BBM 301 – Programming Languages

Lecture 2

Today

- Describing Syntax and Semantics (Chapter 3)
  - How does lexical analyzer work?
  - Describing Syntax
    - Tokens and Lexemes
    - Formal Languages
    - Regular Expressions

Creating computer programs

- Each programming language provides **a set of primitive operations**
- Each programming language provides **mechanisms for combining primitives** to form more complex, but legal, expressions
- Each programming language provides **mechanisms for deducing meanings** or values associated with computations or expressions

E. Grimson, J. Guttag and C. Terman

Aspects of languages

- Primitive constructs
  - ***Programming language***: numbers, strings, simple operators
  - ***English*** : words
- **Syntax**– which strings of characters and symbols are well-formed
  - ***Programming language**:* 3.2 + 3.2 is a valid C expression
  - ***English**:* "cat dog boy" is not syntactically valid, as not in form of acceptable sentence

Aspects of languages

- Static semantics – which syntactically valid strings have a meaning
  - English – "**I are big**" has form <noun> <intransitive verb> <noun>, so syntactically valid, but is not valid English because "I" is singular, "are" is plural
  - Programming language – for example, <literal> <operator> <literal> is a valid syntactic form, but **2.3/"abc"** is a static semantic error!

E. Grimson, J. Guttag and C. Terman

Aspects of languages

- **Semantics** – what is the meaning associated with a syntactically correct string of symbols with no static semantic errors
  - English – can be ambiguous
    - “I cannot recommend this student too highly”
      - "He does not deserve high praise" or
      - "Even the highest praise would be inadequate for him”
    - “Yaşlı adamın yüzüne dalgın dalgın baktı.”
  - Programming languages – always has exactly one meaning
    - But meaning (or value) may not be what programmer intended

Today

- **Syntax:** the form or structure of the expressions, statements, and program units
- **Semantics:** the meaning of the expressions, statements, and program units
- Syntax and semantics provide a language's definition
  - Users of a language definition
    - Other language designers
    - Implementers
    - Programmers (the users of the language)

Example: Syntax and Semantics

- `while` statement in Java
- **syntax:** `while` **(*<boolean_expr>*)**
  <*statement*>
- **semantics:** when ***boolean_expr*** is true, *statement* will be executed

- *The meaning of a statement should be clear from its syntax (Why?)*

Describing Syntax: Terminology

- ***Alphabet:*** Σ, All strings: Σ^*

- A ***sentence*** is a string of characters over some alphabet

- A ***language*** is a set of sentences, $L \subseteq \Sigma^*$

Describing Syntax: Terminology

- A ***language*** is a set of sentences
  - Natural languages: English, Turkish, …
  - Programming languages: C, Fortran, Java,…
  - Formal languages: a^*b^*, 0^n1^n
- String of the language:
  - Sentences
  - Program statements
  - Words (aaaaabb, 000111)

Lexemes

- A ***lexeme*** is the lowest level syntactic unit of a language (e.g., **`*`**, **`sum`**, **`begin`**)
- Lower level constructs are given not by the syntax but by lexical specifications.
- Examples: identifiers, constants, operators, special words.

    `total, sum_of_products, 1254, ++, ( :`

- So, a language is considered as **a set of strings of lexemes** rather than strings of chars.

Token

- A ***token*** of a language is a category of lexemes
- For example, ***identifier*** is a token which may have lexemes, or instances, `sum` and `total`

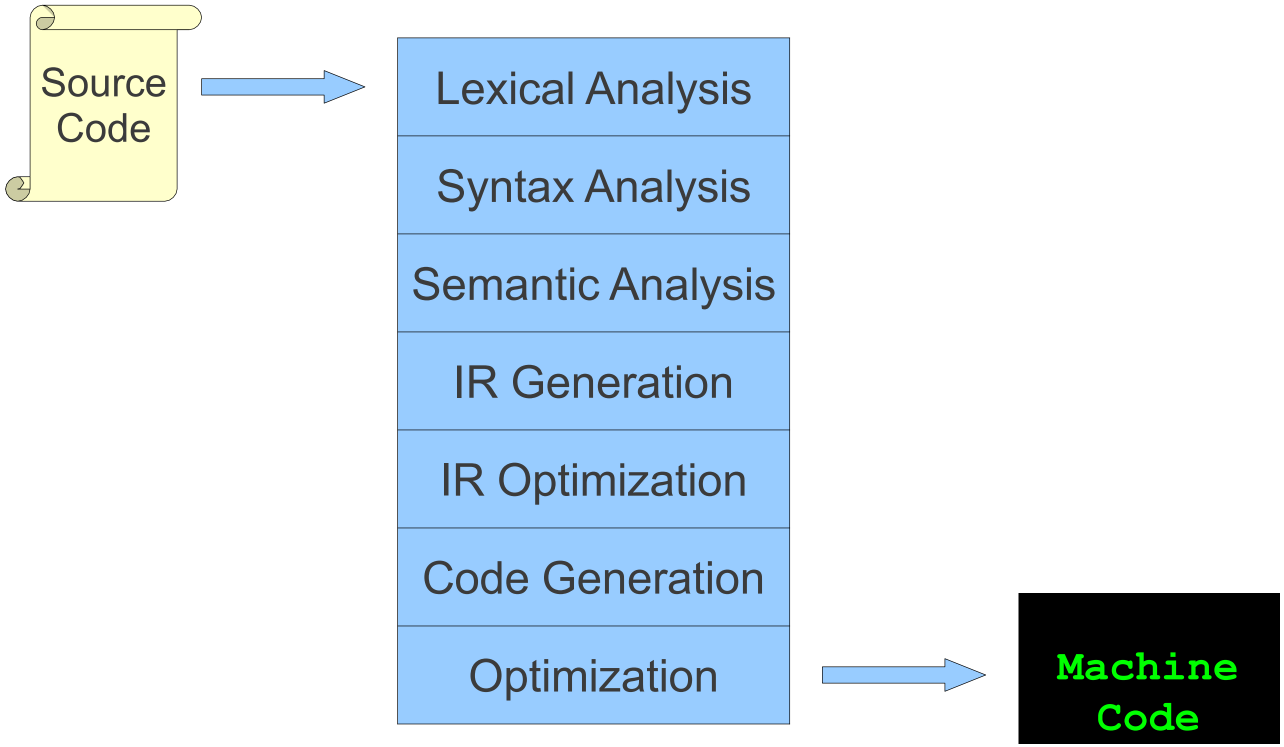
Example in Java Language

`x = (y+3.1) * z_5 ;`

`x = (y+3.1) * z_5;`

Lexemes	*Tokens*
x	identifier
=	equal_sign
(	left_paren
)	right_paren
for	for
y	identifier
+	plus_op
3.1	float_literal
*	mult_op
z_5	identifier
;	semi_colon

