Interpreters

How they work:

Step 1:

Translate the source program to an internal form: quadruples, postfix, abstract syntax tree, ... or to instructions to an abstract machine, e.g. P-code for Pascal and Modula-2, Z-code for Algol, Diana for ADA, Byte-code (JVM—Java Virtual Machine) for Java.

Step 2:

Execute (simulate the abstract machine, interpret) the internal form.

An interpreter is used as a rule for "Step 2".

You can

1. Interpret the source code directly, e.g. command decoders.

2. Interpret the internal form:

   Variant A:

   Translation is not performed too far, the symbol table must be saved (e.g. Basic, Lisp, APL $\Rightarrow$ good for symbolic debugging).

   Variant B:

   Code generation to an abstract machine (e.g. P-code). The symbol table is usually discarded. Type information, size, addresses, etc., are implicit in the code.

P-code (JVM code for Java is similar)

P-code contains instructions for an abstract stack machine.

Examples of instructions:

`SBI`	*Subtract Integers*	`push(-<pop,pop>)`
`SROI Q`	*Store Int. var in* `Q`	`var(Q) := pop`
`LDOI Q`	*Load Int. var*	`push(var(Q))`
`JMP Q`	Jump to `Q`	`pc := Q`

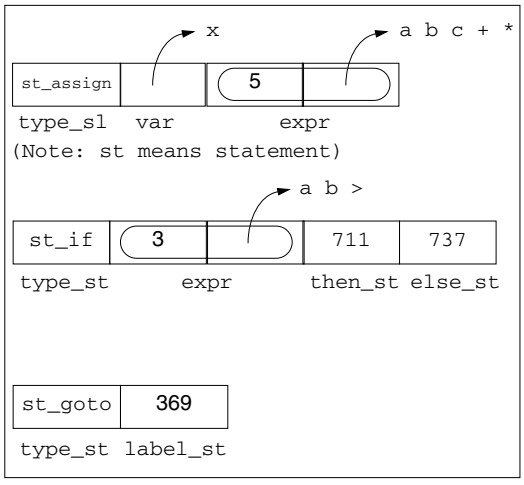
- It is not possible to completely regenerate the source code.

+ Can be stored compactly (each instruction to a byte plus address, where necessary).

+ Easy to write an interpreter for.

Examples of an interpreter in principle

Examples of internal form : (This is a mixture of prefix statement nodes with postfix expression code.)



- The code is in an array

```
type stmts = record
                stmt: type_st;
                info: { variant record }
             end;
var prog: array [1..max] of stmts;
```

The interpreter:

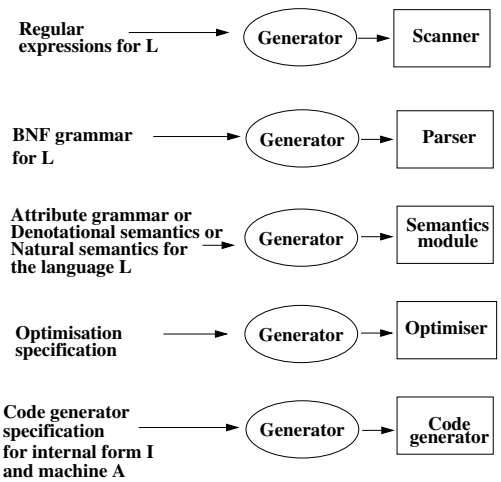
```
pc := 1;

while (not error) and (not done) do
begin

 case prog[pc].type_st of

   st_assign:
        begin
          put(prog[pc].var,
            evalpostfix(prog[pc].expr));
          pc := pc + 1;
        end;

   st_if:  if evalpostfix(prog[pc].expr)
             then pc := prog[pc].then_st
             else pc := prog[pc].else_st

   st_goto:   pc := prog[pc].label_st;

    ...
    ...

 end (* case *);

end (* while *);
```


Advantages and disadvantages of interpreting

- The interpreter plus the internal form usually take more space than object code + run-time system (e.g. LISP)
- The program executes more slowly, by a factor of 10 - 100
+ Execution starts quickly (command decoding)
+ Easy to implement new languages as there is no need for messy code generation or register allocation
+ Better possibilities for error tracing: change and test.
+ Easier to implement symbolic debugger (access to the symbol table)
+ Stepwise execution
+ Self-evaluation, i.e. evaluation in the language
  - Lisp: Explicit call of Eval
  - Mathematica: Automatic call of Eval on expressions

Compiler generators or CWS, Compiler Writing Systems

A CWS is a program which, given a description of the source language and a description of the target language (object code description), produces a compiler for the source language as output.
Different generators within CWS generate different phases of the compiler.



