (if part of `1.23`).

Thus, when we see `1..`, we must move the token-begin marker to the first dot.

PART II: INTRODUCTION TO PARSING

Parsing =

- Check input is well-formed.
- Build a parse tree or similar representation of input.

Recursive Descent Parsing

- A collection of recursive routines, each corresponding to one of the nonterminals of a grammar.
- Procedure A, corresponding to nonterminal A, scans the input and recognizes a string derived from A. If recognized, the appropriate semantic actions are taken, e.g., a node in a parse tree corresponding to A is created.
- In its most general form, recursive descent involves backtracking on the input.

LL-Parsing

- Specialization of recursive descent that requires no backtracking.
- Called LL, because we discover a leftmost derivation while scanning the input from left to right.
- A *leftmost derivation* is one where at each step, the leftmost nonterminal is replaced.

Example:

$$A \Rightarrow BC \Rightarrow DEC \Rightarrow aEC$$

Is leftmost;

$$A \Rightarrow BC \Rightarrow BFG$$

is not.

- A *left-sentential form* is any string of terminals and nonterminals derived by some leftmost derivation.

LL Parser Structure

- Input, a string of terminals, is scanned left-to-right.
- A stack holds the *tail* of the current LSF, which is the portion of the LSF from the leftmost nonterminal to the right.
- The *head* of the LSF, or prefix of terminals up to the first nonterminal, will always match the portion of the input that has been scanned so far.

Parser Actions

1. Match the top-of-stack, which must be a terminal, with the current input symbol. Pop the stack, and move the input pointer right.
2. Replace the top-of-stack, which must be a nonterminal A, by the right side of some production for A.

- The magic of LL parsing: knowing which production to use in (2).

Example:

Consider the following grammar for lists.

E = "element" (atom or bracketed list), the start symbol.

L = "list," one or more elements, separated by commas.

T = "tail," empty or comma-list.

$$\begin{aligned} E &\rightarrow [L] \mid a \\ L &\rightarrow ET \\ T &\rightarrow ,L \mid \epsilon \end{aligned}$$

Beginning of an LL parse for string $[a,[a,a]]$ is:

Left S. F.	Stack	Remaining Input
E	E	$[a,[a,a]]$
$[L]$	$[L]$	$[a,[a,a]]$
$[L]$	$L]$	$a,[a,a]]$
$[ET]$	$ET]$	$a,[a,a]]$
$[aT]$	$aT]$	$a,[a,a]]$
$[aT]$	$T]$	$,[a,a]]$
$[a,L]$	$,L]$	$,[a,a]]$

- Build parse tree by expanding nodes corresponding to the nonterminals exactly as the nonterminals themselves are expanded.

FIRST

To define "LL grammar" precisely, we need two definitions. The first is "FIRST".

If α is any string, FIRST(α) is the set of terminals b that can be the first symbol in some derivation $\alpha\$ \Rightarrow^* b\beta$.

- Note that if $\alpha \Rightarrow^* \epsilon$, then b could be \$, which is an "honorary" terminal.

FOLLOW

FOLLOW(A) is the set of terminals that can immediately follow nonterminal A in any derivation (not necessarily leftmost) from $S\$$, where S is the start symbol.

- Algorithms for computing FIRST and FOLLOW are found in the "Dragon book."

Definition of LL Grammar

We must decide how to expand nonterminal A at the top of the stack, knowing only the next input symbol, b (the "lookahead symbol"). Production $A \rightarrow \alpha$ is a choice if

1. b is in FIRST(α), or
2. $\alpha \Rightarrow^* \epsilon$, and b is in FOLLOW(A).

Otherwise, α, followed by whatever is below A on the stack, could not possibly derive a string beginning with b.

- A grammar is LL(1) [or just "LL"] if and only if, for each A and b, there is at most one choice according to (1) and (2) above.
- A table giving the choice for each nonterminal A (row) and lookahead symbol b (column) is an *LL parsing table*.

Example:

The list grammar,

$$\begin{aligned} E &\rightarrow [L] \mid a \\ L &\rightarrow ET \\ T &\rightarrow ,L \mid \epsilon \end{aligned}$$

is LL. Here is the parsing table.

	a	[	]	,	\$
E	$E \rightarrow a$	$E \rightarrow [L]$			
L	$L \rightarrow ET$	$L \rightarrow ET$			
T			$T \rightarrow \epsilon$	$T \rightarrow ,L$	

For example, on lookahead "," we can only expand T by $,L$. That production is a choice, because "," is in FIRST($,L$). The other production, $T \rightarrow \epsilon$, is not a choice, because "," is not in FOLLOW(T).

