

## Chapter 7

# Interpreters for Imperative Languages

### 7.1 A First Simple Imperative Language

Let us first choose an extremely simple imperative language with three constructs:

- variable definition `def`, defining a single variable. Example: `def x`.
- assignment `:=` to assign a value to a variable. Example: `x := 3`
- sequential composition `;` to compose two constructs. Example: `def x ; x:= 3`

Imperative programs access and modify a global state. When writing an interpreter we thus need to model this environment. Thus, we first provide auxiliary predicate to store and retrieve variable values in an environment:

#### Program 7.1.1

```
/* def(OldEnv, VariableName, NewEnv) */
def(Env,Key,[Key/undefined|Env]).

/* store(OldEnv, VariableName, NewValue, NewEnv) */
store([],Key,Value,[Key/Value]) :- print(assigning_undefined_var(Key,Value)),nl.
store([Key/_Value2|T],Key,Value,[Key/Value|T]).
store([Key2/Value2|T],Key,Value,[Key2/Value2|BT]) :-
    Key \== Key2, store(T,Key,Value,BT).

/* lookup(VariableName, Env, CurrentValue) */
lookup(Key,[],_) :- print(lookup_var_not_found_error(Key)),nl,fail.
lookup(Key,[Key/Value|_T],Value).
lookup(Key,[Key2/_Value2|T],Value) :-
    Key \== Key2,lookup(Key,T,Value).
```

We suppose that `store` and `lookup` will be called with all but the last argument bound ?

```
| ?- store(Old,x,X,[x/3,y/2]).
X = 3,
Old = [x/_A,y/2] ? ;
no
```

The interpreter for our small language is now extremely simple, consisting of the following three clauses.

### Program 7.1.2

```
int(X:=V,In,Out) :- store(In,X,V,Out).
int('def'(X),In,Out) :- def(In,X,Out).
int((X;Y),In,Out) :- int(X,In,I2), int(Y,I2,Out).
```

In order to write programs more conveniently, without developing a full-fledged parser, we use operator declarations:

### Program 7.1.3

```
:- op(900,xfy,':='). /* + has 500, * has 400, ; has 1100 as priority */
:- op(750,fx,'def').
test(R) :- int( (def x; x:= 5; def z; x:= 3; z:= 2) , [],R).
```

The first number of an `op/3` declaration indicates the precedence of an operator, the lower the number the tighter it binds. The second argument indicates whether it is a unary or binary operator and also how it associates. Here *f* indicates the position of the operator. Thus `fx` indicates a prefix, unary operator. `xfy` indicates a right-associative, binary operator.

It can be used as follows:

```
| ?- int((def x; x:= 5; def z; x:= 3; z:= 2) , [],R).
R = [z/2,x/3]

| ?- int( (def x; x:= 5; def z; x:= 3; z:= X) , In, [z/2,x/RX]).
X = 2,
In = [],
RX = 3 ? ;
no
```

Note, that the interpreter works again compositionally for the sequential composition. Also, observe how the environment is “threaded” (i.e., passed from one recursive call to the next).

**Exercise 7.1.4** Rewrite the above interpreter using DCG notation.

Now let us allow to evaluate expressions in the right hand side of an assignment. To make our interpreter a little bit simpler, we suppose that variables used in the right hand side are preceded by a “\$”. E.g., `x := $x+1`.

We need to write a separate Prolog predicate to evaluate arguments. This is again a common pattern in interpreter development: we have one predicate to execute statements, modifying an environment, and one predicate to evaluate expressions returning a value. If the expressions in the language to be interpreted are side-effect free, the `eval` predicate need not return a new environment. This is what we have done below:

```
:- op(200,fx,'$').

/* eval(Expression, Env, ExprValue) */
eval(X,_Env,Res) :- number(X),Res=X.
eval('$'(X),Env,Res) :- lookup(X,Env,Res).
eval('+'(X,Y),Env,Res) :- eval(X,Env,RX), eval(Y,Env,RY), Res is RX+RY.

eint(X:=V,In,Out) :- eval(V,In,Res),store(In,X,Res,Out).
eint('def'(X),In,Out) :- def(In,X,Out).
eint(X;Y,In,Out) :- eint(X,In,I2), eint(Y,I2,Out).

| ?- eint( (def x; x:= 5; def z; x:= $x+1; z:= $x+($x+2)) , [],R).
R = [z/14,x/6] ?
```

**Exercise 7.1.5** Extend the interpreter for other operators, like multiplication `*` and exponentiation `**`.

**Exercise 7.1.6** Rewrite the interpreter so that one does not need to use the “\$”. Make sure that your interpreter only generates a single (and correct) solution.

### 7.1.1 Summary

The general scheme for interpreting statements is:

```
int( Statement[X1,...XN], I1, OutEnv) :-
    int(X1,I1,I2), or eval(X1,I1,R1),I2=I1,
    ...,
    int(Xn,In,OutEnv).
```

For evaluating expressions it is:

```
eval(Expr[X1,...XN], Env, Res) :-
    eval(X1,Env,R1), ...,
    eval(Xn,Env,Rn),
    compute(R1,...,Rn,Res).
```

## 7.2 An imperative language with conditionals

We now want to extend our interpreter to handle conditionals such as the if-then-else. For this, we also need boolean expressions. We basically have two choices to handle and implement them:

- treat a boolean expression `1=x` like an ordinary expression returning a boolean value (or an integer value interpreted as a boolean). In this case, we simply use `eval` to evaluate a boolean expression.
- introduce a separate syntactic class of *boolean expressions*. This means that we will write a separate predicate to evaluate them. Here there are again two sub choices:
  - Write a predicate `check_be(BE,Env)` which succeeds if the boolean expression succeeds in the environment `Env`.
  - Write a predicate `eval_be(BE,Env,BoolRes)` which returns a boolean value (e.g., `true` or `false`). I.e., the predicate will always succeed but return either `true` or `false`.

First, though, we need to examine how to write an if-then-else in Prolog itself. Below we present 4 ways to do this.

### Program 7.2.1

```
ift1(Test,Then,Else) :- Test,!,Then.
ift1(Test,Then,Else) :- Else.

ift2(Test,Then,Else) :- Test, Then.
ift2(Test,Then,Else) :- Not_Test, Else.
% see how to implement negation in earlier chapter

ift3(Test,Then,Else) :- (Test -> Then ; Else).

ift4(Test,Then,Else) :- if(Test,Then,Else). % SICStus Prolog
```

The first version is not declarative. A program is *declarative* if its meaning (i.e., its answers) is independent of the actual execution strategy (left-to-right). I.e., the program can be seen as a logical theory declaring the problem to be solved. Sometimes such a program is also called a *pure logic program*.

