/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

CS4212 -- Compiler Design

T O Y C O M P I L E R F O R ''W H I L E'' L A N G U A G E

W I T H S C O P E D V A R I A B L E S ,

P R O C E D U R E S , A N D I N D E X E D

V A R I A B L E S

This is a toy compiler that provides a code base for Problem Set 4.

It is based on compiler\_procedures.pro, which implements scoped

variables and procedures. It adds an indexing operator, denoted by

'@', which gives access to memory locations adjacent to a variable.

Thus, in a scope with the local declarations:

{ local a,b,c,d,e ; ... }

the expression b@2 will evaluate to the memory location that is 2\*4

bytes away from b, that is, to the value of d. In general, var@x will

access the memory location that is 4\*x bytes away from the address

of var. The value x can be either positive or negative. For instance,

c@-2 will access a, and a@4 will access e. Index expressions can

currently appear only on the right side of the assignment operator,

and in any place where an expression is required. Thus, for instance,

a = e@-2 + e@-1 is equivalent to a = c+d.

The compiler does not check whether the index falls in a memory range

that contains user-defined data, and thus, the @ operator can be used

to access arbitrary memory locations.

For further documentation on the use of the compiler, please consult

the file compiler\_procedures.pro.

Example source program:

global a,b ;

f#(x,y) :: {

local z,w;

z = x@1 ; /\* same as w = y \*/

b = x@1 ; /\* same as a = y \*/

} ;

b = 1 ;

f#(2,3);

a = a@1 /\* a = b \*/

Other usage examples given at the end of the file.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

:- [library(clpfd)]. % We need arithmetic

:- [library(lists)]. % We need 'union'

% operator declarations that allow the above program to be read as a Prolog term

:- op(1099,yf,;).

:- op(960,fx,if).

:- op(959,xfx,then).

:- op(958,xfx,else).

:- op(1050,fx,global).

:- op(1050,fx,local).

:- op(960,fx,for).

:- op(960,fx,while).

:- op(959,xfx,do).

:- op(960,fx,switch).

:- op(959,xfx,of).

:- op(101,xfx,::).

:- op(100,xfx,#).

:- op(950,fx,return).

:- op(200,xfx,@).

% Predicate that creates an array of strings for

% all the variables in the list given as first argument.

% This will be used to create an array of variable names

% for all the global variables, so that the runtime.c

% code can print the final values of the global variables

% at the end of execution.

% 1st arg: list of variable names collected from program

% 2nd arg: allocation of space for each variable

% 3rd arg: array of strings containing names of variables

% 4th arg: array of pointers to each of the strings

allocvars([],[],[],[]).

allocvars([V|VT],[D|DT],[N|NT],[P|PT]) :-

atomic\_list\_concat(['\n',V,':\t\t .long 0'],D),

atomic\_list\_concat(['\n',V,'\_name:\t .asciz "',V,'"'],N),

atomic\_list\_concat(['\n',V,'\_ptr:\t .long ',V,'\_name'],P),

allocvars(VT,DT,NT,PT).

% Generate fresh labels for label placeholders

% 1st arg : list of label placeholders to be instantiated

% 2nd arg : integer suffix not yet used for a label before the call

% 3rd arg : integer suffix not yet used for a label after the call

% -- essentially LabelSuffixOut #= LabelSuffixIn + length(1st\_arg)

generateLabels([],LabelSuffix,LabelSuffix).

generateLabels([H|T],LabelSuffixIn,LabelSuffixOut) :-

atomic\_list\_concat(['L',LabelSuffixIn],H),

LabelSuffixAux #= LabelSuffixIn + 1,

generateLabels(T,LabelSuffixAux,LabelSuffixOut).

% Register allocation is solved via Prolog's clpfd constraint

% solver. For that to work, we encode initially all the registers

% as numbers (in fact, they will be unbound variables for the most

% part of the expression compilation process). The encoding is the

% following: 0 -> %eax, 1 -> %edx, 2 -> %ecx, 3 -> %ebx,

% 4 -> %esi, 5 -> %edi

% A similar encoding exists for 8-bit register numbers

% Once all the register placeholders have been bound to numbers,

% these numbers can be translated into register names. This

% predicate does exactly that. It takes in a list of raw Pentium

% code, and changes every term of the form reg32(N) or reg8(N) into

% its corresponding name.

% 1st arg : code template (list of atoms) with regs represented

% as reg32(N) or reg8(N)

% 2nd arg : same code where each reg32(N) or reg8(N) has been replaced

% with the corresponding register name

replace\_regs([],[]).

replace\_regs([reg32(N)|T],[A|TR]) :-

translate\_regs(N,A),

replace\_regs(T,TR).

replace\_regs([reg8(N)|T],[A|TR]) :-

member((N,A),

[(0,'%al'),(1,'%dl'),(2,'%cl'),(3,'%bl')]),!,

replace\_regs(T,TR).

replace\_regs([H|T],[H|TR]) :- replace\_regs(T,TR).

% Further translations of register numbers into register

% names are required when inspecting the placeholder for

% the result of an expression. This predicate serves that

% purpose.

translate\_regs(0,'%eax') :- !.

translate\_regs(1,'%edx') :- !.

translate\_regs(2,'%ecx') :- !.

translate\_regs(3,'%ebx') :- !.

translate\_regs(4,'%esi') :- !.

translate\_regs(5,'%edi') :- !.

% 'combine\_expr\_code' is the main predicate performing register

% allocation for partial results of expressions. It tries to

% minimize register use, and avoid spilling results into memory.

%

% This predicate is called by the 'ce' (compile expression) predicate,

% after it has generated code for the subexpressions of a binary

% expression. This predicate generates a code template for the

% current operator, allocating registers for partial results,

% yet leaving the registers as underspecified (i.e. unbound) as

% possible. However, these placeholders are constrained with operators

% from the clpfd library to comply with the correctness requirements

% of the generated code. The placeholders are concretized in a global

% round of constraint solution search, performed via a call to 'label'

% in the 'comp\_expr' predicate.

%

% This code is particularly complicated by the peculiarity of the

% idivl instruction on the Pentium. In principle, whenever we have

% an expression of the form (ELeft Op ERight), where Op is some

% binary operator, we compile separately ELeft and ERight, and then

% we bring the resulting codes into this predicates to be combined

% into the code for the entire input expression. We need to consider

% 4 cases for each argument:

% - the result of the expression is a constant

% - the result of the expression is placed in a variable

% - the result of the expression is placed in a register,

% and we have a list of some other registers that needed

% to be used throughout that expression's evaluation (which

% means that we can't rely on those registers holding partial

% results from other expressions, since they will be

% overwritten).

% - the result of the expression is in (%eax,%edx), since the top

% level operator of that expression was a division or a remainder

% operation (which means that for the other 3 items above, the

% top level operation of either ELeft or ERight are not idivl).

% We also have 3 cases for the operator Op:

% - The current operator Op is a binary arithmetic operator:

% +,-,\*,/\,\/ -- differentiate between commutative and

% non-commutative

% - The current Op is division or remainder: /, rem

% - The current Op is a boolean relational operator:

% <,>,=<,>=,==,\=

%

% In principle, that means 3\*4\*4 cases. However, some of these

% combinations of cases are handled together.

%

% The arguments to 'combine\_expr\_code' are as follows:

%

% 1st & 2nd arg : representation of the result holder for the

% (input) left and right subexpressions. May have the

% following format:

% const(N) : the result is the constant N

% id(X) : the result is variable X

% [R1,R2,...] : the result is in register R1

% (which may be an unbound variable)

% and the rest of the list contains

% the registers that MUST be used

% (and therefore will be clobbered)

% throughout the evaluation of

% the subexpression at hand

% 3rd arg : The operator for which code is being generated

% (input) Can be any of the following:

% +,-,\*,/\,\/ : any pair of registers can be used as operands

% /,rem : must use %eax,%edx, and another register

% <,=<,>,>=,==,\= : may have residual code so as to

% allow compilation as regular arithmetic expr,

% and also efficient code when appears as

% if or while condition

% 4th & 5th arg : Code generated for the left and right subexpressions

% (input) (must be consistent with 1st and 2nd arg)

%

% 6th arg : Resulting code for the current expression (return value)

% 7th arg : Residual code, empty for all arithmetic expressions, but

% (outputs) non-empty for boolean expressions.

% The residual code places

% the boolean value in a register. This is not necessary

% if a boolean expr say x < y appears as an 'if' or 'while'

% condition, but it is necessary if we use it in an

% arithmetic expression, in the form (x<y)+1.

% Thus, the decision of whether the residual code should be

% appended to the currently generated code needs to be made

% one level above from the current call.

% 8th arg : The result holder, with the same syntax as args 1 and 2.

% (output) The predicate will decide to either return directly a

% constant or variable, if possible (thus not using any

% of the registers, and allowing them to be used for other

% purposes). Or, it will return a list of registers (encoded

% as numbers, with the encoding given by predicate replace\_regs).

% In that case, the first register in the list holds the

% result of the expression, and the rest of the registers

% will be used (and therefore, their values will be clobbered)

% throughout the evaluation, but once the result has been computed,

% will no longer hold any important value, and can be reused

% for in evaluations of other subexpressions.

combine\_expr\_code(const(N),const(M),Op,[],[],[],[],Result) :- % const opd, all arithmetic ops

% Expressions containing only constants are evaluated

% at compile time and do not generate any code

member(Op,[+,-,\*,/\,\/,/]),!,

% 'Member' is used to classify the operator

ResultExpr =.. [Op,N,M],

ResultVal #= ResultExpr, % For arithmetic expressions, use #= to compute the result

Result = const(ResultVal). % at compile time. Return the result as a constant

combine\_expr\_code(const(N),const(M),Op,[],[],[],[],Result) :- % const opd, relational ops

% Expressions containing only constants are evaluated

% at compile time and do not generate any code

member((Op,Opc),[(<,#<),(>,#>),(=<,#=<),(>=,#>=),(\=,#\=),(==,#=)]),!,

% 'Member' is used to translate between Operator and its

% corresponding constraint operator.

ResultExpr =.. [Opc,N,M],

ResultVal #<==> ResultExpr, % For relational expressions, use #<==> to compute the

Result = const(ResultVal). % result at compile time. Return the result as a constant.

combine\_expr\_code(L,R,Op,[],[],Code,Residue,Result) :- % non-register opd, relational ops

% Subexpressions that do not need to be held in registers

% are expected to have generated empty code

( L = const(Tmp), R = id(Y), atomic\_concat('$',Tmp,X) ;

L = id(X), R = const(Tmp), atomic\_concat('$',Tmp,Y) ;

% Constants must have $ prepended in the assembly code

L = id(X), R = id(Y) ),

% X and Y contain assembly language representations of

% the operands. 'Member' is used to translate between

% Operator and its corresponding OpCode.

member((Op,I),[(<,setl),(=<,setle),(>,setg),(>=,setge),(\=,setne),(==,sete)]),!,

% For boolean expressions, a residue may be needed

% This is possible only with regs eax,edx,ecx,ebx

% The setXX instruction sets an 8-bit reg to 1 or 0

% depending on whether the XX condition is true or false;

% the corresponding 32-bit reg can hold the same result

% if it has been zeroed first.

