Code Generation

Vertalerbouw **HC6**

VB HC6

Arend Rensink
kamer: Zilverling 5090
telefoon: 4862
email: rensink@cs.utwente.nl

Original design: Theo Ruys
University of Twente
Department of Computer Science
Formal Methods & Tools

What have you seen last time?

- Data representation
  - Representation of values for each source language type
- Expression evaluation
  - Register or stack machines
- Storage allocation
  - Global, local and heap
- Routines
  - Calling conventions
  - Parameter passing

Mededelingen

- Practicum week 3
  - Foutje in Calc.g, in productieregel van **program**:

```
program : declarations statements EOF!
        -> ^(PROGRAM declarations? statements)
```

If a rule might generate an empty AST (a null Object), you have to specify this explicitly.

In de oorspronkelijke Calc.g ontbreekt deze "?".

  - Beschrijving van "Rewind" (voor 3.1.9) is niet volledig.
    - De methoden **input.index** en **input.rewind** werken alleen op een **BufferedTreeNodeStream** object (en niet op een 'gewone' **CommonTreeNodeStream**).
    - Oplossing: vervang in **Calc.java** het **CommonTreeNodeStream** object voor de **CalcInterpreter** door een **BufferedTreeNodeStream** object.

needed for do-while

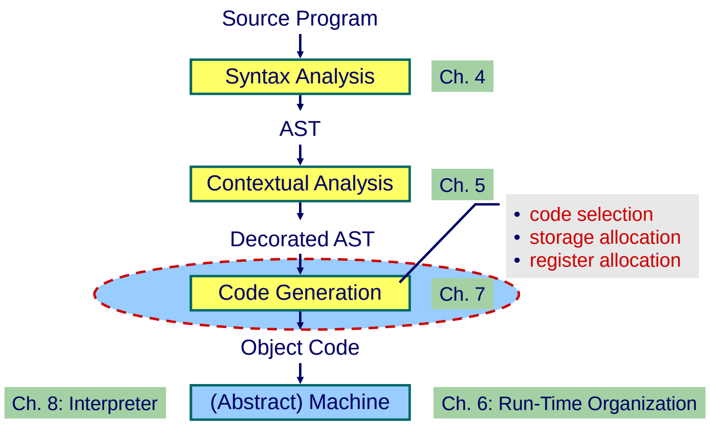
*In deze "Rewind" paragraaf dient **CommonTree** overal vervangen te worden door **BufferedTree**.*

De uitvoer van easter.calc (opg. 3.1.10) is ook incorrect. Dit moeten de jaren 2004-2013 zijn.

Overview of Lecture 6

- Ch 7 – Code Generation

Compiler Phases



Code Generation

- In week 3 (laboratory) we have seen that we can walk the AST to interpret the source program directly.
- A code generator also walks the AST but generates object code instructions, which can be
  - directly written to a file/standard output
    laboratory week 4
  - stored in a temporary array of instructions and written to a file after walking the complete AST

The language Calc has a monolithic block structure; all variables are global and can be allocated on the global stack (relative to **SB**).

Code Selection

- A compiler translates a program from a high-level language into an equivalent program in a low-level language.
  - Source and target program must be semantically equivalent.

```
let
   var x: integer;
   var y: integer
in begin
   y := 2;
   x := 7;
   printint(y);
   printint(x)
end
```

```
PUSH      2
LOADL     2
STORE(1)  1[SB]
LOADL     7
STORE(1)  0[SB]
LOAD(1)   1[SB]
CALL      putint
LOAD(1)   0[SB]
CALL      putint
POP       2
HALT
```

Code generation is concerned with the semantics of the language.

