Class Information

- New accounts on ilab cluster: Interface to create new accounts may not be set up yet.
- New classroom for section 1 (Tuesday 1:55pm - 2:50pm): **LCB 110** (Livingston Classroom Building) next to Tillett Hall.
- Special permission numbers?
- Recitation starts today (section 3).
- First homework is out (see our class web site).

Syntax and Semantics of Prog. Languages

Syntax:
Describes what a legal program looks like

Semantics:
Describes what a correct (legal) program means

A formal language is a (possibly infinite) set of sentences (finite sequences of symbols) over a finite alphabet Σ of (terminal) symbols: $L \subseteq \Sigma^*$

Examples:

- L = { identifiers of length 2 } with $\Sigma = \{a, b, c\}$
- L = { strings of only 1s or only 0s }
- L = { strings starting with **$** and ending with #, and any combination of **0**s and **1**s inbetween }
- L = { all syntactically correct Java programs }

Claim: *The larger the language, the harder it is to formally specify the language. In other words, it get's harder for each i:* $L_1 \subset L_2 \subset L_3 \ldots \subset L_i \subset \ldots$.
True or false?

Syntax and Semantics: How does it work?

Syntactic representation of "values"

What do the following syntactic expressions have in common?

XI

1011

B

$\lambda fx.(f(f(f(f(f(f(f(f(f(f(fx)))))))))))$

$ ||||||||||| #

$3 + 20 - (2 \times 6)$

Answer: They are possible representations of the integer value "11" (written as a decimal number)

What is computation?

Possible answer: A (finite) sequence of syntactic manipulations of value respresentations ending in a "normal form" which is called the result. Normal forms cannot be manipulated any further.

Syntax and Semantics: How does it work?

Here is a "game" (**rewrite system**):

input: Sequence of characters starting with **\$** and ending with #, and any combination of **0**s and **1**s inbetween.

rules: You may replace a character pattern X at any position within the character sequence on the left-hand-side by the pattern Y on the right-hand-side: $X \Rightarrow Y$:

rule 1	$\mathbf{\$\ 1} \Rightarrow \mathbf{1\ \&}$
rule 2	$\mathbf{\$\ 0} \Rightarrow \mathbf{0\ \$}$
rule 3	$\mathbf{\&\ 1} \Rightarrow \mathbf{1\ \$}$
rule 4	$\mathbf{\&\ 0} \Rightarrow \mathbf{0\ \&}$
rule 5	$\mathbf{\$\ \#} \Rightarrow \rightarrow \mathbf{A}$
rule 6	$\mathbf{\&\ \#} \Rightarrow \rightarrow \mathbf{B}$

Replace patterns using the rules as often as you can. When you cannot replace a pattern any more, stop.

Syntax and Semantics: How does it work?

example input:

$ 0 0 #

$\boxed{\mathbf{\$\ 0}}$ **0 #** is rewritten as $\boxed{\mathbf{0\ \$}}$ **0 #** by rule 2

0 $\boxed{\mathbf{\$\ 0}}$ **#** is rewritten as **0** $\boxed{\mathbf{0\ \$}}$ **#** by rule 2

0 0 $\boxed{\mathbf{\$\ \#}}$ is rewritten as **0 0** $\boxed{\rightarrow \mathbf{A}}$ by rule 6

no more rules can be applied (**STOP**)

More examples:

> **$ 0 1 1 0 1 #**
>
> **$ 1 0 1 0 0 #**
>
> **$ 1 1 0 0 1 #**

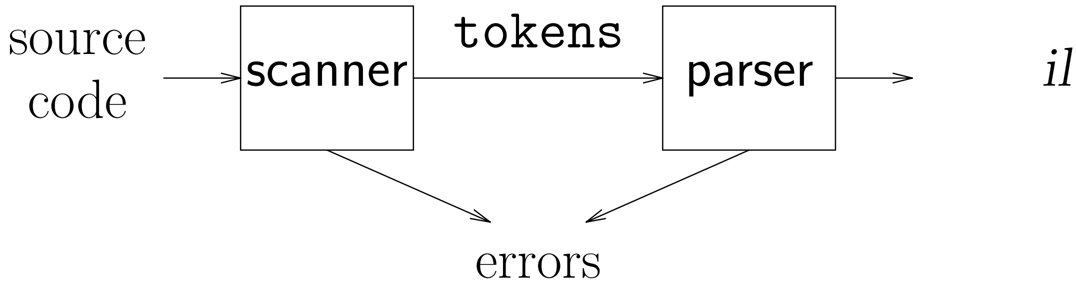
Questions

- Can we get different "results" for the same input string?
- Does all this have a meaning (**semantics**), or are we just pushing symbols?

