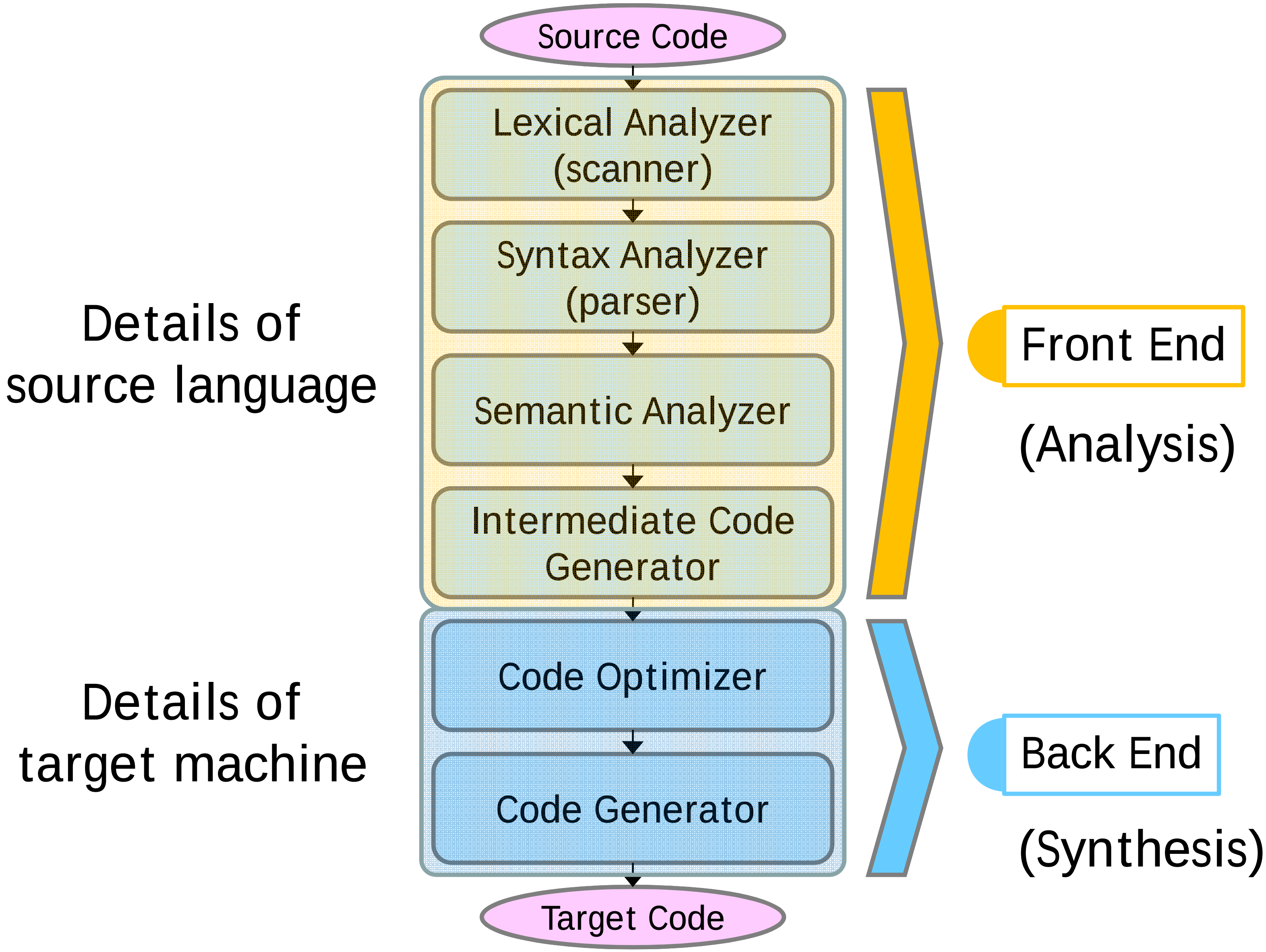
Introduction to Compiler Design

Intermediate-Code Generation

Professor Yi-Ping You
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

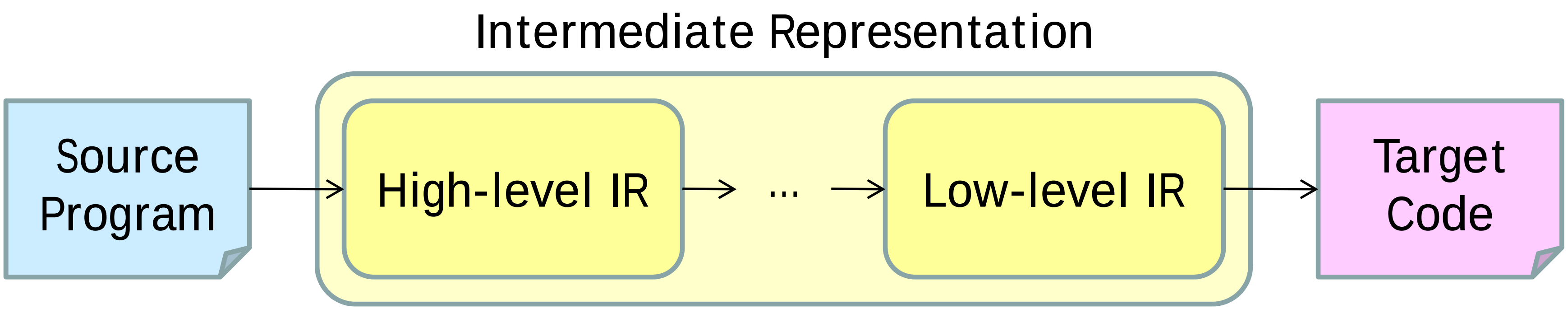
Recall: The Structure of a Compiler



Outline

- Overview of Intermediate Representation
- Intermediate Representation
  - Syntax Trees
  - Three-Address Code
- Intermediate-Code Generation
  - Types and Declarations
    - Translation of Expressions
    - Type Checking
  - Control Flow
  - Backpatching
  - Switch-Statements and Procedures

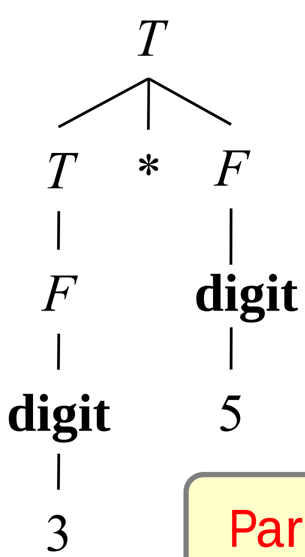
A Sequence of Intermediate Representations



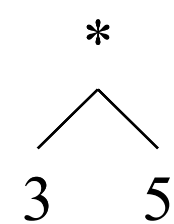
- High-level representations
  - Syntax trees (graphical representations)
  - Closer to source language
  - Suitable for static type checking
- Low-level representations
  - Three-address code
  - Closer to target machine
  - Suitable for machine-dependent tasks like register allocation and instruction selection

Recall: Syntax Trees

- A syntax tree shows the structure of a program by abstracting away irrelevant details from a parse tree
  - Each node represents a computation to be performed
  - The children of the node represents what that computation is performed on



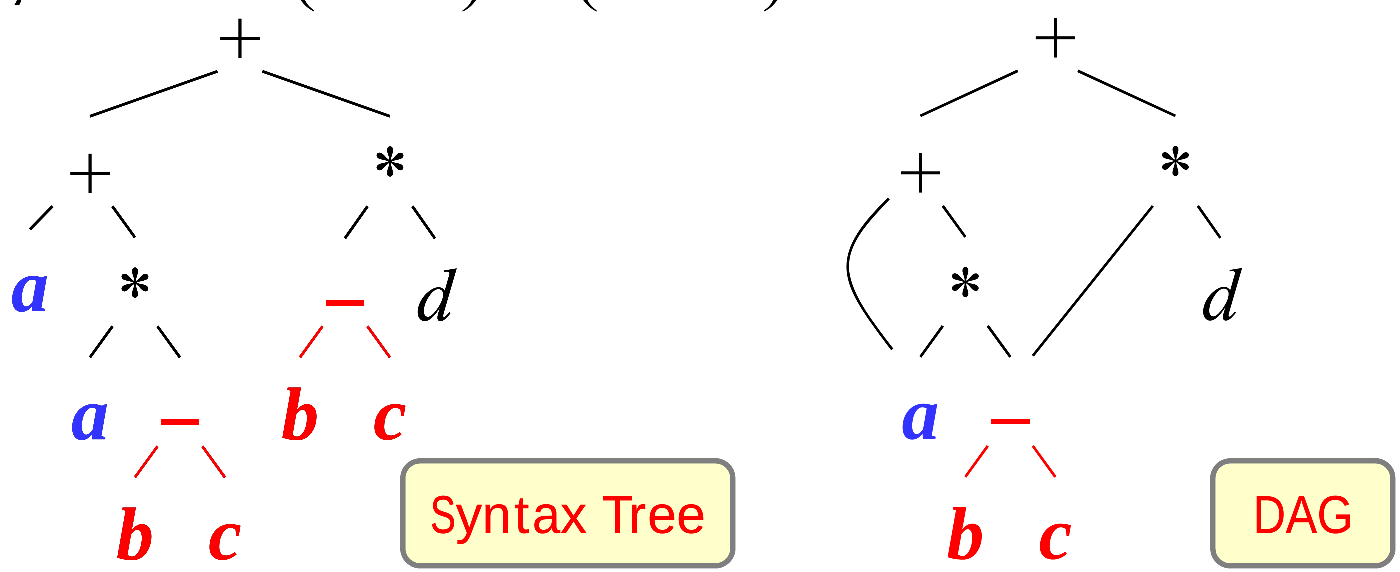
Parse Tree



Syntax Tree

Variant of Syntax Trees

- Directed acyclic graph (DAG)
  - Commons to syntax trees
    - Leaves: atomic operands
    - Interior nodes: operators
  - Differences to syntax trees
    - Common subexpressions are not replicated
- E.g., $a + a * (b - c) + (b - c) * d$



Construction of DAG: An Example

Production	Semantic Rules
1) $E \rightarrow E_1 + T$	$E.node$ = **new** $Node('+', E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node$ = **new** $Node('-', E_1.node, T.node)$
3) $E \rightarrow E_1 * T$	$E.node$ = **new** $Node('*', E_1.node, T.node)$
4) $E \rightarrow T$	$E.node = T.node$
5) $T \rightarrow (E)$	$T.node = E.node$
6) $T \rightarrow$ **id**	$T.node$ = **new** *Leaf(***id**, **id***.entry)*
7) $T \rightarrow$ **num**	$T.node$ = **new** *Leaf(***num**, **num***.entry)*

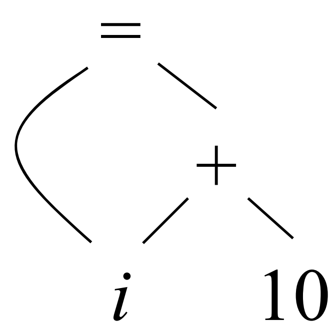
Before creating a new node, we check whether the node exists

- Steps for constructing the DAG for $a + a * (b - c) + (b - c) * d$
  - p1=Leaf(id, entry-a)
  - p2=Leaf(id, entry-a)=p1
  - p3=Leaf(id, entry-b)
  - p4=Leaf(id, entry-c)
  - p5=Node('-',p3,p4)
  - p6=Node('*',p1,p5)
  - p7=Node('+',p1,p6)
  - p8=Leaf(id,entry-b)=p3
  - p9=Leaf(id,entry-c)=p4
  - p10=Node('-',p3,p4)=p5
  - ...

