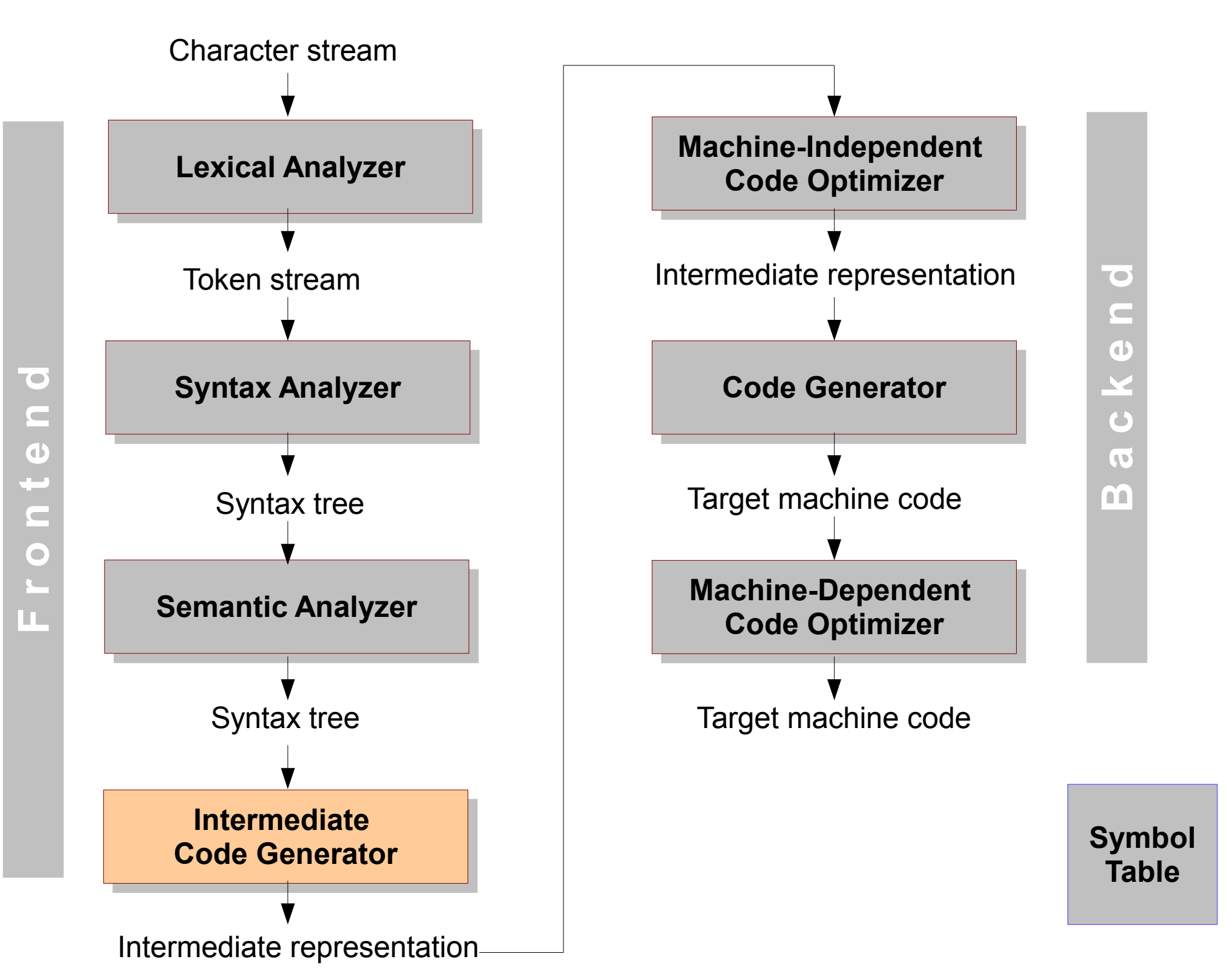
Intermediate Code Generation

Rupesh Nasre.

