INTERMEDIATE CODE GENERATION
Part I

Based on Chapter 6 of Aho, Lam, Sethi, Ullman:

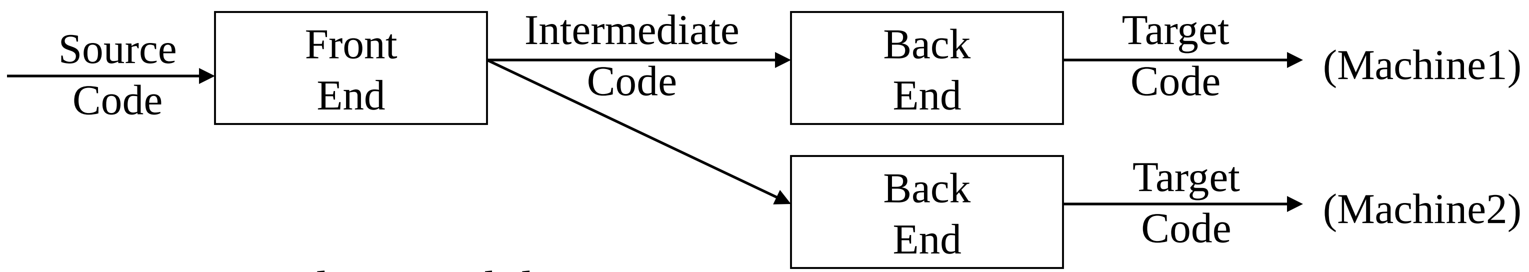
Compilers: Principles, Techniques, & Tools

2nd Ed, Addison Wesley, 2007

Table of Contents

Introduction

- Analysis and synthesis
  - Analysis part of compiler
    - lexical analysis, syntax analysis, semantic analysis, optimization
  - Synthesis part
    - intermediate code generation, code generation, optimization
- Front-end and back-end



  - increases compiler portability

Intermediate Representation

- There are may forms of intermediate representation
  - It may be very high level(syntax tree) or resemble target code
  - It may or may not use detailed information about the target machine (data type size, location of variable etc.)
  - It may or may not incorporate all the information in the symbol table
- Syntax Tree vs Intermediate Code
  - Intermediate code is linearized representation of syntax tree
  - Intermediate code is useful when the compiler is to produce extremely efficient code (significant amount of analysis of intermediate code is required, e.g. optimization)
- Intermediate Code
  - Three Address Code
  - Stack Machine Code(Zero-Address Code)

Three-Address Code

- The general form

  x = y op z

  ** address = name, constant, or temporary variable*

- Example

  x + y * z

  t1 = y * z

  t2 = x + t1

  a = b * -c + b * -c

  t1 = -c

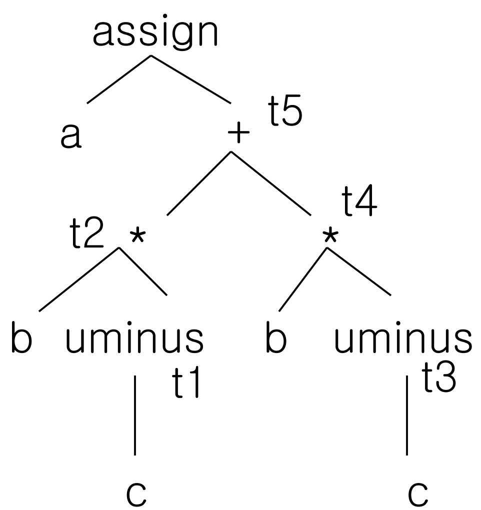
  t2 = b * t1

  t3 = -c

  t4 = b * t3

  t5 = t2 + t4

  a = t5



Three-Address Code

- Types of Three Address Code

1.	Assignment	x = y *op* z and x = *op* y
2.	Copy statement	x = y
3.	Unconditional jump	goto L *L is a label*
4.	Conditional jump	if x *relop* y goto L
		if x goto L
5.	Procedure call	param x_1
		param x_2
		…
		param x_n
		call p, n
6.	Return	return y and return
7.	Indexed assignment	x = y[i] and x[i] = y
8.	Pointer assignment	x = &y, x = *y, and *x = y