Value-Number Method for Constructing DAG's

- Nodes of syntax trees or DAG's are stored in an array of records

DAG



Array

	op		
1	**id**		to entry for *i*
2	**num**	10	
3	+	1	2
4	=	1	3
5	…	*l*	*r*

  - The integer index of the record: value number
  - Signature of an interior node: $<op, l, r>$
    - op: label
    - l : left child's value number
    - r : right child's value number (0 for unary operators)

Algorithm for Value-Number Method

- Input
  - Label op, node l, and node r
- Output
  - The value number of a node in the array with signature $<op, l, r>$
- Method
  - 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 of M
  - Create a new node if not found, and return its value number
- For efficiency, we use a hash table to implement the array structure

Outline

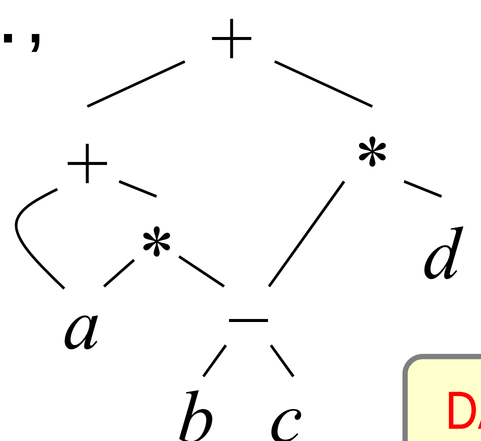
- Overview of Intermediate Representation
- Intermediate Representation
  - Syntax Trees
  - Three-Address Code
- Intermediate-Code Generation
  - Types and Declarations
    - Translation of Expressions
    - Type Checking
  - Control Flow
  - Backpatching
  - Switch-Statements and Procedures

Three-Address Code

- In three-address code, there is at most one operator on the right side of an instruction
  - At most three address (operands)
    - One destination operand
    - Two source operands
- E.g., three-address code for $x + y * z$: $t_1 = y * z$

$$t_2 = x + t_1$$

- E.g.,



DAG

$$\begin{aligned} t_1 &= b - c \\ t_2 &= a * t_1 \\ t_3 &= a + t_2 \\ t_4 &= t_1 * d \\ t_5 &= t_3 + t_4 \end{aligned}$$

Three-address code

Three-Address Code: Addresses and Instructions

- Three-address code is built from two concepts
  - Addresses
  - Instructions
- Three-address code can be implemented using records (data structures or objects) with fields for operations and addresses
  - E.g., quadruples or triples
- Addresses can be
  - A name
    - source-program name stored in the symbol table
  - A constant
  - A compiler-generated temporary

Three-Address Code: Instructions

- Symbolic labels will be used by instructions that alter the flow of control
  - A symbolic label represents the index of a three-address instruction in the sequence of instructions
- 8 common three-address instructions forms:
  - Assignment instructions with a binary operation
    - $x = y \; op \; z$
  - Assignment instructions with a unary operation
    - $x = op \; y$
  - Copy instructions
    - $x = y$

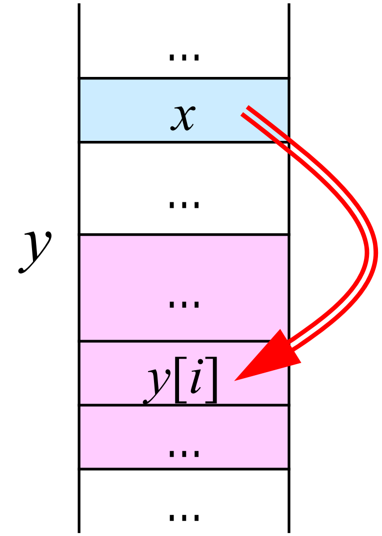
Three-Address Code: Instructions (Cont'd)

- Unconditional jumps
  - `goto` L (L: a label)
- Conditional jumps
  - `if` x `goto` L or `ifFalse` x `goto` L
  - `if` x *relop* y `goto` L
    - Relational operators: $<$, $==$, $>=$, etc.
- Procedure calls and returns
  - $p(x_1, x_2, \ldots, x_n)$
  - `param` x_1
`param` x_2
…
`param` x_n
`call` p, n

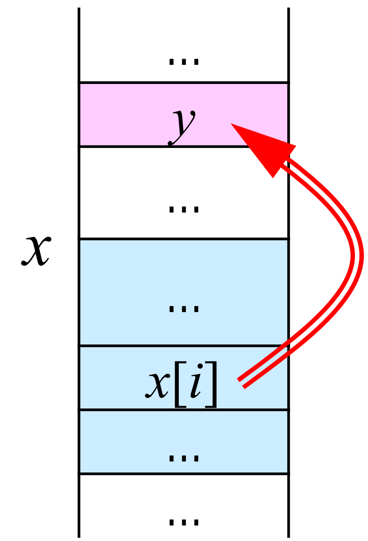
- $f(x_1, p(x_2))$
- `param` x_1
`param` x_2
`param call` $p, 1$
`call` $f, 2$

Three-Address Code: Instructions (Cont'd)

- Indexed copy instructions
  - $x = y[i]$
    - Sets x to the value in the location i memory unit beyond location y

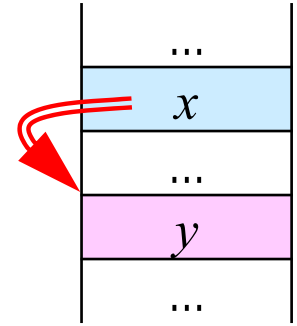


  - $x[i] = y$
    - Sets the contents of the location i units beyond x to the value of y

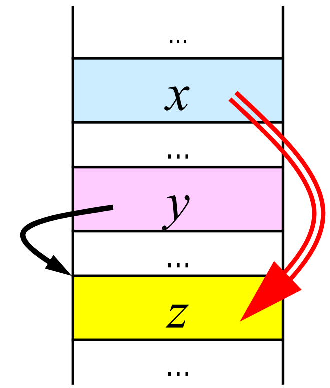


Three-Address Code: Instructions (Cont'd)

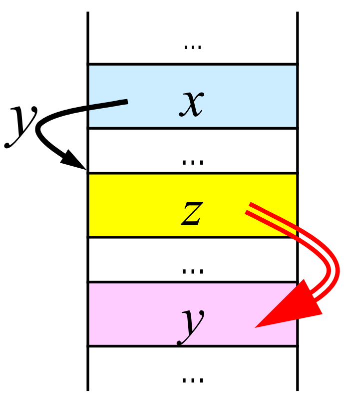
- Address and pointer assignments
  - $x = \&y$
    - – y is a name or temporary
    - – x is a pointer or temporary
    - – Sets the r-value of x to be the location (l-value) of y



  - $x = *y$
    - – y is a pointer or temporary
    - – Sets the r-value of x to be the r-value of the object z pointed by of y



  - $*x = y$
    - – Sets the r-value of the object z pointed by x to be the r-value of y



Three-Address Code: An Example

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

- Two possible three-address code
  - Using symbolic labels
  - Using position numbers

L: t_1 = i + 1
   i = t_1
   t_2 = i * 8
   t_3 = a [t_2]
   if t_3 < v goto L

100: t_1 = i + 1
101: i = t_1
102: t_2 = i * 8
103: t_3 = a [t_2]
104: if t_3 < v goto 100

- Each subexpression typically get its own, new temporary to hold its result, and we learn where to put the value only when the assignment operator is processed
- Suppose each element of the array take 8 units of space

Notes on Three-Address Code

- The choice of allowable operators is important in the design of the IR
- Allow operators that are close to machine instructions
  - Easier to implement the IR on a target machine
  - However, if the front end must generate long sequences of instructions for some source-language operations
    - The optimizer and code generator may have to work harder to
      - Rediscover the structure
      - Generate good code for these operations

Data Structures for Three-Address Code

- Quadruples (or just "quad")
  - Has four fields: op, arg_1, arg_2, and $result$
  - Temporary names are used explicitly
- Triples
  - Has only three fields: op, arg_1, and arg_2
  - Temporaries are not used and instead references to instructions are made (by positions)
- Indirect triples
  - In addition to triples we use a list of pointers to triples

Data Structures for Three-Address Code: Examples

- Example: $a = b * -c + b * -c$

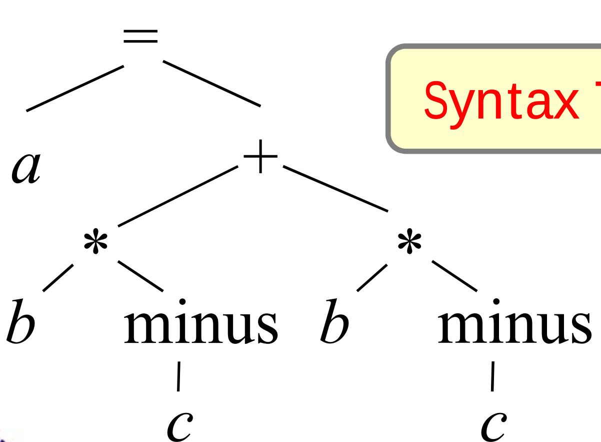
```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5
```

Three-Address Code

Point to symbol-table entry

	op	*arg*$_1$	*arg*$_2$	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		…		

Quads



Syntax Tree

Value number

	op	*arg*$_1$	*arg*$_2$
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		…	

signature

Triples

Quadruples v.s. Triples

- A benefit of quadruples over triples can be seen in an optimizing compiler
  - Instructions are often moved around
  - With quadruples, we can move an instruction without changing the instructions
- With triples, moving an instruction may require us to change all references to that result
  - Use indirect triples to solve the problem

Indirect Triples

pointer

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

	op	*arg_1*	*arg_2*
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		…	

- An optimizing compiler can move an instruction by reordering the *instruction* list, without affecting the triples themselves

Static Single-Assignment (SSA) Form

- A variant of three-address code
- All assignments are to variables with distinct names
  - Each variable is only defined once
  - Hence the term static single-assignment
  - E.g.,

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

$$\begin{aligned} p_1 &= a + b \\ q_1 &= p_1 - c \\ p_2 &= q_1 * d \\ p_3 &= e - p_2 \\ q_2 &= p_3 + q_1 \end{aligned}$$

SSA Form

  - E.g.,

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

$$\begin{aligned} &\texttt{if (flag) } x_1 = -1; \\ &\texttt{else } x_2 = 1; \\ &x_3 = \phi(x_1, x_2); \\ &y = x_3 * a; \end{aligned}$$

- Facilitates certain code optimizations

Outline

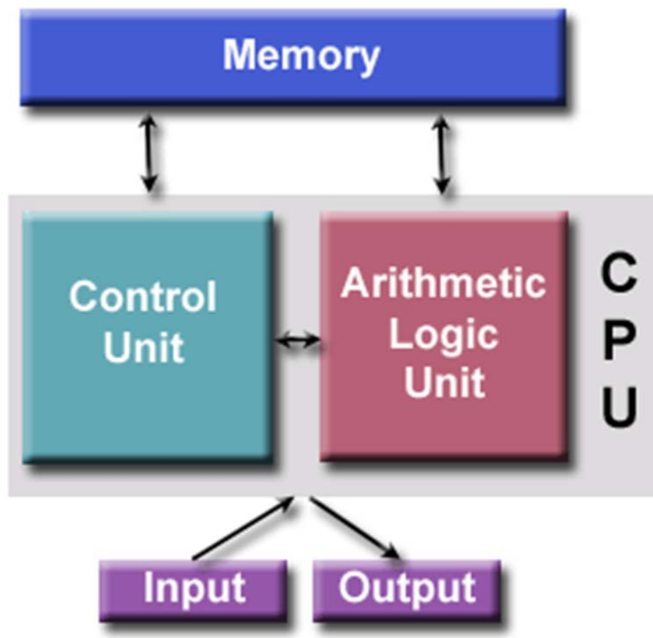
- Overview of Intermediate Representation
- Intermediate Representation
  - Syntax Trees
  - Three-Address Code
- Intermediate-Code Generation
  - Types and Declarations
    - Translation of Expressions
    - Type Checking
  - Control Flow
  - Backpatching
  - Switch-Statements and Procedures

Types and Declarations

- Two important things about types and declarations in the intermediate-code generation
  - Type checking
    - Ensures that the types of the operands match the type expected by an operator
    - E.g., the **and** operator in *P* language expects its two operands to be *bool*
  - The storage layout for names
    - A compiler determines the storage that will be needed for that name at run time
    - Relative address of the name in the memory

Recall: von Neumann Architecture

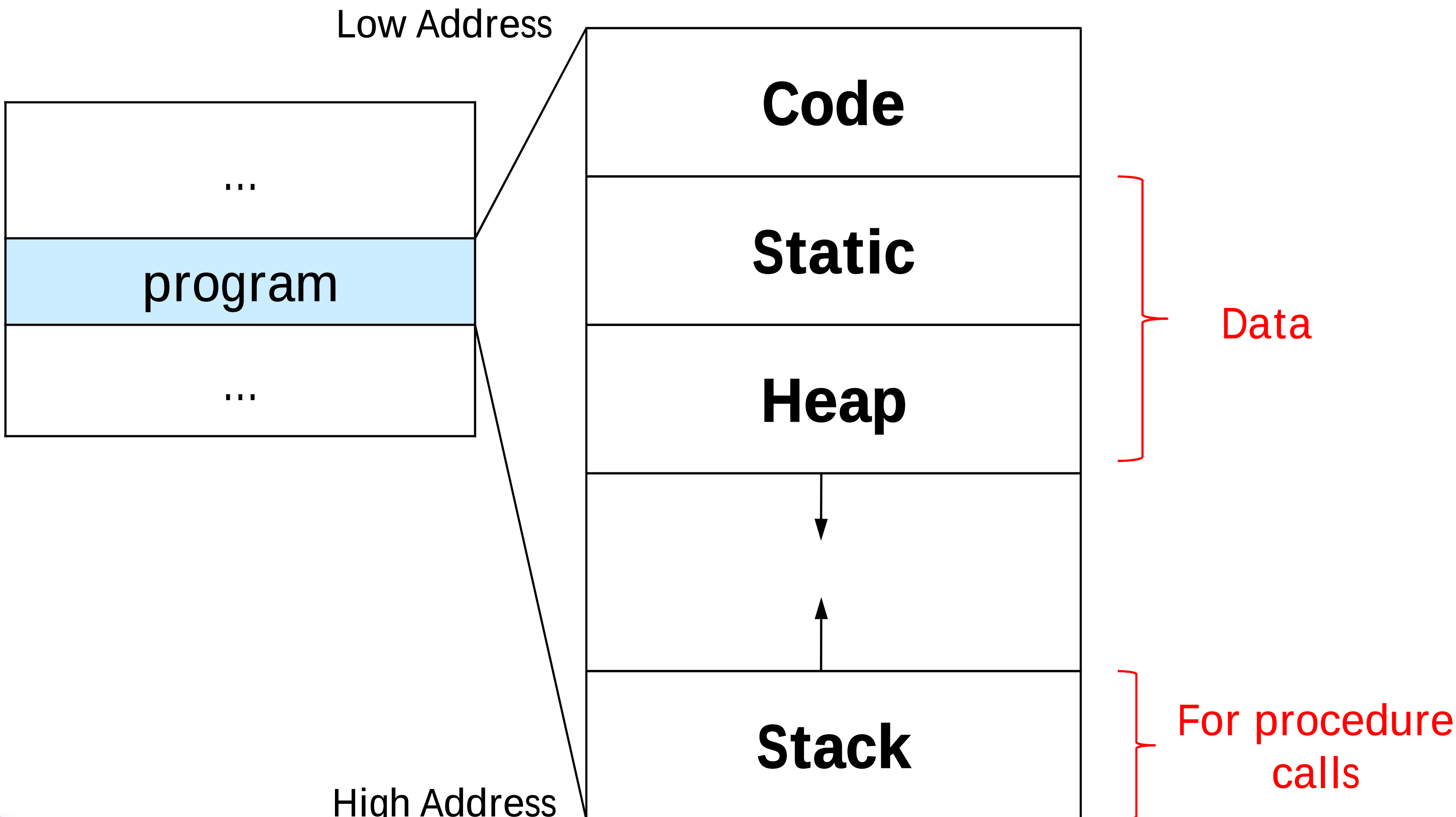
- Imperative languages are abstractions of von Neumann architecture
  - Memory
  - Processor



