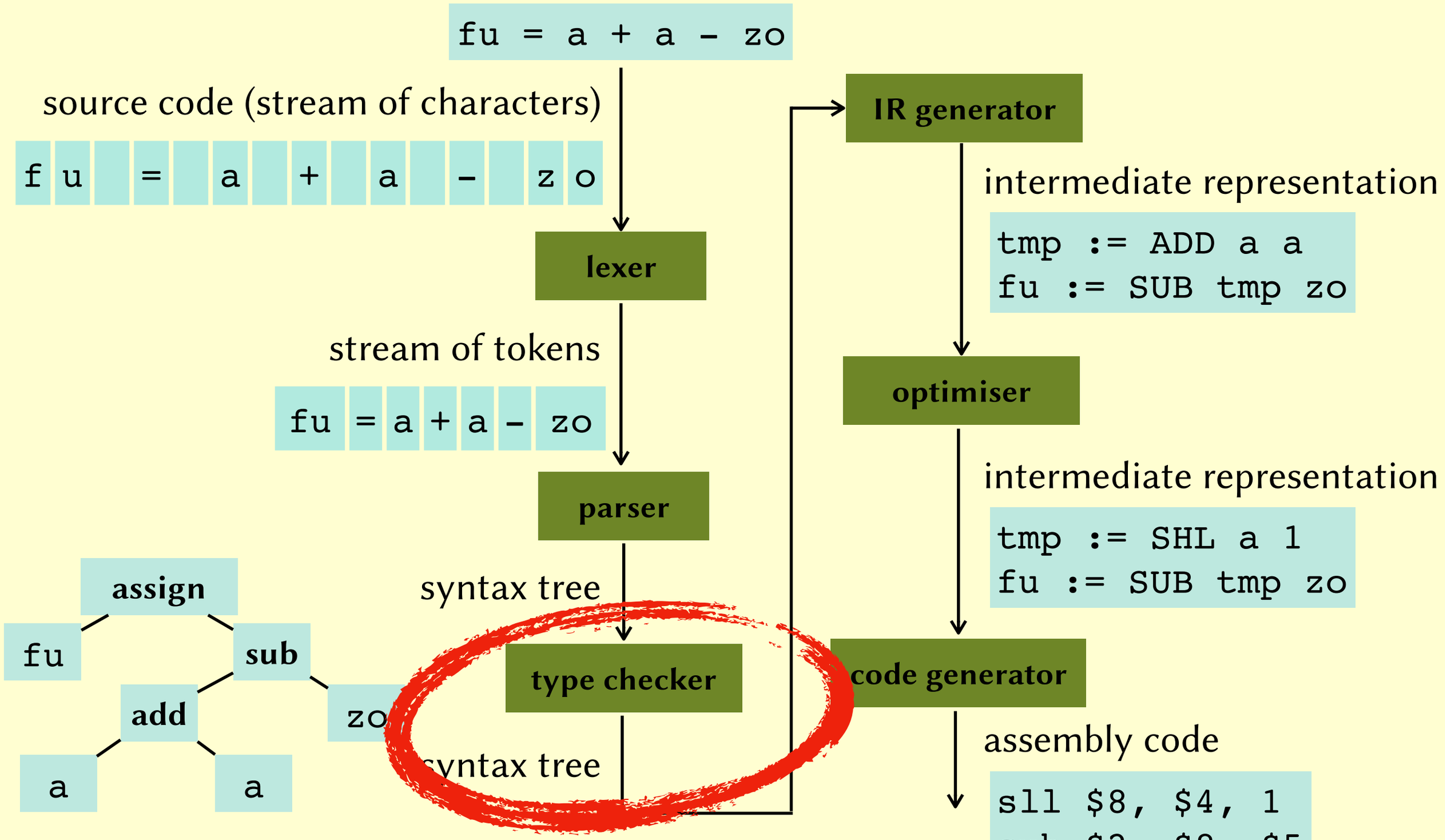
Lecture 7: Types

John Wickerson

Compilers

Anatomy of a compiler



Type checking

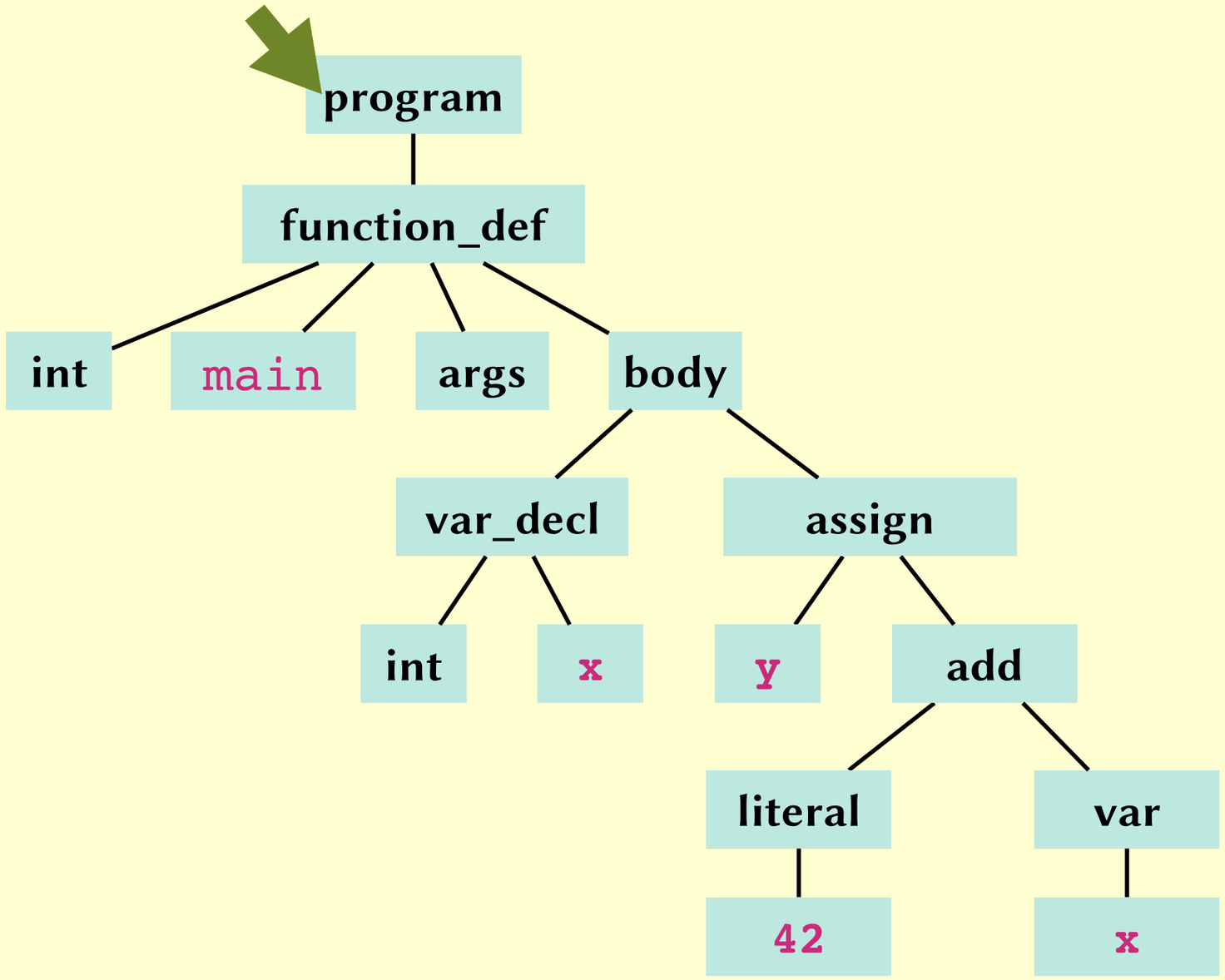
- Some programs are *syntactically valid* but *semantically invalid.*
- Consider this (partial) grammar for C programs:

```
Prog   ::=  Type X ( Args ) { Stmts } | ...
Stmts  ::=  ε | Stmt Stmts
Stmt   ::=  Type X ; | X = Expr ; | ...
```

- The program `int main () { int x; y = 42+x; }` would be accepted by this grammar, despite not being meaningful.

