Languages and Compilers
Lexical Analysis and the Scanner

Allan Milne and Adam Sampson
School of Design and Informatics
Abertay University

Agenda.

- Lexical Analysis.
- Tokens.
- The Scanner.

Lexical Analysis.

- The Lexical Analyser.
- Tokens.
- Character Processing.
- Terminal Representations.

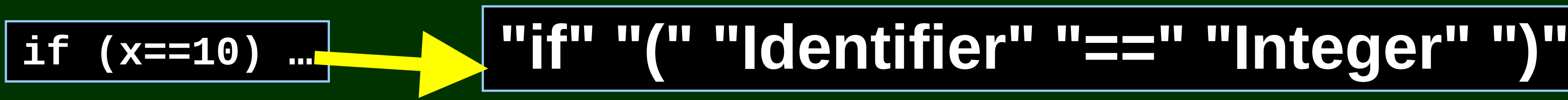
Where Are We Now And Where Are We Going?

- EBNF defines the syntax of a language.
- A source program is written in that language.
- The compiler processes the source program according to the rules of the EBNF.
  - We do not want to do this processing in terms of the individual characters of the program.
- So the first step is to identify the lexical elements (terminals) of the source program.
  - identify groups of characters that form the terminals; keywords, punctuation, microsyntax.

The Lexical Analyser.

- Also known as the *scanner*.

- Its role:
  - to transform an input stream of *characters*
  - into tokens,
  - and expose these tokens to the rest of the compiler.



```
if (x==10) …
```

"if" "(" "Identifier" "==" "Integer" ")"

Character stream

Output tokens

What Are Tokens?

- Tokens are the internal compiler representations of the terminal symbols of a source program as defined by an EBNF.
- Simple terminals
  - keywords; e.g. *begin, for, if, …*
  - single character punctuation; e.g. {, =, *…*
  - multi-character punctuation; e.g. ==, <=, ->, *…*
- Microsyntax terminals
  - defined in the microsyntax of the EBNF.
  - E.g. identifiers, literal constants.

Where Are These Defined?

- Simple terminals are defined implicitly
  - as part of an EBNF rule.
    - **`<If> ::= if (<Expr>) then <Stat>;`**

- Microsyntax terminals are defined as name/pattern pairs in the microsyntax part of the EBNF
  - informally through some descriptive string.
    - **`Integer <| an unsigned integer`**
  - or formally through patterns that are regular expressions (preferred).
    - **`Integer <|\d+`**

Character Processing.

- The scanner must input each character of the source program stream and build the tokens.
- It must also handle
  - invalid tokens & characters,
  - spaces, tabs and newlines (*whitespace*),
  - comments (if part of the language), and
  - end-of-file.
- This is all done character by character.

Terminal Representations.

- Simple terminals such as keywords and punctuation can be represented by themselves;
  - as the actual input string.

- Microsyntax terminals are represented by an object containing:
  - the type of token (eg Integer)
  - the actual value (eg 10).

Tokens.

- Token Role.
- The *IToken* Contract.
- The Token Class.

How To Represent A Terminal?

- As an object of a *Token* class.

- This encapsulates the exposed attributes of the terminal token;
  - token type
    - “>=”, “begin”, “Identifier”, "Integer"
  - actual string value of the token
    - “>=”, “begin”, “total”, "123"
  - position of the token in the source code.

The Token Interface.

```
public interface IToken {
  String TokenType { get; }
  String TokenValue { get; }
  int Line { get;  }
  int Column { get;  }
  bool Is (String s);
  String ToString ();
} // end IToken interface.
```

The *Ardkit* Toolkit.

- We're going to be using Allan's C# toolkit for building simple compilers in some of the practical exercises...

- It's on MyLearningSpace;
  details in Practical 4

Notes On The *Token* Class.

- Objects are immutable.
- Constants are exposed for use with the *TokenType* property.
- The *ToString()* method is overridden to reflect both simple and microsyntax terminals.
- Overloaded constructors are exposed.

The Scanner.

- Scanner Responsibilities.
- Representing Errors.
- The *Scanner* Class.
- Using the scanner.