Result = [Reg], Reg in 0..3,

% We allocate one register to hold the result of this operator,

% and we denote it by Reg. This register will not be explicitly

% specified, but allocated through constraint programming later,

% when the predicate 'label' is called for the entire

% code template, corresponding to the entire expression at hand.

% Thus, currently, Reg is an unbound variable, constrained

% to be between 0 and 3.

% The Result list is the list containing this register

Code = [ '\n\t\t movl ',X,',',reg32(Reg), % Load one operand into a register

'\n\t\t cmpl ',Y,',',reg32(Reg) ],% and compare with the other;

% result will be stored in condition flags

Residue = [ '\n\t\t movl $0,',reg32(Reg), % Zero the register first, and then

'\n\t\t ',I,' ',reg8(Reg) ].% set the lower 8 bits to 0 or 1

combine\_expr\_code(L,R,Op,[],[],Code,[],Result) :- % non-register opds, arithmetic non-div ops

% Subexpressions that do not need to be held in registers

% are expected to have generated empty code

( L = const(Tmp), atomic\_concat('$',Tmp,X), R = id(Y) ;

L = id(X), R = const(Tmp), atomic\_concat('$',Tmp,Y) ;

% Constants must have $ prepended to the name in assembly language

% X and Y are assembly language representations of the partial results

% given in L and R.

L = id(X), R = id(Y) ),

% Handling of arithmetic operators that can be performed

% using any registers. 'Member' is used to translate between

% Operator and its corresponding OpCode.

member((Op,I),[(+,addl),(-,subl),(\*,imull),(/\,andl),(\/,orl)]),!,

Result = [Reg], Reg in 0..5,

% We allocate one register to hold the result of this operator,

% and we denote it by Reg. This register will not be explicitly

% specified, but allocated through constraint programming later,

% when the predicate 'label' is called for the entire

% code template, corresponding to the entire expression at hand.

% Thus, currently, Reg is an unbound variable, constrained

% to be between 0 and 5.

% The Result list is the list containing this register

Code = [ '\n\t\t movl ',X,',',reg32(Reg), % Load one operand into a register

'\n\t\t ',I,' ',Y,',',reg32(Reg) ]. % and perform the operation I with the

% other operand.

combine\_expr\_code(L,R,Op,[],[],Code,[],Result) :- % non-register opds, Op is division or rem

% Subexpressions that do not need to be held in registers

% are expected to have generated empty code

( L = const(Tmp), atomic\_concat('$',Tmp,X), R = id(Y) ;

L = id(X), R = const(Tmp), atomic\_concat('$',Tmp,Y) ;

% Constants must have $ prepended to the name in assembly language

% X and Y are assembly language representations of the partial results

% given in L and R.

L = id(X), R = id(Y) ),

% Handling of division and remainder operators. This is

% complicated because one operand, as well as the result

% must be in registers %edx:%eax. 'Member' is used to translate

% between Operator and its OpCode.

member((Op,I),[(/,idivl),(rem,idivl)]),!,

% 0 = %eax, 1 = %edx

% For division, result will be [%eax,%edx,Reg]

% For reminder, result will be [%edx,%eax,Reg]

(R = const(\_) % if right operand is a constant, the AL representation is Y

-> % Then, prepare the operand for division/remainder by allocating

% some register Reg to hold the value Y, and then generating code

% that loads Y into Reg, so that division can be

% performed between %edx:%eax and Reg. The operand of idivl is not

% allowed to be a constant (damn weird, huh?)

CodeL = [ '\n\t\t movl ',Y,',',reg32(Reg) ], Reg in 0..5, Z = reg32(Reg),

% The result will be either %eax or %edx, depending on whether

% we are translating division or remainder. Variable Z is set

% to hold the operand of idivl, be it a register or a variable.

(Op == / -> Result = [0,1,Reg] ; Result = [1,0,Reg])

; % Otherwise, R must be an identifier. In this case, idivl can

% take its operand directly from memory, therefore CodeL is empty.

% Z is set to represent the AL operand to idivl, be it a

% register, or an identifier.

CodeL = [], (Op == / -> Result = [0,1] ; Result = [1,0]), Z = Y ),

% In this code template, reg32(0) will be later replaced

% with %eax by replace\_regs. Similar for reg32(1) --> %edx

% Here, we have to constrain the registers tightly, since

% idivl can only work with %eax and %edx

% code implementing division, assuming I is the opcode for current operation

% (can only be idivl in this case), and Z is the operand, either a variable or

% a register.

CodeOp = [ '\n\t\t movl ',X,',',reg32(0), % load first opd into %eax

'\n\t\t movl ',reg32(0),',',reg32(1), % copy %eax into %edx

'\n\t\t shrl $31,',reg32(1), % %edx becomes sign extension of %eax

'\n\t\t ',I,' ',Z ], % generate instruction performing Op

all\_distinct(Result), % Set constraint that all allocated registers must be distinct

% The subsequent application of 'label' will comply

append(CodeL,CodeOp,Code).% Finally, lay everything out into generated Code.

combine\_expr\_code(L,R,Op,CL,CR,Code,[],Result) :- % arithmetic Opr, one reg opd, one non-reg opd

% Code generation for the case when one subexpression

% is constant or a variable (i.e. doesn't need registers

% in its evaluation), and the other subexpression does

% need registers. The used registers list remains the same,

% as the result register of the subexpression that needs one

% can be reused to hold the result of the current expression.

( L = [Reg|\_], ( R = const(Tmp),atomic\_concat('$',Tmp,X) ; R = id(X)) ;

(L = const(Tmp), atomic\_concat('$',Tmp,X) ; L = id(X)), R = [Reg|\_] ),

% X and Reg are AL representations of operands

% handle operators that can be performed between any registers

% Commutative operators may invert the order of Left and Right operands.

% Non-commutative operators (-) must have register operands on the left

% -- this is tested at the same time with the translation Op -> OpCode

member((Op,I,L), % moreover, handle only commutative operators

[(+,addl,\_),(-,subl,[\_|\_]),(\*,imull,\_),(/\,andl,\_),(\/,orl,\_)]),!,

( L = [\_|\_] -> Result = L ; Result = R ),

CodeOp = [ '\n\t\t ',I,' ',X,',',reg32(Reg) ], % code to perform operation between

(CL = [] -> CS = CR ; CS = CL), % result reg and const/var operand

append([CS,CodeOp],Code). % order of operands NOT SIGNIFICANT

combine\_expr\_code(L,R,Op,CL,CR,Code,Residue,Result) :- % relational Opr, one reg opd, one non-reg

% One subexpression is const/variable, the other needs

% registers. This rule is for boolean operators.

( L = [Reg|\_], ( R = const(Tmp), atomic\_concat('$',Tmp,X) ; R = id(X)) ;

(L = const(Tmp), atomic\_concat('$',Tmp,X) ; L = id(X)), R = [Reg|\_] ),

% X and Reg are AL representations of L and R, or R and L

member((Op,I),

[(<,setl),(=<,setle),(>,setg),(>=,setge),(==,sete),(\=,setne)]),!,

% check if boolean operator, and set instruction for

% residual code

% The result is the same list of registers as for the

% operand represented in a register

( L = [\_|\_] -> Result = L ; Result = R ),

( CL = [] % Figure out the order of the operands. The one that comes in

% with empty code is the non-reg one

% CS = code subexpression; whatever non-empty code comes from

% arguments must be copied into generated code of the

% current expression.

-> % If the right operand is in register, then CS must reference code from right

% Comparison is between X and Reg holding Result of Right (at&t reverses arg order)

CS = CR, CodeOp = [ '\n\t\t cmpl ',reg32(Reg),',',X ]

; % Otherwise, the left operand must be in register, therefore CS must reference code of left

% Comparison is in oposite order, between Reg and X

CS = CL, CodeOp = [ '\n\t\t cmpl ',X,',',reg32(Reg) ] ),

% Lay out all the code in the generated Code.

append([CS,CodeOp],Code),

Reg in 0..3, % Constrain the registers to be {%eax,%edx,%ecx,%ebx}

% This is because in generating the residual code,

% the corresponding 8-bit reg must be used

% in transferring result of comparison into register, and %esi and %edi

% do not have corresponding 8-bit registers.

Residue = [ '\n\t\t movl $0,',reg32(Reg), % residual code to use in expressions

'\n\t\t ',I,' ',reg8(Reg) ]. % of the form 1 + (a<b).

% NOTE: more aggressive optimization could be used here, since we can find out

% whether the residual code would be needed or not. If residual code not needed,

% registers can be relaxed to the entire set of 6.

combine\_expr\_code(L,R,Op,CL,[],Code,[],Result) :- % Op=div,rem ; Left = reg, Right = non-reg

% For division/remainder, left operand MUST use registers,

% because of the constraints imposed by the idivl instruction.

% This is the case when the right operand is const/variable.

L = [\_|\_], ( R = const(Tmp), atomic\_concat('$',Tmp,X) ; R = id(X)),

member((Op,I), [(/,idivl),(rem,idivl)]),!,

% Left subexpression may have used 1 or more registers, handle each case separately

% A register may need to be allocated for right operand. A legal right operand will

% be computed in Z, and the (possibly empty) code that sets up Z is generated into

% CodeL.

( L = [0|T], select(1,T,Rest),

% If it used multiple registers, we try to make %edx one

% of the other used registers, and then reuse the registers

% in computing the entire expression (i.e. no new register is

% needed)

(R = const(\_) % differentiate between const and id; differentiate between / and rem

-> ( select(Reg,Rest,RRest) ; RRest = Rest),

(Op == / -> Result = [0,1,Reg|RRest] ; Result = [1,0,Reg|RRest]),

CodeL = [ '\n\t\t movl ',X,',',reg32(Reg) ], Reg in 0..5, Z = reg32(Reg)

; (Op == / -> Result = L ; Result = [1,0|Rest] ), CodeL = [], Z = X ) )

% If Op = /, then result in %eax; if remainder, result in %edx

; L = [0], % If a single register was used, then it must be the case that

% this register is %eax, and %edx will be used also (it is

% clobbered by idivl, so it must be indicated as 'used').

(R = const(\_)

-> CodeL = [ '\n\t\t movl ',X,',',reg32(Reg) ], Reg in 0..5, Z = reg32(Reg),

(Op == / -> Result = [0,1,Reg] ; Result = [1,0,Reg])

; CodeL = [], (Op == / -> Result = [0,1] ; Result = [1,0]), Z = X ),

% If Op = /, then result in %eax; if remainder, result in %edx

% Code performing division/rem; Z is now the AL representation of right operand.

CodeOp = [ '\n\t\t movl ',reg32(0),',',reg32(1), % copy %eax into %edx

'\n\t\t shrl $31,',reg32(1), % %edx is now sign extension of %eax

'\n\t\t ',I,' ',Z ], % I = idivl, Z = right operand

all\_distinct(Result), % Constrain all allocated registers to be distinct.

append([CL,CodeL,CodeOp],Code). % Lay out the generated Code.

combine\_expr\_code(L,R,Op,[],CR,Code,[],Result) :- % Op = -, Left = non-reg, Right = reg

% The case when left subexpr is const/variable and

% right subexpression requires registers, for non-commutative

% operators --> may require an extra register

% A similar rule for the same combination of arguments, but

% for commutative operators, was given above.