Type checking in C

```
int main () { int x; y = 42+x; }
```

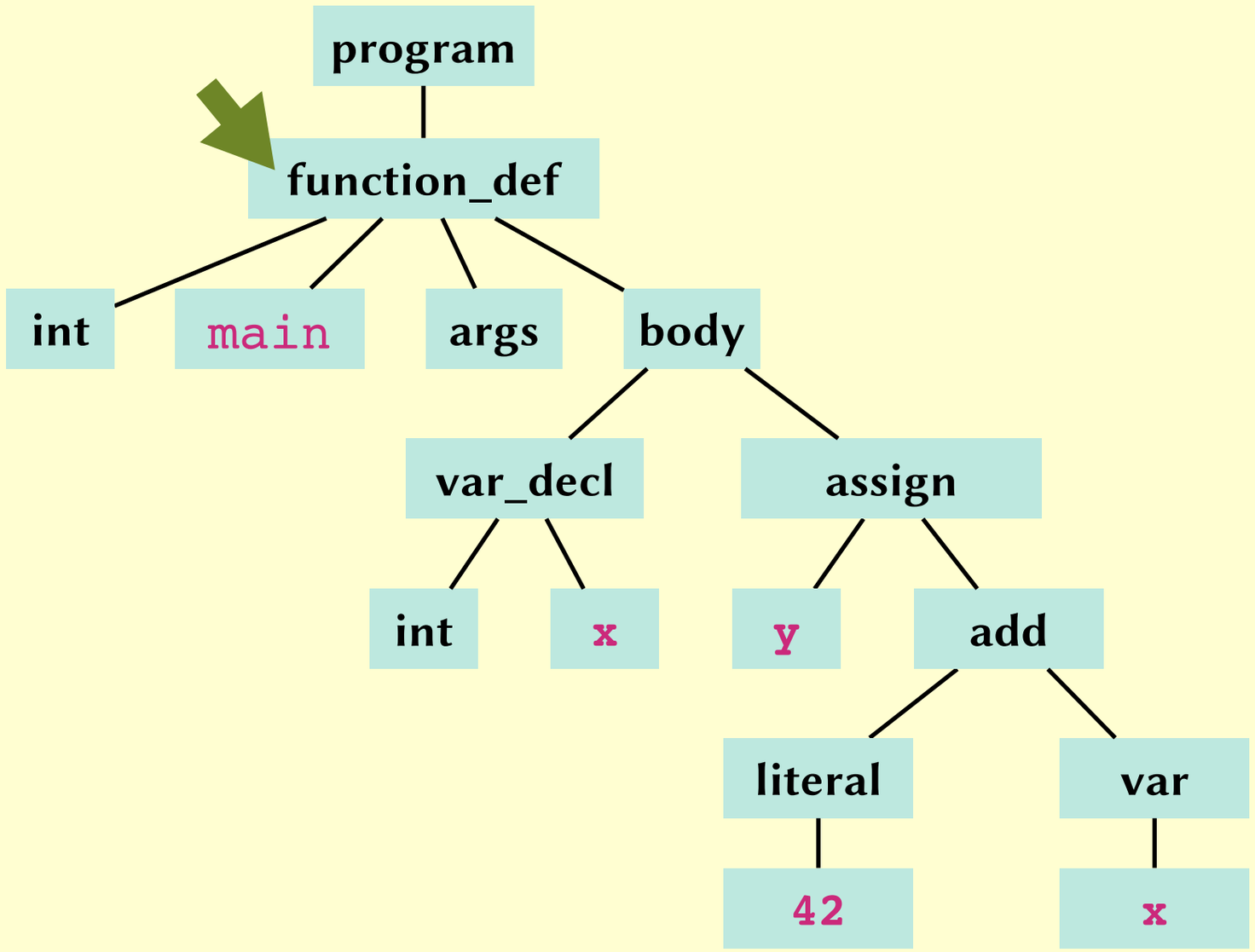


Symbol Table

Name	Type

Type checking in C

```
int main () { int x; y = 42+x; }
```

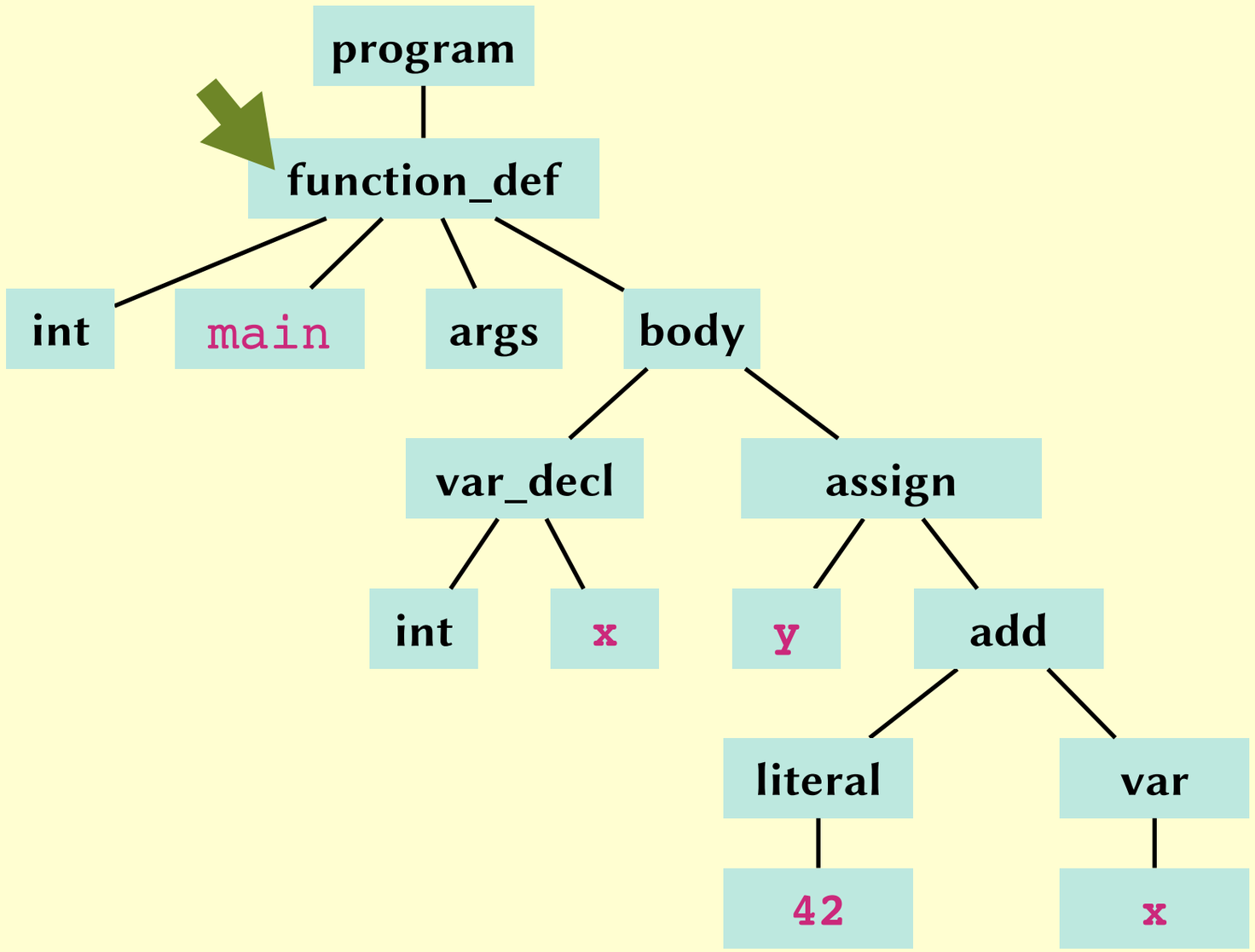


Symbol Table

Name	Type

Type checking in C

```
int main () { int x; y = 42+x; }
```

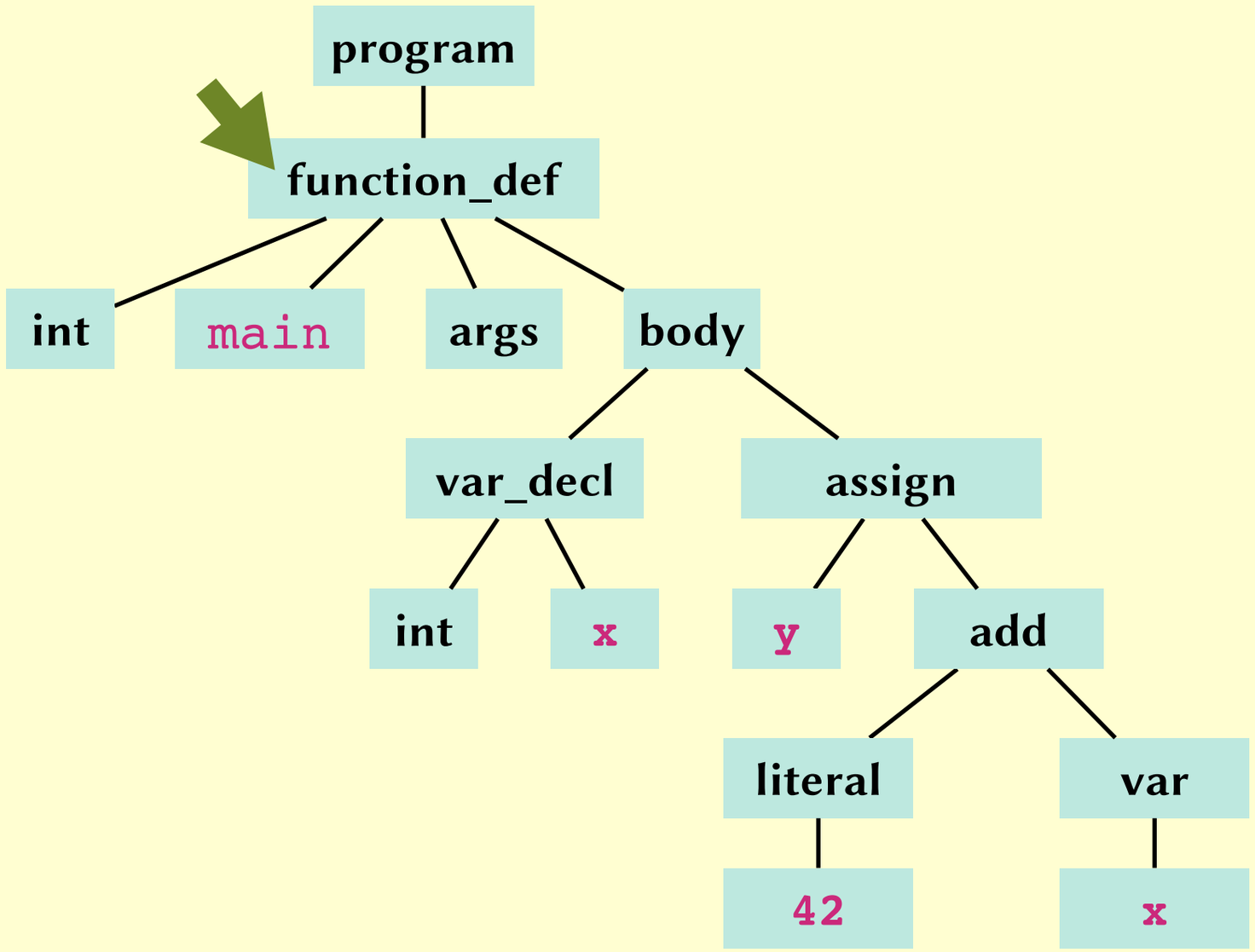


Symbol Table

Name	Type

Type checking in C

```
int main () { int x; y = 42+x; }
```

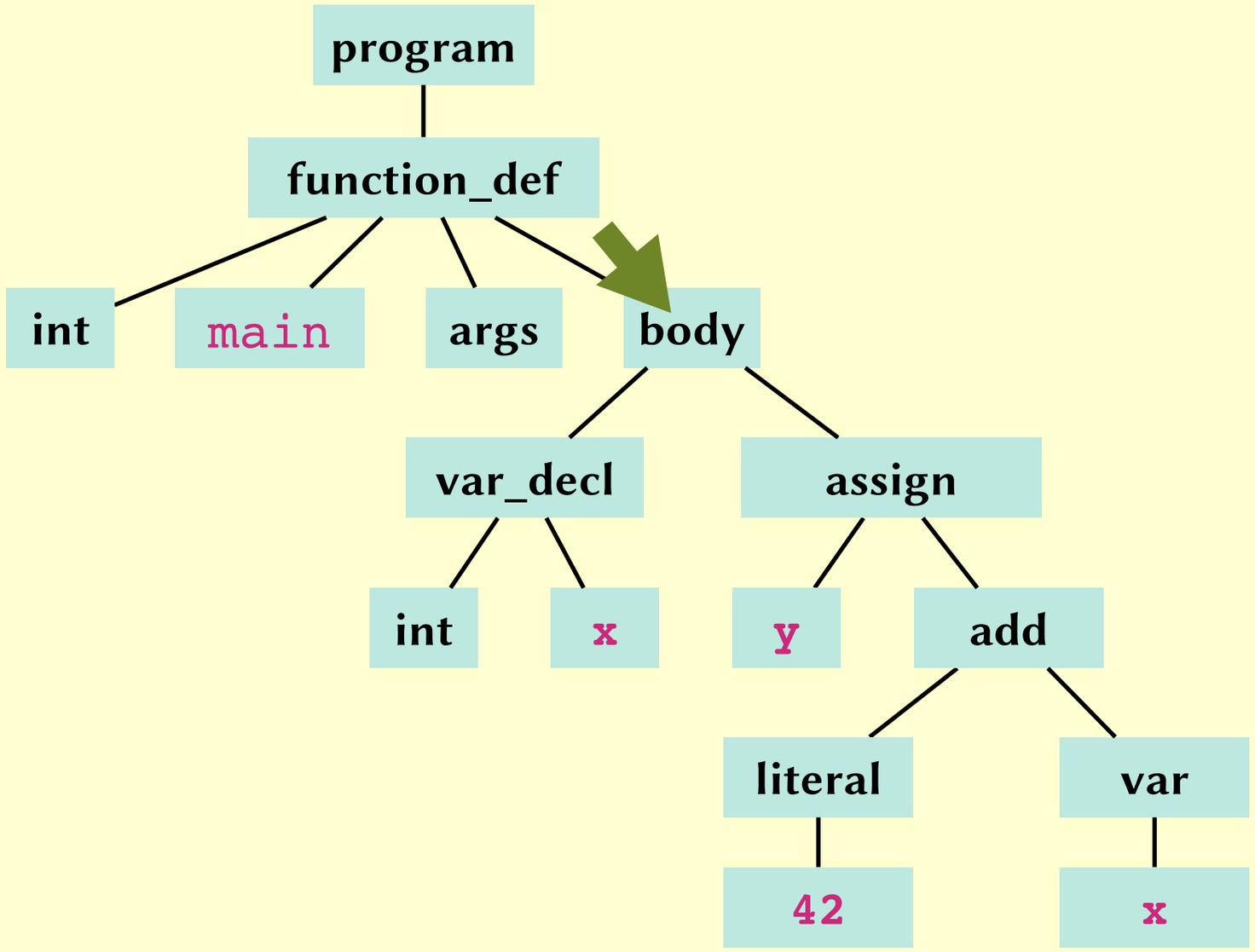


Symbol Table

Name	Type
`main`	void → int

Type checking in C

```
int main () { int x; y = 42+x; }
```

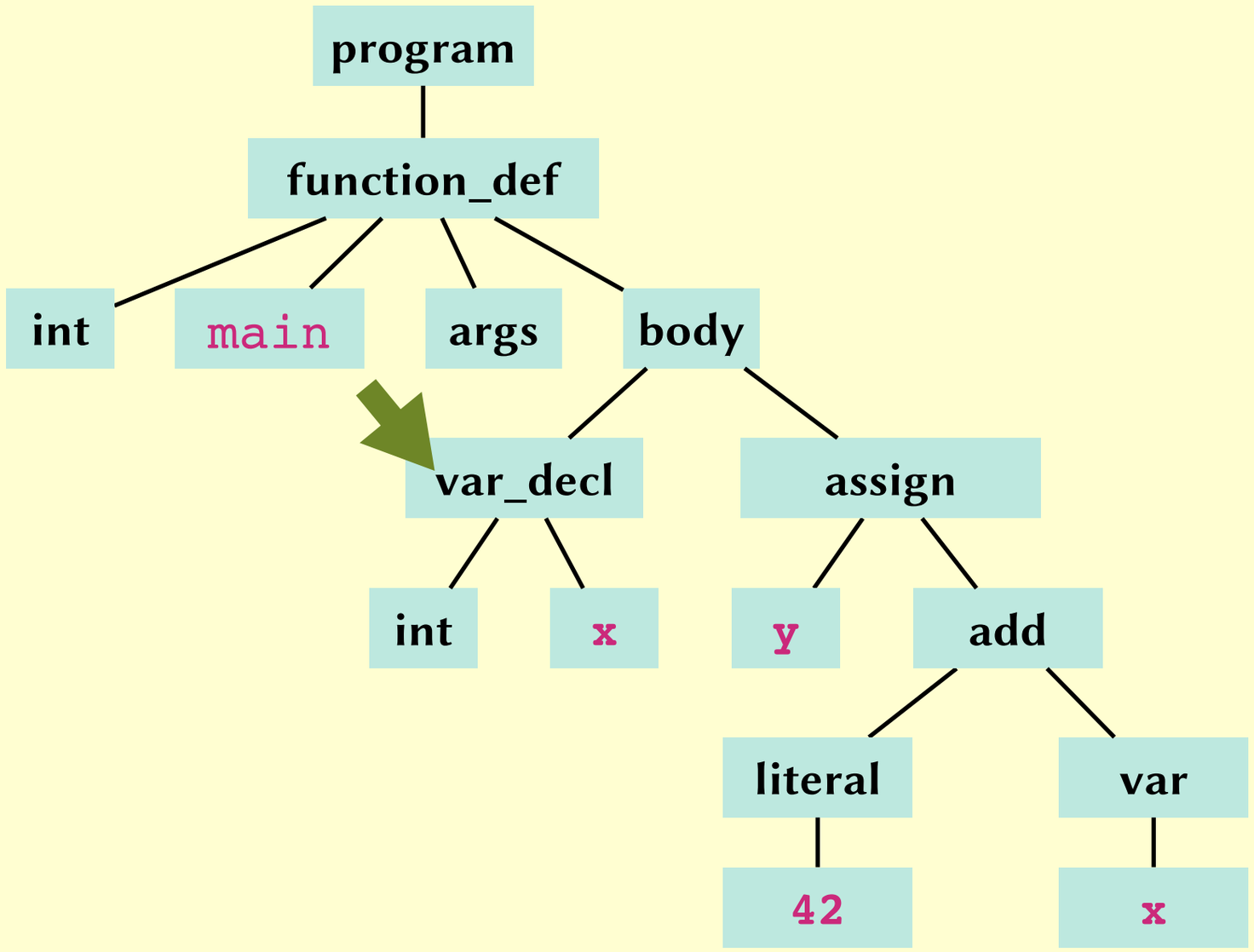


Symbol Table

Name	Type
`main`	void → int

Type checking in C

```
int main () { int x; y = 42+x; }
```

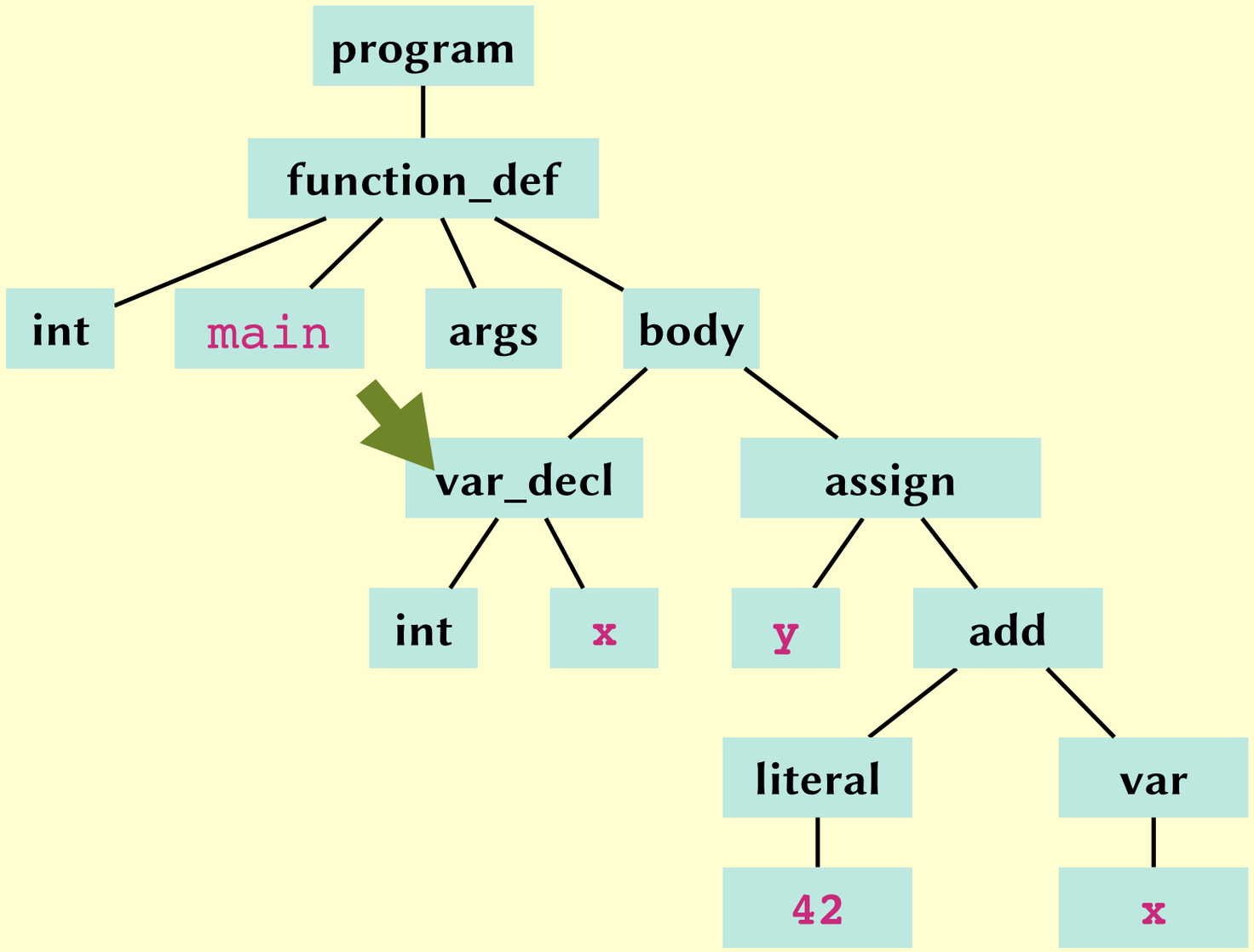


Symbol Table

Name	Type
`main`	void ⟶ int

Type checking in C

```
int main () { int x; y = 42+x; }
```

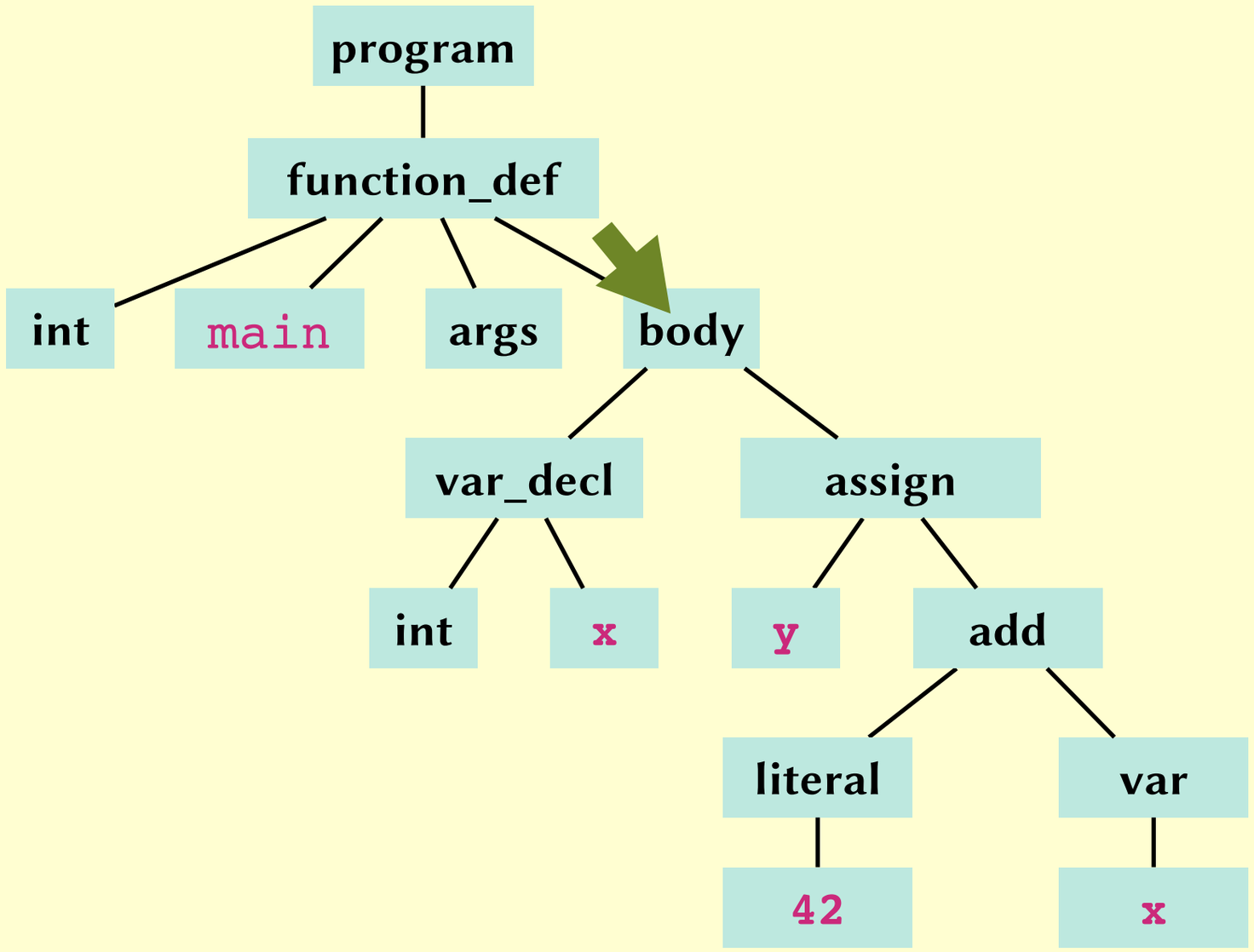