Three-Address Code

- Example

do i = i+1; while (a[i] < v);

L :	t1 = i + 1	100:	t1 = i + 1
	i = t1	101:	i = t1
	t2 = i * 8	102:	t2 = i * 8
	t3 = a[t2]	103:	t3 = a[t2]
	if t3 < v goto L	104:	if t3 < v goto 100

Representation of Three-Address Code

- Quadruples
  - Four fields: op, argument 1, argument 2, result

t1 = -c

t2 = b * t1

t3 = -c

t4 = b * t3

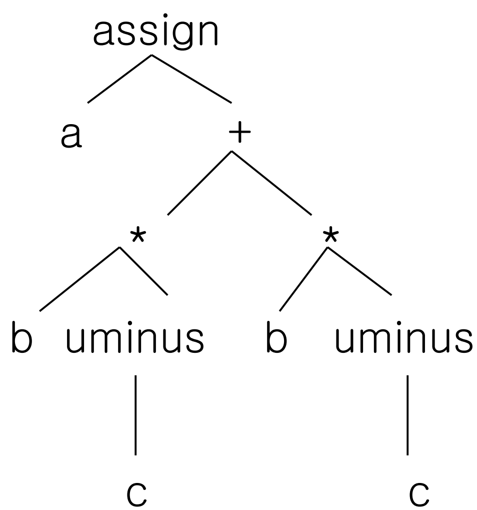
t5 = t2 + t4

a = t5

	op	arg1	arg2	result
0	minus	c		t1
1	*	b	t1	t2
2	minus	c		t3
3	*	b	t3	t4
4	+	t2	t4	t5
5	=	t5		a

Representation of Three-Address Code

- Triples
  - the result of an operation is referred by its position



	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Representation of Three-Address Code

- Indirect Triples
  - consist of a listing of pointers to triples

35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	…

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Exemplary Language

- Grammar for Simple Language

  P -> D; S

  D -> T id ; D | ε

  T -> B C

  B -> int | double

  C -> [num] C | ε

  S -> id = E ;

  | L = E ;

  | if (E) S_1

  | while (E) do S_1

  | S_1 S_2

  E -> E + E | id | L

  L -> id [E] | L [E]

Types and Declaration

- Two Usages of Types
  - type checking
  - translation
- Type checking
  - verifies that the type of a construct matches that expected by its context
    - mod operator has integer operands
    - dereferencing is applied only to a pointer
    - a function is applied to a correct number and type of arguments
    - etc.
  - Static type checking vs dynamic type checking
- Translation Applications
  - storage layout
  - address calculation
  - type conversion
  - etc

Types and Declarations

- Type expressions
  - the type of a language construct can be denoted by a type expression
  - type expression
    - is a basic type
    - is formed by applying type constructor to other type expressions
- Exemplary Type Expressions
  - a basic type is a type expression: *int*, *char*, *float*, *void*, a special type: *type_error*
  - type name is a type expression : e.g. by typedef in C
  - type constructor applied to type expressions is a type expression
    - Arrays. e.g. *array*(2, *array*(3, *integer*)) for int[2][3]
    - Products. If *T1* and *T2* are type expressions, *T1*x*T2* is a type expression

Types and Declarations

- Records. The record type constructor can be applied to a tuple formed from filed names and field types.

  e.g. struct row {

  int address;

  char lexeme[15];

  }

  struct row table[100];

  - the type name *row* denotes the type expression:

    record((address x *integer*) x (lexeme x *array*(15, *char*)))

- Pointers. If T is a type expression , then *pointer*(T) is type expression.

  struct row* ptr; => *pointer*(*row*)