Sometimes we can adopt a more precise definition. We say that a predicate  $p$  of arity  $n$  is declarative iff for all terms  $a_1, \dots, a_n$ :

- $\forall \theta. p(a_1, \dots, a_n), \theta \equiv \theta, p(a_1, \dots, a_n)$  (binding-insensitive)
- $p(a_1, \dots, a_n), fail \equiv fail, p(a_1, \dots, a_n)$  (side-effect free)

Here  $G \equiv H$  means that  $G$  and  $H$  have the same meaning. Usually, this means the same sets of computed answers using Prolog's left-to-right execution strategy, and same observable behaviour.

Version 4 is declarative; Version 2 is declarative if we use a declarative encoding of the `not` (see earlier). Versions 1 and 3 are not declarative. The use of the `->` construct is, however, better in the sense that the cut is local and more predictable than the full blown cut. However, if the Test contains no free variables (or only existential “output” variables) then it will behave like versions 2 and 4, while being more efficient. Below we use version 3. We add anew clause to the main interpreter to handle the conditional. Then we introduce a new predicate `eval_be` to evaluate boolean expressions.

### Program 7.2.2

```
int(if(BE,S1,S2),In,Out) :-
    eval_be(BE,In,Res),
    (Res=true -> int(S1,In,Out) ; int(S2,In,Out)).
eval_be('='(X,Y),Env,Res) :- eval(X,Env,RX), eval(Y,Env,RY),
    (RX=RY -> Res=true ; Res=false).
eval_be('<'(X,Y),Env,Res) :- eval(X,Env,RX), eval(Y,Env,RY),
    (RX<RY -> Res=true ; Res=false).
```

If we use a predicate `check_be` rather than an evaluation predicate, the code gets even simpler:

### Program 7.2.3

```
int(if(BE,S1,S2),In,Out) :-
    (check_be(BE,In) -> int(S1,In,Out) ; int(S2,In,Out)).
check_be('='(X,Y),Env) :- eval(X,Env,RX), eval(Y,Env,RY),RX=RY.
check_be('<'(X,Y),Env) :- eval(X,Env,RX), eval(Y,Env,RY),RX<RY.
```

## 7.3 An imperative language with loops

We first add a simple printing mechanism to our language and interpreter, to help us trace the execution of our imperative programs.

In case we wish to print text, we first add string objects. In Prolog strings are often represented as lists of ASCII codes. As we have seen earlier, Prolog also provides a simple syntax to generate such lists, using double quotation marks. E.g., the call `X = "abc"` unifies `X` with the list `[97,98,99]`. One can also use the built-in `name/2` to convert ASCII lists to Prolog atoms and back.

We can thus add a string object to our expression evaluator, which converts an ASCII list to an atom:

```
eval([HS|String],_,Res) :- name(Res,[HS|String]).
```

We can now add a `println` command to our interpreter:

```
int(println(S),E,E) :- eval(S,E,ES),print(ES),nl.
```

Note that this interpreter directly executes the `println` command and performs a side-effect.

**Exercise 7.3.1** Rewrite the interpreter so that it does not perform side-effects for `println`, by storing the output in the environment of the interpreter.

Let us now add while loops to our interpreter.

```
int(while(BE,S),In,Out) :-
    eval_be(BE,In,Res),
    ((Res=true -> int(';')(S,while(BE,S)),In,Out)) ; In=Out).
```

**Exercise 7.3.2** Rewrite the clause for the while loop so that it uses `check_be`.

**Exercise 7.3.3** Rewrite the interpreter so that it uses a more efficient environment representation, e.g., using the `assoc` library of SWI and SICStus or the AVL library of SICStus Prolog.

An alternate solution is as follows. It has the advantage of reusing existing functionality of the interpreter, and will be easier to maintain. However, it is slightly less efficient than the version above:

```
int(skip,E,E).
int(while(BE,S),In,Out) :- int(if(BE,','(S,while(BE,S)),skip),In,Out).
```

Note the similarity with the treatment of `*(X)` in the regular expression interpreter in Section 5.2.

## 7.4 What we have learned

- when writing an interpreter for an imperative language we need an environment to store values for variables
- statements with side-effects will modify this environment, having an input environment and an output environment; this environment is thus threaded through the interpreter calls and we can use DCG notation to do this for us
- pure parts (expressions without side-effects) only require an input environment
- we can write the interpreter in a compositional style
- implementing while loops can be done very much like the closure operator of regular expressions, by recursion

## 7.5 Simple Three-Address Code

Let us take some sample three-address code, in the format of Chapters 6.2, 9 and 12 of the new Dragon book [1]. Three-address code is an intermediate representation used by compilers, where there is at most one operator on the right-hand side of an instruction. It is relatively straightforward to produce three-address code from an abstract syntax tree, and the three-address code itself is a good basis for code generation, analysis and optimisation.

```
(1) i = 0
(2) x = 2
(3) if i>1 goto 7
(4) x = x*x
(5) i = i + 1
(6) goto 3
(7) res = x
```

Basically, we have:

- assignments with three operands x,y,z:  $x = y \text{ OP } z$
- copy assignments with two operands x,y:  $x = y$
- unconditional jumps to a label L: `goto L`
- conditional jumps with two operands x,y and a label L: `if x OP y goto L`

The difference to the previous interpreter is that instructions now have labels, and that we have the possibility to jump straight to the labels.

We first have to decide how to represent the Three-address-code programs in Prolog, and we have the following two options.

- Clausal Representation: Like in Section 5.1 use clauses to represent the object program.
- Term Representation (Reified representation): Like in Sections 5.2, 6, 7.1-7.3, use a Prolog term to represent the object program to be interpreted.

In principle both techniques can be used. The clausal representation has the advantage that we can use Prolog queries to search and traverse the object program. However, modifying the object program can be expensive. The term representation has the advantage that we can easily generate new terms at runtime (see our second implementation of the while loop in Section 7.3), that it is efficient to obtain direct sub-terms of expressions by simple unification, without having to search the Prolog clause database.

While for the imperative language without labels in Sections 7.1 - 7.3 it was more elegant to use a term representation, the situation here is different as we need to be able to jump to arbitrary labels in the object program.