Symbol Table

Name	Type
`main`	void → int
`x`	int

Type checking in C

```
int main () { int x; y = 42+x; }
```

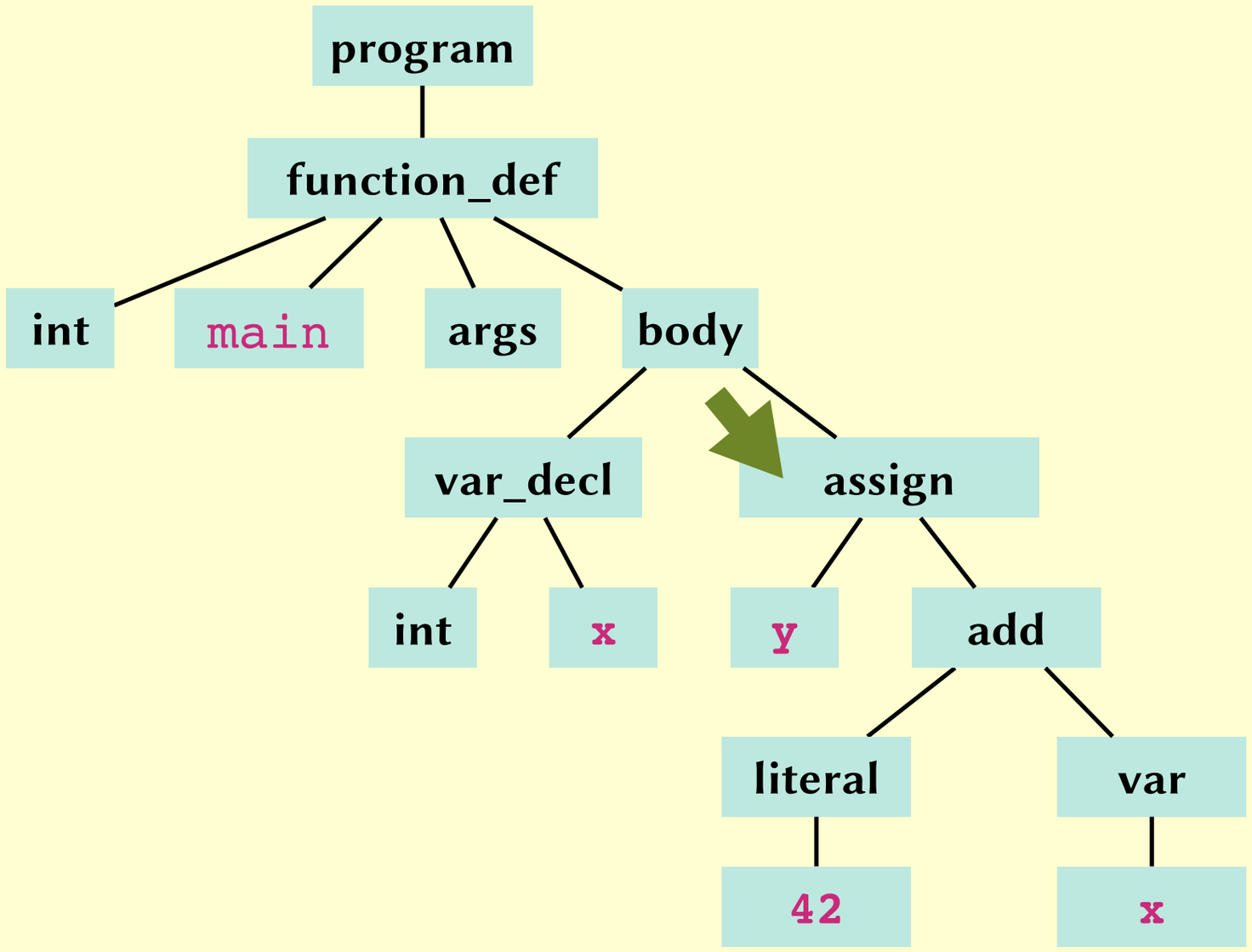


Symbol Table

Name	Type
`main`	void ⟶ int
`x`	int

Type checking in C

```
int main () { int x; y = 42+x; }
```

