CSE443
Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall

Phases of a compiler

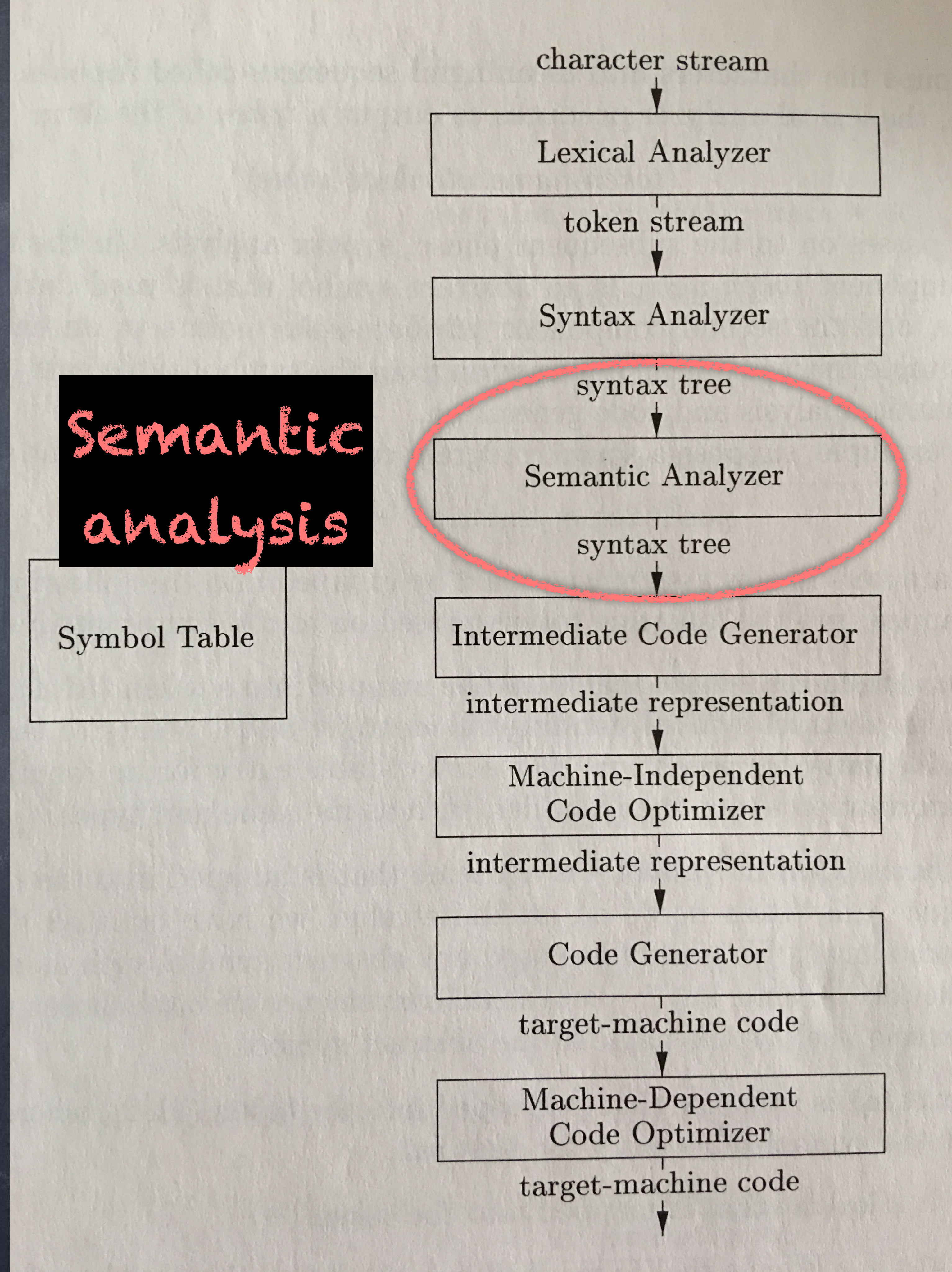


Figure 1.6, page 5 of text

Attribute grammars

- Attribute grammars provide a neater way of encoding such information.
- Each syntactic rule of the grammar can be decorated with:
  - a set of semantic rules/functions
  - a set of semantic predicates

Review

Attributes

- We can associate with each symbol X of the grammar a set of attributes A(X). Attributes are partitioned into:

  synthesized attributes S(X) – pass info up parse tree

  inherited attributes I(X) – pass info down parse tree

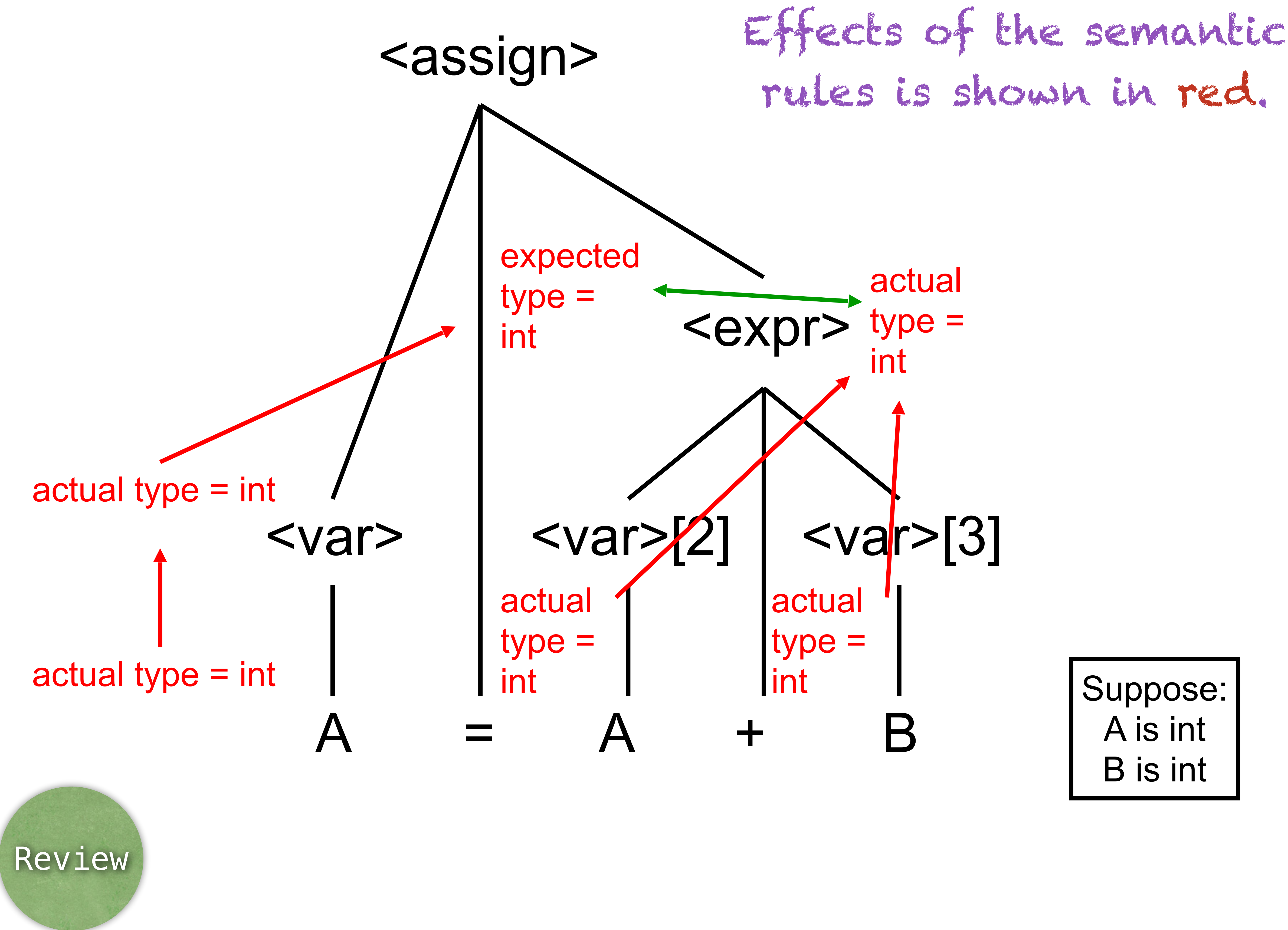
Review

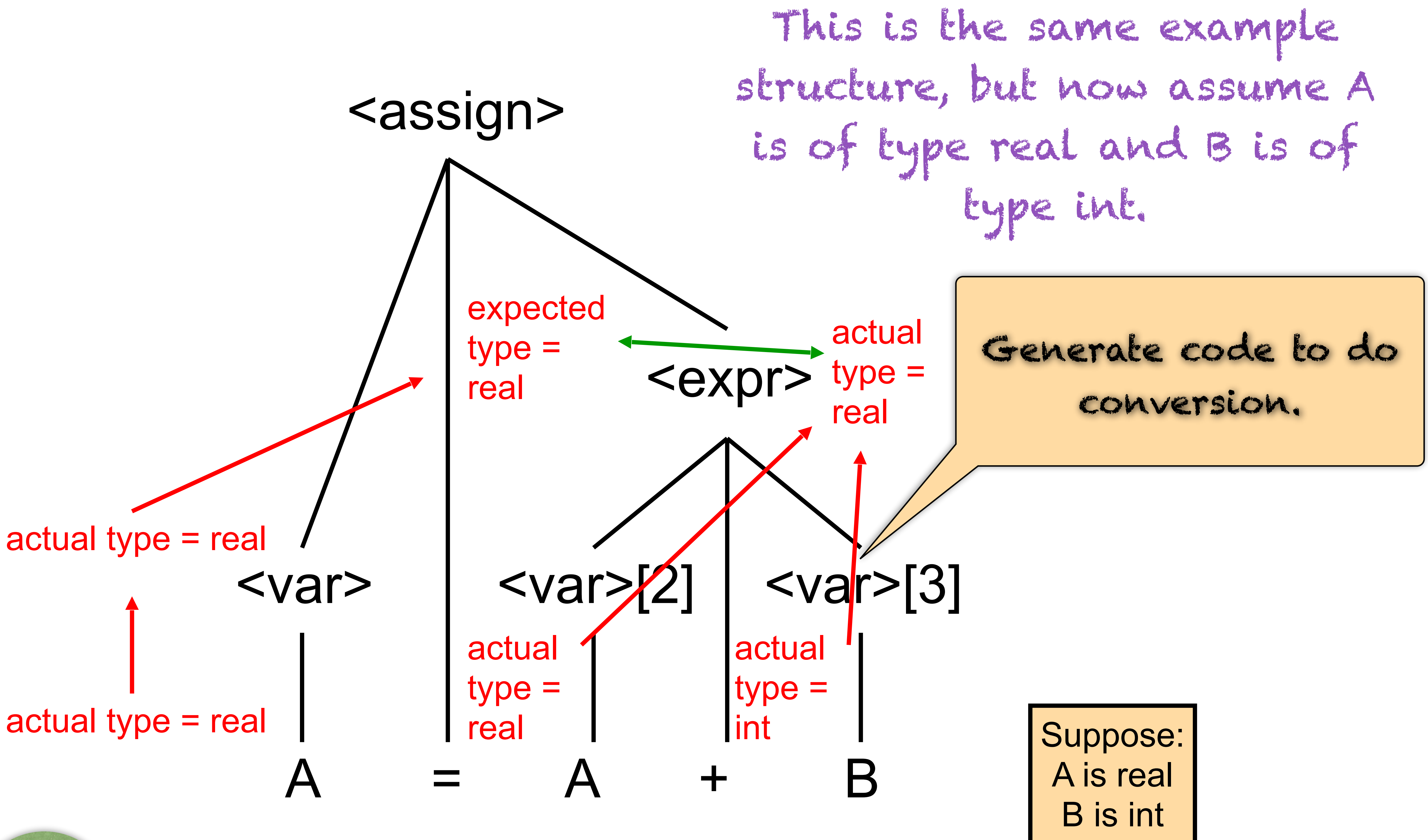
Example

```
<assign> → <var> = <expr>
<expr>.expType ← <var>.actType

<expr> → <var>[2] + <var>[3]
<expr>.actType ← if (var[2].actType = int) and
                      (var[3].actType = int)
                    then int
                    else real
<expr>.actType == <expr>.expType

<expr> → <var>
<expr>.actType ← <var>.actType
<expr>.actType == <expr>.expType

<var> → A | B | C
<var>.actType ← lookUp(<var>.string)
```

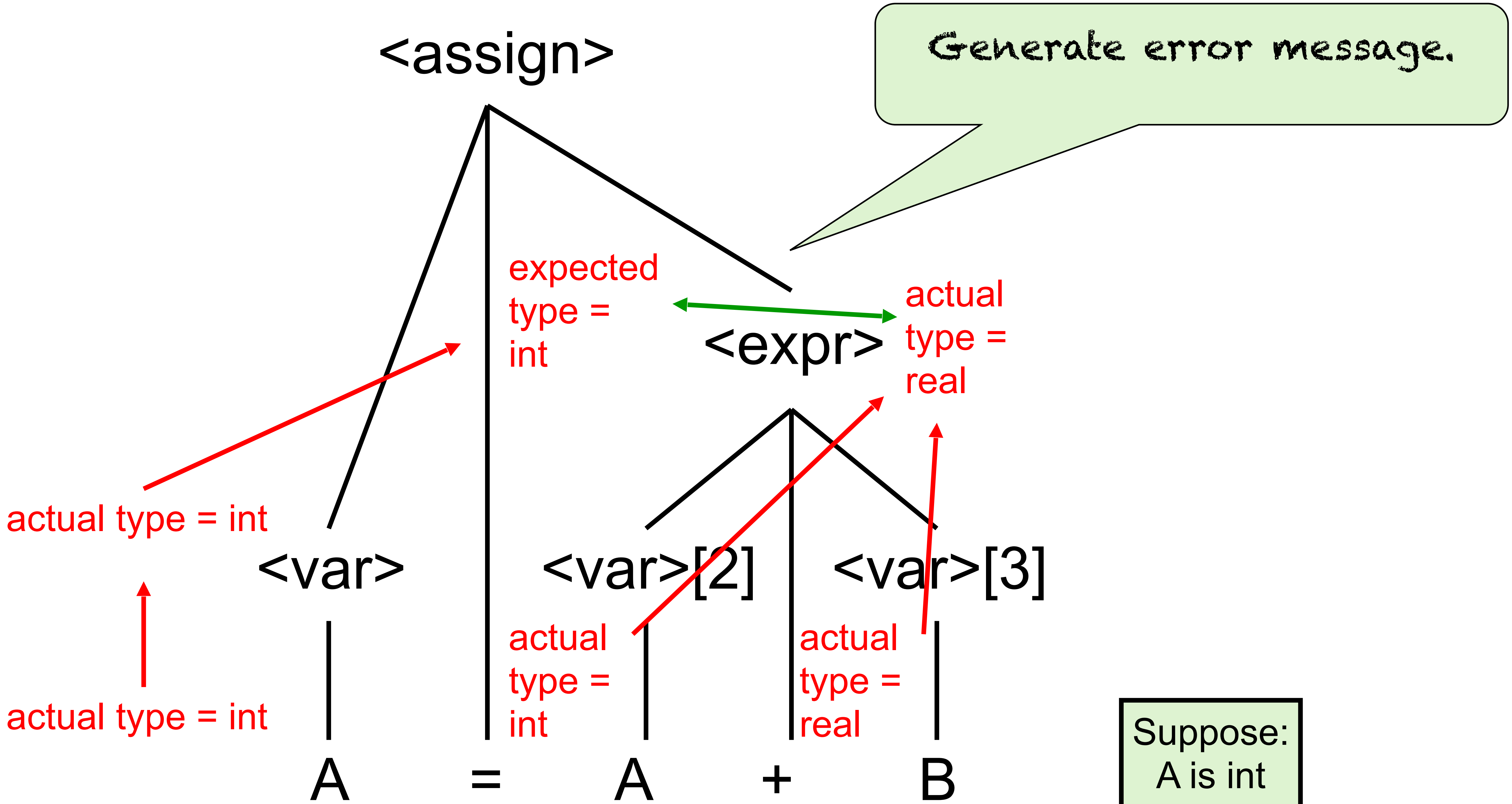
Syntactic rule
Semantic rule/function
Semantic predicate

Review


Effects of the semantic rules is shown in red.
<assign>
expected type = int
<expr>
actual type = int
actual type = int
<var>
<var>[2]
<var>[3]
actual type = int
actual type = int
actual type = int
A
=
A
+
B
Suppose:
A is int
B is int
Review


This is the same example structure, but now assume A is of type real and B is of type int.
<assign>
expected type = real
<expr>
actual type = real
Generate code to do conversion.
actual type = real
<var>
<var>[2]
<var>[3]
actual type = real
actual type = int
actual type = real
A = A + B
Suppose: A is real B is int

Review



Suppose:
A is int
B is real

Review

Syntax-Directed Definitions

"A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions" [p. 304]

Syntax-Directed Translation Schemes

"Syntax-directed translation schemes are a complementary notation to syntax-directed definitions. [...] A syntax-directed translation scheme (SDT) is a context-free grammar with program fragments embedded within production bodies." [p. 324]

Syntax-Directed Translation Schemes

"Any SDT can be implemented by first building a parse tree and then performing the actions in a [...] pre-order traversal." [p. 324]

"Typically, SDT's are implemented during parsing, without building a parse tree." [p. 324]

Syntax-Directed Translation Schemes

"...the simplest SDD implementation occurs when we can parse the grammar bottom-up and the SDD is S-attributed. In that case, we can construct an SDT in which each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production." [p. 324]

Syntax-Directed Translation Schemes

"If the attributes are all synthesized, and the actions occur at the ends of the productions, then we can compute the attributes for the head when we reduce the body to the head." [p. 325]

Syntax-Directed Translation Schemes

"We consider [now] the more general case of an L-attributed SDD." [p. 331]

"The rules for turning an L-attributed SDD into an SDT are as follows:

1. Embed the action that computes the inherited attributes for a nonterminal A immediately before the occurrence of A in the body of the production.

2. Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production." [p. 331]

Syntax-Directed Translation Schemes

"We consider [now] the more general case of an L-attributed SDD." [p. 331]

"The rules for turning an L-attributed SDD into an SDT are as follows:

1. Embed the action that computes the inherited attributes for a nonterminal A immediately before the occurrence of A in the body of the production.

X -> α { inherited attributes of A } A β

2. Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production." [p. 331]

A -> γ { synthesized attributes of A }

Implementing L-Attributed SDD's

"...we discuss the following methods for translating during parsing:

6. Implement an SDT in conjunction an LR parser.

   ... the SDT for an L-attributed SDD typically has actions in the middle of productions, and we cannot be sure during an LR parse that we are even in that production until its entire body has been constructed

   ... [however] if the underlying grammar is LL, we can always handle both the parsing and translation bottom-up." [p. 338]

Bottom-up parsing of L-Attributed SDD's

"...given an L-attributed SDD on an LL grammar, we can adapt the grammar to compute the same SDD on the new grammar during an LR parse" [p. 348]

1. "Start with the SDT [...] which places embedded actions before each nonterminal to compute its inherited attributes and an action at the end of the production to compute synthesized attributes.

2. Introduce into the grammar a marker nonterminal in place of each embedded action. Each such place gets a distinct marker, and there is one production for any marker M, M -> ε.

3. Modify the action a if marker nonterminal M replaces it in some production A -> α {a} β, and associate with M -> ε an action a' that

    (a) Copies, as inherited attributes of M, any attributes of A or symbols of α that action a needs.