A possible encoding of the above Three-Address program using Prolog facts `inst/5` is as follows:

```

inst(1,assign,i,0,_).
inst(2,assign,x,2,_).
inst(3,if(>),i,1,7).
inst(4,assign(*),x,x,x).
inst(5,assign(+),i,i,1).
inst(6,goto,_,_,3).
inst(7,assign,res,x,_).
inst(8,halt,_,_,_).

```

The general format for an individual instruction fact is:

`inst(Label, OPCODE, Op1, Op2, Op3)`. Operands used on the right-hand side can be either names of local variables, or immediate integer values. Operands used on the left-hand side must be variable names.

What we now want to achieve is an interpreter working as follows:

```

| ?- run(Op).
Op = [i/2,x/16,res/16] ? ;
no

```

In contrast to the previous interpreter we now need to keep track of a program counter in addition to the current environment.

We reuse the `lookup` predicate from the previous sections, but extend `store` so that in the case it reaches the empty environment it adds a new entry into the environment rather than printing an error. This accounts for the fact that in our three-address code variables do not have to be defined before assigning to them.

The following is then the complete code for an interpreter for three-address code:

```

run(Out) :- tint(1,[],Out).

tint(PC,In,Out) :-
    inst(PC,Opc,A1,A2,A3),
    NextPC is PC+1,
    ex_opcode(Opc,A1,A2,A3,NextPC,In,Out).

ex_opcode(halt,_,_,_,_,Env,Env).
ex_opcode(goto,_,_,Label,_,In,Out) :-
    tint(Label,In,Out).
ex_opcode(assign,Var,RHS,_,NextPC,In,Out) :-
    load(RHS,In,RHSVAL),
    store(In,Var,RHSVAL,Out2),
    tint(NextPC,Out2,Out).
ex_opcode(if(OP),RHS1,RHS2,Label,NextPC,In,Out) :-
    load(RHS1,In,RHSVAL1),
    load(RHS2,In,RHSVAL2),
    (test_op(OP,RHSVAL1,RHSVAL2) -> tint(Label,In,Out) ; tint(NextPC,In,Out)).

```

```
ex_opcode(assign(OP),Var,RHS1,RHS2,NextPC,In,Out) :-  
    load(RHS1,In,RHSVAL1),  
    load(RHS2,In,RHSVAL2),  
    ex_op(OP,RHSVAL1,RHSVAL2,Res),  
    store(In,Var,Res,Out2),  
    tint(NextPC,Out2,Out).
```

```
ex_op(*,A1,A2,R) :- R is A1 * A2.  
ex_op(+,A1,A2,R) :- R is A1 + A2.  
ex_op(-,A1,A2,R) :- R is A1 - A2.
```

```
test_op(<,A1,A2) :- A1 < A2.  
test_op(>,A1,A2) :- A1 > A2.  
test_op(<=,A1,A2) :- A1 =< A2.  
test_op(>=,A1,A2) :- A1 >= A2.
```

```
load(Key,Env,Val) :-  
    (number(Key) -> Val=Key /* we have an immediate integer value */  
    ; lookup(Key,Env,Val) /* it is the name of a local variable */).
```



## Chapter 8

# A Small Case Study: Interpreting a subset of Java Bytecode

### 8.1 Input Language

In this section we show how to write an interpreter for a subset of Java bytecode. First, let us look at the following piece of Java code:

```
public class Power {
public static void main(String args[])
{
    int base = 2;
    int exp = 5;
    int i = exp;
    int res = 1;
    while (i>0) {
        i--;
        res = res*base;
    }
    System.out.println(res);
}
}
```

For execution, this piece of code is translated into Java bytecode. By compiling this program with `javac` and then using `javap -c Power`, we obtain the bytecode in human readable form:

```
$ javap -c Power
Compiled from "Power.java"
public class Power extends java.lang.Object{
public Power();
```

```

Code:
  0: aload_0
  1: invokespecial #1; //Method java/lang/Object.<init>:()V
  4: return

public static void main(java.lang.String[]);
Code:
  0: iconst_2
  1: istore_1
  2: iconst_5
  3: istore_2
  4: iload_2
  5: istore_3
  6: iconst_1
  7: istore 4
  9: iload_3
 10: ifle 25
 13: iinc 3, -1
 16: iload 4
 18: iload_1
 19: imul
 20: istore 4
 22: goto 9
 25: getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
 28: iload 4
 30: invokevirtual #3; //Method java/io/PrintStream.println:(I)V
 33: return
}

```

Every instruction consists of an opcode (one byte), followed by its arguments. Some opcodes take no arguments, such as `iconst_2`. The opcodes `istore`, `goto` and `iload` take one argument (one byte each). The opcode `iinc` takes two arguments.

We now want to write a parser and interpreter for such Bytecode. For simplicity we will for the moment only treat the bytecode of a single static method, and also not treat the operations `getstatic` and `invokevirtual`. Hence, we assume that the input for our parser and interpreter will look like this:

```

0: iconst_2
1: istore_1
2: iconst_5
3: istore_2
4: iload_2
5: istore_3
6: iconst_1
7: istore 4
9: iload_3
10: ifle 25
13: iinc 3, -1
16: iload 4

```

```

18: iload_1
19: imul
20: istore 4
22: goto 9
25: return

```

**Exercise 8.1.1** Try to find ways to improve the above bytecode.

Answer: If we swap register 4 with register 2 we will get shorter byte code (there is no specialized `iload_4` instructions) and maybe faster execution. A liveness analysis, could even merge registers 2 and 4.

For the remainder of this chapter it is important to know that, for each method, the Java bytecode operates on:

- a set of local variables, which are numbered (from 0..255)
- a local operator stack. Note that this stack must have the important property that for each program point, the stack layout is independent of the way this program point was reached. We will return to this issue later, as it is important for compilation.

The memory layout of our simplified Java bytecode is illustrated in Figure 8.1. The parameters of each method are passed in variables 0,... In particular, for dynamic methods local variable 0 holds a reference to the `this` object (but we ignore this here and we only deal with static methods).