John von Neumann
(1903-1957)

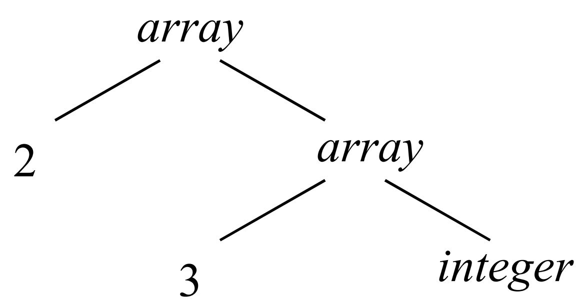
Recall: Storage Layout

- Typical memory layout of a program



Type Expressions

- Types have structure
  - We shall represent them using type expressions
    - Basic types
    - Formed by applying an operator, type constructor, to a type expression
  - E.g., the array type `int[2][3]` can be read as "array of 2 arrays of 3 integers each" and written as a type expression $array(2, array(3, integer))$



DAG

Definitions of Type Expressions

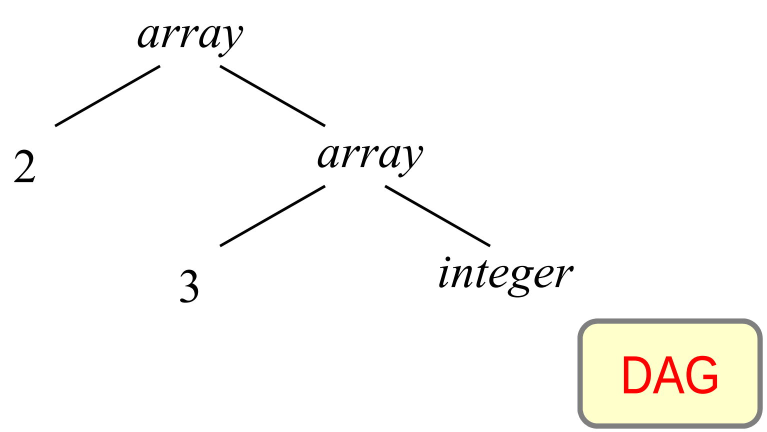
- A basic type is a type expression
  - E.g., *boolean*, *char*, *integer*, *float*, *void*, ...
- A type name is a type expression
  - E.g., `typedef (int *[5]) newtype`
    (an array of 5 pointers to *int*)
- A type expression can be formed by applying the *array* type constructor to a number and a type expression
  - E.g., *array*(3, *integer*)
- A record is a data structure with named fields
  - E.g., structures in C

Definitions of Type Expressions (Cont'd)

- A type expression can be formed by using the type constructor $\rightarrow$ for function types
  - E.g., $s \rightarrow t$ (function from type s to type t)
- If s and t are type expressions, then their Cartesian products $s \times t$ is a type expression
  - Represents a list of types
  - Left associative, higher precedence than $\rightarrow$
- Type expressions may contain variables whose values are type expressions
  - Compiler-generated type variables

Type Expressions (Cont'd)

- A convenient way to represent a type expression is to use a graph
- E.g., the array type `int[2][3]`



- Interior nodes
  - Type constructors
- Leaves
  - Basic types, type names, and type variables

Type Equivalence

- When type expressions are represented by graphs, two types are structurally equivalent iff
  - They are the same basic type, or
  - They are formed by applying the same type constructor to structurally equivalent types, or
    - E.g., `int A[10];`
      `int B[10];`
  - One is a type name that denotes the other
    - E.g., `typedef int dollars;`
- If type names are treated as standing for themselves, two types are name equivalent iff one of the first two conditions above is true

Name Declarations

- A simplified grammar that declares just one name at a time

$$
\begin{aligned}
D &\rightarrow T\ \textbf{id}\ \textbf{;}\ D \mid \varepsilon \\
T &\rightarrow B\ C \mid \textbf{record}\ \text{'}\textbf{\{}\text{'}\ D\ \text{'}\textbf{\}}\text{'} \\
B &\rightarrow \textbf{int} \mid \textbf{float} \\
C &\rightarrow \varepsilon \mid [\ \textbf{num}\ ]\ C
\end{aligned}
$$

- Legal inputs:

```
int a;
float[3][5] b;
record {int a; int b;} c;
```

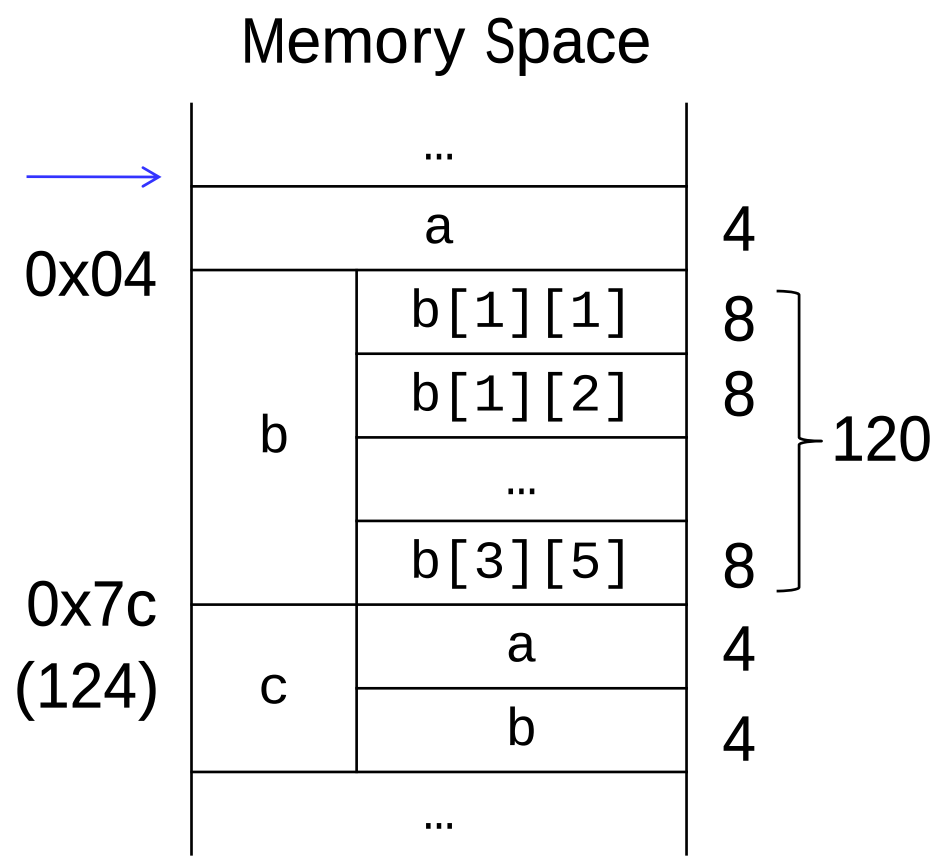
Storage Layout for Names

- Suppose that storage comes in blocks of contiguous bytes
- From the type of a name, we can determine the amount of storage that will be needed for the name at run time
  - At compile time, we can use these amounts to assign each name a relative address (saved in the symbol table)
    - The width of a type is the number of storage units needed for objects of that type
    - E.g., width of integer: 4 (bytes)
      width of float: 8 (bytes)

Storage Layout: An Example

```
int a;
float[3][5] b;
record {int a; int b;} c;
```

Start address of local names

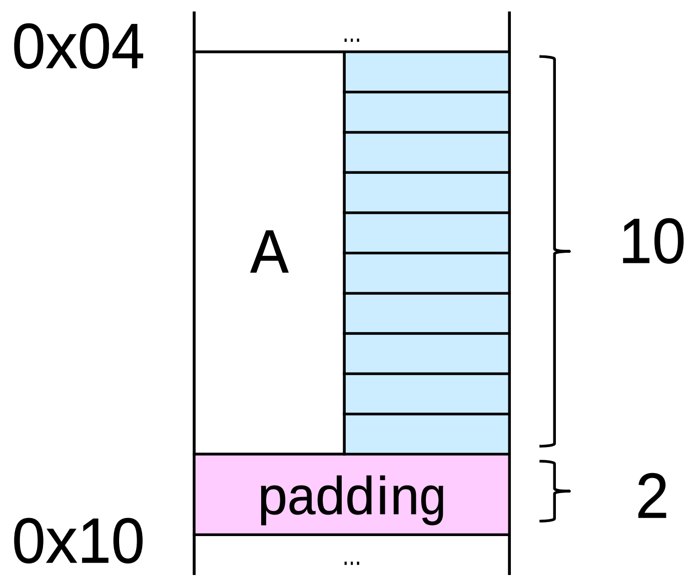


Symbol Table

name	...	location
a	...	0x00
b	...	0x04
c	...	0x7c

Address Alignment

- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine
  - Data should be aligned (e.g., divisible by 4 bytes)
  - E.g., suppose the width of a character is 1 byte. For an array A of 10 characters, 12 bytes are allocated for the array



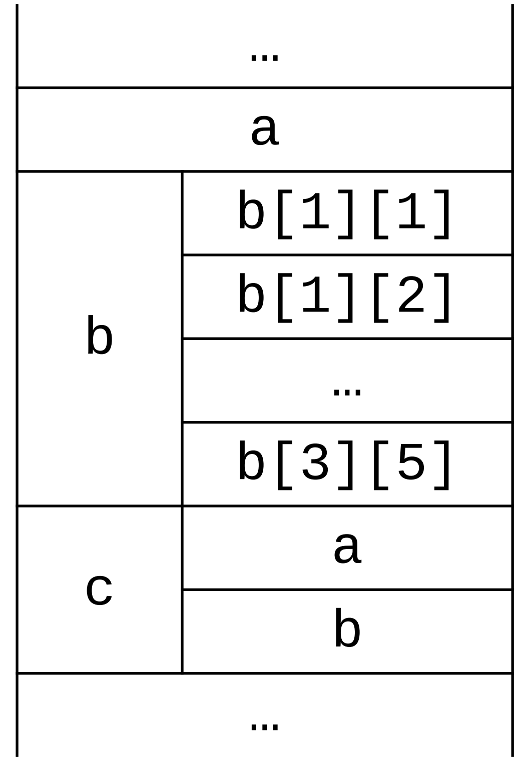
Preview: Code Generation

```
int a;
float[3][5] b;
record {int a; int b;} c;
…
a = c.b;
```

…

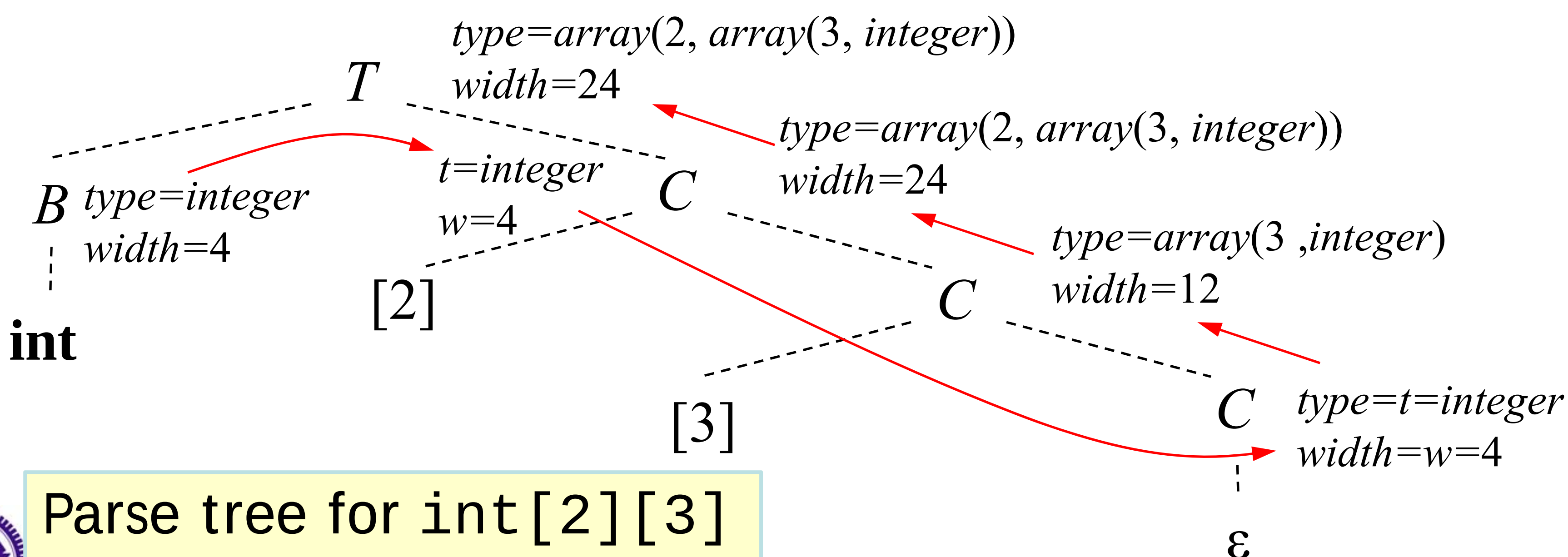
- Load the value of `c.b` from memory to a temporary (e.g,. a register);
- Store the value of the temporary to the address of a;

