Compiler Construction

Chapter 5: Syntax-driven translation

Dittaya Wanvarie

Department of Mathematics and Computer Science
Chulalongkorn University

Second semester, 2024

Outline

Introduction

Compiler's tasks

1. Read the source
2. Build intermediate representation: assemble knowledge on the input
   - Names, types, syntactic structures, etc.
   - Symbol tables
3. Translate into the target

How to build an IR

- Syntax-driven translation: added actions into a parser
- IR traversal to build another IR

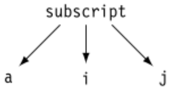
Necessary information for translation

- IR building process
- Name visibility, scope, name biding
- Programming language construct such as variable reference, case statement, heap allocation

We can perform these actions as a side-effect during parsing

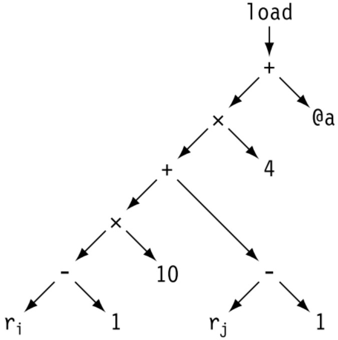
Example: necessary information

Representing `a[i][j]`



Source-Level Tree

As seen in the code text



Low-Level Tree

To generate assembly codes

Example: necessary information

Reference to `x`

- Map `x` to its appropriate runtime instance
  - Name resolution
- Know type, size, structure, and lifetime of `x`
  - E.g. cannot treat a floating point as a memory reference
- Memory location

Outline

Example: positive integer grammar

Parsing **175** and get its value

1	*Number*	$\rightarrow$	*DList*	
2	*DList*	$\rightarrow$	*DList* `digit`	
3		|	`digit`	

(a) The Positive Integer Grammar

	Action		Goto
State	eof	digit	*DList*
0		s 2	1
1	acc	s 3	
2	r 3	r 3	
3	r 2	r 2	

(b) Its *Action* and *Goto* Tables

■ **FIGURE 5.1** The Grammar for Positive Integers.

Example of side actions

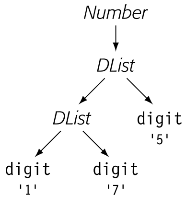
```
Number  : DList          { return $1; } ;
DList   : DList digit    { $$ = $1 * 10 + CToI($2); } ;
        | digit          { $$ = CToI($1); } ;
```

Example: positive integer grammar

Iteration	State	Word	Stack	Action
0	–	1	$ ⟨*Number* 0⟩	*shift 2*
1	2	7	$ ⟨*Number* 0⟩ ⟨digit 2⟩	*reduce 3*
2	1	7	$ ⟨*Number* 0⟩ ⟨*DList* 1⟩	*shift 3*
3	3	5	$ ⟨*Number* 0⟩ ⟨*DList* 1⟩ ⟨digit 3⟩	*reduce 2*
4	1	5	$ ⟨*Number* 0⟩ ⟨*DList* 1⟩	*shift 3*
5	3	eof	$ ⟨*Number* 0⟩ ⟨*DList* 1⟩ ⟨digit 3⟩	*reduce 2*
6	1	eof	$ ⟨*Number* 0⟩ ⟨*DList* 1⟩	*accept*

■ **FIGURE 5.2** Parser Actions for the Number "175".

Example: positive integer grammar



(a) Syntax Tree for "175" with the Left-recursive Grammar

Value(root)

    if (root is type Number*) then*

        return Value(son(root))

    else if (root is type DList*) then*

        return $10 \times$ *Value(left(root))*

            $+$ *Value(right(root))*

    else if (root is type `digit`*) then*

        return integer(root's lexeme)

(b) Treewalk to Compute the Value from the Syntax Tree

■ **FIGURE 5.3** Treewalk Computations for the Positive Integer Grammar.

Form of the grammar

Which one is better?