- Functions. If a function maps domain type D to a range type R, the type of the function is denoted by D -> R

  int* function f(char a, char b) => *char* x *char* -> *pointer*(*integer*)

Types and Declarations

- Type Equivalence
  - name equivalence
    - two variables have compatible types if their type name is the same
    - e.g. structures in C, classes in Java
  - structural equivalence
    - two variables have compatible types if they have the same structure
    - e.g. pointers, arrays, and their combinations in C

Types and Declarations

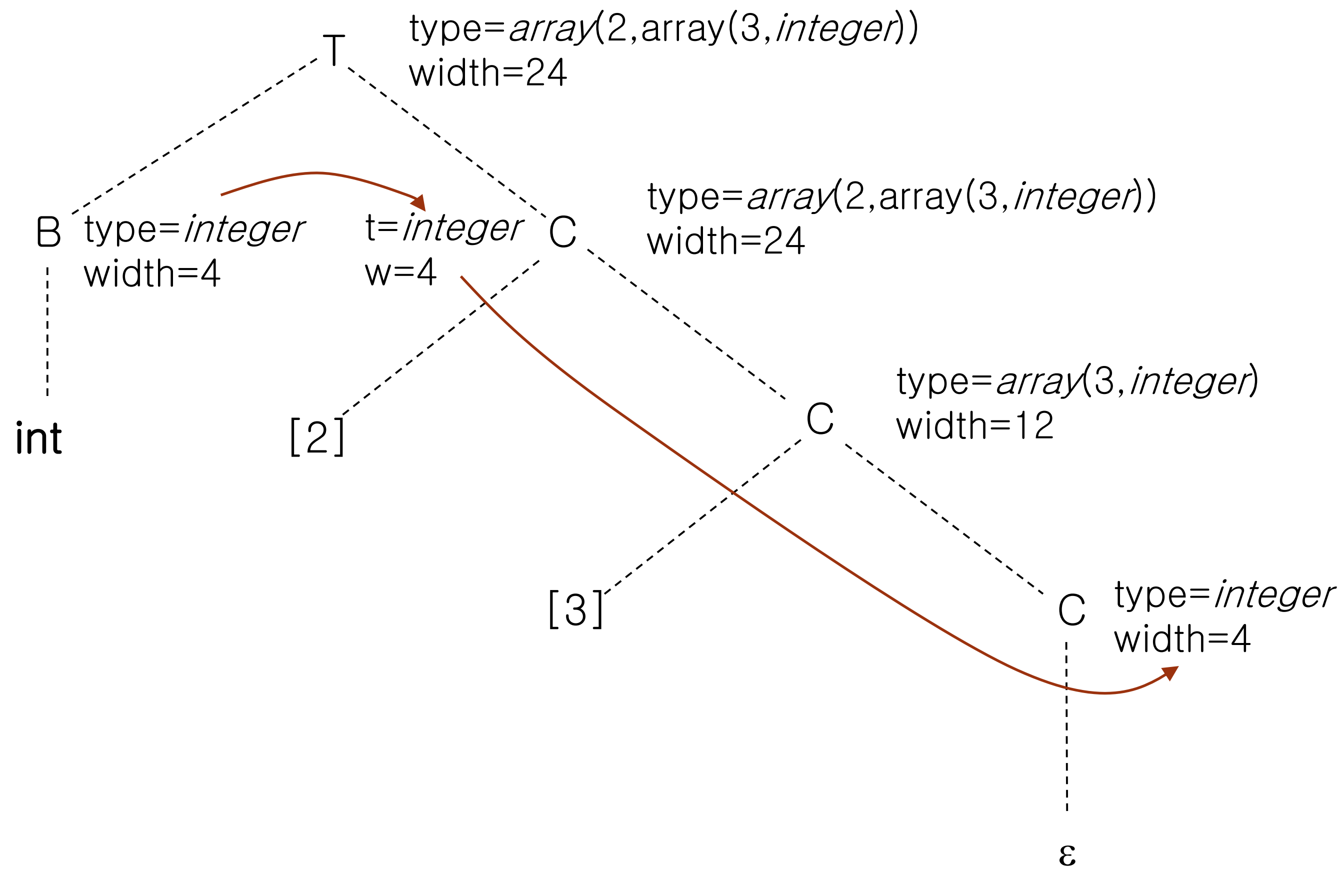
- Declarations and Storage Layout
  - Translation Scheme for calculating type and width information