(L = const(Tmp),atomic\_concat('$',Tmp,X) ; L = id(X)), R = [Reg|\_],

% X and Reg are the AL representation of Left and Right results

Op = -,!, I = subl,% minus is non-commutative

( R = [Reg,Reg1|Rest] % If a single register is used in computing

-> Result = [Reg1,Reg|Rest] % subexpression, then an extra register is required

; Result = [Reg1,Reg] ), % to compute current expression. Reg1 is either

% reused, or newly allocated, depending on the length

% of R (either 1 register or more).

% Generate code to perform subtraction

CodeOp = [ '\n\t\t movl ',X,',',reg32(Reg1), % order of operands is now SIGNIFICANT

'\n\t\t ',I,' ',reg32(Reg),',',reg32(Reg1) ],

Result ins 0..5, all\_distinct(Result), % Constrain newly allocated registers.

append([CR,CodeOp],Code). % Lay out the generated code

combine\_expr\_code(L,R,Op,[],CR,Code,[],Result) :- % Op = /,rem ; Left = non-reg, Right = reg

% Left operand is const/variable, and right operand requires

% registers. Operation is division/remainder

% May require an extra register

(L = const(Tmp),atomic\_concat('$',Tmp,X) ; L = id(X)), R = [H|T],

% X is the AL representation of left operand.

% H is the register holding the right result

member(Op,[/,rem]) ,!, I = idivl,

( H #\= 0, H #\= 1, % easy case, right operand does not require %eax or %edx

% then, register holding result not clobbered by division/rem

% try to force %eax, %edx as used registers, but not holding

% result, since they will be used anyway by idivl --> leads

% to register use minimization

( select(0,T,R1), select(1,R1,Rest) % Try enforcing both %eax,%edx be used in R

; select(0,T,Rest) % If not possible, try only %eax

; select(1,T,Rest) % If not possible, try only %edx

; Rest = T ), % Failsafe, leave it as it is

% Result will be in %eax or %edx, depending on Op = / or Op = rem

(Op == / -> Result = [0,1,H|Rest] ; Result = [1,0,H|Rest]),

% %eax and %edx can be used freely, since they do not hold

% anything important

CodeOp = [ '\n\t\t movl ',X,',',reg32(0), % Left operand must be moved into %eax

'\n\t\t movl ',reg32(0),',',reg32(1), % Sign extend %eax into %edx in usual way

'\n\t\t shrl $31,',reg32(1),

'\n\t\t ',I,' ',reg32(H) ] % I = idivl, H is right operand

; ( select(0,T,Rest) % Tough case, result of right operand in either %eax or %edx

; select(1,T,Rest) % Try enforcing the use of both regs in code of right operand (may fail)

; Rest = T ), % Failsafe case, all registers already concretized

% Check if new register is needed to save the result of

% right operand. Reg will either be an existing 'used'

% register (but not holding anything important, so that

% it can be used to hold a partial result) if possible, or a

% completely new, 'unused' register, if a 'used' one

% cannot be found.

( Rest = [] -> TRest = [] ; Rest = [Reg|TRest] ),

% Reorder the used registers, and

% add %eax and %edx if not in Result already

(Op = / -> Result = [0,1,Reg|TRest] ; Result = [1,0,Reg|TRest]),

% The result of right operand will be saved from either

% %eax or %edx into Reg. Then, %eax and %edx can be used

% in the idivl operation.

CodeOp = [ '\n\t\t movl ',reg32(H),',',reg32(Reg), % save %eax or %edx

'\n\t\t movl ',X,',',reg32(0), % load %eax with left opd

'\n\t\t movl ',reg32(0),',',reg32(1), % sign extend %eax into %edx

'\n\t\t shrl $31,',reg32(1),

'\n\t\t ',I,' ',reg32(Reg) ] % divide by second, saved operand

),

all\_distinct(Result), % Most registers are still unbound,

% make sure they will be distinct when allocated

append([CR,CodeOp],Code). % Lay out generated Code.

% The next 3 rules are the really gruesome ones. Both operands are in

% registers, and we distinguish between

% -- arithmetic operators excluding division and rem,

% -- division and rem

% -- relational operators.

% Since both operands require registers, we must make sure that as many

% of the registers are reused, while not clobbering the result of one

% expression while we're computing the other one. Either or both of the

% left and right subexpressions might have idivl as the toplevel

% operator -- this complicates things tremendously, resulting in many

% extra cases.

combine\_expr\_code(L,R,Op,CL,CR,Code,Residue,Result) :-

% Both expressions require registers, and the operator

% is boolean. Here we would like to use as many common

% registers as possible, so as to minimize register use.

% Generating code for the subexpression that requires more

% registers first will lead to 1 register holding the

% subexpression result, and the remaining registers to

% be potentially reused in the code generation of the

% other subexpression. If both subexpressions require the

% same number of registers, then an extra register will be

% required.

% This rule is for boolean operators, and is simpler

% because the result is stored in the flags (any register

% can be used for the residual code).

L = [HL|TL], R = [HR|TR], L ins 0..5, R ins 0..5, NewReg in 0..5,

% Left and right results are both lists of unbound variables

% The first element holds the result, the other elements

% represent registers that MUST be used during the evaluation

% of that expression (and thus will be clobered in the process,

% so the compiler can't expect the original values in those

% registers to be preserved). We constrain these variables

% to numbers between 0 and 5, so they can be mapped into

% the Pentium registers later.

member((Op,I),

[(<,setl),(=<,setle),(>,setg),(>=,setge),(==,sete),(\=,setne)]),!,

% Use member to translate the operator into the corresponding opcode

length(L,LgL), length(R,LgR),

% We distinguish 3 cases, based on the number of registers

% that are used in the code of each subexpression

% first case, when both subexpressions require same no. regs --> extra reg needed

( LgL #= LgR

-> % happy subcase, the two result registers can be constrained to be different

% (this is not always possible, for instance, not in the case when toplevel

% operator of both sides is division, and the result from both sides must

% reside in %eax)

( HL #\= HR, % enforce register reuse by making the right reg list

% a subset of the left tail + extra reg (denoted by '\_')

% -- this is where constraint programming comes in really handy!!!

permutation([\_|TL],R), Result = [HL,HR|TR],

% current operation implemented as comparison, result in flags

CodeOp = [ '\n\t\t cmpl ',reg32(HR),',',reg32(HL) ],

% Generated code made up of the left subexpression code,

% followed by right subexpression code, followed by

% current operator code.

append([CL,CR,CodeOp],Code),

HL in 0..3, % Residue moves flag result into register, so as to allow

% compilation of expressions of the form 1+(a<b)

% Result of residue can only be placed in either %eax,%edx,%ecx,%ebx

% since only these registers have an 8-bit corresponding register

Residue = [ '\n\t\t movl $0,',reg32(HL),

'\n\t\t ',I,' ',reg8(HL) ]

; % less happy subcase, the two result registers are the same

% this can only happen if the top-level operators of the left

% and right subexpressions are both division or both remainder

% extra register is needed to save right result

(HL #= 0 ; HL #=1), ( HR #= 0 ; HR #= 1 ),

% enforce register reuse, as above

permutation(TL,TR), Result = [NewReg,HL|TL],

% Generated code obtained by laying out

% the left subexpr code, followed by saving

% the result from either %eax or %edx into

% a new register, followed by the code for

% right subexpr, followed by comparison between

% %eax or %edx (depending on whether current Op is / or rem)

% and the saved register.

% Remember, here, HL and HR are actually the same. So, we lay out the code for

% left subexpression, then we need an instruction that saves HL into NewReg,

% which is a register not used in the code of the right subexpression. HL

% is about to be clobbered by the code of the right subexrpession, and without

% saving it into NewReg, we'd lose the result of the left side.

append( [ CL, % eval left expr

[ '\n\t\t movl ',reg32(HL),',',reg32(NewReg) ], % save result into NewReg

CR, % eval right expr

[ '\n\t\t cmpl ',reg32(HR),',',reg32(NewReg) ] ],% perform comparison

Code ) ), % The result is in fact the value of the processor's flags,

% and the caller can directly generate a jXX instruction

% based on the operator at the top level of the expression.

NewReg in 0..3, % only 4 regs allow setXX instructions

% residual code transfers flag result into a register

Residue = [ '\n\t\t movl $0,',reg32(NewReg), % The caller will check the context of the

'\n\t\t ',I,' ',reg8(NewReg) ] % current expression. If say, expression

% (x < y) appears in the context 1+(x<y),

% then the residue needs to be appended to

% Code, so as to have the result of the

% current expression in a register

% Second case, when right subexpr requires fewer regs than left

% No new register needed; lay out code for left subexpr first, then for right

% This makes sense because after evaluating the left expression, which needs more registers,

% we end up with a list of registers that have been used, but do not hold anything

% important, and are in sufficient number to be used in evaluating the subexpression

% on the right side. Thus, the total number of registers used is max(LgR,LgL).

; LgR #< LgL

-> ( HL #\= HR, % Happy case, result registers different.

% Permutation constraint ensures register reuse, as above

% No need for a new register

permutation(TL,TLP), append(R,\_,TLP), Result = L,

CodeOp = [ '\n\t\t cmpl ',reg32(HR),',',reg32(HL) ],

append([CL,CR,CodeOp],Code),

HL in 0..3,

Residue = [ '\n\t\t movl $0,',reg32(HL),

'\n\t\t ',I,' ',reg8(HL) ]

% less happy case, the two result registers are the same

% this can only happen if the top-level operators of the left

% and right subexpressions are both division or both remainder

% extra register may be needed to save right result, when

% len(Left) = len(Right)+1

; (HL #= 0 ; HL #=1), ( HR #= 0 ; HR #= 1 ),

% Enforce register reuse, as above

permutation(TL,TLP),

( append(TR,[NewReg|\_],TLP),!,select(NewReg,TL,TLRest)

; TR = TLP, TLRest = TL ),

% NewReg will be a new, 'unused' register, if

% a 'used' one cannot be found.

Result = [NewReg,HL|TLRest],

% NewReg used to save left subexpr result while

% the right subexrp is computed. NewReg also holds

% final result

append( [ CL,

[ '\n\t\t movl ',reg32(HL),',',reg32(NewReg) ],

CR,

[ '\n\t\t cmpl ',reg32(HR),',',reg32(NewReg) ] ],

Code ) ),

NewReg in 0..3,

Residue = [ '\n\t\t movl $0,',reg32(NewReg), % Residue as usual

'\n\t\t ',I,' ',reg8(NewReg) ]

% third case, when left subexpr requires fewer regs than right

; ( HL #\= HR, % happy case, result registers are different

% enforce register reuse

permutation(TR,TRP), append(L,\_,TRP),

select(HL,R,Rest), Result = [HL|Rest],

CodeOp = [ '\n\t\t cmpl ',reg32(HR),',',reg32(HL) ],

% Result of right subexpr will be naturally untouched

% during the computation of the left subexpr, because

% of the register reuse constraint

append([CR,CL,CodeOp],Code),

HL in 0..3,

Residue = [ '\n\t\t movl $0,',reg32(HL), % residue as usual

'\n\t\t ',I,' ',reg8(HL) ]

% less happy case, the two result registers are the same

; (HL #= 0 ; HL #=1), ( HR #= 0 ; HR #= 1 ),

% attempt to enforce as much reuse as possible

% NewReg may end up being a completely new register

permutation(TR,TRP),

( append(TL,[NewReg|\_],TRP),!,select(NewReg,TR,TRRest)

; TL = TRP, TRRest = TR ),

% NewReg is a new, 'unused' register, only if a 'used'

% one cannot be found. NewReg will be used to save

% the result of the right subexpr while the left one

% is being computed.

Result = [HR,NewReg|TRRest]),

% Generated code obtained by laying out the right

% subexpr code, followed by a save of the result

% into the new register, followed by the left

% subexpr code, followed by the computation of

% the current operator --> result left in flags

append( [ CR,

[ '\n\t\t movl ',reg32(HR),',',reg32(NewReg) ],

CL,

[ '\n\t\t cmpl ',reg32(NewReg),',',reg32(HL) ] ],

Code ),

HL in 0..3,

Residue = [ '\n\t\t movl $0,',reg32(HL),

'\n\t\t ',I,' ',reg8(HL) ] ),

% enforce that all registers used in computing this expression be

% distinct when subjected to labeling.

all\_distinct(Result). % Phew!!!

combine\_expr\_code(L,R,Op,CL,CR,Code,[],Result) :-

% Both expressions require registers, and the operator

% is arithmetic. Here we would like to use as many common

% registers as possible, so as to minimize register use.

% Generating code for the subexpression that requires more

% registers first will lead to 1 register holding the

% subexpression result, and the remaining registers to

% be potentially reused in the code generation of the

% other subexpression. If both subexpressions require the

% same number of registers, then an extra register will be

% required.

L = [HL|TL], R = [HR|TR], L ins 0..5, R ins 0..5, NewReg in 0..5,

% Left and right results are both lists of unbound variables

% The first element holds the result, the other elements

% represent registers that MUST be used during the evaluation

% of that expression (and thus will be clobered in the process,

% so the compiler can't expect the original values in those

% registers to be preserved). We constrain these variables

% to numbers between 0 and 5, so they can be mapped into

% the Pentium registers later.

member((Op,I),[(+,addl),(-,subl),(\*,imull),(/\,andl),(\/,orl)]),!,

% Use member to translate the operator into the corresponding opcode

length(L,LgL), length(R,LgR),

% We distinguish 3 cases, based on the number of registers

% that are used in the code of each subexpression

% first case, when both subexpressions require same no.

% regs --> extra reg needed

( LgL #= LgR

% happy case, the two result registers are different

-> ( HL #\= HR,

% enforce register reuse by making the right reg list

% a subset of the left tail + extra reg (denoted by '\_')

permutation([\_|TL],R), Result = [HL,HR|TR],

% current operation between the two result registers

CodeOp = [ '\n\t\t ',I,' ',reg32(HR),',',reg32(HL) ],

% Generated code made up of the left subexpression code,

% followed by right subexpression code, followed by

% current operator code.

append([CL,CR,CodeOp],Code)

% less happy case, the two result registers are the same

% this can only happen if the top-level operators of the left

% and right subexpressions are both division or both remainder

% extra register is needed to save right result

; (HL #= 0 ; HL #=1), ( HR #= 0 ; HR #= 1 ),

% enforce register reuse, as above

permutation(TL,TR), Result = [NewReg,HL|TL], NewReg in 0..5,

% Generated code obtained by laying out

% the left subexpr code, followed by saving

% the result from either %eax or %edx into

% a new register, followed by the code for

% right subexpr, followed by instruction that

% performs the current operator between the

% result of right subexpr, and the result

% of left subexpr saved in NewReg. The overall

% result is kept in NewReg.

append( [ CL,

[ '\n\t\t movl ',reg32(HL),',',reg32(NewReg) ],

CR,

[ '\n\t\t ',I,' ',reg32(HR),',',reg32(NewReg) ] ],

Code ) )

% second case, when right subexpr requires fewer regs than left

% no new register needed; lay out code for left subexpr first, then for right

; LgR #< LgL

-> ( HL #\= HR, % happy case, result registers different

% permutation constraint ensures register reuse, as above

permutation(TL,TLP), append(R,\_,TLP), Result = L,

CodeOp = [ '\n\t\t ',I,' ',reg32(HR),',',reg32(HL) ],

append([CL,CR,CodeOp],Code)

% less happy case, the two result registers are the same

% this can only happen if the top-level operators of the left

% and right subexpressions are both division or both remainder

% extra register may be needed to save right result, when

% len(Left) = len(Right)+1

; (HL #= 0 ; HL #=1), ( HR #= 0 ; HR #= 1 ),

% Enforce register reuse, as above

permutation(TL,TLP),

( append(TR,[NewReg|\_],TLP),!,select(NewReg,TL,TLRest)

; TR = TLP, TLRest = TL ),

% NewReg will be a new, 'unused' register, if

% a 'used' one cannot be found.

Result = [NewReg,HL|TLRest],

% NewReg used to save left subexpr result while

% the right subexrp is computed. NewReg also holds

% final result

append( [ CL,

[ '\n\t\t movl ',reg32(HL),',',reg32(NewReg) ],

CR,

[ '\n\t\t ',I,' ',reg32(HR),',',reg32(NewReg) ] ],

Code ) )

% third case, when left subexpr requires fewer regs than right

; ( HL #\= HR, % happy case, result registers are different

% enforce register reuse

permutation(TR,TRP), append(L,\_,TRP),

select(HL,R,Rest), Result = [HL|Rest],

CodeOp = [ '\n\t\t ',I,' ',reg32(HR),',',reg32(HL) ],

% Result of right subexpr will be naturally untouched

% during the computation of the left subexpr, because

% of the register reuse constraint

append([CR,CL,CodeOp],Code)

% less happy case, the two result registers are the same

; (HL #= 0 ; HL #=1), ( HR #= 0 ; HR #= 1 ),

permutation(TR,TRP),

( append(TL,[NewReg|\_],TRP),!,select(NewReg,TR,TRRest)

; TL = TRP, TRRest = TR ),

% NewReg is a new, 'unused' register, only if a 'used'

% one cannot be found. NewReg will be used to save

% the result of the right subexpr while the left one

% is being computed.

Result = [HR,NewReg|TRRest]),

% Generated code obtained by laying out the right

% subexpr code, followed by a save of the result

% into the new register, followed by the left

% subexpr code, followed by the computation of

% the current operator

append( [ CR,

[ '\n\t\t movl ',reg32(HR),',',reg32(NewReg) ],

CL,

[ '\n\t\t ',I,' ',reg32(NewReg),',',reg32(HL) ] ],

Code ) ),

all\_distinct(Result). % Phew!!! Phew!!!

combine\_expr\_code(L,R,Op,CL,CR,Code,[],Result) :-

% Both expressions require registers, and the operator

% is either division or remainder. Code generation is

% similar to the rule above, yet slightly

% more difficult due to the fact that the result of

% the left subexpression must end up in %eax

L = [HL|TL], R = [HR|TR], L ins 0..5, R ins 0..5, NewReg in 0..5,

% Left and right results are both lists of unbound variables

% The first element holds the result, the other elements

% represent registers that MUST be used during the evaluation

% of that expression (and thus will be clobered in the process,

% so the compiler can't expect the original values in those

% registers to be preserved). We constrain these variables

% to numbers between 0 and 5, so they can be mapped into

% the Pentium registers later.

member((Op,I),[(/,idivl),(rem,idivl)]),!,

% Use member to translate the operator into the corresponding opcode

length(L,LgL), length(R,LgR),

% We distinguish 3 cases, based on the number of registers

% that are used in the code of each subexpression

% first case, left and right subexpressions require same no of regs

( LgL #= LgR

% Enforce that left subexpr places result in %eax (preferred)

% or in %edx (cannot be avoided if left top level opr is remainder)

% happy case is that right subexpr puts result somewhere else

-> ( ( HL #= 0 ; HL #= 1), HR #\= 0, HR #\= 1,

Other #= 1 - HL,

% enforce reuse of registers, and the use of %eax and %edx

% in the right subexpr

permutation([\_|TR],L),

( select(0,R,R1), select(1,R1,R2)

; select(0,R,R2)

; select(1,R,R2)

; R2 = R ),

( Op == / -> Result = [0,1|R2] ; Result = [1,0|R2] ),

CodeOp = [ '\n\t\t movl ',reg32(HL),',',reg32(Other),

'\n\t\t shrl $31,',reg32(1),

'\n\t\t ',I,' ',reg32(HR) ],

% lay out the right subexpr code first (using %eax and %edx for

% temporary results if possible), and then left subexpr code,

% and then place the division code

append([CR,CL,CodeOp],Code)

% Enforce that left subexpr places result in %eax (preferred)

% or in %edx (cannot be avoided if left top level opr is remainder)

% less happy case -> right subexpr places result in same register

% An extra register is needed here to save result of right subexpr

; ( HL #= 0 #\/ HL #= 1 ), ( HR #= 0 #\/ HR #= 1 ),

OtherL #= 1-HL,

% enforce register reuse, and try reusing %eax and %edx in right subexpr

permutation(L,R),

( select(0,R,R1), select(1,R1,R2)

; select(0,R,R2)

; select(1,R,R2)

; R2 = R ),

% register order depends on operator

( Op == /

-> Result = [0,1,NewReg|R2] % %eax, then %edx

; Result = [1,0,NewReg|R2] ),% %edx, then %eax

% Lay out resulting code with code for right subexpr first,

% then save right result into new register, then copy

% %eax into %edx or viceversa, and then sign extend %edx,

% and then perform the division/remainder between %eax and

% new register

append( [ CR,

[ '\n\t\t movl ',reg32(HR),',',reg32(NewReg) ],

CL,

[ '\n\t\t movl ',reg32(HL),',',reg32(OtherL),

'\n\t\t shrl $31,',reg32(1),

'\n\t\t ',I,' ',reg32(NewReg) ] ],

Code ) )

% second case, left subexpr requires more registers than left one

; LgR #< LgL

% Enforce result of left subexpr be available in either %eax or %edx

-> ( ( HL #= 0 #\/ HL #= 1 ),

OtherL #= 1 - HL,

% Compute Save and Restore codes that may be inserted between

% codes for left and right subexpressions to preserve partial

% results

% enforce register reuse

permutation(L,LP),

( member(HL,R) % if reg of left result used in code of right subexpr

% (may be unavoidable due to / or rem)

-> ( append(R,[NewReg|\_],LP),!,select(NewReg,TL,TLRest)

% right registers subset of left registers

; R = LP, TLRest = TL ),

% find used or new register to save left result

( select(OtherL,TLRest,TLRest2) ; TLRest2 = TLRest ),

( (Op == /) % NewReg can be used to save result of left opd

-> Result = [0,1,NewReg|TLRest2]

; Result = [1,0,NewReg|TLRest2] ),

% the save code saves partial result into new register

Save = [ '\n\t\t movl ',reg32(HL),',',reg32(NewReg) ],

( HR #= 0 % right result in %eax

-> Restore = [ '\n\t\t xchg ',reg32(0),',',reg32(NewReg) ]

; HR #= 1 % right result in %edx

% The corresponding restore code:

