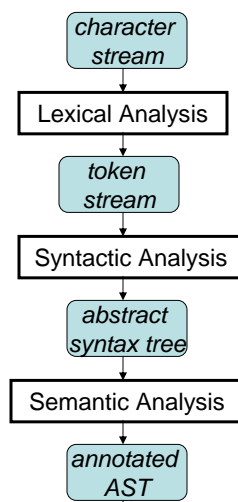


CSE 401 Lexical Analysis

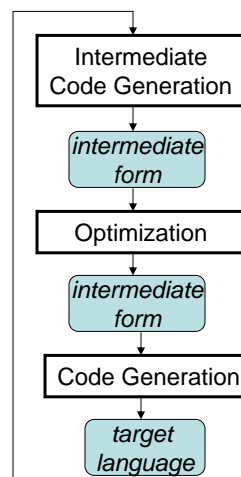
Lexical analysis is the first phase of compilation: The file is converted from ASCII to tokens. It must be fast!

Compiler Passes

Analysis
of input program
(**front-end**)



Synthesis
of output program
(**back-end**)



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 are 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
 - Scanner doesn't know how things are declared ... done in semantic analysis, aka 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 is a particular token

Pattern: typically defined using regular expressions

- REs are the simplest 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

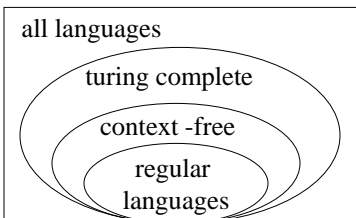
A grammar or automaton specifies a single language

Classes of Languages

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

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 , ϵ)
 - Symbol from the alphabet (e.g. x)
 - Inductive cases
 - Concatenation (sequence of two REs) : E_1E_2
 - Alternation (choice of two REs): $E_1 | E_2$
 - 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
- $\{E\}$ means E^*
- $\text{not}(x)$ means any character in alphabet but x
- $\text{not}(E)$ means any strings from alphabet except those generated by E
- $E_1 - E_2$ means any string matching E_1 that's not in E_2
- There is no additional expressive power here
 - Exercise: Why?

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 | num
```

Grammar-like notation for regular expression is a regular grammar

Can reduce named REs to plain REs by “macro expansion”

No recursive definitions allowed (This is the distinction from full 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 Tokens

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(*/) */
```

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 (Principle of Longest Match)