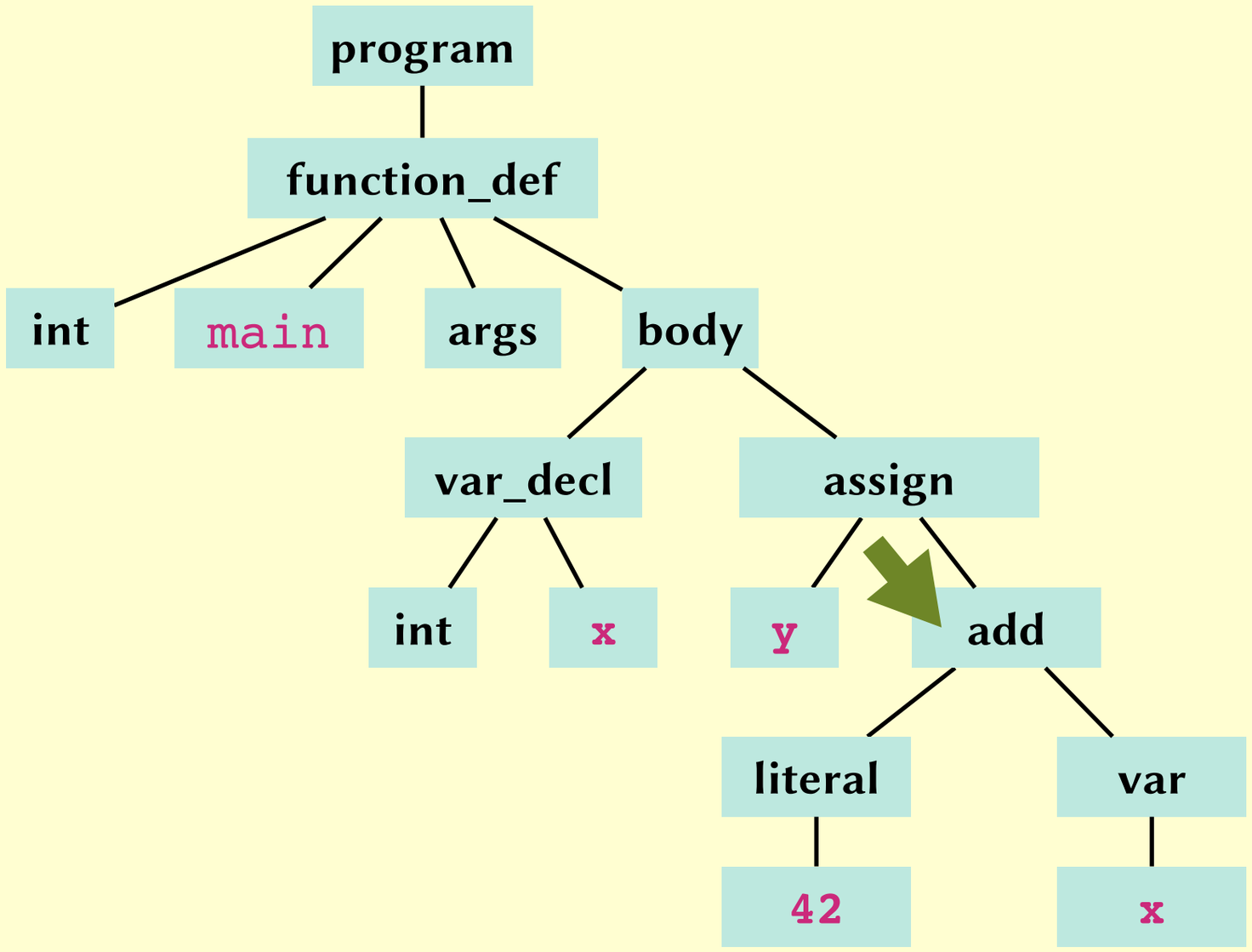


Symbol Table

Name	Type
`main`	void ⟶ int
`x`	int

Type checking in C

```
int main () { int x; y = 42+x; }
```

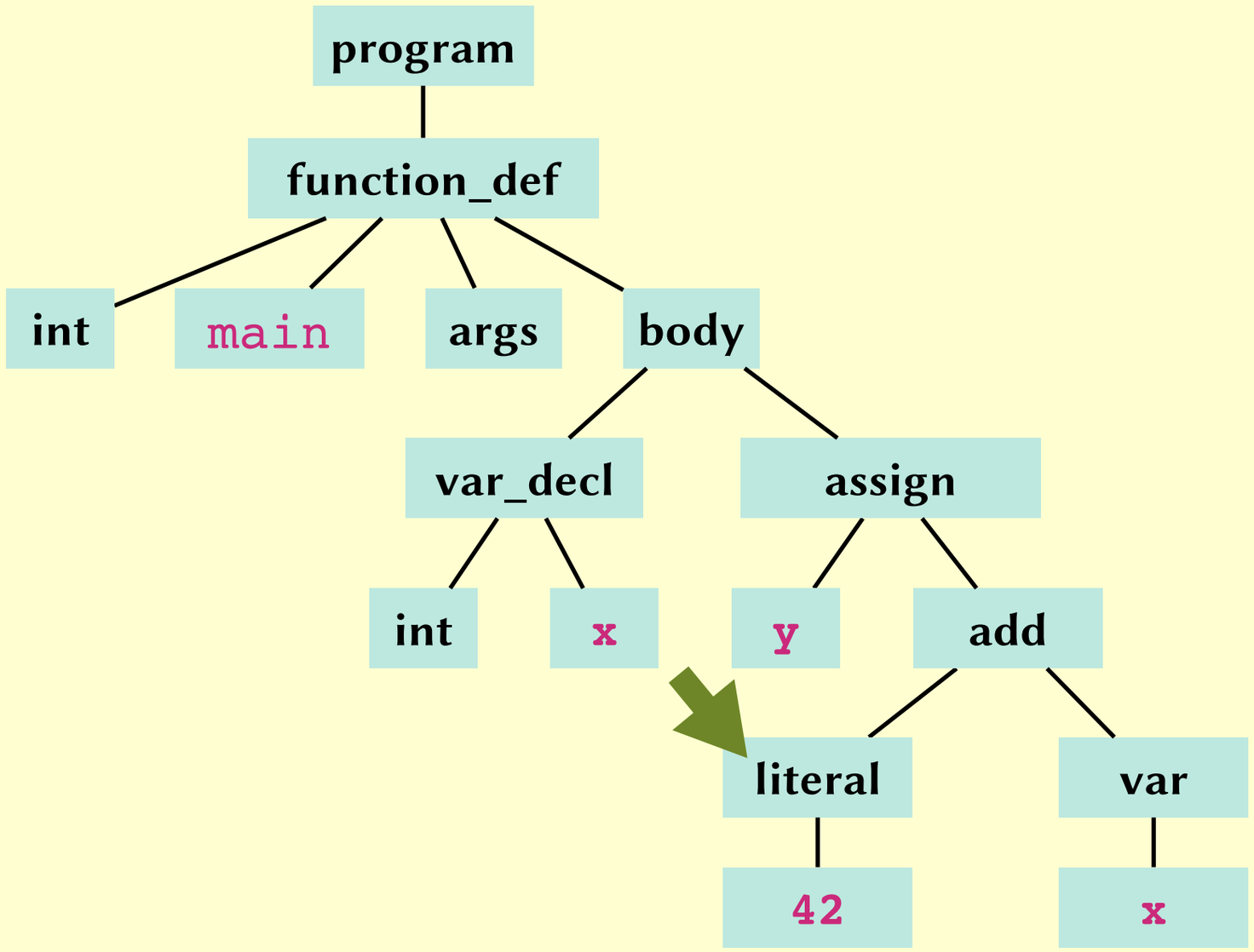


Symbol Table

Name	Type
`main`	void → int
`x`	int

Type checking in C

```
int main () { int x; y = 42+x; }
```