In the rest of this chapter, we only deal with integer values and with the following eleven opcodes, where *Addr* refers to an address in the bytecode, *Var* to the number of a local variable and *Cst* to an immediate constant:

- **nop**: an operation having no effect (apart from increasing the program counter),
- **return**: terminate the execution of the current method,
- **goto Addr**: jump to a specific address *Addr* in the bytecode,
- **istore Var**: pop the top of the operator stack and write it into local variable numbered *Var*,
- **iload Var**: push the contents of variable *Var* on top of the stack,
- **iconst Cst**: push the immediate constant *Cst* on top of the stack,
- **pop**: pop the top of the stack and discard it,
- **ifle Addr**: pop the top of the operator stack and if it is less or equal to 0 jump to the address *Addr*,
- **iinc Var, Cst**: add the immediate constant *Cst* to the contents of the local variable *Var*,
- **imul**: pop the two topmost values of the operator stack, multiply them and push the result onto the stack,
- **iadd**: pop the two topmost values of the operator stack, add them and push the result onto the stack.

## 8.2 The Parser

First we show the parser, developed using DCGs:

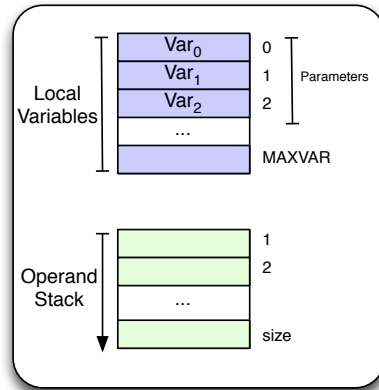


Figure 8.1: Memory layout for simple Java bytecode

```

:- module(javabc_parser, [parse_byc_file/1, instr/3, gen_pl/0]).

parse_byc_file(File) :-
    see(File), read_file(Txt), seen,
    reset, program(Txt, []), !,
parse_byc_file(_) :- print('### Parsing Failed !'), nl.

read_file(Txt) :-
    get_code(CharCode),
    (CharCode < 0 -> Txt = []
    ; Txt = [CharCode|T], read_file(T)
    ).

reset :- retractall(instr(_,_,_)), reset_line_count.

:- dynamic instr/3.

program --> labelled_instr(Nr, Inst, Size), !, {assert(instr(Nr, Inst, Size))},
    program.
program --> ws, !, program.
program([_|_], _) :- !, print('### Could not find instruction label '), print_line_count, nl.
program --> "".

labelled_instr(Nr, Inst, Size) --> label(Nr), !, instr_after_label(Inst, Size).

instr_after_label(Inst, Size) --> instr(Inst, Size), !.
instr_after_label(_, _) -->
    {print('### Could not find valid instruction ')}, {print_line_count}, {nl}, {fail}.

```

```

% whitespace
ws --> [13],!, {increment_line_count}, optws.
ws --> [10],!, {increment_line_count}, optws.
ws --> [9],!, optws.
ws --> " ",!, optws.

% Optional whitespace
optws --> [13],!, {increment_line_count}, optws.
optws --> [10],!, {increment_line_count}, optws.
optws --> [9],!, optws.
optws --> " ",!, optws.
optws --> "".

label(Nr) --> optws,labelnr(Nr),":",optws.

natural(Prev,Nr) --> digit(Dig),!, {P2 is Prev*10+Dig}, natural(P2,Nr).
natural(Prev,Nr) --> {Prev=Nr}.

/* instr(PrologOpCode,ByteSize) */
instr(iconst(Int),1) --> "iconst_", digitft(0,5,Int),!.
instr(iconst(-1),1) --> "iconst_m1",!.
instr(iloop(Reg),1) --> "iloop_", digitft(0,3,Reg),!.
instr(iloop(Reg),2) --> "iloop", ws, registernr(Reg),!.
instr(istore(Reg),1) --> "istore_", digitft(0,3,Reg),!.
instr(istore(Reg),2) --> "istore", ws, registernr(Reg),!.
instr(if1(<=,0,Label),3) --> "if1e", ws, labelnr(Label),!.
instr(goto(Label),3) --> "goto", ws, labelnr(Label),!.
instr(iop(*),1) --> "imul",!.
instr(iop(+),1) --> "iadd",!.
instr(iop(-),1) --> "isub",!.
instr(return,1) --> "return",!.
instr(ireturn,1) --> "ireturn",!.
instr(nop,1) --> "nop",!.
instr(pop,1) --> "pop",!.
instr(iinc(Reg,Inc),3) --> "iinc", ws, registernr(Reg),optws,",",optws, integer(Inc),!.

instr(println(Reg),2) -->
    "println", ws, registernr(Reg),!. % artificial opcode for debugging

integer(Nr) --> "-",!,natural(0,NN), {Nr is 0-NN}.
integer(Nr) --> natural(0,Nr).

labelnr(Nr) --> natural(0,Nr).

registernr(Nr) --> natural(0,Nr).

digitft(From,To,Dig) --> digit(Dig), {Dig >= From, Dig <= To}.

digit(0) --> "0".

```

```

...
digit(9) --> "9".

/* line counting */
:- dynamic cur_line_number/1.
cur_line_number(1).
print_line_count :-
    cur_line_number(N), print('at line '), print(N), print(' ').
increment_line_count :-
    retract(cur_line_number(N)), N1 is N +1,
    assert(cur_line_number(N1)).
reset_line_count :-
    retract(cur_line_number(_N)), assert(cur_line_number(1)).

```

Observations:

- As we do not use a lexing phase, the whitespace has to be treated inside the DCG using the `ws` non-terminal.<sup>1</sup>
- In the abstract syntax tree we group several different opcodes together. This is to allow for a simpler and more elegant interpreter. For example, we have merged `iconst_1`, `iconst_2`, ... into a single abstract syntax tree functor `iconst/1`. Similarly, `istore` has been merged with `istore_1`, `istore_2`, ... and `iload` has been merged with `iload_1`, `iload_2`, .... Finally, `imul`, `iadd`, `isub` have all been grouped together.
- In `digitft` we use the first two arguments to parameterise the non-terminal. This is something that cannot be done in a regular CFG.
- We could have operated directly on the class file. Actually, there are Prolog packages that can do this ... [INSERT REF]

### 8.3 The Interpreter

We can now develop an interpreter for Java bytecode. It is inspired by the interpreter for three-address code from Section 7.5.

The following is the Prolog code of our interpreter. For reasons that will become clear later, we have clearly separated out the execution of operations (such as multiplication) and have written the conditional purely logically (without using the Prolog cut or Prolog negation).<sup>2</sup>

```

execute(Output) :- execute([],Output).
execute(Parameters,Output) :- % Parameters should be [0/Para1,...]
    init_env(Parameters,Start),

```

<sup>1</sup>As an exercise, you may rewrite the parser so that it does use a separate lexing phase.

<sup>2</sup>This means that the interpreter as such is not terribly efficient and also consumes more memory as strictly necessary.

```

interpreter_loop(0,Start,Output).

interpreter_loop(PC,In,Out) :- print('> '),print(PC), print(' '), print(In),
    instr(PC,Opcode,Size), !,
    NextPC is PC+Size,
    print(' --> '),print(Opcode),nl,
    ex_opcode(Opcode,NextPC,In,Out).
interpreter_loop(PC,_In,_Out) :-
    print('*** Unknown instruction label: '), print(PC),nl,fail.

ex_opcode(return,_,Env,Env).
ex_opcode(goto(Label),_,In,Out) :-
    interpreter_loop(Label,In,Out).
ex_opcode(istore(Var),NextPC,In,Out) :-
    pop(In,Top,In2),
    store(In2,Var,Top,Out2),
    interpreter_loop(NextPC,Out2,Out).
ex_opcode(ildload(Var),NextPC,In,Out) :-
    load(Var,In,Val),
    push(In,Val,Out2),
    interpreter_loop(NextPC,Out2,Out).
ex_opcode(iconst(Const),NextPC,In,Out) :-
    push(In,Const,Out2),
    interpreter_loop(NextPC,Out2,Out).
ex_opcode(if1(OP,Cst,Label),NextPC,In,Out) :-
    pop(In,RHSVAL1,In2),
    if_then_else(OP,RHSVAL1,Cst,Label,NextPC,In2,Out).
ex_opcode(iop(OP),NextPC,In,Out) :-
    pop(In,RHSVAL1,In1),
    pop(In1,RHSVAL2,In2),
    ex_op(OP,RHSVAL1,RHSVAL2,Res),
    push(In2,Res,Out2),
    interpreter_loop(NextPC,Out2,Out).
ex_opcode(iinc(Var,Offset),NextPC,In,Out) :-
    load(Var,In,Val),
    ex_op(+,Val,Offset,Res),
    store(In,Var,Res,Out2),
    interpreter_loop(NextPC,Out2,Out).

if_then_else(OP,Arg1,Arg2,TrueLabel,_FalseLabel,In,Out) :-
    test_op(OP,Arg1,Arg2),
    interpreter_loop(TrueLabel,In,Out).
if_then_else(OP,Arg1,Arg2,_TrueLabel,FalseLabel,In,Out) :-
    false_op(OP,Arg1,Arg2),
    interpreter_loop(FalseLabel,In,Out).

ex_op(*,A1,A2,R) :- R is A1 * A2.
ex_op(+,A1,A2,R) :- R is A1 + A2.
ex_op(-,A1,A2,R) :- R is A1 - A2.

```

```

test_op(<=,A1,A2) :- A1 =< A2.

false_op(<=,A1,A2) :- A1 > A2.

/* ----- */
/* Generic Environment Related Predicates */
/* ----- */

init_env(Parameters,env([],Parameters)). % Parameters should be [0/Para1,...]

pop(env([X|S],Vars),Top,R) :- !, Top=X,R=env(S,Vars).
pop(E,_,_) :- print('*** Could not pop from stack: '),print(E),nl,fail.

push(env(S,Vars),X,env([X|S],Vars)).

store(env(Stack,Vars),Key,Value,env(Stack,NVars)) :-
    update(Vars,Key,Value,NVars).

load(Key,env(_S,Env),Val) :- lookup(Key,Env,Val).

/* update(OldEnv, VariableName, NewValue, NewEnv) */
update([],Key,Value,[Key/Value]).
update([Key/_Value2|T],Key,Value,[Key/Value|T]).
update([Key2/Value2|T],Key,Value,Res) :- Key2 \= Key,
    (Key @< Key2 -> Res = [Key/Value, Key2/Value2|T]
     ; (Res = [Key2/Value2|BT], update(T,Key,Value,BT))
    ).

/* lookup(VariableName, Env, CurrentValue) */
lookup(Key,[],_) :- print('*** could not find variable: '),print(Key),nl,fail.
lookup(Key,[Key/Value|_T],Value).
lookup(Key,[Key2/_Value2|T],Value) :-
    Key @> Key2,lookup(Key,T,Value).

```

**Exercise 8.3.1** Improve the efficiency of the interpreter by improving the update and lookup operations and by removing unnecessary choice points (e.g., in `if_then_else`).

## Chapter 9

# From a Concrete Interpreter to an Abstract Interpreter

A mathematical view of a concrete interpreter.

Let us look at the interpreter from the previous chapter. It proceeds in small steps, as follows:

- It takes a state of a program and computes the following state
- every state consists of a program counter, as well as an environment binding variable names to concrete values (in this case integers). The environment also contains a stack of concrete values.
- In order to compute the following state the interpreter applies some concrete operations such as addition, multiplication, equality checks.
- The overall interpreter proceeds computing these small steps until the program terminates. For every state there is a unique successor state.

We have a set of possible program states  $St$ , made up of values from a concreted domain  $\mathcal{C}$ , a set of possible initial states  $S_0 \subseteq St$ , we have for a program  $P$  a semantic function  $succ_P : St \mapsto St$ , which given a state computes the immediate successor state. Let us say we are interested in the set  $R$  of program states reachable from  $S_0$ , e.g., to validate or infer a certain property about our program  $P$ . We can formally define  $R$  as follows:

- $R = F_P \uparrow^\infty (\emptyset)$ , where
- $F_P(S) = S_0 \cup succ_P^*(S)$  and where
- $f^*$  is the lifted version of a function  $f$ , defined by :  $f^*(S) = \{f(s) \mid s \in S\}$ .

- $f \uparrow^0 (S) = S$  and  $f \uparrow^{n+1} (S) = f(f \uparrow^n (S))$ .

Observe that  $F_P$  is monotonic:  $S \subseteq S' \Rightarrow F_P(S) \subseteq F_P(S')$ . Hence,  $F_P$  has by the Knaster& Tarski Fixpoint theorem a least fixpoint  $lfp(F_P)$ . Observe that  $F_P$  is also finitary (and hence continuous), in the sense that for every infinite sequence  $I_0 \subseteq I_1 \subseteq \dots$  we have  $F_P(\bigcup_{n=0}^{\infty} I_n) \subseteq \bigcup_{n=0}^{\infty} F_P(I_n)$ . Hence,  $lfp(F_P) = R$ .

The problem is that the above approach may of course fail to stabilise in finite time (i.e., for no number  $n$  do we have that  $F_P \uparrow^n (\emptyset) = F_P \uparrow^{n+1} (\emptyset) = F_P \uparrow^{\infty} (\emptyset)$ ). Furthermore, even if it does stabilise can be very expensive (basically running the program on all possible inputs), and depending on the concrete domain, the result  $R$  maybe large.