T -> B	{ t = B.type; w = B.width; }
C	{T.type = C.type; T.width = C.width }
B -> int	{ B.type = *integer*; B.width = 4; }
| double	{ B.type = *double*; B.width = 8; }
C -> [num] C_1	{ C.type = array(num.value, C_1.type); C.width = num.value x C_1.width; }
| ε	{ C.type = t; C.width = w; }

Types and Declarations

- Example 6.9 (P. 375)

  int[2][3]



Types and Declarations

- Sequence of Declarations

  P -> D; S

  D -> T id ; D | ε

- Translation Scheme for evaluating type, offset and width

  P -> {D.offset = 0; }

  D

  D -> T id ; { enter(id.lexeme, T.type, D.offset);

  D_1.offset = D.offset + T.width }

  D_1

  | ε

- Using Marker (Syntax Directed Definition)

P -> M D	D.offset = M.offset
M -> ε	M.offset = 0
D -> T id ; N D_1	N.type = T.type; N.width = T.width; N.offset=D.offset
	N.lexeme = id.lexemel; D_1.offset = N.newoffset
D -> ε	
N -> ε	enter(N.lexeme, N.type, N.offset)
	N.newoffset = N.offset + N.width

Types and Declarations

- Simplifying global variables

```
P ->            { offset = 0; }
    D
D -> T id ;     { enter(id.lexeme, T.type, offset);
                  offset = offset + T.width }
    D1
  | ε
```

- Using Marker (Syntax Directed Definition)

```
P -> M D
M -> ε              offset = 0;
D -> T id ; N  D1   N.type = T.type; N.width = T.width;
                    N.lexeme = id.lexeme
D -> ε
N -> ε              enter(N.lexeme,  N.type,  offset)
                    offset= offset + N.width
```

Types and Declarations

- Implementation

  P -> M D

  M -> ε { offset = 0; }

  D -> T id ; N D_1

  D -> ε

  N -> ε { enter(val[top-1].lexeme, val[top-2].type, offset)

  offset= offset + val[top-2].width }

- Alternative Grammar

  P -> M D

  M -> ε { offset = 0; }

  D -> D T id ; { enter(id.lexeme, T.type, offset); offset += T.width; }

  D -> T id ; { enter(id.lexeme, T.type, offset); offset += T.width; }

Translation of Expressions

- Syntax Directed Definition for Expressions

S -> id = E ;	S.code = E.code || gen(get(id.lexeme) '=' E.addr)
E -> E1 + E2	E.addr= newtemp()
	$E.code = E_1.code$ || $E_2.code$ || gen(E.addr '=' E_1. addr '+' E_2.addr)
E -> - E1	E.addr = newtemp()
	E.code = E_1.code || gen(E. addr '=' 'uminus' E_1.addr)
E -> (E_1)	E. addr = E_1.addr
	E.code = E_1.code
E -> id	E.addr = get(id.lexeme)
	E.code = ''

Translation of Expressions

- Example 6.11 (P. 380)

  a = b + -c

  t1 = minus c

  t2 = b + t1

  a = t2

Translation Scheme for Expressions

- *gen* function outputs code incrementally

S -> id = E	{ gen(get(id.lexeme) '=' E.addr); }
E-> E1 + E2	{ E.addr = newtemp; gen(E.addr '=' E_1.addr '+' E_2.addr); }
E -> - E1	{ E. addr = newtemp; gen(E. addr'=' 'uminus ' E_1.addr); }
E -> (E1)	{ E.addr = E_1.addr }
E -> id	{ E.addr = get(id.lexeme); }