Scanners Must …

- expose methods to allow user application (i.e. compiler) interaction.
- input the characters of the source program from some input stream.
- construct token objectss from these characters.
- handle I/O conditions and invalid input.

The Scanner Interface.

```
public interface IScanner {

     IToken CurrentToken { get; }
     bool EndOfFile { get; }
     void Init (TextReader src,
                List<ICompilerError> errs);
     IToken NextToken () ;

} // end IScanner interface.
```

Comments On The Contract.

- *CurrentToken* tells us what we have,
- *NextToken()* moves us on.
- *Init(…)* initialises the scanning process;
  - On a specific input text stream that contains the source program,
  - Returning any detected **errors** in the supplied list collection.

Representing Compiler Errors.

- A primary responsibility of a compiler is to detect and report errors.
- For flexibility these are maintained in a collection that may be rendered to the user in due course.
- See the Ardkit compiler error reference.
  - The error class hierarchy represents all compiler errors including lexical errors.

Scanner Class Outline.

```
public abstract class Scanner : IScanner {
   … private and protected members …

   public Scanner () { … }
   … public methods implementing interface …

   protected void getNextChar () {…}

   protected abstract IToken getNextToken ();

} // end Scanner class.
```

Scanner Class Commentary.

- The base for a concrete scanner;
  - encapsulates functionality common to all languages,
  - Each language-specific subclass implements the *getNextToken*() method.

- See *Ardkit* scanner reference.

Further Commentary.

- Protected members used by *getNextToken()* method in subclass.
- Constructor creates “empty” object.
- *Init (…)* sets up and starts processing by reading the first character.
- *EndOfFile* property true if current token is an *EndOfFile token*.
- *NextToken()* method packages call to *getNextToken()* language-specific method.

The getNextChar() Method.

- Reads and buffers input lines.
- Sets up next input character in the *currentChar* field.
- Maintains the line and column positions.
  - so when we detect an error (e.g. an invalid token), we can report where it occurred
- Catches I/O errors and adds to error collection.

The *getNextToken()* Method.

- Must be overridden in each subclass.
- This is the real work of constructing a token from the input characters.
- Accesses the protected members.
- See next lecture for details.
- The *NextToken()* method packages the returned token in terms of overall scanner functionality.

Creating A Language-Specific scanner.

1) Define a subclass of *Scanner*.

2) Implement the *getNextToken()* method in the subclass to construct a single token.

3) Instantiate and use the scanner.

Using The Scanner.

- Instantiate a scanner object.
- Create an appropriate input stream.
- Create an error collection object.
- Call the *Init(…)* mehtod.
- Call *nextToken()* to retrieve tokens one at a time, and
  - access the methods of the token object returned.
- On completion, access the error collection to report any errors detected.

For Example ...

```
MyScanner scanner = new MyScanner ();
List<ICompilerError> errs = new List<ICompilerError>();
try {
  StreamReader infile = new StreamReader (“mySource.txt”);
  scanner.Init (infile, errs);
  do {
    IToken token = scanner.NextToken();
    display (token);
  } while (!scanner.EndOfFile);
  infile.Close();
} catch (IOException e) { … }

if (errs.Count > 0) {
   foreach (ICompilerError error in errs)
      Console.WriteLine (error);
}
```

… And Using The Tokens …

- Having retrieved the next token the application can then call the accessor methods of the token object to extract detailed information.

```
private void display (IToken token) {
  Console.WriteLine ("{0} at ({1},{2}) ",
         token.TokenType, token.Line, token.Column);
  if (!token.TokenType.Equals (token.TokenValue))
     Console.WriteLine ("   actual token : {0}",
           token.TokenValue);
} // end display method.
```

Summary.

So Now You Can …

… describe the role and responsibilities of the scanner.

… explain & extend an OO token representation for a language specification.

… explain an object-oriented pattern for the design & implementation of a scanner.

… use scanner and token classes in an application.

Any questions?

- **Practical**: further exercises with BNF:
  - look at some more realistic examples of BNF specifications
  - analyse them for LL(1)-ness
  - You may like to work in pairs for this (but you don't have to)

- **Next lecture**: introducing type systems … which is a bit long so I'm going to do the start of it now!