Programming Languages

Janyl Jumadinova

September 10-15, 2020
Most Important Steps in Compilation

- Character stream
  - Token stream
  - Parse tree
  - Abstract syntax tree or other intermediate form
  - Modified intermediate form
  - Target language (e.g., assembler)
  - Modified target language

  - Scanner (lexical analysis)
  - Parser (syntax analysis)
  - Semantic analysis and intermediate code generation
  - Machine-independent code improvement (optional)
  - Target code generation
  - Machine-specific code improvement (optional)

  Front end

  Symbol table

  Back end
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
  - “≤” or “while” to describe an operator or reserved word,
Lexical Analysis

For each token type, give a description:

- either a literal string
  - “≤” or “while” to describe an operator or reserved word,
- or a < rule >
  - the rule < unsigned int > might stand for “a sequence of one or more digits”; the rule < identifier > might stand for “a letter followed by a sequence of zero or more letters or digits.”
Typical Tokens in Programming Languages

- **Operators and Punctuation**
  - + − * / ( ) [ ] ; : :: < <= == = != ! . . . !
  - Each of these is a district lexical class
Typical Tokens in Programming Languages

- **Operators and Punctuation**
  - + - * / ( ) [ ] ; : :: < <= == = != ! ...
  - Each of these is a distinct lexical class

- **Keywords**
  - if while for goto return switch void ...
  - Each of these is also a distinct lexical class (not a string)
Typical Tokens in Programming Languages

- **Operators and Punctuation**
  - + - * / ( ) [ ] ; : :: < <= == = != ! ...!
  - Each of these is a distinct lexical class

- **Keywords**
  - if while for goto return switch void ...
  - Each of these is also a distinct lexical class (not a string)

- **Identifiers** (variables)
  - A single ID lexical class, but parameterized by actual identifier (often a pointer into a symbol table)
Typical Tokens in Programming Languages

- **Operators and Punctuation**
  - `+ - * / ( ) [ ] ; : :: < <= == = != ! ...!`
  - Each of these is a distinct lexical class

- **Keywords**
  - `if while for goto return switch void ...`
  - Each of these is also a distinct lexical class (not a string)

- **Identifiers** (variables)
  - A single ID lexical class, but parameterized by actual identifier (often a pointer into a symbol table)

- **Integer constants**
  - A single INT lexical class, but parameterized by numeric value
Typical Tokens in Programming Languages

- **Operators and Punctuation**
  - + - * / ( ) [ ] ; :: < <= == = != ! ...!
  - Each of these is a distinct lexical class

- **Keywords**
  - if while for goto return switch void ...
  - Each of these is also a distinct lexical class (not a string)

- **Identifiers** (variables)
  - A single ID lexical class, but parameterized by actual identifier (often a pointer into a symbol table)

- **Integer constants**
  - A single INT lexical class, but parameterized by numeric value
  - 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
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, $L(re)$ is the language (set of strings) generated by $re$. 
These are the basic building blocks that other REs are built from.

<table>
<thead>
<tr>
<th>re</th>
<th>$L(re)$</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>{ a }</td>
<td>Singleton set, for each symbol $a$ in the alphabet $\Sigma$</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>{ $\varepsilon$ }</td>
<td>Empty string</td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>{ }</td>
<td>Empty language</td>
</tr>
</tbody>
</table>
Operations on REs

<table>
<thead>
<tr>
<th>re</th>
<th>$L(re)$</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rs$</td>
<td>$L(r)L(s)$</td>
<td>Concatenation – r followed by s</td>
</tr>
<tr>
<td>$r</td>
<td>s$</td>
<td>$L(r) \cup L(s)$</td>
</tr>
<tr>
<td>$r^*$</td>
<td>$L(r)^*$</td>
<td>0 or more occurrences of r (Kleene closure)</td>
</tr>
</tbody>
</table>
Operations on REs

<table>
<thead>
<tr>
<th>re</th>
<th>L(re)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs</td>
<td>L(r)L(s)</td>
<td>Concatenation – r followed by s</td>
</tr>
<tr>
<td>r</td>
<td>s</td>
<td>L(r) ∪ L(s)</td>
</tr>
<tr>
<td>r*</td>
<td>L(r)*</td>
<td>0 or more occurrences of r (Kleene closure)</td>
</tr>
</tbody>
</table>

- Precedence: (R), R*, R₁R₂, R₁|R₂ (lowest).
- Parenthesis can be used to group REs as needed.
## Examples

<table>
<thead>
<tr>
<th>re</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>single + character</td>
</tr>
<tr>
<td>!</td>
<td>single ! character</td>
</tr>
<tr>
<td>!=</td>
<td>2 character sequence “!=&quot;</td>
</tr>
<tr>
<td>xyzzy</td>
<td>5 character sequence “xyzzy”</td>
</tr>
<tr>
<td>(1</td>
<td>0)*</td>
</tr>
<tr>
<td>(1</td>
<td>0)(1</td>
</tr>
<tr>
<td>0</td>
<td>1(1</td>
</tr>
</tbody>
</table>
Abbreviations on REs

There are common abbreviations used for convenience.