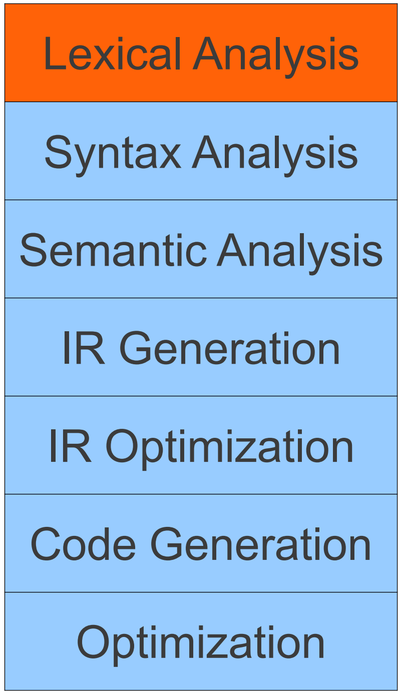
The Structure of a Modern Compiler



```
while (y < z) {
   int x = a + b;
   y += x;
}
```

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- Optimization

```
while (y < z) {
    int x = a + b;
    y += x;
}
```

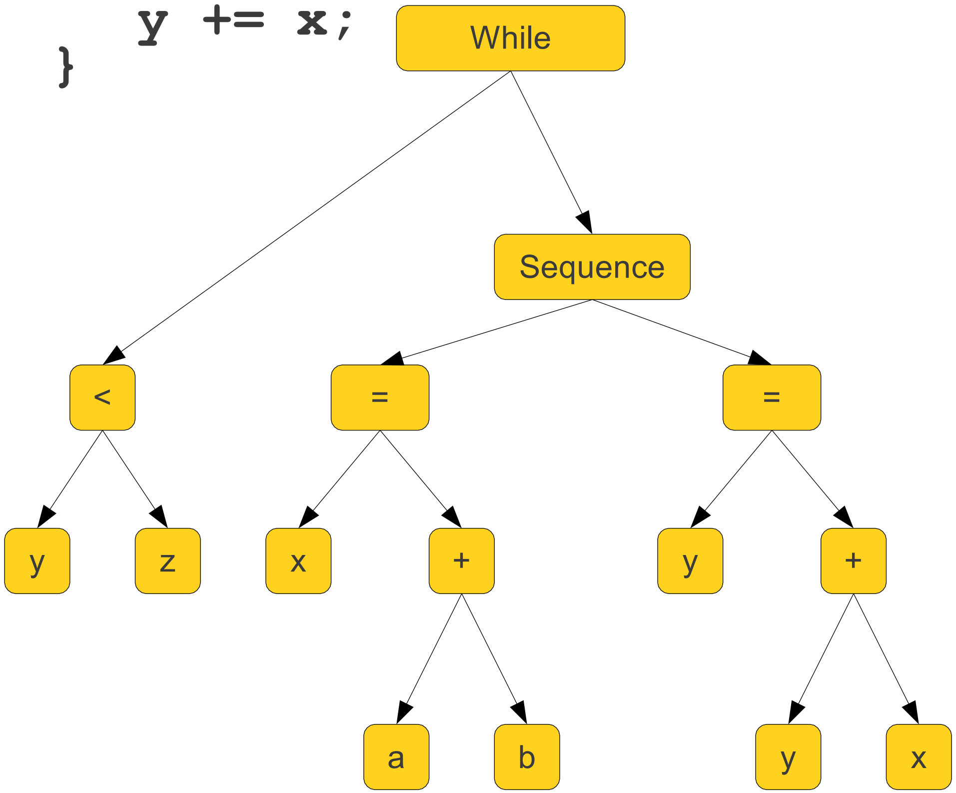


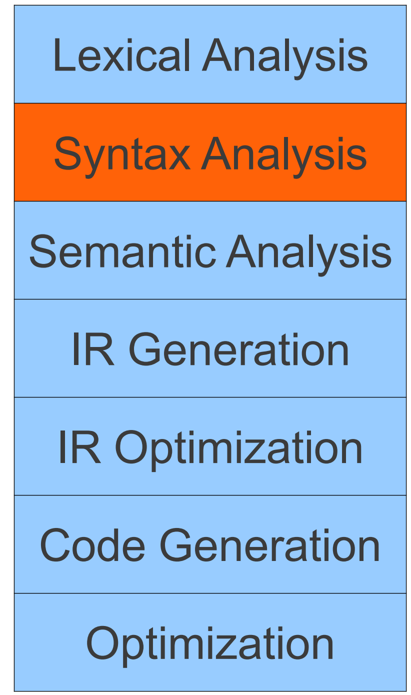
```
while (y < z) {
   int x = a + b;
   y += x;
}
```

```
T_While
T_LeftParen
T_Identifier y
T_Less
T_Identifier z
T_RightParen
T_OpenBrace
T_Int
T_Identifier x
T_Assign
T_Identifier a
T_Plus
T_Identifier b
T_Semicolon
T_Identifier y
T_PlusAssign
T_Identifier x
T_Semicolon
T_CloseBrace
```

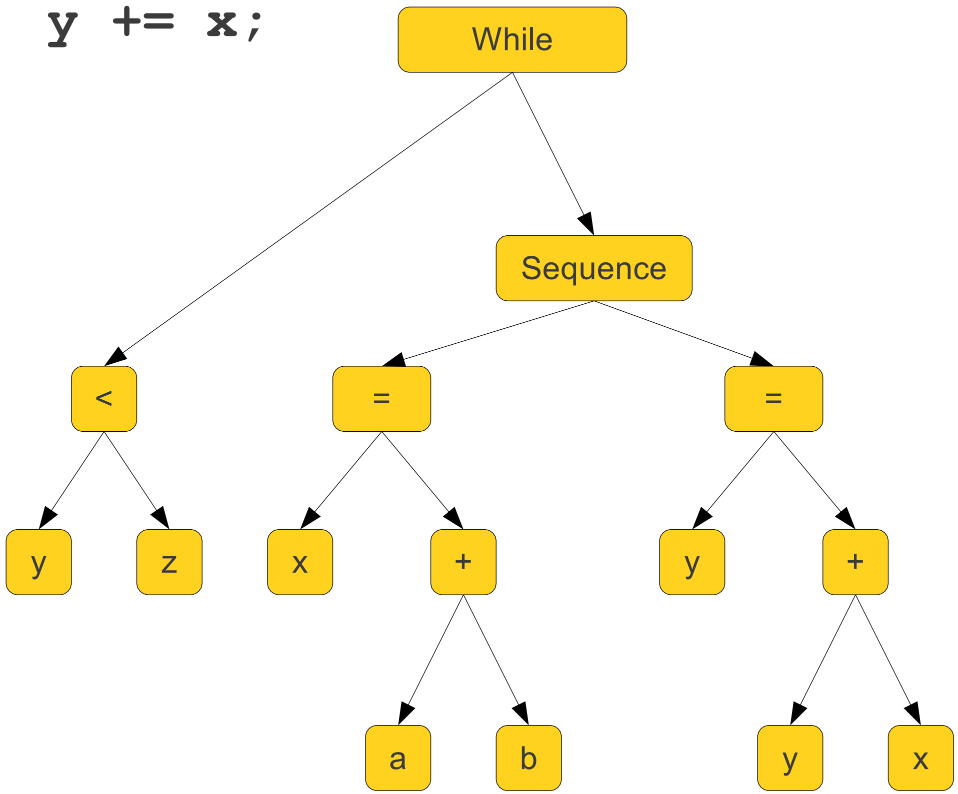
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization

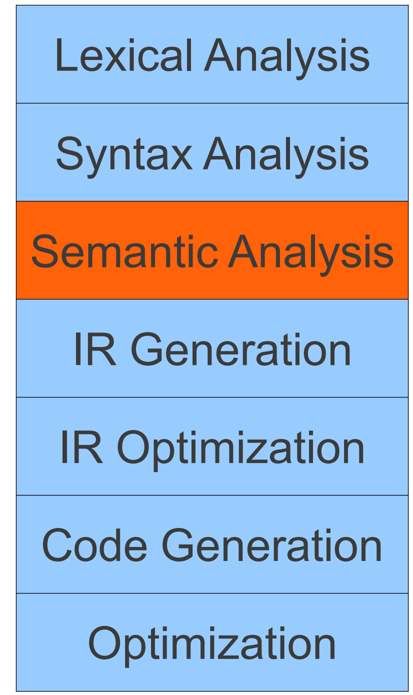
```
while (y < z) {
    int x = a + b;
    y += x;
}
```



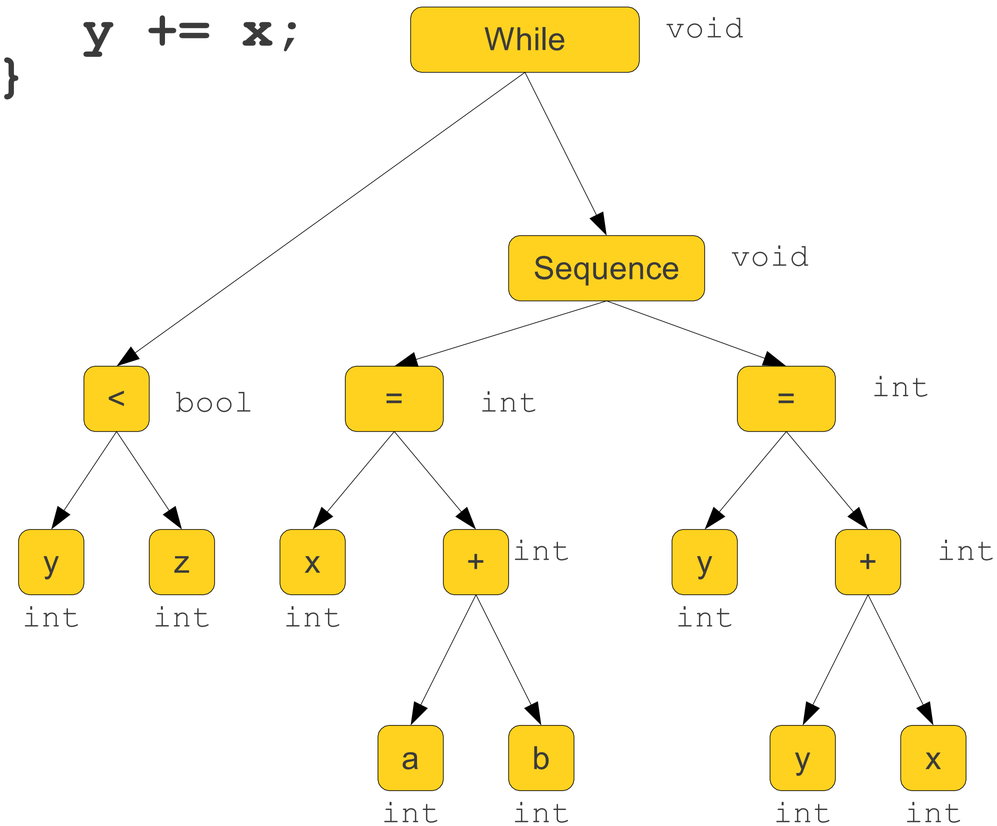


```
while (y < z) {
   int x = a + b;
   y += x;
}
```



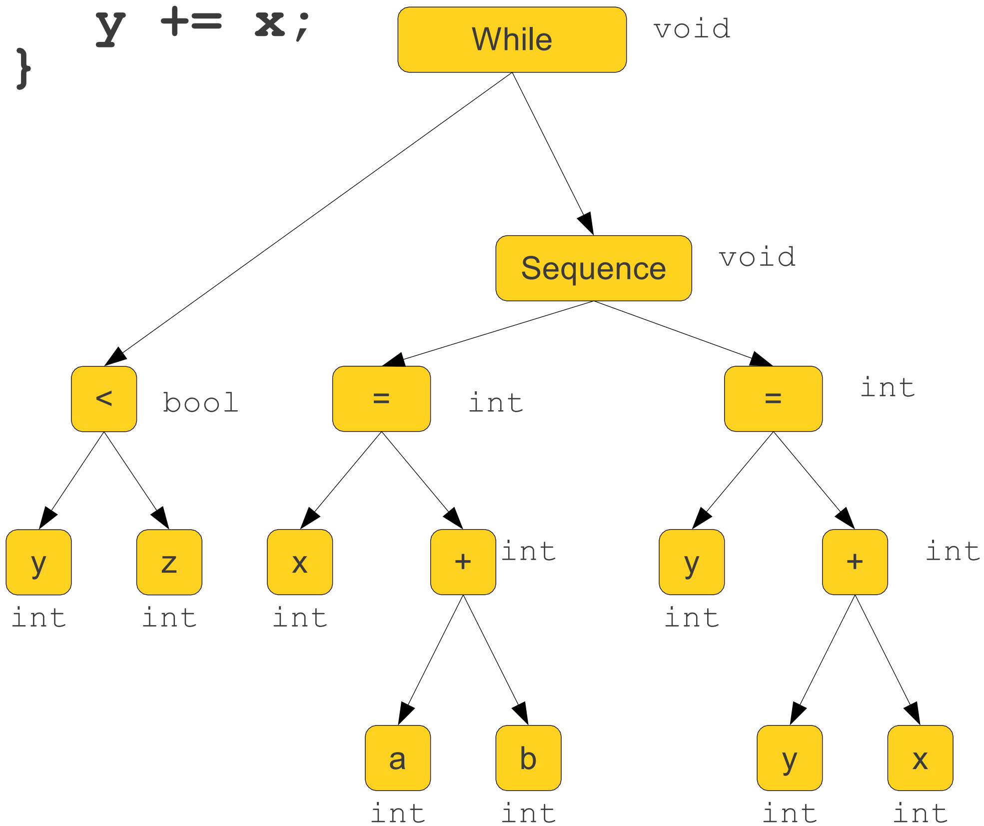


```
while (y < z) {
    int x = a + b;
    y += x;
}
```