Memory Space



SDT: Computing Types and Their Width

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



Parse tree for `int[2][3]`

SDT: Computing the Relative Addresses

- SDT:

$$
\begin{array}{ll}
P \rightarrow & \{offset = 0;\} \\
\quad D & \\
D \rightarrow T\ \mathbf{id}\ ; & \{top.put(\mathbf{id}.lexeme,\ T.type,\ offset); \\
 & \ offset = offset + T.width;\} \\
\quad D_1 & \\
D \rightarrow \varepsilon &
\end{array}
$$

Current symbol table

- E.g.,

```
{
    int a;
    float b;
    {
        float a;
        …
    }
    …
}
```

name	type	...	location
a	float	...	0x00

name	type	...	location
a	int	...	0x00
b	float	...	0x04

SDT: Handling of Field Names in Records

- SDT:

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

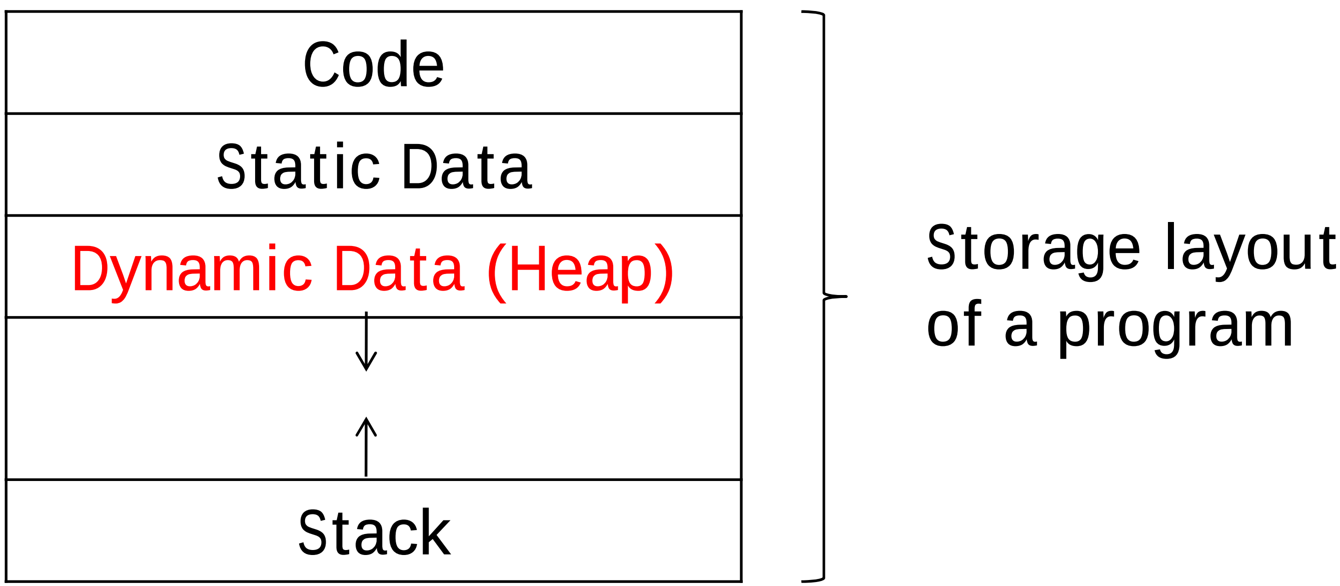
- E.g.,

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

<table>
<tr><th>name</th><th>type</th><th>record fields</th><th>...</th><th>location</th></tr>
<tr><td>x</td><td>float</td><td>N/A</td><td>...</td><td>0x00</td></tr>
<tr><td>p</td><td>record</td><td><table><tr><th>name</th><th>type</th><th>...</th><th>location</th></tr><tr><td>x</td><td>float</td><td>...</td><td>0x00</td></tr><tr><td>y</td><td>float</td><td>...</td><td>0x08</td></tr></table></td><td>...</td><td>0x08</td></tr>
<tr><td>q</td><td>int</td><td>N/A</td><td>...</td><td>0x18</td></tr>
</table>

Storage Layout for Names (Cont'd)

- Dynamic data is handled by reserving a known fixed amount of storage for a pointer to the data (will be discussed in Chapter 7)
  - Data of varying length, such as strings
  - Data whose size cannot be determined until run time, such as dynamic arrays



Outline

- Overview of Intermediate Representation
- Intermediate Representation
  - Syntax Trees
  - Three-Address Code
- Intermediate-Code Generation
  - Types and Declarations
    - Translation of Expressions
    - Type Checking
  - Control Flow
  - Backpatching
  - Switch-Statements and Procedures

SDD: Translation of Expressions

Production	Semantic Rules
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \parallel gen(top.get(\mathbf{id}.lexeme)\ `=`\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \parallel E_2.code \parallel gen(E.addr\ `=`\ E_1.addr\ `+`\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \parallel gen(E.addr\ `=`\ `\mathbf{minus}`\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = `\ `$

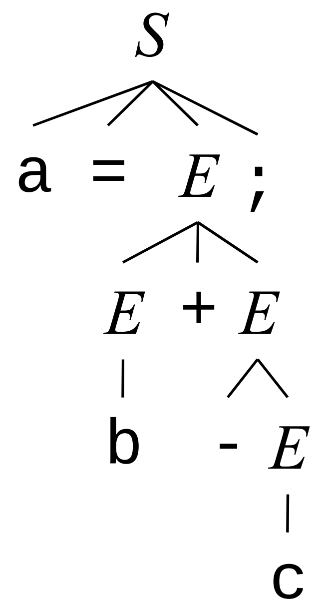
$gen(x\ `=`\ y\ `+`\ z)$: the three-address instruction $x = y + z$

- *code*: three-address code
- $E.addr$: the address that holds the value of E (a name, a constant, or a temporary)

Translation of Expressions: An Example

- An assignment statement: `a = b + -c;`
- Three-address code:

```
t1 = minus c
t2 = b + t1
a = t2
```



Production	Semantic Rules
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \parallel$ $gen(top.get(\textbf{id}.lexeme)\ \text{'='}\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \parallel E_2.code \parallel$ $gen(E.addr\ \text{'='}\ E_1.addr\ \text{'+'}\ E_2.addr)$
$\mid\ -E_1$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \parallel$ $gen(E.addr\ \text{'='}\ \text{'}\textbf{minus}\text{'}\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \textbf{id}$	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = \text{''}$

Incremental Translation

- On-the-fly code generation
- SDT:

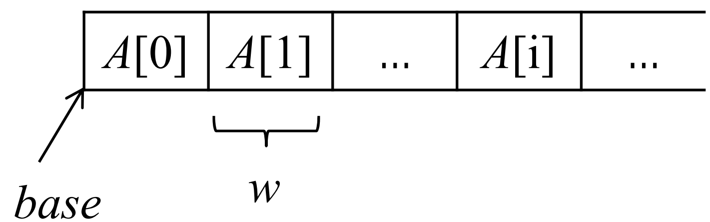
$$
\begin{array}{ll}
S \rightarrow \textbf{id} = E\ ; & \{\ gen(top.get(\textbf{id}.lexeme)\ `=`\ E.addr);\ \} \\
E \rightarrow E_1 + E_2 & \{\ E.addr = \textbf{new}\ Temp(); \\
 & \quad gen(E.addr\ `=`\ E_1.addr\ `+`\ E_2.addr);\ \} \\
\quad \mid\ -E_1 & \{\ E.addr = \textbf{new}\ Temp(); \\
 & \quad gen(E.addr\ `=`\ `\textbf{minus}`\ E_1.addr); \\
\quad \mid\ (\ E_1\ ) & \{\ E.addr = E_1.addr;\ \} \\
\quad \mid\ \textbf{id} & \{\ E.addr\ = top.get(\textbf{id}.lexeme);\ \}
\end{array}
$$

Addressing Array Elements

- Array elements can be accessed quickly if they are stored in a block of consecutive locations
  - In C, array elements are numbered $0, 1, \ldots, n\text{-}1$ for an array with n elements
- The i-th element of array A begins in location

$$base + i \times w$$

  - $base$: relative address of $A[0]$
  - w: width of each element



- For a two-dimension array (row-major), the relative address of $A[i_1][i_2]$ can be calculated by

$$base + i_1 \times w_1 + i_2 \times w_2$$

  - w_1: width of a row
  - w_2: width of an element

$A[0][0]$	$A[0][1]$	$A[0][2]$	$A[0][3]$	...
$A[1][0]$				
...				$A[i_1][i_2]$

Row-Major v.s. Column-Major Layouts

- Row major
  - Row-by-row

$A[1][1]$	$A[1][2]$	$A[1][3]$
$A[2][1]$	$A[2][2]$	$A[2][3]$

$A[1][1]$	First row
$A[1][2]$	
$A[1][3]$	
$A[2][1]$	Second row
$A[2][2]$	
$A[2][3]$	

- Column major
  - Column-by-column

$A[1][1]$	$A[1][2]$	$A[1][3]$
$A[2][1]$	$A[2][2]$	$A[2][3]$

$A[1][1]$	First column
$A[2][1]$	
$A[1][2]$	Second column
$A[2][2]$	
$A[1][3]$	Third column
$A[2][3]$	

Addressing Array Elements (Cont'd)

- In a two-dimension array, the formula is

$$base + i_1 \times w_1 + i_2 \times w_2$$
$$= base + (i_1 \times n_2 + i_2) \times w$$

$A[0][0]$	$A[0][1]$	$A[0][2]$	$A[0][3]$
$A[1][0]$			
$A[2][0]$			$A[2][3]$

  - n_j : number of elements along dimension j
- In k dimensions, the formula is

$$base + i_1 \times w_1 + i_2 \times w_2 + \ldots + i_k \times w_k$$
$$= base + ((\ldots((i_1 \times n_2 + i_2) \times n_3 + i_3)\ldots) \times n_k + i_k) \times w$$

- Array elements need not be numbered starting at 0
  - In a one-dimension array, array elements can be numbered $low, low + 1, \ldots, high$
  - $A[i]$ then can be addressed by $base + (i - low) \times w$

SDT: Translation of Array References

$S \rightarrow \textbf{id} = E\ ;$ { … }

$\quad |\ L = E\ ;$ { *gen*(*L.array.base* '[' *L.addr* ']' '=' *E.addr*); }

$E \rightarrow E_1 + E_2$ { … }

$\quad |\ \textbf{id}$ { *E.addr* = *top.get*(**id**.*lexeme*); }

$\quad |\ L$ { *E.addr* = **new** *Temp*();
gen(*E.addr* '=' *L.array.base* '[' *L.addr* ']'); }

$L \rightarrow \textbf{id}\ [\ E\ ]$ { *L.array* = *top.get*(**id**.*lexeme*);
L.type = *L.array.type.elem*;
L.addr = **new** *Temp*();
gen(*L.addr* '=' *E.addr* '*' *L.type.width*); }

$\quad |\ L_1\ [\ E\ ]$ { *L.array* = L_1.*array*;
L.type = L_1.*type.elem*;
t = **new** *Temp*();
L.addr = **new** *Temp*();
gen(*t* '=' *E.addr* '*' *L.type.width*);
gen(*L.addr* '=' L_1.*addr* '+' *t*);}

A pointer to the symbol-table entry for the array name

Element type

Type of the subarray generated by *L*

A temporary, used while computing the offset for the array reference

...es: An Example

`c+a[i][j]`

$L \rightarrow L_1$ **[** E **]** $\{$ $L.array = L_1.array;$
$L.type = L_1.type.elem;$
$t =$ **new** $Temp();$
$L.addr =$ **new** $Temp();$
$gen(t$ '=' $E.addr$ '*' $L.type.width);$
$gen(L.addr$ '=' $L_1.addr$ '+' $t);$ $\}$

$L.type = L.array.type.elem;$
$L.addr =$ **new** $Temp();$
...$type.width);$ $\}$

$E \rightarrow$ **id** $\{$ $E.addr = top.get($**id**$.lexeme);$ $\}$

$E \rightarrow L$ $\{E.addr =$ **new** $Temp();$
$gen(E.addr$ '=' $L.array.base$ '[' $L.addr$ ']'$);$ $\}$