-> Restore = [ '\n\t\t movl ',reg32(NewReg),',',reg32(0),

'\n\t\t movl ',reg32(1),',',reg32(NewReg) ]

; Restore = [ '\n\t\t movl ',reg32(NewReg),',',reg32(0) ])

; append(R,\_,LP), % reg of left result not used in code of

% right subexpr --> nothing to save/restore

(select(OtherL,TL,TL2) ; TL2 = TL ),

% make sure both %eax and %edx reused in

% code of left subexpr

( Op == /

-> Result = [0,1|TL2]

; Result = [1,0|TL2] ),

Save = [], Restore = [] ), % nothing to save/restore

( ( HR #= 0 #\/ HR #= 1 )

% repeat the test on where the right

% subexpression result is stored to

% generate the correct code for current operator

-> CodeOp = [ '\n\t\t movl ',reg32(0),',',reg32(1),

'\n\t\t shrl $31,',reg32(1),

'\n\t\t ',I,' ',reg32(NewReg) ]

; CodeOp = [ '\n\t\t movl ',reg32(0),',',reg32(1),

'\n\t\t shrl $31,',reg32(1),

'\n\t\t ',I,' ',reg32(HR) ] ),

append([CL,Save,CR,Restore,CodeOp],Code) )

% Third case, when right subexpr requires more registers than the left one

% Here we lay out the code for right subexpr first, and we constrain

% the result of left subexpr to be stored in %eax or %edx

% We also constrain %eax and %edx to be reused in computation

% of right subexpr, if possible. We also try to constrain the right

% result register to not be used in the computation of left subexpr; if

% that is not possible, then we save right result register into a

% new register

; ( ( HL #= 0 #\/ HL #= 1 ),

OtherL #= 1 - HL,

% Enforce register reuse

permutation(R,RP),

append(L,[NewReg|\_],RP),select(NewReg,R,RRest),

( select(0,RRest,RR1),select(1,RR1,RR2)

; select(0,RRest,RR2)

; select(1,RRest,RR2)

; RR2 = RRest ),

( notmember(HR,L), % if true, no need to save right register

Save = [],

CodeOp = [ '\n\t\t movl ',reg32(HL),',',reg32(OtherL),

'\n\t\t shrl $31,',reg32(1),

'\n\t\t ',I,' ',reg32(HR) ] ,

( Op == /

-> Result = [0,1|RR2]

; Result = [1,0|RR2] )

% if member(HR,L), then need to find new register, and save

; Save = [ '\n\t\t movl ',reg32(HR),',',reg32(NewReg) ],

CodeOp = [ '\n\t\t movl ',reg32(HL),',',reg32(OtherL),

'\n\t\t shrl $31,',reg32(1),

'\n\t\t ',I,' ',reg32(NewReg) ] ),

( Op == /

-> Result = [0,1,NewReg|RR2]

; Result = [1,0,NewReg|RR2] ) ),

% There's no need to restore here, since %eax/%edx store result

append([CR,Save,CL,CodeOp],Code) ),

all\_distinct(Result). % make sure all registers are distinct at labeling time.

% PHEW!!! PHEW!!! PHEW!!! PHEW!!! PHEW!!!

% predicate to check for lack of membership

% unlike \+ member(...), it does not fail when

% unbound variables are used

notmember(\_,[]).

notmember(X,[Y|T]) :- X \== Y, notmember(X,T).

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The compiler for statements and expressions starts here. Treat the

code above this line as a black box. You will not be required to

understand or modify that code.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

% Compiler of expressions

% 1st arg : program fragment to be translated

% 2nd arg : generated code for arg 1

% 3rd arg : attributes in

% 4th arg : attributes out

% Relevant attribute: context -- may have one of the following values:

% - expr : causes generation of code as if the current expression

% is a subexpression of a bigger expression

% Eg: current expr is (x+y), and is part of (x+y)/z

% The result will be available as 32 bit entity:

% reg/mem/con

% - stmt : causes generation of code as if the current expression

% is the boolean condition in an 'if' or 'while' statement

% The final instruction of generated code compares the

% result of expression with 0, and flags are set, so that

% a jump can be generated for efficient selection of next

% instruction

ce(X,Code,Ain,Aout) :- % generate code for variable

atom(X),!,

% retrieve memory reference for variable X

get\_assoc(local\_vars,Ain,Locals), % Retrieve list of local vars

get\_assoc(global\_vars,Ain,Globals), % Retrieve list of global vars

( member((X,Ref),Locals) % Check if local variable, and retrieve reference Ref

; member(X,Globals), Ref = X % Otherwise, check if global variable, the reference

% name in AL will be the atom name in this case

; writeln('Encountered undeclared variable'), abort),!,

% If var not found, exit with error message.

% result is id(Ref), update the attributes

put\_assoc(expr\_result,Ain,id(Ref),Aout),

% check context, and generate residue if necessary

get\_assoc(context,Ain,Ctx),

( Ctx = expr

-> Code = [] % empty code in expression context

; Code = [ '\n\t\t cmpl $0',Ref ] ).

% comparison code in 'if' or 'while' condition

ce(X@I,Code,Ain,Aout) :- % generate code for array access

atom(X),!,

get\_assoc(local\_vars,Ain,Locals),

get\_assoc(global\_vars,Ain,Globals),

( member((X,Ref),Locals)

; member(X,Globals), Ref = X

; writeln('Encountered undeclared variables in array access'), abort ),!,

C1 = [ '\n\t\t leal ',Ref,',',reg32(Base) ],

put\_assoc(context,Ain,expr,A0),

ce(I,CI,A0,A1),

get\_assoc(expr\_result,A1,Res),

( Res = [R,Base|\_], NewRes = Res, C2 = []

; ( Res = [R], C2 = []

; ( Res = id(Y) ; Res = const(T), atomic\_concat('$',T,Y)),

C2 = [ '\n\t\t movl ',Y,',',reg32(R) ]),

NewRes = [R,Base]), !,

NewRes ins 0..5, all\_distinct(NewRes),

put\_assoc(expr\_result,A1,NewRes,Aout),

append([CI,C1,C2,[ '\n\t\t movl (',reg32(Base),',',reg32(R),',4),',reg32(R) ]], Code).

ce(P#L,Code,Ain,Aout) :- % generate code for procedure call

% Arguments must be pushed on the stack in reverse order,

% so we reverse the list of actual args, if it's

% not empty or singleton

( L = (H,T) -> rev(T,H,LR) ; LR = L ),

% Code to save caller saved registers

CallerSaved = [ '\n\t\t pushl %ecx',

'\n\t\t pushl %edx' ],

% Generate code to evaluate each argument in LR (arguments

% are expressions) and push it on the stack

push\_args(LR,ArgC,Ain,A0,R), RR #= R\*4 ,

% Arguments will have to be cleared by caller upon return,

% To that end, the numnber of args is computed in R,

% and RR is the number of bytes to clear from the stack

% in order to deallocate the storage for the arguments

% The instruction that calls the procedure

Call = [ '\n\t\t call ',P ],

% The instruction to clear arguments from the stack

ResC = [ '\n\t\t addl $',RR,',%esp' ],

% Code to restore the caller-saved registers

CallerRestored = [ '\n\t\t popl %edx',

'\n\t\t popl %ecx' ],

% See whether the procedure was called in expr or stmt context

get\_assoc(context,Ain,Ctx),

% Generate the adequate residue

( Ctx = expr -> Residue = [] ; Residue = [ '\n\t\t cmpl $0,%eax' ] ),

% Lay out the code

append([CallerSaved,ArgC,Call,ResC,CallerRestored],Code),

% Record in the attributes that the result is in %eax, and

% since all other registers are restored to original values

% then we don't need to specify any other registers as "used"

put\_assoc(expr\_result,A0,[0],Aout).

ce(N,Code,Ain,Aout) :- % generate code for an integer

integer(N),!,

get\_assoc(context,Ain,Ctx),

( Ctx = expr

-> put\_assoc(expr\_result,Ain,const(N),Aout),

Code = [] % nothing to generate in expression context

; put\_assoc(expr\_result,Ain,[0],Aout),

% code for the case where N appears in an 'if' or 'while' cond

Code = [ '\n\t\t movl $',N,',%eax',

'\n\t\t cmpl $0,%eax' ] ).

ce(E,Code,Ain,Aout) :- % generate code for binary operator

E =.. [Op,EL,ER],!,

put\_assoc(context,Ain,expr,A0), % request expression context

ce(EL,CodeL,A0,A1), % recursively compile left subexpr

ce(ER,CodeR,A1,A2), % recursively compile right subexpr

get\_assoc(expr\_result,A1,ERL), % combine the codes and registers

get\_assoc(expr\_result,A2,ERR), % getting code C, Residue, and regs in Result

combine\_expr\_code(ERL,ERR,Op,CodeL,CodeR,C,Residue,Result),

get\_assoc(context,Ain,Ctx),

( Ctx == expr % Figure out if residual code needs to be

-> append(C,Residue,Code) % appended to currently generated code

; ( notmember(Op,[<,=<,==,\=,>=,>])

-> ( Result = [Temp|\_],R = reg32(Temp) ; Result = id(R) ),

append(C,['\n\t\t cmpl $0,',R],Code)

; Code = C ) ),

put\_assoc(expr\_result,A2,Result,Aout).

ce(E,Code,Ain,Aout) :- % generate code for unary operator

E =.. [Op,Es],!, % can be either unary minus, or logical negation

put\_assoc(context,Ain,expr,A0),

ce(Es,CodeS,A0,A1), % recursively compile argument of unary operator

get\_assoc(expr\_result,A1,Regs),

( Op = (-) % unary minus

-> ( Regs = const(N)

-> % If result is a constant

N1 #= (-N), Code = [], RegsOut = const(N1), Residue = []

% Evaluate negative of constant and return new const

; % Else...

( Regs = id(X)

-> % If result is an identifier

RegsOut = [NewReg],

% Allocate new register to hold result

CodeOp = [ '\n\t\t movl ',X,',',reg32(NewReg),

'\n\t\t negl ',reg32(NewReg) ],

% Code that loads identifier into register and

% negates the register

Residue = []

; % Else if result is in a register

Regs = [R|\_], RegsOut = Regs, % output registers same as registers of subexpr

% just perform negl on result register

CodeOp = [ '\n\t\t negl ',reg32(R) ],

Residue = [] ) )

; ( Op = (\) % logical negation

-> ( Regs = const(N)

-> % If result is a constant

N1 #<==> (N #= 0) , Code = [], RegsOut = const(N1)

% Evaluate and return new const

; % Else...