- Lexical Analysis
- Syntax Analysis
- **Semantic Analysis**
- IR Generation
- IR Optimization
- Code Generation
- Optimization

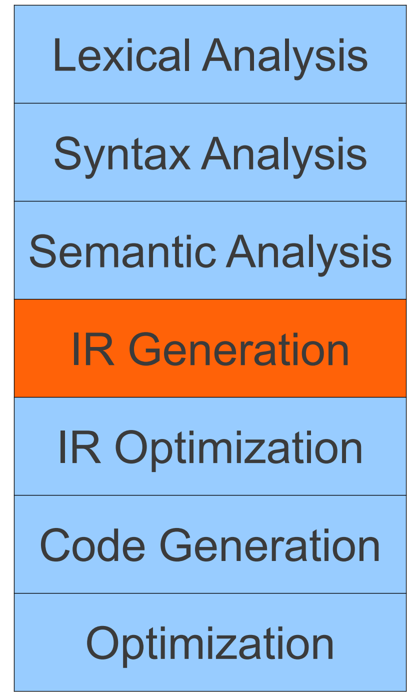
```
while (y < z) {
    int x = a + b;
    y += x;
}
```



- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- Optimization

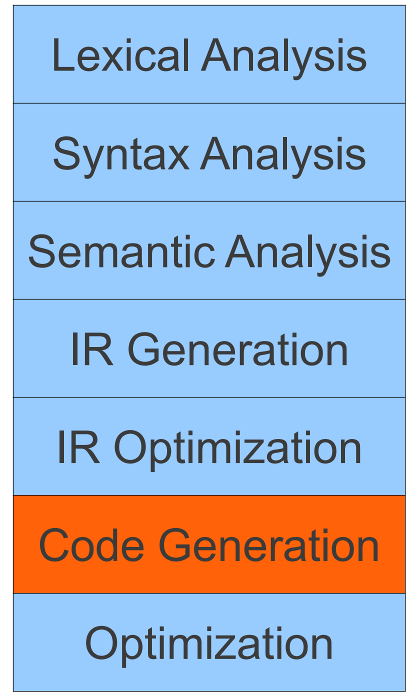
```
while (y < z) {
   int x = a + b;
   y += x;
}
```

```
Loop:    x   =  a  +  b
         y   =  x  +  y
        _t1  =  y  <  z
if _t1 goto Loop
```



```
while (y < z) {
   int x = a + b;
   y += x;
}
```

```
       add  $1,  $2,  $3
Loop:  add  $4,  $1,  $4
       slt  $6,  $1,  $5
       beq  $6,  loop
```

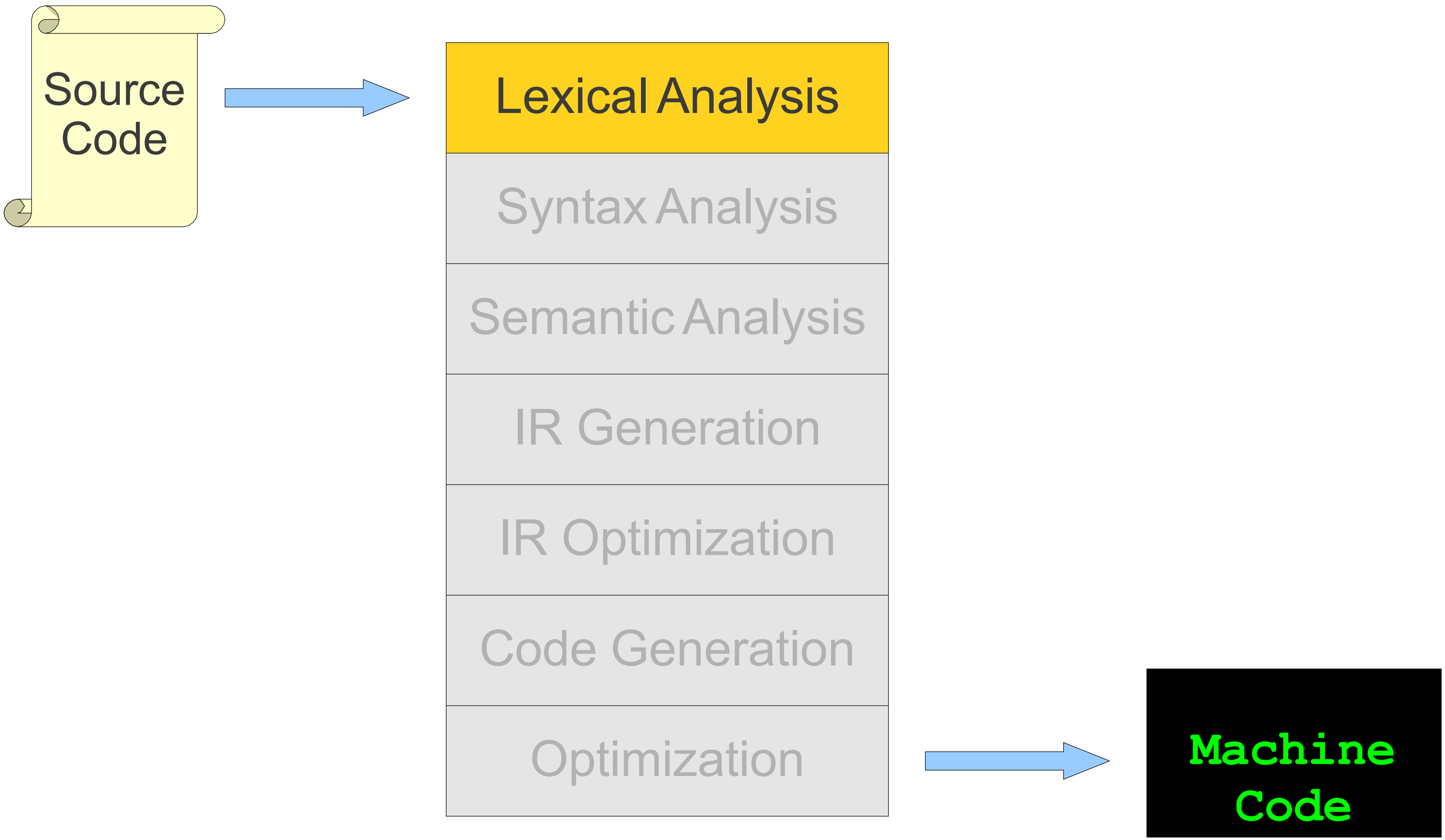


```
while (y < z) {
   int x = a + b;
   y += x;
}
```

```
        add $1, $2, $3
Loop:   add $4, $1, $4
        blt $1, $5, loop
```