CS3300 Compiler Design
IIT Madras
Aug 2015

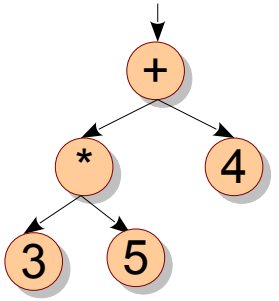

Character stream
Lexical Analyzer
Token stream
Syntax Analyzer
Syntax tree
Semantic Analyzer
Syntax tree
Intermediate Code Generator
Intermediate representation
Machine-Independent Code Optimizer
Intermediate representation
Code Generator
Target machine code
Machine-Dependent Code Optimizer
Target machine code
Frontend
Backend
Symbol Table

Role of IR Generator

- To act as a glue between front-end and backend (or source and machine codes).
- To lower abstraction from source level.
  - To make life simple.
- To maintain some high-level information.
  - To keep life interesting.
- Complete some syntactic checks, perform more semantic checks.
  - e.g. *break* should be inside loop or *switch* only.

Representations

- Syntax Trees
  - Maintains structure of the construct
  - Suitable for high-level representations
- Three-Address Code
  - Maximum three addresses in an instruction
  - Suitable for both high and low-level representations
- Two-Address Code
- …
  - e.g. C




```
t1 = 3 * 5
t2 = t1 + 4
```

3AC

```
mult 3, 5
add 4
```

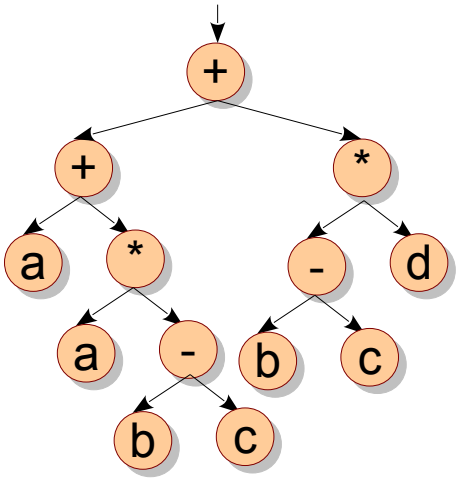
2AC

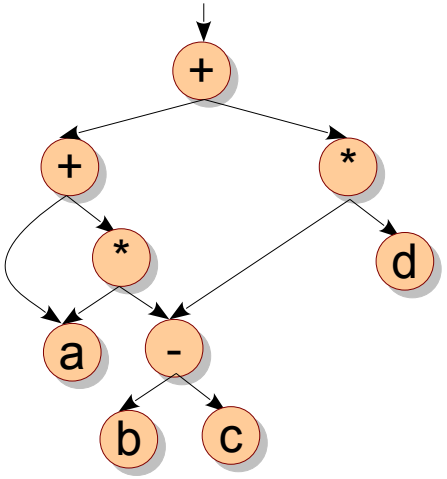
```
push 3
push 5
mult
add 4
```

1AC or stack machine

Syntax Trees and DAGs

a + a * (b – c) + (b – c) * d



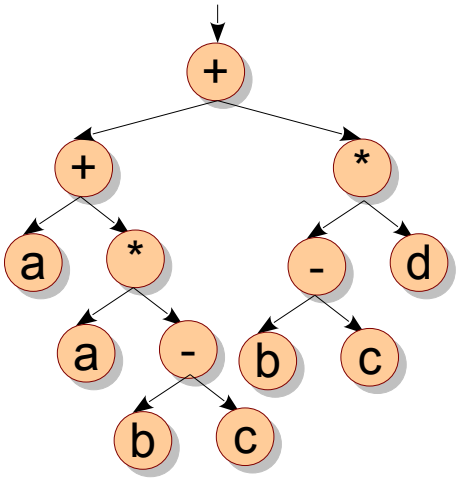


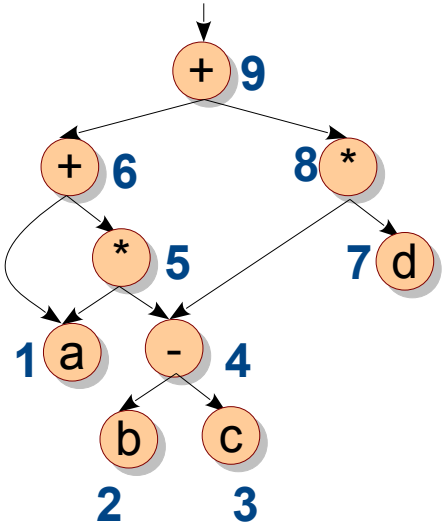
A small problem:
subgraph isomorphism is NP-complete.

