CSc 453

Compilers and Systems Software

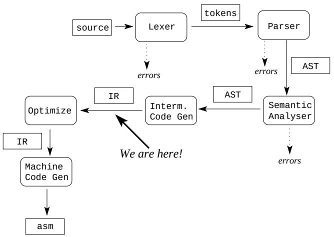
13 : Intermediate Code I

Department of Computer Science
University of Arizona

collberg@gmail.com

Introduction

Compiler Phases



Intermediate Representations

- Some compilers use the AST as the only intermediate representation. Optimizations (code improvements) are performed directly on the AST, and machine code is generated directly from the AST.
- The AST is OK for machine-independent optimizations, such as **inlining** (replacing a procedure call with the called procedure's code).
- The AST is a bit too high-level for machine code generation and machine-dependent optimizations.

Intermediate Representations. . .

- For this reason, some compilers generate a lower level (simpler, closer to machine code) representation from the AST. This representation is used during code generation and code optimization.

Intermediate Code

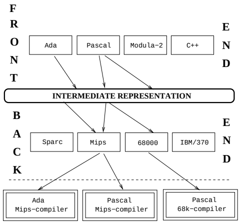
Advantages of:

1. Fitting many front-ends to many back-ends,
2. Different development teams for front- and back-end,
3. Debugging is simplified,
4. Portable optimization.

Requirements:

1. Architecture independent,
2. Language independent,
3. Easy to generate,
4. Easy to optimize,
5. Easy to produce machine code from.

Mix-and-Match Compilers



UNCOL

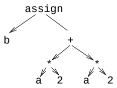
- A representation which is both architecture and language independent is known as an **UNCOL**, a **Universal Compiler Oriented Language**.
- UNCOL is the **holy grail** of compiler design – many have search for it, but no-one has found it. Problems:
  1. Programming language semantics differ from one language to another,
  2. Machine architectures differ.

Intermediate Code. . .

- There are several different types of intermediate representations:
  1. Tree-Based.
  2. Graph-Based.
  3. Tuple-Based.
  4. Linear representations.
- All representations contain the same information. Some are easier to generate, some are easy to generate simple machine code from, some are easy to generate **good** code from.
- **IR** — Intermediate Representation.

Postfix Notation

Postfix Notation



Infix: `b :=` $(a * 2) + (a * 2)$
Postfix: `b a 2 * a 2 * + :=`

- Postfix notation is a parenthesis free notation for arithmetic expression. It is essentially a linearized representation of an abstract syntax tree.
- In postfix notation an operator appears **after** its operands.
- Very simple to generate, very compact, easy to generate straight-forward machine code from, difficult to generate **good** machine code from.

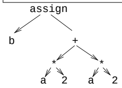
Tree & DAG Representations

Tree & DAG Representation

- Trees make good intermediate representations. We can represent the program as a sequence of **expression trees**. Each assignment, procedure call, or jump becomes one individual tree in the forest.
- **Common Subexpression Elimination** (CSE): Even if the same (sub-) expression appears more than once in a procedure, we should only compute its value **once**, and save the result for future reference.
- One way of doing this is to build a **graph** representation, rather than a tree. In the following slides we see how the expression $a * 2$ gets two subtrees in the tree representation and one subtree in the DAG representation.

Tree & DAG Representation. . .

$\boxed{\text{b} := (a * 2) + (a * 2)}$

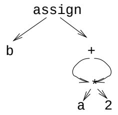


Linearized Tree

Nr	Op	Arg$_1$	Arg$_2$
1	**ident**	a	
2	**int**	2	
3	**mul**	1	2
4	**ident**	a	
5	**int**	2	
6	**mul**	4	5
7	**add**	3	6
8	**ident**	b	
9	**assign**	8	7

Tree & DAG Representation. . .