    (b) Computes the attributes in the same way as a, but makes those attributes be synthesized attributes of M" [p. 349]

Bottom-up parsing of L-Attributed SDD's

"...we shall implement the actions on the LR parsing stack, so the necessary attributes will always be available a known number of positions down the stack." [p. 349]

Example 5.25 [p. 349]

A -> { B.i = f(A.i); } B C

becomes

A -> M B C

M -> {M.i = A.i; M.s = f(M.i); }

Phases of a compiler

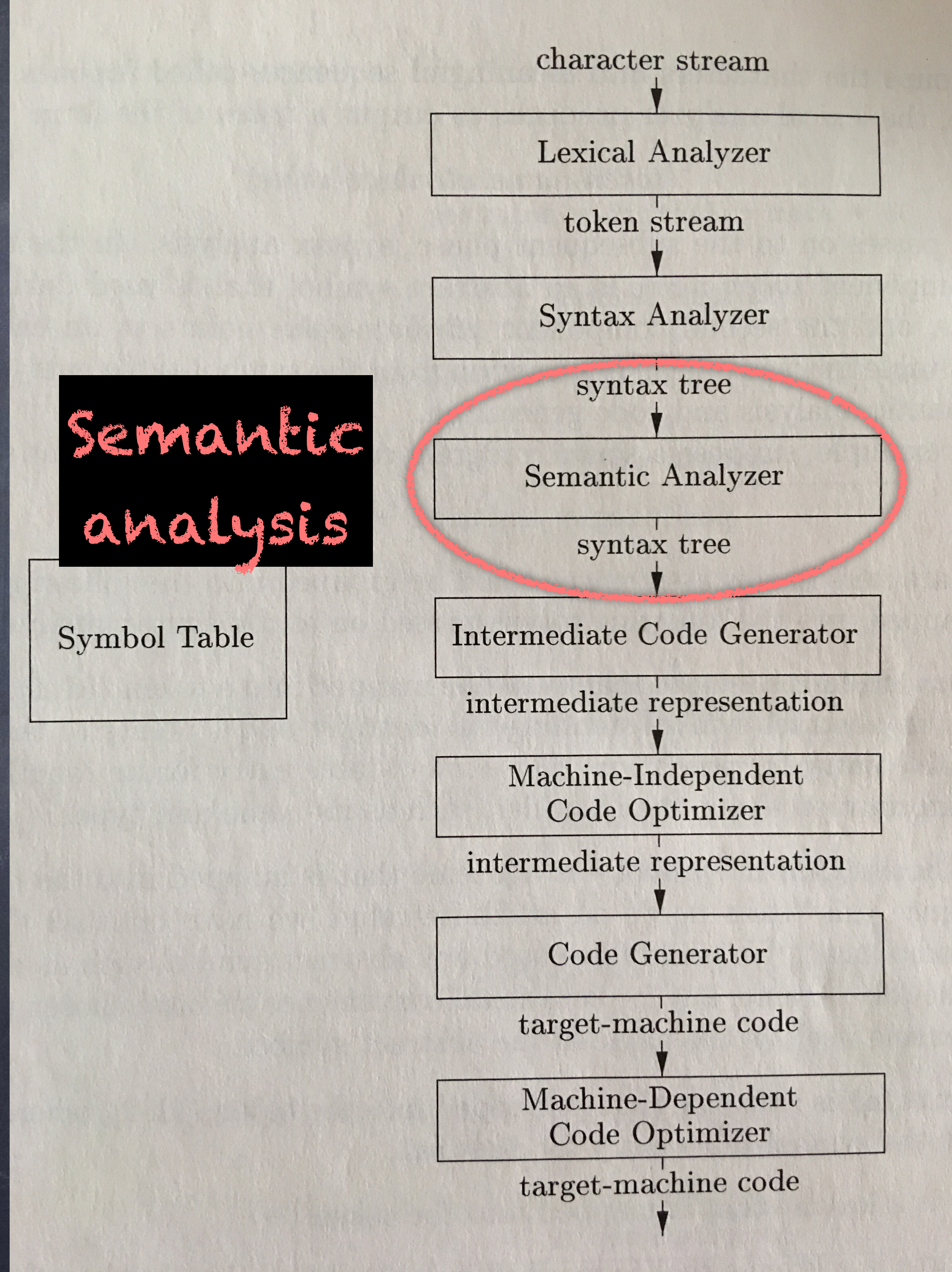


Figure 1.6, page 5 of text

Roadmap

We are going to look at examples 5.19 (p. 335) and 5.26 (p. 349) in some detail. The book revisits these examples in section 6.6.3.

Helpful background is covered in sections 5.3 and 5.4 (pages 318 through 337).

Example 5.19 (p. 335)

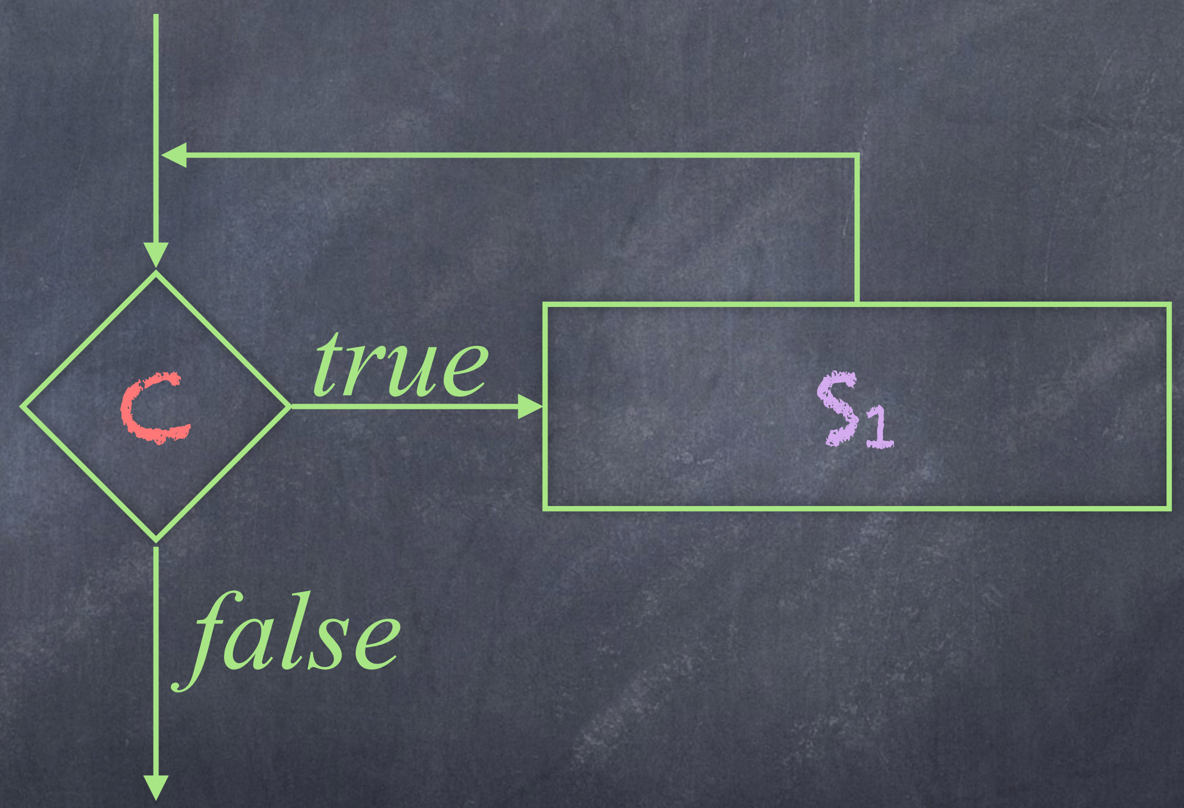
S -> while (C) S_1

What are the semantics of this?

