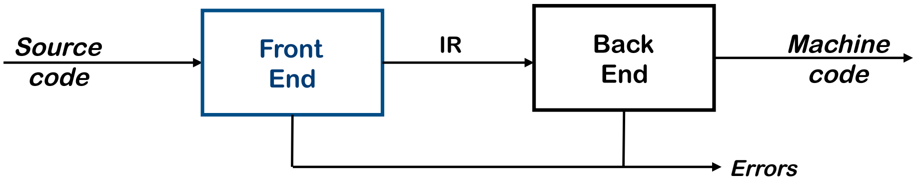
COMP 412
FALL 2009

Lexical Analysis — Introduction
Comp 412

The slides assume some familiarity with finite automata.

For a different (& more intuitive?) introduction to finite automata & recognizers, see Section 2.2 of EaC

The Front End

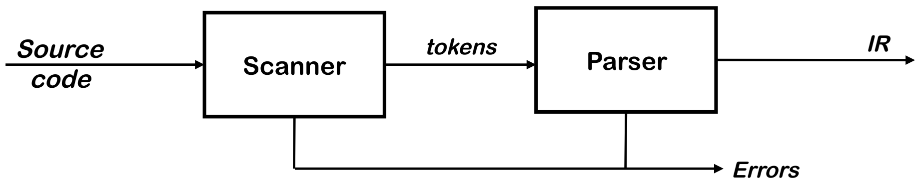


The purpose of the front end is to deal with the input language

- Perform a membership test: code $\in$ source language?
- Is the program well-formed (semantically) ?
- Build an IR version of the code for the rest of the compiler

The front end is not monolithic

The Front End



Source code

Scanner

tokens

Parser

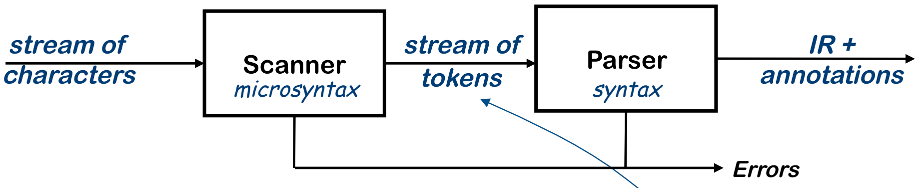
IR

Errors

Implementation Strategy

	Scanning	Parsing
Specify Syntax	regular expressions	context-free grammars
Implement recognizer	Deterministic finite automaton	Push-down automaton
Perform work	Actions on transitions in automaton	

The Front End



stream of characters

Scanner
microsyntax

stream of tokens

Parser
syntax

IR + annotations

Errors

Scanner is only pass that touches every character of the input.

Why separate the scanner and the parser?

- Scanner classifies words
- Parser constructs grammatical derivations
- Parsing is harder and slower
- Separation simplifies implementation
  - smaller grammar for parser
  - faster front end

token is a pair
<part of speech, lexeme>

The Big Picture

also called *syntactic categories* or *tokentypes*

The front end deals with syntax

- Language syntax is specified with *parts of speech*, not *words*
- Syntax checking matches *parts of speech* against a grammar

Simple expression grammar from lecture 2

1. *goal* → *expr*

2. *expr* → *expr op t*
3. | *ter*

4. *term* → num
5. | id

6. *op* → +
7. | –

}

N = { *goal*, *expr*, *term*, *op* }

P = {1, 2, 3, 4, 5, 6, 7}

The scanner turns a stream of characters into a stream of words, classified with their part of speech.

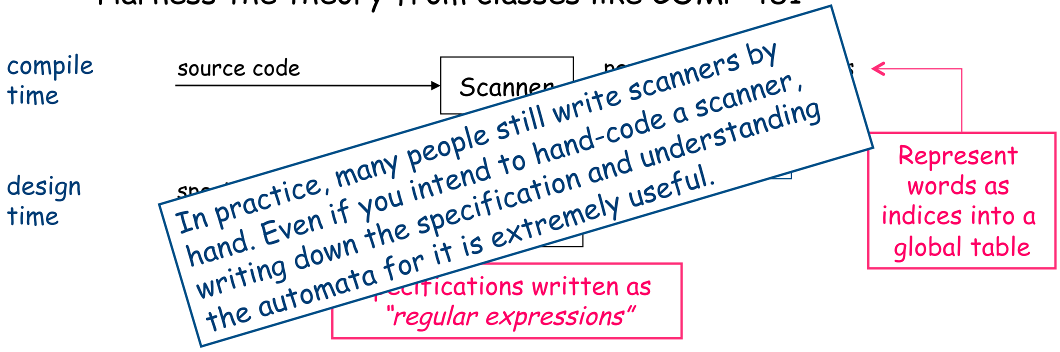
parts of speech

syntactic variables

The Big Picture

Why study automatic scanner construction?