Code Templates (1)

- The translation of source language to target language is defined inductively over the (abstract) syntax of the language.
  Given a phrase of the source language, we specify the sequence of corresponding target code instructions using the translation of its sub-phrases.
- Code functions:

Instruction* is a sequence of target instructions.

run	:	Program → Instruction*
execute	:	Command → Instruction*
evaluate	:	Expression → Instruction*
fetch	:	V-name → Instruction*
assign	:	V-name → Instruction*
elaborate	:	Declaration → Instruction*

Naturally the code functions are dictated by the *semantics* of the language

Code Templates (2)

Summary of code functions for Triangle and TAM

class	code function	effect of the generated code
Program	***run P***	Run the program P and then halt, starting and finishing with an empty stack.
Command	***execute C***	Execute the command C, possibly updating variables, but neither expanding nor contracting the stack.
Expression	***evaluate E***	Evaluate the expression E, pushing its result on the stack top, but having no other effects.
V-name	***fetch V***	Push the value of the constant or variable named V on the stack.
V-name	***assign V***	Pop a value from the stack top, and store it in the variable named V.
Declaration	***elaborate D***	Elaborate the declaration D, expanding the stack to make space for any constants and variables declared therein.

Code Templates (3)

- Sequential Command: $C_1 ; C_2$
  - Semantics: the sequential command $C_1 ; C_2$ is executed as follows: first C_1 is executed then C_2 is executed.
  - ***execute*** $[C_1 ; C_2]$

    = ***execute*** $[C_1]$
      execute $[C_2]$

    Code template for $C_1 ; C_2$: the code to execute $C_1 ; C_2$ consists of the code to execute C_1 followed by the code to execute C_2.

- Assignment Command: *V* **:=** *E*
  - Semantics: the expression E is evaluated to yield a value: the variable identified by V is updated with this value.
  - ***execute*** [*V* **:=** *E*]

    = ***evaluate*** [*E*]
      assign [*V*]

    = `STORE a` where **a** is the address of variable **V**.

Code Templates (4)

- if-then-else Command: **if** *E* **then** C_1 **else** C_2
  - Semantics: the expression E is evaluated; if its value is true, then C_1 is executed; if its value is false, then C_2 is executed.
  - ***execute*** [**if** *E* **then** C_1 **else** C_2]

```
=          evaluate [E]
           JUMPIF(0)   Lelse
           execute [C1]
           JUMP        Lfi
   Lelse:  execute [C2]
   Lfi:
```

A code template specifies the object code to which a phrase is translated, in terms of the object code to which its subphrases are translated.

Code Templates (5)

- while Command: **while** *E* **do** *C*
  - Semantics: the expression *E* is evaluated; if its value is true, then *C* is executed, and then the while-command is executed again; if its value is false, then execution of the while-command is completed.
  - ***execute*** [**while** *E* **do** *C*]

```
=  Lwhile: evaluate [E]
           JUMPIF(0)   Lend
           execute [C]
           JUMP        Lwhile
   Lend:
```

Watt & Brown use a slightly different code template.

- let-in Command: **let** *D* **in** *C*
  - Semantics: the declaration *D* is elaborated; then *C* is executed, in the environment of the block command overlaid by the bindings produced by *D*.
  - ***execute*** [**let** *D* **in** *C*]

```
=          elaborate [D]
           execute [C]
           POP(0)      s
```

only if s > 0 where s = amount of storage allocated by D.

3

Code Templates (6)

execute [*while E do C*]
```
  =  Lwhile:  execute [E]
              JUMPIF(0) Lend
              execute [C]
              JUMP      Lwhile
     Lend:
```

- Example: `while i>0 do i:=i-2`

execute [`while i>0 do i:=i-2`] =

evaluate [`i>0`]
```
50: LOAD       i
51: LOADL      0
52: CALL       gt
```
JUMPIF
```
53: JUMPIF(0)  59
```
execute [`i:=i-2`]
```
54: LOAD       i
55: LOADL      2
56: CALL       sub
57: STORE      i
```
JUMP
```
58: JUMP       50
59:
```

- There often several ways to generate target code for an expression. Sometimes we can get more efficient code for special cases.
  - Example: *evaluate* [`i+1`]

general template:
```
LOAD   i
LOADL  1
CALL   add
```

special case:
```
LOAD  i
CALL  succ
```

More efficient code for the special case "**+1**".

