Intermediate code generation

Jordi Cortadella

Dept. CS, UPC

May 5, 2016

From source to machine code

Compiler front end	Source code ↓ Parser/Semantic Analysis AST
Compiler back end	↓ Intermediate code generation IR ↓ Optimization Assembly Code
Assembler	↓ Assemble Relocatable Object Code
Linker	↓ Link Executable
Loader	↓ Relocate In-memory image

Compiler Frontends and Backends

The front-end focuses on *analysis*:

- Lexical analysis
- Parsing
- Static semantic checking
- AST generation

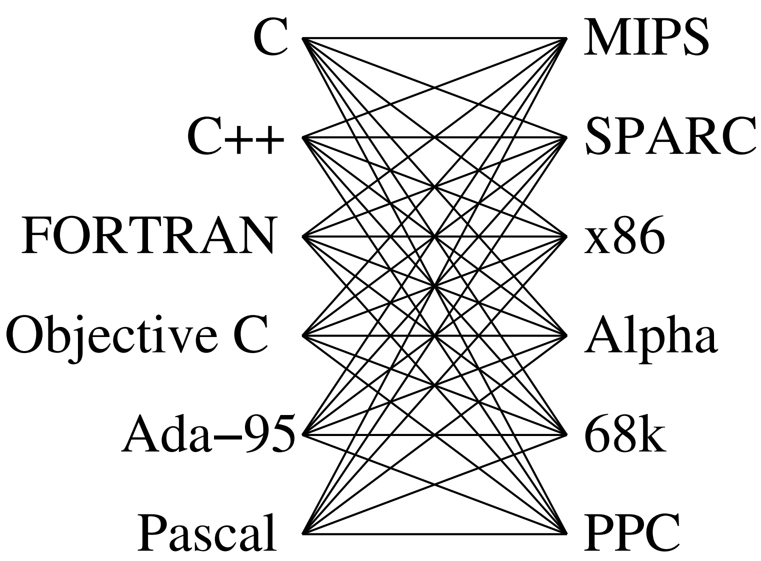
The back-end focuses on *synthesis*:

- Translation of the AST into intermediate code
- Optimization
- Assembly code generation

Portable Compilers

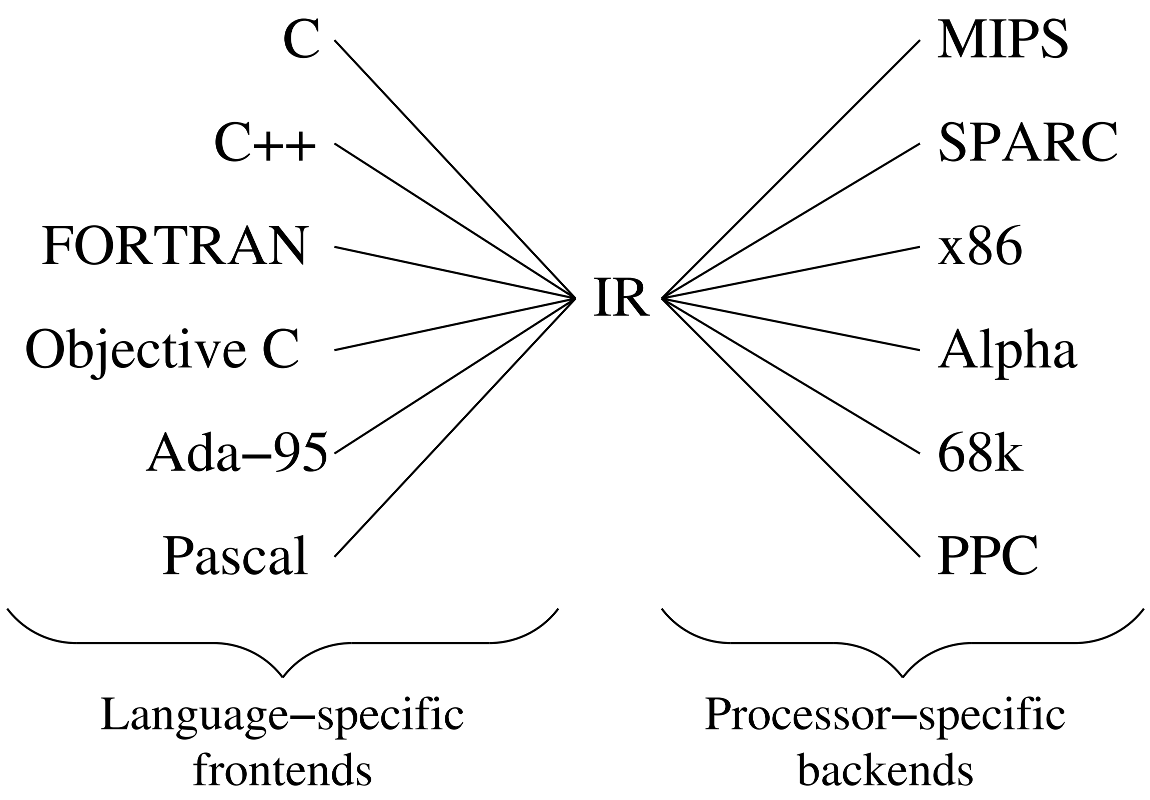
Building a compiler a large undertaking; most try to leverage it by making it portable.

Instead of



Portable Compilers

Use a common intermediate representation



Intermediate language

There are various intermediate languages used in compilers. They are languages of abstract machines used for analysis, optimization and machine language generation.

Some intemediate languages:

- Syntax trees
- Three-address codes
- Stack machines

Three-address code will be used in this course for code generation and optimization.

Three-address code

Most instructions have three addresses:

$$x = y \;\; op \;\; z$$

Only one operator allowed in the right side of an instruction, e.g., the instruction `a = x+y*z` must be translated into this sequence of instructions:

$$\begin{aligned} \mathtt{t}_1 &= \mathtt{y} * \mathtt{z} \\ \mathtt{a} &= \mathtt{x} + \mathtt{t}_1 \end{aligned}$$

where $\mathtt{t}_1$ is a compiler-generated temporary name.

Addresses and instructions

An address can be one of the following:

- A name. Source-program names appear as addresses. In an implementation, every source name is replaced by a pointer to its symbol-table entry.
- A constant.
- A compiler-generated temporary. For code optimization, it is convenient to use a different name each time a temporary is needed.

Three-address instructions:

- $x = y \ op \ z$, where op is an arithmetic or logical operator and x, y and z are addresses.
- $x = op \ y$, where op is a unary operation (minus, logical negation, ...).
- $x = y$ (copy instructions).
- `goto L` (unconditional jump).
- `if` x `goto L` and `ifFalse` x `goto L` (conditional jump).
- `if` x $relop$ y `goto L`, where $relop$ is a relational operator ($<$, $==$, $>=$, ...).

Addresses and instructions

To call the function $p(x_1, x_2, \ldots, x_n)$:

$$\begin{array}{ll} \texttt{param} & x_1 \\ \texttt{param} & x_2 \\ \ldots & \\ \texttt{param} & x_n \\ \texttt{call} & p, n \end{array}$$

The integer n is not redundant because calls can be nested.
The call can return a result as follows: $t = \texttt{call}\ \ p, n$.