Addressing Array Elements

- Relative address of i-*th* element (1-dim array)
  - base : the relative address of the array
  - low : lower bound on the subscript
  - w : width of array element

  base + (i - low) * w

- $A[i_1][i_2]$ (assume that lower bounds are 0)

  $base + (i_1 * n_2 + i_2) * w$

- $A[i_1][i_2]\ldots[i_k]$

  $base + (i_1 * n_2 * n_3 * \ldots * n_k + i_2 * n_3 * \ldots * n_k + \ldots + i_{k-1} * n_k + i_k) * w$

  $\Rightarrow base + i_1 * n_2 * n_3 * \ldots * n_k * w + i_2 * n_3 * \ldots * n_k * w + \ldots +$

  $i_{k-1} * n_k * w + i_k * w$

Addressing Array Elements

- Generating code for array reference

```
S-> id  = E ;          {gen(get(id.lexeme) '=' E.addr); }
    |  L = E ;         { gen(L.array.base '[' L.addr ']' '=' E.addr); }
E-> E1 + E2            { E.addr = newtemp;
                         gen(E. addr '=' E1. addr '+' E2. addr); }
...
    | L                { E.addr = newtemp();
                         gen(E.addr '=' L.array.base '[' L.addr ']'); }
L -> id [ E ]          { L.array = get(id.lexeme);
                         L.type = L.array.type.elem;
                         L.addr = newtemp();
                         gen(L.addr '=' E.addr '*' L.type.width); }
L -> L1 [ E ]          { L.array = L1.array;
                         L.type = L1.type.elem;
                         t = newtemp();
                         L.addr = newtemp();
                         gen(t '=' E.addr '*' L.type.width; );
                         gen(L.addr '='L1.addr '+' t);          }
```

Addressing Array Elements

- Example

int a[2][3];

c + a[i][j]

