CSC 435: Compiler Construction
Intermediate Representation Specification
Rev: 190

February 27, 2018

Contents

List of Grammars

List of Tables

List of Figures

1 Introduction

INTERMEDIATE REPRESENTATION (IR) is a form of program representation used by compilers:

1. It is tiresome, if not difficult and error-prone, to generate code directly from Abstract Syntax Trees (ASTs).

2. The generation of target code from ASTs does not facilitate retargeting to a new platform.

3. Program optimization requires program form(s) conducive to easy analysis and transformation.

For CSC 435, point (1) above provides the main motivation for using an IR: the IR provides a stepping stone in the process of "coaxing" programs (written in the unnamed language) into assembly language for Java Virtual Machine (JVM), described in [LY99]. Therefore, compilers for the unnamed language have the overall structure shown in Figure 1.

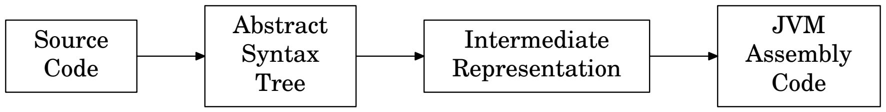

Figure 1: Overall design of compiler for CSC 435

This document describes a form of IR called **quadruples**; each instruction typically consists of four parts: a result, two input operands, and an operator. For example,

$$a = b \oplus c$$

Here a, b, and c are atomic operands, *not* quadruples. That is, IR instructions are "flat"; they do not have a recursive structure, in contrast to the AST. This flatness affords easy production of JVM assembly code.

Instructions with unary and nullary operations (three parts and two parts, respectively) are also supported. They are *also* referred to as quadruples.

Since the unnamed language is an imperative programming language, the IR itself is a small imperative language; IR instructions either modify program state or transfer control to another part of the program. Consequently, the IR has two types of instruction: assignment and jump.

This IR is not completely machine independent. Certain aspects of the JVM, such as types (see Section 2), are represented in the IR. It is indeed possible to create a completely machine-independent IR, but that IR would be overly abstract and would increase the conceptual gap between ASTs and JVM code rather than decrease it.

1.1 Caveat Lector

This specification shows the *textual* form of IR instructions, but this does not imply that compilers using this IR should internally represent IR instructions as strings. IR instructions should be represented succinctly with appropriate data structures, whose design is the responsibility of the compiler-writer.

The provision of a grammar describing the IR does not imply that the IR should be parsed. The grammar is provided only for its *generative* capability: it succintly enumerates all possible programs in IR.

Finally, this document presents the IR is a bottom-up manner: details are introduced first, which are later aggregated to build tangible concepts. If you prefer a top-down approach, you should read this document backwards.

2 Types

The JVM requires each datum to have a properly declared type. It uses type declarations to ensure that operations its performs on data are meaningful. Grammar 1 shows that syntax for typenames in the IR.

Grammar 1 Typenames (see Table 1 on the facing page for an explanation of the letters used).

⟨*type*⟩	→	⟨*basic_type*⟩	
	|	⟨*method_type*⟩	
⟨*atomic_type*⟩	→	Z	BOOLEAN
	|	C	CHARACTER
	|	B | S | I | L	INTEGRAL TYPES
	|	F | D	FLOATING TYPES
⟨*reference_type*⟩	→	U	STRING
	|	AU	ARRAY OF STRING
	|	A⟨*atomic_type*⟩	ARRAY OF ⟨*atomic_type*⟩
⟨*basic_type*⟩	→	⟨*atomic_type*⟩	
	|	⟨*reference_type*⟩	
⟨*return_type*⟩	→	⟨*basic_type*⟩	
	|	V	VOID
⟨*method_type*⟩	→	(⟨*basic_type*⟩*)⟨*return_type*⟩	

Note: ⟨*type*⟩ does not represent all possible Java types since it does not support references to named classes and arrays of non-⟨*atomic_type*⟩, particularly arrays of arrays.

2.1 Examples

1. The function `factorial`, given in the assignment specification, with signature `int factorial(int)` has ⟨*type*⟩

   (I)I

Letter	Java type
Z	`boolean`
C	`char`
B	`byte`
S	`short`
I	`int`
L	`long`
F	`float`
D	`double`
V	`void`
U	`java.lang.String`
A	array of ⟨*atomic_type*⟩

Table 1: Letters used in ⟨*type*⟩s (see Grammar 1 on the facing page) and their meanings.

