Today

- Quiz 1
- Wrap up lexical analysis (from Tues)
- Intro to formal grammars

Assignments

- Exercise 1 due
- HW-1 out due
- HW-2 out
Lexical Analysis – Basics

Goal: simplify syntax analysis (parsing) and detect (token) errors early

- a “lexer” only deals with building tokens, not checking how they “go together”
- allows parser to focus on checking syntax rules (separation of concerns)

The basic idea:

<table>
<thead>
<tr>
<th>Source Code:</th>
<th>Token Sequence as TYPE(lexeme):</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun int f()</td>
<td>FUN(&quot;fun&quot;), INT_TYPE(&quot;int&quot;), ID(&quot;f&quot;), LPAREN(&quot;(&quot;), RPAREN(&quot;)&quot;),</td>
</tr>
<tr>
<td>{}</td>
<td>LBRACE{&quot;&quot;),</td>
</tr>
<tr>
<td>var x = 0</td>
<td>VAR(&quot;var&quot;), ID(&quot;x&quot;), ASSIGN(&quot;=&quot;), INT_VAL(&quot;0&quot;),</td>
</tr>
<tr>
<td>return x</td>
<td>RETURN(&quot;return&quot;), ID(&quot;x&quot;),</td>
</tr>
<tr>
<td>}</td>
<td>RBRACE{&quot;&quot;})</td>
</tr>
</tbody>
</table>

How it works:

- Source code converted to a sequence (or a stream) of tokens
- Skip over non-tokens (white space, comments)
- Keep line and column numbers as part of tokens

Note:

- a sequence is similar to a list
- a stream is similar to an iterator
Exercise: Give the token sequence (token type, lexeme, line, column) for the following MyPL code snippets. Assume the token types:

```
LPAREN, RPAREN, ASSIGN, LBRACE, RBRACE
EQUAL, MODULO, PLUS
INT_VAL, STRING_VAL, ID
VAR, FOR, FROM, UPTO, IF
```

Snippet 1:
```
print("Hello World!")
```

Snippet 2:
```
var x = 0
for i from 1 upto 10 {
    if i % 2 == 0 {
        x = x + i
    }
}
```
A Lexer is implemented using either
- a lexical analyzer tool (e.g., Lex, Flex, JFlex, ...)
- or as an ad hoc program (hand written) ... we’ll do this!

Lexer usually called one token at a time ... like an iterator

• the parser asks the lexer for the next token
• the lexer reads just enough from the source code to create a token
• the token (type, lexeme, line & column number) returned to the parser

Lexer only detects errors in forming tokens ... for example:
- unexpected characters/symbols (like an exclamation mark)
- poorly formed constant values (strings, numbers, etc)
- poorly formed identifiers

Dealing with errors
- lexer returns a special error token ... or enter “panic mode”
- lexer raises an exception ... what we’ll do
- compilers stop (e.g., Python) or keep going (e.g., C++, Java)
Full Set of MyPL Tokens (for HW-2)

```java
public enum TokenType {

    // basic symbols
    COMMA, DOT, PLUS, MINUS, MULTIPLY, DIVIDE, MODULO, 
    LBRACE, RBRACE, LPAREN, RPAREN,

    // comparators
    NOT_EQUAL, EQUAL, GREATER_THAN, GREATER_THAN_EQUAL, LESS_THAN, 
    LESS_THAN_EQUAL,

    // assignment
    ASSIGN,

    // primitive values
    CHAR_VAL, STRING_VAL, INT_VAL, DOUBLE_VAL, BOOL_VAL,

    // boolean operators
    AND, OR, NOT, NEG,

    // data types
    INT_TYPE, DOUBLE_TYPE, CHAR_TYPE, STRING_TYPE, BOOL_TYPE, VOID_TYPE,

    // reserved words
    VAR, TYPE, WHILE, FOR, FROM, UPTO, DOWNTO,
    IF, ELIF, ELSE,
    FUN, NEW, DELETE, RETURN, NIL,

    // identifiers
    ID,

    // end of stream
    EOS
}
```
Formal Grammars

A set of rules that specify a language’s syntax

- a “language” here broadly means a set of allowable strings
- for this class, programming languages

In PL implementation, grammars can be used within

- Lexers (lexical analysis) e.g., numbers, strings, comments
- Parsers (syntax analysis) check if syntax is correct

Different “classes” of grammars

- “regular” grammars specify regular languages (think regular expressions)
- “context free” grammars specify context-free languages (most PLs)
- ... and so on
- we’ll just cover the basics of regular and context-free grammars
**Grammar Rules**

Grammar rules define *productions* (aka *rewritings*).

\[ S \rightarrow a \]

- Here we say *S produces* (or *yields*) *a*
  - \( S \) is a **non-terminal** symbol (LHS of a rule) … sometimes as \(<s>\)
  - \( a \) is a **terminal** symbol
  - terminal and non-terminal symbols are **disjoint**
  - set of terminals is the **alphabet** of the language
  - often a distinguished **start** symbol

- Rules can be applied to create (or check) a **derivation**
  - from start, repeatedly apply rules until only terminals remain