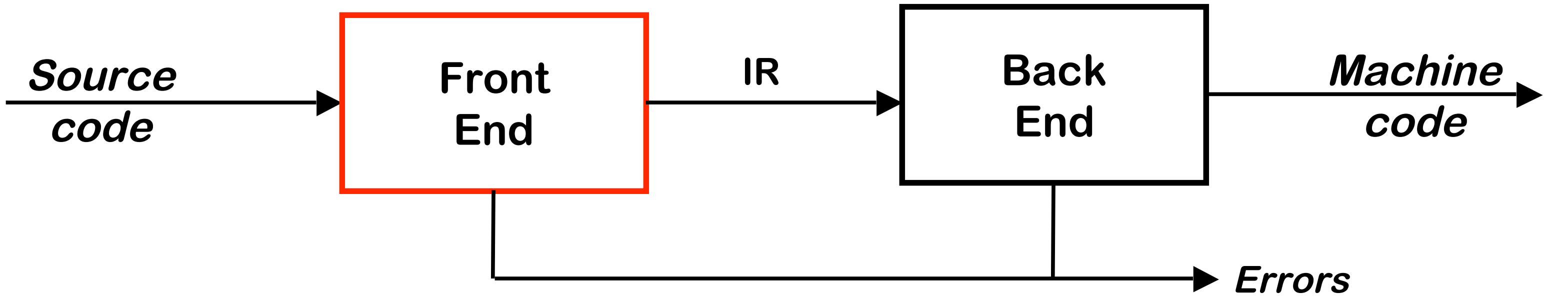
Lexical Analysis - An Introduction

COMP 412
Fall 2005

The Front End

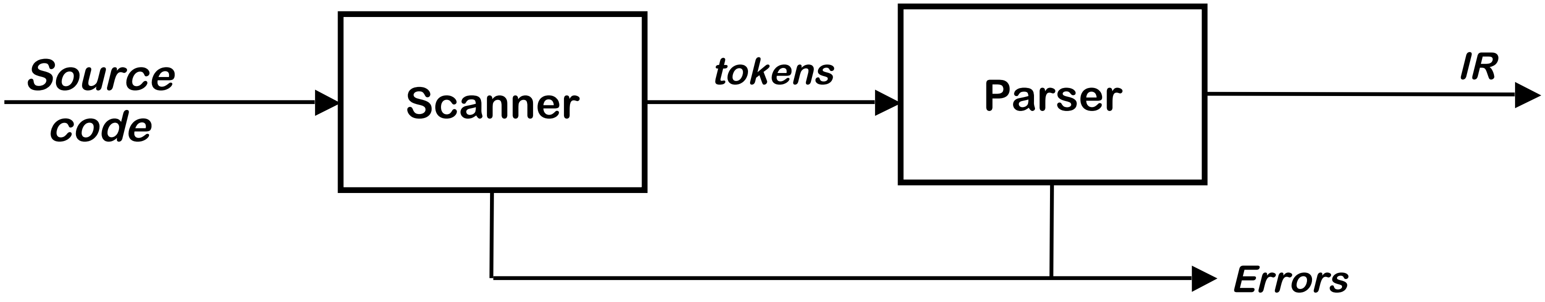


The purpose of the front end is to deal with the input language

- Perform a membership test: code ∈ 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

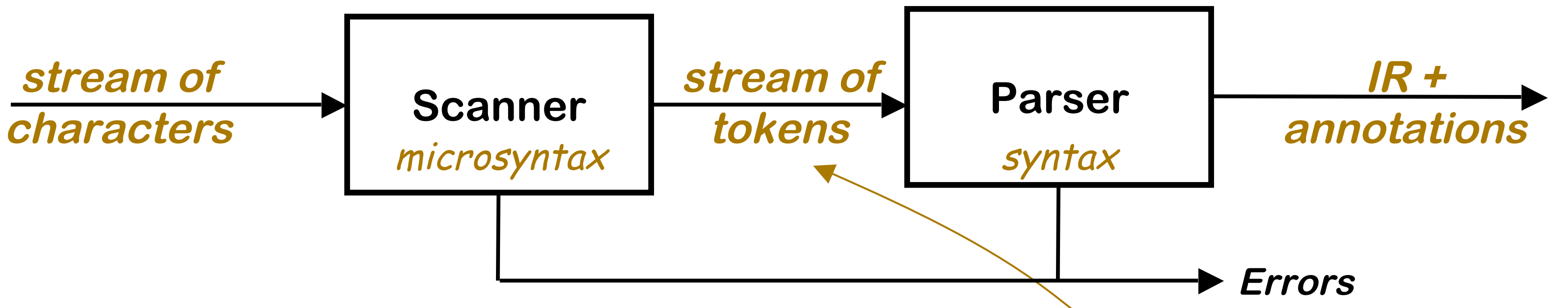
The Front End



Implementation Strategy

- Specify syntax in a formal notation
  - regular expressions in scanning, context-free grammars in parsing
- Simulate an automaton to recognize valid strings
  - finite automata, push-down automata (COMP 481)
- Automate construction of the simulations
  - table-driven simulations or direct-coded simulations
- Add "actions" to automaton to create representations

The Front End



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>

also called *syntactic categories* or *tokentypes*

The Big Picture

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*** → ***exp***
3. | ***ter***