$\boxed{\text{b} := (a * 2) + (a * 2)}$

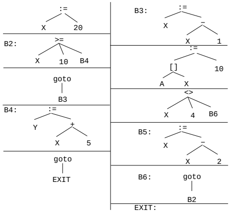


Linearized Dag

Nr	Op	Arg$_1$	Arg$_2$
1	**ident**	a	
2	**int**	2	
3	**mul**	1	2
4	**add**	3	3
5	**ident**	b	
6	**assign**	5	4

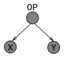
Sequence of Expression Trees

```
X := 20;
WHILE X < 10 DO
   X := X-1;
   A[X] := 10;
   IF X = 4 THEN
      X := X - 2;
   ENDIF;
ENDDO;
Y := X + 5;
```



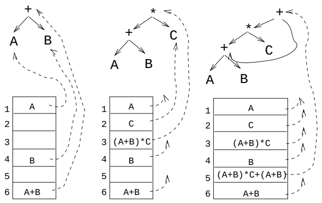
Building DAGs. . .

- Repeatedly add subtrees to build DAG. Only add subtrees not already in DAG. Store subtrees in a hash table. This is the **value-number** algorithm.
- For every insertion of a subtree, check if $(X \text{ OP } Y) \in$ DAG.



```
PROCEDURE InsertNode (
   OP : Operator; L, R :  Node) :  Node;
BEGIN
   V := hashfunc (OP, L, R);
   N := HashTab.Lookup (V, OP, L, R);
   IF N = NullNode THEN
      N := NewNode (OP, L, R);
      HashTab.Insert (V, N);
   END;
   RETURN N;
```

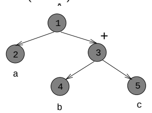
Building DAGs – Example



Building DAGs

- From an expression/expression tree such as the one on the left we might generate the machine code (for some fictitious architecture) on the right:

$a * (b + c)$



```
LOAD     b, r0
LOAD     c, r1
ADD      r0, r1, r2
LOAD     a, r3
MUL      r2, r3, r4
```

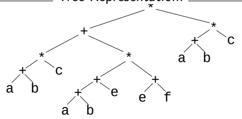
- Can we generate better code from a DAG than a tree?

Building DAGs. . .

Example Expression:

$$[(a + b) * c + \{(a + b) + e\} * (e + f)] * [(a + b) * c]$$

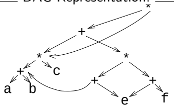
Tree Representation:



Building DAGs. . .

$$[(a+b)*c+\{(a+b)+e\}*(e+f)]*[(a+b)*c]$$

DAG Representation:



Building DAGs. . .

- Generating machine code from the tree yields 21 instructions.

Code from Tree

```
LOAD  a, r0         ; a            ADD   r2, r0, r4  ; (a+b)+e
LOAD  b, r1         ; b            LOAD  f, r0       ; f
ADD   r0, r1, r2    ; a+b          LOAD  e, r1       ; e
LOAD  c, r0         ; c            ADD   r0, r1, r0  ; f+e
MUL   r0, r2, r3    ; (a+b)*c      MUL   r4, r0, r4  ; {(a+b)+e}*
LOAD  a, r0         ; a                              ; (e+f)
LOAD  b, r1         ; b            ADD   r3, r4, r4
ADD   r0, r1, r2    ; a+b          ; Code for (a+b)*c into r3
LOAD  e, r0         ; e            MUL   r4, r3, r0
```

Building DAGs. . .

- Generating machine code from the DAG yields only 12 instructions.

Code from DAG

```
                                 LOAD  e, [r4]        ; e
LOAD  a, r0         ; a          ADD   r4, r2, r1     ; (a+b) -
LOAD  b, r1         ; b          LOAD  f, r0          ; f
ADD   r0, r1, [r2]  ; a+b        ADD   r0, r4, r0     ; f+e
LOAD  c, r0         ; c          MUL   r1, r0, r0
MUL   r0, r2, [r3]  ; (a+b)*c    ADD   r0, r3, r0
                                 MUL   r0, r3, r0
```

Tuple Codes

Three-Address Code

- Another common representation is **three-address code**. It is akin to **assembly code**, but uses an infinite number of **temporaries** (registers) to store the results of operations.
- There are three common realizations of three-address code: **quadruples**, **triples** and **indirect triples**.

