CSc 553

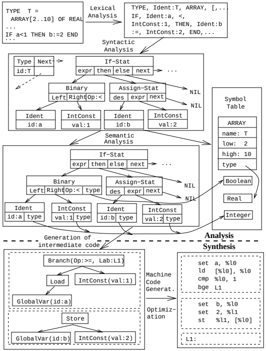
Principles of Compilation

4 : Intermediate Code

Department of Computer Science
University of Arizona

collberg@gmail.com

Introduction



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 otpimizations.
- 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 I

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.

A representation which is both architecture and language independent is known as an **UNCOL**, a **Universal Compiler Oriented Language**.

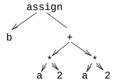
Intermediate Code II

- 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.
- 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.

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.

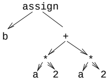
Tree & DAG Representations

Tree & DAG Repr. I

- 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 Ellimination** (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 Repr. II

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

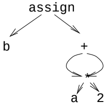


Linearized Tree:

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 Repr. III

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

Tuple Codes

Three-Address Code I

- 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:

x := y op z	Binary arithmetic or logical operation. Example: `Mul`, `And`.
x := op y	Unary arithmetic, conversion, or logical operation. Example: `Abs`, `UnaryMinus`, `Float`.
x := y	Copy statement.
goto L	Unconditional jump.

Three-Address Code II

if x `relop` 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.
param X; call P, n	Make X the next parameter; make a procedure call to `P` with n parameters.
x := y[i]	Indexed assignment. Set `x` to the value in the location `i` memory units beyond `y`.
x := ADDR(y)	Address assignment. Set `x` to the address of `y`.
x := IND(y)	Indirect assignment. Set `x` to the value stored at the address in `y`.
IND(x) := y	Indirect assignment. Set the memory location pointed to by `x` to the value held by `y`.

Three-Address Code III

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

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		

- 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 IV

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.

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

NR	RES	OP	ARG1	ARG2
(1)	t_1	**mul**	a	2
(2)	t_2	**mul**	a	2
(3)	t_3	**add**	t_1	t_2
(4)	t_1	**assign**	b	t_3

Three-Address Code V

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.

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

NR	OP	ARG1	ARG2
(1)	**mul**	a	2
(2)	**mul**	a	2
(3)	**add**	(1)	(2)
(4)	**assign**	b	(3)

Three-Address Code VI

Indirect Triples:

- One problem with triples ("The Trouble With Triples?"[a]) is that the around. We may want to do this during optimization.
- We can fix this by adding a level of indirection, an array of pointers

[a]This is a joke. It refers to the famous Star Trek episode "The Trouble With Tribbl

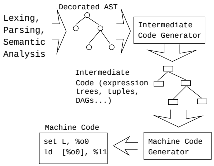
$b := (a * 2) + (a * 2)$

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

NR	OP	ARG1	ARG2
(11)	**mul**	a	2
(12)	**mul**	a	2
(13)	**add**	(11)	(12)
(14)	**:=**	b	(13)

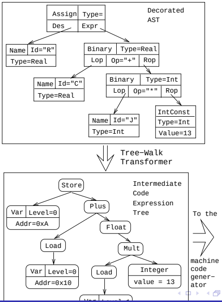
Intermediate Code Generation

Interm. Code Generation



- After semantic analysis we traverse the AST and emit the correct intermediate code.
- The next slide shows how an expression tree is generated from an AST. The `float` can easily be inserted since all types are known in the AST.

Generating Expression Trees



Generating Quadruples I

- Each AST node in an expression sub-tree is given an attribute ⇑Place:**SymbolT** which represents the name of the identifier or temporary in which the value of the subtree will be computed.

Tree-Walk Transformer:

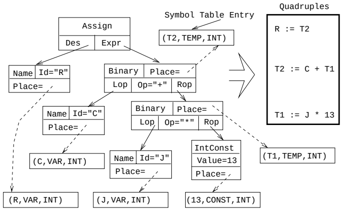
```
PROCEDURE Program (n: Node);
   Decl(n.DeclSeq); Stat(n.StatSeq);
END;

PROCEDURE Decl (n: Node);
   IF n.Kind = ProcDecl THEN
      Decl(n.Locals); Decl(n.Next);
      Stat(n.StatSeq);
   ENDIF
END;
```

Generating Quadruples II

- NewTemp generates a new temporary var.

```
PROCEDURE Stat (n: Node);
  IF n.Kind = Assign THEN
    Expr(n.Des); Expr(n.Expr);
    Emit(n.Des.Place ':=' n.Expr.Place);
    Stat(n.Next);
  ENDIF
END;
PROCEDURE Expr (n: Node);
  IF n.Kind = Add THEN
    Expr(n.LOP); Expr(n.ROP);
    n.Place := NewTemp();
    Emit(n.Place ':='
      n.LOP.Place '+' n.ROP.Place);
  ELSIF n.Kind = VarRef THEN
    n.Place := n.Symbol;
```

Generating Quadruples III



Attribute Grammar Notation

Attribute Grammar Notation I

- The book uses a convenient way to describe attribute computations, called an **attribute grammar** notation.
- We simply combine the abstract syntax notation with the attribute computations that have to be performed at the corresponding AST nodes:

```
A ::= B C
  { C.d := B.c + 1;
    A.b := A.a + B.c; }
```

- Note that it is not directly obvious from this notation which attributes are synthesized and inherited, and in which order the nodes should be visited. We have to figure this out ourselves!

Attribute Grammar Notation II

- Now we can rewrite our tuple generator using the new notation:

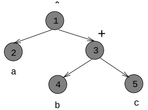
```
Assign ::= Des:Expr Expr:Expr
  {    Emit(n.Des.Place ':=' n.Expr.Place);
  }

Add ::= LOP:Expr ROP:Expr
  {    n.Place := NewTemp();
       Emit(n.Place ':='
            n.LOP.Place '+' n.ROP.Place);
  }

Name ::= Ident
  {    n.Place := n.Symbol; }
```

Building DAGs

Building DAGs I

- From an expression/expression tree such as this one:

$a * (b + c)$



- We might generate this machine code (for some fictious architecture):

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

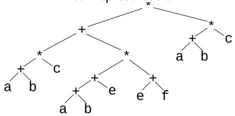
- Can we generate better code from a DAG than a tree?

Building DAGs II

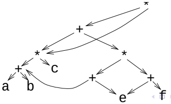
Example Expression:

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

Tree Representation:



DAG Representation:



Building DAGs III

- Generating machine code from the tree yields 21 instructions.

Code from DAG

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

Building DAGs IV

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

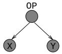
Code from DAG

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

Building DAGs V

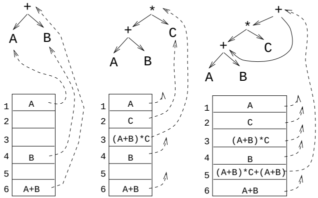
- Repeatedly add subtrees to build DAG. Only add subtrees not already in DAG. Store subtrees in a hash table. This is the
- **value-number** algorithm.

  For every insertion of a subtree, check if $(X \texttt{ OP } Y) \in$ 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 VI – Example



Summary

Readings and References

- Read the Tiger book:
  Translation to Intermediate Code Chapter 7.
- Or, read the Dragon book:
  Postfix notation 33
  DAGs & Value Number Alg. 290–293
  Intermediate Languages 463–468, 470–473
  Assignment Statements 478–481

Summary I

- 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.

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.