Lexical Analysis

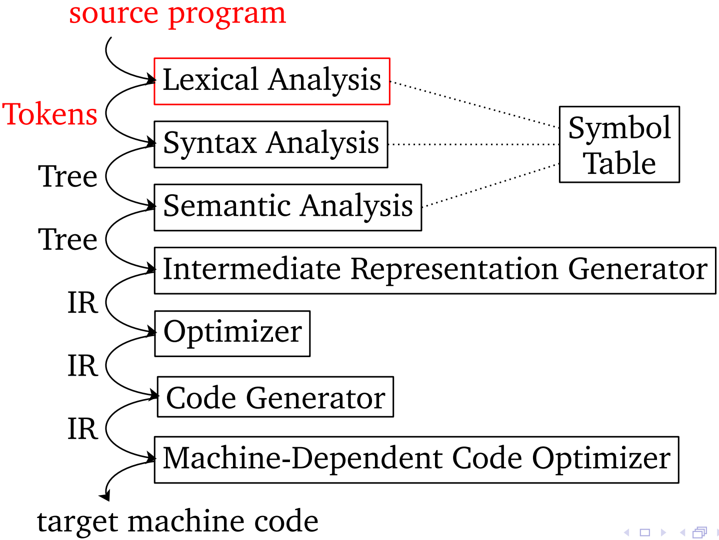
Eva Rose Kristoffer H. Rose

NYU Courant Institute

Febuary 3, 2014

1. Lexers
2. Regular Expressions
3. Finite Automata
4. HACS

First compilation phase



source program

Lexical Analysis

Tokens

Syntax Analysis

Tree

Semantic Analysis

Tree

Intermediate Representation Generator

IR

Optimizer

IR

Code Generator

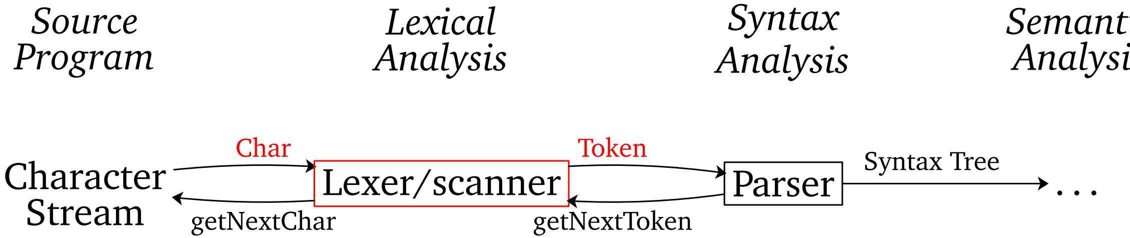
IR

Machine-Dependent Code Optimizer

target machine code

Symbol Table

Front end modules…



Source Program

Lexical Analysis

Syntax Analysis

Semant Analysi

Character Stream

Char

getNextChar

Lexer/scanner

Token

getNextToken

Parser

Syntax Tree

…

Example

```
position = initial + rate * 60
```

scanned into list of *tokens*:

$\langle \mathbf{id}, 1\rangle \ \langle = \rangle \ \langle \mathbf{id}, 2\rangle \ \langle + \rangle \ \langle \mathbf{id}, 3\rangle \ \langle * \rangle \ \langle \mathbf{num}, 60\rangle$

1	`position`
2	`initial`
3	`rate`

Example

Lexeme	Token	Pattern (informal)	Attribute value
`position`	⟨**id**, 1⟩	identifier string	1
`=`	⟨=⟩	equality symbol	
`initial`	⟨**id**, 2⟩	identifier string	2
`+`	⟨+⟩	addition symbol	
`rate`	⟨**id**, 3⟩	identifier string	3
`*`	⟨∗⟩	multiplication symbol	
`60`	⟨**num**, 60⟩	numeric constant	60

Purpose

Primary lexer tasks:

- identify lexemes/tokens
- identify token attributes
- strip whitespace and comments
- error handling

Identifying lexemes : input buffering

Two Buffer Scheme!

- Two N-buffers (one system read command each)
- **lexemeBegin** pointer: beginning of current lexeme
- **forward** pointer: current character scanning
- **eof** special sentinel character

Identifying tokens: lexeme patterns

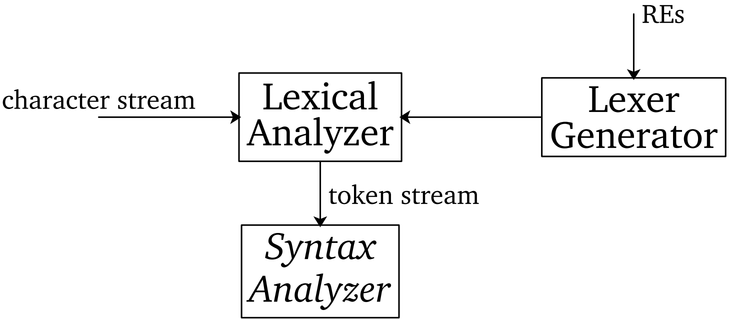
Some lexeme patterns: `+`, `6-0`, `=`, `=-=`, `p-o-s-i-t-i-o-n`, …

