CSD 3202-COMPILER DESIGN

MODULE III

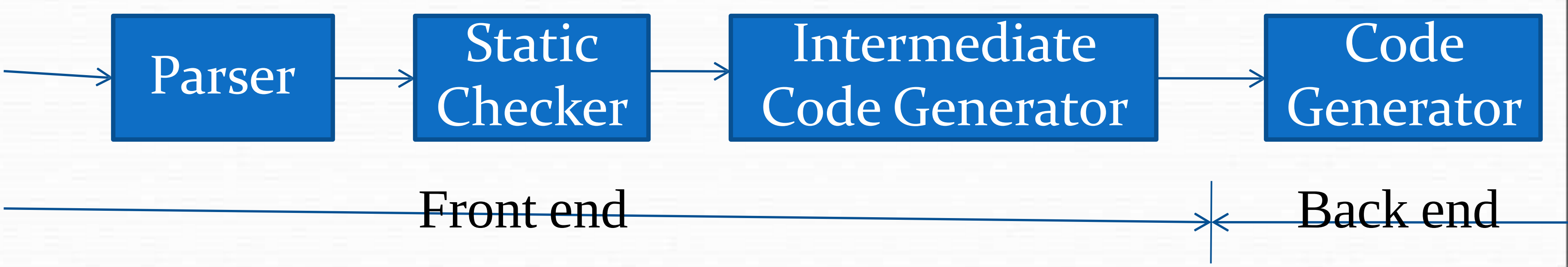
PART-2

Outline

- Variants of Syntax Trees
- Three-Address Code
- Types and Declarations
- Translation of Expressions
- Type Checking

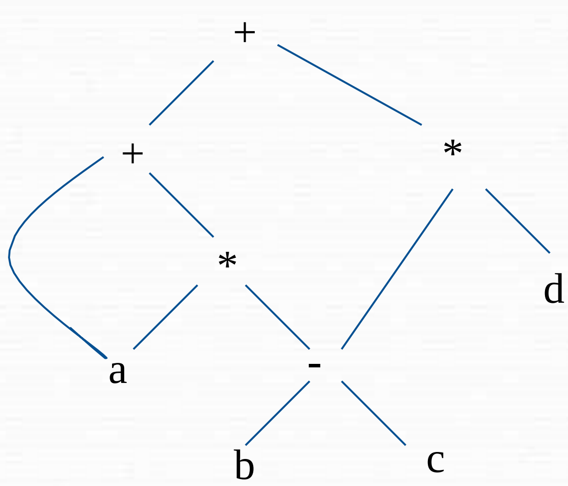
Introduction

- Intermediate code is the interface between front end and back end in a compiler
- Ideally the details of source language are confined to the front end and the details of target machines to the back end (a m*n model)
- In this chapter we study intermediate representations, static type checking and intermediate code generation



Variants of syntax trees

- It is sometimes beneficial to crate a DAG instead of tree for Expressions.
- This way we can easily show the common sub-expressions and then use that knowledge during code generation
- Example: a+a*(b-c)+(b-c)*d



SDD for creating DAG's

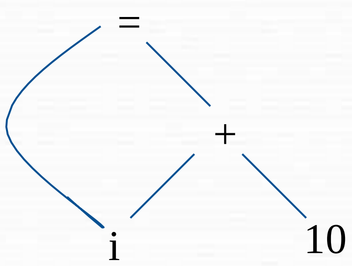
Production	Semantic Rules
1) E -> E1+T	E.node= new Node('+', E1.node,T.node)
2) E -> E1-T	E.node= new Node('-', E1.node,T.node)
3) E -> T	E.node = T.node
4) T -> (E)	T.node = E.node
5) T -> id	T.node = new Leaf(id, id.entry)
6) T -> num	T.node = new Leaf(num, num.val)

Example:

1) p1=Leaf(id, entry-a)
2) P2=Leaf(id, entry-a)=p1
3) p3=Leaf(id, entry-b)
4)p4=Leaf(id, entry-c) 5)
p5=Node('-',p3,p4) 6)
p6=Node('*',p1,p5) 7)
p7=Node('+',p1,p6)

8) p8=Leaf(id,entry-b)=p3
9) p9=Leaf(id,entry-c)=p4
10) p10=Node('-',p3,p4)=p5
11) p11=Leaf(id,entry-d)
12) p12=Node('*',p5,p11)
13) p13=Node('+',p7,p12)