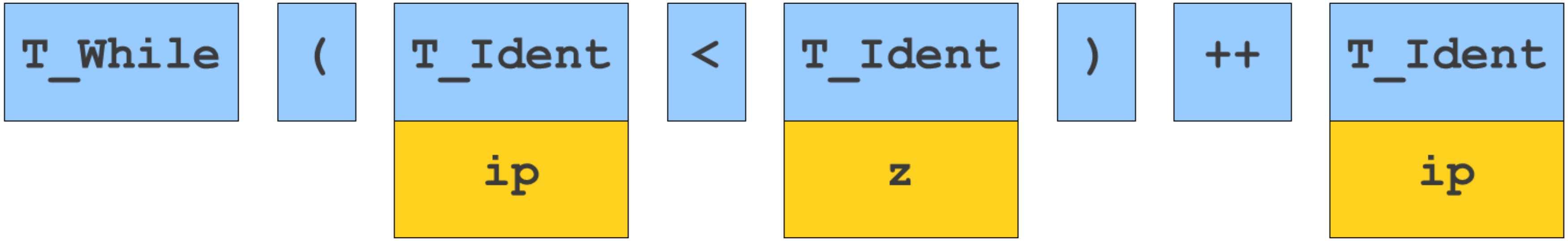
- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- **Optimization**

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

```
while (ip < z)

++ip;
```



w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

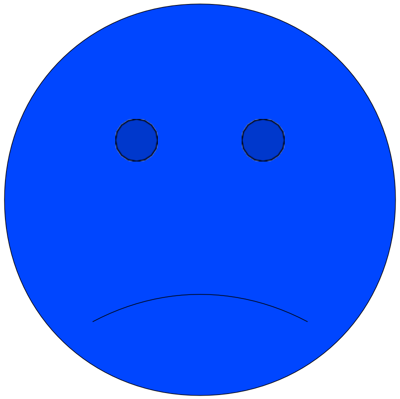
```
while (ip < z)
    ++ip;
```

```
do[for] = new 0;
```

T_Do	[	T_For	]	=	T_New	T_IntConst
						0

d	o	[	f	o	r	]		=		n	e	w		0	;

```
do[for] = new 0;
```



Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

`T_While`

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

The piece of the original program from which we made the token is called a **lexeme**.

`T_While`

A *lexeme* is the lowest level syntactic unit of a language (e.g., `*`, `sum`, `begin`)

This is called a **token**. You can think of it as an enumerated type representing what logical entity we read out of the source code.

A *token* of a language is a category of lexemes

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

`T_While`

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While

Sometimes we will discard a lexeme rather than storing it for later use. Here, we ignore whitespace, since it has no bearing on the meaning of the program.

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While	(

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While	(

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While	(

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While (

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While	(

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While	(

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While	(	T_IntConst
		137

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While

(

T_IntConst

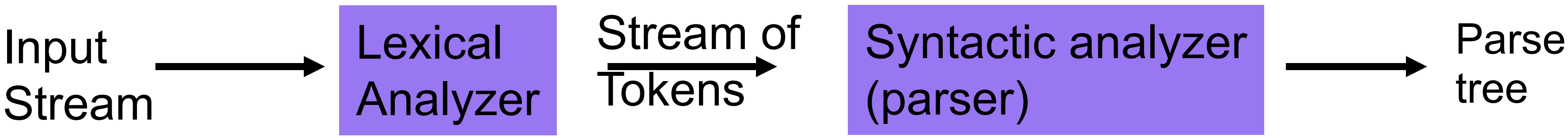
137

Some tokens can have **attributes** that store extra information about the token. Here we store which integer is represented.

Goals of Lexical Analysis

- Convert from physical description of a program into sequence of of **tokens**.
  - Each token represents one logical piece of the source file – a keyword, the name of a variable, etc.
- Each token is associated with a **lexeme**.
  - The actual text of the token: “137,” “int,” etc.
- Each token may have optional **attributes**.
  - Extra information derived from the text – perhaps a numeric value.
- The token sequence will be used in the parser to recover the program structure.

Lexical and syntactic analysis



- **Lexical analyzer:** scans the input stream and converts sequences of characters into tokens.
  (char list) → (token list)
- `Lex` is a tool for writing lexical analyzers.

- **Syntactic Analysis:** reads tokens and assembles them into language constructs using the grammar rules of the language.
- `Yacc` is a tool for constructing parsers.

Lexical Analysis

- What do we want to do? Example:

```
if (i == j)
    z = 0;
else
    z = 1;
```

- The input is just a sequence of characters:

```
if (i == j)\n\tz = 0;\nelse\n\tz = 1;
```

- **Goal:** Partition input strings into substrings
  - And classify them according to their role

Slide credit: Wes Weimer

Implementation of A Lexical Analyzer

- The lexer usually discards **uninteresting** tokens that don't contribute to parsing.
- Examples: Whitespaces, Comments
  - Exception: which language cares about whitespaces?
- The goal is to partition the string. That is implemented by reading left-to-right, recognizing one token at a time.
- Lexical structure described can be specified using ***regular expressions.***

Choosing Tokens

What Tokens are Useful Here?

```
for (int k = 0; k < myArray[5]; ++k) {
    cout << k << endl;
}
```

What Tokens are Useful Here?

```
for (int k = 0; k < myArray[5]; ++k) {
    cout << k << endl;
}
```

`for`	`{`
`int`	`}`
`<<`	`;`
`=`	`<`
`(`	`[`
`)`	`]`
`++`	

What Tokens are Useful Here?

```
for (int k = 0; k < myArray[5]; ++k) {
    cout << k << endl;
}
```

```
for        {
int        }
<<         ;
=          <
(          [
)          ]
++

Identifier
IntegerConstant
```

Choosing Good Tokens

- Very much dependent on the language.
- Typically:
  - Give keywords their own tokens.
  - Give different punctuation symbols their own tokens.
  - Group lexemes representing identifiers, numeric constants, strings, etc. into their own groups.
  - Discard irrelevant information (whitespace, comments)

Challenges in Scanning

- How do we determine which lexemes are associated with each token?
- When there are multiple ways we could scan the input, how do we know which one to pick?
- How do we address these concerns efficiently?

Associating Lexemes with Tokens

Sets of Lexemes

- **Idea:** Associate a set of lexemes with each token.
  - We might associate the "**number**" token with the set **{ 0, 1, 2, … , 10, 11, 12, … }**
  - We might associate the "**string**" token with the set **{ "", "a", "b", "c", … }**
  - We might associate the token for the keyword **while** with the set **{ while}** .

How do we describe which (potentiallly infinite) set of lexemes is associated with each token type?

Formal Languages

- A **formal language** is a set of strings.
- Many infinite languages have finite descriptions:
  - Define the language using an automaton.
  - Define the language using a grammar.
  - Define the language using a regular expression.
- We can use these compact descriptions of the language to define sets of strings.

Regular Expressions

- Regular expressions are a family of descriptions that can be used to capture certain languages (the *regular languages*).
- Often provide a compact and human-readable description of the language.
- Used as the basis for numerous software systems, including the lex/flex tool.