A)

```
Number  : DList          { return $1; } ;
DList   : DList digit    { $$ = $1 * 10 + CToI($2); } ;
        | digit          { $$ = CToI($1); } ;
```

B)

```
Number  : DList          { return second($1); } ;
DList   : digit DList    { $$ = pair(10 * first($2),
                               first($2) * CToI($1) + second($2)); }
        | digit          { $$ = pair(10,CToI($1)); } ;
```

We may have to rewrite the grammar for simpler and faster computation

Example: construct an AST

Production			Syntax-Driven Actions
Expr	$\rightarrow$	*Expr* + *Term*	`{ $$ ←` *MakeNode2*`(plus, $1, $3); };`
	|	*Expr* − *Term*	`{ $$ ←` *MakeNode2*`(minus, $1, $3); };`
	|	*Term*	`{ $$ ← $1; };`
Term	$\rightarrow$	*Term* × *Factor*	`{ $$ ←` *MakeNode2*`(times, $1, $3); };`
	|	*Term* ÷ *Factor*	`{ $$ ←` *MakeNode2*`(divide, $1, $3); };`
	|	*Factor*	`{ $$ ← $1; };`
Factor	$\rightarrow$	(*Expr*)	`{ $$ ← $2; };`
	|	`number`	`{ $$ ←` *MakeLeaf*`(number,` *lexeme*`); };`
	|	`name`	`{ $$ ←` *MakeLeaf*`(name,` *lexeme*`); };`

■ **FIGURE 5.4** Building an Abstract Syntax Tree.

Example: `a - 2 x b`

Example: translate to 3-address code

Example: `a - 2 x b`

Production			Syntax-Driven Actions
Expr	$\rightarrow$	*Expr* + *Term*	{ \$\$ ← *NextRegister*() ; *Emit*(add, \$1, \$3, \$\$) ; };
	|	*Expr* – *Term*	{ \$\$ ← *NextRegister*() ; *Emit*(sub, \$1, \$3, \$\$) ; };
	|	*Term*	{ \$\$ ← \$1 ; };
Term	$\rightarrow$	*Term* × *Factor*	{ \$\$ ← *NextRegister*() ; *Emit*(mult, \$1, \$3, \$\$) ; };
	|	*Term* ÷ *Factor*	{ \$\$ ← *NextRegister*() ; *Emit*(div, \$1, \$3, \$\$) ; };
	|	*Factor*	{ \$\$ ← \$1 ; };
Factor	$\rightarrow$	(*Expr*)	{ \$\$ ← \$2 ; };
	|	number	{ \$\$ ← *NumberIntoReg*(*lexeme*) ; };
	|	name	{ entry ← *STLookup*(*lexeme*); \$\$ ← *ValueIntoReg*(entry) ; };

■ **FIGURE 5.5** Emitting Three-Address Code for Expressions.

loadAI	r_{arp}, @a	$\Rightarrow$	r_1
loadI	2	$\Rightarrow$	r_2
loadAI	r_{arp}, @b	$\Rightarrow$	r_3
mult	r_2, r_3	$\Rightarrow$	r_4
sub	r_1, r_4	$\Rightarrow$	r_5

ILOC Code for a - 2 ×b

SDT in LR(1) parser

push ⟨*INVALID, INVALID,* ***INVALID***⟩ *onto the stack*
push ⟨*start symbol,* s_0*,* ***INVALID***⟩ *onto the stack*
word ← *NextWord( )*

while (true) do
    state ← *state from* ***triple*** *at top of stack*

    if Action[state,word] = "reduce $A \to \beta$*" then*
        value ← ***PerformActions(***$A \to \beta$***)***
        pop $|\beta|$ ***triples*** *from the stack*
        state ← *state from* ***triple*** *at top of stack*
        push ⟨*A, Goto[state, A],* ***value***⟩ *onto the stack*

    else if Action[state,word] = "shift s_i*" then*
        push ⟨*word,* s_i*,* ***lexeme***⟩ *onto the stack*
        word ← *NextWord( )*

    else if Action[state,word] = "accept" and word = `eof`
        then break

    else throw a syntax error

report success / executed the "accept" case */*

■ **FIGURE 5.6** The Skeleton LR(1) Parser with Translation Support.