To return from a function:

$$\texttt{return}\ \ y$$

where y is the returned result (optional)

Addresses and instructions

Indexed copy instructions:

- $x = y[i]$ and $x[i] = y$,
  where $x[i]$ represents the location i memory units beyond location x.

Address and pointer assignments:

- $x = \&y$, sets x to point to the location of y.
- $x = *y$, sets x to the value of the location pointed to by y.
- $*x = y$, sets the value of the location pointed to by x to the value of y.

Code generation: example

The statement

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

can be translated into

$$
\begin{aligned}
\texttt{L:}\quad & \mathtt{t}_1 = i + 1 \\
& i = \mathtt{t}_1 \\
& \mathtt{t}_2 = i * 8 \\
& \mathtt{t}_3 = a[\mathtt{t}_2] \\
& \texttt{if } \mathtt{t}_3 < v \texttt{ goto L}
\end{aligned}
$$

Declarations

Variables need to be allocated in memory regions.

- How much storage do we need for each type?
- What are the relative addresses of each component inside a type?

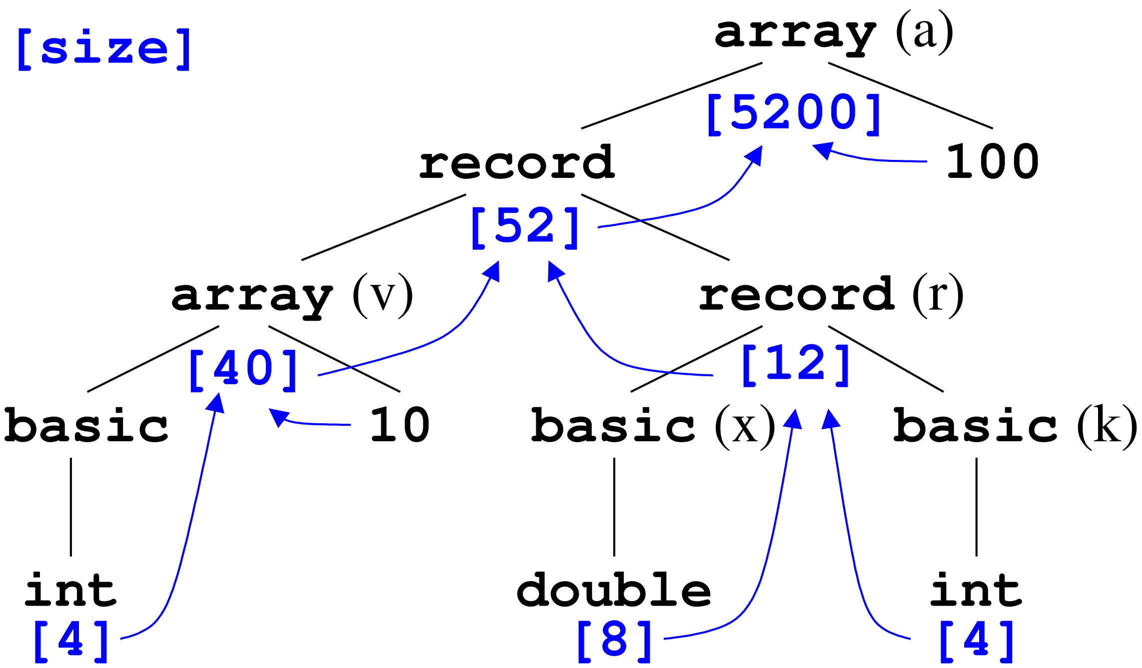
We will use this simple grammar for declarations:

$$
\begin{aligned}
\texttt{Decl} &\rightarrow \texttt{Type}\ \ \textbf{id}\ \textbf{;}\ \texttt{Decl}\ \mid \epsilon \\
\texttt{Type} &\rightarrow \texttt{Basic} \mid \texttt{Array} \mid \texttt{Record} \\
\texttt{Basic} &\rightarrow \textbf{int} \mid \textbf{char} \mid \textbf{float} \mid \ \ldots \\
\texttt{Array} &\rightarrow \texttt{Type}\ \ [\textbf{num}] \\
\texttt{Record} &\rightarrow \textbf{record}\ \{\ \texttt{Decl}\ \}
\end{aligned}
$$

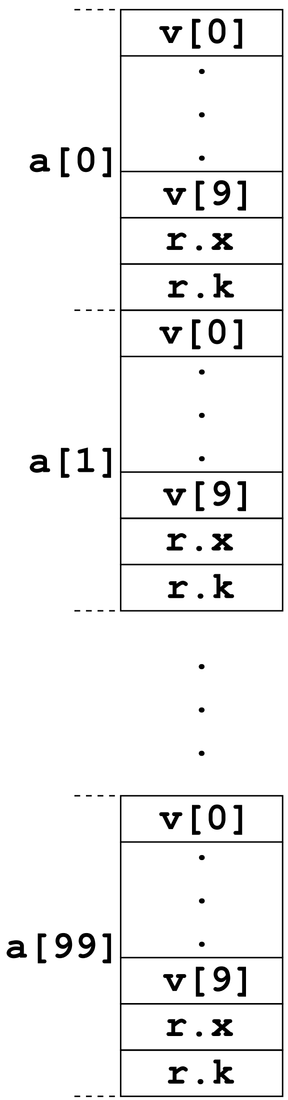
Declarations

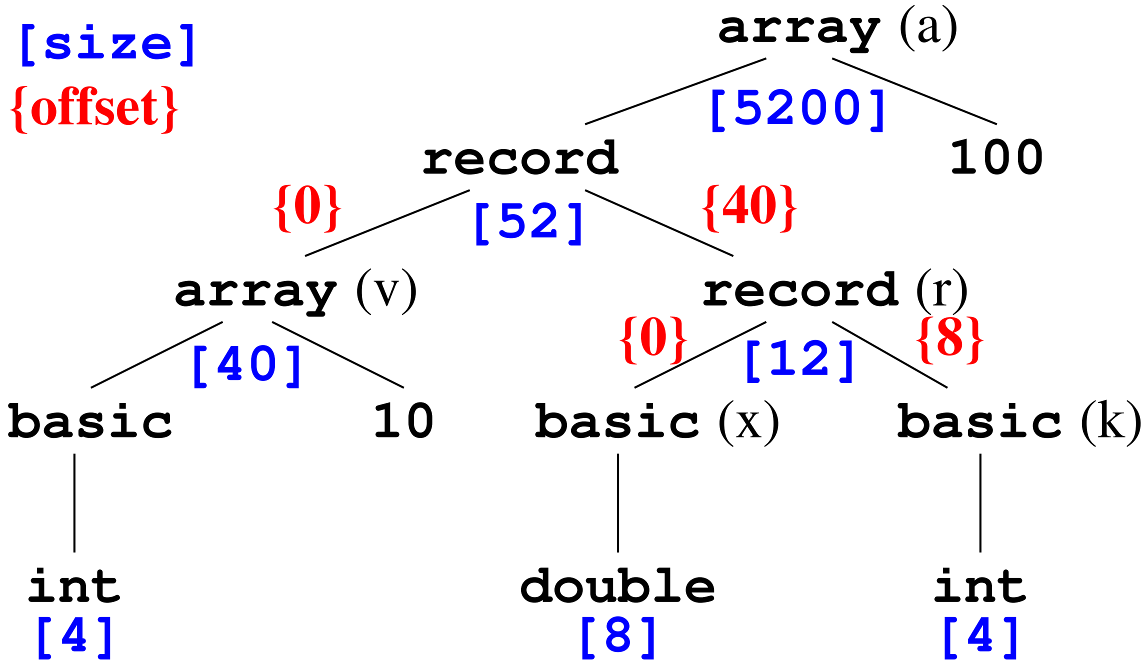
What is the size of each type component?

```
record {
  int [10] v;
  record {
    double x;
    int k;
  } r;
} [100] a;
```



Declarations





Q: What is the memory location for a[13].r.k ?