Some common and well-known compiler generation tools delivered with Unix:

- LEX – generates lexical analysers
- YACC – generates parsers
  - Compiler components that are not generated:
    - semantic analysis and intermediate code gen.
    - the optimisation phase
    - code generators
- In addition YACC produces parsers which are bad at error management

Cocktail – a CWS (Compiler Writing System) from Karlsruhe University suitable for applications where industrial quality is required

- Complete –
  generates all phases of the compiler
  (except possibly parts of the optimiser)
- Very fast compiler modules are generated
  (e.g. parsers are about 4-5 times faster than those from YACC)
- The system is available free of charge
  (But there is a maintained version which costs money)
- Is used
  (e.g. by PELAB, as well as other research groups and companies)
- Also commercial by Cocolab (www.cocolab.de)

"New" compiler generation systems

- The COSY system (COmpilation SYstem)

  Developed at Karlsruhe University, ACE (= the Associated Computer Experts company in Amsterdam), INRIA (in Paris), GMD (in Berlin)
- All compiler parts mostly back-end are generated
- Flexible phase-ordering for better optimisation
- "Parallel" optimisation
  (certain parts of the optimiser can run in parallel on a multiprocessor, e.g. a 4-processor Sparc)
- Structure definition language (SDL) allows access to and extensions of the intermediate form without recompiling and changing other compiler modules.
- Used e. g by PELAB, Ericsson, Phillips.

Research on compiler generation in PELAB

- Generation of efficient compilers from denotational semantics specifications, 1990

  (Mainly generation of semantic analysis and the intermediate code generation phase
- Generation of compilers for data-parallel languages from a special version of denotational semantics
- Generation of efficient compilers and interpreters from natural semantics specifications - the RML system.
  (To ftp, see RML at
  www.ida.liu.se/~pelab/projects.html)

  (About 180 000 times faster than TYPOL from the Centaur system developed at INRIA in Sophia-Antipolis)

Various formalisms to specify language semantics, which are suitable for compiler generation

- Attribute grammars

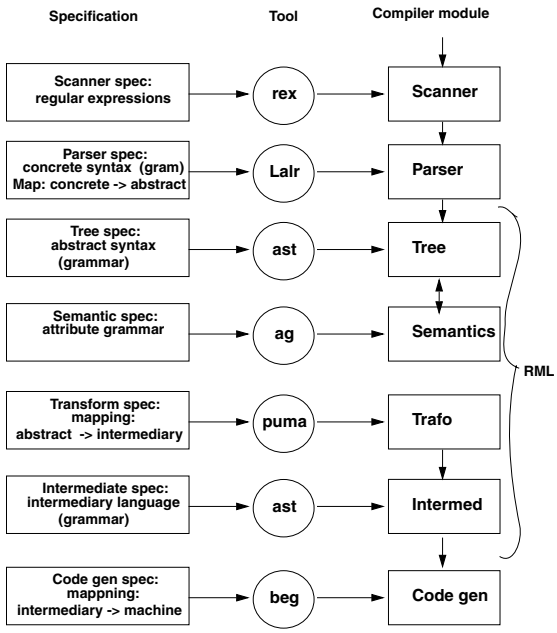
  (Efficient compilers, used industrially, require quite long specifications)
- Denotational semantics

  (Now used mostly for functional languages, problems with modularisation of the specification)
- Natural semantics

  (has recently become popular, and has just recently been implemented efficiently)

  (Easy to use, good abstraction power)

Structure of compilers from Cocktail



Scanner specification for Cocktail (regular expressions) (Generator part: REX)

minilax.scan:
(for the minilax language - subset of Pascal)

```
LOCAL{ char Word [256]; }

DEFAULT{
   MessageI ("illegal character", xxError, Attribute.Position, xx-
Character, TokenPtr);
}

EOF{
    if (yyStartState == Comment) Message ("unclosed comment", xx-
Error, Attribute.Position);
}

DEFINEdigit= {0-9} .
letter= {a-z A-Z} .

STARTComment

RULE
  "(*":- {yyStart (Comment);}
#Comment# "*)":- {yyStart (STD);}
#Comment# "*" | - {*\t\n} + :- {}

#STD# digit +: {(void) GetWord (Word);
   Attribute.IntegerConst.Integer = atoi (Word);
   return IntegerConst;}

#STD# digit * "." digit + (E {+\-} ? digit +) ?
: {(void) GetWord (Word);
   Attribute.RealConst.Real = atof (Word);
   return RealConst;}

INSERT RULES #STD#

#STD# letter (letter | digit) *
: {Attribute.Ident.Ident = MakeIdent (TokenPtr, TokenLength);
   return Ident;}
```