Handling nonlocal computation

What if the information is not in the syntax tree?

- E.g. type, lifetime, visibility

Solution

1. Use global variable, `CurType`

$$
\begin{array}{lcll}
\textit{Declaration} & \rightarrow & \textit{TypeSpec NameList} & \{\ \texttt{CurType} \leftarrow \texttt{invalid;}\ \}; \\
\textit{TypeSpec} & \rightarrow & \texttt{int} & \{\ \texttt{CurType} \leftarrow \texttt{int;}\ \}; \\
 & | & \texttt{float} & \{\ \texttt{CurType} \leftarrow \texttt{float;}\ \}; \\
\textit{NameList} & \rightarrow & \textit{NameList}\ \texttt{,}\ \texttt{name} & \{\ \texttt{err} \leftarrow \texttt{SetType(\$2,CurType);}\ \}; \\
 & | & \texttt{name} & \{\ \texttt{err} \leftarrow \texttt{SetType(\$1,CurType);}\ \};
\end{array}
$$

2. Change grammar, avoid overriding `CurType`

$$
\begin{array}{lcll}
\textit{Declaration} & \rightarrow & \texttt{int}\ \textit{INameList} & \\
 & | & \texttt{float}\ \textit{FNameList} & \\
\textit{INameList} & \rightarrow & \textit{NameList}\ \texttt{name} & \{\ \texttt{err} \leftarrow \texttt{SetType(\$2, int);}\ \}; \\
 & | & \texttt{name} & \{\ \texttt{err} \leftarrow \texttt{SetType(\$1, int);}\ \}; \\
\textit{FNameList} & \rightarrow & \textit{NameList}\ \texttt{name} & \{\ \texttt{err} \leftarrow \texttt{SetType(\$2, float);}\ \}; \\
 & | & \texttt{name} & \{\ \texttt{err} \leftarrow \texttt{SetType(\$1, float);}\ \};
\end{array}
$$

Local context limitation

`a x 2`

1. Load constant to a register, then perform multiplication

```
loadI   2       ⇒ r_i
mult    r_a, r_i ⇒ r_j
```

ILOC Code for a × 2

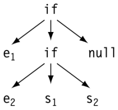
2. Use `multI`
   - Current grammar:
     - $Term \rightarrow Term$ **x** $Factor$
     - $Factor \rightarrow$ **number**
   - Changed grammar:
     - $Term \rightarrow Term$ **x number**
     - $Term \rightarrow Term$ **x name**

However, the new grammar does not support `2 x a`. One better solution is to delay the optimization to later stages.

Control-flow statements

If structure

`if` e_1 `then if` e_2 `then` s_1 `else` s_2



AST for Nested If Statements

1. Evaluation the conditional expression (e_1)
2. Branch to s_1 section if the condition is true, s_2 otherwise. We need the labels of the first command in s_1 and s_2 parts.
3. At the end of each subpart (s_1 and s_2), jump to the next statement ($exit$). We also need the label for this statement.

Modifying production rules

From the if-else production rules

1	*Stmt*	→	`if` *Expr* `then` *Stmt*
2		|	`if` *Expr* `then` *WithElse* `else` *Stmt*
3		|	*Other*
4	*WithElse*	→	`if` *Expr* `then` *WithElse* `else` *WithElse*
5		|	*Other*

FIGURE 5.7 The Unambiguous If-Then-Else Grammar.

We need to create the labels s_1, s_2, $exit$ when we start working on the statement. We have to pass this label information to each subpart.

Modifying production rules

We may construct an object for each information and embed it to the variable.

- However, actions for each piece of information occur at different time.