- Avoid writing scanners by hand
- Harness the theory from classes like COMP 481



In practice, many people still write scanners by hand. Even if you intend to hand-code a scanner, writing down the specification and understanding the automata for it is extremely useful.

Goals:

- To simplify specification & implementation of scanners
- To understand the underlying techniques and technologies

Set Operations (review)

Operation	Definition
*Union of **L** and **M*** *written **L** ∪ **M***	$\mathbf{L} \cup \mathbf{M} = \{ s \mid s \in \mathbf{L} \text{ or } s \in \mathbf{M} \}$
Concatenation of ***L** and **M*** *written **LM***	$\mathbf{LM} = \{ st \mid s \in \mathbf{L} \text{ and } t \in \mathbf{M} \}$
*Kleene closure of **L*** *written **L****	$\mathbf{L}^* = \bigcup_{0 \le i \le \infty} \mathbf{L}^i$
*Positive closure of **L*** *written **L**+*	$\mathbf{L}^+ = \bigcup_{1 \le i \le \infty} \mathbf{L}^i$

These definitions should be well known

Regular Expressions

We constrain programming languages so that the spelling of a word always implies its part of speech *(few exceptions)*

The rules that impose this mapping form a *regular language*

Regular expressions (REs) describe regular languages

Regular Expression (over alphabet Σ)

- ε is a RE denoting the set $\{\varepsilon\}$
- If $\underline{a}$ is in Σ, then $\underline{a}$ is a RE denoting $\{\underline{a}\}$
- If x and y are REs denoting $L(x)$ and $L(y)$ then
  - $x \mid y$ is an RE denoting $L(x) \cup L(y)$
  - xy is an RE denoting $L(x)L(y)$
  - x^* is an RE denoting $L(x)^*$

Precedence is *closure*, then *concatenation*, then *alternation*

Regular Expressions

How do these operators help?

Regular Expression (over alphabet Σ)

- ε is a RE denoting the set {ε}
- If $\underline{a}$ is in Σ, then $\underline{a}$ is a RE denoting {$\underline{a}$}
  → the spelling of any specific word is an RE
- If *x* and *y* are REs denoting *L(x)* and *L(y)* then
  - *x* |*y* is an RE denoting $L(x) \cup L(y)$
    → any finite list of words can be written as an RE (w_0 | w_1 | … | w_n)
  - *xy* is an RE denoting *L(x)L(y)*
  - x^* is an RE denoting $L(x)^*$
    → we can use concatenation & closure to write more concise patterns and to specify infinite sets that have finite descriptions

---

Examples of Regular Expressions

Identifiers:

$\mathit{Letter} \rightarrow (\underline{a}|\underline{b}|\underline{c}| \ldots |\underline{z}|\underline{A}|\underline{B}|\underline{C}| \ldots |\underline{Z})$

$\mathit{Digit} \rightarrow (\underline{0}|\underline{1}|\underline{2}| \ldots |9)$

$\mathit{Identifier} \rightarrow \mathit{Letter}\ (\ \mathit{Letter}\ |\ \mathit{Digit}\ )^*$

shorthand for

$(\underline{a}|\underline{b}|\underline{c}| \ldots |\underline{z}|\underline{A}|\underline{B}|\underline{C}| \ldots |\underline{Z})\ ((\underline{a}|\underline{b}|\underline{c}| \ldots |\underline{z}|\underline{A}|\underline{B}|\underline{C}| \ldots |\underline{Z})\ |\ (\underline{0}|\underline{1}|\underline{2}| \ldots |\underline{9}))^*$

Numbers:

$\mathit{Integer} \rightarrow (\underline{+}|\underline{-}|\varepsilon)\ (\underline{0}|\ (\underline{1}|\underline{2}|\underline{3}| \ldots |\underline{9})(\mathit{Digit}^*)\ )$

$\mathit{Decimal} \rightarrow \mathit{Integer}\ \underline{.}\ \mathit{Digit}^*$

$\mathit{Real} \rightarrow (\ \mathit{Integer}\ |\ \mathit{Decimal}\ )\ \underline{E}\ (\underline{+}|\underline{-}|\varepsilon)\ \mathit{Digit}^*$

$\mathit{Complex} \rightarrow \underline{(}\ \mathit{Real}\ \underline{,}\ \mathit{Real}\ \underline{)}$

Numbers can get much more complicated!