( Regs = id(X)

-> % If result is an identifier

RegsOut = [NewReg], NewReg in 0..3,

% Allocate new register

CodeOp = [ '\n\t\t movl ',X,',',reg32(NewReg),

'\n\t\t cmpl $0,',reg32(NewReg)],

% Code to load X into reg, and compare with 0

Residue = [ '\n\t\t movl $0,',reg32(NewReg),

'\n\t\t sete ',reg8(NewReg)]

% Residue transfers flags into NewReg

; % If result is in a register

Regs = [R|\_], RegsOut = Regs, R in 0..3,

% code to compare reg R with 0

CodeOp = [ '\n\t\t cmpl $0,',reg32(R) ],

% Residue transfers flags into R

Residue = [ '\n\t\t movl $0,',reg32(R),

'\n\t\t sete ',reg8(R) ] ) )

; RegsOut = Regs, CodeOp = [] ) ),

get\_assoc(context,Ain,Ctx),

( Ctx == expr % Figure out if residual code needs to be appended

-> append([CodeS,CodeOp,Residue],Code)

; ( notmember(Op,[<,=<,==,\=,>=,>])

-> ( RegsOut = [Temp|\_],R = reg32(Temp) ; RegsOut = id(R) ),

append([CodeS,CodeOp,['\n\t\t cmpl $0,',R]],Code)

; append([CodeS,CodeOp],Code) ) ),

put\_assoc(expr\_result,A1,RegsOut,Aout).

% Reverse a list made up from pairs of pairs. Useful to reverse

% the list of arguments of a procedure, when the arguments are

% about to be pushed on the stack. Assume that the list of args

% has the form (First,Rest). Then the call should be:

%

% rev(Rest,First,Reversed)

%

rev((X,Y),L,R) :- !, rev(Y,(X,L),R).

rev(X,L,(X,L)).

% Procedure to generate code that pushes a list of

% arguments on the stack. The list of arguments is

% assumed to be already reversed.

push\_args((X,Y),Code,Ain,Aout,Lgth) :- !,

put\_assoc(context,Ain,expr,A0),

comp\_expr(X,CX,A0,A1),

push\_result(A1,PushX),

push\_args(Y,CY,A1,Aout,LY),

append([CX,PushX,CY],Code),

Lgth #= LY + 1.

push\_args(void,[],A,A,0) :- !.

push\_args(X,Code,Ain,Aout,1):-

put\_assoc(context,Ain,expr,A0),

comp\_expr(X,CX,A0,Aout),

push\_result(Aout,PushC),

append(CX,PushC,Code).

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Compile expression wrapper

-- call this to generate code for an expression

-- remember to set attribute 'context' first

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

% -- same arguments as 'ce'

% -- calls label(...) to perform the actual numeric allocation

% -- run 'replace\_regs' to replace 'reg32(X)' constructs by the normal

% register names

% -- result in CER is assembly language code that evaluates expr E

comp\_expr(E,CER,Ain,Aout) :-

ce(E,CE,Ain,Aout),

get\_assoc(expr\_result,Aout,Result),

( Result = [\_|\_], Result ins 0..5, label(Result) ; true ),

replace\_regs(CE,CER).

% sometimes we need the result of an expression to be moved to

% a register, irrespective of where it has been computed.

% This predicate does just that

% Arg 1 (i) : Current attributes, containing the result of most recent

% expression in 'expr\_result'

% Arg 2 (o) : The register where the result has been placed

% Arg 3 (o) : Generated code (may be empty)

move\_result\_to\_reg(Attr,ResultReg,Code) :-

get\_assoc(expr\_result,Attr,RE), % retrieve the result storage for E

( RE = [Reg|\_]

-> Code = [], % if result is already in a register,

translate\_regs(Reg,ResultReg) % just return that register

; % Otherwise, load the const or id into %eax, and return %eax

( RE = id(Y) ; RE = const(Tmp), atomic\_concat('$',Tmp,Y) ),!,

Code = [ '\n\t\t movl ',Y,',%eax' ], ResultReg = '%eax' ).

% Generate code to transfer result of E into %eax

% Sometimes we need the result of the most recent expression transferred

% into the memory location of a variable. This predicate does just that.

% Arg 1 (i) : Current attributes, containing the result of most recent

% expression in 'expr\_result'

% Arg 2 (i) : Variable name where the result must be transferred

% Arg 3 (o) : Generated code

move\_result\_to\_var(Attr,Var,Code) :-

get\_assoc(expr\_result,Attr,Result),

( Result = id(Y)

-> % if the result is an identifier Y, which is not the same as Var

( Y \= Var

-> % move Y into Var via %eax

Code = [ '\n\t\t movl ',Y,',%eax',

'\n\t\t movl %eax,',Var ]

; % if Var == Y, do nothing

Code = [] )

; % otherwise, if the result is a constant or a register

( Result = const(Tmp),atomic\_concat('$',Tmp,Y) ;

Result = [Reg|\_], translate\_regs(Reg,Y) ),!,

% Y is now either the constant or the register in question

% just move the const or register into Var

Code = [ '\n\t\t movl ',Y,',',Var ] ).

% When we evaluate arguments to procedures

% (which are just usual expressions), we want

% their results to be pushed on the stack. This

% procedure achieves just this. The result of the

% expression is available in the 'expr\_result' attribute.

% Code is an output argument that represents the generated code.

push\_result(Attr,Code) :-

get\_assoc(expr\_result,Attr,R), % retrieve the result

( R = [Reg|\_]

-> translate\_regs(Reg,Src)

; ( R = const(T), atomic\_concat('$',T,Src) ; R = id(Src) ) ),!,

% Src is the AL representation of the result,

% irrespective of whether it is stored in a reg,

% id, or as a const

Code = [ '\n\t\t pushl ',Src ].% Just push Src

% Process global variable declarations. The list of global variables is

% enumerated in a pairs of pairs type of list.

% Each variable is added to a list stored in the attribute global\_vars

% Each reference to an identifier will be checked to have been declared

% in either the global or local variable list.

global\_vars((VH,VT),Ain,Aout) :-

get\_assoc(global\_vars,Ain,VS,A0,[VH|VS]),

notmember(VH,VS),!,

global\_vars(VT,A0,Aout).

global\_vars(V,Ain,Aout) :-

V =.. L, L \= [\_,\_|\_],

get\_assoc(global\_vars,Ain,VS,Aout,[V|VS]),

notmember(V,VS),!.

% Helper that would map each local variable into an offset N, so

% that the variable can be referred as -N(%ebp) later in the code

% Called by 'local\_vars'

local\_vars\_helper(V,Ain,Aout) :-

% allocate space on the stack for a local variable

% TopIn indexes the most recently allocated variable, so

% the current variable will be stored at TopIn+4 (downwards from %ebp)

% Store the maximum amount of space allocated so far in max\_local\_vars,

% so as to be able to allocate space conservatively at the start of the program

get\_assoc(top\_local\_vars,Ain,TopIn,A0,TopOut),

get\_assoc(max\_local\_vars,A0,MaxIn,A1,MaxOut),

get\_assoc(local\_vars,A1,VS,Aout,[(V,Ref)|VS]),

TopOut #= TopIn + 4, atomic\_list\_concat([-,TopOut,'(%ebp)'],Ref),

MaxOut #= max(MaxIn,TopOut).

% Process local variable declarations. Each variable is allocated

% on the stack, and translated into a memory reference of the form

% -N(%ebp), where N must be a constant. Every reference to an

% identifier will be searched first in the list of local vars, and

% then in the list of global vars. For local vars, the identifier

% will be translated into the corresponding ebp-based memory reference.

local\_vars((VH,VT),Ain,Aout) :- !,

local\_vars\_helper(VH,Ain,Aaux),

local\_vars(VT,Aaux,Aout).

local\_vars(V,Ain,Aout) :- !,

V =.. L, L \= [\_,\_|\_],

local\_vars\_helper(V,Ain,Aout).

% Helper that would map each formal argument of a procedure

% into an offset N, so that the variable can be referred as

% N(%ebp) later in the code. Called by 'proc\_args'.

proc\_args\_helper(V,Ain,Aout) :-

get\_assoc(top\_args,Ain,Tin,A0,Tout),

get\_assoc(local\_vars,A0,VS,Aout,[(V,Ref)|VS]),

Tout #= Tin + 4, atomic\_list\_concat([Tout,'(%ebp)'],Ref).

% Procedure that iterates through a list of identifiers,

% assumed to be the list of formal arguments of a procedure,

% calling 'proc\_args\_helper' on each of them. The end result

% is that all the arguments will appear in the local symbol

% table, with the corresponding mappings, ready to be referenced

% throughout the compilation of the current scope.

proc\_args((VH,VT),Ain,Aout) :- !,

proc\_args\_helper(VH,Ain,Aaux),

proc\_args(VT,Aaux,Aout).

proc\_args(V,Ain,Aout) :- !,

V =.. L, L \= [\_,\_|\_],

proc\_args\_helper(V,Ain,Aout).

% Compile statement -- implements while language with procedure calls

cs((global VL ; S),Code,Ain,Aout) :-!,

% Process global variable declarations,

% then compile S as usual

% Reverse the list of vars so as to handle arrays properly

( VL = (H,T) -> rev(T,H,VLR) ; VLR = VL ) ,

global\_vars(VLR,Ain,A0),

cs(S,Code,A0,Aout).

cs({local VL ; S},Code,Ain,Aout) :- !,

% Preserve the original attribute, and restore at end of block

get\_assoc(local\_vars,Ain,OriginalLocalVars),

get\_assoc(top\_local\_vars,Ain,OriginalTopLocalVars),

% Process local variable declarations at top of block

( VL = (H,T) -> rev(T,H,VLR) ; VLR = VL ), % reverse list of vars for array handling

local\_vars(VLR,Ain,A0),

% compile the rest of the statements

cs(S,Code,A0,A1),

% restore original list of local variables, and allocation space

put\_assoc(local\_vars,A1,OriginalLocalVars,A2),

put\_assoc(top\_local\_vars,A2,OriginalTopLocalVars,Aout).

cs(break, Code,Ain,Aout) :- !,

% Generate a label name, and generate a jump to that label

Code = [ '\n\t\t jmp ',Lbl ],

get\_assoc(labelsuffix,Ain,LabelSuffixIn,A,LabelSuffixOut),

generateLabels([Lbl],LabelSuffixIn,LabelSuffixOut),

% add the label as an attribute, so that the enclosing 'while'

% will know to generate code for that label

put\_assoc(break,A,Lbl,Aout).

cs(continue, Code,Ain,Aout) :- !,

% Generate a label name, and generate a jump to that label

Code = [ '\n\t\t jmp ',Lbl ],

get\_assoc(labelsuffix,Ain,LabelSuffixIn,A,LabelSuffixOut),

generateLabels([Lbl],LabelSuffixIn,LabelSuffixOut),

% add the label as an attribute, so that the enclosing 'while'

% will know to generate code for that label

put\_assoc(continue,A,Lbl,Aout).

cs((X=E),Code,Ain,Aout) :- !, atom(X), % assignment

put\_assoc(context,Ain,expr,A1), % request expression context

comp\_expr(E,CE,A1,Aout), % compile right hand side

% retrieve memory reference for variable X

get\_assoc(local\_vars,Ain,Locals),

get\_assoc(global\_vars,Ain,Globals),

( member((X,Ref),Locals)

; member(X,Globals), Ref = X

; writeln('Encountered undeclared variable'), abort),!,

% Check where the result of rhs is stored, and transfer it into

% storage of variable X

move\_result\_to\_var(Aout,Ref,Cop),

append(CE,Cop,Code).

cs((if B then { S1 } else { S2 }), Code,Ain,Aout) :- !,

% For if-then-else statement, compile boolean

% condition first. Set context to 'stmt', so that

% residual code is not used.

