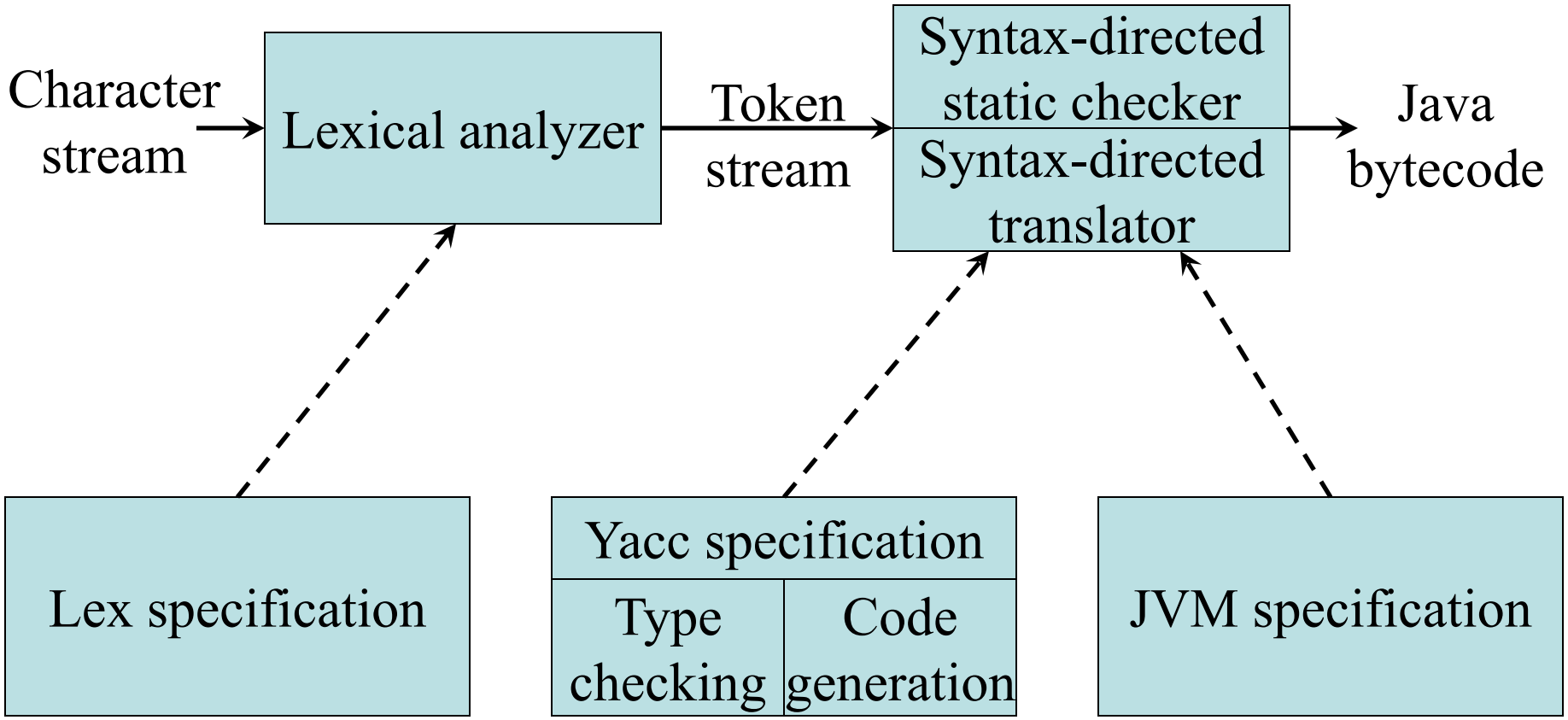
Static Checking and Type Systems

Chapter 6

The Structure of our Compiler Revisited



Static versus Dynamic Checking

- *Static checking*: the compiler enforces programming language's *static semantics*
  - Program properties that can be checked at compile time
- *Dynamic semantics*: checked at run time
  - Compiler generates verification code to enforce programming language's dynamic semantics

Static Checking

- Typical examples of static checking are
  - Type checks
  - Flow-of-control checks
  - Uniqueness checks
  - Name-related checks