Parser specification for Cocktail (BNF gram.) (Generator part: LALR)

```
PARSER
...
Type= <
   Int= INTEGER .
   Real= REAL .
   Bool= BOOLEAN .
   Array= ARRAY '[' Lwb: IntegerConst '..' Upb: IntegerConst ']'
OF Type .
> .
...
Stats= <
   Stats1= Stat .
   Stats2= Stats ';' Stat .
> .
Stat= <
   Assign= Adr ':=' Expr .
   Call0= Ident .
   Call= Ident '(' Actuals ')' .
   If= IF Expr THEN Then: Stats ELSE Else: Stats 'END' .
   While= WHILE Expr DO Stats 'END' .
   Read= READ '(' Adr ')' .
   Write= WRITE '(' Expr ')' .
> .
...
Expr= <
   Less= Lop: Expr '<' Rop: Expr .
   Plus= Lop: Expr '+' Rop: Expr .
   Times= Lop: Expr '*' Rop: Expr .
   Not= NOT Expr .
   '()'= '(' Expr ')' .
   IConst= IntegerConst .
   RConst= RealConst .
   False= FALSE .
   True= TRUE .
   Adr= <
      Name= Ident .
      Index= Adr '[' Expr ']' .
   > .
> .
/* Some terminals (with attributes): */
IntegerConst: [Integer] { Integer:= 0; } .
RealConst: [Real : float ] { Real:= 0.0; } .

NB:You can give names to right sides and parts of right sides.
```


Semantic actions for constructing abstract syntax trees

```
MODULE Tree
/* import functions for tree construction*/
PARSER GLOBAL{
# include "Tree.h"
tTree  nInteger, nReal, nBoolean;
}

BEGIN{
   nInteger= mInteger();
   nReal= mReal();
   nBoolean= mBoolean();
}
/* attributes for tree construction*/
DECLARE
  Decls Decl Formals Formal Type Stats Stat Actuals
  Expr = [Tree: tTree] .

RULE/* tree construction =*/
/* mapping: concrete syntax -> abstract syntax*/

Real= { Tree := nReal;} .
Bool= { Tree := nBoolean;} .
Array= { Tree := mArray (Type:Tree, Lwb:Integer,
                         Upb:Integer, Lwb:Position);} .
Stats1= { Tree := {Stat:Tree->\Stat.Next = mNoStat ();
                   Tree = Stat:Tree;};} .
Stats2= { Tree := {Stat:Tree->\Stat.Next = Stats:Tree;
          Tree = Stat:Tree;};} .
Assign= { Tree := mAssign(NoTree, Adr:Tree, Expr:Tree,
                          ':=':Position);} .
Call0= { Tree := mCall (NoTree,
      mNoActual (Ident:Position), Ident:Ident, Ident:Position);} .
Call= { Tree := mCall (NoTree, ReverseTree
                (Actuals:Tree), Ident:Ident, Ident:Position); } .
If=   { Tree := mIf (NoTree, Expr:Tree,
               ReverseTree (Then:Tree),ReverseTree (Else:Tree));} .
While= { Tree := mWhile (NoTree, Expr:Tree,
                         ReverseTree (Stats:Tree));} .
} .
Plus=  { Tree := mBinary ('+':Position, Lop:Tree, Rop:Tree,
                          Plus);} .
Times= { Tree := mBinary ('*':Position, Lop:Tree, Rop:Tree,
                          Times);} .
```