put\_assoc(context,Ain,stmt,A1),

comp\_expr(B,CB,A1,A2),

% The result is in the flags, and the appropriate

% jump instruction must be used to select

B =.. [Op|\_], % between 'then' and 'else' branches

( member((Op,I),[(<,jl),(=<,jle),(==,je),(\=,jne),(>,jg),(>=,jge)])

-> true

; I = jne ), % code for 'then' branch appears below code for 'else'

COpB = [ '\n\t\t ',I,' ',Lthen ],

cs({S2},C2,A2,A3), % generate code for 'else', and jump to skip the 'then'

Cif1 = [ '\n\t\t jmp ',Lifend,

'\n',Lthen,':' ],

% label for 'then' branch is 'Lthen:'

cs({S1},C1,A3,A4), % generate code for 'then' branch

Cif2 = [ '\n',Lifend,':' ],

% 'Lifend:' is the label at end of 'if',

% target of jump placed after 'else'

get\_assoc(labelsuffix,A4,Kin,Aout,Kout),

generateLabels([Lthen,Lifend],Kin,Kout),

% generate concrete labels for label placeholders

% and lay out the code

append([CB,COpB,C2,Cif1,C1,Cif2],Code).

cs((if B then { S }), Code,Ain,Aout) :- !,

% Code for 'if-then', similar to the one above.

put\_assoc(context,Ain,stmt,A1),

comp\_expr(B,CB,A1,A2),

B =.. [Op|\_], % The condition of the jump must now be negated

( member((Op,I),[(<,jge),(=<,jg),(==,jne),(\=,je),(>,jle),(>=,jl)])

-> true

; I = je ),

COpB = [ '\n\t\t ',I,' ',Lifend ],

cs({S},C,A2,A3),

Cif = [ '\n',Lifend,':' ],

get\_assoc(labelsuffix,A3,Kin,Aout,Kout),

generateLabels([Lifend],Kin,Kout),

append([CB,COpB,C,Cif],Code).

cs((while B do { S }), Code,Ain,Aout) :- !,

% The while has the body first, and then the condition

% 'break' statement are allowed inside, so we must

% make sure that we accomodate only breaks issued

% from inside this loop's body. For this reason,

% previous 'break' labels will be saved here, and

% restored before returning.

get\_assoc(break,Ain,OrigBreak,A0,none),

% The local 'break' attribute currently set to 'none'.

% Will be checked again after generating code for the

% body. If 'break' attribute is changed, we are sure

% it comes from inside this loop's body.

get\_assoc(continue,A0,OrigContinue,A1,none),

Cw1 = [ '\n\t\t jmp ',LCond ,

'\n',LTop,':' ],

% Jump to while condition; generate label for looping

cs({S},C,A1,A2), % Then generate code for body

Cw2 = [ '\n',LCond,':' ],

% Generate label 'LCond:' for bool condition

put\_assoc(context,A2,stmt,A3),

% Generate code for boolean condition, in

% 'stmt' context so that residual code is not used.

comp\_expr(B,CB,A3,A4), % Results will be in the flags

B =.. [Op|\_], % Appropriate conditional jump must be selected

( member((Op,I),[(<,jl),(=<,jle),(==,je),(\=,jne),(>,jg),(>=,jge)])

-> true

; I = jne ),

COpB = [ '\n\t\t ',I,' ',LTop ], % Generate code for conditional jump

% that repeats the loop. Loop is exited if

% conditional jump not taken.

get\_assoc(labelsuffix,A4,Kin,A5,Kout),

generateLabels([LTop,LCond],Kin,Kout),

% Generate concrete label names

get\_assoc(break,A5,Break,A6,OrigBreak),

% If 'break' was encountered in the body,

% then generate 'break' target outside

% while loop. Also restore the original

% break label, so that any enclosing 'while'

% loop may have its own 'break' handled

% correctly.

( Break = none -> Cbreak = [] ; Cbreak = ['\n',Break,':'] ),

% Then do the same for 'continue'

get\_assoc(continue,A6,Continue,Aout,OrigContinue),

( Continue = none -> Ccontinue = [] ; Ccontinue = ['\n',Continue,':'] ),

% Lay out the entire code.

append([Cw1,C,Cw2,Ccontinue,CB,COpB,Cbreak],Code).

cs((for (X1=E1;B;X2=E2) do { S }), Code,Ain,Aout) :- !,

% Implementation of 'for' loops, is similar to 'while'

% Syntax of 'for' is more restricted than the one of 'C'

% 1st and 3rd components of the 'for' bracket are restricted

% to assignments. They could, in principle, be any expressions,

% but in our language, the assignment is not an expression,

% and then we would need to accomodate both expressions and assigs

get\_assoc(break,Ain,OrigBreak,A0,none),

get\_assoc(continue,A0,OrigContinue,A1,none),

% Handle 'break' and 'continue' attrs similar to 'while' rule

cs(X1=E1,C1,A1,A2), % compile first assignment

Cf1 = [ '\n',LTop,':' ], % Generate 'top-of-loop' label

put\_assoc(context,A2,stmt,A3), % Compile boolean condition in statement context

comp\_expr(B,CB,A3,A4),

B =.. [Op|\_], % select the correct jump instruction after boolean condition evaluation

( member((Op,I),[(<,jge),(=<,jg),(==,jne),(\=,je),(>,jle),(>=,jl)])

-> true

; I = je ), % generate jump code, to 'out-of-loop' label

COpB = [ '\n\t\t ',I,' ',Lforend ],

cs({S},CS,A4,A5), % compile body of for loop

get\_assoc(continue,A5,Continue,A6,OrigContinue),

( Continue = none -> Ccontinue = [] ; Ccontinue = ['\n',Continue,':'] ),

% Generate 'continue' label placeholder if a 'continue' stmt

% was encountered, otherwise leave the code empty ;

% also restore original 'continue' label, so as to allow correct

% handling of 'continue' statements placed outside the current loop

cs(X2=E2,C2,A6,A7), % Compile updating assignment

Cf2 = [ '\n\t\t jmp ',LTop, % Generate jump to top of loop, to repeat the loop

'\n',Lforend,':'], % Generate 'out-of-loop' label placeholder

get\_assoc(labelsuffix,A7,Kin,A8,Kout),

generateLabels([LTop,Lforend],Kin,Kout),

% Fill placeholders with concrete label names

get\_assoc(break,A8,Break,Aout,OrigBreak),

( Break = none -> Cbreak = [] ; Cbreak = ['\n',Break,':'] ),

% Generate label for 'break' if a 'break' stmt was encountered,

% otherwise leave the code emtpy. Also restore original 'break'

% attribute, so as to allow correct handling of 'break' statements

% residing outside the current loop.

append([C1,Cf1,CB,COpB,CS,Ccontinue,C2,Cf2,Cbreak],Code).

% Lay out the code.

cs((N::{S} ; Rest),Code,Ain,Aout) :-

% Compilation of the arms of a 'switch' statement

% It compiles the arm N::{S} first, and recursively invokes

% the compiler for the remaining arms.

integer(N),!,

get\_assoc(case\_table\_labels,Ain,(VL,LL),A1,([N|VL],[L|LL])),

% Create a new label placeholder and associate it to case label N

% Add the association to the case table labels attribute.

Ccase1 = [ '\n',L,':' ], % Generate code with label placeholder for current arm

cs({S},CS,A1,A2), % Compile statement S

get\_assoc(label\_case\_end,A2,Lend),

% Retrieve 'end-of-switch' label

Ccase2 = [ '\n\t\t jmp ',Lend ], % Generate jump to 'end-of-switch' label so as to implement the

% invisible break.

cs(Rest,CodeRest,A2,Aout), % Recursively compile the remaining arms

append([Ccase1,CS,Ccase2,CodeRest], Code).

% Lay out the entire code.

cs((default::{S}),Code,Ain,Aout) :- !,

% Last arm of 'switch' is 'default' arm

get\_assoc(case\_table\_labels,Ain,(VL,LL),A1,([default|VL],[L|LL])),

% Create new label placeholder and associate it with 'default' case label

Ccase1 = [ '\n',L,':' ], % Generate code with new label placeholder

cs({S},CS,A1,Aout), % Compile body of 'default' arm

append([Ccase1,CS], Code). % Lay out the entire code.

cs(switch E of { CaseList },Code,Ain,Aout) :-

% Compilation of 'switch' statements

get\_assoc(label\_case\_end,Ain,OldCaseLabel,A0,Lcaseend),

% Generate new 'out-of-switch-statement' label placeholder

% and place it in the attribute set, so it can be accessed

% by the cs(N::{S} ; ... ) rule given above

% Also, save the original value of this attribute, so it

% it can be restored later

get\_assoc(case\_table\_labels,A0,OldCaseTableLabels,A1,([],[])),

% Initialize the table of case labels, and store it in the attribute set

% Again, save the old value so it can be restored later

put\_assoc(context,A1,expr,A2),

comp\_expr(E,CE,A2,A3), % compile E in expr context

move\_result\_to\_reg(A3,X,Cop),

CJ = [ '\n\t\t jmp \* ',Lcasetab,'(,',X,',4)',

'\n',Lcasetab,':' ], % Generate indirect jump to arm code via the case table

% Generate label placeholder destined to hold the base address of

% case table

get\_assoc(labelsuffix,A3,Lin,A4,Lout),

generateLabels([Lcasetab,Lcaseend],Lin,Lout),

% Generate concrete label names for the two placeholders

cs({CaseList},CC,A4,A5), % Recursively compile the case list; build a (list-based)

% dictionary of (CaseLabel,AssemblyLabel) pairs

casetable(CodeCaseTab,A5,A6), % Build case-table, filled with labels pointing to arms ;

% each entry corresponds to either an exisitng case label,

% or to 'default', and the contents of the entry is filled

% accordingly with the assembly label associated with the

% case label in the dictionary.

CodeEnd = [ '\n',Lcaseend,':' ], % Generate code for 'out-of-switch-statement' label

put\_assoc(label\_case\_end,A6,OldCaseLabel,A7),

put\_assoc(case\_table\_labels,A7,OldCaseTableLabels,Aout),

% Restore original case label attribute values. Necessary for correct

% handling of nested switch statements.

append([CE,Cop,CJ,CodeCaseTab,CC,CodeEnd],Code).