Likewise, the function `main`, given in the assignment specification, with signature `void main(void)` has ⟨*type*⟩

()V

2. A function which takes an array of integers and returns their sum has ⟨*type*⟩

   (AI)I

3. A function which checks whether the two strings arguments passed to it are palindromes has ⟨*type*⟩

   (UU)Z

3 Constants

Constants provide the building blocks from which other values can be computed. There are five types of ⟨*constant*⟩:

1. ⟨*boolean*⟩, which has two forms:

   TRUE and FALSE

2. ⟨*integer*⟩, a sequence of digits representing a number in the closed interval $\left[-2^{31}, 2^{31}-1\right]$. A minus sign precedes negative numbers. For example,

   123 and -456

3. ⟨*character*⟩, a single character from the Unicode character set enclosed in single quotes. For example,

   ‘a’

4. ⟨*string*⟩, a finite sequence of ⟨*character*⟩s enclosed in double quotes. For example,

   "Supercalifragilistic"

5. ⟨*floating_point*⟩, a finite sequence of decimal digits followed by a decimal point followed by a finite sequence of decimal digits. For example,

   3.14156 and 2.718282

Note:

1. The constants above have the following ⟨*type*⟩s, respectively:

   $$\mathsf{Z}, \mathsf{I}, \mathsf{C}, \mathsf{U}, \text{ and } \mathsf{F}$$

2. A compiler supporting extensions to the unnamed language may extend IR's definition of ⟨*constant*⟩ to include forms for ⟨*type*⟩s B, S, L, and D.

4 Operands

There is just one type of ⟨*operand*⟩: the ⟨*temporary*⟩.

4.1 Temporaries

If IR were an economy, **temporary variables** would be its "currency";[1] a) local variables, b) method parameters and c) the intermediate results of statement execution will all be stored in temporary variables (henceforth called "temporaries" or "temps").

Each method can have up to 65535 temporaries. Their names are drawn from the set

$$\{ \mathsf{T}_i \mid 0 < i < 65535 \}$$

Values of each ⟨*type*⟩, with the exception of ⟨*type*⟩s L and D, require one temporary. Values of ⟨*type*⟩s L and D require two consecutive temporaries.

Each temporary has one and only one ⟨*basic_type*⟩. The (implicit) type promotion rules provided for the unnamed language[CP07] do not apply; it is illegal to use a temporary of one ⟨*basic_type*⟩ in an instruction which expects a temporary of a different ⟨*basic_type*⟩. To effect type promotion, *explicit* type conversion operations must be performed.

[1] At the risk of belaboring a metaphor, IR instructions (introduced in Section 6) would be the agents of that economy.

4.1.1 Parameters and Local Variables

For each method, the set $\{\mathsf{T}_i\}$ is partitioned into three subsets:

1. The set of parameters $\{\mathsf{P}_i\}$ defined

   $$\mathsf{P}_i = \mathsf{T}_i \text{ where } 0 < i < p$$

   where p is the number of temporaries required to represent the parameters of the method. (Note, carefully, the words "required to represent" above).

2. The set of local variables $\{\mathsf{L}_i\}$ defined

   $$\mathsf{L}_i = \mathsf{T}_{i+p} \text{ where } 0 < i < l$$

   where l is the number of temporaries required to represent the local variables of the method.

   The local variables of a method shall be allocated to temporaries (starting from L_0) in the order that they are declared. This facilitates both easier debugging and easier marking.

3. The set of temporaries required to hold intermediate values

   $$\{\, \mathsf{T}_{i+p+l} \mid 0 < i < t \,\}$$

   where t is the number of temporaries, a number whose value is derived during the process of translation into IR.

Note:

1. Parameters (in $\{\mathsf{P}_i\}$) and local variables (in $\{\mathsf{L}_i\}$) have aliases (in $\{\mathsf{T}_i\}$) as defined by the equations in items (1) and (2) above. The IR is not required to use aliases, but should do so for clarity.

2. It is indeed possible to formulate programs which cause the compiler to run out of temporary variables. Compilers for the unnamed language do not have to handle this scenario.

5 Operators

At the IR level, operators are *typed*: the IR must specify *integer* addition or *floating point* addition, *not* merely addition.

Hence, IR operators have a two-part form comprising:

1. an ⟨*operator_type_specifier*⟩ representing the precise type on which the operation will be performed;

2. the basic operation, a ⟨*unary_operator*⟩ or a ⟨*binary_operator*⟩.