"The world" of string patterns include finite, infinite, cyclic, and balanced patterns …

partition into

- Regular expressions (RE) (tokens)
- other…

Lexer Generator


REs

character stream

Lexical Analyzer

Lexer Generator

token stream

Syntax Analyzer

- Lex, Flex
- HACS has its own lexical-analyser *generator*

1. Lexers
2. **Regular Expressions**
3. Finite Automata
4. HACS

Token specification

Some lexeme patterns as RE:

- operator: $+$
- equality: $(=|==)$
- number: $\{\, d^{+} \mid d \in \mathit{Digit} \,\}$
- identifiers: $\{\, l(d|l)^{*} \mid d \in \mathit{Digit},\ l \in \mathit{Letter} \,\}$
- empty string: ϵ (epsilon)

Definition

Basic RE taxonomy:

Concept	Syntax	A Pattern	Matches
Character	*itself*	b	b
Concatenation	$e_1\, e_2$	bc	bc
Choice	$e_1 \mid e_2$	a | bc	a, bc
Repetition($\geq$0)	e^*	$\mathtt{a}^*$	ϵ, a, aa, ...
Grouping	(e)	(a|b)c	ac, bc

where $*$ denotes the *Kleene Closure* of the generated language.

Definition

Extended RE taxonomy:

Concept	Syntax	A Pattern	Matches
Optional	$e?$	$(\mathtt{a} \mid \mathtt{b})?$	ϵ, a, b
Repetition($\geq$1)	e^+	$\mathtt{a}^+$	a, aa, ...
Subtoken	$\langle \mathit{name} \rangle$	$\langle \mathtt{Int} \rangle$	12, 0, 55
Character class	$[\ldots]$	`[0-9_a-z]`	7, _, c

where $+$ denotes the *Positive Closure* of the generated language.

Formal Languages

Set of strings

Approaches for defining formal languages:

- basic RE taxonomy
- extended RE taxonomy
- context-free grammar (CFG) <—
- nondeterministic finite automaton
- determinstic finite automaton

Context-free formal grammars

Formal grammar:

$$V \rightarrow \texttt{w} \quad \text{(production/rewrite rules)}$$

- V is non-terminal symbol
- `w` is *empty*, or string of terminals/non-terminals

Context-free:

Production rules independent of the nonterminal context

Expressive Language Power

“Language power” given by the possibly generated strings

Comparisons:

- extended and basic RE has *same power*
- CFG more powerful than RE …
- finite automata and RE has *same power*

Comparing CFG and RE

CFG at least as expressive as RE:
Can emulate RE with CFG:

RE:	CFG:
$(\mathsf{a} \mid \mathsf{b})^* \, \mathsf{abb}$	$X \to Y\,\mathsf{abb}$ $Y \to YZ \mid \epsilon$ $Z \to \mathsf{a} \mid \mathsf{b}$

Comparing CFG and RE

CFG more expressive than RE:

CFG:	RE:
$X \to \texttt{[}\ X\ \texttt{]} \mid \texttt{a}$	?

Captures a, [a], [[a]], . . . , $\texttt{[}^n\ \texttt{a}\ \texttt{]}^n$.

HACS: Lexical Analysis

Example 2.1 first.hx

```
space [ \t\n] ;

token Int       | ⟨Digit⟩+ ;
token Float     | ⟨Int⟩ "." ⟨Int⟩ ;
token Id        | ⟨Lower⟩+ ('_'? ⟨Int⟩)? ;

token fragment Digit  | [0-9] ;
token fragment Lower  | [a-z] ;
```

1 Lexers

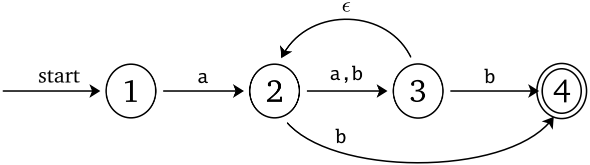
2 Regular Expressions

3 Finite Automata

4 HACS

NFA: Nondeterministic Finite Automata

NFA *accepting* $a(a|b)^*b$:



NFA accepting $a(a|b)^*b$

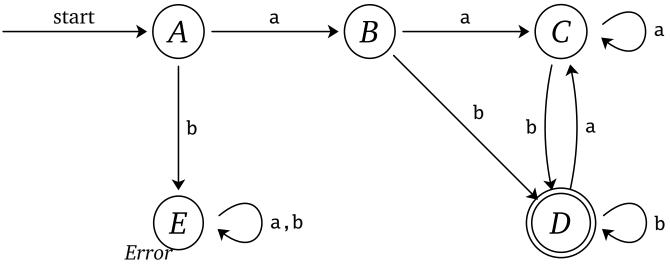
- $States = \{1, 2, 3, 4\}$
- $\Sigma = \{\mathtt{a}, \mathtt{b}\}$
-

	a	b	ϵ
1	$\{2\}$	$\{\}$	$\{\}$
2	$\{3\}$	$\{3, 4\}$	$\{\}$
3	$\{\}$	$\{4\}$	$\{2\}$
4	$\{\}$	$\{\}$	$\{\}$

