Lexical Analysis / Scanning

Purpose: turn **character** stream (input program) into **token** stream

- groups characters into tokens
- ignoring whitespace
- associate line number in program & error message
- handling I/O, machine dependencies

Token: group of characters forming basic, atomic chunk of syntax

- identifiers
- operators
- keywords
- constants

Whitespace:
characters between tokens that are ignored

Separate Lexical and Syntactic Analysis

Separation of function

- scanner:
  - handle grouping chars into tokens
- parser:
  - handle grouping tokens into syntax trees

Advantages:

- simpler design
- faster scanning
  - scanning is time-consuming in many compilers
- can build lexical analyzer & parser generators
- scanner a subroutine of parser
  - "get the next token"

Wide applicability of lexical analysis

Pattern matching: match input string to specified patterns

- query language for a database
- configuration parameters for a cache simulator
- silicon compiler
- editing language
-
-

Lexemes, tokens, and patterns

Lexeme: group of characters that form a token

Token: set of lexemes that match a pattern

Pattern: description of string of characters
rules that describes a set of lexemes that represent a particular token

Token may have attributes, if more than one lexeme in token

Regular expressions

Notation for specifying patterns of lexemes in a token

Regular expressions:

- powerful enough to do this
- simple enough to be implemented efficiently
- precise
- equivalent in power to finite state machines

Syntax of regular expressions

REs built out of simpler REs according to rules

Defined inductively

- base cases:
  the empty string (ε)
  a symbol from the alphabet (x)
- inductive cases:
  concatenation: sequence of two RE's: E_1E_2
  union: either of two RE's: $E_1 \mid E_2$
  Kleene closure: zero or more occurrences of a RE: E^*

Notes:

- precedence: * highest, concatenation, | lowest
- can use parens for grouping
- whitespace insignificant

Notational conveniences

E^+ means 1 or more occurrences of E

E^k means k occurrences of E

`[`E`]` means 0 or 1 occurrence of E (optional E)

`{`E`}` means E^*

`not``(`x`)` means any character in the alphabet but x

`not``(`E`)` means any string of characters in the alphabet but those matching E

E_1-E_2 means any string matching E_1 except those matching E_2

`[ab]` means `a | b`

`[a-z]` means `a | b | ... | z`

Naming regular expressions

Can assign names to regular expressions

Can use the name of a RE in the definition of another RE

Examples:

```
letter      ::= a | b | ... | z
digit       ::= 0 | 1 | ... | 9
alphanum    ::= letter | digit
```

BNF-like notation for RE's

Can reduce named RE's to plain RE by "macro expansion"

- no recursive definitions allowed

Using regular expressions to specify tokens

Identifiers

```
ident       ::= letter (letter | digit)*
```

Integer constants

```
integer     ::= digit+
sign        ::= + | -
signed_int  ::= [sign] integer
```

Real number constants

```
real        ::= signed_int
                [fraction] [exponent]
fraction    ::= . digit+
exponent    ::= (E|e) signed_int
```


String and character constants

```
string      ::= " char* "
character   ::= ' char '

char        ::= not("|'|\) | escape
escape      ::= \("|'|\|n|r|t|v|b|a)
```

Whitespace (not a token)

```
whitespace  ::= <space> | <tab> | <newline>
                | comment
comment     ::= /* not(*/) */
```


Regular expressions for PL/0 lexical structure

```
Program  ::= (Token | White)*

Token    ::= Id | Integer | Keyword | Operator |
             Punct
Punct    ::= ; | : | . | , | ( | )
Keyword  ::= module | procedure | begin | end |
             const | var | if | then | while |
             do | input | output | odd | int
Operator ::= := | * | / | + | - |
             = | <> | <= | < | >= | >
Integer  ::= Digit+
Id       ::= Letter AlphaNum*
AlphaNum ::= Letter | Digit
Digit    ::= 0 | ... | 9
Letter   ::= a | ... | z | A | ... | Z

White    ::= <space> | <tab> | <newline>
```


Building scanners from RE patterns: the big picture

Specify patterns with
regular expressions

Convert RE specification into
nondeterministic finite state machine

Convert nondeterministic finite state machine into a
deterministic finite state machine

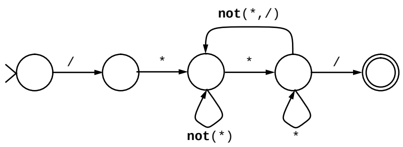
Convert deterministic finite state machine into
scanner implementation

- a collection of procedures
- table-driven scanner

Finite State Machines/Automata

A FSA has:

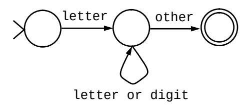
- a set of states
  - one marked the initial state
  - some marked final states
- a set of transitions from state to state
  - each transition labelled with a symbol from the alphabet or ε



  - Operate by reading symbols and taking transitions, beginning with the start state
- if no transition with a matching label is found, reject

If reach the final state, accept; otherwise reject

Identifiers & keywords

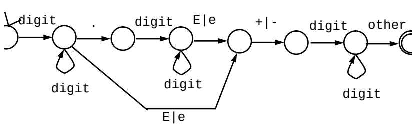


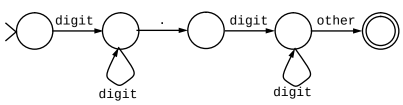
FSM would be complicated if included keywords

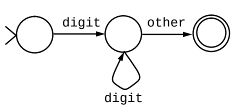
Solution:

- put keywords in the symbol table
- lookup after reach accept state for identifiers

Unsigned numbers







Determinism

FSA can be **deterministic** or **nondeterministic**

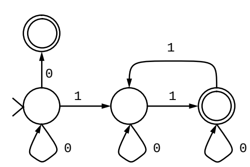
Deterministic: always know which way to go

- at most 1 arc leaving a state with particular symbol
- no ε arcs

Nondeterministic: may need to explore multiple paths

- multiple arcs leaving a state with the same symbol
- ε is a legal arc
- accepts input string if **some** path leads to final state

Example:



Comparing complexity of NFA and DFA

RE's map to NFA's easily

Can write code from DFA easily

Converting DFAs to code

Option 1: implement scanner using procedures
- one procedure for each token (FSM diagram)
- each procedure reads characters until failure
- choices implemented using case statements

Pros
- straightforward to write by hand
- fast

Cons (if written by hand)
- more work than using a tool
- sometimes hard to interpret the REs correctly

Pl0 scanner

Parser calls scanner
- input file already open

Constructor does initialization (*Scanner*)

Scanner repeatedly scans characters & returns tokens
(*Get()* in *ScanProgram()*)
- scans over whitespace (*SkipWhiteSpace()*)
- identifiers & keywords (*GetIdent()*)
  - string compare in binary search
- integers (*GetInt()*)
  - converts to decimal
- operators & punctuation (*GetPunc()*)
  - conditional reads
  - case statements

Converting DFAs to code

Option 2: implement table-driven scanner
- rows: states of DFA
- columns: input characters
- entries: action
  - go to new state
  - accept token, go to start state
  - error
- actions written in code
- interpreter for the table

Pros
- convenient for automatic generation (e.g. `lex`)

Cons
- table lookups slower than direct code

Automatic construction of scanners

Approach:

1) Convert RE into NFA
2) Convert NFA into DFA
3) Convert DFA into table-driven code

RE -> NFA (via Thompson's algorithm)

(1) Expand RE into basic symbols

(2) Construct NFAs for the symbols

(3) Combine NFAs inductively
- 1 start state, 1 final state

RE ⇒ NFA

Define by cases

ε

x

$E_1\ E_2$

$E_1 \mid E_2$

E^*

NFA $\Rightarrow$ DFA

Problem: NFA can "choose" among alternative paths,
while DFA must have only one path

Solution: **subset construction** of DFA

- each state in DFA represents *set of states in NFA* that can be reached by a given input symbol

Subset construction algorithm

Given NFA with states and transitions

- label all NFA states uniquely

Create start state of DFA

- label it with the set of NFA states that can be reached by ε transitions (i.e. without consuming any input)

Process the start state

To process a DFA state *S* with label *{s1,..,sN}*:

For each input symbol *x*:

- compute the set *T* of NFA states reached from any of the NFA states *s1,..,sN* by a *x* transition followed by any number of ε transitions
- if *T* not empty:
  - if *T* is already in the DFA, add a transition labeled *s* from *S* to *T*
  - otherwise create a new DFA state labeled *T*, add a transition labeled *x* from *S* to *T*, and process *T*

A DFA state is final iff
at least one of the NFA states in its label is final