Specification of abstract syntax (Generator part: AST)

minilax.cg: (is preprocessed and input to **AST**)

```
MODULE AbstractSyntax

TREE

EVAL Semantics

PROPERTY INPUT

RULE

MiniLax= Proc .
Decls= <
   NoDecl= .
   Decl= Next: Decls REV [Ident: tIdent] [Pos: tPosition] <
      Var= Type .
      Proc= Formals Decls Stats .
   >.
>.
Formals= <
   NoFormal= .
    Formal= Next: Formals REV [Ident: tIdent] [Pos: tPosition]
Type .
>.
Type= <
   Integer= .
   Real= .
   Boolean= .
   Array= Type OUT            [Lwb] [Upb] [Pos: tPosition] .
   Ref= Type OUT .
   NoType= .
   ErrorType= .
>.
Stats= <
   NoStat= .
   Stat= Next: Stats REV <
      Assign= Adr Expr                    [Pos: tPosition] .
      Call= Actuals       [Ident: tIdent] [Pos: tPosition] .
      If= Expr Then: Stats Else: Stats .
      While= Expr Stats .
      Read= Adr .
      Write= Expr .
   >.
>.
```


A partial BEG code generator specification for PLEX *) processor APZ

```
(* Part of a code generator spec for PLEX and machine APZ *)
(* Costs are not realistic. They are only guiding BEG. *)

%noonthefly (* No on the fly register allocation          *)
%test       (* Option for BEG to generate test output routines *)
%RegNameTable


CODE_GENERATOR_DESCRIPTION  Plex;
INTERMEDIATE_REPRESENTATION
NONTERMINALS Value;
OPERATORS
  Constant  ( v : int)                      -> Value;
  Plus                      Value + Value -> Value;
  Mult Value * Value * Value * Value * Value -> Value;
  Content                   Value         -> Value;
  Assign                    Value * Value;
  Enter ( signal : string ) ;
  Sub   (offset: int;
         length: int)       Value         -> Value;
  Simple_var    (name: string;
                 length: int)               -> Value;
  Str_var       (name: string;
                 length: int)               -> Value;
  TempVar      (name: string;
                 length: int)               -> Value;
  Matrix             Value * Value * Value -> Value;
  Reg (name: string)                        -> Value;

(* Is used to get a temporary register in Mult *)
  FreeReg                                   -> Value;
  Exit ;

REGISTERS
(*******)
  dr0, dr1, dr2, dr3, dr4, dr5, dr6, dr7,
  dr8, dr9, dr10, dr11, dr12, dr13, dr14, dr15,
  dr16, dr17, dr18, dr19, dr20, dr21, dr22, dr23,
  ar0, ar1, ar2, ar3,
  wr0, wr1, wr2, wr3, wr4, wr5, wr6, wr7,
  wr8, wr9, wr10, wr11, wr12, wr13, wr14, wr15,
  wr16, wr17, wr18, wr19, wr20, wr21, wr22, wr23,
  wr24, wr25, wr26, wr27, wr28, wr29,
  ir, pr0, pr1, cr, sr0, sr1;

NONTERMINALS
```



```
(**********)
  Const    COND_ATTRIBUTES      (v: int;
                                 length: int);
  DS       ADRMODE
           COND_ATTRIBUTES      (name: string;
                                 length: int);

  DSS      ADRMODE
           COND_ATTRIBUTES      (name: string;
                                 length: int;
                                 offset: int);

  Register  REGISTERS
(dr0, dr1, dr2, dr3, dr4, dr5, dr6, dr7,
 dr8, dr9, dr10, dr11, dr12, dr13, dr14, dr15,
 dr16, dr17, dr18, dr19, dr20, dr21, dr22, dr23,
 ar0, ar1, ar2, ar3,
 wr0, wr1, wr2, wr3, wr4, wr5, wr6, wr7,
 wr8, wr9, wr10, wr11, wr12, wr13, wr14, wr15,
 wr16, wr17, wr18, wr19, wr20, wr21, wr22, wr23,
 wr24, wr25, wr26, wr27, wr28, wr29,
 ir, pr0, pr1, cr, sr0, sr1)
            COND_ATTRIBUTES     (length: int);

(* ------------------------------------------- *)
(* Constant folding is not needed. (Is taken care of by CE.) *)
(* To calculate the length of the constant *)

  COST 0;
  EVAL {Const.v = Constant.v;
        Const.length = SizeofConst(Constant.v);};

(* Statements ---------------------------------- *)

RULE Enter;
  COST 10;
  EMIT {
.>      RECEIVE {s Enter.signal}
        };

RULE Exit;
  COST 10;
  EMIT {
.>      EP
        };

RULE Assign
        DS
        Register.a;
  COST 10;
```



```
  EMIT {
.>      WS {s DS.name}-{*a}
        };

RULE Assign
        DS
        Const;
  CONDITION {DS.length <= 8};
  COST 10;
  EMIT {
.>      WHC {s DS.name}-{i (Const.v & ((1<<DS.length)-1))}

(* Expressions --------------------------------- *)

RULE Register.r
  -> TempVar.v;
  COST 0;
  EVAL {v.length = r.length};
  TARGET r;

RULE Simple_var.v
  -> DS;
  COST 0;
  EVAL {DS.name = v.name;
        DS.length = v.length;};

RULE Matrix
        DS.a
        Register(pr0)
        Register(ir)
  -> DS.b;
  COST 0;
  EVAL {b.name = a.name;
        b.length = a.length;};

RULE Sub
        DS
  -> DSS;
  COST 0;
  EVAL {DSS.name = DS.name;
        DSS.length = Sub.length;
        DSS.offset = Sub.offset/Sub.length};
```

