

CS 602/702 COMPILER DESIGN I PROJECT

Due Tuesday, December 11, 2007, at 5:00 P.M.

Write a compiler for the TinyJava¹ language.

1. The lexical analyzer should recognize all necessary tokens and comments in the source program. Java comments are preceded by `//` and terminated by the end of the line. You may assume that all keywords will be lower case but upper and lower case letters should be distinguished in identifiers. Identifiers must begin with a letter and then may be followed by zero or more letters, digits and underscores such that there are no two consecutive underscores and an underscore does not appear at the end. Integers are unsigned strings of one or more decimal digits. The source program text should also be output in this phase.
2. The parser should recognize exactly the described syntax. If there are any syntactic errors, an appropriate error message should be output and compilation may be terminated. The concrete syntax of the subject language expressed in extended BNF is given on the following page. Note that `{`, `}`, `[`, and `]` are terminal symbols as well as grammar symbols, the former distinguished from the latter by underlining. The start symbol of this grammar is `program`.
3. Types include primitive integers and booleans, first-class objects, arrays of these types, and functions (methods) returning values of these types. Type checking should make sure that these data types are used correctly. All function identifiers are public and all variable identifiers are protected.
4. The symbol table should contain all identifiers used in the source program along with the appropriate attributes. The symbol table may be considered as an environment in which static semantics checking can be done.
5. The target code you generate will be an equivalent program in an executable language such as the Microsoft Common Intermediate Language (CIL) or Java Virtual Machine (JVM) code.

The compiler may be written in any programming language that you wish and implemented on any computer that you wish. In the past, most projects have benefitted from the Java software tools JLex and CUP which allow automatic generation of lexical and syntactic analyzers. These tools expect to interface with programs written in Java. There are similar tools available for C and C++. It is also possible to use alternative languages by adding additional passes in the compiler. For example, CUP could be used to generate a tree representation of the program which could be output in a form compatible with programming language L. Programs in L could then read this intermediate form to complete the additional compilation phases.

¹TinyJava is a distilled version of Java

program	::=	import java.util.* ; class-definition {class-definition}
class-definition	::=	class class-identifier [extends class-identifier] { {member-list} }
member-list	::=	member-declaration {member-declaration}
member-declaration	::=	declarator ; function-definition
declarator	::=	protected [static] type { [] } object-identifier
function-definition	::=	function-declaration { { declarator ;} [statement-list] return expression ; } }
function-declaration	::=	public [static] type function-identifier ([argument-declaration-list]) main-declaration
main-declaration	::=	[Scanner in = new Scanner (System . in) ; public static void main (String [] args)]
type	::=	class-identifier int boolean
argument-declaration-list	::=	argument-declaration {, argument-declaration}
argument-declaration	::=	type { [] } object-identifier
compound-statement	::=	{ statement-list }
statement-list	::=	statement {statement}
statement	::=	compound-statement assignment-statement ; if (expression) statement [else statement] while (expression) statement System . out . println (expression) ;
assignment-statement	::=	variable = expression variable = new class-identifier () variable = new int [<i>integer</i>] { [<i>integer</i>] } variable = in . nextInt ()
expression	::=	term expression binary-operator expression
expression-list	::=	expression {, expression}
term	::=	primary-expression unary-operator term
primary-expression	::=	object <i>integer</i> (expression)
object	::=	variable function-call
variable	::=	this [object .] object-identifier { [expression] }
function-call	::=	[object .] function-identifier ([expression-list])

Syntactic and Semantic Conventions The keywords and the token symbols in the above subset of Java are in bold. The unary operators are +, - and ! (negation) Binary operators obey the customary precedence rules, from highest to lowest:

multiplicative	*, /
binary additive	+, -
relational inequality	<, >, <=, >=
relational equality	==, !=
conjunction	&&
disjunction	

class-identifiers, object-identifiers, and function-identifiers are the same lexical/syntactic items as identifiers but have the semantics given by the appropriate qualifier.

It is suggested that you keep the following tentative schedule in order to guarantee your successful completion of this project. The schedule may be revised based on class coverage of the necessary prerequisite lecture material. Furthermore, the phases of the project listed below may be further decomposed for purposes of better modularity.

<u>Compiler Phase</u>	<u>Estimated Completion Date</u>
Lexical analyzer	Tuesday, September 11
Syntax analyzer	Tuesday, October 2
Symbol table routines	Tuesday, October 23
Type checking routines	Tuesday, November 6
Code generation	Tuesday, December 4