Syntax without Semantics?

Sorry, not useful!

Review - Front end of a compiler



Parser: syntax & semantic analyzer, *il* code generator
(syntax-directed translator)

Front End Responsibilities:

- recognize legal programs
- report errors
- produce *il*
- preliminary storage map
- shape the code for the back end

Much of front end construction can be automated

Syntax and Semantics of Prog. Languages

The syntax of programming languages is often defined in two layers: *tokens* and *sentences*.

- *tokens* – basic units of the language
  Question: $\boxed{\text{How to spell a token (word)?}}$
  Answer: regular expressions

- *sentences* – legal combination of tokens in the language
  Question:
  $\boxed{\text{How to build correct sentences with tokens?}}$
  Answer: (context-free) grammars (CFG)

- E.g., Backus-Naur form (BNF) is a formalism used to express the syntax of programming languages.

Formalisms for Lexical and Syntactic Analysis

1. Lexical Analysis: Converts source code into sequence of tokens.
2. Syntax Analysis: Structures tokens into parse tree.

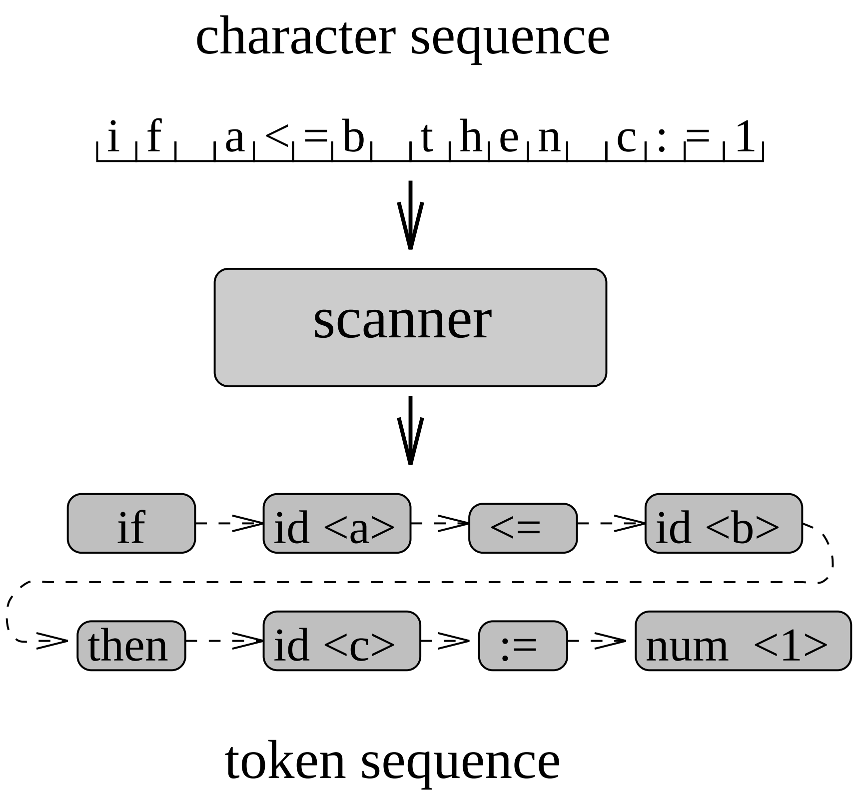
Two issues in Formal Languages:

- Language Specification → formalism to describe what a valid program (sentence) looks like.
- Language Recognition → formalism to describe a machine and an algorithm that can verify that a program is valid or not.

For (2), we use **context-free grammars** to specify programming languages. Note: recognition, i.e., parsing algorithms using PDAs (push-down automata) will be covered in CS415.

For (1), we use **regular grammars/expressions** for specification and **finite (state) automata** for recognition.

Lexical Analysis (Scott 2.1, 2.2)



Tokens (Terminal Symbols of CFG, Words of Lang.)

- Smallest "atomic" units of syntax
- Used to build all the other constructs
- Example, Pascal:

  keywords: `program begin if then` ...

  `= * / - < > = <= >= <>`

  `( ) [ ] ; := . ,` ...

  `number` (Example: `3.14 28` ...)

  `identifier` (Example: `b square addEntry` ...)

Lexical Analysis (cont.)

Identifiers

- Names of variables, etc.
- Sequence of terminals of restricted form;
  Example, Pascal: `A31`, but not `1A3`
- Upper/lower case sensitive?