Value-number method for constructing DAG's



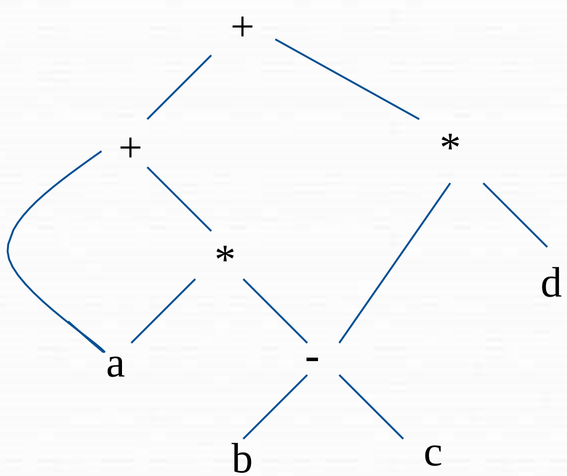
id		
num	10	
+	1	2
3	1	3

To entry for i

- Algorithm
  - Search the array for a node M with label op, left child l and right child r
  - If there is such a node, return the value number M
  - If not create in the array a new node N with label op, left child l, and right child r and return its value
- We may use a hash table

Three address code

- In a three address code there is at most one operator at the right side of an instruction
- Example:



t1 = b – c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4

Forms of three address instructions

- x = y op z
- x = op y
- x = y
- goto L
- if x goto L and ifFalse x goto L
- if x relop y goto L
- Procedure calls using:
  - param x
  - call p,n
  - y = call p,n
- x = y[i] and x[i] = y
- x = &y and x = *y and *x =y

Example

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

```
L:    t1 = i + 1
      i = t1
      t2 = i * 8
      t3 = a[t2]
      if t3 < v goto L
```

Symbolic labels

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

Position numbers

Data structures for three address codes

- Quadruples
  - Has four fields: op, arg1, arg2 and result
- Triples
  - Temporaries are not used and instead references to instructions are made
- Indirect triples
  - In addition to triples we use a list of pointers to triples

Example

- b * minus c + b * minus c

Three address code

t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5

Quadruples

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

Triples

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

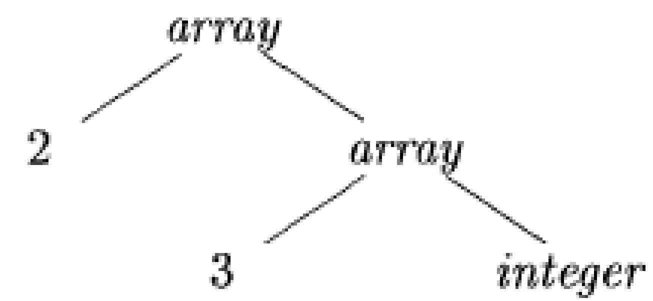
Indirect Triples

	op
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)

Type Expressions

Example: int[2][3]
array(2,array(3,integer))



- A basic type is a type expression
- A type name is a type expression
- A type expression can be formed by applying the array type constructor to a number and a type expression.
- A record is a data structure with named field
- A type expression can be formed by using the type constructor for function types
- If s and t are type expressions, then their Cartesian product s*t is a type expression
- Type expressions may contain variables whose values are type expressions

Type Equivalence

- They are the same basic type.
- They are formed by applying the same constructor to structurally equivalent types.
- One is a type name that denotes the other.

Declarations

$$
\begin{array}{lcl}
D & \rightarrow & T \ \mathbf{id} \ ; \ D \ | \ \epsilon \\
T & \rightarrow & B \ C \ | \ \mathbf{record} \ '\{' \ D \ '\}' \\
B & \rightarrow & \mathbf{int} \ | \ \mathbf{float} \\
C & \rightarrow & \epsilon \ | \ [\ \mathbf{num} \ ] \ C
\end{array}
$$