The idea of abstract interpretation is to replace the set of concrete values by a set of abstract values, in order to ensure that the above process always terminates and provides finite representations of all possible program behaviours. In particular, the state of an abstract interpreter would contain abstract values rather than concrete values. Also, instead of applying concrete operations, the abstract interpreter applies abstract counterparts of these operations.

Let us take an example: the interpreter from the previous chapter operated on the concrete set  $\mathcal{C}$  of integer values. A possible abstract set of values is  $\mathcal{A} = \{\mathbf{0}, Z^+, Z^-, \top\}$ .

To give these abstract values a meaning, abstract interpretation introduces two functions:

- The concretisation function  $\gamma : \mathcal{A} \mapsto 2^{\mathcal{C}}$
- The abstraction function  $\alpha : 2^{\mathcal{C}} \mapsto \mathcal{A}$

In our case we define  $\gamma(\mathbf{0}) = \{0\}$ ,  $\gamma(Z^-) = MININT.. - 1$ ,  $\gamma(Z^+) = 1..MAXINT$ , and  $\gamma(\top) = \mathcal{C}$ . The function  $\gamma$  also induces a partial order on  $\mathcal{A}$ :

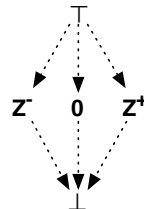
$$a_1 \sqsubseteq a_2 \text{ iff } \gamma(a_1) \subseteq \gamma(a_2)$$

We also write

$$a_1 \sqsubset a_2 \text{ iff } a_1 \sqsubseteq a_2 \wedge a_1 \not\sqsubseteq a_2$$

Hence, in our case,  $\mathbf{0} \sqsubset \top$ ,  $Z^+ \sqsubset \top$ ,  $Z^- \sqsubset \top$ .

Quite often, one also works with a bottom element  $\perp$  such that  $\gamma(\perp) = \emptyset$ . We thus have,  $\perp \sqsubset \mathbf{0}$ ,  $\perp \sqsubset Z^+$ ,  $\perp \sqsubset Z^-$  and  $\perp \sqsubset \top$ , illustrated graphically as follows:



For correctness it is required that for any  $S \subseteq \mathcal{C}$  we have  $\gamma(\alpha(S)) \supseteq S$ . Quite often one defines  $\alpha(S)$  from  $\gamma$  by requiring that  $\alpha(S)$  is the most precise element of  $\mathcal{A}$  such that  $\gamma(\alpha(S)) \supseteq S$ . In our case,  $\alpha(\{1, 3, 5, 7\})$  could be either  $Z^+$  or  $\top$ , as  $\gamma(Z^+) \supseteq \{1, 3, 5, 7\}$  and  $\gamma(\top) \supseteq \{1, 3, 5, 7\}$ . However, the most precise abstraction is  $Z^+$ , as  $Z^+ \sqsubset \top$ .

In order to merge two abstract values, we introduce the following operation  $\sqcup$  (also called least upper bound, lub), which is the counterpart of the set union at the concrete level. First,  $b$  is called an upper bound of  $a_1$  and  $a_2$  iff  $b \sqsupseteq a_1$  and  $b \sqsupseteq a_2$ . Then  $a_1 \sqcup a_2$  denotes the least upper bound of  $a_1$  and  $a_2$ , i.e., it is smaller ( $\sqsubseteq$ ) than all upper bounds of  $a_1$  and  $a_2$ . The least upper bound is unique if it exists. From now on, we will suppose it exists (which is the case if our abstract domain is a semi-lattice). By  $\sqcup_S$  we denote the least upper bound of a set of abstract elements.

Let  $f$  be an  $n$ -ary function on the concrete domain, i.e.,  $f : \mathcal{C} \times \dots \times \mathcal{C} \mapsto \mathcal{C}$ . Let  $f_a$  be an  $n$ -ary function on the abstract domain, i.e.,  $f_a : \mathcal{A} \times \dots \times \mathcal{A} \mapsto \mathcal{A}$ . We say that  $f_a$  is a *safe approximation* of  $f$  iff  $\forall a_1, \dots, a_n$  we have  $\gamma(f_a(a_1, \dots, a_n)) \supseteq \{f(c_1, \dots, c_n) \mid c_1 \in \gamma(a_1) \wedge \dots \wedge c_n \in \gamma(a_n)\}$ .

[In our case this concept can be applied both to the small step function *succ*, to the semantic function  $F_P$ , or to more elementary operations such as multiplication employed by *succ*]

Let  $F_P^A$  be a safe approximation of  $F_P$ . Abstract interpretation consists in computing  $R^A = F_P^A \uparrow^\infty (\perp)$ , where  $\perp$  is such that  $\gamma(\perp) = \emptyset$ . Under certain conditions [ELABORATE] we have that  $lfp(F_P) \subseteq \gamma(lfp(F_P^A))$ . Also, for well chosen abstract domains, the construction  $F_P^A \uparrow^\infty (\perp)$  will stabilise after some finite number of steps.

**Exercise 9.0.2** Write abstract versions of addition, subtraction, multiplication, division, modulo for our abstract domain. Verify that you have developed a safe approximation.

**Exercise 9.0.3** Use the abstract domain to approximate the possible values of  $a$  and  $b$  after executing the following piece of pseudo-code. Suppose that  $a$  and  $b$  are strictly positive upon entry.

```
a := a+b;
b := b+1;
a := a*b;
```

What about the following piece of code:

```
a := a+b;
b := b-1;
a := a*b;
```

How could one improve the situation, i.e., obtain a more accurate result?

**Exercise 9.0.4** Use the abstract domain to approximate the possible values of  $a$  and  $b$  during and after executing the following piece of pseudo-code. Suppose that  $a$  and  $b$  are strictly positive upon entry.

```

while b>=0 do
  a := a+b;
  b := b-1
od
a := a*b

```

What has changed wrt the previous exercise?

## 9.1 A schema to compute $lfp(F_P^A)$

We need to develop an abstract version of the semantic function  $F_P^A$ . In our case, having an interpreter which computes  $succ$ , we can start as follows:  $F_P^A(S) = succ_P^A(S) \sqcup \alpha(S_0)$  and where  $succ^A$  is a safe approximation of  $succ$ .

