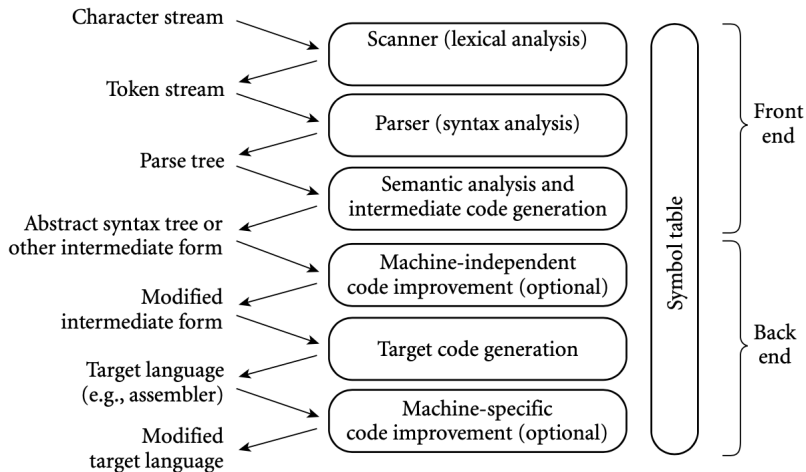


# Programming Languages

Janyl Jumadinova

February 6-10, 2023

# Most Important Steps in Compilation



# Lexical Analysis

**Lexical analysis** produces a “token stream” in which the program is reduced to a sequence of token types, each with its identifying number and the actual string (in the program) corresponding to it.

# Lexical Analysis

**For each token type, give a description:**

- either a literal string
  - “ $\leq$ ” or “while” to describe an operator or reserved word,

# Lexical Analysis

## For each token type, give a description:

- either a literal string
  - “ $\leq$ ” or “while” to describe an operator or reserved word,
- or a *< rule >*
  - a rule *< unsigned int >* example: “a sequence of one or more digits”;
  - a rule *< identifier >* example: “a letter followed by a sequence of zero or more letters or digits.”

# Typical Tokens in Programming Languages

- *Operators and Punctuation*

- `+ - * / ( ) [ ] ; : :: < <= == = != ! ...!`

- *Keywords*

- `if while for goto return switch void ...`

- *Identifiers (variables)*

- *Integer constants*

- Other constants (string, floating point, boolean, ...), etc.

# Lexical Complications

- Most modern languages are free-form
  - Layout doesn't matter
  - White space separates tokens
- Alternatives
  - Haskell, Python - indentation and layout can imply grouping

# Syntactic Analysis

- The syntax of a language is described by a **grammar** that specifies the legal combinations of tokens.



# Syntactic Analysis

- The syntax of a language is described by a **grammar** that specifies the legal combinations of tokens.
- Grammars are often specified in BNF notation (“Backus Naur Form”):

# Syntactic Analysis

- The syntax of a language is described by a **grammar** that specifies the legal combinations of tokens.
- Grammars are often specified in BNF notation (“Backus Naur Form”):

`<item1> ::= valid replacements for <item1>`

`<item2> ::= valid replacements for <item2>`

# Alternative Notations

- There are several syntax notations for productions in common use; all mean the same thing. E.g.:

`ifStmt ::= if ( expr ) statement`

`ifStmt → if ( expr ) statement`

`<ifStmt> ::= if ( <expr> ) <statement>`

## Example: Grammar for Pigese (or Pigish?)

- A formal grammar for a “pig language” could be:  
PigTalk ::= oink PigTalk (Rule 1)  
          | oink! (Rule 2)

# Example: Grammar for Pigese (or Pigish?)

- A formal grammar for a “pig language” could be:  
PigTalk ::= oink PigTalk (Rule 1)  
          | oink! (Rule 2)
- PigTalk can then generate, for example:  
    ① PigTalk ::= oink! (Rule 2)

## Example: Grammar for Pigese (or Pigish?)

- A formal grammar for a “pig language” could be:

PigTalk ::= oink PigTalk (Rule 1)  
          | oink! (Rule 2)

- PigTalk can then generate, for example:

- ① PigTalk ::= oink! (Rule 2)
- ② PigTalk ::= oink PigTalk (Rule 1)  
   ::= oink oink!

# Example: Grammar for Pigese (or Pigish?)

- A formal grammar for a “pig language” could be:

PigTalk ::= oink PigTalk (Rule 1)  
          | oink! (Rule 2)

- PigTalk can then generate, for example:

- ① PigTalk ::= oink! (Rule 2)
- ② PigTalk ::= oink PigTalk (Rule 1)  
      ::= oink oink!
- ③ PigTalk ::= oink PigTalk (Rule 1)  
      ::= oink oink PigTalk (Rule 1)  
      ::= oink oink oink! (Rule 2)

# Grammars (Context-free Grammars)

- Collection of VARIABLES (things that can be replaced by other things), also called NON-TERMINALS.



# Grammars (Context-free Grammars)

- Collection of VARIABLES (things that can be replaced by other things), also called NON-TERMINALS.
- Collection of TERMINALS (“constants”, strings that can't be replaced)

# Grammars (Context-free Grammars)

- Collection of VARIABLES (things that can be replaced by other things), also called NON-TERMINALS.
- Collection of TERMINALS (“constants”, strings that can't be replaced)
- One special variable called the START SYMBOL.