...ddr = t

...r = j]

j

$L.type = array(3, integer)$
$L.addr =$ t_1

a.$type$
$= array(2, array(3, integer))$ [$E.addr =$ i]

i

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a[t3]
t5 = c + t4
```

Three-address code

Outline

- Overview of Intermediate Representation
- Intermediate Representation
  - Syntax Trees
  - Three-Address Code
- Intermediate-Code Generation
  - Types and Declarations
    - Translation of Expressions
    - Type Checking
  - Control Flow
  - Backpatching
  - Switch-Statements and Procedures

Type Checking

- A compiler needs to assign a type expression to each component of the source program
  - Type system
    - The compiler determines that these type expressions conform to a collection of logical rules
    - In principle, any check can be done dynamically
    - A sound type system eliminates the need for dynamic checking for type errors
      - No type errors occur at run time
      - An implementation of a language is strongly typed if a compiler guarantees that the programs it accepts will run without type errors

Rules for Type Checking

- Two forms of type checking
  - Type synthesis
  - Type inference

- Type synthesis

  if f has type $s \rightarrow t$ **and** x has type s,
  then expression $f(x)$ has type t

  - E.g., $E_1 + E_2$ can be viewed as a function application $add(E_1, E_2)$ int × int → int
  - E.g., **if** (E) S can be viewed as a function application if to E and S. Then function if expects to be applied to a $boolean$ and a $void$, and the result is a $void$ boolean × void → void

Rules for Type Checking (Cont'd)

- 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

  - α, β: type variables
  - Needed for languages that do not require names to be declared
    - E.g., ML language (strongly typed)
    - Type inference ensures that names are used consistently

Type Conversions

- The representation of two data types, such as integer and float, is different within a computer
  - Different machine instructions are used for operations on integers and floats
  - Compilers may need to do type conversions to ensure that both operands are of the same type
- E.g., an expression `2 + 3.14`

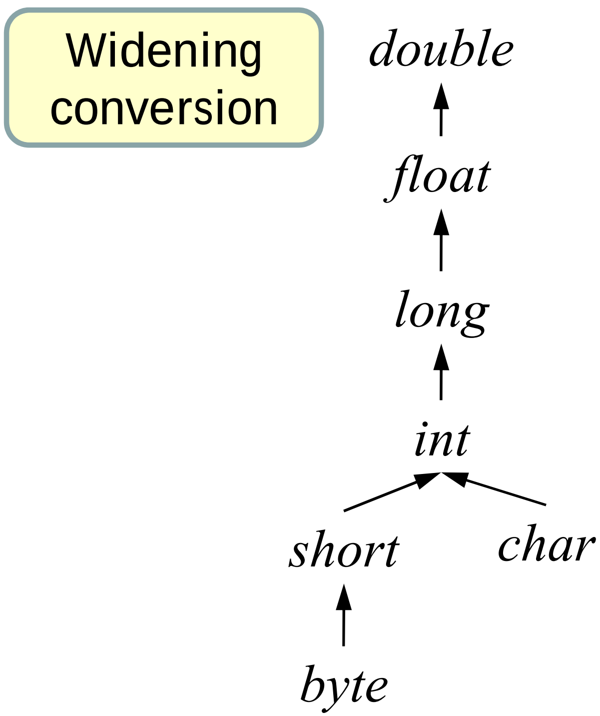
t_1 = (float) 2

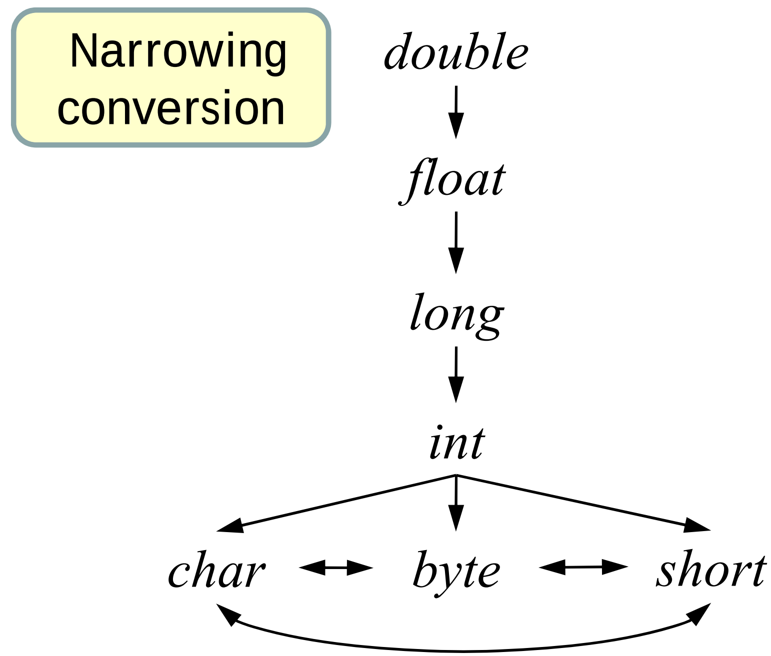
t_2 = t_1 + 3.14

Production	Semantic Rules
$E \to E_1 + E_2$	**if** ($E_1.type = integer$ **and** $E_2.type = integer$) $E.type = integer$ **else if** ($E_1.type = float$ **and** $E_2.type = integer$) … …

Widening and Narrowing Conversions

- Widening conversion
  - Intended to preserve information
- Narrowing conversion
  - Can lose information





Type Conversions (Cont'd)

- Implicit type conversion (coercion)
  - Done automatically by the compiler
  - Limited to widening conversion in many languages
- Explicit type conversion (cast)
  - The programmer must write something to cause the conversion

Type Checking: An Example

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

- $max(t_1, t_2)$ returns the maximum of the two types in the widening hierarchy, and it declares an error if either t_1 or t_2 is not in the hierarchy

$$
\begin{array}{l}
Addr\ widen(Addr\ a,\ Type\ t,\ Type\ w)\ \{ \\
\quad \mathbf{if}\ (\ t = w)\ \mathbf{return}\ a; \\
\quad \mathbf{else\ if}\ (\ t = integer\ \mathbf{and}\ w = float)\ \{ \\
\qquad temp = \mathbf{new}\ Temp(); \\
\qquad gen(temp\ \text{`='}\ \text{`}(\mathbf{float})\text{'}\ a); \\
\qquad \mathbf{return}\ temp; \\
\quad \}\ \mathbf{else\ error}; \\
\}
\end{array}
$$

Overloading of Functions and Operators

- An overloaded symbol has different meanings depending on its context
  - Overloading is resolved when a unique meaning is determined for each occurrence of a name
  - Assume we can resolve overloading by looking at the arguments of a function or the operands of an operator
  - E.g.,

```
void err() {…}
void err(String s) {…}
```

  - Type-synthesis rule for overloaded functions

if f can have type $s_i \to t_i$, for $1 \le i \le n$, where $s_i \ne s_j$ for $i \ne j$
and x has type s_k, for some $1 \le k \le n$
then expression $f(x)$ has type t_k

Polymorphic Functions

- Polymorphic
  - Refers to any code fragment that can be executed with arguments of different types
- An example for polymorphic functions
  - Length of a list x

    Tests whether a list is empty

    Returns the remainder of a list after the first element is removed

  - $$\textbf{fun } length(x) = \textbf{ if } \underline{null(x)} \textbf{ then } 0 \textbf{ else } length(\underline{tl(x)}) + 1;$$
  - E.g., $length($`["sun","mon",tue"]`$) + length([10,9,8,7])$
  - The type of $length$ can be described as
    - for any type α, $length$ maps a list of elements of type α to an integer
    - $\boxed{\forall \underline{\alpha}.\ \underline{list}(\alpha) \rightarrow integer}$ polymorphic type expression

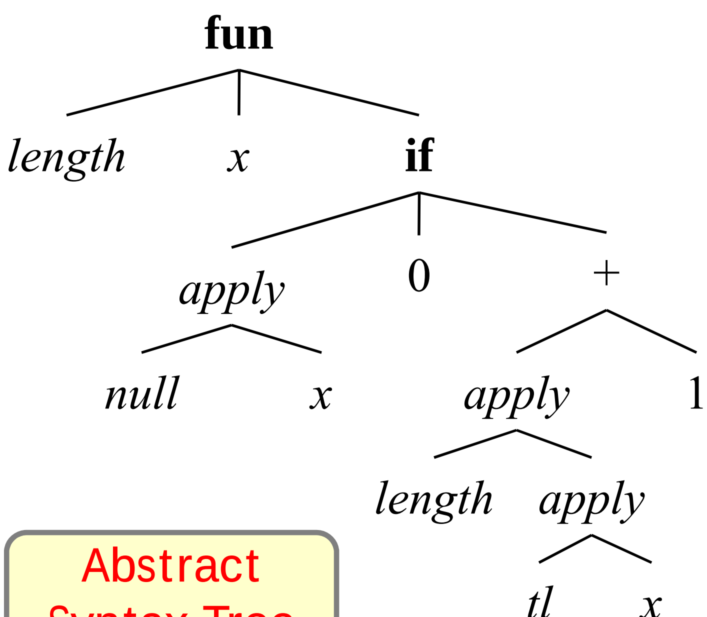
type variable   type constructor

Polymorphic Functions (Cont'd)

fun $length(x) =$
if $null(x)$ **then** 0 **else** $length(tl(x)) + 1;$

- $\forall \alpha.\ list(\alpha) \rightarrow integer$ (type expression)
- We can use type inference rules to infer a type for *length*

if $f(x)$ is an expression,
then for some α and β, f has type $\alpha \rightarrow \beta$ **and** x has type α



Abstract Syntax Tree

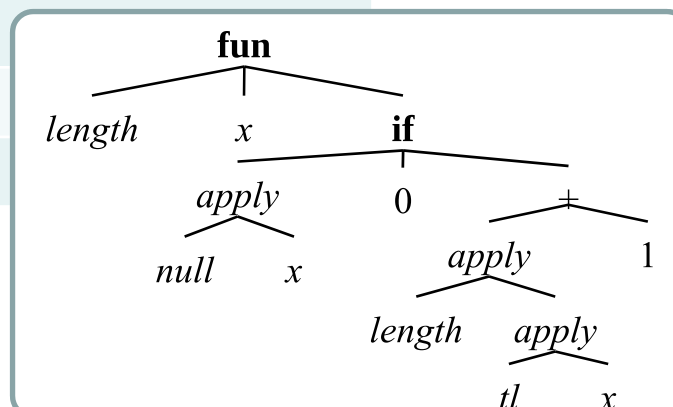
- Since *null* expects to be applied to lists, x must be a list
- Suppose x has type "list of α", where α is a type variable
- The type of *length* must be "function from list of α to integer"

Inferring a Type for the Polymorphic Function

Line	Expression : Type	Unify
1)	$length : \beta \to \gamma$	
2)	$x : \beta$	
3)	$\mathbf{if} : boolean \times \alpha_i \times \alpha_i \to \alpha_i$	
4)	$null : list(\alpha_n) \to boolean$	
5)	$null(x) : boolean$	$list(\alpha_n) = \beta$
6)	$0 : integer$	$\alpha_i = integer$
7)	$+ : integer \times integer \to integer$	
8)	$tl : list(\alpha_t) \to list(\alpha_t)$	
9)	$tl(x) : list(\alpha_t)$	$list(\alpha_t) = \beta = list(\alpha_n)$
10)	$length(tl(x)) : \gamma$	$\gamma = integer$
11)	$1 : integer$	
12)	$length(tl(x)) + 1 : integer$	
13)	$\mathbf{if}(\ldots) : integer$	

$\forall \alpha.\ boolean \times \alpha \times \alpha \to \alpha$

$\forall \alpha.\ list(\alpha) \to boolean$



- Type of *length*

  $\forall \alpha_n.\ list(\alpha_n) \to integer$

Substitution and Unification

- Since variables can appear in type expressions, we have to re-examine the notion of equivalence of types
  - Suppose E_1 (type: $s \to s'$) is applied to E_2 (type: t)
  - Instead of simply determining the equality of s and t, we must "unify" them
  - We determine whether s and t can be made structurally equivalent by replacing the type variable in s and t by type expressions
- A substitution is a mapping from type variables to type expressions
  - E.g., a mapping from α to *integer*

Substitution and Unification (Cont'd)

- Suppose t is a type and S is a substitution
- We write $S(t)$ for the result of consistently replacing all occurrences of each type variable α in t by $S(\alpha)$; $S(t)$ is called an instance of t
  - E.g., if $list(\alpha)$ is a type expression and
    a mapping from α to $integer$ is a substitution,
    then $list(integer)$ is an instance of $list(\alpha)$
  - E.g., $integer \rightarrow float$ is not an instance of $\alpha \rightarrow \alpha$
- Substitution S is a unifier of type expressions t_1 and t_2 if $S(t_1) = S(t_2)$
  - Two expressions t_1 and t_2 unify where exists some substitution S such that $S(t_1) = S(t_2)$

Unification

- Unification is the problem of determining whether two expressions s and t can be made identical by substituting expressions for the variables in s and t
- Testing equality of expressions is a special case of unification
  - If s and t have no type variables, then s and t unify iff they are identical
- Unification algorithm
  - To test structural equivalence of types via a graph-theoretic formulation
    - Type expressions are represented by DAG

Unification Algorithm

- Unification of two nodes, m and n, in a type graph