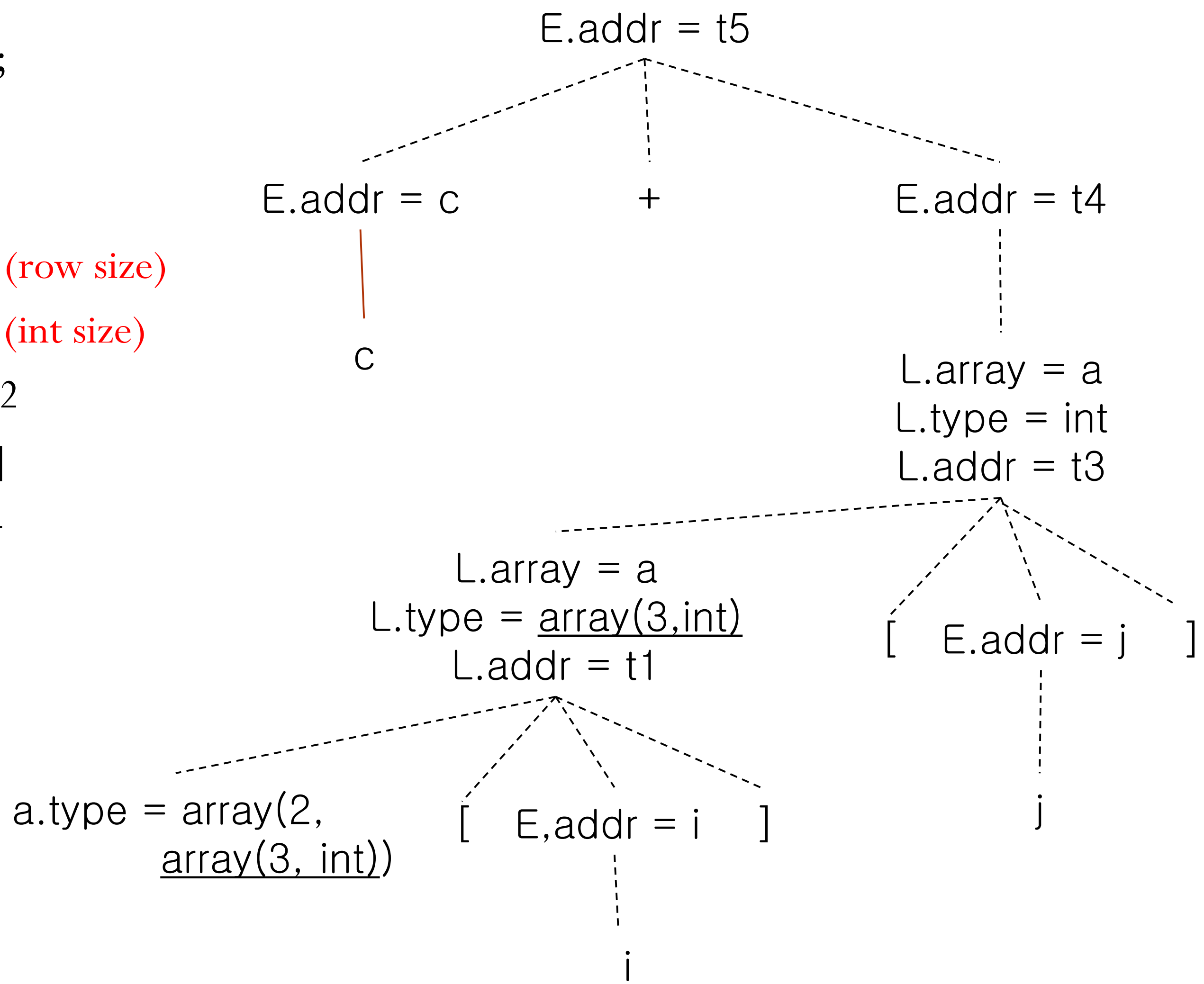
t1 = i * 12 (row size)

t2 = j * 4 (int size)

t3 = t1 + t2

t4 = a [t3]

t5 = c + t4



Type Checking

E-> literal	{ E.*type* = *char* }
E -> num	{ E.*type* = *integer* }
E -> id	{ E.*type* = getType(id.*lexeme*) }
E -> E_1 mod E_2	{ E.*type* = if E_1.*type* = *integer* and E_2.*type* = *integer* then *integer* else *type_error* }
E -> E_1 [E_2]	{ E.type = if E_2.*type* = *integer* and E_1.*type* = *array*(*s*,*t*) then *t* else *type_error* }
E -> *E_1	{ E.type = if E_1.*type* = *pointer*(*t*) then *t* else *type_error* }
E -> E_1(E_2)	{ E.type = if E_1.*type* = *s* -> *t* and E_2.*type* = *s* then *t* else *type_error* }

Type Conversions

E-> E1 + E2

```
{
E.addr = newtemp;
if E1.type = integer and E2.type = integer then
    gen(E. addr ‘=‘ E1. addr ‘int+’ E2.addr);
    E.type = integer;
else if E1.type = real and E2.type = real then
    gent(E. addr ‘=‘ E1. addr ‘real+’ E2addr);
    E.type = real;
else if E1.type = integer and E2.type = real then
    u = newtemp;
    gen (u ‘=‘ ‘inttoreal’ E1. addr);
    gen (E. addr ‘=‘ u ‘real+’ E2. addr);
    E.type = real;
else if E1.type = real and E2.type = integer then
    u = newtemp;
    gen (u ‘=‘ ‘inttoreal’ E2. addr);
    gen (E. addr ‘=‘ E1. addr ‘real+’ u);
    E.type = real;
else
   E.type = type_error;
endif
}
```

Type Conversions

- Example

x = y + i * j

=>

t1 = i int* j

t3 = inttoreal t1

t2= y real+ t3

x = t2