Declarations

Let us assume an array $A[l \ldots u]$ containing the elements $A[l], A[l+1], \ldots, A[u]$:

$$\texttt{Address}(A[i]) = \texttt{Base}(A) + (i - l) \times \texttt{SizeElem}(A)$$

For arrays in C, C++ and Java ($l = 0$),

$$\texttt{Address}(A[i]) = \texttt{Base}(A) + i \times \texttt{SizeElem}(A)$$

Example: let us consider the array int $A[100][25]$. What is the address of $A[i][j]$?

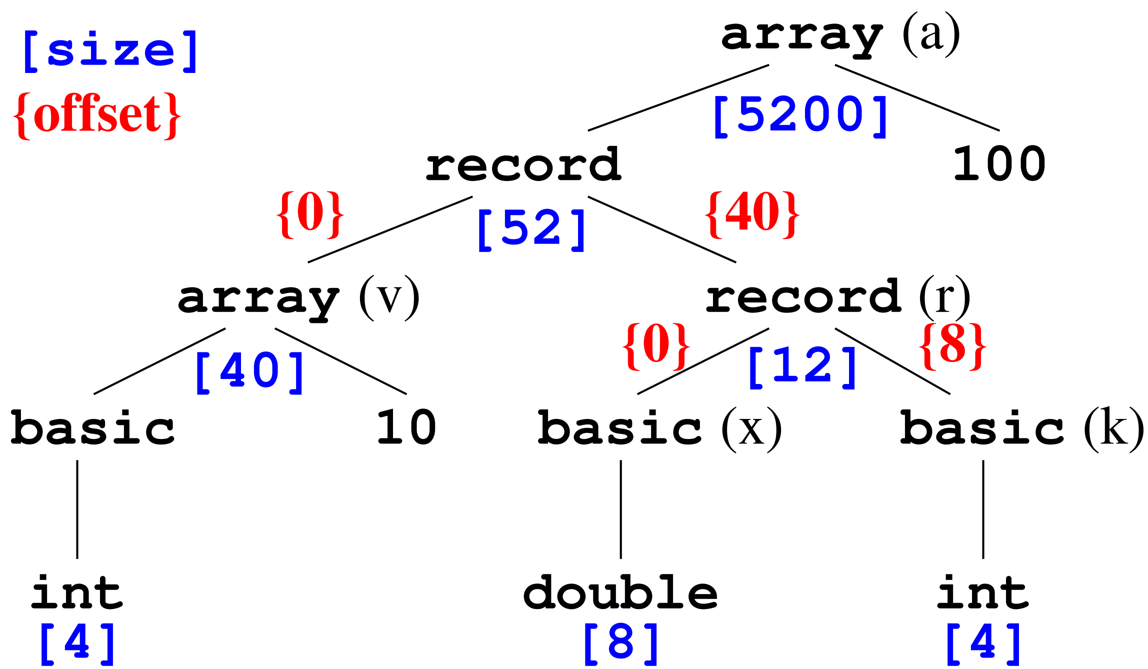
$$\begin{aligned}
\texttt{Size}(\texttt{int}) &= 4 \\
\texttt{Size}(A[i]) &= 25 \times \texttt{Size}(\texttt{int}) = 100 \\
\texttt{Address}(A[i][j]) &= \texttt{Address}(A[i]) + j \times 4 = \texttt{Base}(A) + i \times 100 + j \times 4
\end{aligned}$$

Records:

$$\texttt{Address}(A.z) = \texttt{Base}(A) + \texttt{Offset}(A, z)$$

Declarations

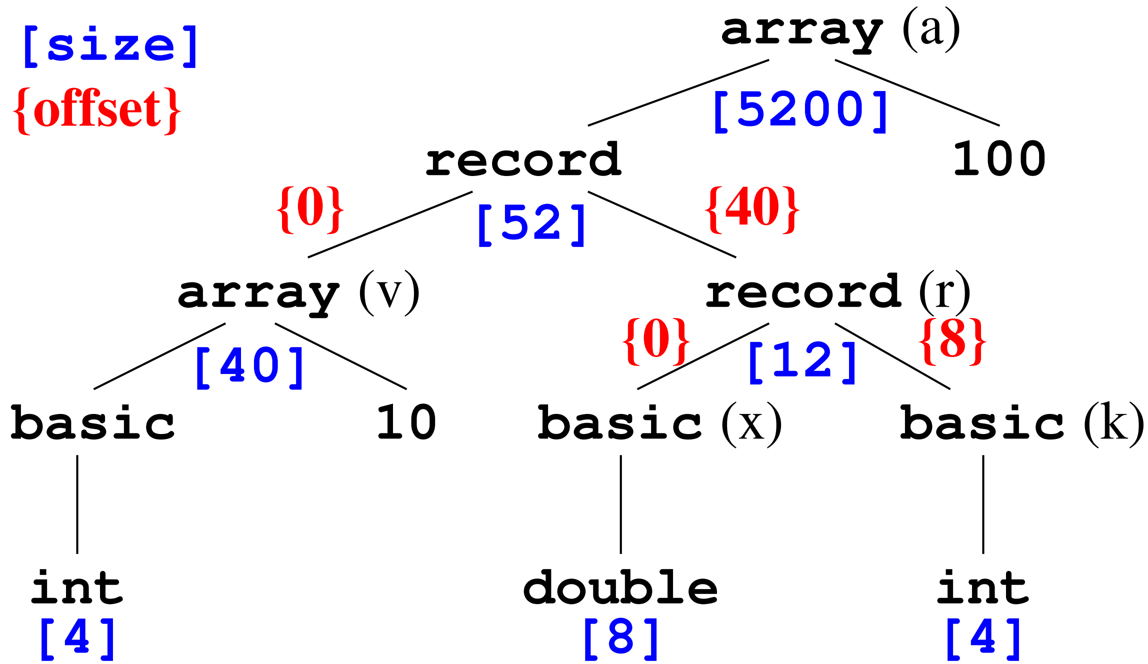
```
record {
  int [10] v;
  record {
    double x;
    int k;
  } r;
} [100] a;
```



Q: What is the memory location for a[13].r.k ?