Type Checks, Overloading, Coercion, and Polymorphism

```
int op(int), op(float);
int f(float);
int a, c[10], d;

d = c+d;            // FAIL

*d = a;             // FAIL

a = op(d);          // OK: overloading (C++)

a = f(d);           // OK: coersion of d to float

vector<int> v;      // OK: template instantiation
```

Flow-of-Control Checks

```
myfunc()
{ …
  break; // ERROR
}
```

```
myfunc()
{ …
  while (n)
  { …
    if (i>10)
      break; // OK
  }
}
```

```
myfunc()
{ …
  switch (a)
  { case 0:
      …
      break; // OK
    case 1:
      …
  }
}
```

Uniqueness Checks

```
myfunc()
{ int i, j, i; // ERROR
  …
}
```

```
cnufym(int a, int a) // ERROR
{   …
}
```

```
struct myrec
{ int name;
};
struct myrec // ERROR
{ int id;
};
```

Name-Related Checks

```
LoopA: for (int I = 0; I < n; I++)
       { …
         if (a[I] == 0)
           break LoopB; // Java labeled loop
         …
       }
```

One-Pass versus Multi-Pass Static Checking

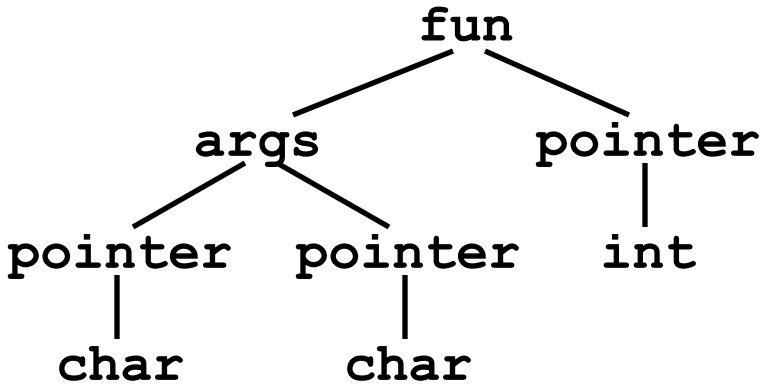
- *One-pass compiler*: static checking for C, Pascal, Fortran, and many other languages is performed in one pass while intermediate code is generated
  - Influences design of a language: placement constraints
- *Multi-pass compiler*: static checking for Ada, Java, and C# is performed in a separate phase, sometimes by traversing the syntax tree multiple times

Type Expressions

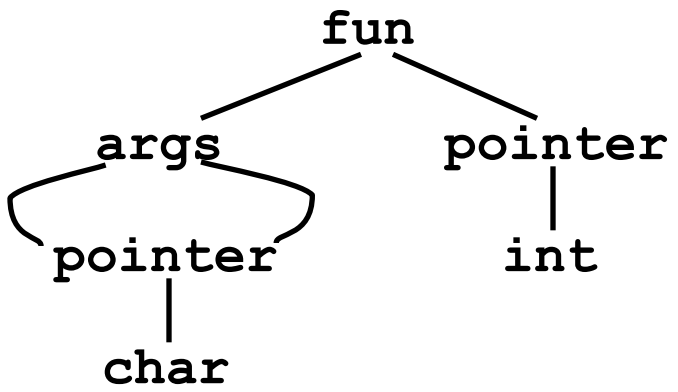
- *Type expressions* are used in declarations and type casts to define or refer to a type
  - *Primitive types*, such as **`int`** and **`bool`**
  - *Type constructors*, such as pointer-to, array-of, records and classes, templates, and functions
  - *Type names*, such as typedefs in C and named types in Pascal, refer to type expressions