5.1 Unary

The names of unary operators are given by Grammar 2.

Grammar 2 Unary Operators

⟨*operator_type_specifier*⟩	→	⟨*atomic_type*⟩
⟨*basic_unary_operator*⟩	→	`-` | `!`
⟨*unary_operator*⟩	→	⟨*operator_type_specifier*⟩⟨*basic_unary_operator*⟩
	|	⟨*operator_type_specifier*⟩2⟨*operator_type_specifier*⟩

The unary operators `-` and `!` represent numerical negation and bitwise inversion, respectively. The form ⟨*atomic_type*⟩$_1$2⟨*atomic_type*⟩$_2$ represents a unary operator which converts its operand from ⟨*atomic_type*⟩$_1$ to ⟨*atomic_type*⟩$_2$.

Some of the unary operators whose names are enumerated by Grammar 2 are not meaningful; those that are meaningful are indicated by Table 2. Each column header shows the type of the operand to which the operator is applied (⟨*operator_type_specifier*⟩$_1$).

For operators `-` and `!`, the corresponding table entry contains a — if that operator cannot be applied to an operand of that type. Otherwise, the type of the result is shown.

For conversion operators, each row header shows the type of the result (⟨*operator_type_specifier*⟩$_2$). If it is illegal to form the unary conversion operator ⟨*operator_type_specifier*⟩$_1$2⟨*operator_type_specifier*⟩$_2$, the corresponding table entry will contain a —. Otherwise, the corresponding table entry will contain the type of the result produced (viz., ⟨*operator_type_specifier*⟩$_2$).

	Z	C	B	S	I	L	F	D
NEGATION AND INVERSION								
-	—	—	B	S	I	L	F	D
!	Z	—	B	S	I	L	—	—
CONVERSION OPERATORS								
2Z	Z	—	—	—	—	—	—	—
2C	—	C	C	C	C	C	—	—
2B	—	B	B	B	B	B	B	B
2S	—	S	S	S	S	S	S	S
2I	—	I	I	I	I	I	I	I
2L	—	L	L	L	L	L	L	L
2F	—	—	F	F	F	F	F	F
2D	—	—	D	D	D	D	D	D

Table 2: Unary operators. Each legal combination provides the ⟨*atomic_type*⟩ of the result in the appropriate table entry. The symbol — indicates illegal combinations.

5.2 Binary

The names of binary operators are given by Grammar 3. The operators shown have their usual interpretations. Comparison operators on strings use lexicographic ordering.

Grammar 3 Binary Operators

⟨*operator_type_specifier*⟩	→	⟨*atomic_type*⟩
	|	U
⟨*basic_binary_operator*⟩	→	+ | - | * | / | rem
	|	< | <= | == | != | >= | >
⟨*binary_operator*⟩	→	⟨*operator_type_specifier*⟩⟨*basic_binary_operator*⟩

Some of the binary operators whose names are enumerated by Grammar 3 are not meaningful; those combinations that are meaningful are indicated by Table 3. Each column header shows the type of the *pair* of operands to which the operator is applied. If it is illegal to apply an operator to that pair of types, the corresponding table entry will contain a —. Otherwise, the type of the result is shown.

	(Z, Z)	(C, C)	(B, B)	(S, S)	(I, I)	(L, L)	(F, F)	(D, D)	(U, U)
					ARITHMETIC OPERATORS				
+	—	C	B	S	I	L	F	D	U
-	—	C	B	S	I	L	F	D	—
*	—	C	B	S	I	L	F	D	—
/	—	C	B	S	I	L	F	D	—
rem	—	C	B	S	I	L	F	D	—
					COMPARISON OPERATORS				
<	—	Z	Z	Z	Z	Z	Z	Z	Z
<=	—	Z	Z	Z	Z	Z	Z	Z	Z
==	Z	Z	Z	Z	Z	Z	Z	Z	Z
!=	Z	Z	Z	Z	Z	Z	Z	Z	Z
>=	—	Z	Z	Z	Z	Z	Z	Z	Z
>	—	Z	Z	Z	Z	Z	Z	Z	Z

Table 3: Binary operators. Each legal combination provides the ⟨*type*⟩ of the result in the appropriate table entry. The symbol — indicates illegal combinations.