Solution

- Then, we construct an empty string production for the actions.

$$
\begin{array}{lcl}
\mathit{WithElse} & \rightarrow & \texttt{if}\ \mathit{Expr}\ \mathit{CreateBranch} \\
 & & \quad \texttt{then}\ \mathit{WithElse}\ \mathit{ToExit1} \\
 & & \quad \texttt{else}\ \mathit{WithElse}\ \mathit{ToExit2} \\
\mathit{CreateBranch} & \rightarrow & \epsilon \\
\mathit{ToExit1} & \rightarrow & \epsilon \\
\mathit{ToExit2} & \rightarrow & \epsilon
\end{array}
$$

■ **FIGURE 5.8** Creating Mid-Production Actions.

Outline

Attribute Grammar

For a context-free grammar

- Each symbol associates with one or more attributes
- Each production is augmented with attribute computations
- There is no evaluation order between computations for the same production

Evaluation of attributes

Evaluation order? Dependency?

- We may evaluate the attribute during parsing if all required values are available when we construct the node

Synthesized attribute

- An attribute defined wholly in terms of the attributes of the node, **its children**, and constants
- We can compute this attribute during **bottom-up** parsing

Inherited attribute

- An attribute defined wholly in terms of the node's own attributes and those of its **siblings** or **its parent** in the parse tree, and constants
- We can compute this attribute during **top-down** parsing

Naming values

YACC notation

- `$$` refers to the result location for the current production, i.e. the left-hand-side symbol.
- `$1`, `$2`, . . . , `$n` refer to the locations for the first, second, through the `$n` symbol on the right-hand side.

Object-oriented notation

- Simply refer to a value as an attribute of the variable.
- E.g. $Number.value$

Exercise: Signed binary numbers

Set the value of $Number$

	Production			Code snippet
1	$Number$	$\rightarrow$	$Sign\ List$	
2	$Sign$	$\rightarrow$	+	
3	$Sign$	$\rightarrow$	-	
4	$List$	$\rightarrow$	Bit	
5	$List_0$	$\rightarrow$	$List_1\ Bit$	
6	Bit	$\rightarrow$	0	
7	Bit	$\rightarrow$	1	

Outline

Modeling the naming environment

Name resolution

- Static binding: determine the name-to-entity binding at the **compile time**
- Dynamic binding: defer the name-to-entity binding until **runtime**

Lexical hierarchies

Rule

At a point p in a program, an occurrence of name n refers to the entity named n that was created, explicitly or implicitly, in the scope that is lexically closest to p.

Handling scopes

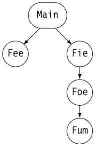
Static coordinate

```
program Main0(input0, output0);
  var x1, y1, z1: integer;
  procedure Fee1;
     var x2: integer;
     begin { Fee1 }
        x2 := 1;
        y1 := x2 * 2 + 1
     end;
  procedure Fie1;
     var y2: real;
     procedure Foe2;
        var z3: real;
           procedure Fum3;
              var y4: real;
              begin { Fum3 }
                 x1 := 1.25 * z3;
                 Fee1;
                 writeln('x = ',x1)
              end;
        begin { Foe2 }
           z3 := 1;
           Fee1;
           Fum3
        end;
     begin { Fie1 }
        Foe2;
        writeln('x = ',x1)
     end;
  begin { Main0 }
     x1 := 0;
     Fie1
  end.
```

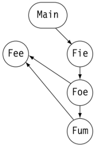
(a) PASCAL Program

Scope	x	y	z
Main	$\langle 1,0\rangle$	$\langle 1,4\rangle$	$\langle 1,8\rangle$
Fee	$\langle 2,0\rangle$	$\langle 1,4\rangle$	$\langle 1,8\rangle$
Fie	$\langle 1,0\rangle$	$\langle 2,0\rangle$	$\langle 1,8\rangle$
Foe	$\langle 1,0\rangle$	$\langle 2,0\rangle$	$\langle 3,0\rangle$
Fum	$\langle 1,0\rangle$	$\langle 4,0\rangle$	$\langle 3,0\rangle$

(b) Static Coordinates



(c) Nesting Relationships



(d) Calling Relationships

■ **FIGURE 5.9** Nested Lexical Scopes in PASCAL.