```
boolean unify(Node m, Node n) {
    s = find(m); t = find(n);
    if ( s = t ) return true;
    else if (nodes s and t represent the same basic type ) return true;
    else if (s is an op-node with children s1 and s2 and
             t is an op-node with children t1 and t2 and
             they have the same op) {
        union(s, t);
        return unify(s1, t1) and unify(s2, t2);
    }
    else if (s or t represents a variable) {
        union(s, t);
        return true;
    }
    else return false;
}
```

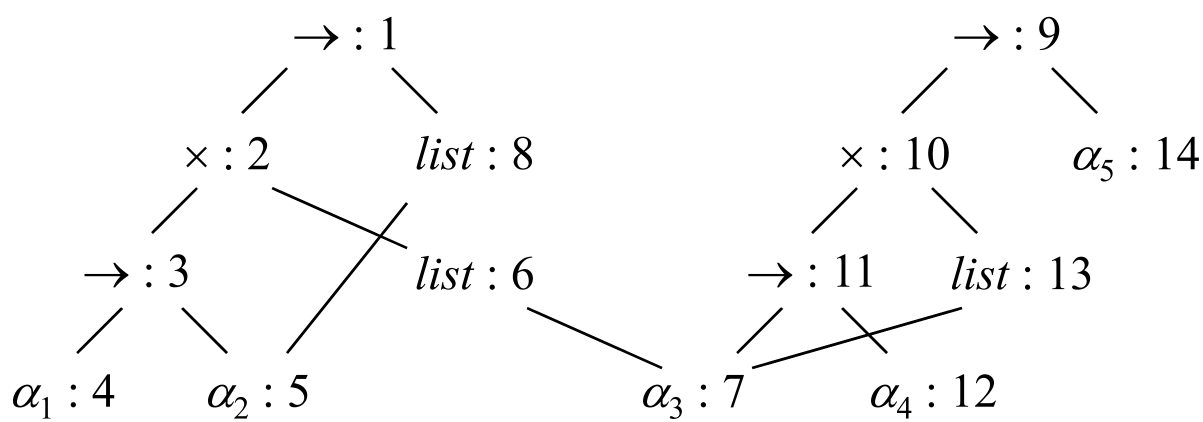
Returns the representative node of the equivalence class currently containing node n

Merges the equivalence classes containing nodes s and t

Unification: An Example

- Consider the two type expressions

$$((\alpha_1 \rightarrow \alpha_2) \times list(\alpha_3)) \rightarrow list(\alpha_2)$$
$$((\alpha_3 \rightarrow \alpha_4) \times list(\alpha_3)) \rightarrow \alpha_5$$

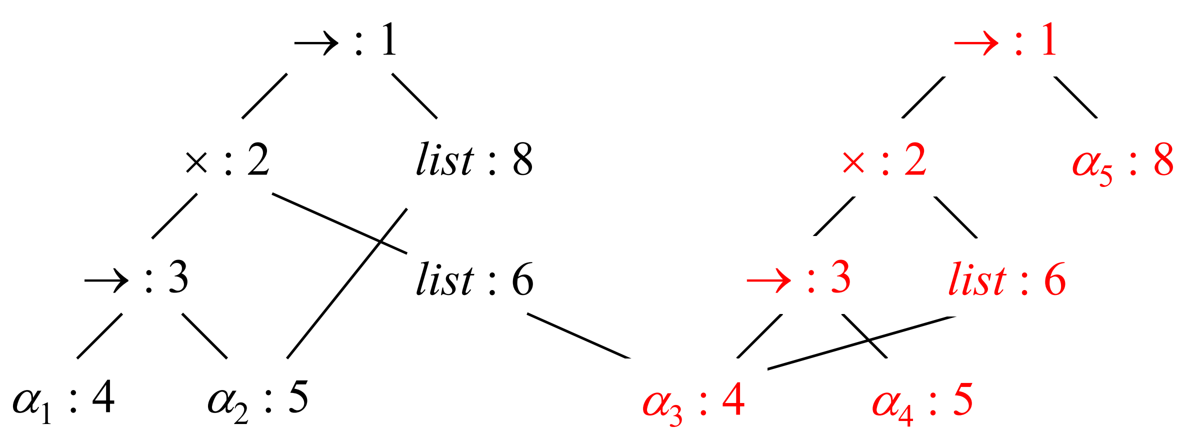


- Use $unify(1, 9)$ to check if the two type expressions can be unified

Unification: An Example (Cont'd)

```
boolean unify(Node m, Node n) {
        s = find(m); t = find(n);
        if ( s = t ) return true;
        else if (nodes s and t represent the same basic type ) return true;
        else if (s is an op-node with children s1 and s2 and
                        t is an op-node with children t1 and t2) {
                union(s, t);
                return unify(s1, t1) and unify(s2, t2);
        }
        else if (s or t represents a variable) {
                union(s, t);
                return true;
        }
        else return false;
}
```

- *unify*(1, 9) true
  - *unify*(2, 10) true
    - *unify*(3, 11) true
      - *unify*(4, 7) true
      - *unify*(5, 12) true
    - *unify*(6, 13) true
      - *unify*(4, 4) true
  - *unify*(8, 14) true



x	$S(x)$
α_1	α_1
α_2	α_2
α_3	α_1
α_4	α_2
α_5	$list(\alpha_2)$

Unification: An Example (Cont'd)

$$((\alpha_1 \rightarrow \alpha_2) \times list(\alpha_3)) \rightarrow list(\alpha_2)$$
$$((\alpha_3 \rightarrow \alpha_4) \times list(\alpha_3)) \rightarrow \alpha_5$$

- Substitution S, the most general unifier for the expressions

x	$S(x)$
α_1	α_1
α_2	α_2
α_3	α_1
α_4	α_2
α_5	$list(\alpha_2)$

- The substitution maps the two type expressions to

$$((\alpha_1 \rightarrow \alpha_2) \times list(\alpha_1)) \rightarrow list(\alpha_2)$$

Outline

- Overview of Intermediate Representation
- Intermediate Representation
  - Syntax Trees
  - Three-Address Code
- Intermediate-Code Generation
  - Types and Declarations
    - Translation of Expressions
    - Type Checking
  - Control Flow
  - Backpatching
  - Switch-Statements and Procedures

Boolean Expressions

- Boolean expressions are often used to
  - Alter the flow of control
    - Flow-of-control statements are tied to boolean expressions
      - If-statements or if-else-statements
      - While-statements
  - Compute logical values
    - E.g., $x = a < b$
- Grammar for boolean expressions

$$B \rightarrow B \;||\; B \mid B \;\&\&\; B \mid !B \mid (B) \mid E \;\textbf{rel}\; E \mid \textbf{true} \mid \textbf{false}$$

More on Boolean Expressions

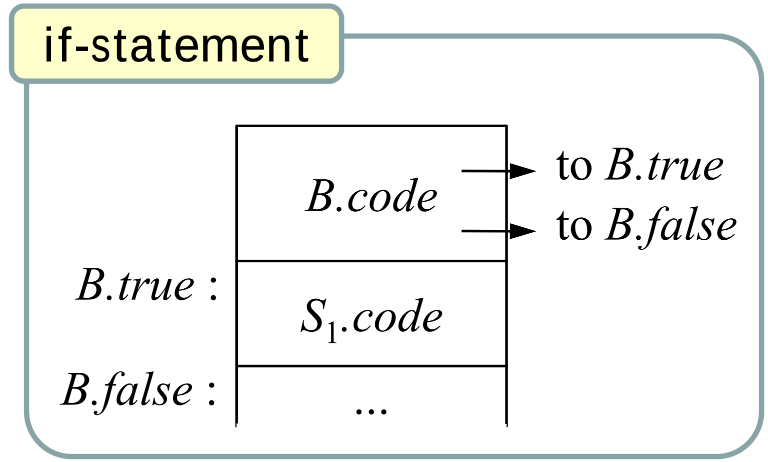
- Short-Circuit Code
  - Given the expression $B_1 \parallel B_2$
    - If B_1 is true $\Rightarrow$ the entire expression is true
      - No need to evaluate B_2
  - Given the expression $B_1 \,\&\&\, B_2$
    - If B_1 is false $\Rightarrow$ the entire expression is false
      - No need to evaluate B_2

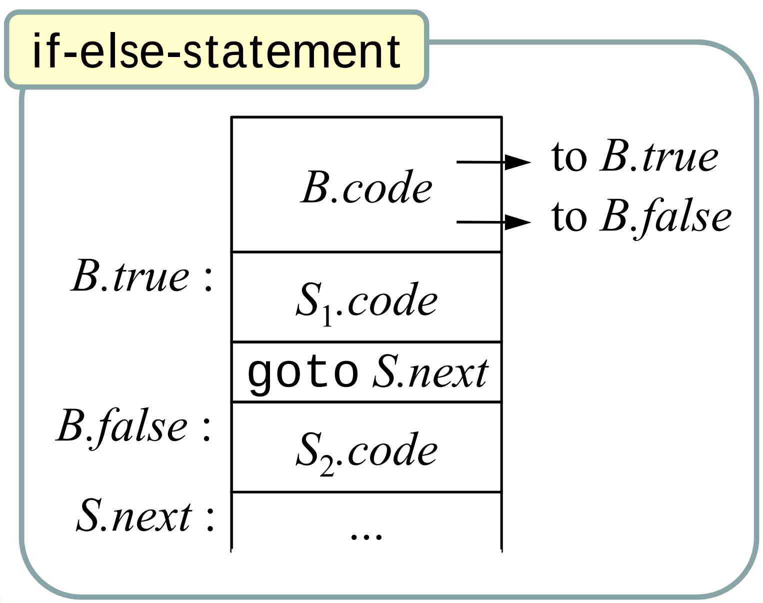
- Unless either B1 or B2 is an expression with side effects (e.g., it contains a function that changes a global variables)
  - An unexpected answer may be obtained
  - E.g., (a > b) || (b++ / 3)

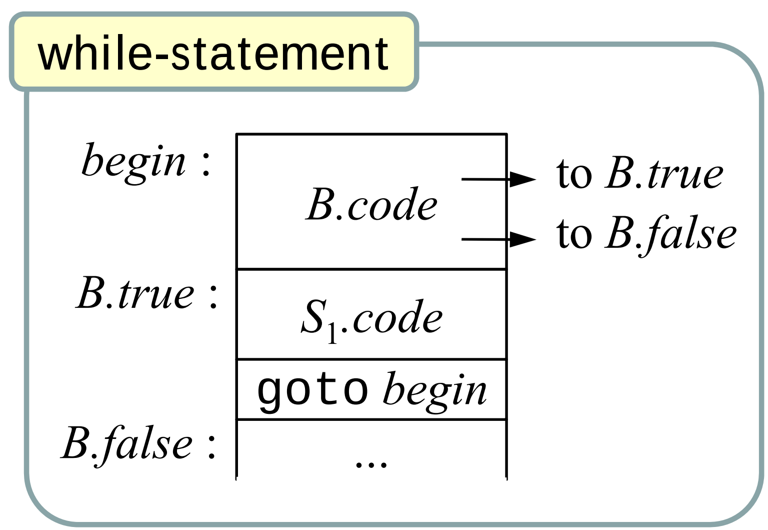
Flow-of-Control Statements

- Grammar for flow-of-control statements

$$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$$
$$S \rightarrow \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2$$
$$S \rightarrow \textbf{while}\ (\ B\ )\ S_1$$

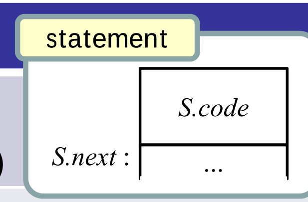


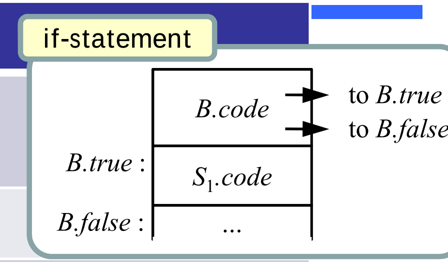


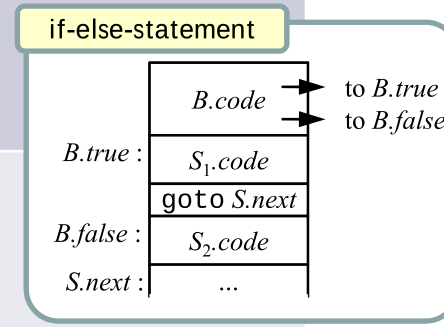


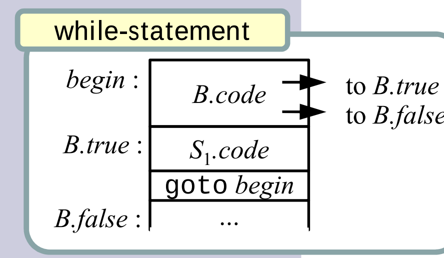
SDD for Flow-Of-Control Statements

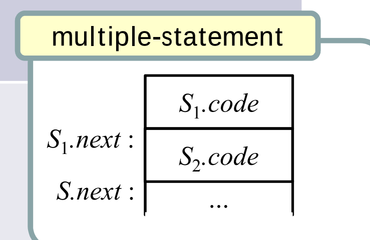
Production	Semantic Rules
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \parallel label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \parallel label(B.true) \parallel S_1.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code \parallel label(B.true) \parallel S_1.code$ $\parallel gen(\text{'}\texttt{goto}\text{'}\ S.next) \parallel label(B.false) \parallel S_2.code$
$S \rightarrow \textbf{while}\ (\ B\ )\ S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \parallel B.code \parallel label(B.true)$ $\parallel S_1.code \parallel gen(\text{'}\texttt{goto}\text{'}\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \parallel label(S_1.next) \parallel S_2.code$





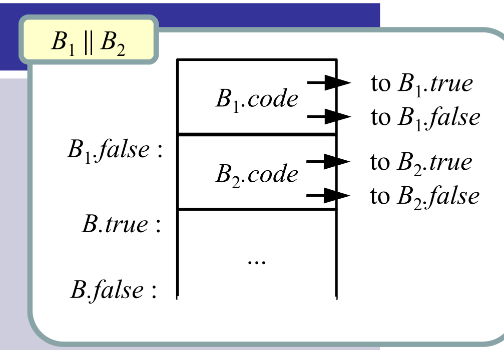


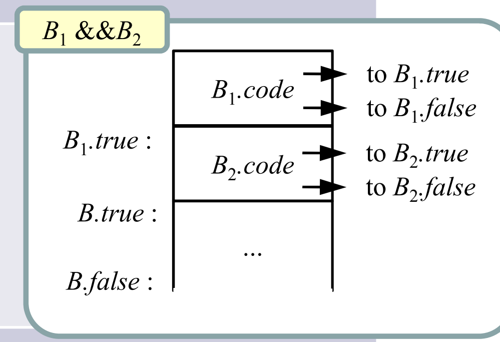




SDD for Boolean Expressions

Production	Semantic Rules
$B \rightarrow B_1 \parallel B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \parallel label(B_1.false) \parallel B_2.code$
$B \rightarrow B_1 \&\& B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \parallel label(B_1.true) \parallel B_2.code$
$B \rightarrow !\ B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \parallel E_2.code$ $\parallel gen($'`if`' $E_1.addr$ **rel**$.op$ $E_2.addr$ '`goto`' $B.true)$ $\parallel gen($'`goto`' $B.false)$
$B \rightarrow$ **true**	$B.code = gen($'`goto`' $B.true)$
$B \rightarrow$ **false**	$B.code = gen($'`goto`' $B.false)$





Translating If-Statements: An Example

- Consider the following statement

  `if (x < 100 || x > 200 && x != y) x = 0;`

  - `||` and `&&` are left-associative
  - Precedence: `!` > `&&` > `||`

- Translation