Various formalisms for describing a language

Scanner

- Reg. expressions → Automata theory → Efficient scanner

Parser

- CFG → Parsing theory → LR-parser (or other)

The above two parts are in all CWSs.

Semantic analyser

- Hand-written (syntax-driven translation) or,
- Attribute gram. → Attribute evaluator → Semantic analysis.

  The result is inefficient for large programs.

Code generation

- Often totally or partially hand-written.
- Description of code templates + some hand-writing. → code-generator generator → code generator

Error manager

- Included in respective parts.

Syntax error manager

- CFG → table inspection → error manager
- Language-independent error managers are available.

RML—Relational Meta Language

A compiler"middle-end" generation system and specification language based on Natural Semantics

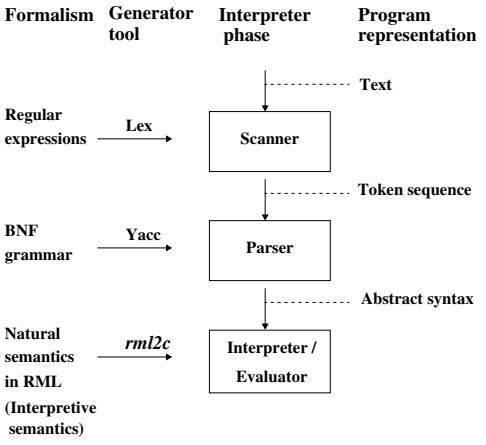
Goals

- Efficient code — comparable to hand-written compilers
- Simplicity — simple to learn and use
- Compatibility with "typical natural semantics" and with Standard ML

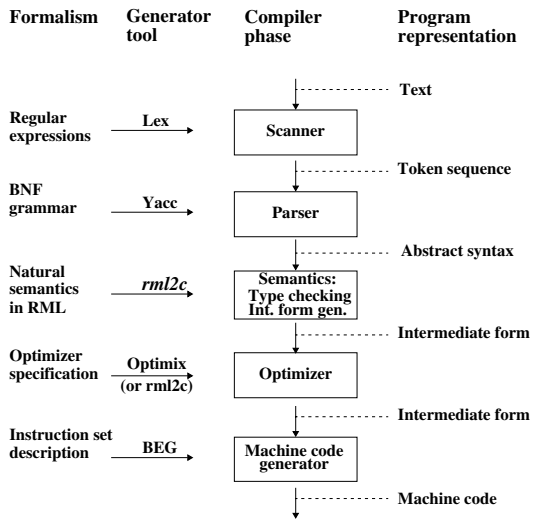
Properties

- Deterministic
- Separation of input and output arguments/results
- Statically strongly typed
- Polymorphic type inference
- Efficient compilation of pattern-matching

Example Use: Generating an interpreter implemented in C, using rml2C



Example Use: Generating a compiler implemented in C



RML Syntax

- Goal: Eliminate phletora of special symbols usually found in Natural Semantics specifications

  Software engineering viewpoint: identifiers are more readable in large specifications

- A Natural semantics rule:

$$\frac{H_1 \vdash T_1 : R_1 \;\; .. \;\; H_n \vdash T_n : R_n}{H \vdash T : R} \text{ if } \langle cond \rangle$$

- Typical RML rule:

```
rule  NameX(H1,T1) => R1  &
      ...
      NameY(Hn,Tn) => Rn  &
      <cond>
      ------------------------------
      RelationName(H,T) => R
```