Graph Representations for Type Expressions

```
int *f(char*,char*)
```



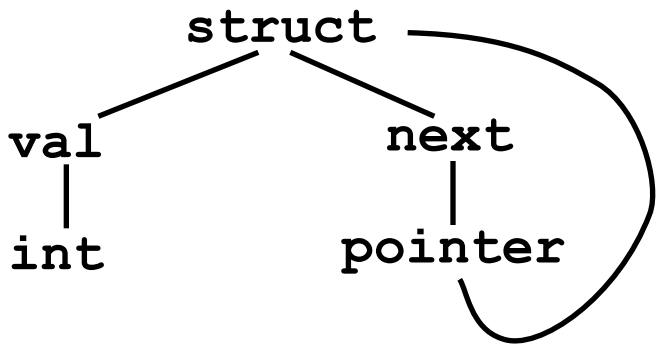
Tree forms



DAGs

Cyclic Graph Representations

```
struct Node
{ int val;
  struct Node *next;
};
```



Cyclic graph

Name Equivalence

- Each *type name* is a distinct type, even when the type expressions the names refer to are the same
- Types are identical only if names match
- Used by Pascal (inconsistently)

```
type link = ^node;
var next : link;
    last : link;
       p : ^node;
    q, r : ^node;
```

With name equivalence in Pascal:

p ≠ next

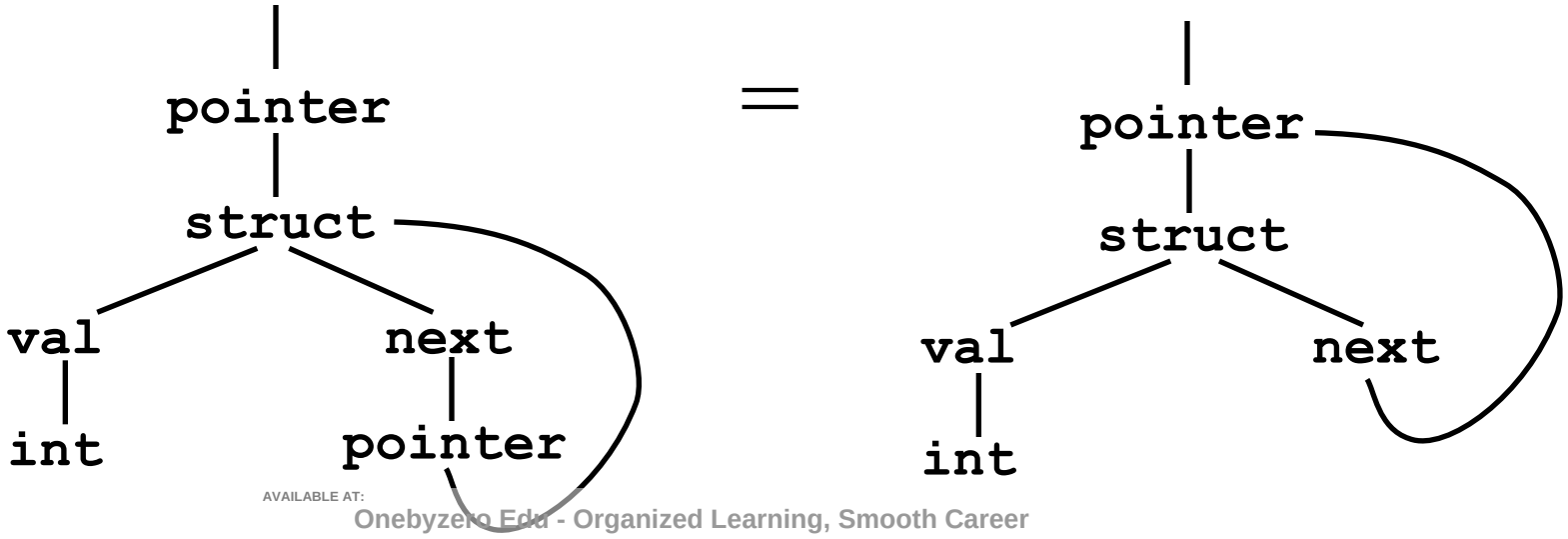
p ≠ last

p = q = r

next = last

Structural Equivalence of Type Expressions

- Two types are the same if they are *structurally identical*
- Used in C, Java, C#