Production	Semantic Rules
E → E + T	$$.node = new Op($1.node, '+', $3.node)
E → E - T	$$.node = new Op($1.node, '-', $3.node)
E → T	$$.node = $1.node
T → (E)	$$.node = $2.node
T → *id*	$$.node = new Leaf($1)
T → *num*	$$.node = new Leaf($1)

Value Numbering

a + a * (b – c) + (b – c) * d





A small problem:
subgraph isomorphism is NP-complete.

- Uniquely identifies a node in the DAG.
- A node with value number V contains children of numbers < N.
- Thus, an ordering of the DAG is possible.
- This corresponds to an evaluation order of the underlying expression.
- For inserting l op r, search for node op with children l and r.
- **Classwork:** Find value numbering for a + b + a + b.

Three-Address Code

- An address can be a name, constant or temporary.
- Assignments x = y op z; x = op y.
- Copy x = y.
- Unconditional jump goto L.
- Conditional jumps if x relop y goto L.
- Parameters param x.
- Function call y = call p.
- Indexed copy x = y[i]; x[i] = y.
- Pointer assignments x = &y; x = *y; *x = y.

3AC Representations

- Triples
- Quadruples

Instructions cannot be reordered.

Instructions can be reordered.

Assignment statement: **a = b * - c + b * - c;**

			op	arg1	arg2	result
t1	=	minus c	minus	c		t1
t2	=	b * t1	*	b	t1	t2
t3	=	minus c	minus	c		t3
t4	=	b * t3	*	b	t3	t4
t5	=	t2 + t4	+	t2	t4	t5
a	=	t5	=	t5		a

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

3AC Representations

- Triples
- Quadruples

Instructions cannot be reordered.

Assignment statement: **a = b * - c + b * - c;**

(0)
(1)
(2)
(3)
(4)
(5)

(2)
(3)
(0)
(1)
(4)
(5)

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

Indirect triples can be reordered

SSA

- **Classwork**: Allocate registers to variables.
- Some observations
  - Definition of a variable *kills* its previous definition.
  - A variable's use refers to its most *recent* definition.
  - A variable holds a register for a long time, if it is *live* longer.

```
p = a + b
q = p – c
p = q * d
p = e – p
q = p + q
```

a	r1	r1
b	r2	r2
p	r3	r1
c	r4	r2
q	r5	r1
d	r6	r2
e	r7	r3

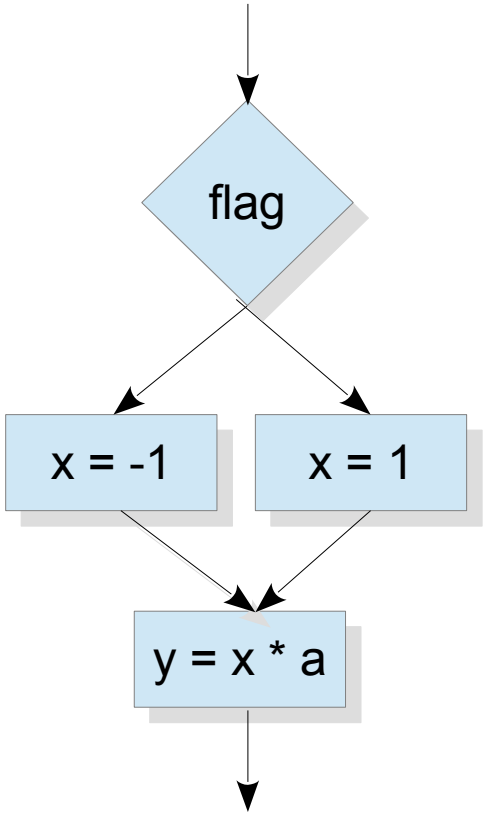
$p_1 = a + b$
$q_1 = p_1 – c$
$p_2 = q_1 * d$
$p_3 = e – p_2$
$q_2 = p_3 + q_1$