4. ***term*** → **num**
5. | **id**

6. ***op*** → **+**
7. | **–**

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

, - }

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

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

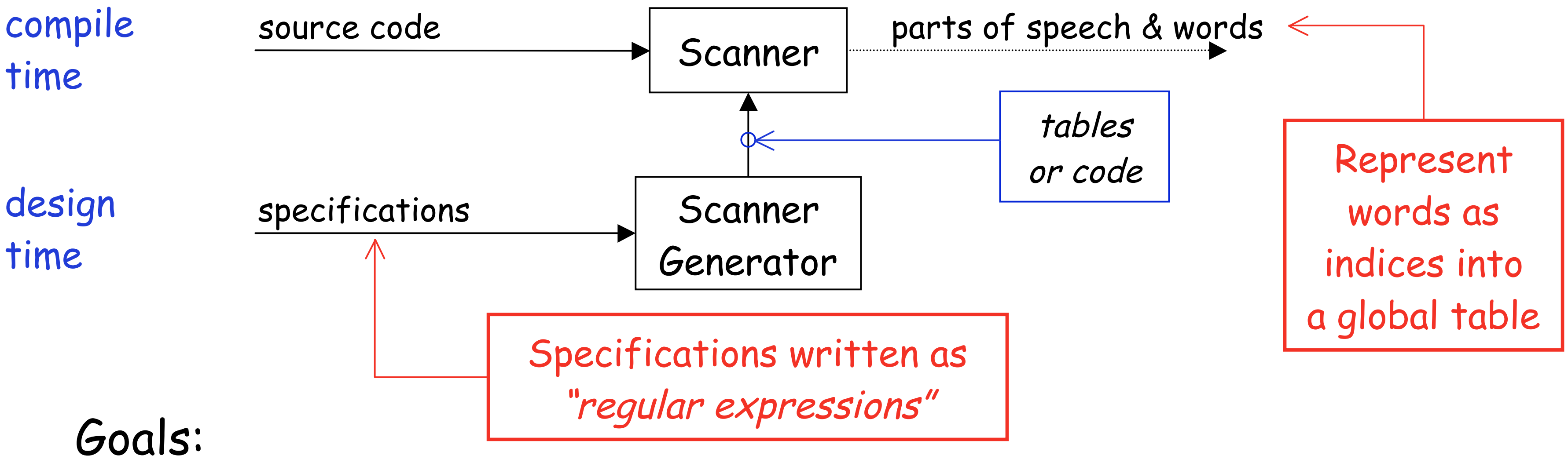
parts of speech

syntactic variables

The Big Picture

Why study scanner construction?

- We want to avoid writing scanners by hand
- We want to harness the theory from classes like COMP 481



compile time

design time

source code

Scanner

parts of speech & words

tables or code

specifications

Scanner Generator

Represent words as indices into a global table

Specifications written as *"regular expressions"*

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

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 or patterns that impose this maping 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 $\{\varepsilon\}$
- If $\underline{a}$ is in Σ, then $\underline{a}$ is a RE denoting $\{\underline{a}\}$
  - → the spelling of a word is an RE
- If x and y are REs denoting $L(x)$ and $L(y)$ then
  - – $x \mid y$ is an RE denoting $L(x) \cup L(y)$
    - → any finite list of words can be written as an RE $(w_0 \mid w_1 \mid \dots \mid w_n)$
  - – 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:

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

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

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

shorthand for

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

Numbers:

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

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

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

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

Numbers can get much more complicated!

underlining indicates a letter in the input stream

Regular Expressions (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, …

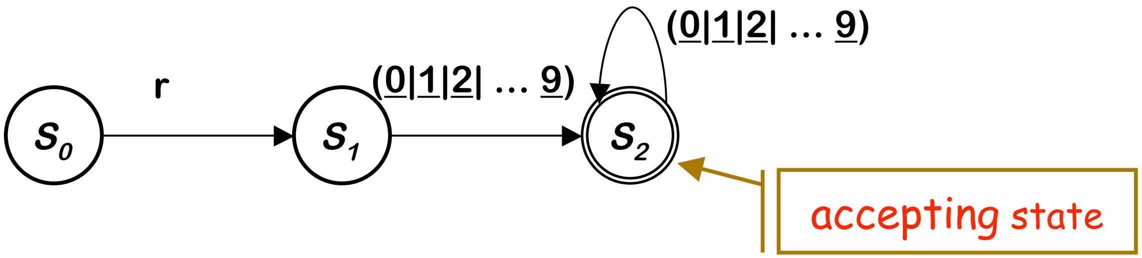
Example (from Lab 1)

Consider the problem of recognizing ILOC register names

$$\textit{Register} \rightarrow r\ (\underline{0}|\underline{1}|\underline{2}|\ \ldots\ |\ \underline{9})\ (\underline{0}|\underline{1}|\underline{2}|\ \ldots\ |\ \underline{9})^*$$

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

RE corresponds to a recognizer (or DFA)



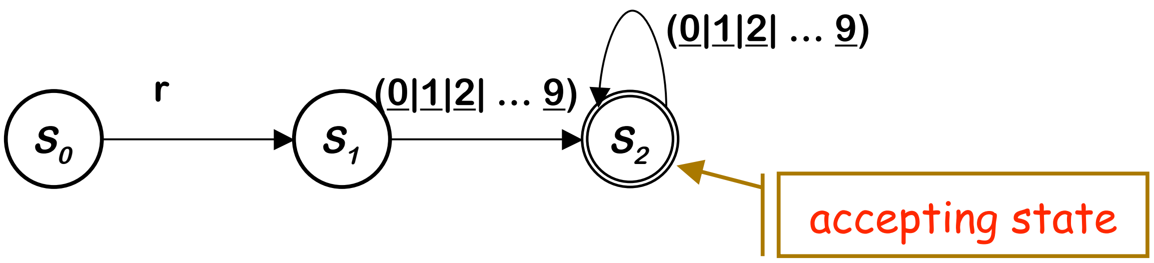
Recognizer for *Register*

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 ← $s_0$

while (Char ≠ EOF)
    State ← $\delta$(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 ← δ(State,Char)
    Act  ← α(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