RE Specification of initial MiniJava Lex

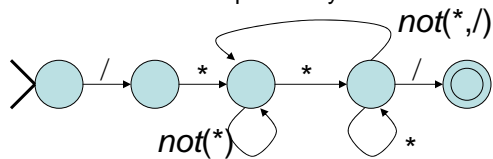
```
Program ::= (Token | Whitespace)*
Token ::= ID | Integer | ReservedWord | Operator |
         Delimiter
ID ::= Letter (Letter | Digit)*
Letter ::= a | ... | z | A | ... | Z
Digit ::= 0 | ... | 9
Integer ::= Digit+
ReservedWord ::= class | public | static | extends |
                void | int | boolean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

Building Scanners with REs

- Convert RE specification into a **finite state automaton (FSA)**
- Convert FSA into a scanner implementation
 - By hand into a collection of procedures
 - Mechanically into a table-driven scanner

Finite State Automata

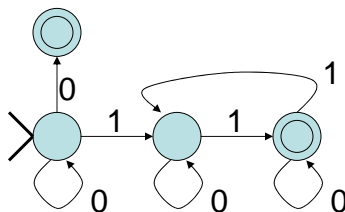
- A Finite State Automaton has
 - A set of states
 - One marked initial
 - Some marked final
 - A set of transitions from state to state
 - Each labeled with an alphabet symbol or ϵ



- Operate by beginning at the start state, reading symbols and making indicated transitions
- When input ends, state must be final or else reject

Determinism

- FSA can be deterministic or nondeterministic
- Deterministic: always know uniquely which edge to take
 - At most 1 arc leaving a state with a given symbol
 - No ϵ arcs
- Nondeterministic: may need to guess or explore multiple paths, choosing the right one later



NFAs vs DFAs

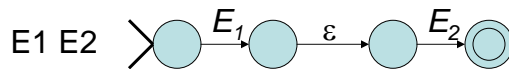
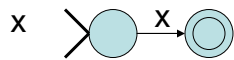
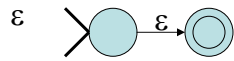
- A problem:
 - REs (e.g. specifications map easily to NFAs)
 - Can write code for DFAs easily
- How to bridge the gap?
- Can it be bridged?

A Solution

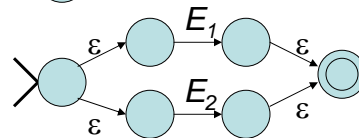
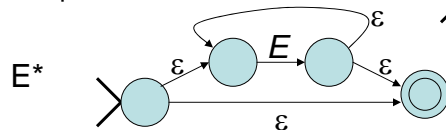
- Cool algorithm to translate any NFA to a DFA
 - Proves that NFAs aren't any more expressive
- Plan:
 - 1) Convert RE to NFA
 - 2) Convert NFA to DFA
 - 3) Convert DFA to code
- Can be done by hand or fully automatically

RE => NFA

Construct Cases Inductively



$E_1 | E_2$



NFA => DFA

- Problem: NFA can “choose” among alternative paths, while DFA must pick only one path
- Solution: subset construction
 - Each state in the DFA represents the set of states the NFA could possibly be in
 - Question: if a NFA has n states, how many states could there be in the corresponding DFA? (i.e. how many possible subsets of NFA states are there?)

Subset Construction

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. w/o consuming input

- Process the start state

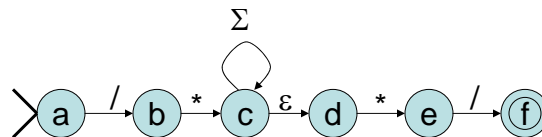
To process a DFA state S with label $[S_1, \dots, S_n]$

For each symbol x in the alphabet:

- Compute the set T of NFA states from S_1, \dots, 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 an x transition from S to T
 - Otherwise create a new DFA state T and add an x transition from S to T

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

Subset
Construction



To Tokens

- Every “final” symbol of a DFA emits a token
- Tokens are the internal compiler names for the lexemes
 - `==` becomes `equal`
 - `(` becomes `leftParen`
 - `private` becomes `private`
- You choose the names
- Also, there may be additional data ... `\r\n` might include line count

DFA => Code

- Option 1: Implement by hand using procedures
 - one procedure for each token
 - each procedure reads one character
 - choices implemented using if and switch statements
- Pros
 - straightforward to write
 - fast
- Cons
 - a fair amount of tedious work
 - may have subtle differences from the language specification

DFA => code [continued]

- Option 2: use tool to generate table driven parser
 - Rows: states of DFA
 - Columns: input characters
 - Entries: action
 - Go to next state
 - Accept token, go to start state
 - Error
- Pros
 - Convenient
 - Exactly matches specification, if tool generated
- Cons
 - “Magic”
 - Table lookups may be slower than direct code, but switch implementation is a possible revision

Automatic Scanner Generation in MiniJava

We use the `jflex` tool to automatically create a scanner from a specification file, `Scanner/minijava.jflex`

(We use the CUP tool to automatically create a parser from a specification file, `Parser/minijava.cup`, which also generates all of the code for the token classes used in the scanner, via the `Symbol` class)

The MiniJava `Makefile` automatically rebuilds the scanner (or parser) whenever its specification file changes

Symbol Class

Lexemes are represented as instances of class Symbol

```
class Symbol {
    Int sym;      // which token class?
    Object value; // any extra data for this lexeme
    ...
}
```

A different integer constant is defined for each token class in the sym helper class

```
class sym {
    static int CLASS = 1;
    static int IDENTIFIER = 2;
    static int COMMA = 3;
    ...
}
```

Can use this in printing code for Symbols; see symbolToString in minijava.jflex

Token Declarations

Declare new token classes in Parser/minijava.cup, using terminal declarations

- include Java type if Symbol stores extra data

- Examples

```
/* reserved words: */
terminal CLASS, PUBLIC, STATIC, EXTENDS;
...
/* operators: */
terminal PLUS, MINUS, STAR, SLASH, EXCLAIM;
...
/* delimiters: */
terminal OPEN_PAREN, CLOSE_PAREN;
terminal EQUALS, SEMICOLON, COMMA, PERIOD;
...
/* tokens with values: */
terminal String IDENTIFIER;
terminal Integer INT_LITERAL;
```

jflex Token Specifications

Helper definitions for character classes and regular expressions

```
letter = [a-z A-Z]
eol = [\r\n]
```

Simple) token definitions are of the form:

```
regexp { Java stmt }
```

regexp can be (at least):

- a string literal in double-quotes, e.g. "class", "<="
- a reference to a named helper, in braces, e.g. {letter}
- a character list or range, in square brackets, e.g. [a-z A-Z]
- a negated character list or range, e.g. [^\r\n]
- . (which matches any single character)
- *regexp regexp, regexp|regexp, regexp*, regexp+, regexp?, (regexp)*

jflex Tokens [Continued]

Java stmt (the accept action) is typically:

- `return symbol(sym.CLASS);` for a simple token
- `return symbol(sym.CLASS,yytext());` for a token with extra data based on the lexeme
`stringyytext()`
- empty for whitespace