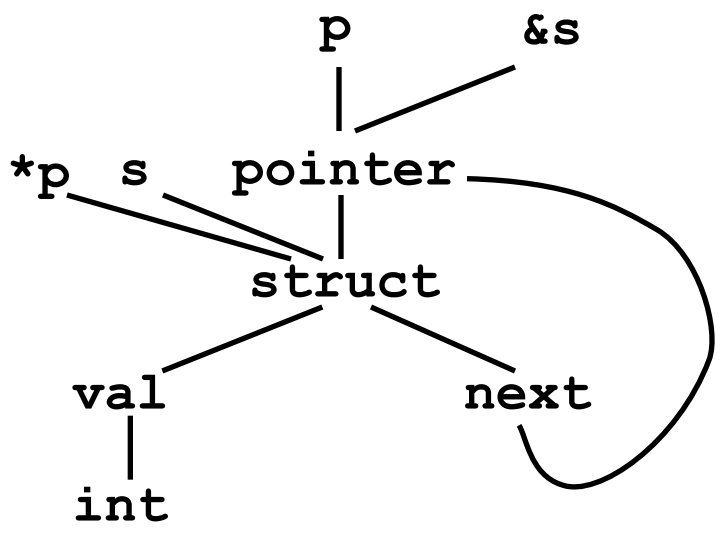


Structural Equivalence of Type Expressions (cont'd)

- Two structurally equivalent type expressions have the same pointer address when constructing graphs by sharing nodes

```
struct Node
{ int val;
  struct Node *next;
};
struct Node s, *p;

… p = &s; // OK
… *p = s; // OK
```



Constructing Type Graphs in Yacc

`Type *mkint()`	construct int node if not already constructed
`Type *mkarr(Type*,int)`	construct array-of-type node if not already constructed
`Type *mkptr(Type*)`	construct pointer-of-type node if not already constructed

Syntax-Directed Definitions for Constructing Type Graphs in Yacc

```
%union
{ Symbol *sym;
  int num;
  Type *typ;
}
%token INT
%token <sym> ID
%token <int> NUM
%type <typ> type
%%
decl : type ID                  { addtype($2, $1); }
     | type ID '[' NUM ']'      { addtype($2, mkarr($1, $4)); }
     ;
type : INT                      { $$ = mkint(); }
     | type '*'                 { $$ = mkptr($1); }
     | /* empty */              { $$ = mkint(); }
     ;
```

Type Systems

- A *type system* defines a set of types and rules to assign types to programming language constructs
- Informal type system rules, for example "*if both operands of addition are of type integer, then the result is of type integer*"
- Formal type system rules: Post system

Type Rules in Post System Notation

Type judgments

$e : \tau$

where e is an expression and τ is a type, are provable or not

$$\frac{\rho(v) = \tau}{\rho \vdash v : \tau}$$

Environment ρ maps objects v to types τ:

$\rho(v) = \tau$

$$\frac{\rho(v) = \tau \qquad \rho \vdash e : \tau}{\rho \vdash v := e : void}$$

$$\frac{\rho \vdash e_1 : integer \qquad \rho \vdash e_2 : integer}{\rho \vdash e_1 + e_2 : integer}$$

Type System Example

Environment ρ binds objects to types, for example

$\rho = \{ \langle \mathbf{x}, \mathit{integer} \rangle, \langle \mathbf{y}, \mathit{integer} \rangle, \langle \mathbf{z}, \mathit{char} \rangle, \langle 1, \mathit{integer} \rangle, \langle 2, \mathit{integer} \rangle \}$

Type checking = theorem proving

The proof that **x := y + 2** is typed correctly:

$$\dfrac{\dfrac{\rho(\mathbf{x}) = \mathit{integer} \qquad \dfrac{\dfrac{\rho(\mathbf{y}) = \mathit{integer}}{\rho \vdash \mathbf{y} : \mathit{integer}} \quad \dfrac{\rho(\mathbf{2}) = \mathit{integer}}{\rho \vdash \mathbf{2} : \mathit{integer}}}{\rho \vdash \mathbf{y} + \mathbf{2} : \mathit{integer}}}{\rho \vdash \mathbf{x} := \mathbf{y} + \mathbf{2} : \mathit{void}}$$