Regular Expressions

In computing, a **regular expression**, also referred to as "regex" or "regexp", provides a concise and flexible means for **matching strings of text**, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a **regular expression processor**.

http://en.wikipedia.org/wiki/Regular_expression

Regular Expressions

- Regular expressions are used in many programming languages and software tools to specify patterns and match strings.
- Regular expressions are well suited for matching lexemes in programming languages.
- Regular expressions use a finite alphabet of symbols and defined by the operators
  - (i) union
  - (ii) concatenation
  - (iii) Kleene closure.

Slide credit: Wes Weimer

Designing patterns

Designing the proper patterns can be very tricky, but you are provided with a broad range of options for your regular expressions.

- .   A dot will match any single character except a newline.
- *,+ Star and plus used to match zero/one or more of the preceding expressions.
- ?   Matches zero or one copy of the preceding expression.

Designing patterns

- | A logical ‘or’ statement - matches either the pattern before it, or the pattern after.
- ^ Matches the very beginning of a line.
- $ Matches the end of a line.
- / Matches the preceding regular expression, but only if followed by the subsequent expression.

Designing patterns

- [ ]   Brackets are used to denote a character class, which matches any single character within the brackets. If the first character is a ‘^’, this negates the brackets causing them to match any character except those listed. The ‘-’ can be used in a set of brackets to denote a range.
- “ ”   Match everything within the quotes literally - don’t use any special meanings for characters.
- ( )   Group everything in the parentheses as a single unit for the rest of the expression.

Regular expressions

- **a** matches **a**
- **abc** matches **abc**
- **[abc]** matches **a, b or c**
- **[a-f]** matches **a, b, c, d, e, or f**
- **[0-9]** matches any digit
- **X+** matches one or more of **X**
- **X*** matches zero or more of **X**
- **[0-9]+** matches any integer
- **(…)** grouping an expression into a single unit
- **|** alternation (or)
- **(a|b|c)*** is equivalent to **[a-c]***

Regular expressions

- **`X?`** **`X`** is optional (0 or 1 occurrence)
- **`if(def)?`** matches **`if`** or **`ifdef`** (equivalent to **`if|ifdef`**)
- **`[A-Za-z]`** matches any alphabetical character
- **`.`** matches any character except newline character
- **`\.`** matches the **`.`** character
- **`\n`** matches the newline character
- **`\t`** matches the tab character
- **`\\`** matches the **`\`** character
- **`[ \t]`** matches either a space or tab character
- **`[^a-d]`** matches any character other than **`a`**,**`b`**,**`c`** and **`d`**

Atomic Regular Expressions

- The regular expressions we will use in this course begin with two simple building blocks.
- The symbol **ε** is a regular expression matches the empty string.
- For any symbol `a`, the symbol **`a`** is a regular expression that just matches `a`.

Operator Precedence

- Regular expression operator precedence is

$$(R)$$

$$R^*$$

$$R_1 R_2$$

$$R_1 \mid R_2$$

- So **ab*c|d** is parsed as **((a(b*))c)|d**

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings containing `00` as a substring:

(0 | 1)*00(0 | 1)*

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings containing `00` as a substring:

(0 | 1)*00(0 | 1)*

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings containing `00` as a substring:

(0 | 1)*00(0 | 1)*

11011100101
0000
11111011110011111

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings containing `00` as a substring:

(0 | 1)*00(0 | 1)*

11011100101
0000
111110111100111111

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings of length exactly four:

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings of length exactly four:

(0|1)(0|1)(0|1)(0|1)

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings of length exactly four:

(0|1)(0|1)(0|1)(0|1)

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings of length exactly four:

(0|1)(0|1)(0|1)(0|1)

0000
1010
1111
1000

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings of length exactly four:

(0|1)(0|1)(0|1)(0|1)

0000
1010
1111
1000

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings of length exactly four:

(0|1){4}

0000
1010
1111
1000

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings of length exactly four:

(0|1){4}

0000
1010
1111
1000

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings that contain at most one zero:

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings that contain at most one zero:

$$1^*(0 \mid \varepsilon)1^*$$

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings that contain at most one zero:

$$1^*(0 \mid \varepsilon)1^*$$

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings that contain at most one zero:

1*(0 | ε)1*

11110111
111111
0111
0

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings that contain at most one zero:

$$1^*(0 \mid \varepsilon)1^*$$

11110111
111111
0111
0

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings that contain at most one zero:

1*0?1*

11110111
111111
0111
0

Applied Regular Expressions

- Suppose our alphabet is `a`, `@`, and `.`, where `a` represents "some letter."
- A regular expression for email addresses is

aa* (.aa*)* @ aa*.aa* (.aa*)*

Applied Regular Expressions

- Suppose our alphabet is **a**, **@**, and **.**, where **a** represents "some letter."
- A regular expression for email addresses is

aa* (.aa*)* @ aa*.aa* (.aa*)*

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose our alphabet is **a**, **@**, and **.**, where **a** represents "some letter."
- A regular expression for email addresses is

aa* (.aa*)* @ aa*.aa* (.aa*)*

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose our alphabet is **`a`**, **`@`**, and **`.`**, where **`a`** represents "some letter."
- A regular expression for email addresses is

$$a^+ \ (.aa^*)^* \ @ \ aa^*.aa^* \ (.aa^*)^*$$

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose our alphabet is **a**, **@**, and **.**, where **a** represents "some letter."
- A regular expression for email addresses is

$a^+ \quad (.a^+)^* \quad @ \quad a^+.a^+ \quad (.a^+)^*$

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose our alphabet is **a**, **@**, and **.**, where **a** represents "some letter."
- A regular expression for email addresses is

$$a^+ \quad (.a^+)^* \quad @ \quad a^+.a^+ \quad (.a^+)^*$$

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose our alphabet is **`a`**, **`@`**, and **`.`**, where **`a`** represents "some letter."
- A regular expression for email addresses is

$$\mathbf{a^{+}} \quad \mathbf{(.a^{+})^{*}} \quad \mathbf{@} \quad \mathbf{a^{+}} \quad \mathbf{(.a^{+})^{+}}$$

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose our alphabet is `a`, `@`, and `.`, where `a` represents "some letter."
- A regular expression for email addresses is

$$a^+(.a^+)^*@a^+(.a^+)^+$$

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose that our alphabet is all ASCII characters.
- A regular expression for even numbers is

$$(+|-)?(0|1|2|3|4|5|6|7|8|9)^*(0|2|4|6|8)$$

Applied Regular Expressions

- Suppose that our alphabet is all ASCII characters.
- A regular expression for even numbers is

(+|-)?(0|1|2|3|4|5|6|7|8|9)*(0|2|4|6|8)

42
+1370
-3248
-9999912

Applied Regular Expressions

- Suppose that our alphabet is all ASCII characters.
- A regular expression for even numbers is

(+|-)?(0|1|2|3|4|5|6|7|8|9)*(0|2|4|6|8)

42
+1370
-3248
-9999912

Applied Regular Expressions

- Suppose that our alphabet is all ASCII characters.
- A regular expression for even numbers is

(+|-)?[0123456789]*[02468]