Example 5.19 (p. 335)

S -> while (C) S_1

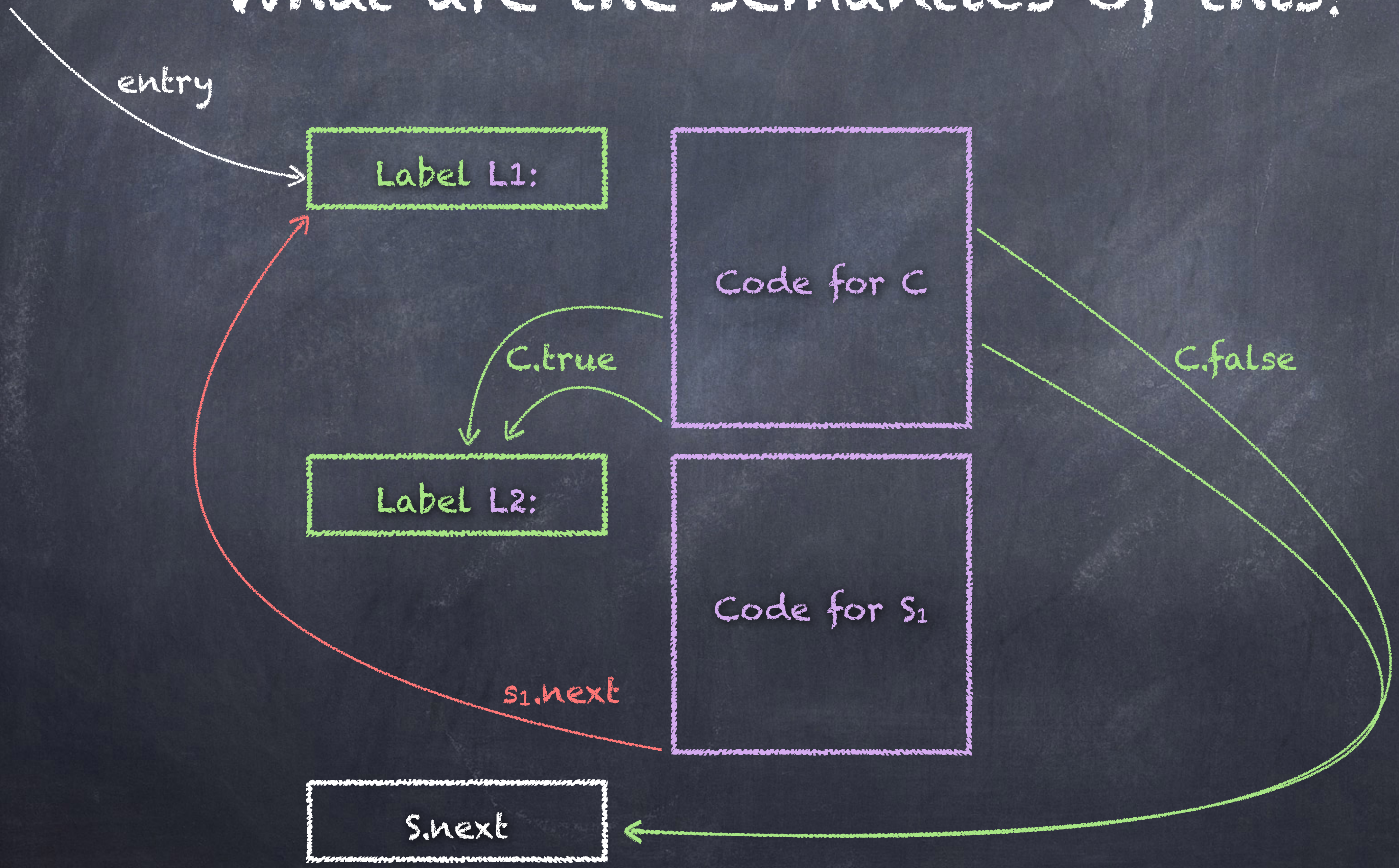
What are the semantics of this?



Example 5.19 (p. 335)

$S \rightarrow \text{while} \; (\; C \;) \; S_1$

What are the semantics of this?



Example 5.19 (p. 335)