Using symbolic names does not imply recursion

underlining indicates a letter in the input stream

Regular Expressions

So what's the point?

We use regular expressions to specify the mapping of words to parts of speech for the lexical analyzer

Using results from automata theory and theory of algorithms, we can automate construction of recognizers from REs

⇒ We study REs and associated theory to automate scanner construction !

⇒ Fortunately, the automatic techiques lead to fast scanners

  → used in text editors, URL filtering software, …

Example

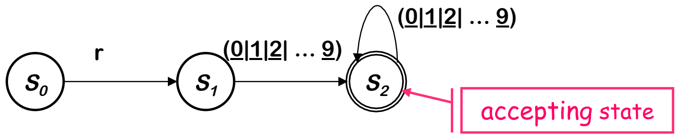
(from Lab 1)

Consider the problem of recognizing ILOC register names

Register → r $(\underline{0}|\underline{1}|\underline{2}|\ \dots\ |\ \underline{9})\ (\underline{0}|\underline{1}|\underline{2}|\ \dots\ |\ \underline{9})^*$

- Allows registers of arbitrary number
- Requires at least one digit

RE corresponds to a recognizer (or DFA)



accepting state

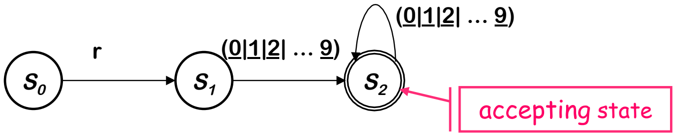
Recognizer for *Register*

Transitions on other inputs go to an error state, s_e

Example (continued)

DFA operation

- Start in state S_0 & make transitions on each input character
- DFA accepts a word x *iff* x leaves it in a final state (S_2)



accepting state

Recognizer for *Register*

So,

- r17 takes it through s_0, s_1, s_2 and accepts
- r takes it through s_0, s_1 and fails
- a takes it straight to s_e

Example (continued)

To be useful, the recognizer must be converted into code

```
Char ← next character
State ← s0

while (Char ≠ EOF)
    State ← δ(State,Char)
    Char ← next character

if (State is a final state )
    then report success
    else  report failure
```

Skeleton recognizer

δ	r	0,1,2,3,4,5,6,7,8,9	*All others*
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

Table encoding the RE

O(1) cost per character (or per transition)

Example (continued)

We can add "actions" to each transition

```
Char ← next character
State ← $s_0$

while (Char ≠ EOF)
    Next ← $\delta$(State,Char)
    Act   ← $\alpha$(State,Char)
    perform action Act
    State ← Next
    Char ← next character

if (State is a final state )
    then report success
    else  report failure
```

Skeleton recognizer

δ α	r	0,1,2,3,4, 5,6,7,8,9	*All others*
s_0	s_1 *start*	s_e *error*	s_e *error*
s_1	s_e *error*	s_2 *add*	s_e *error*
s_2	s_e *error*	s_2 *add*	s_e *error*
s_e	s_e *error*	s_e *error*	s_e *error*

Table encoding RE

Typical action is to capture the lexeme

What if we need a tighter specification?

r *Digit Digit** allows arbitrary numbers

- Accepts r00000
- Accepts r99999
- What if we want to limit it to r0 through r31 ?

Write a tighter regular expression

- *Register* → r ((0|1|2) (*Digit* | ε) | (4|5|6|7|8|9) | (3|30|31))
- *Register* → r0|r1|r2| … |r31|r00|r01|r02| … |r09

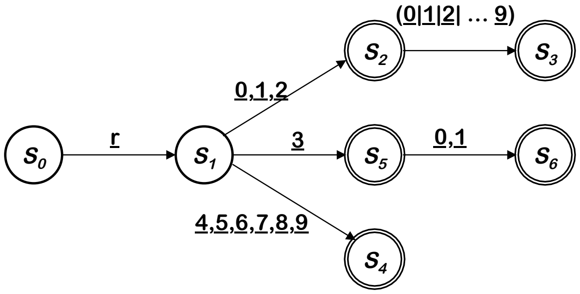
Produces a more complex DFA

- DFA has more states
- DFA has same cost per transition (*or per character*)
- DFA has same basic implementation

Tighter register specification (continued)

The DFA for

Register → r ((0|1|2) (*Digit* | ε) | (4|5|6|7|8|9) | (3|30|31))