Example: the Exp1 expression language

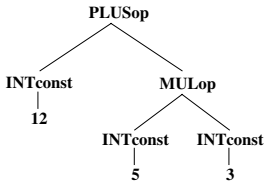
- Typical expressions

```
12 + 5*3

-5 * (10 - 4)
```

- Abstract syntax (defined in RML):

```
datatype Exp    =  INTconst of  int
                |  PLUSop   of  Exp * Exp
                |  SUBop    of  Exp * Exp
                |  MULop    of  Exp * Exp
                |  DIVop    of  Exp * Exp
                |  NEGop    of  Exp
```

- Abstract syntax tree of 12 + 5*3



Evaluator for Exp1

```
relation eval: Exp => int  =
```

Evaluation of an integer constant `ival` is the integer itself

```
  axiom eval( INTconst(ival) ) => ival
```

Evaluation of an addition node `PLUSop` is `v3`, if `v3` is the result of adding the evaluated results of its children `e1` and `e2`

Subtraction, multiplication, division operators have similar specifications. (we have removed division below)

```
  rule  eval(e1) => v1  &  eval(e2) => v2  &
        int_add(v1,v2) => v3
        ------------------------------------
        eval( PLUSop(e1,e2) ) => v3


  rule  eval(e1) => v1  &  eval(e2) => v2  &
        int_sub(v1,v2) => v3
        ------------------------------------
        eval( SUBop(e1,e2) ) => v3

  rule  eval(e1) => v1  &  eval(e2) => v2  &
        int_mul(v1,v2) => v3
        ------------------------------------
        eval( MULop(e1,e2) ) => v3

  rule  eval(e) => v1  &  int_neg(v1) => v2
        ------------------------------------
        eval( NEGop(e) ) => v2
end
```


Simple lookup in environments represented as linked lists

```
relation lookup: (Env,Ident) => Value  =
```

`lookup` returns the value associated with an identifier. If no association is present, `lookup` will fail.

Identifier `id` is found in the first pair of the list, and value is returned.

```
  rule  id = id2
        ------------------------------
        lookup((id2,value) :: _, id) => value
```

`id` is not found in the first pair of the list, and `lookup` will recursively search the rest of the list. If found, value is returned.

```
  rule  not id=id2  &  lookup(rest, id) => value
        ------------------------------------
        lookup((id2,_) :: rest, id)  => value
end
```

Translational semantics of the PAM language – abstract syntax to machine code

- Machine code:

LOAD	Load accumulator
STO	Store
ADD	Add
SUB	Subtract
MULT	Multiply
DIV	Divide
GET	Input a value
PUT	Output a value
J	Jump
JN	Jump on negative
JP	Jump on positive
JNZ	Jump on negative or zero
JPZ	Jump on positive or zero
JNP	Jump on negative or positive
LAB	Label (no operation)
HALT	Halt execution

Example:

- PAM code

```
read x,y;
while x<> 99 do
   ans :=  (x+1) - (y / 2)
  write ans;
  read x,y
end
```

- Translated machine code:

```
     GET   x          STO   T2
     GET   y          LOAD  T1
 L1  LAB              SUB   T2
     LOAD  x          STO   ans
     SUB   99         PUT   ans
     JZ    L2         GET   x
     LOAD  x          GET   y
     ADD   1          J     L1
     STO   T1       L2  LAB
     LOAD  y          HALT
     DIV   2
```

- Representation:

```
MGET(   I(x) )    MSTO(   T(2) )
MGET(  I(y) )     MLOAD(  T(1) )
MLABEL( L(1) )    MB(MSUB,T(2) )
MLOAD(  I(x) )    MSTO(   I(ans) )
MB(MSUB,N(99) )   MPUT(   I(ans) )
MJ(MJZ, L(2) )    MGET(   I(x) )
MLOAD(  I(x) )    MGET(   I(y) )
MB(MADD,N(1) )    MJMP(   L(1) )
MSTO(   T(1) )    MLABEL( L(2) )
MLOAD(  I(y) )    MHALT
MB(MDIV,N(2) )
```


Arithmetic expression translation

- Relation trans_expr

```
relation trans_expr:  Exp => Mcode list  =

  axiom  trans_expr(INT(v))    => [MLOAD( N(v))]
  axiom  trans_expr(IDENT(id)) => [MLOAD( I(id))]
  ....
```