42
+1370
-3248
-9999912

Applied Regular Expressions

- Suppose that our alphabet is all ASCII characters.
- A regular expression for even numbers is

(+|-)?[0-9]*[02468]

42
+1370
-3248
-9999912

Regular Expressions

[a-z]	any letter a through z
[a\-z]	one of: a - z
[-az]	one of: - a z

<table>
<tr><td>[^ab]</td><td>anything except: a b</td></tr>
<tr><td>[a^b]</td><td>one of: a ^ b</td></tr>
<tr><td>[a|b]</td><td>one of: a | b</td></tr>
</table>

Examples

Real numbers, e.g., 0, 27, 2.10, .17

```
[0-9]*(\.)?[0-9]+
```

To include an optional preceding sign:

```
[+-]?[0-9]*(\.)?[0-9]+
```

Integer or floating point number

```
[0-9]+(\.[0-9]+)?
```

Integer, floating point, or scientific notation.

```
[+-]?[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)?
```

Expanded Regex Syntax

operation	example	matches	does not match
any character (except newline)	`.U.U.U.`	`CUMULUS` `JUGULUM`	`SUCCUBUS` `TUMULTUOUS`
character class	`[A-Za-z][a-z]*`	`word` `Capitalized`	`camelCase` `4illegal`
at least one	`jo+hn`	`john` `joooooohn`	`jhn` `jjohn`
zero or one	`joh?n`	`jon` `john`	`any other string`
repeated exactly {a} times	`j[aeiou]{3}hn`	`jaoehn` `jooohn`	`jhn` `jaeiouhn`
repeated from a to b times: {a,b}	`j[ou]{1,2}hn`	`john` `juohn`	`jhn` `jooohn`

More Regular Expression Examples

regex	matches	does not match
.*SPB.*	RASPBERRY CRISPBREAD	SUBSPACE SUBSPECIES
[0-9]{3}-[0-9]{2}-[0-9]{4}	231-41-5121 573-57-1821	231415121 57-3571821
[a-z]+@([a-z]+\.)+(edu|com)	horse@pizza.com horse@pizza.food.com	frank_99@yahoo.com hug@cs

Even More Regular Expression Syntax

operation	example	matches	does not match
built-in character classes	`\w+` `\d+`	`fawef` `231231`	`this person` `423 people`
character class negation	`[^a-z]+`	`PEPPERS3982` `17211!↑å`	`porch` `CLAmS`
escape character	`cow\.com`	`cow.com`	`cowscom`

Suppose you want to match one of our special characters like . or [or]

- In these cases, you must “escape” the character using the backslash.
- You can think of the backslash as meaning “take this next character literally”.

Even More Regular Expression Features

operation	example	matches	does not match
beginning of line	`^ark`	`ark two` `ark o ark`	`dark`
end of line	`ark$`	`dark` `ark o ark`	`ark two`

A few additional common regex features are listed above.

- There are even more out there!

Challenges in Scanning

- How do we determine which lexemes are associated with each token?
- When there are multiple ways we could scan the input, how do we know which one to pick?
- How do we address these concerns efficiently?

Lexing Ambiguities

`T_For`	`for`
`T_Identifier`	`[A-Za-z_][A-Za-z0-9_]*`

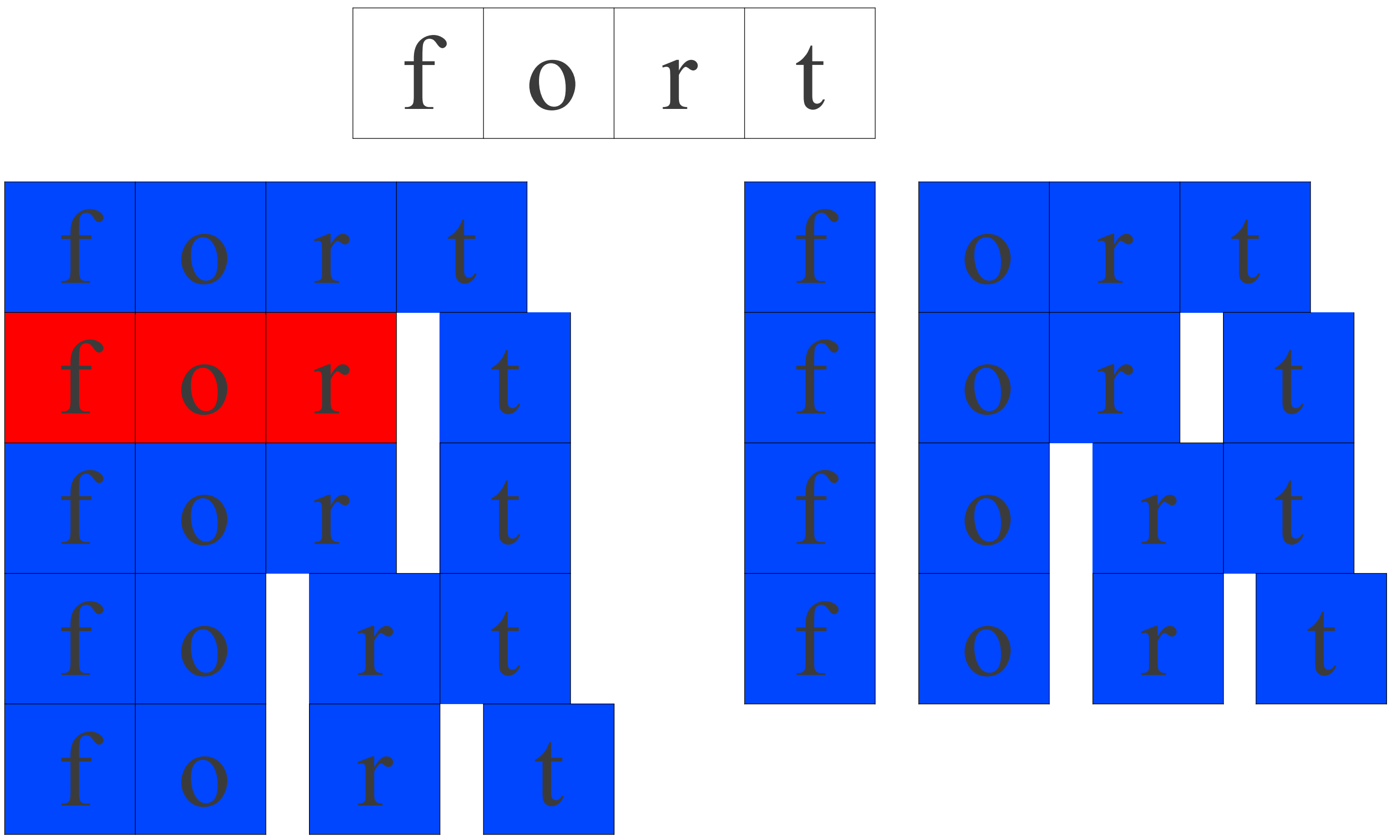
Lexing Ambiguities

`T_For`	`for`
`T_Identifier`	`[A-Za-z_][A-Za-z0-9_]*`

f	o	r	t

Lexing Ambiguities

```
T_For          for
T_Identifier   [A-Za-z_][A-Za-z0-9_]*
```



Conflict Resolution

- Assume all tokens are specified as regular expressions.
- Algorithm: Left-to-right scan.
- Tiebreaking rule one: Maximal munch.
  - Always match the longest possible prefix of the remaining text.