SSA

- Static Single Assignment
  - Each definition refers to a different variable (instance)

```
if (flag)
      x = -1;
else
      x = 1;
y = x * a;
```

```
if (flag)
      x1 = -1;
else
      x2 = 1;
x3 = Φ(x1, x2)
y = x3 * a;
```

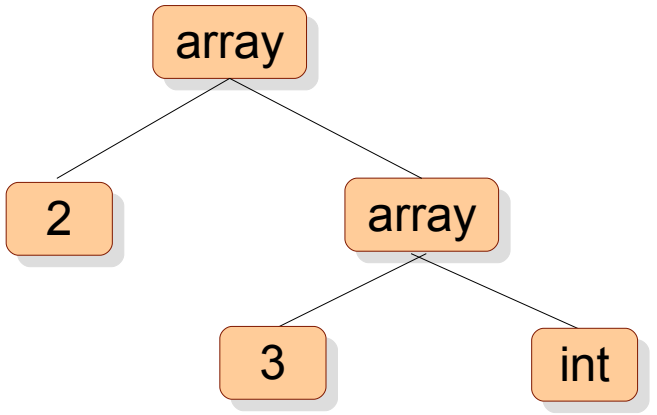


Language Constructs

- Declarations
  - Types (int, int [], struct, int *)
  - Storage qualifiers (array expressions, const, static)
- Assignments
- Conditionals, switch
- Loops
- Function calls, definitions

SDT Applications

- Finding type expressions
  - int a[2][3] is array of 2 arrays of 3 integers.
  - in functional style: *array(2, array(3, int))*

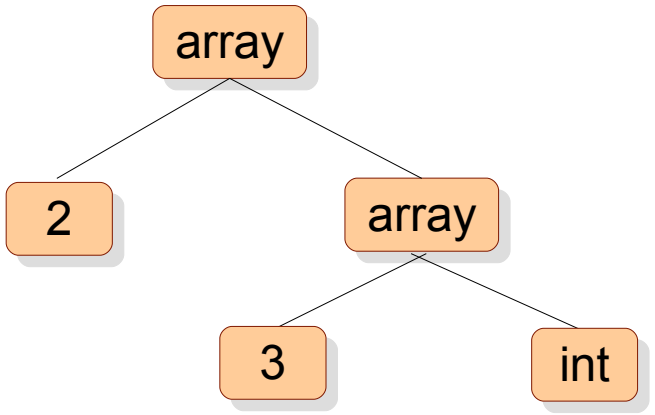


Production	Semantic Rules
T → B id C	T.t = C.t C.i = B.t
B → *int*	B.t = *int*
B → *float*	B.t = *float*
C → [num] C_1	C.t = array(num, C_1.t) C_1.i = C.i
C → ε	C.t = C.i

Classwork: Write productions and semantic rules for computing types and finding their widths in bytes.

SDT Applications

- Finding type expressions
  - int a[2][3] is array of 2 arrays of 3 integers.
  - in functional style: *array(2, array(3, int))*

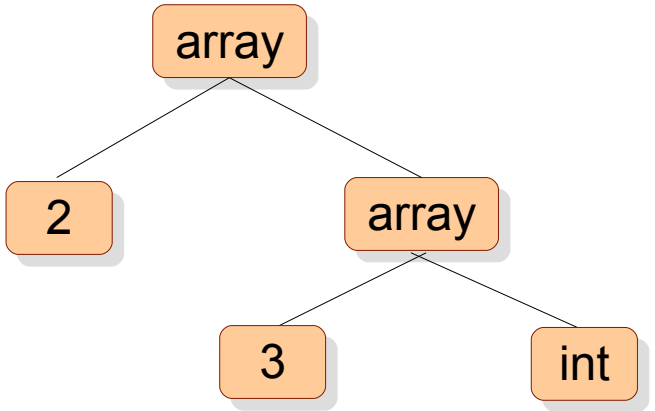


Production	Semantic Rules
T → B id C	T.t = C.t; C.iw = B.sw; C.i = B.t; T.sw = C.sw;
B → *int*	B.t = *int;* B.sw = 4;
B → *float*	B.t = *float*; B.sw = 8;
C → [num] C_1	C.t = array(num, C_1.t); C_1.i = C.i; C.sw = C_1.sw * num.value;
C → ε	C.t = C.i; C.sw = C.iw;