Keywords

- Special identifiers which represent tokens in the language
- May be reserved (*reserved words*) or not
  - E.g., Pascal: "**if**" is reserved.
  - E.g., FORTRAN: "**if**" is not reserved.

Delimiters – When does character string for token end?

- Example: identifiers are longest possible character sequence that does not include a delimiter
- Few delimiters in Fortran (not even '␣')
  - `DO I = 1.5` same as `DOI=1.5`
- Most languages have more delimiters such as '␣', new line, keywords, . . .

Regular Expressions

A syntax (notation) to specify regular languages.

RE r	**Language L(r)**
$\mathbf{a}$	$\{\mathbf{a}\}$
ϵ	$\{\epsilon\}$
$\mathrm{r} \mid \mathrm{s}$	$\mathrm{L(r)} \cup \mathrm{L(s)}$
rs	$\{rs \mid r \in \mathrm{L(r)}, s \in \mathrm{L(s)}\}$
r^+	$\mathrm{L(r)} \cup \mathrm{L(rr)} \cup \mathrm{L(rrr)} \cup \ldots$ (any number of r's concatenated)
r^* ($\mathrm{r}^* = \mathrm{r}^+ \vert \epsilon$)	$\{\epsilon\} \cup \mathrm{L(r)} \cup \mathrm{L(rr)} \cup \mathrm{L(rrr)} \cup \ldots$
(s)	L(s)

(all left-assoc. in order of increasing precedence.)

$\Rightarrow$ Note: Inductive definition!

Examples of Expressions

RE	Language
$\mathbf{a}\vert\mathbf{bc}$	
$(\mathbf{a}\vert\mathbf{b})\mathbf{c}$	
$\mathbf{a}\epsilon$	
$\mathbf{a}^*\vert\mathbf{b}$	
$\mathbf{ab}^*$	
$\mathbf{ab}^*\vert\mathbf{c}^+$	
$(\mathbf{a}\vert\mathbf{b})^*$	
$(\mathbf{0}\vert\mathbf{1})^*\mathbf{1}$	

Examples of Expressions - Solution

RE	Language
$\mathbf{a}\|\mathbf{bc}$	$\{\mathbf{a}, \mathbf{bc}\}$
$(\mathbf{a}\|\mathbf{b})\mathbf{c}$	$\{\mathbf{ac}, \mathbf{bc}\}$
$\mathbf{a}\epsilon$	$\{\mathbf{a}\}$
$\mathbf{a}^*\|\mathbf{b}$	$\{\epsilon, \mathbf{a}, \mathbf{aa}, \mathbf{aaa}, \mathbf{aaaa}, \ldots\} \cup \{\mathbf{b}\}$
$\mathbf{ab}^*$	$\{\mathbf{a}, \mathbf{ab}, \mathbf{abb}, \mathbf{abbb}, \mathbf{abbbb}, \ldots\}$
$\mathbf{ab}^*\|\mathbf{c}^+$	$\{\mathbf{a}, \mathbf{ab}, \mathbf{abb}, \mathbf{abbb}, \mathbf{abbbb}, \ldots\} \cup$ $\{\mathbf{c}, \mathbf{cc}, \mathbf{ccc}, \ldots\}$
$(\mathbf{a}\|\mathbf{b})^*$	$\{\epsilon, \mathbf{a}, \mathbf{b}, \mathbf{aa}, \mathbf{ab}, \mathbf{ba}, \mathbf{bb}, \mathbf{aaa}, \mathbf{aab}, \ldots\}$
$(\mathbf{0}\|\mathbf{1})^*\mathbf{1}$	binary numbers ending in 1

Regular Expressions for Programming Languages

Let letter stand for A | B | C | . . . | Z
Let digit stand for 0 | 1 | 2 | . . . | 9

integer constant:

identifier:

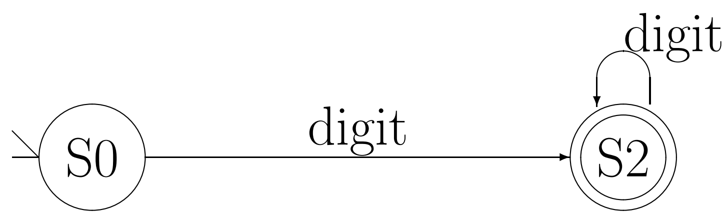
real constant:

Recognizers for Regular Expressions

Example 1: integer constant

RE: digit$^+$

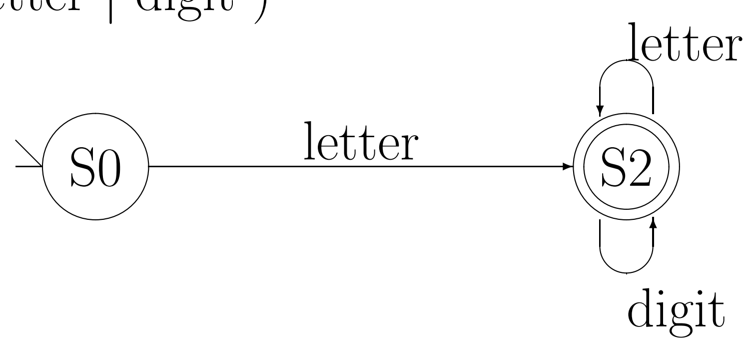
FSA:



Example 2: identifier

RE: letter (letter | digit)*

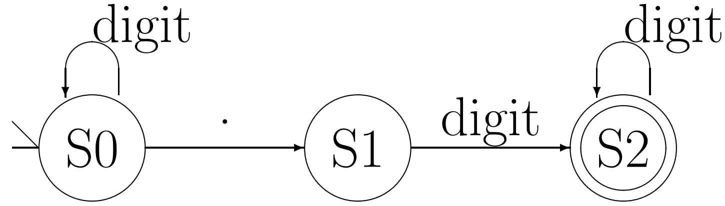
FSA:



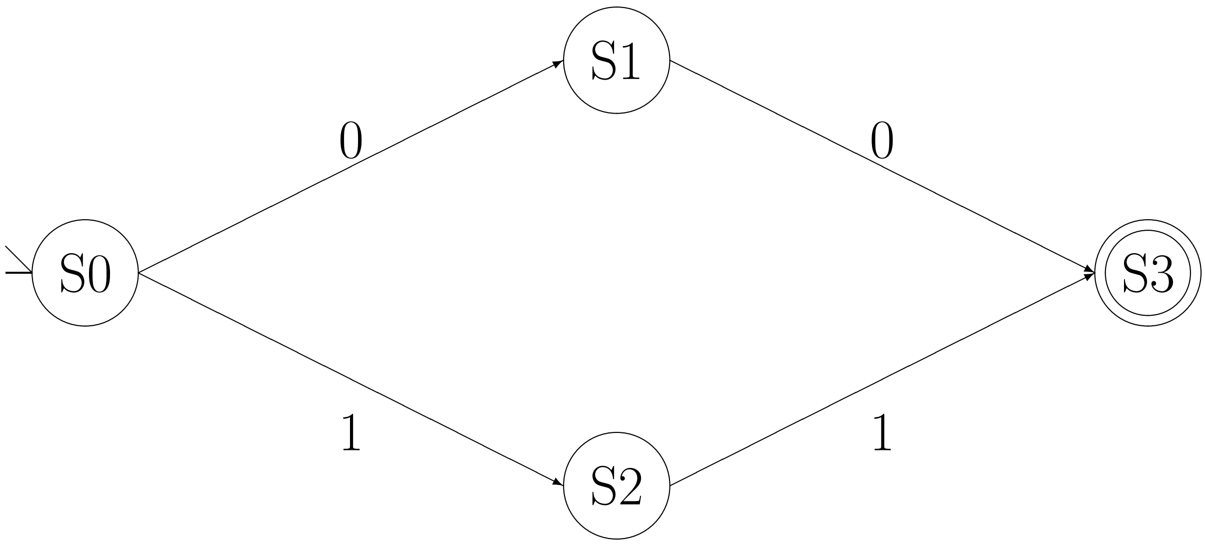
Example 3: Real constant

RE: digit*.digit$^+$

FSA:



Finite State Automata



A Finite-State Automaton is a quadruple:
$< S, s, F, T >$

- S is a set of *states*, e.g., $\{S0, S1, S2, S3\}$
- s is the *start state*, e.g., $S0$
- F is a set of *final states*, e.g., $\{S3\}$
- T is a set of *labeled transitions*, of the form $(state, input) \mapsto state$ [i.e., $S \times \Sigma \rightarrow S$]

Finite State Automata

Transitions can be represented using a transition table:

		0	1	Input
State	S0	S1	S2	
	S1	S3	-	
	S2	-	S3	

An FSA *accepts* or *recognizes* an input string iff there is some path from its start state to a final state such that the labels on the path are that string.

Lack of entry in the table (or no arc for a given character) indicates an error—reject.