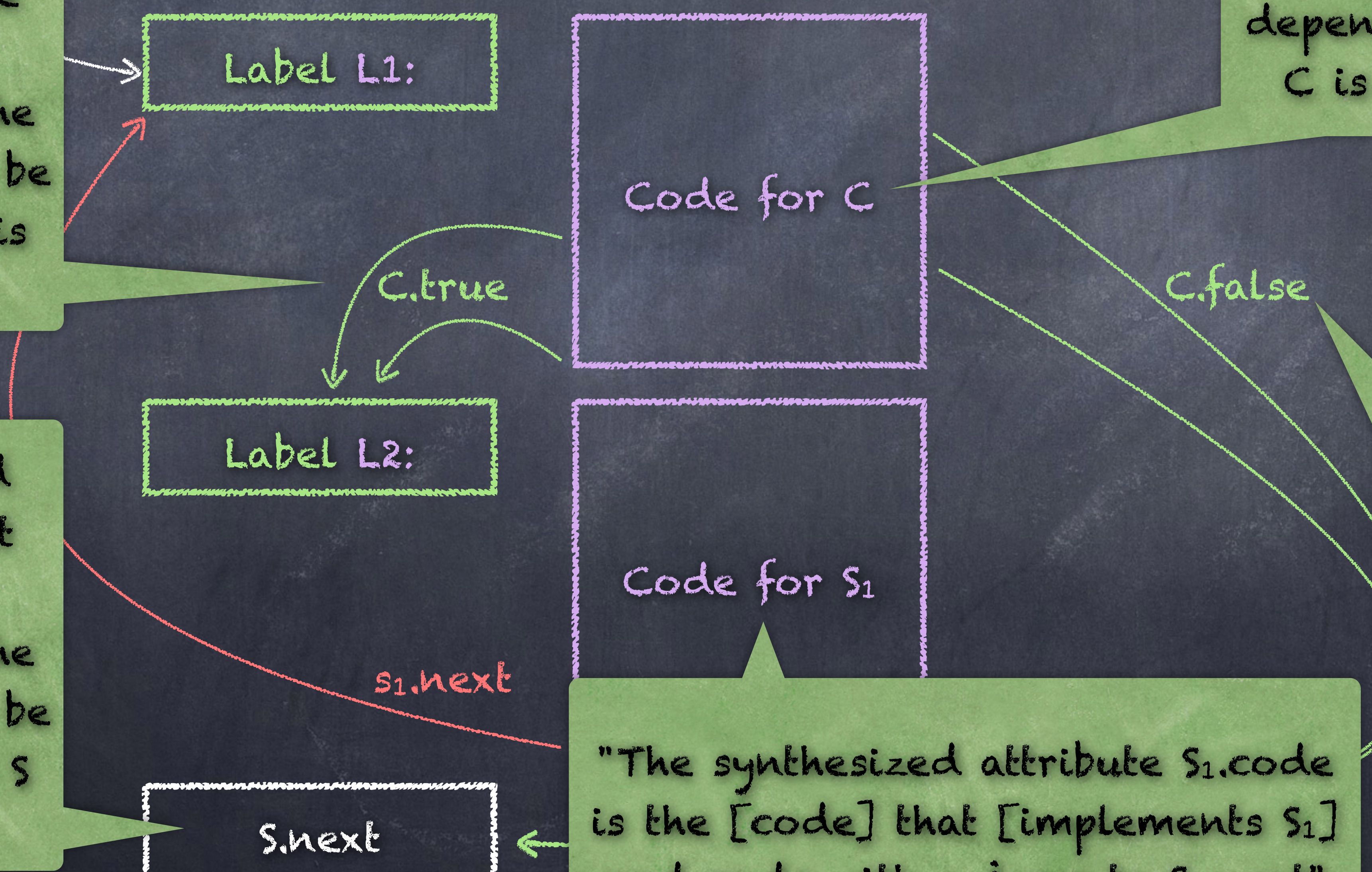
S -> while (C) S_1

What are the semantics of t

"The synthesized attribute S.code is the [code] that [implements S]"

"The synthesized attribute C.code is the [code] that [implements C] and jumps either to C.true or to C.false, depending on whether C is true or false."

"The inherited attribute C.true labels the beginning of the code that must be executed if C is true."



"The inherited attribute C.false labels the beginning of the code that must be executed if C is false."

"The inherited attribute S.next labels the beginning of the code that must be executed after S is finished."

"The synthesized attribute S_1.code is the [code] that [implements S_1] and ends with a jump to S_1.next"

Figure 5.28 (p. 336)

SDT for while statement

```
S -> while (  { L1 = new(); L2 = new();
                C.false = S.next; C.true = L2;
              }
    C )       { S1.next = L1;
              }
    S1        { S.code = label || L1 || C.code ||
                label || L2 || S1.code
              }
```

Example 5.26 [p. 349]

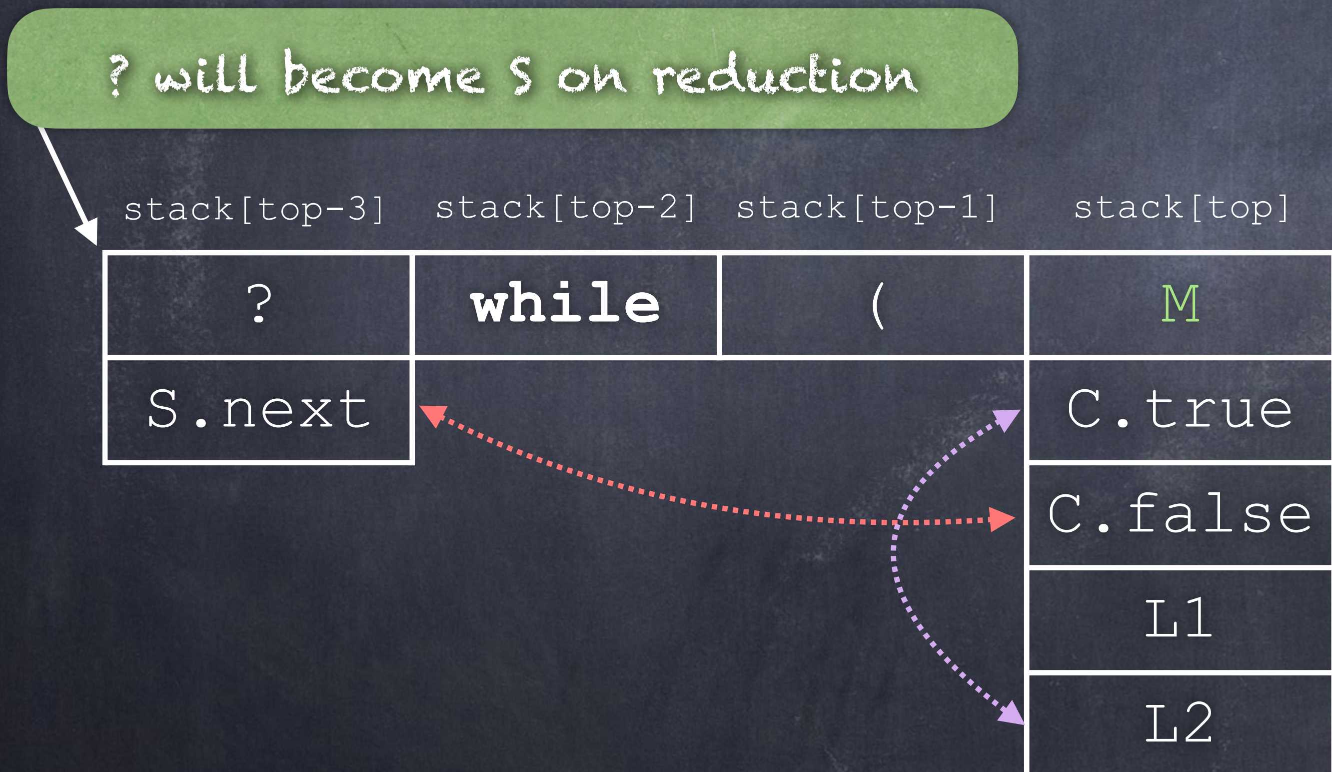
```
S -> while ( { L1=new(); L2=new(); C.false=S.next; C.true=L2; }
     C )     { S1.next=L1; }
     S1      { S.code=label || L1 || C.code || label || L2 || S1.code}
```

Example 5.26 [p. 349]

S -> while (
M C)
N S_1 { S.code=label || L1 || C.code || label || L2 || S_1.code}

M -> ε { L1=new(); L2=new(); C.false=S.next; C.true=L2; }

N -> ε { S_1.next=L1; }




```
L1 = new(); L2 = new();
C.true = L2;
C.false = stack[top-3].next;
```

Example 5.26 [p. 349]

