Compiler Construction

ICode

Lennart Andersson

Revision 2012-02-16

2012

ICode program

Abstract grammar

Code ::= Instruction*;
abstract Instruction;

CFG

code → *instruction**

Semantics

By default, the instructions in a program are executed in sequence. Some instructions will override this behavior, e.g., jumps.

Formatting conventions

Each instruction is written on one line and intented by a tab. Arguments are separated by tabs.

Examples

```
ADDINT  3    7    #1
SUBINT  #1   6    #2
MOV     #2   #3
```


Label declarations

Abstract grammar

LabelDecl: Instruction ::= <Label:String>;

CFG

instruction → *LABEL*':'

Tokens

LABEL → $[a-zA-Z][a-zA-Z0-9\_]*$

Semantics

A LabelDecl serves as a name of the next instruction. It can be used for jumping to that instruction from somewhere else in the program. All LabelDecls in the program must have unique identifiers.

Formatting conventions

A LabelDecl is either written on a line of its own, or on the same line as the instruction it names.

Examples

```
L1:     ADDINT  1    2    var(0,0)
```


The Move instruction

Abstract grammar

Move: Instruction ::= Operand Address;
abstract Operand;
abstract Address: Operand;

CFG

instruction → `"MOV"` *operandaddress*

Semantics

A Move instruction copies the value of the Operand into the memory location at Address.

Examples

```
MOV   3    #1   // copy the value 3 into the temporary
                // variable #1
MOV   #1   #2   // copy the value of the temporary
                // variable #1 into #2
```

Integer constants

Abstract grammar

IntConst: Operand ::= <Value:String>;

CFG

Operand → *INT*

Tokens

INT → $[0-9]+$

Semantics

Integer constants are written as literal values, as in the example below

Examples

```
MOV    3        #1      // copy the value 3 into the
                        // temporary variable #1
```


Temporary variables

Abstract grammar

Temp: Address ::= <Number:int>;

CFG

address → `"#"` *INT*

Semantics

Temp represents a temporary variable, used for storing intermediate results in computations of expressions. Number refers to the number of the temporary variable in the currently executing method. The temporaries are enumerated from 0 and upwards for each method.

Formatting conventions

Examples

```
MOV    5        #4      // copy the value 5 into the
                        // temporary variable #4
```


Variable, Parameter

Abstract grammar

Variable: Address ::= <Levels:int><Number:int>;
Parameter: Variable

CFG

address → (`"var"` | `"par"`)'('*INT*','*INT*')'

Semantics

Variable and **Parameter** are used for addressing variables and formal parameters declared in the current or a statically enclosing activation (following static links).
Levels is the static level difference for this use relative to its declaration. For local variables this value is 0.
Number refers to the variable or parameter number, where numbering starts from 0.

Variable, Parameter

Examples

```
MOV  var(0,2)   #0    // copy the value of the 3rd variable
                      // in the current frame into #0
MOV  par(1, 0)  #1    // copy the value of the first parameter in
                      // the immediately statically enclosing
                      // block into #1.
```

Argument

Abstract grammar

Argument: Address ::= <Number:int>;

CFG

address → `"arg" "("` *INT*')'

Semantics

Argument is used for addressing arguments when preparing for a procedure call.
arg(n) refers to argument number n. The arguments are numbered from 0 and upwards.

Examples

MOV 8 arg(0) // copy the value 8 into argument number 0.

Method calls

Abstract grammar

Call: Instruction ::= <Label:String> <Levels:int>;

CFG

instruction → `"CALL"` *LABEL*'levels'`"("` *INT*')'

Semantics

CALL stands for "call method". The execution will continue at **Label** which should be the first instruction of a method. The value of Levels and the return address are transferred to the called method.

Levels denotes how many levels up the called method's statically enclosing block is relative to the calling method.

E.g., levels(0) means that the called method is declared in block where the call occurs. The name of a method is declared one level above the body of the method. A direct recursive call of a method from the body block will have levels(1).

Method calls

Semantics

If the method has arguments, memory for them must be allocated using ALLOC and their values moved to the proper locations. After returning this memory should be deallocated using DEALLOCATE.
The called method should start with an ENTER instruction for allocating its activation record, and should end with a RETURN instruction. At execution of RETURN the execution continues at the instruction following the corresponding CALL.