Modeling lexical scopes

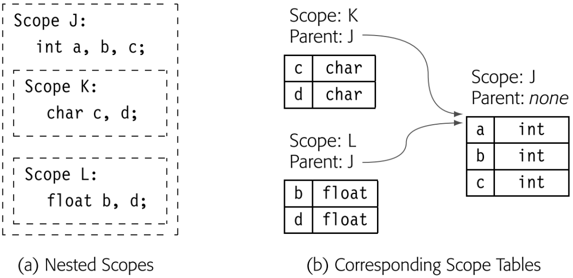


FIGURE 5.10 Tables for the Lexical Hierarchy.

- Chains of symbol tables
- A search path

Building the model

Via syntax-driven translation

- When the source language constructs enter and leave distinct scopes
  - Create tables, search paths

Examples

- Block demarcations
  - On entry: Create a new table and link it to the current scope
  - On exit: Mark the table as final (not in use)
- Variable declarations: Create entries in a local table
  - Existing variables: Populate attributes
  - New variable: Infer attributes
- References: search for information in the local table, and the entire search path

Lexical scope: examples

C

- Procedure names and global variables exist in the global scope
- Each procedure creates its own local scope fore variables, parameters, and lables
- Blocks ({ and }), create their own local scopes
- `static` name has a global lifetime
  - `static` global name is visible inside the file, while `static` local name is visible locally

Python

- 3 kinds of scopes
  - A local function-specific scope
  - A global scope
  - A builtin scope e.g. `print`
  - The first use of `x` defines the scope of `x`
    - If it is an assignment, `x` is in a local scope
    - If it is a use, `x` is in a global scope

Inheritance hierarchies

- A subclass will **inherit** all methods and members from its superclass
- Single inheritance vs multiple inheritance
- Each class definition has its own scope, with links to its superclass
- Overriding
- The compiler must map an <object, member> pair back to a specific definition

Issues in inheritance hierarchies

- Links to its superclass(es)
- Compile-time vs. runtime resolution
- Lookup with inheritance or search paths
- Construct scope tables using syntax-driven actions

Tables for the inheritance hierarchy

```
Class Point {
    public int x, y;
    private int z;
    public void draw() {...};
    public void move() {...};
}
```

```
Class ColorPoint extends Point {
    private Color c;
    public void draw() {...};
    public void setc( Color x )
        { this.c = x };
}
```

(a) Class Definitions

Class: Point
Superclass: *none*

x	int	public
y	int	public
z	int	private
draw	void()	public
move	void()	public

Class: ColorPoint
Superclass: Point

c	Color	private
draw	void()	public
setc	void()	public

(b) Corresponding Scope Tables

FIGURE 5.11 Tables for the Inheritance Hierarchy.

Visibility

For example,

- `public`
- `private`
- `protected`
- `default`

A typical implementation will include a visibility tag in the symbol table record of each name

Outline

Type information

Fundamental properties of a name

- Storage size
- Range
- Low-level representation
- Etc.

Behaviors of the program depends on data types (overloading)

Uses for types in translation

- Conformable: the operators and arguments (operands) are well defined
- Efficient translation
- Casting
- Information for garbage collection

Type checking

- Type signature
- Function prototype

```
if (tag(a) = tag(b)) then  // take the short path
    switch (tag(a)) into {
        case SHORT:    // use SHORT add
            value(c) ← value(a) + value(b)
            tag(c) ← SHORT
            break
        case INTEGER:  // use INTEGER add
            value(c) ← value(a) + value(b)
            tag(c) ← INTEGER
            break
        case LONG INTEGER:  // use LONG INTEGER add
            value(c) ← value(a) + value(b)
            tag(c) ← LONG INTEGER
            break
    }
else    // take the long path
    (c, tag(c)) ← AddMixedTypes(a, tag(a), b, tag(b))
```

■ **FIGURE 5.12** Integer Addition with Runtime Type Checking.