Types of 3-Addr Statements:

$\boxed{x := y \text{ op } z}$	Binary arithmetic or logical operation. Example: `Mul`, `And`.
$\boxed{x := \text{op } y}$	Unary arithmetic, conversion, or logical operation. Example: `Abs`, `UnaryMinus`, `Float`.
$\boxed{x := y}$	Copy statement.

Three-Address Code...

$\boxed{\text{goto L}}$	Unconditional jump.
$\boxed{\text{if x } \texttt{relop} \text{ y goto L}}$	Conditional jump. `relop` is one of `<`,`>`,`<=`, etc. If `x relop y` evaluates to `True`, then jump to label L. Otherwise continue with the next tuple.
$\boxed{\text{param X}}$; $\boxed{\text{call P, } n}$	Make X the next parameter; make a procedure call to `P` with *n* parameters.
$\boxed{x := y[i]}$	Indexed assignment. Set `x` to the value in the location `i` memory units beyond `y`.

Three-Address Code...

$\boxed{x := \text{ADDR}(y)}$	Address assignment. Set `x` to the address of `y`.
$\boxed{x := \text{IND}(y)}$	Indirect assignment. Set `x` to the value stored at the address in `y`.
$\boxed{\text{IND}(x) := y}$	Indirect assignment. Set the memory location pointed to by `x` to the value held by `y`.

Three-Address Code...

- Many three-address statements (particularly those for binary arithmetic) consist of one operator and three addresses (identifiers or temporaries):

$$\boxed{b := (a * 2) + (a * 2)}$$

t_1	:=	a	**mul**	2
t_2	:=	a	**mul**	2
t_3	:=	t_1	**add**	t_2
b	:=	t_3		

Three-Address Code...

- There are several ways of implementing three-address statements. They differ in the amount of space they require, how closely tied they are to the symbol table, and how easily they can be manipulated.
- During optimization we may want to move the three-address statements around.

Three-Address Code — Quadruples

- Quadruples can be implemented as an array of records with four fields. One field is the operator.
- The remaining three fields can be pointers to the symbol table nodes for the identifiers. In this case, literals and temporaries must be inserted into the symbol table.

$\boxed{b := (a * 2) + (a * 2)}$

NR	RES	OP	ARG$_1$	ARG$_2$
(1)	t_1	**mul**	a	2
(2)	t_2	**mul**	a	2
(3)	t_3	**add**	t_1	t_2
(4)	b	**assign**	t_3	

Three-Address Code — Triples

- **Triples** are similar to quadruples, but save some space.
- Instead of each three-address statement having an explicit **result** field, we let the statement itself represent the result.
- We don't have to insert temporaries into the symbol table.

$\boxed{b := (a * 2) + (a * 2)}$

NR	OP	ARG$_1$	ARG$_2$
(1)	**mul**	a	2
(2)	**mul**	a	2
(3)	**add**	(1)	(2)
(4)	**assign**	b	(3)

Three-Address Code — Indirect Triples

- One problem with triples ("The Trouble With Triples?"[a]) is that they cannot be moved around. We may want to do this during optimization. We can fix this by adding a level of indirection, an array of pointers to the "real" triples.

$b := (a * 2) + (a * 2)$

Abs	Real
(1)	(10)
(2)	(11)
(3)	(12)
(4)	(13)

NR	OP	ARG$_1$	ARG$_2$
(11)	**mul**	a	2
(12)	**mul**	a	2
(13)	**add**	(11)	(12)
(14)	**:=**	b	(13)

[a] This is a joke. It refers to the famous Star Trek episode "The Trouble With Tribbles." OK, so it's not funny.

Summary

Readings and References

- Read Louden:
  Intermediate Code 398–407
- Or, read the Dragon book:
  Postfix notation 33
  DAGs & Value Number Alg. 290–293
  Intermediate Languages 463–468, 470–473

Summary

- We use an intermediate representation of the program in order to isolate the back-end from the front-end.
- A high-level intermediate form makes the compiler retargetable (easily changed to generate code for another machine). It also makes code-generation difficult.
- A low-level intermediate form make code-generation easy, but our compiler becomes more closely tied to a particular architecture.
- A basic block is a *straight-line* piece of code, with no jumps in or out except at the beginning and end.