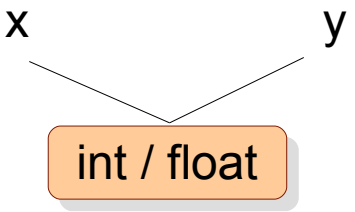
Classwork: Write productions and semantic rules for computing types and finding their widths in bytes.

Type Equivalence

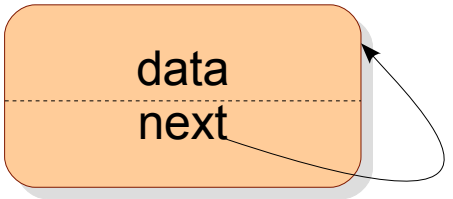
- Since type expressions are possible, we need to talk about their equivalence.
- Let's first structurally represent them.
- **Question:** Do type expressions form a DAG? Can they contain cycles?



int a[2][3]




```
union {
    int x;
    float y;
};
```




```
struct node {
    int data;
    struct node *next;
};
```

Type Equivalence

- Two types are structurally equivalent iff one of the following conditions is true.
  1. They are the same basic type.
  2. They are formed by applying the same constructor to structurally equivalent types.

  Name equivalence

  3. One is a type name that denotes the other. — ***typedef***

  - *int a[2][3]* is not equivalent to *int b[3][2]*;
  - *int a* is not equivalent to *char b[4]*;
  - *struct {int, char}* is not equivalent to *struct {char, int}*;
  - *int ** is not equivalent to *void **.

Type Checking

- Type expressions are checked for
  - Correct code
  - Security aspects
  - Efficient code generation
  - …
- Compiler determines that type expressions conform to a collection of logical rules, called as the *type system* of the source language.
- *Type synthesis*: if *f* has type $s \rightarrow t$ and *x* has type *s*, then expression *f(x)* has type *t*.
- *Type inference*: if f(x) is an expression, then for some α and β, f has type $\alpha \rightarrow \beta$ and x has type α.

Type System

- Potentially, everything can be checked dynamically...
  - if type information is carried to execution time.
- A *sound* type system eliminates the need for dynamic type checking.
- A language implementation is *strongly typed* if a compiler guarantees that the valid source programs (it accepts) will run without type errors.

Type Conversions

- int a = 10; float b = 2 * a;
- **Widening** conversion are safe.
  - int → long → float → double.
  - Automatically done by compiler, called *coercion*.
- **Narrowing** conversions may not be safe.
  - int → char.
  - Usually, enforced by the programmers, called *casts*.
  - Sometimes, deferred until runtime, dyn_cast<...>.

Declarations

- When declarations are together, a single offset on the stack pointer suffices.
  - int x, y, z; fun1(); fun2();
- Otherwise, the translator needs to keep track of the current offset.
  - int x; fun1(); int y, z; fun2();
- A similar concept is applicable for fields in structs.
- Blocks and Nestings
  - Need to push the current environment and pop.

Expressions

- We have studied expressions at length.
- To generate 3AC, we will use our grammar and its associated SDT to generate IR.
- For instance, a = b + - c would be converted to
  - t1 = minus c
  - t2 = b + t1
  - a = t2

Array Expressions

- For instance, create IR for c + a[i][j].
- This requires us to know the types of a and c.
- Say, c is an integer (4 bytes) and a is int [2][3].
- Then, the IR is

```
t1 = i * 12    ; 3 * 4 bytes
t2 = j * 4     ; 1 * 4 bytes
t3 = t1 + t2   ; offset from a
t4 = a[t3]     ; assuming base[offset] is present in IR.
t5 = c + t4
```