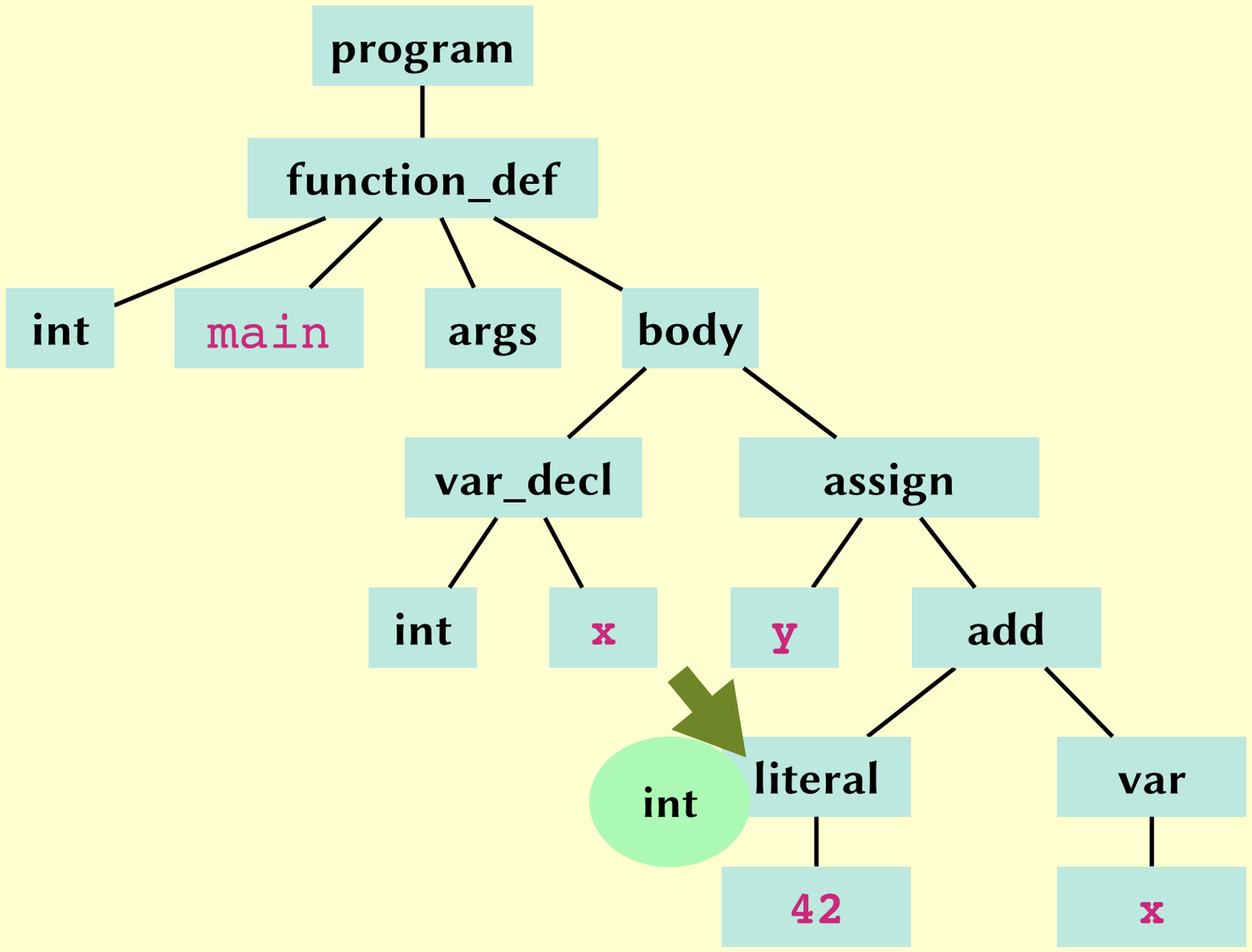


Symbol Table

Name	Type
`main`	void → int
`x`	int

Type checking in C

```
int main () { int x; y = 42+x; }
```

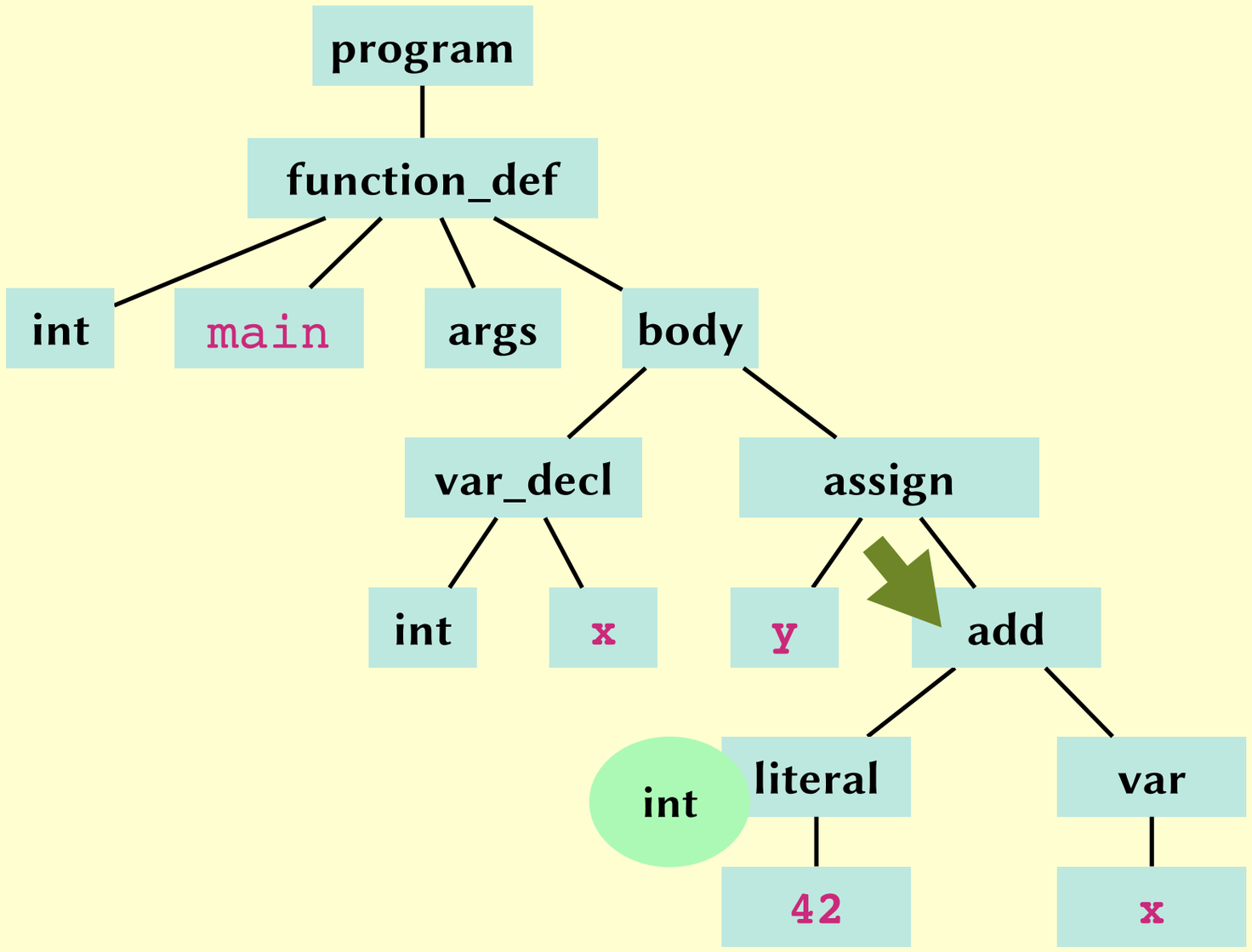


Symbol Table

Name	Type
`main`	void ⟶ int
`x`	int

Type checking in C

```
int main () { int x; y = 42+x; }
```