On the other hand, "]" is in FOLLOW(T), so $T \rightarrow \epsilon$ is a choice on this lookahead.

- Note that on lookaheads a, \$, and "[" there is no choice of expansion for T. Such a situation represents an error; the input cannot be derived from the start symbol.

Example of Non-LL Construct

Consider the following grammar fragment, which represents the "dangling else."

(1) <stat> → **if** <cond> **then**
<stat> <else-clause>
(2) <else-clause> → **else** <stat>
(3) <else-clause> → ϵ

How do we expand <else-clause> on lookahead **else**?

- Surely, **else** is in FIRST(**else** <stat>), so production (2) is a choice.
- However, (3) is also a choice, because **else** is in FOLLOW(<else-clause>), as seen by a parse tree of the following form.

<stat>

if <cond> **then** <stat> <else-clause>

if <cond> **then** <stat> <else-clause> **else** <stat>

ϵ

- Normally, we pick (2), even though there is a technical LL violation. This choice corresponds to the idea "an **else** matches the previous unmatched **then**."

Designing LL Grammars

There are several tricks that help us put grammars into an LL form.

1. Eliminating left-recursion.
2. Left-factoring.

Eliminating Left-Recursion

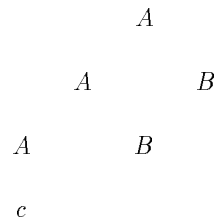
A grammar with *left-recursion*, a nontrivial derivation of the form $A \Rightarrow^* A\alpha$, cannot possibly be LL.

Example

Consider

$$A \rightarrow AB \mid c$$

A typical derivation from A looks like:



Seeing only lookahead c, we have no clue as to how many times we should apply $A \rightarrow AB$ before applying $A \rightarrow c$.

The "Dragon book" gives a general algorithm for eliminating left-recursion. The following simple rule applies in most cases. Replace the productions

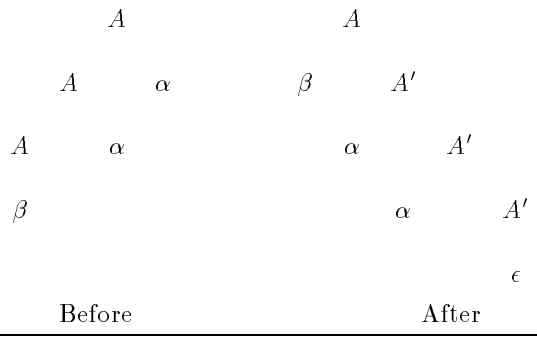
$$A \rightarrow A\alpha \mid \beta$$

by

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

The idea is that every leftmost derivation from A eventually uses $A \rightarrow \beta$, so in the new productions, β is generated first, followed by zero or more occurrences of α. The new nonterminal A' takes care of the α's.

The implied transformation on parse trees is suggested by the following diagram; $\beta\alpha\alpha$ is derived in each case.



Example:

A more natural grammar for lists is

$$E \rightarrow [L] \mid a$$
$$L \rightarrow L, E \mid E$$

We may replace the left-recursion for L by

$$E \rightarrow [L] \mid a$$
$$L \rightarrow EL'$$
$$L' \rightarrow ,EL' \mid \epsilon$$

- Note that L' plays the role of T in our earlier example. However, instead of $L' \rightarrow ,L$, the L has been expanded by its lone production.

Left-Factoring

Sometimes, two productions for a nonterminal have right sides with a common prefix. For example, the following is also a natural grammar for lists.

$$E \rightarrow [L] \mid a$$
$$L \rightarrow E, L \mid E$$

If L is at the top of stack, and we see a lookahead symbol, like a, that is in FIRST(E), then we cannot tell whether to expand L by E, L or just E.

The trick is to "factor" the E out, and defer the decision until we have examined the portion of the input E derives. New grammar:

$$E \rightarrow [L] \mid a$$
$$L \rightarrow EL'$$
$$L' \rightarrow , L \mid \epsilon$$

- This grammar is exactly our original, with L' in place of T.

General Rule

Replace $A \rightarrow \alpha\beta$ and $A \rightarrow \alpha\gamma$ by

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta \mid \gamma$$