Home
Contact
Short Vita
Research
Teaching
LSDIS Lab
Links

TinyJava Language Definition

TinyJava is a small subset of the well-known programming language Java.

(For a definitive specification of Java, see James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *The Java Language Specification*, *Third Edition*, Addison Wesley, 2005; also available on-line here.)

1. Lexical conventions

Just like Java, TinyJava is case sensitive. That is, the corresponding upper- and lower-case letters are treated as different when used to form identifiers and keywords.

1.1. Ignored tokens

A *comment* is a sequence of characters enclosed within a pair of matching `/*` and `*/`, or any characters following `//` until the end of the line containing the `//` sequence. Comments are always ignored. Note that nested comments are not permitted. However, the `/*` … `*/` comments can extend over multiple lines.

Other ignored tokens include the *newline*, *horizontal tab*, *carriage return*, and the *space* character.

1.2. Basic tokens of TinyJava

1.2.1 Identifiers

An *identifier*, denoted by Identifier, is a finite sequence of letters and digits which begins with a letter. An underscore (_) is regarded as a letter. Identifiers may be of any length.

```
Letter     =  "a" | "b" | ... | "z" | "A" | "B" | ... | "Z" | "_"
Digit      =  "0" | "1" | ... | "9"
Identifier =  {Letter} ( {Letter} | {Digit} )*
```

Note that even though the dollar (`$`) can be used in Java identifiers, it is not allowed in TinyJava.

1.2.2. Numeric literals

A *decimal integer literal*, denoted by DecimalIntegerLiteral, is a sequence of decimal digits. Leading 0's are not allowed.

```
NonZeroDigit           =  "1" | ... | "9"
DecimalIntegerLiteral  =  "0" | { NonZeroDigit }{Digit}*
```

A *floating point literal*, denoted by FloatingPointLiteral, is defined as follows:

```
Exponent               =  ("e" | "E" ) ( ("+" | "−" ) {Digit}+ | {Digit}+)
FloatingPointLiteral   =  {Digit}+ "." {Digit}+ ("f" | "F") |
                       =  {Digit}+ "." {Digit}+ {Exponent} ("f" | "F")
```

Any numeric literal must be separated from an identifier or a keyword by at least one other different token (e.g. an ignored token).

1.2.3. String constants

A *string literal*, denoted by StringLiteral, is a sequence of characters enclosed within two double quotes (`"`). A string literal may include the sequence `\"` which represents a double quote character in the string in which it occurs and does not terminate the string. In addition, the sequence \n represents the

string in which it occurs and does not terminate the string. In addition, the sequence \n represents the NEWLINE character, while the sequence \\ represents the backslash character and may be included in a string as well. A sequence composed of the backslash followed by any character other than `'n'`, `'\'`, or `'"'` is illegal. A string literal may not include newline characters. A pair of matching `/*` ... `*/` within a string literal is not treated as a comment.

The precise regular expression defining a StringLiteral is not provided here.

1.2.4. Other tokens

Other tokens appearing in the grammar below are enclosed within double quotes. Keywords are additionally boldfaced. Tokens defined above (e.g. Identifier) are in boldface but not quoted.

2. TinyJava Syntax

tiny_java_program	—>	*class_decl*
class_decl	—>	"**public**" "**class**" **Identifier** "{" *member_decl_list* "}"
member_decl_list	—>	*member_decl* | *member_decl member_decl_list*
member_decl	—>	*field_decl* | *method_decl*
field_decl	—>	"**static**" *type* **Identifier** "=" *literal* ";"
method_decl	—>	"**static**" *return-type* **Identifier** "(" *formal_param_list* ")" "{" *method-body* "}" | "**static**" *return_type* **Identifier** "(" ")" "{" *method_body* "}" | "**public**" "**static**" "**void**" **Identifier** "(" **Identifier** "[" "]" **Identifier** ")" "{" *method-body* "}"
type	—>	"**int**" | "**float**"
return_type	—>	*type* | "**void**"
formal_param_list	—>	*formal_param* | *formal_param* "," *formal_param_list*
formal_param	—>	*type* **Identifier**
method_body	—>	*local_decls method_statement_list*
local_decl_list	—>	*local_decl local_decl_list* | ε
local_decl	—>	*type* **Identifier** "=" *literal* ";"
method_statement_list	—>	*statement method_statement_list* | *return_statement*
statement_list	—>	*statement statement_list* | ε
statement	—>	**Identifier** "=" *expression* ";" | "**if**" "(" *expression* ")" *statement* | "**if**" "(" *expression* ")" *statement* "**else**" *statement* | "**while**" "(" *expression* ")" *statement* | *method_invocation* ";" | "{" *statement_list* "}" | ";"
return_statement	—>	"**return**" *expression* ";" | "**return**" ";"
method_invocation	—>	*qualified_name* "(" *argument_list* ")" | *qualified_name* "(" ")"
qualified_name	—>	**Identifier** "." **Identifier** | **Identifier**
argument_list	—>	*expression* | *expression* "," *argument_list*
expression	—>	*relational_expression* | *relational_expression* "==" *relational_expression* | *relational_expression* "!=" *relational_expression*
relational_expression	—>	*additive_expression* | *additive_expression* "<" *additive_expression* |