Declarations

```
record {
  int [10] v;
  record {
    double x;
    int k;
  } r;
} [100] a;
```

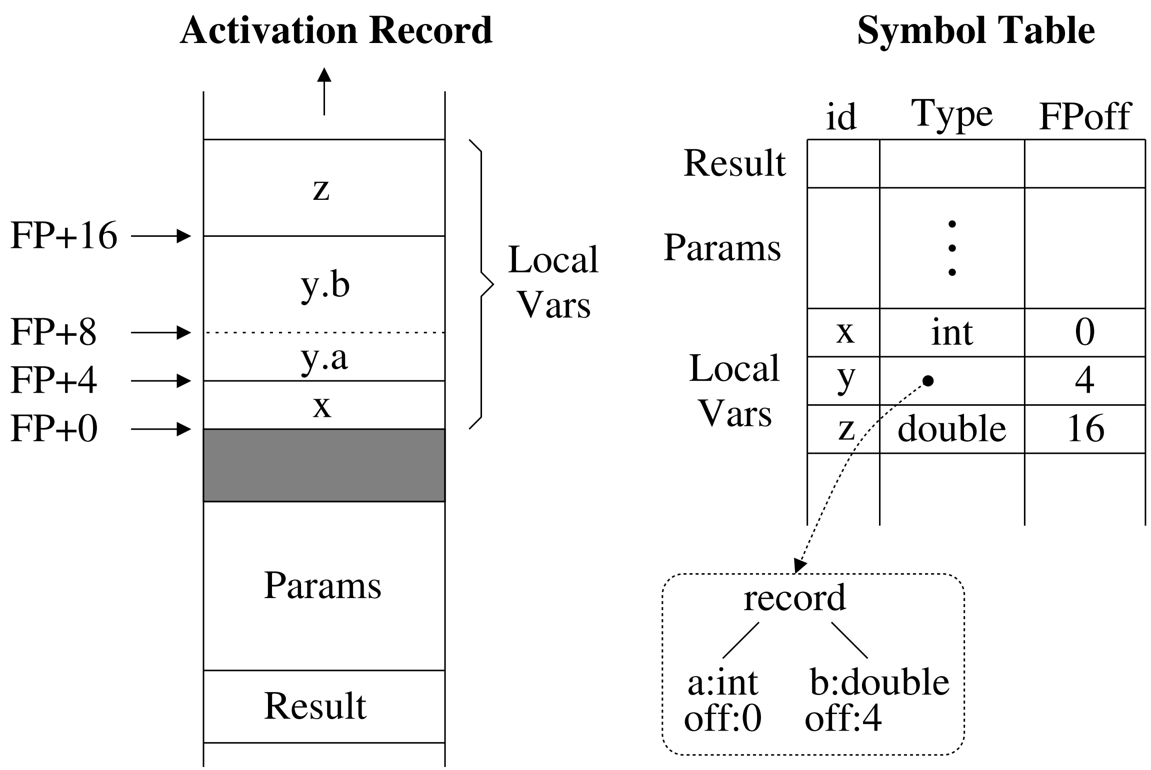


Q: What is the memory location for a[13].r.k ?

$$\texttt{Address}(a[13].r.k) = \texttt{Base}(A) + 13 \times 52 + 40 + 8 = \texttt{Base}(A) + 724$$

Storage allocation for declarations

```
int f(int p, int q) {
  int x;
  record {
    int a;
    double b;
  } y;
  double z;
  ...
}
```



The symbol table contains information on how the variables are stored in the activation record (offsets with regard to the frame pointer).

Storage allocation for declarations

Ex.: declaration of local variables (grammar and semantic actions)

$$
\begin{array}{rcl}
\texttt{LocalVars} & \rightarrow & \{\mathit{FPoffset} = 0;\} \ \ \texttt{ListDecls} \\
\texttt{ListDecls} & \rightarrow & \texttt{Decl} \ \ \texttt{ListDecls} \mid \varepsilon \\
\texttt{Decl} & \rightarrow & \texttt{Type} \ \ \textbf{id} \ \ \textbf{;} \\
 & & \{\textsc{SymTab}.\mathit{put}\ (\textbf{id}.\mathit{lexeme},\ \texttt{Type}.\mathit{type},\ \mathit{FPoffset}); \\
 & & \ \ \mathit{FPoffset} = \mathit{FPoffset} + \texttt{Type}.\mathit{size};\}
\end{array}
$$

Exercise

Consider the semantic actions for storage allocation of:

- Parameters and results of a function.
- Nested scopes within a function.

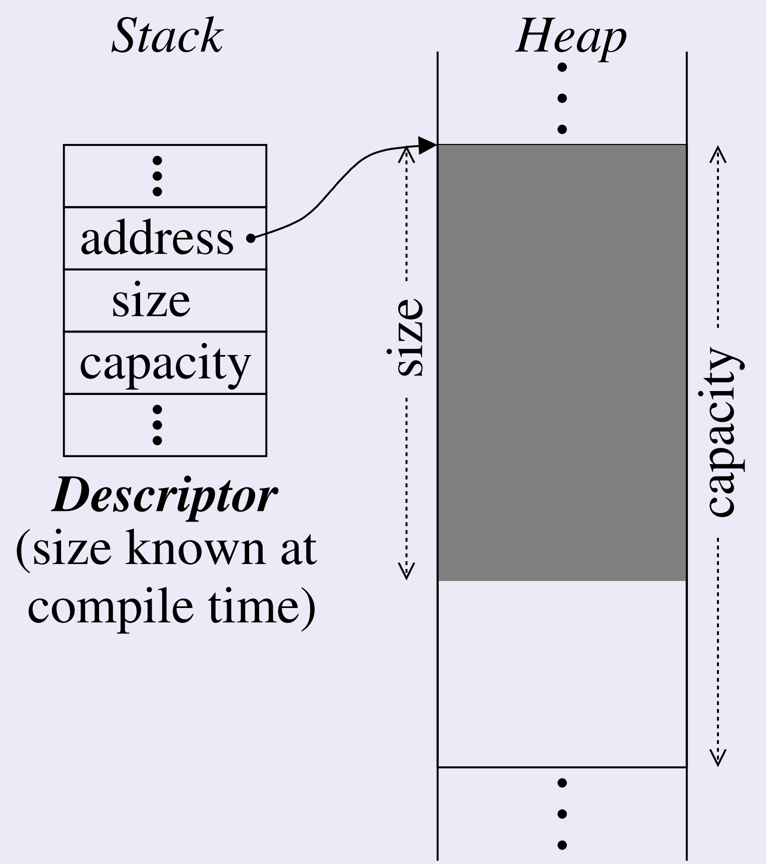
Dynamic storage allocation

Memory allocation for dynamic data structures is not easy. A combination of *static* and *dynamic* allocation is typically used:

- A fixed-sized descriptor of the data structure is allocated statically at compile time in the stack.
- The dynamic data storage is allocated and managed in the heap at runtime.

The heap is also used to allocate variables that outlive the call to the procedure that created them. These variables are usually managed by the program (e.g., *new/delete* statements in C++).

Ex.: STL vectors in C++



Attributes for code generation

Let us assume every node N of the syntax tree is annotated with three attributes:

- N.`addr`: denotes the address that will hold the value of N
  (e.g., an entry in the symbol table).
- N.`offset`: denotes the offset applied to the address (for array/record access).
- N.`code`: a string holding the three-address code associated to N.

Let us assume that top is the current symbol table and top.get(name) returns the symbol table entry associated to the string name.