- $Start = 1$
- $Finals = \{4\}$

DFA: Deterministic finite automata

DFA *accepting* $a(a|b)^*b$:



DFA accepting $a(a|b)^*b$

- $States = \{A, B, C, D, E\}$
- $\Sigma = \{\mathtt{a}, \mathtt{b}\}$
State	a	b
A	B	E
B	C	D
C	C	D
D	C	D
E	E	E
- $Start = A$
- $Finals = \{D\}$

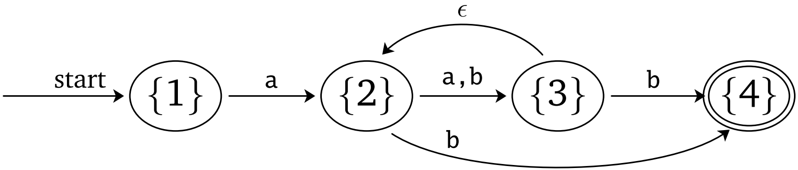
Automaton conversion

Converting an NFA to a DFA:

- ϵ-*elimination*
- *ambiguity resolution*

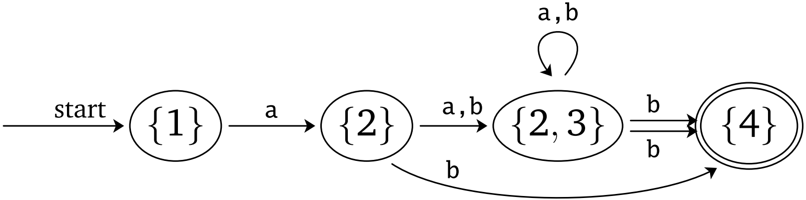
Converting NFA to DFA accepting $a(a|b)^*b$

Create subsets:



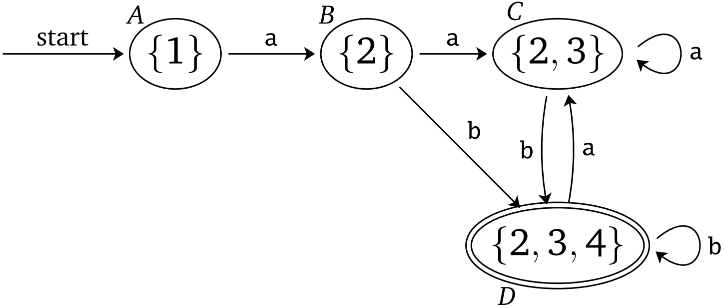
NFA to DFA accepting $a(a|b)^*b$

Eliminate ϵ-transitions:



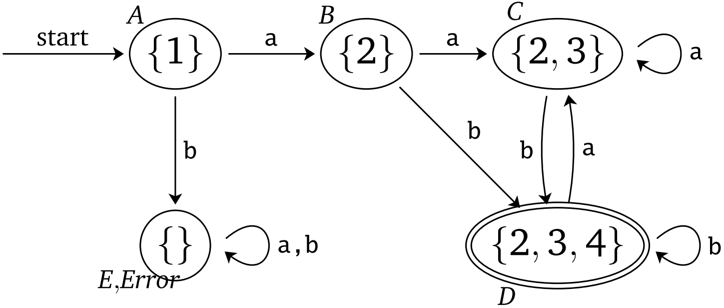
NFA to DFA accepting $a(a|b)^*b$

Eliminate ambiguities:



NFA to DFA accepting $a(a|b)^*b$

Add missing transitions:



Comparing DFA and NFA properties

	DFA	*NFA*
labels from same state	unique complete	may be same may be incomplete
ϵ-labels	no	yes
transition table	states	state sets
expressive power	same as RE	same as RE
trade-off	easy to run	easy to design

1 Lexers

2 Regular Expressions

3 Finite Automata

4 HACS

The HACS lexer

Regular expression specification:

```
space [ \t\n] ;

token Int       | ⟨Digit⟩+ ;
token Float     | ⟨Int⟩ "." ⟨Int⟩ ;
token Id        | ⟨Lower⟩+ ('_'? ⟨Int⟩)? ;

token fragment Digit  | [0-9] ;
token fragment Lower  | [a-z] ;
```

- longest match; if same length then *first definition*
- subtokens (Hacs:fragments) must not recurse (no cycles)
- concrete tokens in syntactic sorts are self-defining (Hacs Sec. 3)