- Code template for simple subtraction expression:

```
<code for expression e1>
MB(MSUB (e2))
```

and in assembly text form:

```
<code for expression e1>
SUB  e2
```

- RML rule for simple (expr1 binop expr2):

```
rule trans_expr(e1) => cod1  &
     trans_expr(e2) => [MLOAD(operand2)]  &
     trans_binop(binop) => opcode  &
       list_append(cod1, [MB(opcode,operand2)]) => cod3
     ----------------------------------
     trans_expr(BINARY(e1,binop,e2) => cod3
```


The complete trans_expr relation

```
relation trans_expr:  Exp => Mcode list  =
(* Evaluation of expressions in the current environment *)

  axiom  trans_expr(INT(v)) => [MLOAD( N(v))]     (* integer constant *)

  axiom  trans_expr(IDENT(id)) => [MLOAD( I(id))]    (* identifier id *)

(* Arith binop: simple case, expr2 is just an identifier or constant *)
  rule   trans_expr(e1) => cod1  &
         trans_expr(e2) => [MLOAD(operand2)]  &  (* expr2 simple *)
         trans_binop(binop) => opcode  &
         list_append(cod1, [MB(opcode,operand2)]) => cod3
         ----------------------------------    (* expr1 binop expr2 *)
         trans_expr(BINARY(e1,binop,e2) => cod3

(* Arith binop: general case, expr2 is a more complicated expr *)
  rule   trans_expr(e1) => cod1  &
         trans_expr(e2) => cod2  &
         trans_binop(binop) => opcode  &
         gentemp => t1  &
         gentemp => t2  &
         list_append6(
           cod1,                        (* code for expr1 *)
           [MSTO(t1)],                  (* store expr1 *)
           cod2,                        (* code for expr2 *)=
           [MSTO(t2)],                  (* store expr2 *)
           [MLOAD(t1)],                 (* load expr1 value into Acc *)
           [MB(opcode,t2)] ) => cod3    (* Do arith operation *)
         ----------------------------------    (* expr1 binop expr2 *)
         trans_expr(BINARY(e1,binop,e2)) => cod3

end (* trans_expr *)


and (*relation*) trans_binop:  BinOp => MBinOp  =
  axiom  trans_binop(PLUS) =>  MADD
  axiom  trans_binop(SUB)  =>  MSUB
  axiom  trans_binop(MUL)  =>  MMULT
  axiom  trans_binop(DIV)  =>  MDIV
end


and (*relation*) gentemp: () => MTemp  =

  rule   tick => no
         ----------
         gentemp => T(no)

end (* gentemp *)
```

Some applications of RML

- Small functional language with call-by-name semantics (mini-Freja, a subset of Haskell)

  Interpreter performance compared to Centaur/Typol:

#primes	Typol	RML	T/R
3	13s	0.0026s	5000
4	72s	0.0037s	19459
5	1130s	0.0063s	**179365**

- Almost full Pascal with some C features (Petrol)

  (specification size around 2000 lines)

  Generated compiler ran 39% faster than hand-written compiler for "similar" language

- Mini-ML including type inference

- Ongoing application work:
  - **Specification of Java 1.2**
  - **Specification of Modelica 1.2**

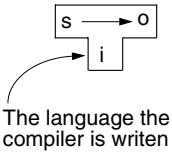
Portability and bootstrapping

A compiler is specified by 3 programming languages:

1. s = Source language (input)
2. o = Object language (output)
3. i = Implementation language

A compiler is denoted: $\overset{s\to o}{C_{\,i}}$ source out-lang. C impl-lang.

or as a T-diagram:



The language the compiler is writen in

How do you write a compiler from the beginning?

1. Use a *compiler generator*, or
2. By bootstrapping.

Bootstrapping:

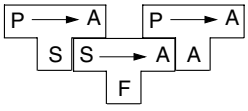
The methodology to get a compiler $\overset{x\to o}{C_{\,x}}$ i.e. a compiler implemented in the language it is to compile.

Example. How can we have a Pascal compiler written in Pascal which can run on machine A?
A only has assembly language, A, and Fortran.