```
     if x < 100 goto L2
     goto L3
L3:  if x > 200 goto L4
     goto L1
L4:  if x != y goto L2
     goto L1
L2:  x = 0
L1:
```

- The code is not optimal
  - `goto` L_3 is redundant
  - `goto` L_4 can be eliminated by using `ifFalse` instead of `if` instructions

Avoiding Redundant Gotos

- The `ifFalse` instruction takes advantage of the natural flow from one instruction to the next in sequence

```
     if x > 200 goto L4
     goto L1
L4:  …
```

```
     ifFalse x > 200 goto L1
L4:  …
```

- Control simply "falls through" to label L_4 if $x > 200$, thereby avoiding a jump

Rewriting SDDs Using `ifFalse`

Production	Semantic Rules
…	…
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	$B.true = \cancel{newlabel()}\ fall$ $B.false = S_1.next = S.next$ $S.code = B.code\ \cancel{\Vert\ label(B.true)}\ \Vert\ S_1.code$
…	…
$B \rightarrow B_1 \Vert B_2$	$B_1.true = \cancel{B.true}\ \textbf{if } B.true \neq fall \textbf{ then } B.true \textbf{ else } newlabel()$ $B_1.false = \cancel{newlabel()}\ fall$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = \cancel{B_1.code \Vert label(B_1.false) \Vert B_2.code}$ $\textbf{if } B.true \neq fall \textbf{ then } B_1.code \Vert B_2.code$ $\textbf{else } B_1.code \Vert B_2.code \Vert label(B_1.true)$
$B \rightarrow E_1\ \textbf{rel}\ E_2$	$test = E_1.addr\ \ \textbf{rel}.op\ \ E_2.addr$ $s = \textbf{if } B.true \neq fall \textbf{ and } B.false \neq fall \textbf{ then}$ $gen(\texttt{'if'}\ test\ \texttt{'goto'}\ B.true) \Vert gen(\texttt{'goto'}\ B.false)$ $\textbf{else if } B.true \neq fall \textbf{ then } gen(\texttt{'if'}\ test\ \texttt{'goto'}\ B.true)$ $\textbf{else if } B.false \neq fall \textbf{ then } gen(\texttt{'ifFalse'}\ test\ \texttt{'goto'}\ B.false)$ $B.code = E_1.code \Vert E_2.code$ $\cancel{\Vert\ gen(\texttt{'if'}\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ \texttt{'goto'}\ B.true)}$ $\cancel{\Vert\ gen(\texttt{'goto'}\ B.false)}$ $\Vert\ s$
…	…

Translating If-Statements: An Example (Cont'd)

- Consider the following statement

```
if (x < 100 || x > 200 && x != y) x = 0;
```

$$
\begin{array}{ll}
 & \texttt{if x < 100 goto } L_2 \\
 & \texttt{goto } L_3 \\
L_3: & \texttt{if x > 200 goto } L_4 \\
 & \texttt{goto } L_1 \\
L_4: & \texttt{if x != y goto } L_2 \\
 & \texttt{goto } L_1 \\
L_2: & \texttt{x = 0} \\
L_1: &
\end{array}
$$

$$
\begin{array}{ll}
 & \texttt{if x < 100 goto } L_2 \\
 & \texttt{ifFalse x > 200 goto } L_1 \\
 & \texttt{ifFalse x != y goto } L_1 \\
L_2: & \texttt{x = 0} \\
L_1: &
\end{array}
$$

Using `ifFalse` instructions

Boolean Expressions for Computing Values

$S \to \textbf{id} = E \,;\, | \textbf{ if } (E)\, S \,|\, \textbf{while } (E)\, S \,|\, S\, S$

$E \to E \,||\, E \,|\, E \,\&\&\, \mathrm{E} \,|\, E \textbf{ rel } E \,|\, E + E \,|\, (E) \,|\, \textbf{id} \,|\, \textbf{true} \,|\, \textbf{false}$

- When E appears in $S \to \textbf{id} = E\,;$
  - If E has the form $E_1 + E_2$
    - Already discussed in Section 6.4 (slide 43)
  - If E has the from $E \,||\, E$ or $E \,\&\&\, \mathrm{E}$
    - Similar to the translation in Section 6.6.4 (slide 76)
- Needs two passes to translate E
  - Build the syntax tree
  - Walk the tree

Boolean Assignment: An Example

- Consider a boolean-assignment statement

  `x = a < b && c < d`

```
    ifFalse a < b goto L1
    ifFalse c < d goto L1
    t = true
    goto L2
L1: t = false
L2: x = t
```

  - First generate jumping code for E
  - Assign true or false to a new temporary `t` at the true and false exists

Remark of Section 6.6

- The key problem when generating code for boolean expression and flow-of-control statements is
  - Matching a jump instruction with the target of the jump
- Solution:
  - Passing labels as inherited attributes to where the relevant jump instructions were generated

Binding Labels to Addresses

- A separate pass is needed to bind labels to address

```
if (x < 100 || x > 200 && x != y) x = 0
```

- We first generate code with labels

```
      if x < 100 goto L2
      goto L3
L3:   if x > 200 goto L4
      goto L1
L4:   if x != y goto L2
      goto L1
L2:   x = 0
L1:
```

- Then we bind labels to addresses

```
100: if x < 100 goto 106
101: goto 102
102: if x > 200 goto 104
103: goto 107
104: if x != y goto 106
105: goto 107
106: x = 0
107:
```

Outline

- Overview of Intermediate Representation
- Intermediate Representation
  - Syntax Trees
  - Three-Address Code
- Intermediate-Code Generation
  - Types and Declarations
    - Translation of Expressions
    - Type Checking
  - Control Flow
  - Backpatching
  - Switch-Statements and Procedures

Backpatching

- A technique to generate code for boolean expressions and flow-of-control statements in one pass
- For a nonterminal B, two synthesized attributes are used
  - *truelist*
    - A list of jump instructions, which jump to some labels when B is true
  - *falselist*
    - A list of jump instructions, which jump to some labels when B is false
- We generate instructions into an array and labels will be indices into the array

Backpatching for Boolean Expressions

$B \rightarrow B_1 \;||\; M\, B_2$ $\{\; backpatch(B_1.falselist,\, M.instr);$
$B.truelist = merge(B_1.truelist,\, B_2.truelist);$
$B.falselist = B_2.falselist; \;\}$

$B \rightarrow B_1 \;\&\&\; M\, B_2$ $\{\; backpatch(B_1.truelist,\, M.instr);$
$B.truelist = B_2.truelist;$
$B.falselist = merge(B_1.falselist,\, B_2.falselist); \;\}$

$B \rightarrow \,!\, B_1$ $\{\; B.truelist = B_1.falselist;$
$B.falselist = B_1.truelist; \;\}$

$B \rightarrow E_1 \;\mathbf{rel}\; E_2$ $\{\; B.truelist = makelist(newinstr);$
$B.falselist = makelist(newinstr + 1);$
$gen($'`if`' $E_1.addr\; \mathbf{rel}.op\; E_2.addr$ '`goto –`'$);$
$gen($'`goto –`'$); \;\}$

$B \rightarrow \mathbf{true}$ $\{\; B.truelist = makelist(newinstr);$
$gen($'`goto –`'$); \;\}$

$B \rightarrow \mathbf{false}$ $\{\; B.falselist = makelist(newinstr);$
$gen($'`goto –`'$); \;\}$

$M \rightarrow \varepsilon$ $\{\; M.instr = newinstr; \;\}$

Backpatching for Boolean Expressions

$B \to B_1 \mid\mid M\, B_2$ $\{\ backpatch(B_1.falselist,\ M.instr);$
$B.truelist = merge(B_1.truelist,\ B_2.truelist);$
$B.falselist = B_2.falselist;\ \}$

- If B_1 is false $\Rightarrow$ the target of $B_1.falselist$ must be the beginning of the code of B_2
- Use marker nonterminal M to obtain the target and store it to $M.instr$, a synthesized attribute
- The variable $newinstr$ holds the index of the next instruction to follow
- $backpatch(p, i)$ inserts i as the target label for each of the instructions on the list pointed to by p
  - Each instruction in $B_1.falselist$ will receive $M.instr$ as its target label
- $merge(p_1, p_2)$ concatenates the list pointed to by p_1 and p_2, and returns a pointer to the concatenated list

$M \to \varepsilon$ $\{\ M.instr = newinstr;\ \}$

Backpatching for Boolean Expressions

$B \rightarrow B_1 \;||\; M\, B_2$ $\quad$ $\{\, backpatch(B_1.falselist,\, M.instr);$
$\quad B.truelist = merge(B_1.truelist,\, B_2.truelist);$
$\quad B.falselist = B_2.falselist;\ \}$

$B \rightarrow B_1 \;\&\&\; M\, B_2$ $\quad$ $\{\, backpatch(B_1.truelist,\, M.instr);$

- $makelist(i)$ creates a new list containing only i, an index into the array of instructions, and returns a pointer to the newly created list

$B \rightarrow E_1\ \mathbf{rel}\ E_2$ $\quad$ $\{\, B.truelist = makelist(newinstr);$
$\quad B.falselist = makelist(newinstr + 1);$
$\quad gen($`'if'` $E_1.addr\ \mathbf{rel}.op\ E_2.addr$ `'goto -'`$);$
$\quad gen($`'goto -'`$);\ \}$

$B \rightarrow \mathbf{true}$ $\quad$ $\{\, B.truelist = makelist(newinstr);$
$\quad gen($`'goto -'`$);\ \}$

$B \rightarrow \mathbf{false}$ $\quad$ $\{\, B.falselist = makelist(newinstr);$
$\quad gen($`'goto -'`$);\ \}$

$M \rightarrow \varepsilon$ $\quad$ $\{\, M.instr = newinstr;\ \}$

Backpatching for Boolean Expressions: An Example

- Consider the boolean expression

  `x < 100 || x > 200 && x != y`

- Initially, $newinstr = 100$

```
100:  if x < 100 goto -
101:  goto -
```

$B \to E_1$ **rel** E_2 { $B.truelist = makelist(newinstr);$ {100}
$B.falselist = makelist(newinstr + 1);$ {101}
$gen($'`if`' $E_1.addr$ **rel**$.op$ $E_2.addr$ '`goto –`'$);$
$gen($'`goto –`'$);$ }

B
\$

Backpatching for Boolean Expressions: An Example

- Consider the boolean expression

  `x < 100 || x > 200 && x != y`

- Initially, $newinstr = 100$

```
100:  if x < 100 goto -
101:  goto –
```

$$B \rightarrow B_1 \mid\mid M\, B_2 \quad \{\ backpatch(B_1.falselist, M.instr);$$
$$B.truelist = merge(B_1.truelist, B_2.truelist);$$
$$B.falselist = B_2.falselist;\ \}$$

…

$$M \rightarrow \varepsilon \quad \{\ M.instr = newinstr;\ \}$$

102

M	
$\mid\mid$	
B	
$\$$	

Backpatching for Boolean Expressions: An Example

- Consider the boolean expression

  `x < 100 || `**`x > 200`**` && x != y`

- Initially, $newinstr = 100$