**Exercise 9.1.1** (We could add  $\sqcup S$  to  $F_P^A(S)$ . Does it change anything ?)

In order to define  $succ^A$  we need to look at the concrete state (in our case  $PC * seq(INT) * (Var \mapsto INT)$ ) and decide which parts of it will be abstracted, and how. A typical approach would be to not abstract the program counter (after all there are only finitely many program points), and not to abstract the “structure” of the stack/environment but only abstract the concrete data values. We would also provide a special abstract value for program points which have not been reached (bottom). I.e, an abstract state would be a set of elements from either  $PC * \perp$  or  $PC * (seq(A) * (Var \mapsto A))$ . Hence,  $succ^A$  is basically the interpreter as before, but:

- using elements of the concrete data domain  $\mathcal{A}$  rather than  $\mathcal{C}$
- using abstract operations on  $\mathcal{A}$ , which are safe approximations of the concrete counterparts for  $\mathcal{C}$  in  $succ$

This will guarantee that  $F_P^A$  is a safe approximation of  $F_P$ , and hence that the result of abstract interpretation is sound.

Note that, while  $succ$  is deterministic,  $succ^A$  can be non-deterministic. E.g., *if*  $2 < 3$  *goto* 10 is deterministic, *if pos < pos goto* 10 needs to explore both alternatives.

In order to compute  $lfp(F_P^A)$  we can simply proceed as follows:

- keep a table which for every program point tracks the abstract information about the possible environments at that program point. This can be most easily done with a dynamic fact database. The fact database would be initialised with the empty relation, signifying that initially no program point is reachable (this corresponds to setting the abstract environment to  $\perp$  for all program points).

- when reaching a program point:
  - if this program point was not reached before: simply store the current abstract environment in the table and proceed normally
  - if this program point was reached before: we need to merge the previous and new abstract environment. This is the LUB (Least Upper Bound) operation. We then need to check if the LUB is different from the currently store abstract environment. If it is: we need to update the table and proceed with the abstract interpretation. If not, we can simply return.

Note: because of the non-determinism, we need an outer loop to drive the interpreter until no more updates to the table occur. After that the table contains the representation of  $lfp(F_P^A)$ .

First, implementing the least upper bound for abstract values:

```
/* Abstract Domain:
   0, pos, neg, top
   lub(X,X,R) :- !,R=X.
   lub(X,Y,top) :- X\=Y.
```

Now, implementing abstract operations for arithmetic operators:

```
aex_op(*,0,_,0).
aex_op(*,pos,X,X).
aex_op(*,neg,0,0).
aex_op(*,neg,pos,neg).
aex_op(*,neg,neg,pos).
aex_op(*,top,0,0).
aex_op(*,top,X,top) :- X\=0.
```

```
aex_op(+,0,X,X).
aex_op(+,pos,0,pos).
aex_op(+,pos,pos,pos).
aex_op(+,pos,neg,top).
aex_op(+,pos,top,top).
aex_op(+,neg,0,neg).
aex_op(+,neg,pos,top).
aex_op(+,neg,neg,neg).
aex_op(+,neg,top,top).
aex_op(+,top,_,top).
% - missing
% TO DO for homework
```

Now, implementing boolean test predicates:

```
% Redefinition of predicate from javabc_interpreter.pl:
test_op(<=,X,X).
```

```

test_op(<=,top,X) :- X \= top.
test_op(<=,neg,X) :- X \= neg.
test_op(<=,0,pos).
test_op(<=,0,top).
test_op(<=,pos,top).

test_op(<,top,_).
test_op(<,neg,_).
test_op(<,0,pos).
test_op(<,0,top).
test_op(<,pos,pos).
test_op(<,pos,top).

test_op(>=,X,Y) :- test_op(<=,Y,X).
test_op(>,X,Y) :- test_op(<,Y,X).

% Redefinition of predicate from javabc_interpreter.pl:
false_op(<,A1,A2) :- test_op(>=,A1,A2).
false_op(>,A1,A2) :- test_op(<=,A1,A2).
false_op(<=,A1,A2) :- test_op(>,A1,A2).
false_op(>=,A1,A2) :- test_op(<,A1,A2).

```

Note: here we become non-deterministic, for some combinations of abstract values both `test_op` and `false_op` succeed. Success means: this test could be true (resp. false) for some concrete values.

Now, for the rest of the code:

```

aint :- print('Starting ABSTRACT Interpretation'),nl,
        execute(_X),fail.
aint :- print('FIXPOINT reached'),nl,
        print('ABSTRACT INFORMATION AT PROGRAM POINTS: '),nl,
        memo(PC,AEnv),
        print(PC), print(' : '), print(AEnv),nl,
        fail.
aint :- nl,flush_output(user).

:- dynamic memo/2, sol/2.

% Redefinition of predicate from javabc_interpreter.pl:
init_env(env([],[])) :- retractall(memo(_,_)).

% Redefinition of predicate from javabc_interpreter.pl:
interpreter_loop(PC,In,_Out) :- print(PC), print(' : '),
    lub_program_point(PC,In,AIn,Change),
    (Change=true ->
        (instr(PC,Opcode,Size),
         NextPC is PC+Size,

```

```

    print(Opcode),print(' '), print(AIn),nl,
    ex_opcode(Opcode,NextPC,AIn,_)
  )
; print(fix(AIn)),nl
).

lub_program_point(PC,env(S,Vars),Res,Change) :-
  (retract(memo(PC,env(SM,VarsM)))
  -> lub_stack(S,SM,SR),
    lub_local_vars(Vars,VarsM,VarsR),
    Res = env(SR,VarsR), assert(memo(PC,Res)),
    (Res=env(SM,VarsM) -> Change=false
      ; print(' <REANALYZE> '),Change = true)
  ; assert(memo(PC,env(S,Vars))),
    Res = env(S,Vars),
    Change=true
  ).

lub_stack([],[],[]).
lub_stack([],[_|_|_],_) :- print('*** Illegal bytecode: Varying stack pattern at PC'),nl,fail.
lub_stack([_|_|_],[],_) :- print('*** Illegal bytecode: Varying stack pattern at PC'),nl,fail.
lub_stack([H1|T1],[H2|T2],[H3|T3]) :-
  lub(H1,H2,H3),
  lub_stack(T1,T2,T3).

lub_local_vars([],X,X).
lub_local_vars([H|T],[],[H|T]).
lub_local_vars([Key/Val1|T1],[Key/Val2|T2],[Key/Val3|T3]) :- !,
  lub(Val1,Val2,Val3),
  lub_local_vars(T1,T2,T3).
lub_local_vars([Key1/Val1|T1],[Key2/Val2|T2],[H|T3]) :- Key1 \= Key2,
  (Key1 @<Key2
  -> (H=Key1/Val1, lub_local_vars(T1,[Key2/Val2|T2],T3))
  ; (H=Key2/Val2, lub_local_vars([Key1/Val1|T1],T2,T3))
  ).

push(env(S,Vars),Value,env([AV|S],Vars)) :-
  abstract_value(Value,AV).

store(env(Stack,Vars),Key,Value,env(Stack,NVars)) :-
  abstract_value(Value,AV),
  update(Vars,Key,AV,NVars).

abstract_value(X,AV) :- number(X),!,
  (X=0 -> AV=0
  ; (X>0 -> AV = pos ; AV = neg)
  ).
abstract_value(X,X).