Often used for inlining constant denotations.

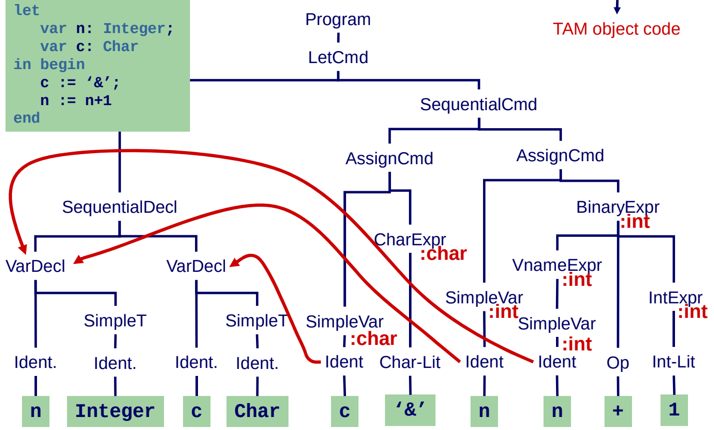
Overview of Lecture 6

- Ch 7 – Code Generation
  - 7.1 Code selection
  - 7.2 A code generation algorithm
  - 7.3 Constants and variables
  - HC7 7.4 Procedures and functions
  - 7.5 Case study – Triangle compiler

Code Generator: input

Decorated AST → Code Generator → TAM object code

```
let
   var n: Integer;
   var c: Char
in begin
   c := '&';
   n := n+1
end
```



TAM object code (1)

`package TAM;`

```
public class Instruction {
    public int op;    // op-code (LOADop, LOADAop, etc.)
    public int n;     // length field
    public int r;     // register field (SBr, LBr, L1r, etc.)
    public int d;     // operand field

    public static final byte // op-codes (Table C.2)
        LOADop = 0, LOADAop = 1, ...

    public static final byte // register numbers (Table C.1)
        CBr = 0, CTr = 1, PBr = 2, PTr = 3, ...;
}
public class Machine {
    private static Instruction[] code = new Instruction[1024];
    private short nextInstrAddr = 0;
}
public class Interpreter {
    ...
}
```

W&B (p. 260) uses **byte**-variables. **TAM.Instruction.java**, however, uses **int**-variables.

You now know that **enum** is superior

An implementation of the Triangle Abstract Machine.

TAM object code (2)

`package Triangle.CodeGenerator;`

Again, W&B specifies **byte**s here, but the Triangle source code uses **int**-variables.

```
public class Encoder implements Visitor {
    /** Append an instruction to the object program. */
    private void emit(int op, int n, int r, int d) {
        Instruction nextInstr = new Instruction();
        if (n > 255) {
            reporter.reportRestriction(
                "length of operand can't exceed 255 words");
            n = 255; // to allow code generation to continue
        }
        nextInstr.op = op;
        nextInstr.n  = n;
        nextInstr.r  = r;
        nextInstr.d  = d;
        if (nextInstrAddr == Machine.PB)
            reporter.reportRestriction(
                "too many instructions for code segment");
        else {
            Machine.code[nextInstrAddr] = nextInstr;
            nextInstrAddr = nextInstrAddr + 1;
        }
    }
    private short nextInstrAddr = 0;
}
```

Address (within `Machine.code`) of the next instruction.

