Intermediate Language Specification

The Instruction Set

The following table describes the available instructions, their formats, and their effects. Any instruction not fully defined here will be discussed below.

General Format				
INSTRUCTION	SRC1	SRC2	DEST	Effect
Arithmetic Operations				
ADD	op1	op2	op3	$op3 := op1 + op2$
SUB	op1	op2	op3	$op3 := op1 - op2$
MULT	op1	op2	op3	$op3 := op1 * op2$
DIV	op1	op2	op3	$op3 := op1 DIV op2$
NEG	op1	-	op3	$op3 := -op1$
Logical Operations				
NOT	op1	-	op3	$if(op1 <> 0)op3 := 0$ else $op3 := 1$
Relational Operations				
EQ	op1	op2	op3	$op3 := op1 = op2$
GT	op1	op2	op3	$op3 := op1 > op2$
LT	op1	op2	op3	$op3 := op1 < op2$
GE	op1	op2	op3	$op3 := op1 >= op2$
LE	op1	op2	op3	$op3 := op1 <= op2$
NE	op1	op2	op3	$op3 := op1 <> op2$
Assignment				
ASSIGN	op1	op2	op3	$op3 := op1$
Control Forms				
LABEL	op1	-	-	The next statement is labeled with op1
BR	-	-	op3	goto op3
BREQ	op1	op2	op3	$if(op1 = op2)$ goto op3
BRGT	op1	op2	op3	$if(op1 > op2)$ goto op3
BRLT	op1	op2	op3	$if(op1 < op2)$ goto op3
BRGE	op1	op2	op3	$if(op1 >= op2)$ goto op3
BRLE	op1	op2	op3	$if(op1 <= op2)$ goto op3
BRNE	op1	op2	op3	$if(op1 <> op2)$ goto op3
HALT	-	-	-	Immediately halt execution

The Instruction Set (continued)

Procedure Call Operations				
ARGS	op1	-	-	The next call requires op1 arguments.
REFOUT	op1	-	-	Pass op1 by reference.
VALOUT	op1	-	-	Pass op2 by value.
CALL	op1	-	-	Call the procedure named op1
PROCENTRY	op1	op2	op3	Mark beginning of the procedure named op1
ENDPROC	-	-	-	Mark the end of the current procedure
RETURN	-	-	-	Return control to the caller
Additional Statements				
BOUND	op1	op2	op3	$if(op3 < op1) or (op3 > op2)$ then HALT
ADDR	op1	-	op3	op3 := address of op1
GLOBAL	op1	op2	-	Declare op1 as a global of size op2
STRING	op1	op2	-	Associate string op1 with label op2
COMMENT	op1	-	-	op1 is a comment

Operand Types

Operands in the intermediate language are type-value pairs. Depending on the type of an operand, the value may be interpreted in various ways. Each operand in the instructions above may have one of several different types. Please note that not every type is allowable for a given operand of an instruction. See below for allowable operand types.

Type	Meaning
LOCAL	Local variable
GLOB	Global variable
ITEMP	Compiler temporary (for integers)
FTEMP	Compiler temporary (for floats)
CONS	Absolute constant
INDR	Operand whose address is in a compiler temporary
LABEL	Label attached to a statement
REFARG	Reference argument
VALARG	Value argument
STRING	String literal
NONE	Placeholder operand

LOCAL

The value field of a LOCAL operand is interpreted as an index into an imaginary 0-based array of words for local variables. There is no need to declare individual local variables; however, the number of words in the entire set of local variables must be declared in the PROCENTRY instruction. See below for more information on procedure calling conventions.

LOCAL operands may be used as both lvalues and rvalues and are preserved across procedure calls and branches.

Example: (local 3)

GLOB

The value field of a GLOB operand is interpreted as the unquoted name of a global variable. All global variables must be declared at some point in the output with an appropriate GLOBAL instruction.

GLOB operands may be used as both lvalues and rvalues and are preserved across procedure calls and branches.

Example: (glob foo)

TEMP and FTEMP

The value field of a TEMP operand is interpreted as an index into an imaginary 0-based array of compiler temporaries. There is no need to declare compiler temporaries at any time.