A Simple Language Example

$P \rightarrow D$ **;** S

$D \rightarrow D$ **;** D
| **id :** T

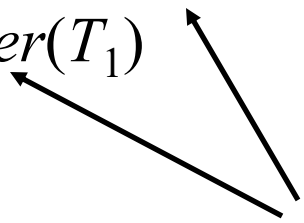
$T \rightarrow$ **boolean**
| **char**
| **integer**
| **array [num] of** T
| **^** T

$S \rightarrow$ **id :=** E
| **if** E **then** S
| **while** E **do** S
| S **;** S

$E \rightarrow$ **true**
| **false**
| **literal**
| **num**
| **id**
| E **and** E
| E **+** E
| E **[** E **]**
| E **^**

Simple Language Example: Declarations

$D \rightarrow$ **id :** T	{ *addtype*(**id**.entry, T.type) }
$T \rightarrow$ **boolean**	{ T.type := *boolean* }
$T \rightarrow$ **char**	{ T.type := *char* }
$T \rightarrow$ **integer**	{ T.type := *integer* }
$T \rightarrow$ **array [num] of** T_1	{ T.type := *array*(1..**num**.val, T_1.type) }
$T \rightarrow$ **^** T_1	{ T.type := *pointer*(T_1)



Parametric types: type constructor

Simple Language Example: Checking Statements

$$\frac{\rho(v) = \tau \qquad \rho \vdash e : \tau}{\rho \vdash v := e : void}$$

$S \rightarrow$ **id :=** E { S.type := **if id**.type = E.type **then** *void* **else** *type_error* }

Note: the type of **id** is determined by scope's environment:
id.type = *lookup*(**id**.entry)

Simple Language Example: Checking Statements (cont'd)

$$\frac{\rho \vdash e : \textit{boolean} \qquad \rho \vdash s : \tau}{\rho \vdash \textbf{if } e \textbf{ then } s : \tau}$$

$S \rightarrow \textbf{if } E \textbf{ then } S_1$ { S.type := **if** E.type = *boolean* **then** S_1.type **else** *type_error* }

Simple Language Example: Statements (cont'd)

$$\frac{\rho \vdash e : \textit{boolean} \qquad \rho \vdash s : \tau}{\rho \vdash \textbf{while } e \textbf{ do } s : \tau}$$

$S \rightarrow \textbf{while } E \textbf{ do } S_1$ { S.type := **if** E.type = *boolean* **then** S_1.type **else** *type_error* }

Simple Language Example: Checking Statements (cont'd)

$$\frac{\rho \vdash s_1 : \mathit{void} \qquad \rho \vdash s_2 : \mathit{void}}{\rho \vdash s_1 \,;\, s_2 : \mathit{void}}$$

$S \rightarrow S_1 \,;\, S_2$ { S.type := **if** S_1.type = *void* **and** S_2.type = *void* **then** *void* **else** *type_error* }

Simple Language Example: Checking Expressions

$$\frac{\rho(v) = \tau}{\rho \vdash v : \tau}$$

$E \rightarrow$ **true**	{ E.type = *boolean* }
$E \rightarrow$ **false**	{ E.type = *boolean* }
$E \rightarrow$ **literal**	{ E.type = *char* }
$E \rightarrow$ **num**	{ E.type = *integer* }
$E \rightarrow$ **id**	{ E.type = *lookup*(**id**.entry) }

…

Simple Language Example: Checking Expressions (cont'd)

$$\frac{\rho \vdash e_1 : \textit{integer} \qquad \rho \vdash e_2 : \textit{integer}}{\rho \vdash e_1 + e_2 : \textit{integer}}$$

$E \rightarrow E_1 + E_2$ { E.type := **if** E_1.type = *integer* **and** E_2.type = *integer* **then** *integer* **else** *type_error* }

Simple Language Example: Checking Expressions (cont'd)

$$\frac{\rho \vdash e_1 : \textit{boolean} \qquad \rho \vdash e_2 : \textit{boolean}}{\rho \vdash e_1 \textbf{ and } e_2 : \textit{boolean}}$$

$E \rightarrow E_1$ **and** E_2 { E.type := **if** E_1.type = *boolean* **and** E_2.type = *boolean* **then** *boolean* **else** *type_error* }