Recall: Triangle Grammar

HC3

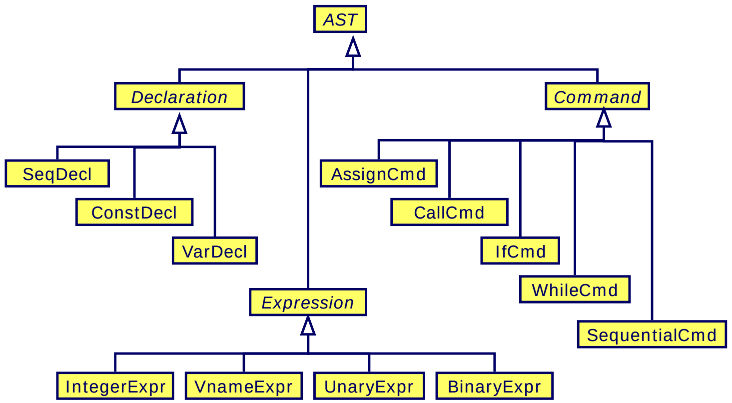
Grammar for Mini-Triangle's abstract syntax.

Program	::=	Command	Program
Command	::=	Command **;** Command	SequentialCmd
	|	V-name **:=** Expression	AssignCmd
	|	Identifier **(** Expression **)**	CallCmd
	|	**if** Expression **then** Command **else** Command	IfCmd
	|	**while** Expression **do** Command	WhileCmd
	|	**let** Declaration **in** Command	LetCmd
Expression	::=	Integer-Literal	IntegerExpr
	|	V-name	VnameExpr
	|	Operator Expression	UnaryExpr
	|	Expression Operator Expression	BinaryExpr
V-name	::=	Identifier	SimpleVname
Declaration	::=	Declaration **;** Declaration	SeqDecl
	|	**const** Identifier ~ Expression	ConstDecl
	|	**var** Identifier **:** Type-denoter	VarDecl
Type-denoter	::=	Identifier	SimpleTypeDen

AST nodes of Mini-Triangle

Recall: Triangle AST Node Classes

HC3



Recall: Visitor pattern

HC3

```
public class XYZ extends ... {
    Object visit(Visitor v, Object arg) {
        return v.visitXYZ(this, arg);
    }
}
```

```
public interface Visitor {
    public Object visitProgram
            (Program prog, Object arg);
    ...
    public Object visitAssignCmd
            (AssignCmd cmd, Object arg);
    public Object visitSequentialCmd
            (SequentialCmd cmd, Object arg);
    ...
    public Object visitVnameExpression
            (VnameExpression e, Object arg);
    public Object visitBinaryExpression
            (BinaryExpression e, Object arg);
    ...
}
```

The Visitor interface defines **visitXYZ** methods for all AST node types

```
    public Object visitXYZ
            (XYZ x, Object arg);
```

Encoder (1)

When using a compiler generator (like ANTLR), one only has to specify the code templates (in tree parser): the generator will generate the "visiting" methods.

```
public class Encoder implements Visitor {
   public Object visitProgram(Program prog, Object arg ) {
      prog.C.visit(this,arg);
      emit(Machine.HALTop, 0, 0, 0);
      return null;
   }
   ...
}
```

Code Generator as Visitor

phrase class	*visitor method*	*behaviour of the visitor method*
Program	`visitProgram`	generate code as specified by ***run*** [*P*]
Command	`visit..Cmd`	generate code as specified by ***execute*** [*C*]
Expression	`visit..Expr`	generate code as specified by ***evaluate*** [*E*]
V-name	`visit..Vname`	return "entity description" for the visited variable or constant name (i.e. use the "decoration").
Declaration	`visit..Decl`	generate code as specified by ***elaborate*** [*D*]
Type-Den	`visit..TypeDen`	return the size of the type

Encoder (2)

- For variables we use two distinct code generation methods: fetch and assign.

  These two methods deal with the scope information of the variables.

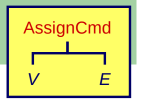
```
public class Encoder implements Visitor {
   ...
   public void encodeFetch(Vname name) {
      // as specified by fetch code template ...
   }
}

   public void encodeAssign(Vname name) {
      // as specified by assign code template ...
   }
}
```

These methods are not implemented as visitor methods but as separate methods of the class **`Encoder`**.