Components of a type system

Base types

- Integer, floating point, string
- Multiple size such as byte, word, double word, quadruple word

Compound and constructed types

- Graphs, trees, tables, lists, stacks, maps
- Arrays
- Strings
- Enumerated types
- Structures and variants
- Objects and classes

Type equivalence

- Name equivalence
- Structural equivalence

```
struct Tree {
    int value;
    struct Tree *left;
    struct Tree *right;
}
struct Bush {
    int value;
    struct Bush *left;
    struct Bush *right;
}
```

Type inference for expressions

Production			Syntax-Driven Action
Expr	$\rightarrow$	*Expr* + *Term*	`{ set_type($$,`$\mathcal{F}_{+}$`(type($1),type($3))); };`
	|	*Expr* − *Term*	`{ set_type($$,`$\mathcal{F}_{-}$`(type($1),type($3))); };`
	|	*Term*	`{ set_type($$,type($1)); };`
Term	$\rightarrow$	*Term* × *Factor*	`{ set_type($$,`$\mathcal{F}_{\times}$`(type($1),type($3))); };`
	|	*Term* ÷ *Factor*	`{ set_type($$,`$\mathcal{F}_{\div}$`(type($1),type($3))); };`
	|	*Factor*	`{ set_type($$,type($1)); };`
Factor	$\rightarrow$	(*Expr*)	`{ set_type($$,type($2)); };`
	|	num	`{ set_type($$,type(num)); };`
	|	name	`{ set_type($$,type(name)); };`

■ **FIGURE 5.13** Framework to Assign Types to Subexpressions.

Outline

Storage classes and data areas

Automatic variables

- Identical lifetime of its declaring scope
- Stored in the local data area

Activation record

- A region of memory for an invocation of a procedure
- Activation record pointer is normally a register pointing to the first address of AR

Static variables

- The lifetime is the same of the program
- Single storage choice with visibility tag or individual data area for each variable

Storage classes and data areas

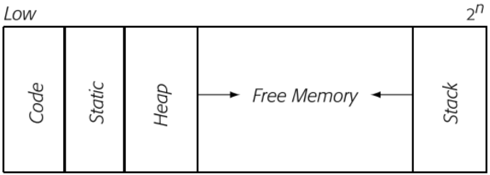
Irregular entities

- Dynamic allocation
- Stored in runtime heap

Temporary values

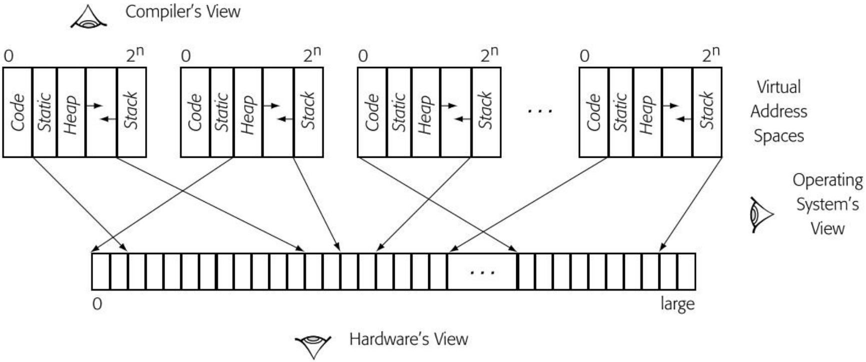
- Usually stored in registers
- Local data area if the size is known
- Heap if the size is unknown

Layout within a virtual address space



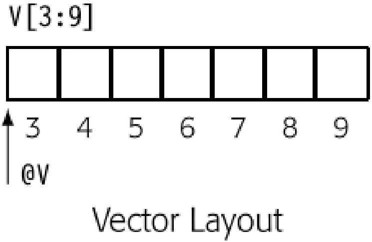
■ **FIGURE 5.14** Virtual Address-Space Layout.