6 Instructions

Instructions are the agents of the IR economy. The six types of ⟨*instruction*⟩ in the IR (see Grammar 4 on the next page) fall into two major categories:

state modifying instructions: These assign values to parameters, local variables, and temporary variables. In this IR, ⟨*assignment*⟩s, ⟨*call*⟩s, and ⟨*print*⟩s modify state.

control transferring instructions: These transfer control to other parts of the program. In this IR, ⟨*label*⟩s, ⟨*jump*⟩s, ⟨*call*⟩s and ⟨*return*⟩s transfer control.

Grammar 4 Statements

⟨*instruction*⟩	→	⟨*assignment*⟩	SEE GRAMMAR 5 ON THE FACING PAGE
	|	⟨*label_inst*⟩	SEE GRAMMAR 6 ON THE FACING PAGE
	|	⟨*jump*⟩	SEE GRAMMAR 6 ON THE FACING PAGE
	|	⟨*call*⟩	SEE GRAMMAR 7 ON PAGE 12
	|	⟨*return*⟩	SEE GRAMMAR 7 ON PAGE 12
	|	⟨*print*⟩	SEE GRAMMAR 8 ON PAGE 12

6.1 Assignment

There are seven types of assignments statement, as described in Grammar 5:

1. Assignment of a ⟨*constant*⟩ to an ⟨*operand*⟩. This operation is fundamental: since all other types of assignment instruction use ⟨*operand*⟩ operands only, a ⟨*constant*⟩s must first be loaded into a temporary variables, before they can be used as an ⟨*operand*⟩s.

   The ⟨*type*⟩s of ⟨*operand*⟩ (as declared in the ⟨*temporary_list*⟩ defined in Grammar 9 on page 13) and ⟨*constant*⟩ must be the same.

2. Creation of a new array of the given ⟨*atomic_type*⟩ (or U) with ⟨*integer*⟩ elements. No assumption may be made about the initial value of any element.

   ⟨*operand*⟩ must have ⟨*type*⟩ A⟨*atomic_type*⟩ (or AU).

3. Assignment of one ⟨*operand*⟩ to another. The ⟨*type*⟩s of the ⟨*operand*⟩s must be the same.

4. Assignment to an element of an array.

   If the ⟨*operand*⟩ being assigned has ⟨*type*⟩, then the array whose element is being assigned to must have type A⟨*type*⟩. The index ⟨*operand*⟩ must have type I.

5. Assignment from an element of an array.

   If the array, whose element is being assigned, has type A⟨*type*⟩, the ⟨*operand*⟩ being assigned to must have ⟨*type*⟩. The index ⟨*operand*⟩ must have type I.

6. Assignment of the result of a unary operation.

   Values for $\langle\textit{unary\_operator}\rangle$ are given in Grammar 2 on page 8. The constraints on the $\langle\textit{type}\rangle$s of the $\langle\textit{operand}\rangle$s are given in Table 2 on page 8.

7. Assignment of the result of a binary operation.

   Values for $\langle\textit{binary\_operator}\rangle$ are given in Grammar 3 on page 9. The constraints on the $\langle\textit{type}\rangle$s of the $\langle\textit{operand}\rangle$s are given in Table 3 on page 9.

Grammar 5 Assignments

$\langle\textit{assignment}\rangle$	$\rightarrow$	$\langle\textit{operand}\rangle$:= $\langle\textit{constant}\rangle$
	|	$\langle\textit{operand}\rangle$:= NEWARRAY $(\langle\textit{atomic\_type}\rangle \mid \mathsf{U})$ $\langle\textit{integer}\rangle$
	|	$\langle\textit{operand}\rangle$:= $\langle\textit{operand}\rangle$
	|	$\langle\textit{operand}\rangle$ [$\langle\textit{operand}\rangle$] := $\langle\textit{operand}\rangle$
	|	$\langle\textit{operand}\rangle$:= $\langle\textit{operand}\rangle$ [$\langle\textit{operand}\rangle$]
	|	$\langle\textit{operand}\rangle$:= $\langle\textit{unary\_operator}\rangle$ $\langle\textit{operand}\rangle$
	|	$\langle\textit{operand}\rangle$:= $\langle\textit{operand}\rangle$ $\langle\textit{binary\_operator}\rangle$ $\langle\textit{operand}\rangle$

6.2 Jump

Control flow within a function is effected by (conditional and unconditional) $\langle\textit{jump}\rangle$s to $\langle\textit{label}\rangle$s. Their forms are shown in Grammar 6.