additive_expression “<” *additive_expression* |
additive_expression “>” *additive_expression* |
additive_expression “<=” *additive_expression* |
additive_expression “>=” *additive_expression*

additive_expression —> *multiplicative_expression* |
additive_expression “+” *multiplicative_expression* |
additive_expression “–” *multiplicative_expression*

multiplicative_expression —> *unary_expression* |
multiplicative_expression “*” *unary_expression* |
multiplicative_expression “/” *unary_expression*

unary_expression —> *primary_expression* |
“+” *unary_expression* |
“–” *unary_expression* |
“(” *type* “)” *unary_expression*

primary_expression —> *literal* |
Identifier |
method_invocation |
“(” *expression* “)”

literal —> **DecimalIntegerLiteral** |
FloatingPointLiteral |
StringLiteral

3. TinyJava correct programs

TinyJava is a small subset of the well-known Programming Language Java. All TinyJava programs have the same semantics as their “regular” Java equivalents and should compile and execute properly

using any standard Java Development Kit, such as Oracle’s (formerly Sun’s) JDK, IBM’s JDK, or OpenJDK.

3.1. Data types

TinyJava features only two primitive data types: `int` and `float` which can be used as types of class fields, variables, parameters, and method return types.

3.2. Expressions

Expressions represent literal or computed values. *Primary expressions* are composed of literals (note that a floating point literal must have an `f` or `F` suffix), field, parameter, and variable references, as well as method invocations and type casts. Larger expressions may be formed using various *unary* and *binary operators*, and *parentheses*.

TinyJava offers five *binary arithmetic operators*: `+`, `-`, `*`, and `/`. The *additive operators* `+` and `-` have a lower precedence than *multiplicative operators* `*` and `/`. In addition, `+` and `-` may be used as unary operators which have the highest precedence. As in Java, the parenthesis may be used to alter the default precedence of operators.

Values represented by expressions have types. Given `int` arguments, each arithmetic operation returns a result of type `int`. If at least one of the arguments is of type `float`, the result is always `float`. In case of mixed type arguments, the `int` argument is automatically converted to `float` before the operation is performed. A method invocation represents a value consistent with the return type of the method. A void method invocation may not be used in an expression. Explicit casts may be used to convert the type from `int` to `float` and from `float` to `int`, if needed.

There are six binary *relational operators*: `<`, `<=`, `==`, `!=`, `>=`, `>`. Types are handled as above. The *equality operators* (`==` and `!=`) have lower precedence than the other relational operators. Relational operators can only be applied to numeric operands (expressions of type `int` or `float`). Each relational operator returns a `boolean` result. All of the operators (arithmetic and relational) can only be applied to numeric arguments. For example, it is not legal to compare two relational expressions for equality, even though TinyJava syntax allows it.

even though TinyJava syntax allows it.

Please note that according the TinyJava's syntax, an expression may be composed of a StringLiteral. In this case, the type of the expression is `String`. However, a `String` expression cannot be used as an operand to any of the arithmetic or relational operators.

TinyJava does not allow method overloading. Consequently, for every method invocation, there should be a *single* method declaration matching it. A method invocation matches a declaration if the invocation has the same number of arguments as the number of formal parameters in the corresponding method declaration. Furthermore, the type of the actual argument expression and the corresponding formal parameter must be the same or *convertible*. If the actual argument expression of type `int` is provided in place of a formal parameter of type `float`, an automatic conversion of the actual argument expression to (wider) type `float` is performed before the call. An explicit type cast must be used to pass a `float` actual argument expression in place of a formal parameter of type `int`.

Parameters of primitive types (`int` and `float`) are passed by value. `String` type parameters are passed by reference (see the SimpleIO class below).

3.3. Statements

TinyJava has seven statements: the assignment statement, the return statement, the while statement, the if statement (with and without the else clause), the method invocation statement, the block statement, and the empty statement. The semantics of each statement is the same as in Java.

The type of a variable, parameter, or field used on the left hand side of an assignment statement must be the same as that of the right hand side expression. In addition, if the left hand side is of type `float` and the expression of type `int`, an automatic conversion of the expression to `float` is performed. An explicit type cast must be used if a `float` expression is assigned to an `int` left hand side.