Do as follows:

1. Write a compiler $\overset{S\to A}{C_{\,F}}$ where S = subset of Pascal. Compile it using the Fortran compiler.
2. Write a compiler for Pascal in S $\Rightarrow$ $\overset{P\to A}{C_{\,S}}$
3. Compile $\overset{P\to A}{C_{\,S}}$ with $\overset{S\to A}{C_{\,F}}$ which produces a new compiler $\overset{P\to A}{C_{\,A}}$

   As T-diagram:



4. Modify and clean up $\overset{P\to A}{C_{\,S}}$ using full Pascal which gives us $\overset{P\to A}{C_{\,P}}$ and gives the compiler a better structure.

This can be done in several steps:
Sub-Pascal $\Rightarrow$ Larger-Sub-Pascal $\Rightarrow$ .... $\Rightarrow$ Pascal

How do you move a compiler?

Assume we want to move the Pascal compiler which is now on machine A to machine B. On B we only have Fortran and the assembly language B.

We have : $\underset{C\ P}{P\to A}$ and $\underset{C\ A}{P\to A}$

We want: $\underset{C\ P}{P\to B}$ and $\underset{C\ B}{P\to B}$

Solution 1:

Work on machine B.

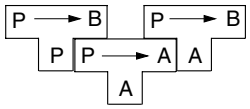
Bootstrapping: Start from the beginning with assembler B or C/Fortran.

Solution 2:

Work on machine A, taking the following steps:

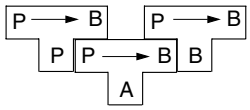
1. Rewrite the code generation for the Pascal compiler on A so that it generates code for B $\Rightarrow \underset{C\ P}{P\to B}$
2. Run $\underset{C\ P}{P\to B}$ through the old (i.e. $\underset{C\ A}{P\to A}$) so we get $\underset{C\ A}{P\to B}$ which is a cross-compiler (generates code for another machine).

As T-diagram:



3. Compile $\underset{C\ P}{P\to B}$ with $\underset{C\ A}{P\to B}$, which gives $\underset{C\ B}{P\to B}$

As T-diagram:



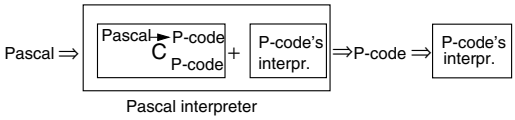
Example of a portable compiler: Pascal-P

The implementation of the Pascal-P compiler :

1. Pascal compiler which generates P-code, written in Pascal, i.e. $\underset{C\ \text{Pascal}}{\text{Pascal}\to\text{P-code}}$
2. Pascal compiler which generates P-code, written in P-code, i.e. $\underset{C\ \text{P-code}}{\text{Pascal}\to\text{P-code}}$
3. P-code's interpreter written in Pascal.

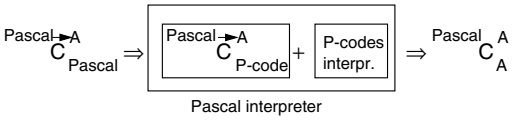
Assume we are going to move the compiler to a machine where only assembly language A exists.

1. Write a P-code interpreter in A.
2. Now we can use $\underset{C\ \text{P-code}}{P\to\text{P-code}}$ but it will run extremely slowly as it interprets on two levels:



Pascal interpreter

3. Modify $\underset{C\ \text{Pascal}}{\text{Pascal}\to\text{P-code}}$ to generate $\underset{C\ \text{Pascal}}{\text{Pascal}\to A}$ (e.g. by macro-expansion of P-code to A).
4. Now run $\underset{C\ \text{Pascal}}{\text{Pascal}\to A}$ through the Pascal interpreter in step 2 which gives us $\underset{C\ \text{P-cod}}{\text{Pascal}\to A}$
5. Run $\underset{C\ \text{Pascal}}{\text{Pascal}\to A}$ through $\underset{C\ \text{P-cod}}{\text{Pascal}\to A}$ using the P-code interpreter.



Pascal interpreter

NB: Steps 4 and 5 correspond to the compilation of $\underset{C\ \text{Pascal}}{\text{Pascal}\to A}$ by itself.

Performance criteria for a compiler

1. **The speed of the compiler** (as few passes as possible, preferably one pass, but goes against 2., 3. and 4.)
2. **Code quality**
3. **Portability**

   a) **Retargetability**
   A compiler that can easily be modified togenerate code for a new target machine.

   b) **Rehostability**
   A compiler that can easily be moved to and run on a new machine.

4. **Maintainability** (preferably modularisation, clear borders between the various passes).
5. **Error management**