A conditional $\langle\textit{jump}\rangle$ requires a single $\langle\textit{operand}\rangle$ of $\langle\textit{type}\rangle$ Z. If the value of $\langle\textit{operand}\rangle$ is TRUE, control is transfered to $\langle\textit{label}\rangle$. Otherwise, control proceeds to the next instruction in static order.

The following points regarding $\langle\textit{label}\rangle$s should be noted:

1. $\langle\textit{label}\rangle$s do not have to be declared prior to the $\langle\textit{jump}\rangle$ instructions which transfer control to them.

2. If there are n $\langle\textit{label}\rangle$s used in the translation of a function, they should have $\langle\textit{integer}\rangle$s in the interval $[0, n-1]$.

3. Each $\langle\textit{label\_inst}\rangle$ instruction must introduce a unique $\langle\textit{label}\rangle$ within a $\langle\textit{function}\rangle$ (defined in Grammar 9 on page 13).

4. Each $\langle\textit{label\_inst}\rangle$ is a perfectly legitimate instruction, by itself. It may be considered a “no-op” in this regard.

Grammar 6 Jumps

$\langle\textit{label}\rangle$	$\rightarrow$	L$\langle\textit{integer}\rangle$
$\langle\textit{label\_inst}\rangle$	$\rightarrow$	$\langle\textit{label}\rangle$:
$\langle\textit{jump}\rangle$	$\rightarrow$	IF $\langle\textit{operand}\rangle$ GOTO $\langle\textit{label}\rangle$
	|	GOTO $\langle\textit{label}\rangle$

6.3 Call

The transfer of control between functions is effected by ⟨*call*⟩ and ⟨*return*⟩ instructions.

The ⟨*call*⟩ instruction transfers control to the function named by the ⟨*member_name*⟩ operand. The form of ⟨*member_name*⟩s are precisely those strings recognized by the regular expression `[a-zA-Z_][0-9a-zA-Z_]*`. Section 9 provides examples.

In order for a ⟨*call*⟩ statement to be correctly "typed":

1. the ⟨*type*⟩s of its ⟨*operand*⟩s must match the ⟨*type*⟩s of the parameters of the callee, in order.

2. the type of the ⟨*operand*⟩ which holds the return value (or the ⟨*type*⟩ V otherwise) must match the type of the value that the callee returns.

Note: If the callee does not have a return ⟨*type*⟩ of V, the second type of the ⟨*call*⟩ statement (which stores the returned value in an ⟨*operand*⟩) must be used. This is because, by point (2) above, the absence of an ⟨*operand*⟩ in which to store the returned value implies a return ⟨*type*⟩ of V. Such a ⟨*call*⟩ would be ill-typed (also by point (2) above).

Grammar 7 Calls and Returns

⟨*call*⟩	→	CALL ⟨*member_name*⟩(⟨*operand*⟩*)
	|	⟨*operand*⟩:= CALL ⟨*member_name*⟩(⟨*operand*⟩*)
⟨*return*⟩	→	RETURN
	|	RETURN ⟨*operand*⟩

6.4 Output

The output statements are self-explanatory; only a few miscellaneous points merit mention:

1. The ⟨*type*⟩ of the ⟨*operand*⟩ must match the given ⟨*atomic_type*⟩.

2. To print strings, the "U" forms must be used and the ⟨*operand*⟩ must have ⟨*type*⟩ U.

3. The "LN" forms print a new line (`\n`) after printing their ⟨*operand*⟩.

Grammar 8 Output Statements

⟨*print*⟩	→	PRINT⟨*atomic_type*⟩ ⟨*operand*⟩
	|	PRINTU ⟨*operand*⟩
	|	PRINTLN⟨*atomic_type*⟩ ⟨*operand*⟩
	|	PRINTLNU ⟨*operand*⟩

7 Functions

Grammar 9 summarizes the form of a function in the IR, which consists of:

1. its name, a $\langle\textit{member\_name}\rangle$ (as defined in Section 6.3);
2. its signature, a $\langle\textit{method\_type}\rangle$;
3. a list of $\langle\textit{temporary}\rangle$ variables (and their $\langle\textit{type}\rangle$s) used by the $\langle\textit{instruction}\rangle$s in the $\langle\textit{instruction\_list}\rangle$;
4. a list of $\langle\textit{instruction}\rangle$s (as specified in Grammar 4 on page 10) that comprise the translated statements of the function.