Different views of the address space



■ **FIGURE 5.15** Different Views of the Address Space.

Internal layout for arrays



2-d array

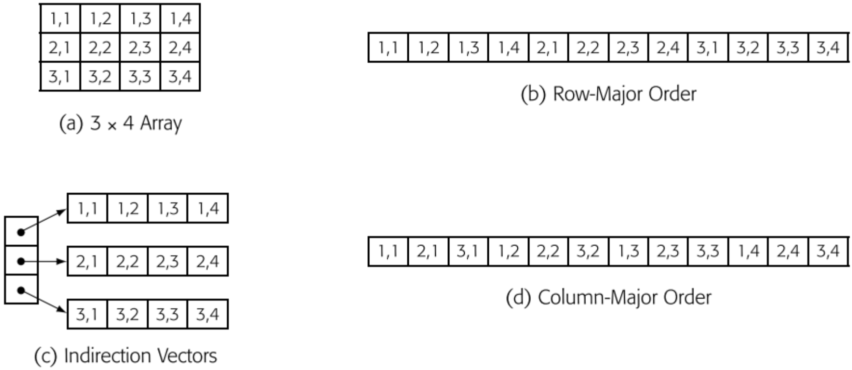
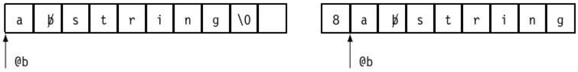


FIGURE 5.16 Two-Dimensional Array Layouts.

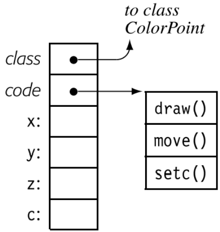
Internal layout for strings



String with Null Termination

String with Explicit Length Field

Internal layout for structures



Object Record for Instance
Of Class ColorPoint

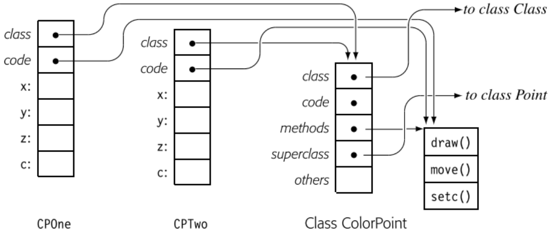
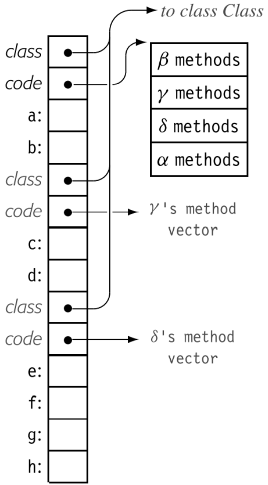


FIGURE 5.17 Multiple Instances of Class ColorPoint.

Object record for multiple inheritance



Object Record for α

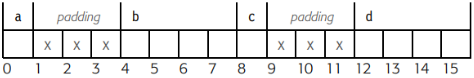
Storage assignment and translation

- Declaration before any executable statement appears
- Build up the symbol table during parsing and perform type inference on IR
- Build an IR with abstract references and perform type inference on the IR

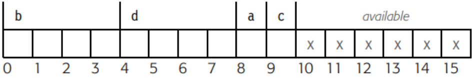
Alignment restrictions and padding

Name	Bytes	Constraint
a	1	$@a \bmod 1 = 0$
b	4	$@b \bmod 4 = 0$
c	1	$@c \bmod 1 = 0$
d	4	$@d \bmod 4 = 0$

(a) Variables and Their Alignments



(b) A Layout That Wastes Space



(c) A Better Layout

FIGURE 5.18 Alignment Issues in Data-Area Offset Assignment.