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;
a "word"

Whitespace:
characters between tokens that are ignored

Why separate lexical from syntactic analysis?

Separation of concerns / good design

- scanner:
  - handle grouping chars into tokens
  - ignore whitespace
  - handle 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

Complications

Most languages today are "free-form"

- layout doesn't matter
- whitespace separates tokens

Alternatives:

- Fortran: line-oriented, whitespace doesn't separate

```
        do 10 i = 1.100
          .. a loop ..
     10 continue
```

- Haskell: can use identation & layout to imply grouping

Most languages separate scanning and parsing
Alternative: C/C++/Java: ***type*** vs. ***identifier***

- parser wants scanner to distinguish names that are types from names that are variables
- but scanner doesn't know how things declared -- that's done during semantic analysis a.k.a. typechecking!

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

Pattern: typically defined using a **regular expression**

- REs are simplest language class that's powerful enough

Languages and language specifications

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

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

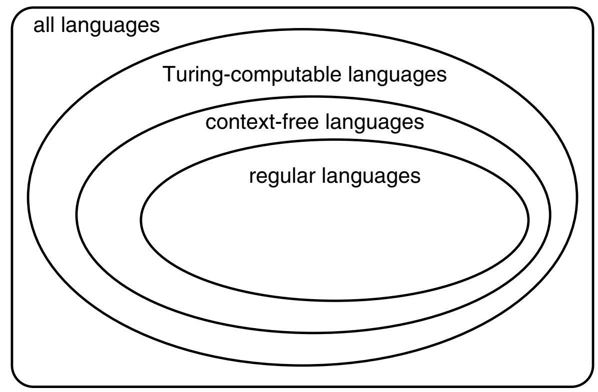
A grammar or automaton specifies only one language

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 or ∈)
  - a symbol from the alphabet (e.g. **x**)
- inductive cases:
  - sequence of two RE's: E_1E_2
  - either of two RE's: $E_1 | E_2$
  - Kleene closure (zero or more occurrences) of a RE: E^*

Notes:
- can use parentheses 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 strings matching E

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

No additional expressive power through these conveniences

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
```

Grammar-like notation for named RE's: a regular grammar

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

- no recursive definitions allowed,
  unlike full context-free grammars

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
```


More token specifications

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(*/) */
```


Meta-rules

Can define a rule that a legal program is a sequence of tokens and whitespace

```
program ::= (token|whitespace)*
token ::= ident | integer | real | string | ...
```

But this doesn't say how to uniquely break up an input program into tokens -- it's highly ambiguous!

E.g. what tokens to make out of `hi2bob`?

- one identifier, `hi2bob`?
- three tokens, `hi 2 bob`?
- six tokens, each one character long?

The grammar states that it's legal, but not how tokens should be carved up from it

Apply extra rules to say how to break up string into sequence of tokens

- longest match wins
- yield tokens, drop whitespace

RE specification of initial MiniJava lexical structure

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

Token       ::= ID | Integer | ReservedWord |
                Operator | Delimiter
ID          ::= Letter (Letter | Digit)*
Letter      ::= a | ... | z | A | ... | Z
Digit       ::= 0 | ... | 9
Integer     ::= Digit+
ReservedWord::= class | public | static |
                extends | void | int |
                boolean | if | else |
                while | return | true | false |
                this | new | String | main |
                System.out.println
Operator    ::= + | - | * | / | < | <= | >= |
                > | == | != | && | !
Delimiter   ::= ; | . | , | = |
                ( | ) | { | } | [ | ]

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


Building scanners from RE patterns

Convert RE specification into **finite state automaton (FSA)**

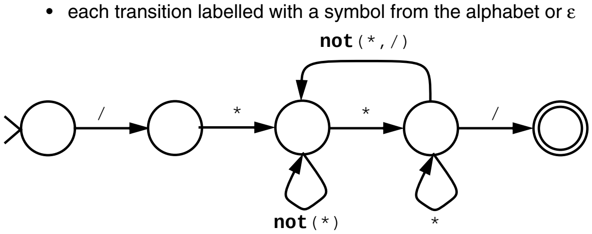
Convert FSA into scanner implementation

- by hand into collection of procedures
- mechanically into table-driven scanner

Finite state 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

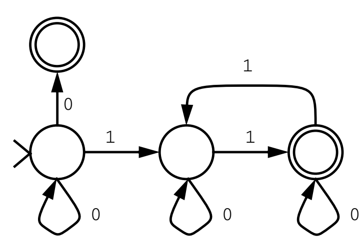
FSA can be **deterministic** or **nondeterministic**

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, only choose right one later

Example:



NFAs vs. DFAs

A problem:

- RE's (e.g. specifications) map to NFA's easily
- Can write code from DFA easily

How to bridge the gap?
Can it be bridged?

A solution

Cool algorithm to translate any NFA into equivalent DFA!

- proves that NFAs aren't more expressive than DFAs

Plan:

1) Convert RE into NFA [they're equivalent]
2) Convert NFA into DFA
3) Convert DFA into code

Can be done by hand, or fully automatically

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 $\{s_1,..,s_N\}$:

For each symbol *x* in the alphabet:

- compute the set *T* of NFA states reached from any of the NFA states $s_1,..,s_N$ by an *x* 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 *x* 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

DFA ⇒ code

Option 1: implement scanner by hand using procedures

- one procedure for each token
- each procedure reads characters
- choices implemented using if & switch statements

Pros

- straightforward to write by hand
- fast

Cons

- a fair amount of tedious work
- may have subtle differences from language specification

DFA ⇒ code (cont.)

Option 2: use tool to generate 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
- exactly matches specification, if tool-generated

Cons

- “magic”
- table lookups may be slower than direct code
  - but switch statements get compiled into table lookups, so....
  - can translate table lookups into switch statements, if beneficial