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>r+</td>
<td>(rr*)</td>
<td>1 or more occurrences</td>
</tr>
<tr>
<td>r?</td>
<td>(r</td>
<td>ε )</td>
</tr>
<tr>
<td>[a-z]</td>
<td>(a</td>
<td>b</td>
</tr>
<tr>
<td>[abxyz]</td>
<td>(a</td>
<td>b</td>
</tr>
</tbody>
</table>
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/
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"): 
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.:

\[ \text{ifStmt} ::= \text{if ( expr ) statement} \]
\[ \text{ifStmt} \rightarrow \text{if ( expr ) statement} \]
\[ <\text{ifStmt}> ::= \text{if ( <expr> ) <statement>} \]
A formal grammar for a “pig language” could be:

\[
\text{PigTalk ::= } oink \text{ PigTalk (Rule 1)} \\
\text{ | } oink! \text{ (Rule 2)}
\]
Example: Grammar for Pigese (or Pigish?)

- A formal grammar for a “pig language” could be:
  \[
  \text{PigTalk ::= oink PigTalk (Rule 1)} \\
  \quad \mid \ oink! \quad \text{(Rule 2)}
  \]

- PigTalk can then generate, for example:
  1. \[
     \text{PigTalk ::= oink! \quad (Rule 2)}
  \]
Example: Grammar for Pigese (or Pigish?)

- A formal grammar for a “pig language” could be:
  \[ \text{PigTalk} ::= \text{oink PigTalk} \quad \text{(Rule 1)} \]
  \[ \quad | \quad \text{oink!} \quad \text{(Rule 2)} \]

- PigTalk can then generate, for example:
  1. \[ \text{PigTalk} ::= \text{oink!} \quad \text{(Rule 2)} \]
  2. \[ \text{PigTalk} ::= \text{oink PigTalk} \quad \text{(Rule 1)} \]
     \[ ::= \text{oink oink!} \]
A formal grammar for a “pig language” could be:

\[ \text{PigTalk ::= oink PigTalk (Rule 1)} \]
\[ \mid \text{oink! (Rule 2)} \]

PigTalk can then generate, for example:

1. \[ \text{PigTalk ::= oink! (Rule 2)} \]
2. \[ \text{PigTalk ::= oink PigTalk (Rule 1)} \]
\[ \text{ ::= oink oink!} \]
3. \[ \text{PigTalk ::= oink PigTalk (Rule 1)} \]
\[ \text{ ::= oink oink oink PigTalk (Rule 1)} \]
\[ \text{ ::= oink oink oink oink! (Rule 2)} \]
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 Gramars)

- 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 Gramars)

- 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 (as in the book)
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): EXAMPLE

**Grammar**

A, B, and C are non-terminals.
0, 1, and 2 are terminals.
The start symbol is A.
The rules are:

- \( A \rightarrow 0A | 1C | 2B | 0 \)
- \( B \rightarrow 0B | 1A | 2C | 1 \)
- \( C \rightarrow 0C | 1B | 2A | 2 \)

https://itempool.com/jjumadinova/live

Can 2011020 be parsed?
Grammar
A, B, and C are non-terminals.
0, 1, and 2 are terminals.
The start symbol is A, the rules are:

- \( A \rightarrow 0A|1C|2B|0 \)
- \( B \rightarrow 0B|1A|2C|1 \)
- \( C \rightarrow 0C|1B|2A|2 \)
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 1112202 be parsed?
Can 00102 be parsed?
Can 2121 be parsed?
Syntactic Analysis

prog → {statement}⁺
statement → assignment | loop | io
assignment → id = expression
loop → while ( expression ) prog

“A program is one or more statements.”
“A statement is an assignment, a loop, or an input/output command.”
“An assignment is an identifier, followed by “=”, followed by an expression.”

The “{...}⁺” means “one or more repetitions of the items in {...}”

In this example, “=”, “while”, “(“, and “)” are terminals.
Syntactic Analysis

- The process of verifying that a token stream represents a valid application of the rules is called **parsing**.
- Using the BNF rules we can construct a parse tree:
Sample Parse Tree (portion)

```
while (a == 3) {
    found = 1
}
```
Sample Parse Tree (failed)

Parse halts after "while" -- unable to match the "(" in the rule with a "(" in the input. An error is reported by the compiler.

```
while x <= 10
    a = x+1
```
Grammar for Java (version 8)

- Overview of notation used:
  https://docs.oracle.com/javase/specs/jls/se8/html/jls-2.html

- The full syntax grammar: