Lexical Analysis / Scanning

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

- parser turns token stream into syntax tree

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

Whitespace:
characters between tokens that are ignored

Separate lexical and syntax analysis

Separation of concerns / good design

- scanner:
  - handle grouping chars into tokens
  - ignoring whitespace
  - handling I/O, machine dependencies
- parser:
  - handle grouping tokens into syntax trees

Restricted nature of scanning allows faster implementation

- scanning is time-consuming in many compilers

Language design issues

Most languages today are “free-form”

- layout doesn’t matter
- use whitespace to separate tokens, if needed

Alternatives:

- Fortran, Algol 68: whitespace ignored
- Haskell: use layout to imply grouping

Most languages today have “reserved words”

- can’t be used as identifiers

Alternative: PL/I: have “keywords”

- keywords treated specially only in certain contexts, otherwise just idents

Most languages separate scanning and parsing
Alternative: C++: ***type*** vs. ***ident***

- parser wants scanner to distinguish types from vars
- scanner doesn’t know how things declared

Lexemes, tokens, and patterns

Lexeme: group of characters that form a token

Token: class of lexemes that match a pattern

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

Set of tokens in programming language specified using **regular expressions**

- REs are simplest language class that’s powerful enough

Languages

Alphabet: a finite set of characters/symbols

String: a finite, possibly empty sequence of characters in alphabet

Language: a (possibly empty or infinite) set of strings

Language specifications

Grammar: a finite specification of a set of strings

Language automaton:
a finite machine for accepting a set of strings and rejecting all others

A language can be specified by many different grammars and automata

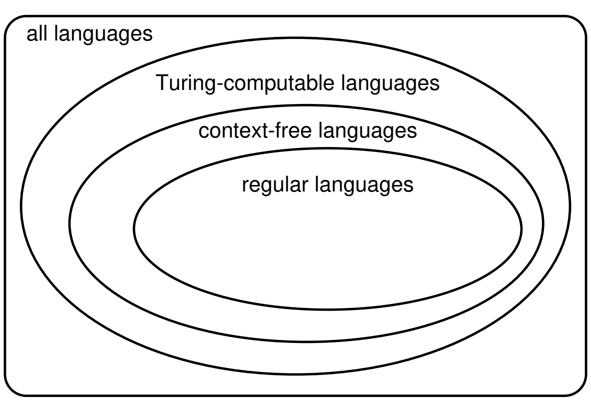
A grammar or automaton specifies only one language

Classes of languages

Regular languages can be specified by
regular expressions/grammars, finite-state automata (FSAs)

Context-free languages can be specified by
context-free grammars, push-down automata (PDAs)

Turing-computable languages can be specified by
general grammars, Turing machines



Syntax of regular expressions

Defined inductively

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

Notes:

- can use parens for grouping
- precedence: * highest, sequence, $|$ lowest
- 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

No additional expressive power

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
```

EBNF-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

```
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>
```

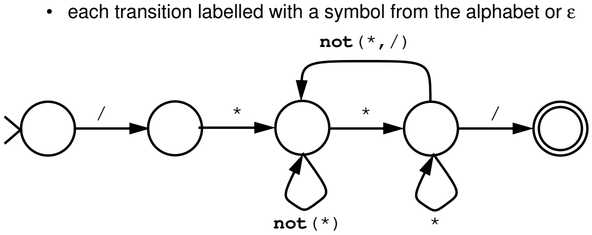

Building scanners from RE patterns

Convert RE specification into
deterministic finite state machine
- by eye
- mechanically by way of
  nondeterministic finite state machine

Convert DFA into scanner implementation
- by hand into collection of procedures
- mechanically into table-driven scanner

Finite State Machines/Automata

An 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

When done with input, accept if in final state, reject otherwise

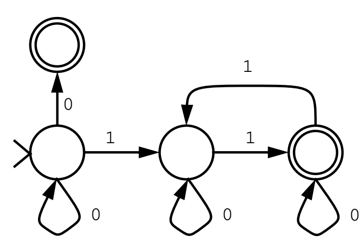
Determinism

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

Example:



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
- each procedure reads characters
- choices implemented using case statements

Pros

- straightforward to write by hand
- fast

Cons

- more work than using a tool
- may have subtle differences from spec

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

Pros

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

Cons

- table lookups slower than direct code
- awkward to build by hand

Automatic construction of scanners

Approach:

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

RE ⇒ NFA

Define by cases

ε

x

$E_1 \; E_2$

$E_1 \mid E_2$

E^{*}

NFA ⇒ 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*, all that the NFA might be in during its traversal

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 symbol *s* in the alphabet:

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