CS 242P – Winter Quarter 2023 – Prof. Dr. Michael Franz

MCS Course on Compilers
Optional Tokenizer & Parser Interface

Using specification in this document is totally optional, but if you use this interface for your tokenizer and parser modules, you may be able to benefit from a testing harness that we will make available.

Token Values

The tokenizer produces tokens that are consumed by the parser. The tokens are really an *enumeration type*, but we are going to use integer constants for simplicity.

As will be mentioned in class, the token order below has been carefully chosen to make error handling particularly easy in the parser.

Characters	Token	Value	Characters	Token	Value
	errorToken	0	*number*	number	60
			identifier	ident	61
*	timesToken	1			
/	divToken	2	;	semiToken	70
+	plusToken	11	}	endToken	80
-	minusToken	12	od	odToken	81
			fi	fiToken	82
==	eqlToken	20			
!=	neqToken	21	else	elseToken	90
<	lssToken	22			
>=	geqToken	23	let	letToken	100
<=	leqToken	24	call	callToken	101
>	gtrToken	25	if	ifToken	102
			while	whileToken	103
.	periodToken	30	**return**	**returnToken**	**104***
,	commaToken	31			
[	openbracketToken	32	var	varToken	110
]	closebracketToken	34	array	arrToken	111
)	closeparenToken	35	void	voidToken	112
			function	funcToken	113
<-	becomesToken	40	procedure	procToken	114
then	thenToken	41			
do	doToken	42	{	beginToken	150
			computation	mainToken	200
(	openparenToken	50	end of file	eofToken	255

```
FileReader.Error =0
FileReader.EOF = 255
```

A Three-Tiered Read-Ahead Architecture

Many smartly designed programming languages have LL(1) grammars that can be parsed reading from **L**eft to right, while forming the **L**eftmost derivation of the parse tree, using a **1**-symbol lookahead.

The tokenizer is in fact also a parser, for a regular-grammar subset of the larger language. Each token is a sentence of this regular subset language, and the tokenizer uses a **one-character lookahead** from the input stream, which consists of characters.

For example, for the regular language (which can be parsed by a finite state machine)

```
letter  =  “a” | “b” | … | “z”.
digit  =  “0” | “1” | … | “9”.

identifier  =  letter {letter | digit}.
number  =  digit {digit}.

token = number | identifier | “+” | “-“ .
```

with starting symbol *token*, if the input sequence is 1234+, the tokenizer will recognize that it is done parsing the number 1234 after it has already inspected the next input symbol +. The next time the tokenizer is called, it needs to revisit the + character a second time and it will recognize it as the plus token (there are no other tokens that start with a plus character).

The parser in turn uses a **one token lookahead** from its input stream, which consists of tokens. It also may need to inspect its lookahead token multiple times while making its decisions.

So overall, our compiler uses a lookahead of one token plus one character and each layer buffers its lookahead token so that it can be inspected multiple times if necessary.

Lowest Tier: Streams of Characters

The *FileReader* class encapsulates the **file** operations. A call of *FileReader.GetNext()* returns the frontmost character on the input and advances the underlying stream to the next character.

A special character FileReader.EOF signifies an end of file condition. Subsequent calls of FileReader.GetNext will continue returning FileReader.EOF and not advance the input furher.

A call of FileReader.Error() will output an error message with the current file position and set an internal error state. Subsequent calls of FileReader.GetNext will return the FileReader.Error character and not advance the input further.

```
public class FileReader; {

  public char GetNext(); // return current and advance to the next character on the input

  public void Error(String errorMsg); // signal an error message
  public FileReader(String fileName); // constructor: open file
}
```

Middle Tier: Streams of Tokens

The *Tokenizer* class encapsulates **token** operations. It caches the frontmost character on its input stream because it may need to inspect it multiple times.

A call of *Tokenizer.GetNext()* returns the frontmost token on the input, consuming as many of the symbols from the underlying character stream as necessary.

A special token Tokenizer.EOF signifies an end of file condition. Subsequent calls of Tokenizer.GetNext will continue returning Tokenizer.EOF and not advance the input further.

A call of Tokenizer.Error() will output an error message by calling the underlying FileReader.Error(). It will also set an internal error state. Subsequent calls of Tokenizer.GetNext will return the Tokenizer.Error token and not advance the input further.

When Tokenizer.GetNext discovers an identifier, the instance variable *Tokenizer.id* is simultaneously updated to indicate which identifier was just seen. If the token is a number, the instance variable *Tokenizer.number* reflects the value of the number.

The Tokenizer class also maintains a table of all unique identifiers it has encountered. Any new identifier is automatically added to the table. As already stated, the index number of the most recently seen identifier is stored in the instance variable *Tokenizer.id*. The two methods *Id2String* and *String2Id* translate between an identifier's textual representation and its id number. [Recognizing reserved words of the language can be accomplished trivially by pre-populating the string table with keywords.]

```
public class Tokenizer; {
  private FileReader myFileReader;
  private char inputSym; // the current character on the input
  private void next(); { inputSym = myFileReader.GetNext(); } // advance to the next character

  /* symmetrical to the FileReader class */
  public int GetNext(); // return current and advance to the next token on the input
  public int number; // the last number encountered
  public int id; // the last identifier encountered

  public void Error(String errorMsg); { myFileReader.Error(errorMsg); … }
  public Tokenizer(String fileName); // constructor: open file and cache its frontmost char in "inputSym'

  /* identifier table methods */
  public String Id2String(int id);
  public int String2Id(String name);

…
}
```

Top Tier: Context-Free Grammar

The *Parser* class caches the frontmost token on the token stream so that it can examine it multiple times, just like the Tokenizer class caches the frontmost character on its input stream.

```
public class Parser; {
  private Tokenizer myTokenizer;
  private int inputSym; // the current token on the input
  private void next(); { inputSym = myTokenizer.GetNext(); } // advance to the next token


  void CheckFor(int token); {if (inputSym==token) then next(); else myTokenizer.Error("SyntaxErr"); }

…

  public Parser(String fileName); // constructor: open file and cache its frontmost token in "inputSym'
    {
    myTokenizer = new Tokenizer(filename); next();
   CheckFor(mainToken);
    ...
    CheckFor(periodToken);
    }

...

}
```