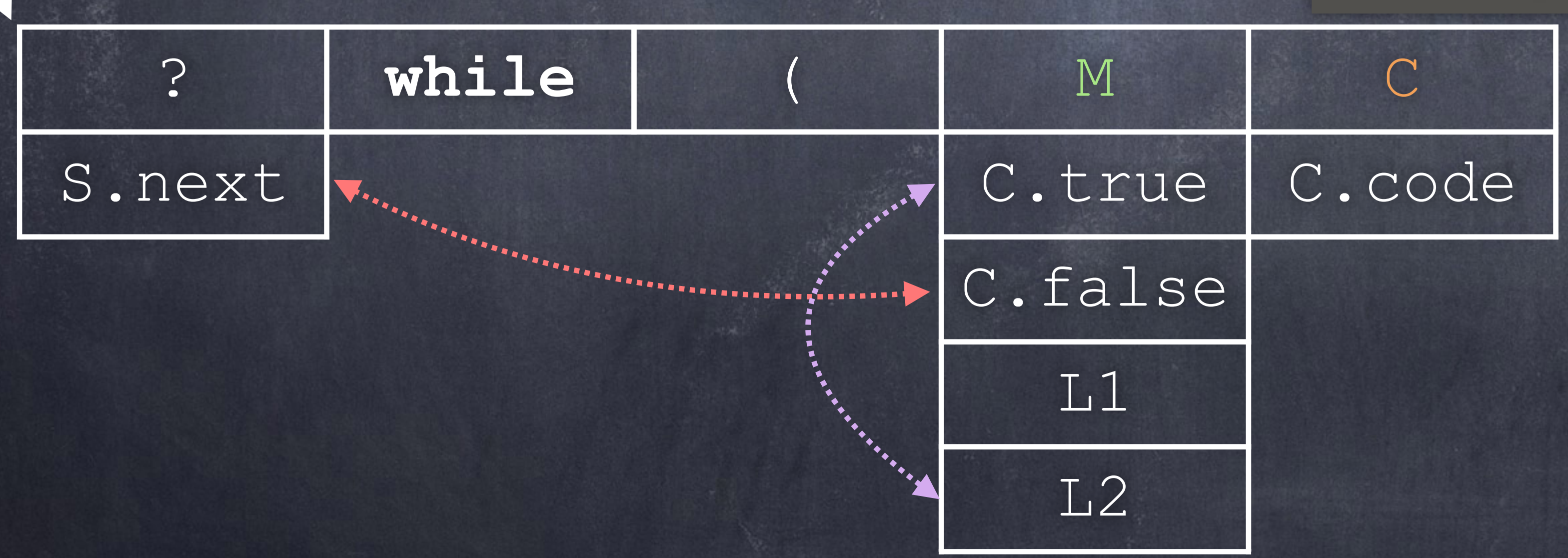
S -> while (
M C)
N S_1 { S.code=label || L1 || C.code || label || L2 || S_1.code}

M -> ε { L1=new(); L2=new(); C.false=S.next; C.true=L2; }

N -> ε { S_1.next=L1; }

C can appear in many productions; M ensures that attributes are in known positions on stack

? will become S on reduction

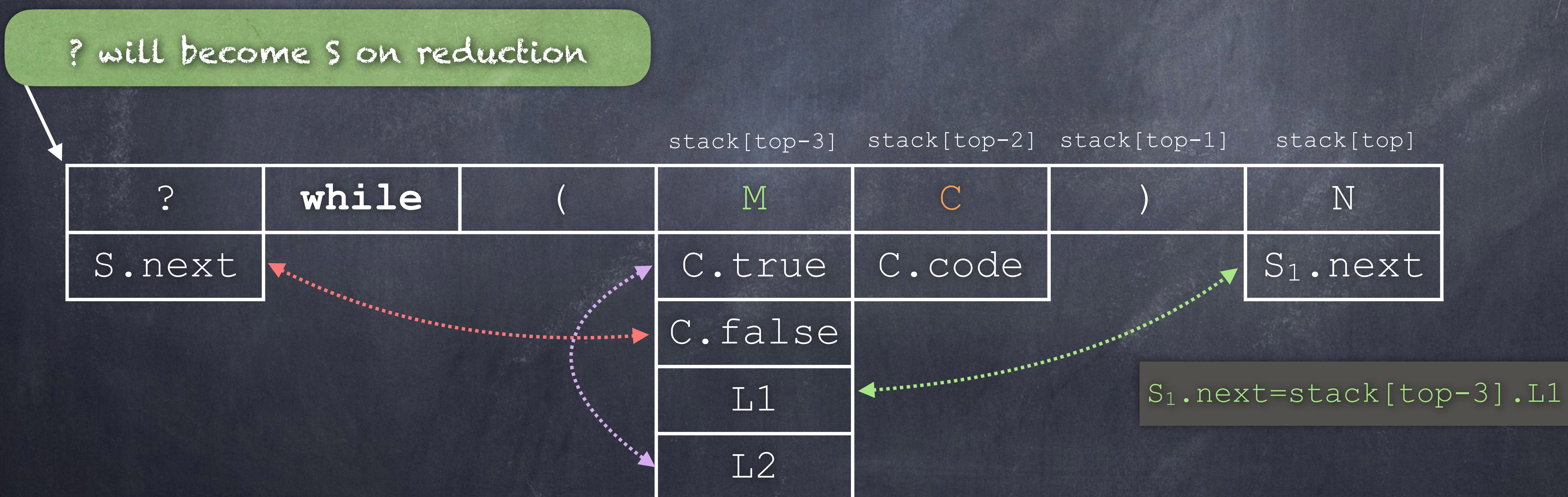


Example 5.26 [p. 349]

S -> while (
M C)
N S_1 { S.code=label || L1 || C.code || label || L2 || S_1.code}

M -> ε { L1=new(); L2=new(); C.false=S.next; C.true=L2; }

N -> ε { S_1.next=L1; }

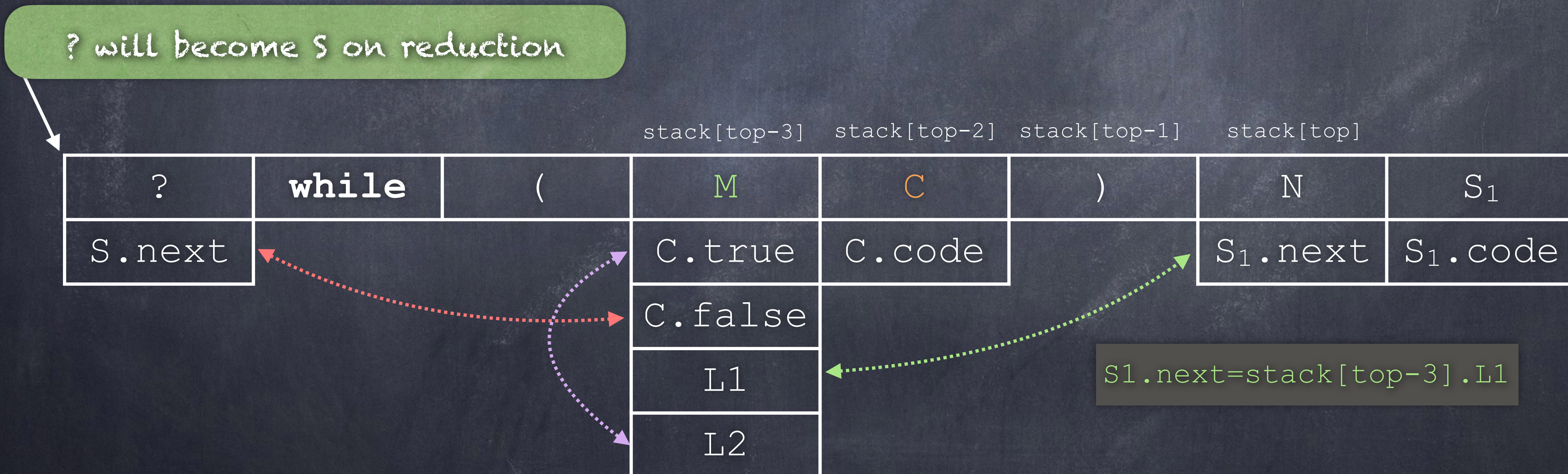


Example 5.26 [p. 349]

$S \rightarrow$ while (
    M C)
    N S_1 { S.code=label || L1 || C.code || label || L2 || S_1.code}

$M \rightarrow \varepsilon$ { L1=new(); L2=new(); C.false=S.next; C.true=L2; }

$N \rightarrow \varepsilon$ { S_1.next=L1; }



Roadmap

We will revisit how the semantics of flow-of-control statements can be expressed in section 6.6.3 Flow-of-Control Statements.