Examples

CALL m levels(0)

Result, Returned

Abstract grammar

Result: Address;
Returned: Address;

CFG

address → `"result"` | 'returned'

Semantics

Result is used as the address for storing the return value in the called procedure and Returned is used by the calling procedure to access the returned value. These two addresses will refer to the same register or memory location on the target hardware, but the location specification may differ.

Examples

```
MOV    8          result    // copy the value 8 into the return locat
MOV    returned   #0        // copy the returned value to #0
```

Allocate, Deallocate

Abstract grammar

Allocate: Instruction;
Deallocate: Instruction;

CFG

instruction → ("ALLOC" | "DEALLOC")'size'"(" *INT*')'

Semantics

These instructions are used to allocate/deallocate memory or registers for actual arguments.

Examples

```
ALLOC     size(1)          // allocate one argument
MOV       4       arg(0)   // set argument to 4
CALL      m       levels(1)// call m
DEALLOC   size(1)          // deallocate one argument
```


Enter

Abstract grammar

Enter: Instruction ::= <Vars:int> <Temps:int>

CFG

instruction → "ENTER" "vars" "("*INT*")" "temps""("*INT*")

Semantics

ENTER allocates and initializes a new activation record on the stack. The return address that the CALL instruction generates is saved in the activation record.
The new record is large enough to hold Vars variables and Temps temporaries.
The code for a procedure must start with an ENTER instruction.

Examples

m: ENTER vars(3) temps(1)

Return

Abstract grammar

Return: Instruction;

CFG

instruction → "RETURN"

Semantics

RETURN stands for "return to calling method".
The instruction deallocates the current activation record and jumps to the instruction after the corresponding CALL instruction. The return address has previously been saved in the activation record by the ENTER instruction.

Examples

RETURN

Arithmetic instructions

Abstract grammar

abstract BinOpr: Instruction ::= Operand1:Operand
Operand2:Operand Address;
abstract IntOpr: BinOpr;
IntAdd: IntOpr;
IntSub: IntOpr;
IntMul: IntOpr;
IntDiv: IntOpr;

CFG

instruction → "INTADD" *operand operand address*
instruction → "INTSUB" *operand operand address*
instruction → "INTMUL" *operand operand address*
instruction → "INTDIV" *operand operand address*

Semantics

the operands must be integers
the result is an integer that is stored in Addr

Arithmetic instructions

Examples

INTADD	op1 op2 addr // addr = op1 + op2
INTSUB	op1 op2 addr // addr = op1 - op2
INTMUL	op1 op2 addr // addr = op1 * op2
INTDIV	op1 op2 addr // addr = op1 / op2

Jumps

Abstract grammar

Jmp: Instruction ::= <Label:String>;
abstract JmpCond: Instruction ::= Operand <Label:String>;
JmpF: JmpCond;
JmpT: JmpCond;

CFG

instruction → `"JMP"` *LABEL*
instruction → `"JMPF"` *operand LABEL*
instruction → `"JMPT"` *operand LABEL*

Semantics

Unconditional jump (ovillkorligt hopp):
JMP L // jump to the instruction at L
.
Conditional jumps (villkorliga hopp)
JMPF Op L // jump to L if Op is false
JMPT Op L // jump to L if Op is true

Integer comparisons

Abstract grammar

abstract BoolOpr: BinOpr ;
IntEq: BoolOpr;
IntNe: BoolOpr;
IntLt: BoolOpr;
IntLe: BoolOpr;
IntGt: BoolOpr;
IntGe: BoolOpr;

CFG

instruction → `"INTEQ"` *operand operand address*
instruction → `"INTNE"` *operand operand address*
instruction → `"INTLT"` *operand operand address*
instruction → `"INTLE"` *operand operand address*
instruction → `"INTGT"` *operand operand address*
instruction → `"INTGE"` *operand operand address*

Integer comparisons

Semantics

Store the result as false or true
INTEQ Op1 Op2 Addr // Addr = Op1 == Op2
INTNE Op1 Op2 Addr // Addr = Op1 != Op2
INTLT Op1 Op2 Addr // Addr = Op1 < Op2
INTLE Op1 Op2 Addr // Addr = Op1 <= Op2
INTGT Op1 Op2 Addr // Addr = Op1 > Op2
INTGE Op1 Op2 Addr // Addr = Op1 >= Op2