Three functions will be used for code generation that will define the previous attributes:

- EvalExpr(`E`) will generate code to evaluate an expression.
- EvalAddr(`E`) will generate code to calculate the address of an expression.
- Execute(S) will generate code to execute a statement.

Code generation for expressions

EvalExpr(E) calculates the attributes E.`addr` and E.`code` of an expression. A grammar rule as parameter denotes the semantic contents of node E, e.g.,

$$\texttt{EvalExpr}(\texttt{E} \to \texttt{E}_1 + \texttt{E}_2)$$

denotes the call to EvalExpr(E), where E is a syntax-tree node representing the sum of two expressions.

A simple expression

```
EvalExpr(E → E₁ + E₂)
  EvalExpr(E₁); EvalExpr(E₂); E.addr = new Temp();
  E.code = E₁.code || E₂.code || "E.addr = E₁.addr + E₂.addr";
```

Let us assume that type(E$_1$)=float and type(E$_2$)=int. Then:

```
EvalExpr(E → E₁ + E₂)
  EvalExpr(E₁); EvalExpr(E₂);
  E.addr = new Temp(); t = new Temp();
  E.code = E₁.code || E₂.code ||
           "t = int2float E₂.addr" || "E.addr = E₁.addr + t";
```

Code generation for expressions

Pointers:

```
EvalExpr(E → *E₁)
  EvalExpr(E₁);
  E.addr = new Temp();
  E.code = E1.code || "E.addr = *E₁.addr";

EvalExpr(E → &E₁)
  EvalExpr(E₁);
  E.addr = new Temp();
  E.code = E₁.code || "E.addr = &E₁.addr";
```

Code generation to calculate addresses

Identifiers:

```
EvalAddr(E → id)
  E.addr = top.get(id.lexeme); E.code = ""; E.offset = 0;
```

Arrays:

```
EvalAddr(E → E1[E2])
  t = new Temp(); EvalAddr(E1); EvalExpr(E2);
  E.addr = E1.addr; E.offset = t;
  E.code = E1.code || E2.code ||
           "t = E2.addr * SizeElem(E1)" ||
           "t = t + E1.offset";
```

Records:

```
EvalAddr(E → E1.field)
  t = new Temp(); EvalAddr(E1); E.addr = E1.addr; E.offset = t;
  E.code = E1.code ||
         "E.offset = E1.offset + Offset(E1, field)";
```

Code generation for expressions

Identifiers:

```
EvalExpr(E → id) ≡ EvalAddr(E);
```

Arrays and records:

```
EvalExpr(E → E₁[E₂] | E₁.field)
  EvalAddr(E); t = new Temp();
  E.code = E.code || "t = E.addr[E.offset]";
  E.addr = t; E.offset = 0;
```

Code generation for statements

Assignment

```
Execute(S → E₁ := E₂)
  EvalAddr(E₁); EvalExpr(E₂);
  S.code = E₁.code || E₂.code ||
           "E₁.addr[E₁.offset] = E₂.addr";
```

In case $\texttt{E}_1\texttt{.offset = 0}$, this reduces to

```
Execute(S → E₁ := E₂)
  EvalAddr(E₁); EvalExpr(E₂);
  S.code = E₁.code || E₂.code || "E₁.addr = E₂.addr";
```

Execution order: are these two generated codes equivalent?

```
S.code = E₁.code || E₂.code || "E₁.addr = E₂.addr";

S.code = E₂.code || E₁.code || "E₁.addr = E₂.addr";
```

Code generation for statements

while B **do** S: Two possible schemes

```
L1: t = eval(B)
    ifFalse t goto L2
    S
    goto L1
L2:
```

Code generation for statements

while B **do** S: Two possible schemes

```
L1: t = eval(B)
    ifFalse t goto L2
    S
    goto L1
L2:
```

```
    goto L2
L1: S
L2: t = eval(B)
    if t goto L1
```

Code generation for statements

while B **do** S: Two possible schemes

```
L₁: t = eval(B)
    ifFalse t goto L₂
    S
    goto L₁
L₂:
```

```
    goto L₂
L₁: S
L₂: t = eval(B)
    if t goto L₁
```

Code generation for loops

```
Execute(I → while B do S)
  L₁ = new Label(); L₂ = new Label();
  EvalExpr(B); Execute(S);
  I.code = "L₁:" || B.code ||
           "ifFalse B.addr goto L₂" ||
           S.code || "goto L₁" || "L₂:";
```

Code generation for statements

if B **then** S

```
        t = eval(B)
        ifFalse t goto L
        S
     L:
```

Code generation for if-then

```
Execute(I → if B then S)
  L = new Label();
  EvalExpr(B); Execute(S);
  I.code = B.code || "ifFalse B.addr goto L" ||
           S.code || "L:";
```

Code generation for statements

if B **then** S_1 **else** S_2

```
        t = eval(B)
        ifFalse t goto L₁
        S₁
        goto L₂
    L₁:
        S₂
    L₂:
```

Code generation for if-then-else

```
Execute(I → if B then S₁ else S₂)
  L₁ = new Label(); L₂ = new Label();
  EvalExpr(B); Execute(S₁); Execute(S₂);
  I.code = B.code || "ifFalse B.addr goto L₁" || S₁.code
            || "goto L₂" || L₁:" || S₂.code || "L₂:";
```

Code generation for a function call

$f(E_1, \ldots, E_n)$

$$
\begin{aligned}
&\mathtt{t_1} = \mathtt{EvalExpr/EvalAddr}(E_1)\\
&\ldots\\
&\mathtt{t_n} = \mathtt{EvalExpr/EvalAddr}(E_n)\\
&\mathtt{param}\ t_1\\
&\ldots\\
&\mathtt{param}\ t_n\\
&t = \mathtt{call}\ f, n
\end{aligned}
$$

- The value or address of each expression will be evaluated depending on whether the parameter is passed by value or reference.
- Only addressable expressions can be passed by reference.
- No result is returned in case of a procedure call.
- The parameter count is required. Function calls can be nested, e.g. $f(w, g(x, y), z)$.

Evaluating Boolean expressions

Two schemes (semantically different):

- Eager evaluation: all operands are evaluated.
- Lazy (short-circuit) evaluation: the second operand is only evaluated if the value of the first one is not sufficient to determine the value of the expression.

E_1 and E_2

Eager evaluation

```
t1 = EvalExpr(E1)
t2 = EvalExpr(E2)
t3 = t1 and t2
```

E1 or E2

Eager evaluation

```
t1 = EvalExpr(E1)
t2 = EvalExpr(E2)
t3 = t1 or t2
```

Evaluating Boolean expressions

Two schemes (semantically different):

- Eager evaluation: all operands are evaluated.
- Lazy (short-circuit) evaluation: the second operand is only evaluated if the value of the first one is not sufficient to determine the value of the expression.