```
100:  if x < 100 goto -
101:  goto –
102:  if x > 200 goto -
103:  goto -
```

$B \rightarrow E_1 \ \mathbf{rel} \ E_2 \quad \{ \ B.truelist = makelist(newinstr);$ {102}

$B.falselist = makelist(newinstr + 1);$ {103}

$gen($'`if`' $E_1.addr \ \mathbf{rel}.op \ E_2.addr$ '`goto –`'$);$

$gen($'`goto –`'$); \ \}$

B
M
$\|\|$
B
$\$$

Backpatching for Boolean Expressions: An Example

- Consider the boolean expression

  `x < 100 || x > 200 && x != y`

- Initially, $newinstr = 100$

```
100:  if x < 100 goto -
101:  goto –
102:  if x > 200 goto -
103:  goto –
```

$B \rightarrow B_1$ && $M\ B_2$ { $backpatch(B_1.truelist, M.instr);$
$B.truelist = B_2.truelist;$
$B.falselist = merge(B_1.falselist, B_2.falselist);$ }

…

$M \rightarrow \varepsilon$ { $M.instr = newinstr;$ } 104

M
&&
B
M
||
B
\$

Backpatching for Boolean Expressions: An Example

- Consider the boolean expression

  `x < 100 || x > 200 && x != y`

- Initially, $newinstr = 100$

$$B \rightarrow E_1 \ \mathbf{rel}\ E_2 \quad \{\ B.truelist = makelist(newinstr);\ \{104\}$$
$$B.falselist = makelist(newinstr + 1);\ \{105\}$$
$$gen(\text{'}\texttt{if}\text{'}\ E_1.addr\ \mathbf{rel}.op\ E_2.addr\ \text{'}\texttt{goto -}\text{'});$$
$$gen(\text{'}\texttt{goto -}\text{'});\ \}$$

```
103:  goto –
104:  if x != y goto -
105:  goto -
```

B
M
&&
B
M
||
B
\$

Backpatching for Boolean Expressions: An Example

- Consider the boolean expression

  `x < 100 || x > 200 && x != y`

- Initially, $newinstr = 100$

```
100:  if x < 100 goto -
101:  goto –
102:  if x > 200 goto 104
103:  goto –
104:  if x != y goto -
105:  goto –
```

B
M
$\|\|$
B
$\$$

{102} 104

$B \rightarrow B_1 \mathbin{\&\&} M\, B_2 \quad \{\ backpatch(B_1.truelist, M.instr);$

$B.truelist = B_2.truelist;$ {104}

$B.falselist = merge(B_1.falselist, B_2.falselist);\ \}$ {103,105}

{103} {105}

Backpatching for Boolean Expressions: An Example

- Consider the boolean expression

  x < 100 || x > 200 && x != y

- Initially, $newinstr = 100$

```
100:  if x < 100 goto -
101:  goto 102
102:  if x > 200 goto 104
103:  goto –
104:  if x != y goto -
105:  goto –
```

{101} 102

$B \rightarrow B_1 \;||\; M\, B_2 \quad \{\ backpatch(B_1.falselist,\ M.instr);$

{104}

$B.truelist = merge(B_1.truelist,\ B_2.truelist);$

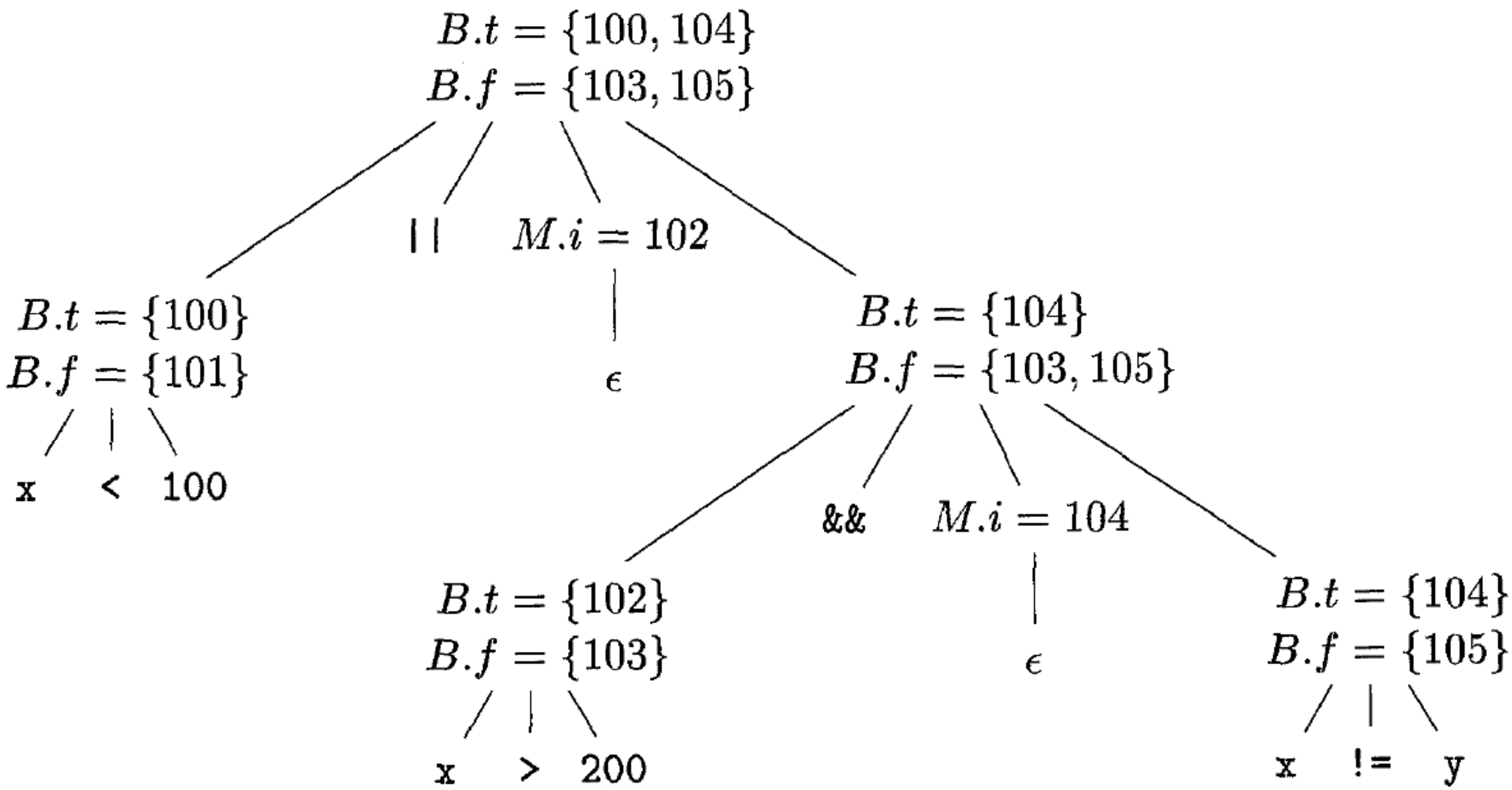
{100}

$B.falselist = B_2.falselist;\ \}$

{103,105} {100, 104}

B
$\$$

Annotated Parse Tree for the Previous Example



- The entire expression is
  - true iff the gotos of instruction 100 or 104 are reached
  - false iff the gotos of instruction 103 or 105 are reached

What About the Other Labels?

- Consider the flow-of-control statement

```
if (x < 100 || x > 200 && x != y) x = 0
```

```
100:  if x < 100 goto 106
101:  goto 102
102:  if x > 200 goto 104
103:  goto 107
104:  if x != y goto 106
105:  goto 107

106:  x = 0
107:
```

Backpatching for Flow-Of-Control Statements

$S \rightarrow \textbf{if}\ (B)\ M\ S_1$	$\{\ backpatch(B.truelist, M.instr);$ $S.nextlist = merge(B.falselist, S_1.nextlist);\ \}$
$S \rightarrow \textbf{if}\ (B)\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2$	$\{\ backpatch(B.truelist, M_1.instr);$ $backpatch(B.falselist, M_2.instr);$ $temp = merge(S_1.nextlist, N.nextlist);$ $S.nextlist = merge(temp, S_2.nextlist);\ \}$
$S \rightarrow \textbf{while}\ M_1\ (B)\ M_2\ S_1$	$\{\ backpatch(S_1.nextlist, M_1.instr);$ $backpatch(B.truelist, M_2.instr);$ $S.nextlist = B.flaselist;$ $gen(\text{'}\texttt{goto}\text{'}\ M_1.instr);\ \}$
$S \rightarrow \{\ L\ \}$	$\{\ S.nextlist = L.nextlist;\ \}$
$S \rightarrow A\ ;$	$\{\ S.nextlist = \textbf{null};\ \}$
$M \rightarrow \varepsilon$	$\{\ M.instr = newinstr;\ \}$
$N \rightarrow \varepsilon$	$\{\ N.nextlist = makelist(newinstr);$ $gen(\text{'}\texttt{goto -}\text{'});\ \}$
$L \rightarrow L_1\ M\ S$	$\{\ backpatch(L_1.nextlist, M.instr);$ $L.nextlist = S.nextlist;\ \}$
$L \rightarrow S$	$\{\ L.nextlist = S.nextlist;\ \}$

Backpatching for Flow-Of-Control Statements

$S \rightarrow$ **if** $(B)\ M\ S_1$	$\{$ *backpatch*$(B.truelist, M.instr)$;
$B.true$	$S.nextlist = merge(B.falselist, S_1.nextlist); \}$
$S \rightarrow$ **while** $M_1\ (B)\ M_2\ S_1$	$\{$ *backpatch*$(S_1.nextlist, M_1.instr)$;
	backpatch$(B.truelist, M_2.instr)$;
	$S.nextlist = B.flaselist$;
	gen(‘`goto`’ $M_1.instr$); }
$S \rightarrow \{\ L\ \}$	$\{ S.nextlist = L.nextlist; \}$
$S \rightarrow A\ ;$	$\{ S.nextlist =$ **null**; }
$M \rightarrow \varepsilon$	$\{ M.instr = newinstr; \}$
$N \rightarrow \varepsilon$	$\{ N.nextlist = makelist(newinstr)$;
	gen(‘`goto -`’); }
$L \rightarrow L_1\ M\ S$	$\{$ *backpatch*$(L_1.nextlist, M.instr)$;
	$L.nextlist = S.nextlist; \}$
$L \rightarrow S$	$\{ L.nextlist = S.nextlist; \}$

- Nonterminal statement S has a synthesized attribute *nextlist*, which is a list of jumps to the instructions following the of code of statement S

Backpatching for Flow-Of-Control Statements

$S \rightarrow \textbf{if}\ (B)\ M\ S_1$	$\{\ backpatch(B.turelist,\ M.instr);$ $S.nextlist = merge(B.falselist,\ S_1.nextlist);\ \}$
$S \rightarrow \textbf{if}\ (B)\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2$ *B.true* *B.false* *goto -*	$\{\ backpatch(B.truelist,\ M_1.instr);$ $backpatch(B.falselist,\ M_2.instr);$ $temp = merge(S_1.nextlist,\ N.nextlist);$ $S.nextlist = merge(temp,\ S_2.nextlist);\ \}$
$S \rightarrow \textbf{while}\ M_1\ (B)\ M_2\ S_1$ *begin* *B.true* *goto* M_1*.instr*	$\{\ backpatch(S_1.nextlist,\ M_1.instr);$ $backpatch(B.truelist,\ M_2.instr);$ $S.nextlist = B.flaselist;$ $gen($`'goto'` $M_1.instr);\ \}$
$S \rightarrow \{\ L\ \}$	$\{\ S.nextlist = L.nextlist;\ \}$
$S \rightarrow A\ ;$	$\{\ S.nextlist = \textbf{null};\ \}$
$M \rightarrow \varepsilon$	$\{\ M.instr = newinstr;\ \}$
$N \rightarrow \varepsilon$	$\{\ N.nextlist = makelist(newinstr);$ $gen($`'goto -'`$);\ \}$
$L \rightarrow L_1\ M\ S$	$\{\ backpatch(L_1.nextlist,\ M.instr);$ $L.nextlist = S.nextlist;\ \}$
$L \rightarrow S$	$\{\ L.nextlist = S.nextlist;\ \}$

Outline

- Overview of Intermediate Representation
- Intermediate Representation
  - Syntax Trees
  - Three-Address Code
- Intermediate-Code Generation
  - Types and Declarations
    - Translation of Expressions
    - Type Checking
  - Control Flow
  - Backpatching
  - Switch-Statements and Procedures

Translation of Switch-Statements

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

- Break the switch-case after one case is matched

	code to evaluate E into t
	`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`$_{n\text{-}2}$`:`	`if t !=` $V_{n\text{-}1}$ `goto L`$_{n\text{-}1}$
	code for $S_{n\text{-}1}$
	`goto next`
`L`$_{n\text{-}1}$`:`	code for S_n
`next:`	

Translation of Procedures

- Assume that parameters are passed by value

```
n = f(a[i]);
```

$$
\begin{array}{l}
t_1 = i * 4 \\
t_2 = a [t_1] \\
\text{param } t_2 \\
t_3 = \text{call } f,\ 1 \\
n = t_3
\end{array}
$$

- Details will be discussed in Chapter 7