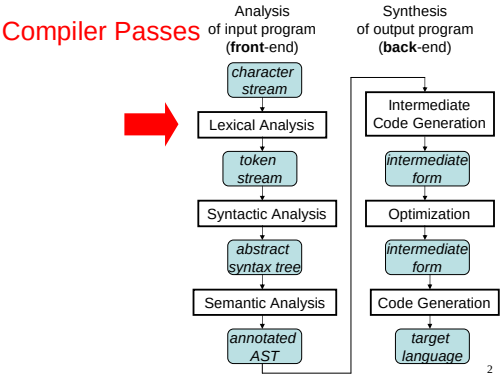
Lexical Analysis

(Part 1)

Lexical analysis is the first phase of compilation: The file is converted from ASCII to tokens. It must be fast!

Compiler Passes



Lexical Pass/Scanning

Purpose: Turn the character stream (program input) into a **token** stream

- *Token*: a group of characters forming a basic, atomic unit of syntax, such as a identifier, number, etc.
- *White space*: characters between tokens that is ignored

Why separate lexical / syntactic analysis

Separation of concerns / good design

- scanner:
  - handle grouping chars into tokens
  - ignore white space
  - 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 to Scanning

- Most languages today are free form
  - Layout doesn't matter
  - White space separates tokens
- Alternatives
  - Fortran -- line oriented
  - Haskell -- indentation and layout can imply grouping
- Separating scanning from parsing is standard
- Alternative: C/C++/Java: ***type*** vs ***identifier***
  - Parser wants scanner to distinguish between names that are types and names that are variables
  - *But* Scanner doesn't know how things are declared … done in semantic analysis, a\k\a type checking!

```
    do 10 i = 1,100
     ...loop code...
10 continue
```


Lexemes, tokens, patterns

Lexeme: group of characters that forms a pattern

Token: class of lexemes matching a pattern

- Token may have attributes if more than one lexeme in a token

Pattern: typically defined using regular expressions

- REs are the simplest language class that's powerful enough for this purpose

Languages and Language Specification

Alphabet: finite set of characters and symbols

String: a finite (possibly empty) sequence of characters from an alphabet

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

Grammar: a finite specification for a set of strings

Language Automaton: an abstract machine accepting a set of strings and rejecting all others

A language can be specified by many different grammars and automata

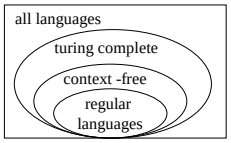
A grammar or automaton specifies a single language

Classes of Languages

Regular languages specified by regular expressions/grammars & finite automata (FSAs) (or just FAs)

Context-free languages specified by context-free grammars and pushdown automata (PDAs)

Turing-computable languages are specified by general grammars and Turing machines



Syntax of Regular Expressions

- Defined inductively
  - Base cases
    - Empty string (ε, ∈)
    - Symbol from the alphabet (e.g. **x**)
  - Inductive cases
    - Concatenation (sequence of two REs) : E_1E_2
    - Alternation (choice of two REs): $E_1 \mid E_2$
    - Kleene closure (0 or more repetitions of RE): E^*
- Notes
  - Use parentheses for grouping
  - Precedence: * is highest, then concatenate, | is lowest
  - White space not significant

Notational Conveniences

- E^+ means 1 or more occurrences of E
- E^k means exactly k occurrences of E
- [E] means 0 or 1 occurrences of E *(i.e., optional)*
- {E} means 0 or more occurrences of E (aka E^*)
- ***not****(x)* means any character in alphabet but x
- ***not****(E)* means any strings from alphabet except those in E
- E_1-E_2 means any string matching E_1 that's not in E_2

There is no additional expressive power through these short cuts

Naming Regular Expressions

Can assign names to regular expressions

Can use the names in regular expressions

Example:

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

Grammar-like notation for regular expression is a regular grammar

Can reduce named REs to plain REs by "macro expansion"

No recursive definitions allowed as in normal context-free grammars

Using REs to Specify Tokens

Identifiers

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

Integer constants

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

Real numbers

```
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)
```

White space

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

13

Meta-Rules

Can define a rule that a legal program is a sequence of tokens and white space:

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

But this doesn't say how to uniquely breakup a program into its 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 to decide

Apply extra rules to say how to break up a string

- Longest sequence wins
- Reserved words take precedence over identifiers

14

RE Specification of initial MiniJava Lex

```
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 ::= ; | . | , | = | ( | ) | { | } | [ | ]
```

15