Grammar 9 Function

$$\begin{aligned}
\langle\textit{function}\rangle &\rightarrow \mathsf{FUNC}\ \langle\textit{member\_name}\rangle\ \langle\textit{method\_type}\rangle\ \langle\textit{function\_body}\rangle \\
\langle\textit{function\_body}\rangle &\rightarrow \{\ \langle\textit{temporary\_list}\rangle\ \langle\textit{instruction\_list}\rangle\ \} \\
\langle\textit{temporary\_list}\rangle &\rightarrow (\langle\textit{temporary\_declaration}\rangle\,\mathtt{;})^{*} \\
\langle\textit{temporary\_declaration}\rangle &\rightarrow \mathsf{TEMP}\ \langle\textit{temporary}\rangle\,\mathtt{:}\,\langle\textit{basic\_type}\rangle \\
\langle\textit{instruction\_list}\rangle &\rightarrow (\langle\textit{instruction}\rangle\,\mathtt{;})^{*}
\end{aligned}$$

Note: For each $\langle\textit{function}\rangle$, the combination of the $\langle\textit{member\_name}\rangle$ and $\langle\textit{method\_type}\rangle$, must be unique.

8 Programs

A program consists of:

1. its name, a $\langle\textit{class\_name}\rangle$, whose derivation is implementation defined. For example, the program name can be derived from the base name of the source file containing the program;

   A $\langle\textit{class\_name}\rangle$ is a sequence of Java identifiers, separated by periods. $\langle\textit{class_name}\rangle$s represent the fully-qualified names of classes and interfaces. For example, `java.lang.String`.

2. a sequence of one or more $\langle\textit{function}\rangle$s.

Grammar 10 summarizes the overall form of a program represented in IR.

Grammar 10 Program

$$\langle\textit{program}\rangle \rightarrow \langle\textit{class\_name}\rangle\ \langle\textit{function}\rangle^{+}$$

9 Example

Figures 2 and 3 provide the translation of the sample program (which computes factorials) which was provided in the language specification.

The examples show legal ⟨*function*⟩ clauses corresponding to functions `factorial` and `main`.

9.1 Function `factorial`

Function `factorial` has just 1 parameter which will be stored in P_0 (also known as T_0).

There are no local variables.

```
// sample.c
int factorial(int n)
{
    if (n == 1)
    {
        return 1;
    }
    else
    {
        return (n
        *factorial(n-1));
    }
}
```

(a) Function `factorial`, as given in the language specification.

```
FUNC factorial (I)I
{
    TEMP 0:I;
    TEMP 1:Z;
    TEMP 2:I;
    T2 := 1;
    T1 := P0 I!= T0;
    IF T1 GOTO L1;
    T2 := 1;
    T2 := P0 I- T2;
    T2 := CALL factorial(T2);
    T2 := P0 I* T2;
    RETURN T2;
    GOTO L2;
L1:;
    T2 := 1;
    RETURN T2;
L2:;
}
```

(b) A translation of function `factorial` into IR.

Figure 2: A translation of the function `factorial`, as given in the language specification, into IR.

Note: Although the number of parameters and local variables are known prior to translation of a function's body, the total number of temporary variables will remain unknown until the entire function body has been translated. Only then can the local variable table be emitted.

9.2 Function `main`

The translation of function `main` is shown in Figure 3.

```
void main(void)
{
    print "The ..."
          " of 8 is ";
    print factorial(8);
}
```

(a) Function `main`, as given in the language specification.

```
FUNC main ()V
{
    TEMP 0:U;
    TEMP 1:I;
    T0 := “The factorial...”;
    PRINTU T0;
    T1 := 8;
    T1 := CALL factorial(T1);
    PRINTI T1;
}
```

(b) A translation of function `main` into IR.

Figure 3: A translation of function `main`, as given in the language specification, into IR.

9.3 The Top Level

Since the translation of a program into IR is simply the juxtaposition of its translated functions (by Grammar 10 on page 13), the IR provided above (in the order given) provides the translated program.

References

[CP07] J. Corless and D. Pereira. Semantic rules for the Unnamed Language. `http://www.csc.uvic.ca/content/csc435F07/?q=system/files/refmanual11.pdf`, 2007. Cited on page(s) 6

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Prentice Hall PTR, 1999. Cited on page(s) 3