Encoder (3)

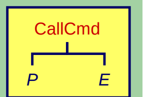
execute [*V* := *E*] = ***evaluate*** [*E*]
assign [*V*]

```
public Object visitAssignCmd(AssignCmd cmd, Object arg) {
   cmd.E.visit(this, arg);
   encodeAssign(cmd.V);
}
```



execute [*P*(*E*)] = ***evaluate*** [*E*]
CALL p

```
public Object visitCallCmd(CallCmd cmd, Object arg) {
   cmd.E.visit(this, arg);
   short p = address of primitive routine for name cmd.P
   emit(Instruction.CALLop,
         Instruction.SBr,
         Instruction.PBr, p);
   return null;
}
```

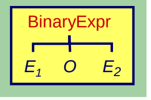


Calling user-defined procedures is much more complex due to parameter passing and scoping.

Encoder (4)

evaluate [E_1 *op* E_2] = ***evaluate*** [E_1]
evaluate [E_2]
CALL p

```
public Object visitBinaryExpression(
                        BinaryExpression expr, Object arg) {
   expr.E1.visit(this, arg);
   expr.E2.visit(this, arg);
   short p = address for expr.O operation
   emit(Instruction.CALLop,
          Instruction.SBr,
          Instruction.PBr, p);
   return null;
}
```



- Visiting methods for LetCmd, IfCmd, WhileCmd are more complex:
  - LetCmd involves scope information
  - IfCmd and WhileCmd are complicated due to jumps

Encoder (5)

```
execute [while E do C] =  Lwhile:  evaluate [E]
                                   JUMPIF(0)   Lend
                                   execute [C]
                                   JUMP        Lwhile
                          Lend:
```

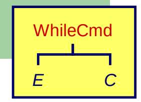
- Backwards jumps are easy: the "address" of the target has already been generated and is known.
- Forward jumps are harder: when the jump is generated the target is not yet generated so its address is not (yet) known.
- Solution: backpatching
  1. Emit jump with "dummy" address (e.g. simply 0).
  2. Remember the address where the jump instruction occurred.
  3. When the target label is reached, go back and patch the jump instruction.

Encoder (6)

```
execute [while E do C] =  Lwhile:  evaluate [E]
                                   JUMPIF(0)   Lend
                                   execute [C]
                                   JUMP        Lwhile
                          Lend:
```

```
public Object visitWhileCmd(WhileCmd cmd, Object arg) {
    short lwhile = nextInstrAddr;
    cmd.E.visit(this, arg);
    short jump2end = nextInstrAddr;
    emit(Instruction.JUMPIFop, 0, Instruction.CBr, 0);
    cmd.C.visit(this, arg);
    emit(Instruction.JUMPop, 0, Instruction.CBr, lwhile);
    short lend = nextInstrAddr;
    code[jump2end].d = lend;
}
```

backpatching



Overview of Lecture 6

- Ch 7 – Code Generation
  - 7.1 Code selection
  - 7.2 A code generation algorithm
  - 7.3 Constants and variables
  - HC7 7.4 Procedures and functions
  - 7.5 Case study – Triangle compiler

Constants and Variables (1)

- In Mini-Triangle, the LetCmd is where declarations appear.

```
execute [let D in C] =  elaborate [D]
                        execute [C]
                        POP(0)      s
```

only if s > 0 where s = amount of storage allocated by D.

- How to "elaborate a declaration"?
  - Variables (and unknown constants) are given a memory location relative to SB.
  - When a scope is closed, the locations of the old scope can be popped.