Lexing Ambiguities

```
T_For             for
T_Identifier      [A-Za-z_][A-Za-z0-9_]*
```

f	o	r	t

f	o	r	t

Other Conflicts

```
T_Do         do
T_Double     double
T_Identifier [A-Za-z_][A-Za-z0-9_]*
```

Other Conflicts

```
T_Do         do
T_Double     double
T_Identifier [A-Za-z_][A-Za-z0-9_]*
```

d	o	u	b	l	e

Other Conflicts

```
T_Do         do
T_Double     double
T_Identifier [A-Za-z_][A-Za-z0-9_]*
```

d	o	u	b	l	e

d	o	u	b	l	e
d	o	u	b	l	e

More Tiebreaking

- When two regular expressions apply, choose the one with the greater "priority."
- Simple priority system: **pick the rule that was defined first.**

Other Conflicts

```
T_Do         do
T_Double     double
T_Identifier [A-Za-z_][A-Za-z0-9_]*
```

d	o	u	b	l	e

d	o	u	b	l	e

Other Conflicts

```
T_Do         do
T_Double     double
T_Identifier [A-Za-z_][A-Za-z0-9_]*
```

d	o	u	b	l	e

d	o	u	b	l	e

Why isn't this a problem?

One Last Detail...

- We know what to do if *multiple* rules match.
- What if *nothing* matches?
- Trick: Add a “catch-all” rule that matches any character and reports an error.

Extra Slides

Real-World Scanning:

Python

Python Blocks

- Scoping handled by whitespace:

```
if w == z:
    a = b
    c = d
else:
    e = f
g = h
```

- What does that mean for the scanner?

Whitespace Tokens

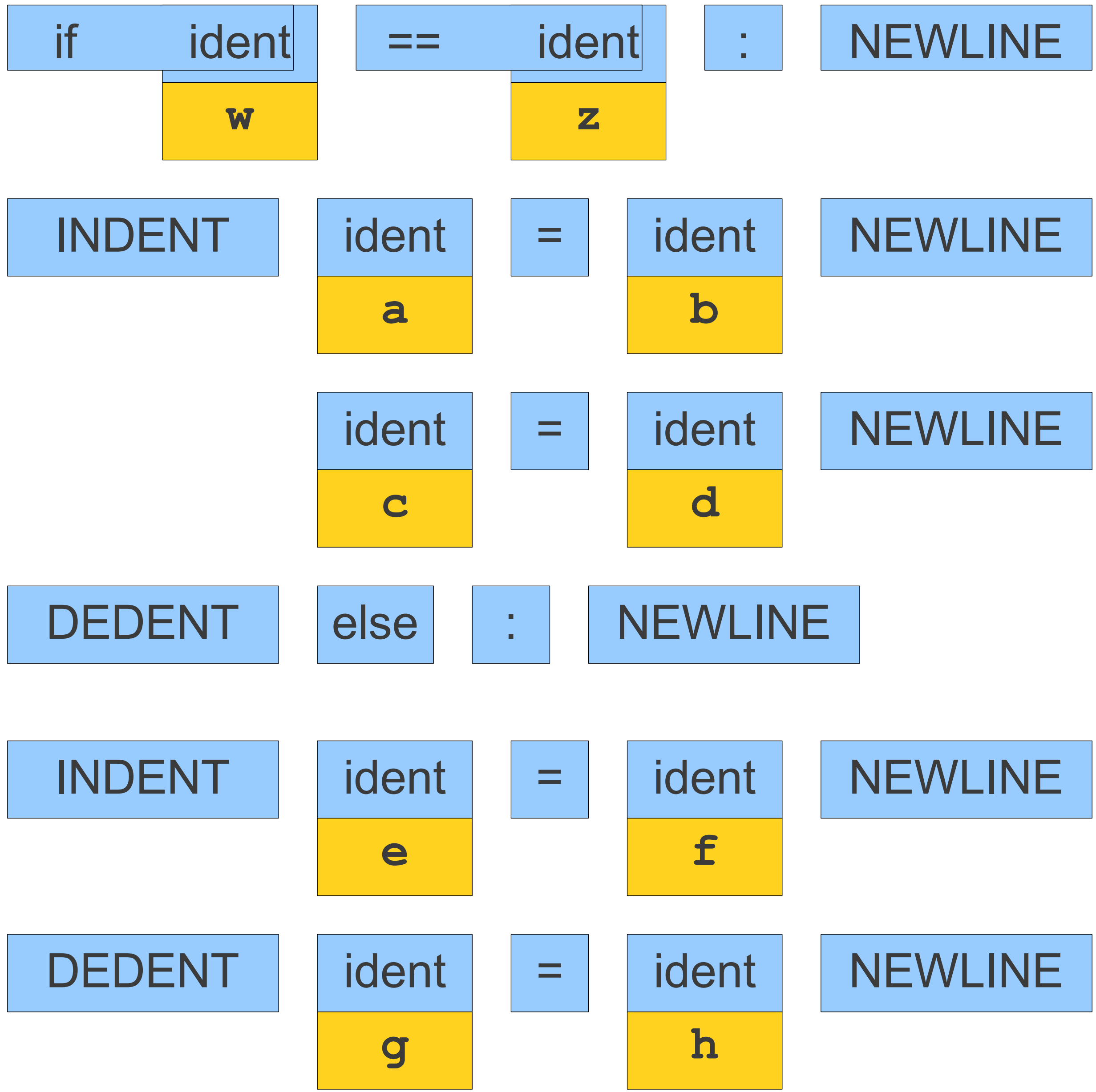
- Special tokens inserted to indicate changes in levels of indentation.
- **NEWLINE** marks the end of a line.
- **INDENT** indicates an increase in indentation.
- **DEDENT** indicates a decrease in indentation.
- Note that INDENT and DEDENT encode *change* in indentation, not the total amount of indentation.

Scanning Python

```
if w == z:
    a = b
    c = d
else:
    e = f
g = h
```

Scanning Python

```
if w == z:
    a = b
    c = d
else:
    e = f
g = h
```



Scanning Python

```
if w == z: {
       a = b;
       c = d;
} else {
       e = f;
}
g = h;
```

if	ident	==	ident	:	NEWLINE
	w		z		
INDENT	ident	=	ident	NEWLINE	
	a		b		
	ident	=	ident	NEWLINE	
	c		d		
DEDENT	else	:	NEWLINE		
INDENT	ident	=	ident	NEWLINE	
	e		f		
DEDENT	ident	=	ident	NEWLINE	
	g		h		

Scanning Python

```
if w == z: {
    a = b;
    c = d;
} else {
    e = f;
}
g = h;
```