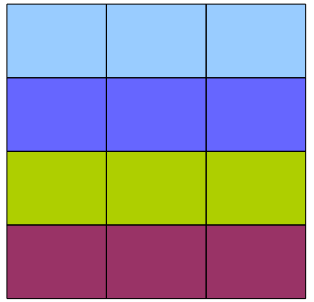
Array Expressions

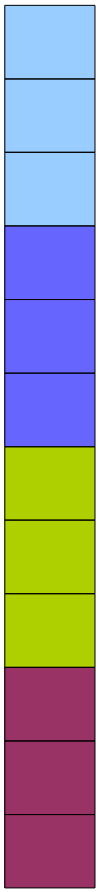
- a[5] is a + 5 * sizeof(type)
- a[i][j] for a[3][5] is a + i * 5 * sizeof(type) + j * sizeof(type)
- This works when arrays are zero-indexed.
- **Classwork:** Find array expression to be generated for accessing a[i][j][k] when indices start with low, and array is declared as type a[10][20][30].
- **Classwork:** What all computations can be performed at compile-time?
- **Classwork:** What happens for malloced arrays?

Array Expressions

```
void fun(int a[][]) {
        a[0][0] = 20;
}
void main() {
        int a[5][10];
        fun(a);
        printf("%d\n", a[0][0]);
}
```

We view an array to be a D-dimensional matrix. However, for the hardware, it is simply single dimensional.



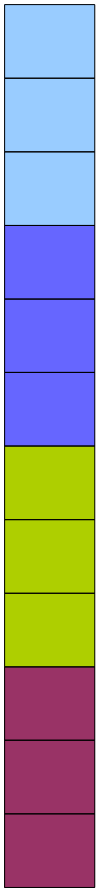


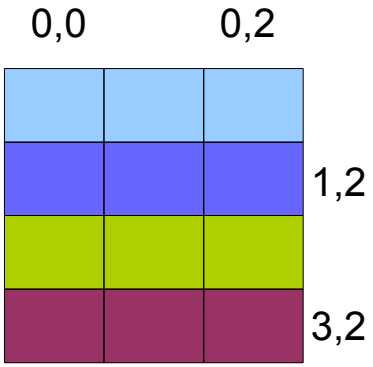
ERROR: type of formal parameter 1 is incomplete

- How to optimize computation of the offset for a long expression a[i][j][k][l] with declaration as int a[w4][w3][w2][w1]?
  - i * w3 * w2 * w1 + j * w2 * w1 + k * w1 + l
  - Use Horner's rule: ((i * w3 + j) * w2 + k) * w1 + l

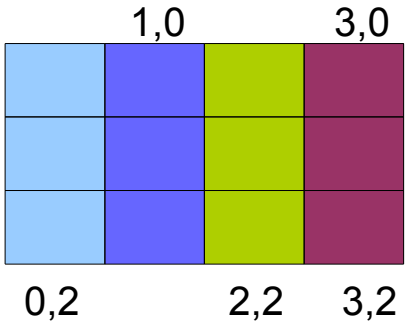
Array Expressions

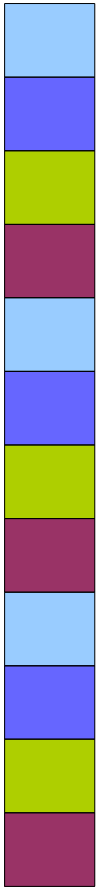
- In C, C++, Java, and so far, we have used *row-major* storage.
  - All elements of a row are stored together.





- In Fortran, we use *column-major* storage format.
  - each column is stored together.