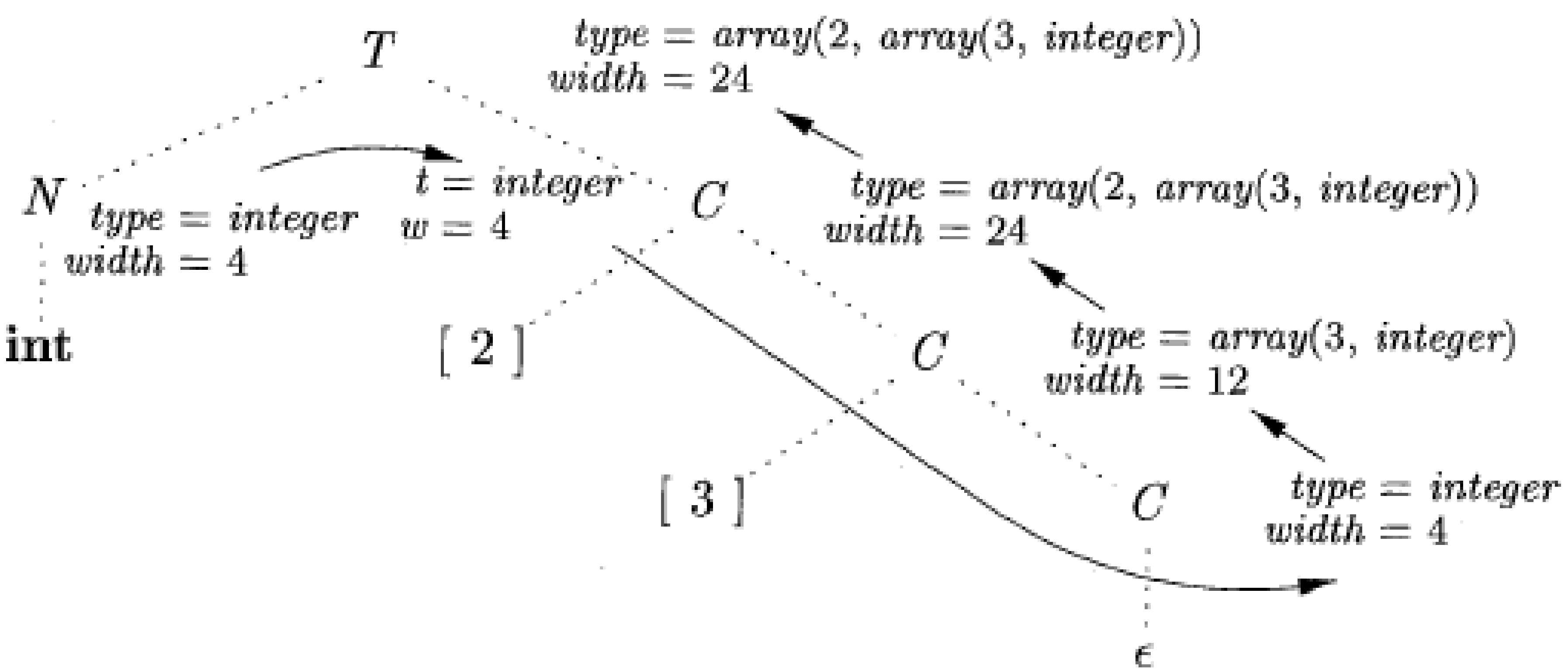
Storage Layout for Local Names

- Computing types and their widths

$$
\begin{array}{lcll}
T & \rightarrow & B & \{\ t = B.type;\ w = B.width;\ \} \\
 & & C & \\
B & \rightarrow & \textbf{int} & \{\ B.type = integer;\ B.width = 4;\ \} \\
B & \rightarrow & \textbf{float} & \{\ B.type = float;\ B.width = 8;\ \} \\
C & \rightarrow & \epsilon & \{\ C.type = t;\ C.width = w;\ \} \\
C & \rightarrow & [\ \textbf{num}\ ]\ C_1 & \{\ array(\textbf{num}.value,\ C_1.type); \\
 & & & \quad C.width = \textbf{num}.value \times C_1.width;\ \}
\end{array}
$$

Storage Layout for Local Names

- Syntax-directed translation of array types



Sequences of Declarations

- $$\begin{array}{lll} P & \rightarrow & \{\ offset = 0;\ \} \\ & & D \\ D & \rightarrow & T\ \textbf{id}\ ; \quad \{\ top.put(\textbf{id}.lexeme,\ T.type,\ offset); \\ & & \quad offset = offset + T.width;\ \} \\ & & D_1 \\ D & \rightarrow & \epsilon \end{array}$$
- Actions at the end:
- $$\begin{array}{lll} P & \rightarrow & M\ D \\ M & \rightarrow & \epsilon \quad \{\ offset = 0;\ \} \end{array}$$

Fields in Records and Classes

- ```
  float x;
  record { float x; float y; } p;
  record { int tag; float x; float y; } q;
  ```