Homework

Homework I

- Translate the program below into quadruples, triples, and a 'sequence of expression trees.'

```
PROGRAM P;
VAR X : INTEGER; VAR Y : REAL;
BEGIN
   X := 1; Y := 5.5;
   WHILE X < 10 DO
      Y := Y + FLOAT(X);
      X := X + 1;
      IF Y > 10 THEN Y := Y * 2.2; ENDIF;
   ENDDO;
END.
```

Homework II

Consider the following expression:

$$((x * 4) + y) * (y + (4 * x)) + (z * (4 * x))$$

1. Show how the value-number algorithm builds a DAG from the expression (remember that $+$ and $*$ are commutative).
2. Show the resulting DAG when stored in an array.
3. Translate the expression to postfix form.
4. Translate the expression to indirect triple form.

Homework III

- Translate the program below into quadruples, triples, and a 'sequence of expression trees.'

```
PROGRAM P;
VAR X : INTEGER; VAR Y : REAL;
BEGIN
   X := 1; Y := 5.5;
   WHILE X < 10 DO
      Y := Y + FLOAT(X);
      X := X + 1;
      IF Y > 10 THEN Y := Y * 2.2; ENDIF;
   ENDDO;
END.
```

Tree Code Example

`record ProcBegin (Name,Level,VarSize,FormalSize,LineNo)`
The beginning of each procedure is marked by a `ProcBegin` instruction.

Attributes: `Name` is the name of the procedure. `Level` is the declaration level of the procedure. `VarSize` is the amount of local data (in bytes) required for the procedure. `FormalSize` is the size of the actual parameters passed to the procedure.

Children: None.

`record ProcEnd (Name,Level,VarSize,FormalSize,LineNo)`
Same attributes as `ProcBegin`

`record VarDecl (Name,Type,Level,Offset,Size,LineNo)`
Each *global* variable is given an explicit declaration in the intermediate code.

Attributes: `Type` is one of the strings `"INT"`, `"CHAR"`, `"STRING"`, `"REAL"`, or `"STRUCT"`. `Offset` gives the relative address of the variable

Children: None.

`record Store (Type,Des,Expr,LineNo)` Store instructions are generated from assignment statements.

Attributes: `Type` is the type.

Children: `Des` is the subtree which computes the address (L-value) at which the new value should be stored. `Expr` is a subtree computing the R-value.

`record VarRef (Type,Name,Level,LineNo)` `VarRef` nodes are used to refer to the **address** of a global or local variable.

Attributes: `Level` is the static nesting level at which the variable was declared.

Children: None.

`record Literal (Type,Value,LineNo)` `Literal` nodes represent integer, real, character, or string constants.

Attributes: `Type` is as for `VarDecl`, but there are no `"STRUCT"` constants.

Children: None.

`record BinExpr (Op,Type,Left,Right,LineNo)` `BinExpr` instructions represent binary arithmetic.

Attributes: `Type` is either `"INT"` or `"REAL"`. `Op` is one of `"+"`, `"-"`, `"*"`, `"/"`, or `"%"`.

Children: `Left` and `Right` are subtrees holding the left and right hand sides of the arithmetic operator, respectively.

`record Load (Type,Des,LineNo)` Load instructions represent the loading of an R-value from an L-value.

Attributes: `Type` is the type of the loaded value.

Children: `Des` is the subtree which computes the address (L-value) whose R-value should be loaded.

record Branch (Op,Type,Left,Right,Label,LineNo)
Conditional branch, equivalent to 'IF Left Op Right THEN GOTO Label'.

Attributes: Type is "INT", "CHAR", or "REAL". Op is one of "=", "<", ">", "#", "<=", ">=". Label is the number of the label to which we should jump.

Children: Left and Right are expression subtrees computing the arguments to the comparison operator.

record Goto (Label,LineNo) Unconditional branch.

Attributes: Label is the label number to which we should jump.

record Label (Number,LineNo) A program location to which a Branch or Goto may jump.

Attributes: Number is the label number.

record ProcCall (Name, Actuals, Level, LineNo)
Procedure call.