# Grammars (Context-free Grammars)

- Collection of VARIABLES (things that can be replaced by other things), also called NON-TERMINALS.
- Collection of TERMINALS (“constants”, strings that can't be replaced)
- One special variable called the START SYMBOL.
- Collection of RULES, also called PRODUCTIONS.

# Grammars (Context-free Grammars)

- Collection of VARIABLES (things that can be replaced by other things), also called NON-TERMINALS.
- Collection of TERMINALS (“constants”, strings that can't be replaced)
- One special variable called the START SYMBOL.
- Collection of RULES, also called PRODUCTIONS.

`variable → rule1 | rule2 | rule3 | ...`

You can also write each rule on a separate line

# Grammars (Context-free Grammars): EXAMPLE

## Grammar

A, B, and C are non-terminals.

0, 1, and 2 are terminals.

The start symbol is A.

The rules are:

- $A \rightarrow 0A|1C|2B|0$
- $B \rightarrow 0B|1A|2C|1$
- $C \rightarrow 0C|1B|2A|2$

# Grammars (Context-free Grammars): EXAMPLE

## Grammar

A, B, and C are non-terminals.

0, 1, and 2 are terminals.

The start symbol is A.

The rules are:

- $A \rightarrow 0A|1C|2B|0$
- $B \rightarrow 0B|1A|2C|1$
- $C \rightarrow 0C|1B|2A|2$

Can 2011020 be parsed?

# Grammars (Context-free Grammars): ACTIVITY

## Grammar

A, B, and C are non-terminals.

0, 1, and 2 are terminals.

The start symbol is A, the rules are:

- $A \rightarrow 0A|1C|2B|0$
- $B \rightarrow 0B|1A|2C|1$
- $C \rightarrow 0C|1B|2A|2$

<https://itempool.com/jjumadinova/live>

# Grammars (Context-free Grammars): ACTIVITY

## Grammar

A, B, and C are non-terminals.

0, 1, and 2 are terminals.

The start symbol is A, the rules are:

- $A \rightarrow 0A|1C|2B|0$
- $B \rightarrow 0B|1A|2C|1$
- $C \rightarrow 0C|1B|2A|2$

<https://itepool.com/jjumadinova/live>

Can 1112202 be parsed?

Can 00102 be parsed?

Can 2121 be parsed?



# Regular Expressions used for Scanning

- Defined over some alphabet  $\Sigma$ .
  - For programming languages, alphabet is usually ASCII or Unicode.
- If `re` is a regular expression,  $L(\text{re})$  is the language (set of strings) generated by `re`.

# Fundamentals of Regular Expressions (REs)

- These are the basic building blocks that other REs are built from.

$re$	$L(re)$	Notes
$a$	$\{ a \}$	Singleton set, for each symbol $a$ in the alphabet $\Sigma$
$\varepsilon$	$\{ \varepsilon \}$	Empty string
$\emptyset$	$\{ \}$	Empty language

# Operations on REs

$re$	$L(re)$	Notes
$rs$	$L(r)L(s)$	Concatenation – $r$ followed by $s$
$r s$	$L(r) \cup L(s)$	Combination (union) – $r$ or $s$
$r^*$	$L(r)^*$	0 or more occurrences of $r$ (Kleene closure)

# Operations on REs

$re$	$L(re)$	Notes
$rs$	$L(r)L(s)$	Concatenation – $r$ followed by $s$
$r s$	$L(r) \cup L(s)$	Combination (union) – $r$ or $s$
$r^*$	$L(r)^*$	0 or more occurrences of $r$ (Kleene closure)

- Precedence:  $(R)$ ,  $R^*$ ,  $R_1R_2$ ,  $R_1|R_2$  (lowest).
- Parenthesis can be used to group REs as needed.

# Examples

<i>re</i>	Meaning
+	single + character
!	single ! character
!=	2 character sequence "!="
xyzyy	5 character sequence "xyzyy"
(1 0)*	Zero or more binary digits
(1 0)(1 0)*	Binary constant (possible leading 0s)
0 1(1 0)*	Binary constant without extra leading 0s, i.e, 0 or starts with 1 (  has lowest precedence)

# Abbreviations on REs

- There are common abbreviations used for convenience.

Abbr.	Meaning	Notes
$r^+$	$(rr^*)$	1 or more occurrences
$r?$	$(r \mid \varepsilon)$	0 or 1 occurrence
$[a-z]$	$(a b \dots z)$	1 character in given range
$[abxyz]$	$(a b x y z)$	1 of the given characters

# Example

- Possible syntax for numeric constants

`digit ::= [0-9]`

`digits ::= digit +`

`number ::= digits ( . digits )?`

`([eE] (+ | -)? digits )?`

- Notice that this allows (unnecessary) leading 0s, e.g., 00045.6. (0, or 0.14 would be necessary 0s).

# Example

- **Possible syntax for numeric constants**

`digit ::= [0-9]`

`nonzero_digit ::= [1-9]`

`digits ::= digit +`

`number ::= (0 | nonzero_digit digits?)`

`( . digits )?`

`([eE] (+ | -)? digits )?`



# RE Practice:

`https://regexone.com/`