- $$
\begin{array}{llll}
T & \rightarrow & \textbf{record}\ '\{' & \{\ Env.push(top);\ top = \textbf{new}\ Env(); \\
 & & & \quad Stack.push(offset);\ offset = 0;\ \} \\
 & & D\ '\}' & \{\ T.type = record(top);\ T.width = offset; \\
 & & & \quad top = Env.pop();\ offset = Stack.pop();\ \}
\end{array}
$$
  ```

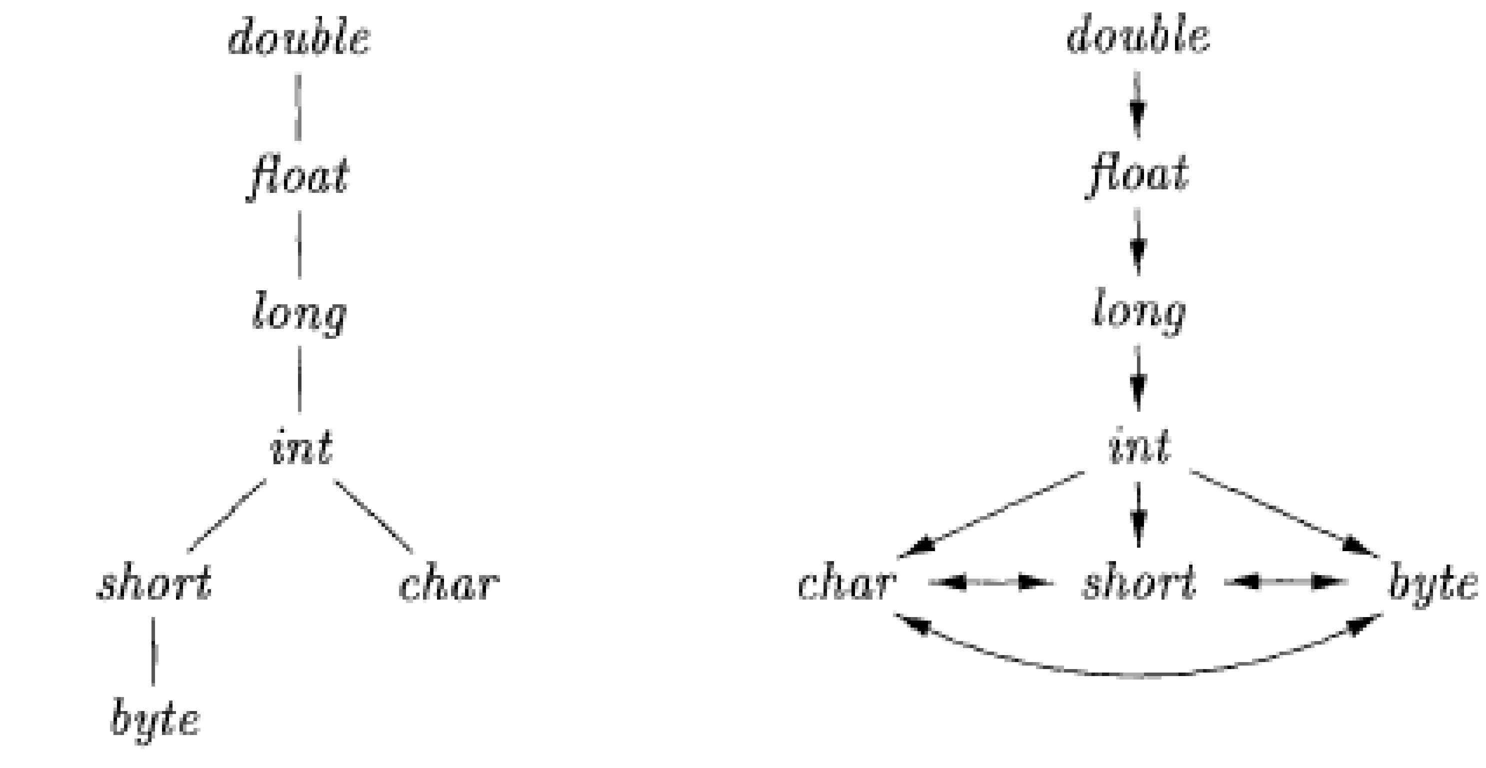
Translation of Expressions and Statements

- We discussed how to find the types and offset of variables
- We have therefore necessary preparations to discuss about translation to intermediate code
- We also discuss the type checking

Three-address code for expressions

PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E \ ;$	$S.code = E.code \ \Vert$ $gen(top.get(\textbf{id}.lexeme) \ '=' \ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new} \ Temp()$ $E.code = E_1.code \ \Vert \ E_2.code \ \Vert$ $gen(E.addr \ '=' \ E_1.addr \ '+' \ E_2.addr)$
$\ \ \mid \ - E_1$	$E.addr = \textbf{new} \ Temp()$ $E.code = E_1.code \ \Vert$ $gen(E.addr \ '=' \ '\textbf{minus}' \ E_1.addr)$
$\ \ \mid \ (\ E_1 \ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\ \ \mid \ \textbf{id}$	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

Conversions between primitive types in Java



(a) Widening conversions (b) Narrowing conversions

Introducing type conversions into expression evaluation

$$
\begin{array}{lll}
E \;\rightarrow\; E_1 + E_2 & \{ & E.type \;=\; max(E_1.type, E_2.type); \\
 & & a_1 \;=\; widen(E_1.addr, E_1.type, E.type); \\
 & & a_2 \;=\; widen(E_2.addr, E_2.type, E.type); \\
 & & E.addr = \mathbf{new}\ Temp\,(); \\
 & & gen(E.addr\ '='\ a_1\ '+'\ a_2);\ \}
\end{array}
$$