E_1 and E_2

Eager evaluation

```
  t1 = EvalExpr(E1)
  t2 = EvalExpr(E2)
  t3 = t1 and t2
```

Lazy evaluation

```
  t = EvalExpr(E1)
  ifFalse t goto L
  t = EvalExpr(E2)
L:
```

E_1 or E_2

Eager evaluation

```
  t1 = EvalExpr(E1)
  t2 = EvalExpr(E2)
  t3 = t1 or t2
```

Lazy evaluation

```
  t = EvalExpr(E1)
  if t goto L
  t = EvalExpr(E2)
L:
```

Boolean conditions and jumps

Example: **while** `not` $(E_1 < E_2)$ `and` $(E_3 = E_4)$ **do** S;

Naïve code

```
While:
   t1 = EvalExpr(E1)
   t2 = EvalExpr(E2)
   t5 = (t1 < t2)
   t6 = not t5
   ifFalse t6 goto EndCond
   t3 = EvalExpr(E3)
   t4 = EvalExpr(E4)
   t6 = (t3 = t4)
EndCond:
   ifFalse t6 goto Endwhile
   S
   goto While
EndWhile:
```

Efficient code

```
While:
   t1 = EvalExpr(E1)
   t2 = EvalExpr(E2)
   if t1 < t2 goto EndWhile
   t3 = EvalExpr(E3)
   t4 = EvalExpr(E4)
   ifFalse t3 = t4 goto EndWhile
   S
   goto While
EndWhile:
```

How to generate efficient code for Boolean conditions combined with jumps?

Backpatching

A method to generate efficient code for Boolean expressions.

EvalBoolExpr(E, L_t, L_f)

Evaluates a Boolean expression generating code to jump to L_t or L_f in case the condition evaluates to true or false, respectively.

EvalBoolExpr(E → E_1 and E_2, L_t, L_f) (lazy evaluation)

```
Lf1 = new Label();
EvalBoolExpr(E1, Lf1, Lf);
EvalBoolExpr(E2, Lt, Lf);
E.code = E1.code || "Lf1:" || E2.code;
```

EvalBoolExpr(E → E_1 or E_2, L_t, L_f) (lazy evaluation)

```
Lf1 = new Label();
EvalBoolExpr(E1, Lt, Lf1);
EvalBoolExpr(E2, Lt, Lf);
E.code = E1.code || "Lf1:" || E2.code;
```

Backpatching

EvalBoolExpr($E \rightarrow$ not E_1, L_t, L_f)

```
EvalBoolExpr(E1, Lf, Lt);
E.code = E1.code;
```

EvalBoolExpr($E \rightarrow E_1$ relop E_2, L_t, L_f)

```
EvalExpr(E1);
EvalExpr(E2);
E.code = E1.code ||
         E2.code ||
         "if E1.addr relop E2.addr goto Lt" ||
         "goto Lf";
```

Jump optimization

Some generated jumps are redundant and can be easily optimized, e.g., jumps to the next instruction, jumps to unconditional/conditional jumps, etc.
(see code optimization for more details)

Backpatching: revisiting **while** and **if-then-else**

Execute(I → while B do S)

```
L₁ = new Label();
L₂ = new Label();
L₃ = new Label();
EvalBoolExpr(B, L₂, L₃);
Execute(S);

I.code = "L₁:" ||
               B.code ||
         "L₂" ||
               S.code ||
               "goto L₁" ||
         "L₃:";
```

Execute(I → if B then S_1 else S_2)

```
L₁ = new Label();
L₂ = new Label();
L₃ = new Label();
EvalBoolExpr(B, L₁, L₂);
Execute(S₁); Execute(S₂);

I.code =      B.code ||
         "L₁:" ||
              S₁.code ||
              "goto L₃" ||
         "L₂:" ||
              S₂.code ||
         "L₃:";
```

Switch statements

- Implemented with a sequence of conditional jumps.
- It can be inefficient if the number of branches is large.

$$
\begin{array}{l}
\texttt{switch (E) \{} \\
\quad \texttt{case } v_1\texttt{: } S_1 \\
\quad \texttt{case } v_2\texttt{: } S_2 \\
\qquad \vdots \\
\quad \texttt{case } v_{n-1}\texttt{: } S_{n-1} \\
\quad \texttt{default: } S_n \\
\texttt{\}}
\end{array}
\quad \Rightarrow \quad
\begin{array}{l}
\texttt{t = EvalExpr(E);} \\
\texttt{if (t == } v_1\texttt{) } S_1\texttt{;} \\
\texttt{else if (t == } v_2\texttt{) } S_2\texttt{;} \\
\qquad \vdots \\
\texttt{else if (t == } v_{n-1}\texttt{) } S_{n-1}\texttt{;} \\
\texttt{else } S_n\texttt{;}
\end{array}
$$

Switch statements with computed jumps

Table of computed jumps

JumpTable:

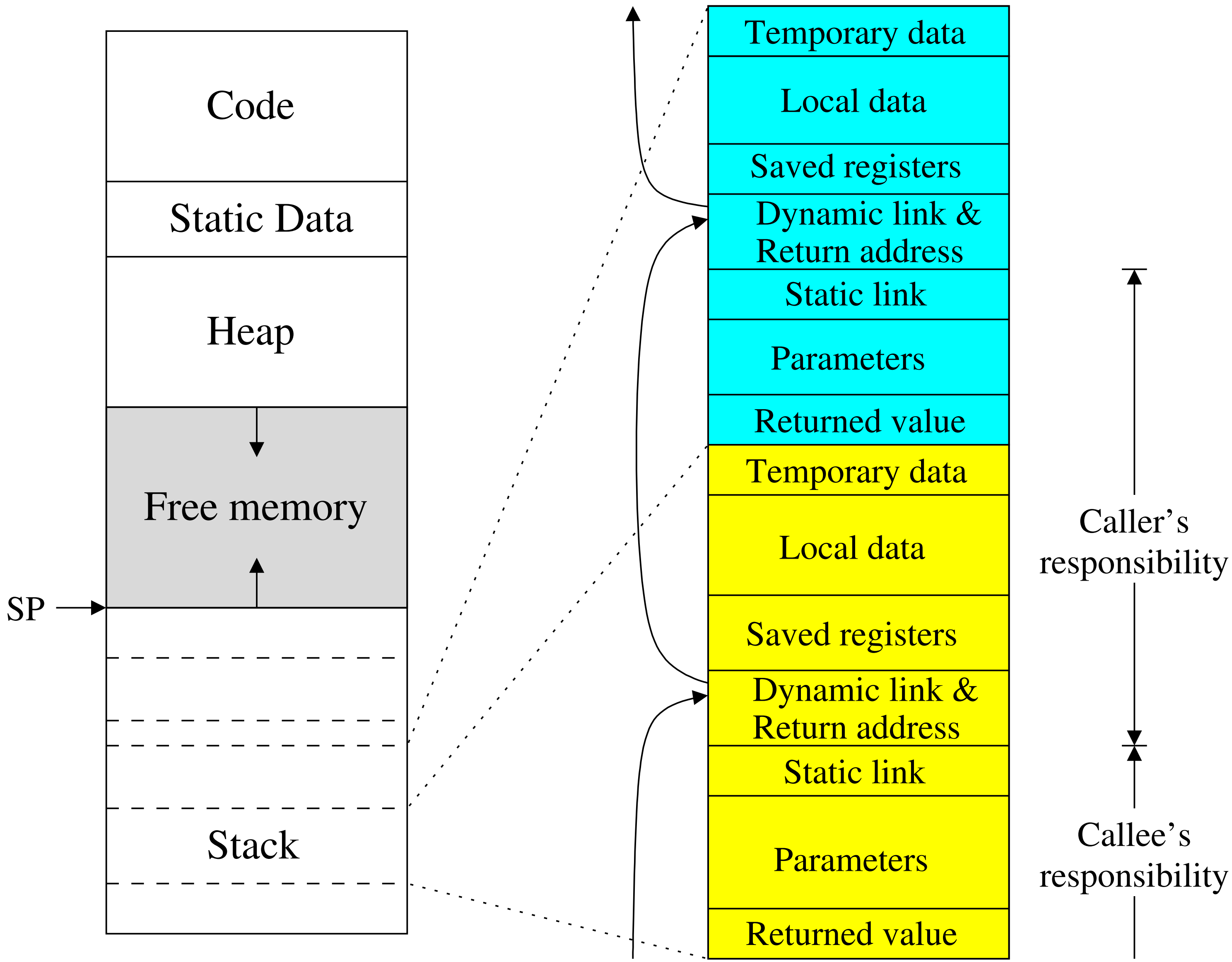
v_1 :	$\mathtt{L}_1$
	...
	$\mathtt{L}_n$
	...
v_2 :	$\mathtt{L}_2$
	...
	$\mathtt{L}_n$
	...
v_3 :	$\mathtt{L}_3$
	⋮
	⋮
v_{n-1} :	$\mathtt{L}_{n-1}$