IR for Array Expressions

- $L \rightarrow id\ [E] \mid L\ [E]$

$L \rightarrow id\ [\ E\ ]$	{ L.type = id.type; L.addr = new Temp(); gen(L.addr '=' E.addr '*' L.type.width); }
$L \rightarrow L_1\ [\ E\ ]$	{ L.type = L_1.type; T = new Temp(); L.addr = new Temp(); gen(t '=' E.addr '*' L.type.width); gen(L.addr '=' L_1.addr '+' t); }
$E \rightarrow id$	{ E.addr = id.addr; }
$E \rightarrow L$	{ E.addr = new Temp(); gen(E.addr '=' L.base '[' L.addr ']'); }
$E \rightarrow E_1 + E_2$	{ E.addr = new Temp(); gen(E.addr '=' E_1.addr + E_2.addr); }
$S \rightarrow id = E$	{ gen(id.name '=' E.addr); }
$S \rightarrow L = E$	{ gen(L.base '[' L.addr ']' '=' E.addr); }

Note that the grammar is left-recursive.

```
t1 = i * 12     ; 3 * 4 bytes
t2 = j * 4      ; 1 * 4 bytes
t3 = t1 + t2    ; offset from a
t4 = a[t3] ; assuming base[offset] is present in IR.
t5 = c + t4
```

$L \rightarrow id\ [\ E\ ]$	{ L.type = id.type; L.addr = new Temp(); gen(L.addr '=' E.addr '*' L.type.width); }
$L \rightarrow L_1\ [\ E\ ]$	{ L.type = L_1.type; T = new Temp(); L.addr = new Temp(); gen(t '=' E.addr '*' L.type.width); gen(L.addr '=' L_1.addr '+' t); }
$E \rightarrow id$	{ E.addr = id.addr; }
$E \rightarrow L$	{ E.addr = new Temp(); gen(E.addr '=' L.base '[' L.addr ']'); }
$E \rightarrow E_1 + E_2$	{ E.addr = new Temp(); gen(E.addr '=' E_1.addr + E_2.addr); }
$S \rightarrow id = E$	{ gen(id.name '=' E.addr); }
$S \rightarrow L = E$	{ gen(L.base '[' L.addr ']' '=' E.addr); }

Control Flow

- Conditionals
  - if, if-else, switch
- Loops
  - for, while, do-while, repeat-until
- We need to worry about
  - Boolean expressions
  - Jumps (and labels)

Control-Flow – Boolean Expressions

- B → B || B | B && B | !B | (B) | E relop E | *true* | *false*
- relop → < | <= | > | >= | == | !=
- What is the associativity of ||?
- What is its precedence over &&?
- How do we optimize evaluation of (B_1 || B_2) and (B_3 && B_4)?
  - Short-circuiting: *if (x < 10 && y < 20) ...*
  - **Classwork:** Write a C program to find out if C uses short-circuiting or not.
    - *if (p && p->next) ...*

Control-Flow – Boolean Expressions

- Source code:
  - if (x < 100 || x > 200 && x != y) x = 0;
- IR:

without short-circuit

```
    b1 = x < 100
    b2 = x > 200
    b3 = x != y
    iftrue b1 goto L2
    iffalse b2 goto L3
    iffalse b3 goto L3
L2:
    x = 0;
L3:
    ...
```

with short-circuit

```
    b1 = x < 100
    iftrue b1 goto L2
    b2 = x > 200
    iffalse b2 goto L3
    b3 = x != y
    iffalse b3 goto L3
L2:
    x = 0;
L3:
    ...
```

3AC for Boolean Expressions

$B \rightarrow B_1 \,||\, B_2$

```
B1.true = B1.true;
B1.false = newLabel();
B2.true = B.true;
B2.false = B.false;
B.code = B1.code +
         label(B1.false) +
         B2.code;
```

$B \rightarrow B_1 \,\&\&\, B_2$

```
B1.true = newLabel();
B1.false = B.false;
B2.true = B.true;
B2.false = B.false;
B.coee = B1.code +
         label(B1.true) +
         B2.code;
```

3AC for Boolean Expressions

B → !B$_1$

```
B1.true = B.false;
B1.false = B.true;
B.code = B1.code;
```

B → E$_1$ relop E$_2$

```
B.code = E1.code + E2.code +
         gen('if' E1.addr relop E2.addr
                 'goto' B.true) +
         gen('goto' B.false);
```