% Lay out the entire code

cs(skip,[],A,A) :- !.

cs((S1;S2),Code,Ain,Aout) :- !,

( S1 \= (global \_), S1 \= (\_#\_::\_),

get\_assoc(top\_args,Ain,none),

get\_assoc(entry,Ain,none,A1,some)

-> Pre = [ '\n\t\t .globl \_entry',

'\n\_entry:',

'\n\t\t pushl %ebp',

'\n\t\t movl %esp,%ebp']

; Pre = [], A1 = Ain ),

cs(S1,C1,A1,A2),

cs(S2,C2,A2,Aout),

( Pre \= []

-> get\_assoc(max\_local\_vars,Aout,Max),

( Max == 0

-> AllocC = []

; AllocC = ['\n\t\t subl $',Max,',%esp'] ),

Post = ['\n\t\t movl %ebp,%esp',

'\n\t\t popl %ebp',

'\n\t\t ret']

; Post = [], AllocC = [] ),

append([Pre,AllocC,C1,C2,Post],Code).

cs((S;),Code,Ain,Aout) :- !, % statement terminated by semicolon

cs((S;skip),Code,Ain,Aout). % compile S without semicolon

cs({S},Code,Ain,Aout) :- !, % statement enclosed in braces

cs(S,Code,Ain,Aout). % compile S without braces

cs(return X, Code, Ain, Aout) :- !, % evaluates expression X and loads result into %eax

put\_assoc(context,Ain,expr,A1),

ce(X,CX,A1,Aout), % Don't call 'comp\_expr', try to enforce result into %eax through constraint solving

get\_assoc(expr\_result,Aout,Result),

% Figure out where the result is, and generate correct transfer instruction

% If result in register, unify first element with 0, in attempt to end up with

% result right there, and save on the transfer instruction

( Result = [0|\_], Result ins 0..5, label(Result), Res = [],! ;

(Result = const(T),atomic\_concat('$',T,N) ; Result = [R|\_],N=reg32(R) ; Result = id(N)),!,

Res = ['\n\t\t movl ',N,',%eax'] ),

% registers are still in reg32(X) form, need to be translated

replace\_regs(CX,C),

% lay out the code

append(C,Res,Code).

cs((P#L::{S}),Code,Ain,Aout) :- !, % Generate code for procedure definition

% Generate procedure label

ProcLabel = ['\n',P,':'],

% Preserve the original attributes, and restore at end of block

get\_assoc(local\_vars,Ain,OriginalLocalVars),

get\_assoc(top\_local\_vars,Ain,OriginalTopLocalVars,Atlv,0),

get\_assoc(max\_local\_vars,Atlv,OriginalMaxLocalVars,Amlv,0),

put\_assoc(top\_args,Amlv,4,Ata),

% Process formal arguments

proc\_args(L,Ata,Apa),

% compile the procedure body (may contain local variable declarations)

cs({S},CodeS,Apa,Acs),

% retrieve the amount of space used for local variables

get\_assoc(max\_local\_vars,Acs,Max),

% Generate procedure's prologue, which saves the old frame pointer

% and loads the frame pointer register with the current top of the stack.

% After execution of this code, all arguments can be referred to via their

% mappings stored in the local variables attribute

Prologue = [ '\n\t\t pushl %ebp',

'\n\t\t movl %esp,%ebp' ],

% Check if allocation needed for local variables and generate adequate code

( Max == 0 -> AllocCode = [] ; AllocCode = [ '\n\t\t subl $',Max,',%esp' ] ),

% Registers %ebx, %esi, %edi are callee saved. The procedure should preserve

% their original values. We save them unconditionally, which is not very efficient.

% A better alternative would be to check the code for the procedure's body, and

% save them only if they are used there.

CalleeSaved = [ '\n\t\t pushl %ebx\n\t\t pushl %esi\n\t\t pushl %edi' ],

% What is saved needs to be restored

CalleeRestored = [ '\n\t\t popl %edi\n\t\t popl %esi\n\t\t popl %ebx' ],

% The epilogue restores the frame pointer to its original value, and returns

% to the caller

Epilogue = [ '\n\t\t movl %ebp,%esp',

'\n\t\t popl %ebp',

'\n\t\t ret'],

% lay out the code

append([ProcLabel,Prologue,AllocCode,CalleeSaved,CodeS,CalleeRestored,Epilogue],Code),

% Restore saved attributes

put\_assoc(top\_args,Acs,none,Atopa),

put\_assoc(local\_vars,Atopa,OriginalLocalVars,Alv),

put\_assoc(max\_local\_vars,Alv,OriginalMaxLocalVars,Aomlv),

put\_assoc(top\_local\_vars,Aomlv,OriginalTopLocalVars,Aout).

cs((P#L),Code,Ain,Aout) :- !, % Procedure call as statement (and not as expression)

put\_assoc(context,Ain,expr,A0), % Compile as expression, do not store value.

comp\_expr((P#L),Code,A0,Aout).

% Generate the case table, after having processed all the case arms

% and having obtained the list of associations (CaseLabel,AssemblyLabel)

casetable(Code,Ain,Aout) :-

get\_assoc(case\_table\_labels,Ain,(CTV,CTL),A1,none),

% get the association lists, and replace them with 'none' at the same time

get\_assoc(labelsuffix,A1,Lin,Aout,Lout),

generateLabels(CTL,Lin,Lout),

% The Assembly Labels are actually still unbound, so label names need to

% be generated

CTV = [default|CTVT], CTL = [DefaultLabel|CTLT],

% The first label is 'default'. This will be used as a filler for all

% entries that do not have a corresponding case label, so it's useful

% to have it as a separate argument fed into the helper predicate.

reverse(CTVT,VR), reverse(CTLT,LR),

% The association lists have grown in reverse order, so we restore them

% to the original label, as it appears in the list of case arms

casetable\_helper(VR,LR,CodeL,0,DefaultLabel),

% All the work is done here. 0 is the first table entry to be processed

% and this argument will increase throughout the recursive calls, to

% allow iteration through all the table entries

append(CodeL,Code). % Lay out the entire code.

% Generate the case table, line by line. Case labels are restricted

% to values between 0-255, and thus the table will have only

% 256 labels. The first 2 arguments are expecte to have the same length

% Args:

% 1st : list of numeric case labels encountered in the list of

% case arms

% 2nd : list of assembly language labels, in the same order, so

% that elements of the same rank in the 1st and 2nd arg

% are associated with each other

% 3rd : List of assembly language lines (each in a separate list --

% so that the final result is a list of lists), each line

% being a directive to reserve space for 1 table entry

% 4th : current table position, incremented by 1 at each

% recursive call of the helper.

% 5th : Assembly language label corresponding to 'default'.

% Used to fill in location for which a case label was

% not provided.

casetable\_helper([],[],[],256,\_) :- !.

% If we exhausted all the case labels, and reached the end of the array,

% generate empty code in 3rd arg

casetable\_helper([],[],[C|CT],N,DL) :-

% If we exhausted all the case labels, but not reached the end of

% the array, fill the current entry with 'default' label

N < 256, !,

C = [ '\n\t\t .long ',DL ],

N1 #= N+1,

% move on to next position in the table

casetable\_helper([],[],CT,N1,DL).

% recursive call to fill remaining positions in the table

casetable\_helper([VH|VT],[LH|LT],[CH|CT],N,DL) :-

% if VH is the first unprocessed label, and yet the

% current table position is smaller than VH, then

% fill the current position with 'default' label

N < VH, !,

CH = [ '\n\t\t .long ',DL ],

N1 #= N+1,

% move on to next position in the table

casetable\_helper([VH|VT],[LH|LT],CT,N1,DL).

% recursive call to fill the remaining positions in the table.

casetable\_helper([N|VT],[LH|LT],[CH|CT],N,DL) :-

% If the first unprocessed label is N, equal to

% the current table position, then fill the current entry

% with LH, the assembly language label associated to N

CH = [ '\n\t\t .long ',LH ],

N1 #= N+1,

% move on to next position

casetable\_helper(VT,LT,CT,N1,DL).

% recursive call to fill the remaining positions in the table.

% Main predicate

% 1st arg : Program to be compiled

% 2nd arg : File for output

% The generated file has to be compiled together with runtime-stmt.c

% to produce a valid executable. Should work on Linux, Mac, and Cygwin.

compile(P,File) :-

tell(File), % open output file

empty\_assoc(Empty), % initialize attribute dict

AbreakIn = Empty,

put\_assoc(break,AbreakIn,none,AbreakOut),

AcontIn = AbreakOut, % initial 'break' label is none

put\_assoc(continue,AcontIn,none,AcontOut),

AcaseendIn = AcontOut, % initial 'continue' label is none

put\_assoc(label\_case\_end,AcaseendIn,none,AcaseendOut),

AcasetablelabelsIn = AcaseendOut, % initial case-end label is none

put\_assoc(case\_table\_labels,AcasetablelabelsIn,([],[]),AcasetablelabelsOut),

AlabelsuffixIn = AcasetablelabelsOut, % initial table has no labels (empty lists)

put\_assoc(labelsuffix,AlabelsuffixIn,0,AlabelsuffixOut),

AlocalvarsIn = AlabelsuffixOut, % initialize label counter

put\_assoc(local\_vars,AlocalvarsIn,[],AlocalvarsOut),

AglobalvarsIn = AlocalvarsOut, % initial local vars list is empty

put\_assoc(global\_vars,AglobalvarsIn,[],AglobalvarsOut),

AtoplocalIn = AglobalvarsOut, % initial global vars list is empty

put\_assoc(top\_local\_vars,AtoplocalIn,0,AtoplocalOut),

AmaxlocalIn = AtoplocalOut, % current allocation size is 0

put\_assoc(max\_local\_vars,AmaxlocalIn,0,AmaxlocalOut),

% max allocation size is 0

AentryIn = AmaxlocalOut,

put\_assoc(entry,AentryIn,none,AentryOut),

AtaIn = AentryOut,

put\_assoc(top\_args,AtaIn,none,AtaOut),

Ainit = AtaOut,

cs(P,Code,Ainit,Aresult),!, % Compile program P into Code

% -- Code is now a list of atoms

% that must be concatenated to get

% something printable

All = ['.text'|Code],

atomic\_list\_concat(All,AllWritable), % Now concat and get writable atom

writeln(AllWritable), % Print it into output file

get\_assoc(global\_vars,Aresult,VarList), % Create data declarations for all vars

allocvars(VarList,VarCode,VarNames,VarPtrs),

% Code to allocate all global variables

atomic\_list\_concat(VarCode,WritableVars),

% Compound the code into writable atom, for output into file

write('\n\t\t .data\n\t\t .globl \_\_var\_area\n\_\_var\_area:\n'),

% Write declarations to output file

write(WritableVars),

% Create array of strings representing

% global variable names, so that vars can

% be printed nicely from the runtime

atomic\_list\_concat(VarNames,WritableVarList),

write('\n\n\t\t .globl \_\_var\_name\_area\n\_\_var\_name\_area:\n'),

write(WritableVarList),

% Create array of pointers to strings

% so that runtime code doesn't need

% to be changed every time we compile

atomic\_list\_concat(VarPtrs,WritableVarPtrs),

write('\n\n\t\t .globl \_\_var\_ptr\_area\n\_\_var\_ptr\_area:\n'),

write(WritableVarPtrs),

write('\n\n\_\_end\_var\_ptr\_area:\t .long 0\n'),

% Put null pointer at the end of array of string pointers,

% to indicate that the array has ended.

told. % close output file

% Usage example

%

:- Program =

( global i,a,b,c,d,e,f,min1,min2 ;

fmin#(void) ::

{ local i,x,x1,x2,x3,x4,x5 ;

x = 9 ; x1 = 7 ; x2 = 4 ; x3 = 6 ; x4 = 8 ; x5 = 10 ;

i = 0 ; min1 = 100 ;

while i < 6 do

{ if min1 > x@i then { min1 = x@i } ;

i = i + 1

}

} ;

fmin#(void);

a = 90 ; b = 59 ; c = 30 ; d = 45 ; e = 23 ; f = 94 ;

min2 = 100 ; i = 0 ;

while i < 6 do

{ if min2 > a@i then { min2 = a@i } ;

i = i + 1 ;

}

), compile(Program,'test.s').