Code (assuming $v_1 < v_2 < \ldots < v_{n-1}$)

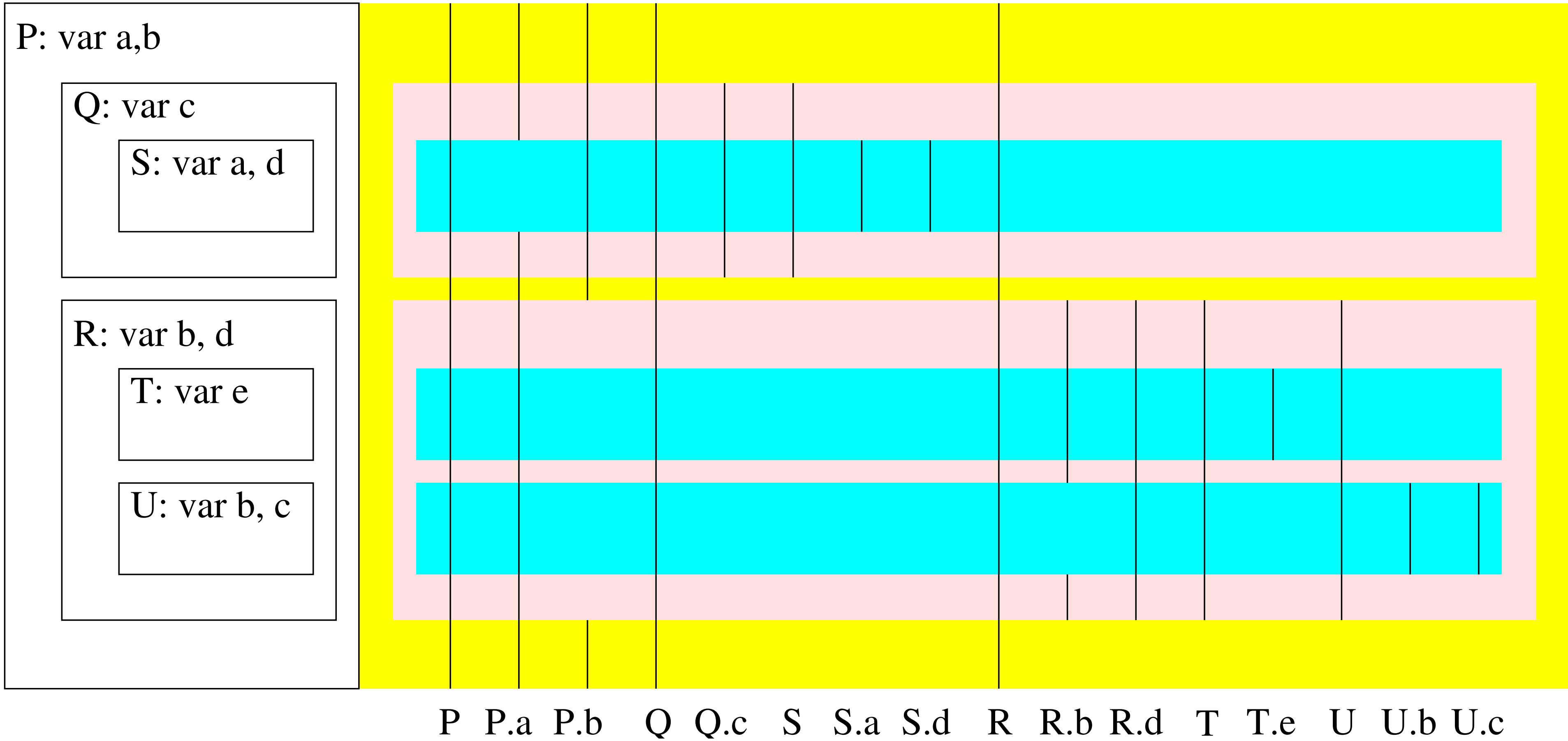
```
        t = EvalExpr(E);
        if (t < v1 or t > vn-1) goto Ln;
        goto JumpTable[t-v1];

L1:     S1; goto endsw;
L2:     S2; goto endsw;
...
Ln-1:   Sn-1; goto endsw;
Ln:     Sn;

endsw:
```

Hybrid approaches are also possible, e.g., using hash tables or several tables with min/max buckets.