TEMP operands may be used as both lvalues and rvalues. However, once a TEMP operand is used as an rvalue its value is no longer preserved. It is an error to use the same temporary as an lvalue more than once. TEMP operands are not preserved across procedure calls and branches.

Example: (temp 42)

CONS and FCONS

The value field of a CONS operand is interpreted as an actual integer value. Please note that named program constants defined via the CONST keyword are not permissible in this context. CONS operands may be used an unlimited number of times in any appropriate context.

CONS operands may be used as rvalues but never as lvalues. Preservation of CONS operands across procedure calls and branches is inherently undefined.

Example: (cons -57732)

<u>INDR</u>

The value field of an INDR operand is interpreted as the index of a compiler temporary holding the address of the operand itself. Using a temporary in an INDR context as an rvalue causes the value in the compiler temporary to be lost. Using a temporary in an INDR context as an lvalue does NOT cause the value in the compiler temporary to be destroyed.

INDR operands may be used as both lvalues and rvalues. In all other ways, INDR operands are subject to the same restrictions as TEMP operands.

Example: (indr 51)

<u>LABEL</u>

The value field of a LABEL operand is interpreted as the unsigned integral index into an imaginary 0-based array of labels. It is legal to skip indices in this imaginary array while declaring and using labels. For example, it is permissible for labels with indices 0 and 2 to be used in the program, but for no reference to a label with index 1 to appear.

LABEL operands are neither lvalues nor rvalues and are valid only in a small number of contexts. The absolute location of LABEL operands is guaranteed to be consistent across all procedure calls and branches.

Example: (label 21)

<u>REFARG</u>

The value field of a REFARG operand is interpreted as an index into an imaginary 0-based array of *all* arguments to the current procedure.

REFARG operands may be used as both lvalues and rvalues. Use of a REFARG as an lvalue causes the stored value to be propagated to the calling procedure. REFARG operands are preserved across procedure calls and branches.

Example: (refarg 1)

<u>VALARG</u>

The value field of a VALARG operand is interpreted as an index into an imaginary 0-based array of *all* arguments to the current procedure.

VALARG operands may be used as both lvalues and rvalues. Use of a VALARG as an lvalue causes no observable effects in the calling procedure. VALARG operands are preserved across procedure calls and branches.

Example: (valarg 3)

STRING

The value field of a STRING operand is interpreted as a double-quoted (") string and the number of characters between the opening and closing " characters must not exceed 63.

STRING operands may not be used as values of any kind. Preservation of STRING operands across procedure calls and branches is inherently undefined.

Example: (string "This is a string.")

NONE

The value field of a NONE operand is ignored but must not be empty.

Use of a NONE operand as a value of any kind will cause undefined results.

Example: (none none)

Valid Operand Types

Each instruction has restrictions on which type(s) each of its operands may have. For purposes of brevity, we define the following classes of operand types:

```
ADDRESSABLE : LOCAL | GLOB | INDR | REFARG | VALARG

LVALUE : ADDRESSABLE | TEMP

RVALUE : LVALUE | CONS
```

and the following classes of instructions:

```
BINARY : ADD | SUB | MULT | DIV | EQ | GT | LT | GE | LE | NE

UNARY : NEG | NOT | ASSIGN

CONDITIONAL : BREQ | BRGT | BRLT | BRGE | BRLE | BRNE

NOARG : HALT | ENDPROC | RETURN
```

Instruction	op1	op2	op3
BINARY	RVALUE	RVALUE	LVALUE
UNARY	RVALUE	NONE	LVALUE
LABEL	LABEL	NONE	NONE
BR	NONE	NONE	LABEL
CONDITIONAL	RVALUE	RVALUE	LABEL
NOARG	NONE	NONE	NONE
ARGS	CONS	NONE	NONE
REFOUT	ADDRESSABLE	NONE	NONE
VALOUT	RVALUE, LABEL	NONE	NONE
CALL	GLOB	NONE	NONE
PROCENTRY	GLOB	CONS	CONS
BOUND	RVALUE	RVALUE	RVALUE
ADDR	ADDRESSABLE	NONE	LVALUE
GLOBAL	GLOB	CONS	NONE
STRING	STRING	LABEL	NONE
COMMENT	STRING	NONE	NONE