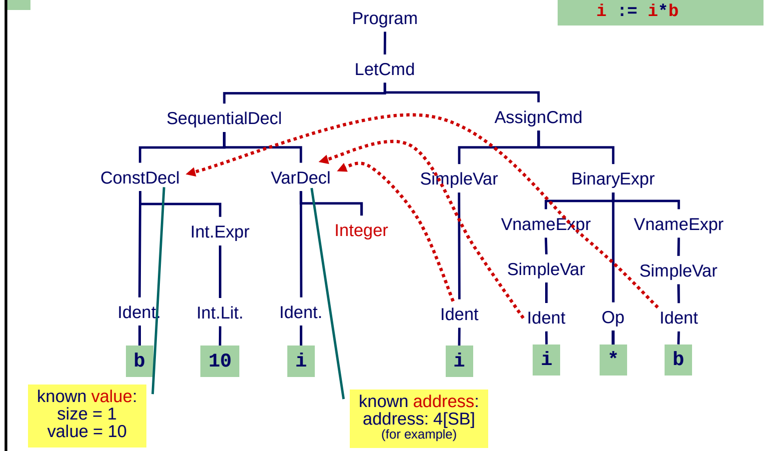
Remember: Mini-Triangle has a flat memory-model (no user-defined procedures).

```
fetch [V]   =  LOAD(1)  d[SB]
assign [V]  =  STORE(1) d[SB]
```

where d is the address of V relative to SB

Constants and Variables (2)

```
let
   const b ~ 10;
   var i: Integer
in
   i := i*b
```



Constants and Variables (3)

- Known value and known address:

```
let
   const b ~ 10;
   var i: Integer
in
   i := i*b
```

```
PUSH      1       ; room for x
LOAD(1)   4[SB]   ; load i
LOADL     10
call      mult
STORE(1)  4[SB]   ; store i
POP(0)    1
```

- Unknown value and known address:

```
let
   var x: Integer
in let
      const y ~ 365 + x
   in putint(y)
```

known address:
address = 5

unknown value:
size = 1
address = 6

```
PUSH      1       ; room for x
PUSH      1       ; room for y
LOADL     365
LOAD(1)   5[SB]   ; load x
CALL      add     ; 365+x
STORE(1)  6[SB]   ; y ~ 365+x
LOAD(1)   6[SB]
CALL      putint
POP(0)    1
POP(0)    1
```

Not really needed in this case

Constants and Variables (4)

- Code generator: declarations and applied occurrences:
  - When a declaration of identifier id is encountered, the code generator binds id to a newly created *entity description*.
    - Known value: record the value + its size
    - Known address: record the address + reserve space
  - When an applied occurrence of identifier id is encountered, the code generator consults the entity description bound to id, and translates the applied occurrence w.r.t. the entity.

known value	const declaration using a literal
unknown value	const declaration using an expression
known address	variable declaration
unknown address	argument address bound to a var-parameter

Constants and Variables (5)

```
public abstract class RuntimeEntity {
   public short size;
   ...
}
public class KnownValue extends RuntimeEntity {
   public short value;
   ...
}
public class UnknownValue extends RuntimeEntity {
   public short address;
   ...
}
public class KnownAddress extends RuntimeEntity {
   public short address;
   ...
}

public abstract class AST {
   public RuntimeEntity entity;
   ...
}
```

Mostly used within Declaration.

Static Storage Allocation (1)

- Example: global variables

```
let
   var a: Integer;
   var b: Boolean;
   var c: Integer
in begin
   ...
end
```

var	size	address
a	1	[0]SB
b	1	[1]SB
c	1	[2]SB

- Example: nested blocks

```
let var a: Integer
in  begin
      ...
    let var b: Boolean;
        var c: Integer
    in ...

    let var d: Integer
    in ...
end
```

var	size	address
a	1	[0]SB
b	1	[1]SB
c	1	[2]SB
d	1	[1]SB

Note that variable **d** "reuses" the location of **b** of the previous scope.