Simple Language Example: Checking Expressions (cont'd)

$$\frac{\rho \vdash e_1 : \mathit{array}(s, \tau) \quad \rho \vdash e_2 : \mathit{integer}}{\rho \vdash e_1[e_2] : \tau}$$

$E \rightarrow E_1$ **[** E_2 **]** { E.type := **if** E_1.type = $\mathit{array}(s, t)$ **and** E_2.type = *integer* **then** t **else** *type_error* }

Simple Language Example: Checking Expressions (cont'd)

$$\frac{\rho \vdash e : \mathit{pointer}(\tau)}{\rho \vdash e\,\hat{} : \tau}$$

$E \rightarrow E_1\ \hat{}$ { E.type := **if** E_1.type = *pointer*(t) **then** t **else** *type_error* }

A Simple Language Example: Functions

$T \rightarrow T \text{ -> } T$

$E \rightarrow E\,(\,E\,)$

For example:

```
v : integer;
odd : integer -> boolean;
if odd(3) then
    v := 1;
```

Simple Language Example: Function Declarations

$T \rightarrow T_1$ -> T_2 { T.type := *function*(T_1.type, T_2.type) }

Parametric type:
type constructor

Simple Language Example: Checking Function Invocations

$$\frac{\rho \vdash e_1 : \mathit{function}(\sigma, \tau) \qquad \rho \vdash e_2 : \sigma}{\rho \vdash e_1(e_2) : \tau}$$

$E \rightarrow E_1 \; (\; E_2 \;)$ { E.type := **if** E_1.type = $\mathit{function}(s, t)$ **and** E_2.type = s **then** t **else** *type_error* }

Type Conversion and Coercion

- *Type conversion* is explicit, for example using type casts
- *Type coercion* is implicitly performed by the compiler
- Both require a type system to check and infer types for (sub)expressions

Syntax-Directed Definitions for Type Checking in Yacc

```
%{
enum Types {Tint, Tfloat, Tpointer, Tarray, … };
typedef struct Type
{ enum Types type;
  struct Type *child; // at most one type parameter
} Type;
%}

%union
{ Type *typ;
}

%type <typ> expr

%%
…
```

Syntax-Directed Definitions for Type Checking in Yacc (cont'd)

```
...
%%

expr : expr '+' expr { if ($1->type != Tint
                         || $3->type != Tint)
                          semerror("non-int operands in +");
                       $$ = mkint();
                       emit(iadd);
                     }
```

Syntax-Directed Definitions for Type Coercion in Yacc

```
...
%%
expr : expr '+' expr
     { if ($1->type == Tint && $3->type == Tint)
       { $$ = mkint(); emit(iadd);
       }
       else if ($1->type == Tfloat && $3->type == Tfloat)
       { $$ = mkfloat(); emit(fadd);
       }
       else if ($1->type == Tfloat && $3->type == Tint)
       { $$ = mkfloat(); emit(i2f); emit(fadd);
       }
       else if ($1->type == Tint && $3->type == Tfloat)
       { $$ = mkfloat(); emit(swap); emit(i2f); emit(fadd);
       }
       else semerror("type error in +");
         $$ = mkint();
     }
```

Syntax-Directed Definitions for L-Values and R-Values in Yacc

```
%{
typedef struct Node
{ Type *typ;  // type structure
  int islval; // 1 if L-value
} Node;
%}

%union
{ Node *rec;
}

%type <rec> expr

%%
...
```

Syntax-Directed Definitions for L-Values and R-Values in Yacc

```
expr : expr '+' expr
          { if ($1->typ->type != Tint || $3->typ->type != Tint)
              semerror("non-int operands in +");
            $$->typ = mkint();
            $$->islval = FALSE;
            emit(…);
          }
     | expr '=' expr
          { if (!$1->islval || $1->typ != $3->typ)
              semerror("invalid assignment");
            $$->typ = $1->typ;
            $$->islval = FALSE;
            emit(…);
          }
     | ID
          { $$->typ = lookup($1);
            $$->islval = TRUE;
            emit(…);
          }
```