Lexical Format of Instructions

Instructions, as indicated above, consist of an operation followed by zero to three operands. Operations and operand types are not case-sensitive. Operand values are case-sensitive and case-preserving if the operand is of type GLOB or STRING. The instructions may be presented to the back-end in any of the following formats:

```
operation op1:op1 op2:op2 op3:op3

operation op1 op1 op2 op2 op3 op3

operation op1, op1 op2, op2 op3, op3

operation (op1, op1) (op2, op2) (op3, op3)

operation (op1 op1) (op2 op2) (op3 op3)

(operation (op1 op1) (op2 op2) (op3 op3))
```

It is likely that other formats are also acceptable. Please note that if an instruction requires fewer than three operands, and the required operands are left-justified (for example, op1 and op2 are required but op3 is not), then only the required operands need be included. If the operands required are *not* left-justified, then one or more NONE operands must be used as placeholders.

Detailed Discussion of Selected Instructions

Some instructions are not easily described in a single line. This section presents full descriptions of the GLOBAL, and STRING instructions, and a detailed discussion of the procedure calling conventions.

The STRING and GLOBAL Instructions

These two instructions are special declarations. Simply put, you must place one such declaration at some place in your instruction stream for each string and global you use, respectively. These declarations need not be placed before a use of the indicated variable; the back-end will reorder them as necessary. The STRING instruction is used to associate a string with a label. The label is an ordinary integer label like all others you use in your instruction stream. The purpose of the STRING instruction is to allow you to pass strings to procedures, such as WriteString. When you do so, however, you must pass the label rather than the string itself. You should not worry about producing duplicate STRING instructions; the back-end will cause duplicates to be eliminated. The purpose of the GLOBAL instruction is similar; it defines the size of the global variable (for example, an array may require 30 storage units). If you do not indicate a size, the default is 1 word (integer). If you use a global variable without placing a GLOBAL instruction corresponding to it in the instruction stream, or if you declare the same GLOBAL twice, the generated assembly code may be incorrect.

Procedure Calling Conventions

Procedure calls require coordination between different pieces of code in several areas of the compiler. Therefore, the intermediate language's calling conventions are explicitly stated here. Failure to adhere to these calling conventions may cause erroneous or undefined behaviour by the back-end. Note that there is no requirement here for the front-end to know the sizes of data objects, except as a multiple of the word size or INTEGER type. This allows output from the front-end to be used by one or more different back-ends without modification. If you choose to implement data types of sizes which are not integer multiples of the machine's native word size, you will have to modify the back-end and possibly provide additional instructions.

- The Caller
  The caller must do three things:
  1. Issue an ARGS instruction. This operation tells the back-end how many word-sized arguments will be passed.
  2. Issue one VALOUT or REFOUT instruction for each argument being passed. Note that the following are equivalent:

     ```
     REFOUT (op1)
     ```

```
ADDR (op1) (none none) (temp i)
VALOUT (temp i)
```

Nevertheless, you must ALWAYS use REFOUT to pass arguments by reference.

3. Issue a CALL instruction with the procedure name as op1.

As a special case, you may pass labels, but only by value and only when they refer to string literals.

- The Callee
  The callee need only mark its beginning and end, as follows:
  - The beginning should be a PROCENTRY instruction with the following operands:
    * op1: The name of the procedure
    * op2: The number of word-size parameters it takes
    * op3: The number of word-sized locals.
  - The end should be an ENDPROC, which takes no arguments.
- Boilerplate calling code:

```
PROCENTRY (glob FooFunc) (cons 2) (cons 3)
... code for FooFunc ...
ENDPROC
... code for main ...
ARGS (cons 2)
VALOUT (temp 6)
REFOUT (glob variable)
CALL (glob FooFunc)
... code for main ...
```

Note that the back-end will consider all procedures to be in the global scope. Therefore, if procedures are nested in thus in different scopes, they must not have the same name.