Static Storage Allocation (2)

- The code generator must keep track of how much storage has been allocated at each point in the source program.
  - We use the extra argument `Object arg` to the visiting methods to pass the *current* amount of storage in use.
  - Furthermore, we let a visiting method return an `Object` with the *extra* amount of storage it needed.
  - We encode both numbers in a `Short`-object
  - (Recall: generic visitor to avoid type casting)

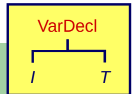
```
public Object visitXYZ(XYZ xyz, Object arg)
```

if not null, a `Short`-object with the *extra* storage needed.

`Short`-object with the *current* amount storage so far.

Static Storage Allocation (3)

elaborate [**var** *I* : *T*] = `PUSH s` *where s = size of T*

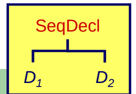



```
public Short visitVarDecl(VarDecl decl, Short arg) {
   short s  = decl.T.visit(this, null);
   decl.entity = new KnownAddress(s, arg);
   emit(Instruction.PUSHop, 0, 0, s);
   return s;
}
```

Next free address.

Remember the size and address of the variable.

elaborate [D_1 ; D_2] = ***elaborate*** [D_1] ***elaborate*** [D_2]

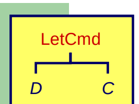



```
public Short visitSeqDecl(SeqDecl decl, Short arg) {
   short s1 = decl.D1.visit(this, arg);
   short s2 = decl.D2.visit(this, arg+s1);
   return s1+s2;
}
```


Static Storage Allocation (4)

execute [**let** *D* **in** *C*] = ***elaborate*** [*D*]
execute [*C*]
`POP(0)` s

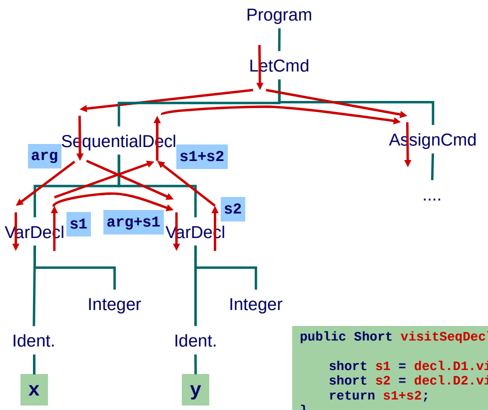
only if s > 0 where s = amount of storage allocated by D.




```
public Short visitLetCmd(LetCmd cmd, Short arg) {
   short s  = cmd.D.visit(this, arg);
   short r  = cmd.C.visit(this, arg+s);
   if (s > 0)
      emit(Instruction.POPop, 0, 0, s);
   return r;
}
```

Static Storage Allocation (5)

```
let
    var x: Integer;
    var y: Integer
in
    x := y
```




```
public Short visitSeqDecl(SeqDecl decl,
                          Short arg) {
    short s1 = decl.D1.visit(this, arg);
    short s2 = decl.D2.visit(this, arg+s1);
    return s1+s2;
}
```


Static Storage Allocation (6)

fetch [*I*] =	LOADL	v	*where v = value bound to I*
fetch [*I*] =	LOAD(s)	d[SB]	*where d is address bound to I and s = size(type of I)*

```
public Object encodeFetch(Vname name, short s) {
    RuntimeEntity entity =
        (RuntimeEntity) name.getEntity();
    if (entity instanceof KnownValue) {
        short v = ((KnownValue)entity).value;
        emit(Instruction.LOADLop, 0, 0, v);
    } else {
        short d = (entity instanceof UnknownValue) ?
            ((UnknownValue)entity).address :
            ((KnownAddress)entity).address;
        emit(Instruction.LOADop, s, Instruction.SBr, d);
    }
}
```

W&B uses Vname.visit here, but this conflicts with choice of Short return type

Alternative: let every **RuntimeEntity**-object fetch itself

What have you seen today?

- Code templates
  - Define conceptual translation
  - With the help of code functions
- Code generation algorithm
  - Visitor pattern
- Allocation for constants and variables
  - Maintain count of allocated space