- Accepts a more constrained set of register names
- Same set of actions, more states

Tighter register specification (continued)

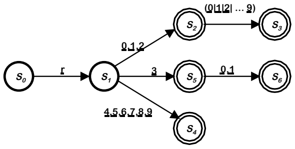
δ	r	0,1	2	3	4-9	All others
s_0	s_1	s_e	s_e	s_e	s_e	s_e
s_1	s_e	s_2	s_2	s_5	s_4	s_e
s_2	s_e	s_3	s_3	s_3	s_3	s_e
s_3	s_e	s_e	s_e	s_e	s_e	s_e
s_4	s_e	s_e	s_e	s_e	s_e	s_e
s_5	s_e	s_6	s_e	s_e	s_e	s_e
s_6	s_e	s_e	s_e	s_e	s_e	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e

This table runs in the same skeleton recognizer

Table encoding RE for the tighter register specification

Tighter register specification (continued)

State Action	r	0,1	2	3	4,5,6 7,8,9	other
0	1 *start*	e	e	e	e	e
1	e	2 *add*	2 *add*	5 *add*	4 *add*	e
2	e	3 *add*	3 *add*	3 *add*	3 *add*	e *exit*
3,4	e	e	e	e	e	e *exit*
6	e	6 *add*	e	e	e	e *exit*
6	e	e	e	e	e	x *exit*
e	e	e	e	e	e	e



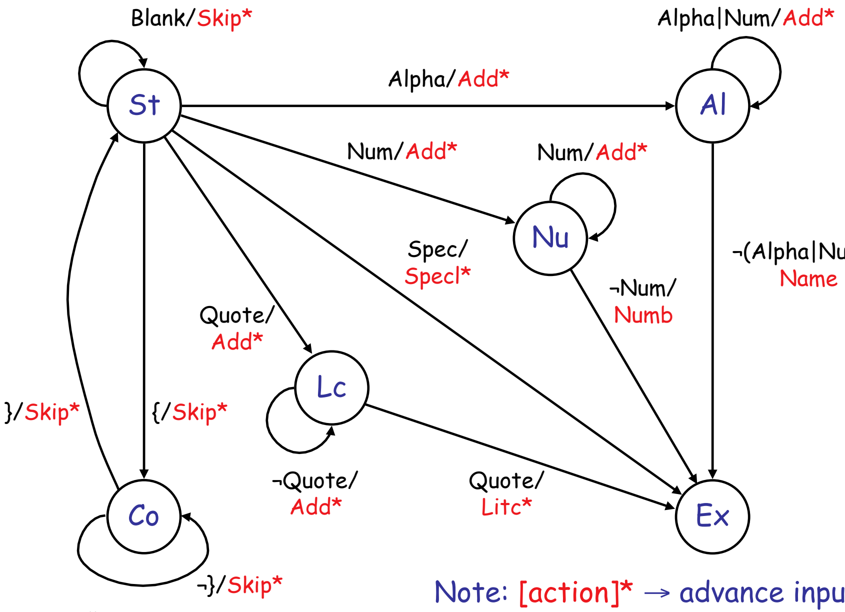
STOP

Review of Scanners from Comp 314 (1998)

- Lexical Analysis Strategy: Simulation of Finite Automaton
  - States, characters, actions
  - State transition δ(state,charclass) determines next state
- Next character function
  - Reads next character into buffer
  - Computes *character class* by fast table lookup
- Transitions from state to state
  - Current state and next character determine (via δ)
    - → Next state and action to be performed
    - → Some actions *preload* next character
- Identifiers distinguished from keywords by hashed lookup
  - This differs from EAC advice (discussion later)
  - Permits translation of identifiers into <type, symbol_index>
    - → Keywords each get their own type

Material Ken added from 314

A Lexical Analysis Example



Note: [action]* → advance input stream

Comp 314 Lexical Scan Code

```
current = START_STATE;
token = "";
// assume next character has been preloaded into a buffer
while (current != EX)
{
    int charClass = inputstream->thisClass();
    switch (current->action(charClass))
    {
        case SKIP:
            inputstream->advance();break;
        case ADD:
            char* t = token; int n = ::strlen(t);
            token = new char[n + 2]; ::strcpy(token, t);
            token[n] = inputstream->thisChar(); token[n+1] = 0;
            delete [] t; inputstream->advance(); break;
        case NAME:
            Entry * e = symTable->lookup(token);
            tokenType = (e->type==NULL_TYPE ? NAME_TYPE : e->type);
            break;
        ...
    }
    current = current->nextState(charClass);
}
```