At that point we will learn the backpatching approach, which you will implement in your compiler.

§6.3 Types and Declarations

Type equivalence

Name equivalence: two types are equivalent if and only if they have the same name.

Structural equivalence: two types are equivalent if and only if they have the same structure. A type is structurally equivalent to itself (i.e. int is both name equivalent and structurally equivalent to int)

Name equivalence

```
int x = 3;
int y = 5;
int z = x * y;
```

The type of z is int.
The type of x * y is int.
The names of the types are the same, so the assignment is legal.

Structural equivalence

```
struct S { int v; double w; };
struct T { int v; double w; };

int main() {
  struct S x;
  x.v = 1; x.w = 4.5;
  struct T y;
  y = x;
  return 0;
}
```

types, names and order of fields all align

Under name equivalence the assignment is disallowed.

Under structural equivalence the assignment is permitted.

What does C do?

C does not allow the assignment

```
bash-3.2$  gcc type.c
type.c:9:5: error: assigning to
'struct T' from incompatible type
'struct S'
  y = x;
    ^ ~
1 error generated.
```

Structural equivalence

```
struct S { int v; double w; };
struct T { int a; double b; };

int main() {
  struct S x;
  x.v = 1; x.w = 4.5;
  struct T y;
  y = x;
  return 0;
}
```

types and order of fields align, but names differ

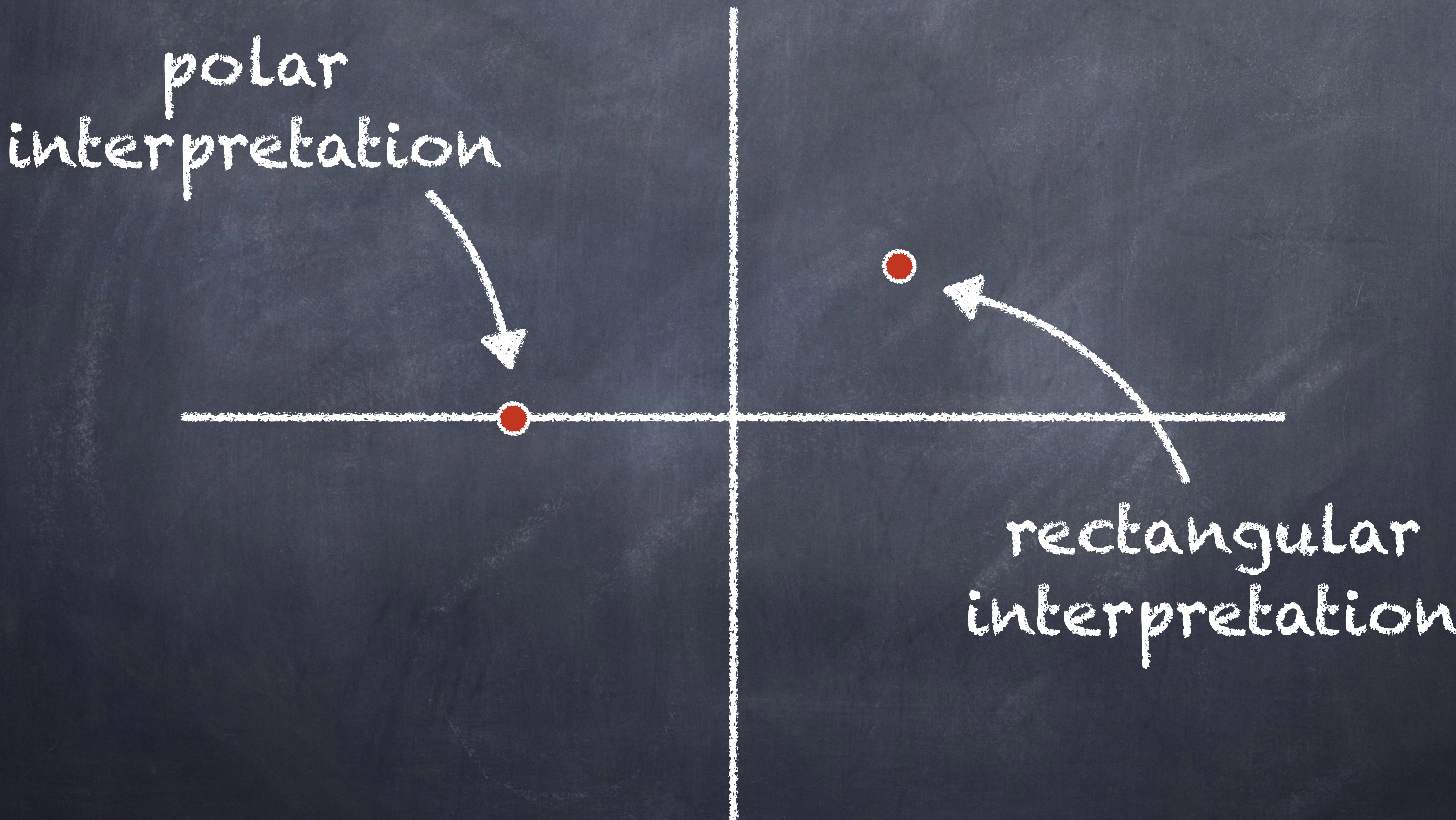
Should this be allowed?

Consider...

```
struct Rectangular { double x; double y; };
struct Polar { double r; double theta; };

int main() {
  struct Rectangular p;
  p.x = 3.14; p.y = 3.14;
  struct Polar q;
  q = p;
  return 0;
}
```

Should this be allowed?

Interpretation matters



Our language uses name equivalence

(use pointer to symbol table entry to identify type)

- built-in types:
  - primitive types: integer, Boolean, character
  - non-primitive type: string
- user-defined types:
  - record types have names
    - type recType : [real : x; real : y]
  - array types have names
    - type arrType : 2 -> string
  - function types have names
    - type funType : real -> recType

Recursive records

A record type must allow a component to be of the same type as the type itself:

```
type Node : [ integer : datum ; Node : rest ]
```

Recursive records

A record type must allow a component to be of the same type as the type itself:

type Node : [integer : datum ; Node : rest]

Be careful how you process declaration: you need to ensure that the second occurrence of Node does not trigger an undefined name