B → true

```
B.code = gen('goto' B.true);
```

B → false

```
B.code = gen('goto' B.false);
```

SDD for *while*

$S \rightarrow$ while (C) S_1

L1	= newLabel();
L2	= newLabel();
S_1.next	= L1;
C.false	= S.next;
C.true	= L2;
S.code	= “label” + L1 + C.code + ”label” + L2 + S_1.code **+** **gen('goto' L1);**

3AC for if / if-else

S → if (B) S_1

B.true = newLabel();
B.false = S_1.next = S.next;
S.code = B.code +
label(B.true) +
S_1.code;

S → if (B) S_1 else S_2

B.true = newLabel();
B.false = newLabel();
S_1.next = S_2.next = S.next;
S.code = B.code +
label(B.true) + S_1.code +
Gen('goto' S.next) +
label(B.false) + S_2.code;

Control-Flow – Boolean Expressions

- Source code: if (x < 100 || x > 200 && x != y) x = 0;

without optimization

```
      b1 = x < 100
      b2 = x > 200
      b3 = x != y
      iftrue b1 goto L2
      goto L0
L0:
      iftrue b2 goto L1
      goto L3
L1:
      iftrue b3 goto L2
      goto L3
L2:
      x = 0;
L3:
      ...
```

with short-circuit

```
      b1 = x < 100
      iftrue b1 goto L2
      b2 = x > 200
      iffalse b2 goto L3
      b3 = x != y
      iffalse b3 goto L3
L2:
      x = 0;
L3:
      ...
```

Avoids redundant gotos.

Homework

- Write SDD to generate 3AC for *for*.
  - *for (S1; B; S2) S3*
- Write SDD to generate 3AC for *repeat-until.*
  - *repeat S until B*

Backpatching

- *if (B) S* required us to pass label while evaluating B.
  - This can be done by using inherited attributes.
- Alternatively, we could leave the label unspecified now...
  - … and fill it in later.
- Backpatching is a general concept for one-pass code generation
  - Recall stack offset computation in A3.

B → true

```
B.code = gen('goto –');
```

B → B_1 || B_2

```
backpatch(B1.false);
...
```

break and continue

- break and continue are disciplined / special gotos.
- Their IR needs
  - currently enclosing loop / switch.
  - goto to a label just outside / before the enclosing block.
- How to write the SDD to generate their 3AC?
  - either pass on the enclosing block and label as an inherited attribute, or
  - use backpatching to fill-in the label of goto.
  - Need additional restriction for *continue*.
- **Classwork**: How to support *break label*?

IR for switch

- Using nested if-else
- Using a table of pairs
  - $<V_i, S_i>$
- Using a hash-table
  - when i is large (say, > 10)
- Special case when V_is are consecutive integrals.
  - Indexed array is sufficient.

```
switch(E) {
    case V1: S1
    case V2: S2
    …
    case Vn-1: Sn-1
    default: Sn
}
```

t = code for E
goto test
L_1: code for S1
goto next
L2: code for S_2
goto next
…
L_{n-1}: code for S_{n-1}
goto next
L_n: code for S_n
goto next
test:
    if t = V_1 goto L_1
    if t = V_2 goto L_2
    …
    if t = V_{n-1} goto L_{n-1}
    goto L_n
next:

t = code for E
if t != V_1 goto L_1
code for S_1
goto next
L_1: if t != V_2 goto L_2
code for S_2
goto next
L_2:
…
L_{n-2}: if t != V_{n-1} goto L_{n-1}
code for S_{n-1}
goto next
L_{n-1}: code for S_n
next:

switch(E) {
    case V_1: S_1
    case V_2: S_2
    …
    case V_{n-1}: S_{n-1}
    default: S_n
}

Functions

- Function definitions
  - Type checking / symbol table entry
  - Return type, argument types, void
  - Stack offset for variables
  - Stack offset for arguments
- Function calls
  - Push parameters
  - Switch scope / push environment
  - Jump to label for the function
  - Switch scope / pop environment
  - Pop parameters

Language Constructs

- Declarations
  - Types (int, int [], struct, int *)
  - Storage qualifiers (array expressions, const, static)
- Assignments
- Conditionals, switch
- Loops
- Function calls, definitions