

ASSIGNMENT #2

Due Tuesday, September 23, 2003

Consider the following extended BNF grammar for a subset of the C# programming language, called MicroC#.

compilation-unit	::=	using System; class class-identifier <u>{</u> { class-member-declaration } main-declaration <u>}</u>
class-member-declaration	::=	field-declaration method-declaration
field-declaration	::=	static type variable-identifier ;
main-declaration	::=	public static void Main () main-body
main-body	::=	<u>{</u> { local-variable-declaration ; } [statement-list] <u>}</u>
local-variable-declaration	::=	type variable-identifier
type	::=	int List
method-declaration	::=	method-header method-body
method-header	::=	static type function-identifier ([formal-parameter-list])
formal-parameter-list	::=	formal-parameter {, formal-parameter }
formal-parameter	::=	type variable-identifier
method-body	::=	<u>{</u> { local-variable-declaration ; } [statement-list] return expression ; <u>}</u>
block	::=	<u>{</u> statement-list <u>}</u>
statement-list	::=	statement { statement }
assignment	::=	variable-identifier = expression
statement	::=	assignment ; block if (expression) statement [else statement] while (expression) statement Console . WriteLine (expression) ;
expression	::=	term expression binary-op expression
term	::=	primary-expression unary-op term
primary-expression	::=	variable-identifier integer (expression) function-identifier ([expression-list]) new List () primary-expression . cons (expression) primary-expression . head () primary-expression . tail () primary-expression . isNull ()
expression-list	::=	expression {, expression }

Syntactic and Semantic Conventions The keywords and the token symbols in MicroC# are in bold. Note that MicroC# like C++, has symbols {, }, [, and] which are distinguished from grammar metasympols {, }, [, and], respectively, by underlining. The unary operators are +, - and ! (negation) Binary operators obey the customary precedence rules, from highest to lowest:

multiplicative	*, /
binary additive	+, -
relational	==, !=, <, >, <=, >=
boolean and	&&
boolean or	

Assume that an identifier can only contain letters (only alphabetic characters), digits, and underscores (_) with the restrictions that it must begin with a letter, cannot end with an underscore and cannot have two consecutive underscores. For example, give_2_Joe, tell_me and A45Asm3 are valid identifiers, but 6gh, two__bad, and no_end_ are not. class-identifiers, variable-identifiers, and function-identifiers are the same lexical/syntactic items as identifiers but have the semantics given by the appropriate qualifier. integer is an unsigned integer. Assume that comments are indicated by being preceded by //.

1. Determine the set of tokens which a lexical analyzer would need to recognize.
2. Design and implement a lexical analyzer procedure to read a source program in the above language and print the next token in the input stream. If the token detected is a valueless token, such as a keyword, then it is sufficient to print only the keyword. If it has a value, then both the token type and lexeme should be printed.
3. You will be given several MicroC# programs with which to test your lexical analyzer. These will be located on the class WWW page and will be of the form Test1.cs, Test2.cs, etc.

Suggestions:

1. Construct a token class to represent the token data structure, including a method to print a token.
2. Use the JLex tool to automatically construct a lexical analyzer in Java from a set of regular expressions specifying tokens. This can interface with your token class to return a token to the main program which then prints the token.