```

```
% Redefinition of predicate from javabc_interpreter.pl:
ex_op(OP,A1,A2,R) :- abstract_value(A1,AV1),
    abstract_value(A2,AV2),
    aex_op(OP,AV1,AV2,R).
```

**Exercise 9.1.2** Adapt an interpreter from Chapter 7 to perform abstract interpretation.

**Exercise 9.1.3** Adapt the above interpreter to work on a more precise domain, adding the abstract values  $Z_0^+$  and  $Z_0^-$ .

Below is a sample output of our abstract interpreter. For every program point, we obtain information about the operand stack and the local variables, where `top` denotes the abstract value which represents every possible value, `pos` only denotes strictly positive values.

```
Starting ABSTRACT Interpretation
FIXPOINT reached
ABSTRACT INFORMATION AT PROGRAM POINTS:
0 : iconst(2) : env([], [])
1 : istore(1) : env([pos], [])
2 : iconst(5) : env([], [1/pos])
3 : istore(2) : env([pos], [1/pos])
4 : iload(2) : env([], [1/pos,2/pos])
5 : istore(3) : env([pos], [1/pos,2/pos])
6 : iconst(1) : env([], [1/pos,2/pos,3/pos])
7 : istore(4) : env([pos], [1/pos,2/pos,3/pos])
9 : iload(3) : env([], [1/pos,2/pos,3/top,4/pos])
10 : if1(<=,0,25) : env([top], [1/pos,2/pos,3/top,4/pos])
13 : iinc(3,-1) : env([], [1/pos,2/pos,3/top,4/pos])
16 : iload(4) : env([], [1/pos,2/pos,3/top,4/pos])
18 : iload(1) : env([pos], [1/pos,2/pos,3/top,4/pos])
19 : iop(*) : env([pos,pos], [1/pos,2/pos,3/top,4/pos])
20 : istore(4) : env([pos], [1/pos,2/pos,3/top,4/pos])
22 : goto(9) : env([], [1/pos,2/pos,3/top,4/pos])
25 : println(4) : env([], [1/pos,2/pos,3/top,4/pos])
27 : return : env([], [1/pos,2/pos,3/top,4/pos])
```

As can be seen, we have inferred for every program point the layout of the stack (the first part of the `env` term). For example for program point 19, performing the `imul` instruction, we have as stack layout `[pos,pos]`, i.e., there are exactly two values on the stack (which are also guaranteed to be positive). For compilation this means that when generating the compiled code for program point 19 we know exactly how the stack looks like and exactly from which memory location we need to take the two operands for the multiplication.

- when performing abstract interpretation, non-determinism arises naturally: because of the abstraction one can often not decide whether a test will fail or succeed. As such, Prolog is a good language to encode abstract interpreters in, even if we analyse deterministic, imperative languages.

**Exercise 9.1.4** Add support for equality and disequality in `test_op` and `false_op`.

## 9.2 Improving Precision

After a test is taken, we can often make an abstract value more precise. E.g., if we have  $x > 0$  and  $x$  has as abstract value  $\top$ , then the test can succeed at runtime. If it does, we can narrow down  $x$  to be of type  $Z^+$ .

**Exercise 9.2.1** Improve the abstract interpreter to perform this optimisation.

Hint: `test_op` will now take two extra arguments, the new abstract value of the operands. For example, for the `<` operator, we have:

```
test_op(<,top,0,neg,0).
test_op(<,top,neg,neg,neg).
test_op(<,top,pos,top,pos).
test_op(<,top,top,top,top).
test_op(<,neg,X,neg,X).
test_op(<,0,top,0,pos).
test_op(<,0,pos,0,pos).
test_op(<,pos,pos,pos,pos).
test_op(<,pos,top,pos,pos).
```

You will then have to adapt `if_then_else` to store the changes.

## 9.3 Other Domains

Constant Propagation.

Domain: constants, + `nac` (not a constant; equivalent to  $\top$ ).

With respect to the previous abstract interpreter, we only need to update the definitions of

1. the least upper bound `lub/3`.
2. the binary boolean comparators `test_op/3`, and `false_op/3`
3. the arithmetic operators `ex_op/4`.
4. the abstraction of concrete values `abstract_value/2`.

```
lub(X,X,R) :- !,R=X.
lub(_X,_Y,nac).
```

```
% Redefinition of predicate from javabc_interpreter.pl:
test_op(OP,X,Y) :- number(X),number(Y),test_op_concrete(OP,X,Y).
test_op(_OP,X,Y) :- \+ number(X) ; \+ number(Y).
```

```
% Redefinition of predicate from javabc_interpreter.pl:
false_op(OP,X,Y) :- number(X),number(Y),false_op_concrete(OP,X,Y).
false_op(_OP,X,Y) :- \+ number(X) ; \+ number(Y).
```

```
abstract_value(X,X).
```

```
ex_op(OP,X1,X2,R) :- %print(ex_op(OP,X1,X2,R)),nl,  
                    number(X1), number(X2), ex_op_concrete(OP,X1,X2,R).  
ex_op(_,_,_,_nac) :- \+ number(X) ; \+ number(Y).
```

### 9.3.1 Related Work

The CLIP group from the Technical University of Madrid has developed a class file loader Prolog library and analyse Java bytecode using their CiaoPP abstract interpretation engine; see, for example, [4, 3, 2, 35]. Another related work is [30] as well as part of [44]. Also, since Java SE 6 (version 50.0 of the class file format), class files now also contain information about the stack layout (see, e.g., Section 4.8.4 of [17]). Note that Section 4.11 of [17] contains Prolog code as a specification of the type checking verification procedure for class files.