Storage organization

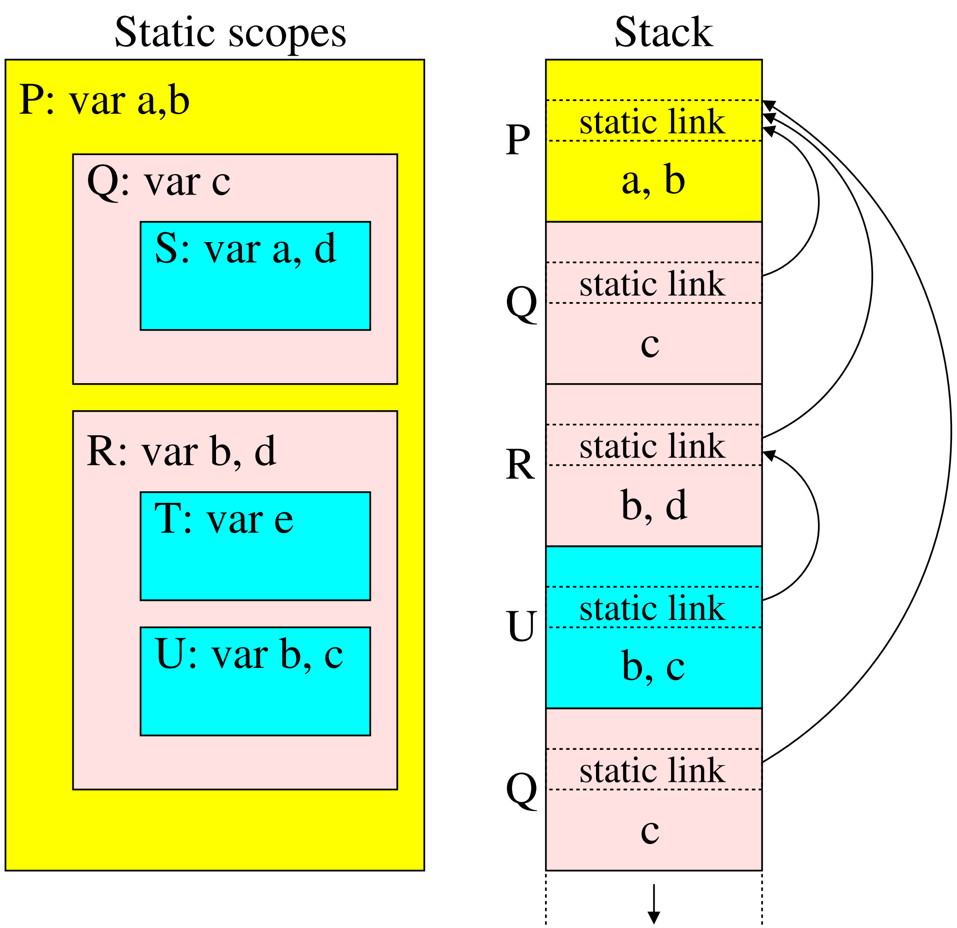


Static scopes



- How to read/write variables stored at different activation records?
- How to manage the static links?

Static scopes



- Each activation record has a static link pointing at the AR of the static ancestor in the source code.
- At every function call, the caller passes the static link as an *implicit* parameter.
- Every access to a non-local variable requires to traverse the chain of static links until the AR storing the variable is reached.

Static scopes

Function P calls $Q(x_1, \ldots, x_n)$.
We know that $depth(Q) - depth(P) \leq 1$

Code

$$
\left.\begin{array}{l}
\mathtt{t_{sl}} = \mathtt{*t_{sl}} \\
\quad\vdots \\
\mathtt{t_{sl}} = \mathtt{*t_{sl}}
\end{array}\right\} depth(P) - depth(Q) + 1
$$

```
t_sl = &static_link
t_sl = *t_sl
   ⋮
t_sl = *t_sl
param t_sl
param t_1
   ⋮
param t_n
call Q, n+1
```

Static scopes

Function P calls $Q(x_1, \ldots, x_n)$.
We know that $depth(Q) - depth(P) \leq 1$

Code

$$\left.\begin{array}{l}\texttt{t}_{\texttt{sl}} = \texttt{\&static\_link} \\ \left.\begin{array}{l}\texttt{t}_{\texttt{sl}} = *\texttt{t}_{\texttt{sl}} \\ \quad\vdots \\ \texttt{t}_{\texttt{sl}} = *\texttt{t}_{\texttt{sl}}\end{array}\right\} depth(P) - depth(Q) + 1 \\ \texttt{param } \texttt{t}_{\texttt{sl}} \\ \texttt{param } \texttt{t}_1 \\ \quad\vdots \\ \texttt{param } \texttt{t}_n \\ \texttt{call } Q, n+1\end{array}\right.$$

Case $depth(Q) = depth(P) + 1$

$$\begin{array}{l}\texttt{t}_{\texttt{sl}} = \texttt{\&static\_link} \\ \texttt{param } \texttt{t}_{\texttt{sl}} \\ \quad\vdots\end{array}$$

Case $depth(Q) < depth(P) + 1$

$$\begin{array}{l}\texttt{t}_{\texttt{sl}} = \texttt{static\_link} \\ \left.\begin{array}{l}\texttt{t}_{\texttt{sl}} = *\texttt{t}_{\texttt{sl}} \\ \quad\vdots \\ \texttt{t}_{\texttt{sl}} = *\texttt{t}_{\texttt{sl}}\end{array}\right\} depth(P) - depth(Q) \\ \texttt{param } \texttt{t}_{\texttt{sl}} \\ \quad\vdots\end{array}$$

Static scopes

Access to variable v declared in P while executing function Q.
We know that $depth(Q) - depth(P) \leq 1$

Code (assume non-local access: $depth(Q) > depth(P)$)

$$
\begin{aligned}
\mathtt{t}_{\mathtt{sl}} &= \mathtt{static\_link} \\
\left.\begin{aligned}
\mathtt{t}_{\mathtt{sl}} &= *\mathtt{t}_{\mathtt{sl}} \\
&\vdots \\
\mathtt{t}_{\mathtt{sl}} &= *\mathtt{t}_{\mathtt{sl}}
\end{aligned}\right\} & \, depth(Q) - depth(P) - 1 \\
\mathtt{t}_{\mathtt{sl}} &= \mathtt{t}_{\mathtt{sl}} + \mathtt{offset}(P :: v)
\end{aligned}
$$

$\mathtt{t}_{\mathtt{sl}}$ contains the address of variable v

Exercises

Exercises

Generate code for the following statements:

1. `a = b[i] + c[i]`
2. `a[i] = b*c - b*d`
3. `x = f(y+1) + 2`
4. `x = *p + &y`

Exercises

Generate code for the following C++ function:

```
int search(int *a, int n, int x) {
  int i = 0;
  while (i < n and a[i] != x) i = i + 1;
  if (i < n) return i;
  return -1;
}
```

Note: Use backpatching for the evaluation of Boolean expressions

Exercises

Consider the following data structure:

```
struct {
    double v[100];      // sizeof(double)=8
    struct {
        int a, b;       // sizeof(int)=4
        tree *n;        // sizeof(void *)=8
    } m[10];
} s[1000];
```

- What is the size of variable *s*?

Generate code for the following statement:

```
s[i].v[j] = s[k+1].m[k].b * 0.5;
```

Exercises

Consider the following C++ code:

```
int f(vector<int>& A, double d) {
  struct { double a; int b; } s;
  vector<int> X(100);
  int i;
  ...
  L1:
  X.resize(200);
  ...
  while (i > 0) {
    int j = 0;
    vector<double> M(i);
    L2:
    ...
  }
  ...
  L3:
  return X[0];
}
```

- Describe the contents of the symbol table when compiling the code at lines L1, L2 and L3.
- Describe the allocated memory (stack and heap) at the same lines during the execution of the program.
- Describe a situation in which a program could generate *memory leaks*, i.e., not all the allocated objects are deallocated before the completion of the program.
- Consider a C++ program that never uses the *new/delete* statements. Can it generate memory leaks? Reason your answer.

Exercises

De Morgan's law says that $\neg(p \vee q) \equiv \neg p \wedge \neg q$.

Show that the code generated by the following two statements (using backpatching) is identical:

- **if** not (p or q) **then** S_1 **else** S_2
- **if** not p and not q **then** S_1 **else** S_2

Exercises

Consider the following code with nested static scopes:

```
func P (int a) {
    int x;
    func Q (int b) {
        int y;
        x = a + y;
    }
    func R () {
        int a;
        func S() {
            int c;
            c = a - x;
            P(a);
        }
    }
}
```

- Generate code for the three statements in red.