The expression used as part of the return statement must be of the same type as the return type of the method in which the return statement is used. However, if the method return type is `float` and the expression is of type `int`, an automatic conversion of the expression to `float` is performed. An explicit cast must be used if a `float` expression is returned from a method with the `int` return type.

The expression used in the if and while statements must be of type `boolean`. That is, in TinyJava, it must be a relational expression.

The method invocation statement may invoke a `void` or a non-`void` method.

Unlike in Java, the block statement in TinyJava may not contain local declarations (only executable statements are allowed).

3.4. Classes

A TinyJava program must be composed of only a single public class. All fields and methods declared in the class must be static and have package visibility (private, protected, and public access qualifiers are not allowed). In addition, all field declarations must include their initializations. Furthermore, there must be a single `public void main` method, which must have an array of `String`s as the only argument.

TinyJava follows the scoping rules of Java. However, package declarations and class imports are not permitted. Consequently, the only TinyJava class is placed in the default package.

A TinyJava method may include local variable declarations, which must be placed ahead of any executable statements. All local variable declarations must include their initializations. Methods may be recursive, i.e. a method may invoke itself. Every method must have a *single* return statement which must be the *last* statement in the method body (this requirement is already captured by the TinyJava grammar above).

The TinyJava system includes a class library including just one class, called `SimpleIO`, described below.

3.5. The TinyJava Class Library

3.5.1. The SimpleIO class

The `SimpleIO` class offers a number of static methods to conveniently perform simple reading and writing of TinyJava data. Reading is from the *standard input* and writing is to *standard output*. The meaning of the `SimpleIO`'s methods is self explanatory. The read methods read and return a single value of the requested type. If a requested `int` or `float` value cannot be read (the data does not exist, or it cannot be interpreted as an `int` or a `float`), an error message is printed on the error stream and `0` or `0.0f` is returned, respectively. The print methods write a single value on the standard output. The `println` method accepts no arguments and writes the end of line. Note that `printString` may write a string literal with a newline (`\n`) in it, effectively printing the end of line, as well.

The list of `SimpleIO`'s public static methods is given below:

```
public static int    readInt();
public static float  readFloat();
public static void   println();
public static void   printInt( int ival );
public static void   printFloat( float fval );
public static void   printString( String sval );
```

3.6. Examples

```
/*
 * Basic IO and simple int arithmetic
 */
public class Sample1
{
    public static void main( String[] args )
    {
        int i = 0;
        int j = 0;

        i = SimpleIO.readInt();               // get i
        j = 9 + i * 8;                        // evaluate j
        SimpleIO.printString( "Result is " );  // print the label
        SimpleIO.printInt( j );               // print j
        SimpleIO.println();                   // print the newline
        return;                               // return from method (last stmt)
    }
}
```

```
/*
 * Basic IO and simple mixed type arithmetic
 */
public class Sample1R
{
    public static void main( String[] args )
    {
        float x = 0.0f;
        float z = 0.0f;
        int   y = 0;


        x = SimpleIO.readFloat();          // read x
        z = 9 + x * 8;                     // assign to z (mixed types)
        y = (int) (9 + x * 8);             // assign to y with a cast

        SimpleIO.printString( "Result (int) is " );
        SimpleIO.printInt( y );            // print y
        SimpleIO.println();                // print newline

        SimpleIO.printString( "Result (float) is " );
        SimpleIO.printFloat( z );          // print z
        SimpleIO.println();                // print newline
        return;                            // return from method (last stmt)
    }
```

```
}

/*
 * Basic method invocation and a looping statement
 */
public class Sample2
{
    static int count( int n )
    {
        int i = 0;
        int sum = 0;

        i = 1;
        sum = 0;
        while ( i <= n ) {
            sum = sum + i;
            i = i + 1;
        }
        return sum;
    }

    public static void main( String[] args )
    {
        int i = 0;
        int sum = 0;

        i = SimpleIO.readInt();    // read i
        sum = count( i );          // call count
        SimpleIO.printInt( sum );  // print the result
        SimpleIO.println();        // print the newline
        return;                    // return from method (last stmt)
    }
}


/*
 * recursive factorial calculation
 */

public class FactorialRec
{

    static int n = 0;
    static int fact = 0;

    static int factorial( int n )
    {
        int retValue = 1;

        if ( n <= 1 )
            retValue = 1;
        else
            retValue = n * factorial( n - 1 );

        return retValue;
    }

    public static void main( String[] args )
    {

        SimpleIO.printString( "Enter an integer: " );
        n =  SimpleIO.readInt();                  // read n
        fact = factorial( n );                    // call factorial
        SimpleIO.printString( "Factorial of " );
        SimpleIO.printInt( n );                   // print n
        SimpleIO.printString( " = " );
        SimpleIO.printInt( fact );                // print the result
        SimpleIO.println();
        return;                                   // return from method (last stmt)
    }
}
```