Attributes: Name is the name of the procedure. Level is the level at which the procedure is declared.

Children: Actuals is the list of actual parameters.

record Field (BaseAddress,Name,Offset,LineNo)
Compute the address of a field within a record variable, i.e. BaseAddress + Offset.

Attributes: Offset is the offset of the field within the record variable.

Children: BaseAddress is a subtree computing the address of the record variable.

record Index (BaseAddress,IndexExpr,ElementSize,ElementCou
Compute the address of an array element, i.e. BaseAddress + IndexExpr * ElementSize. If $0 >$ IndexExpr $\geq$ ElementCount then a run-time error message should be generated.

Attributes: ElementSize is the size of the array elements. ElementCount is the length of the array.

Children: BaseAddress and IndexExpr are subtrees computing the address of the beginning of the array the number of the indexed array element.

record Actual (Type,Number,Offset,Expr,LineNo) Pass the value of Expr as argument to a procedure call.

Attributes: If Type="STRUCT" then the formal is passed by reference, otherwise by value. Number is the argument number, Offset is the relative address within the activation record.

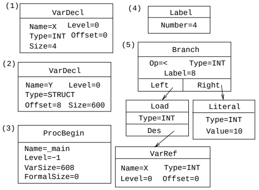
Children: Expr is the subtree computing the value (or address) of the actual.

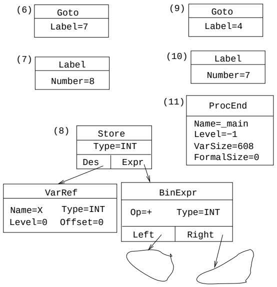
record FormalRef(Type,Name,Level,Offset,Number,LineNo)
Reference to the value of a formal parameter. Similar to VarRef.

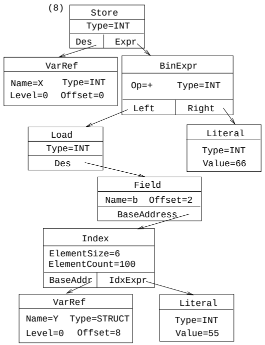
Attributes: Level is the static nesting level at which the formal is declared. If Type="STRUCT" then the formal is passed by reference, otherwise by value.

Children: None.

```
PROGRAM P;
TYPE R=RECORD[a;CHAR;b:INTEGER];
TYPE A=ARRAY 100 OF R;
VAR X:INTEGER;
VAR Y:A;
BEGIN
   WHILE X<10 DO
      X := Y[55].b + 66;
   ENDDO;
END.
```







```
(1) [VarDecl]: Name='X':Level=0:Type=INT:Offset=0:Size=4
(2) [VarDecl]: Name='Y':Level=0:Type=STRUCT:Offset=8:Size=6
(3) [ProcBegin]: Name='_main':Level=-1:VarSize=608:FormalSi
(4) [Label]: Number=4
(5) [Branch]: Op='<':Type=INT:Label=8
      Left=
         [Load]: Type=INT
             Des=
                [VarRef]: Name='X':Type=INT:Level=0:Offs
      Right=
         [Literal]: Type=INT:Value='10'
(6) [Goto]: Label=7
(7) [Label]: Number=8
```

```
(8) [Store]: Type=INT
      Des=
         [VarRef]: Name='X':Type=INT:Level=0:Offset=0
      Expr=
         [BinExpr]: Op='+':Type=INT
           Left= [SEE NEXT SLIDE]
           Right=
              [Literal]: Type=INT:Value='66'
(9) [Goto]: Label=4
(10 [Label]: Number=7
(11 [ProcEnd]: Name='_main':Level=-1:VarSize=608:FormalSiz
```

```
Expr=
   [BinExpr]: Op='+':Type=INT
   Left=
      [Load]: Type=INT
          Des=
             [Field]: Name='b':Offset=2
                BaseAddress=
                   [Index]: ElementSize='6':ElementCount=1
                       BaseAddress=
                          [VarRef]: Name='Y':Type=STRUCT:
                                    Level=0:Offset=8
                       IndexExpr=
                          [Literal]: Type=INT:Value='55'
```