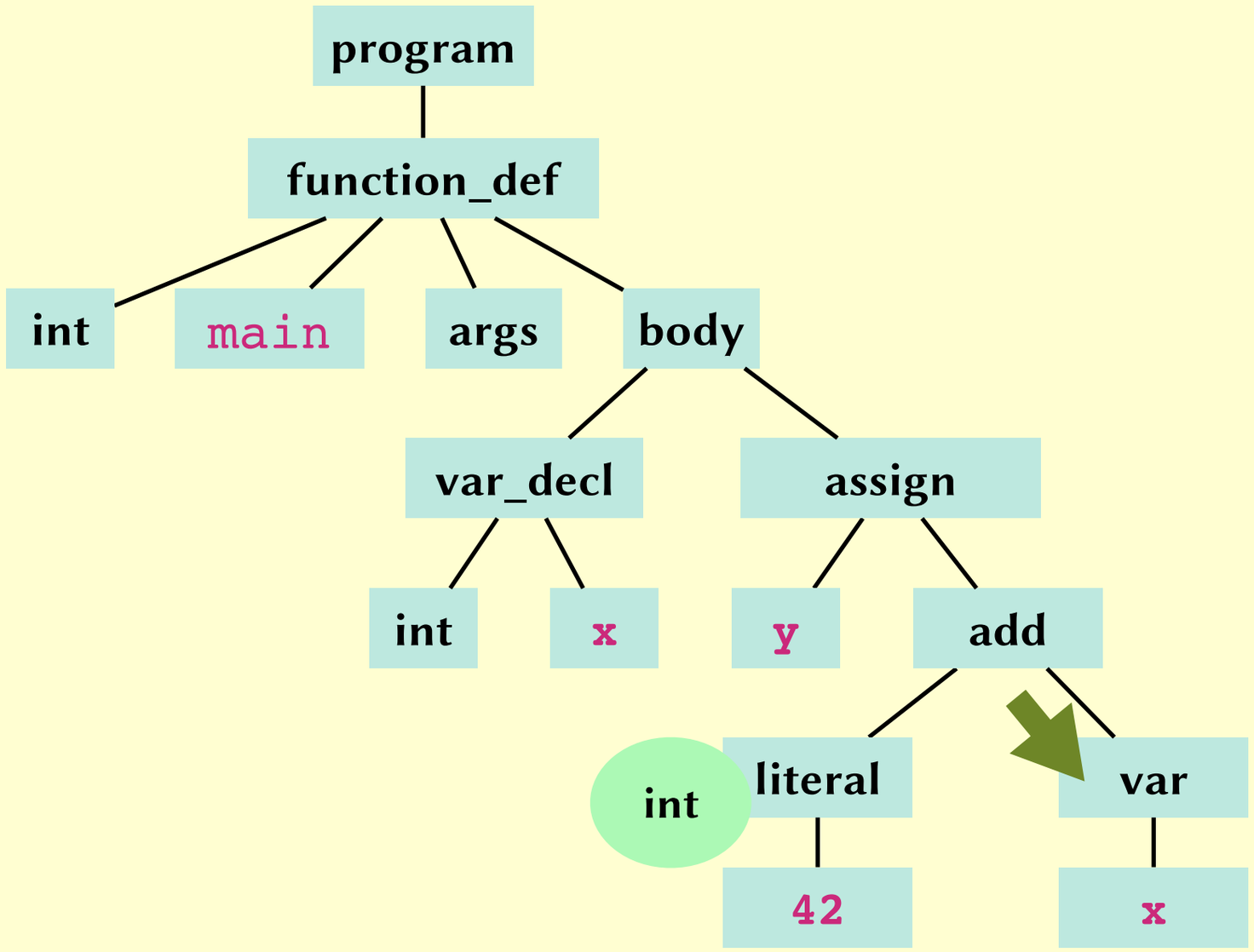


Symbol Table

Name	Type
`main`	void ⟶ int
`x`	int

Type checking in C

```
int main () { int x; y = 42+x; }
```

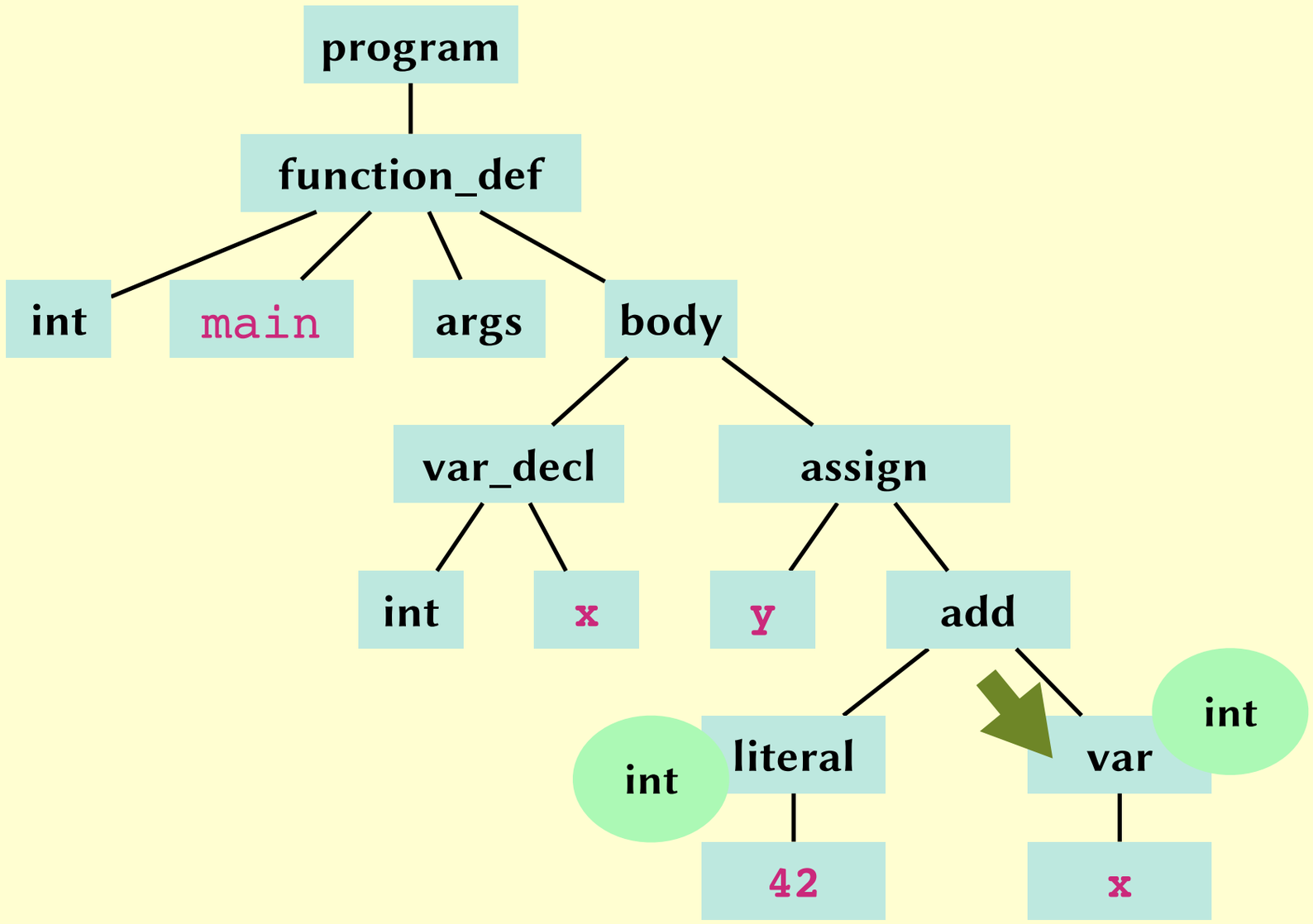


Symbol Table

Name	Type
`main`	void ⟶ int
`x`	int

Type checking in C

```
int main () { int x; y = 42+x; }
```

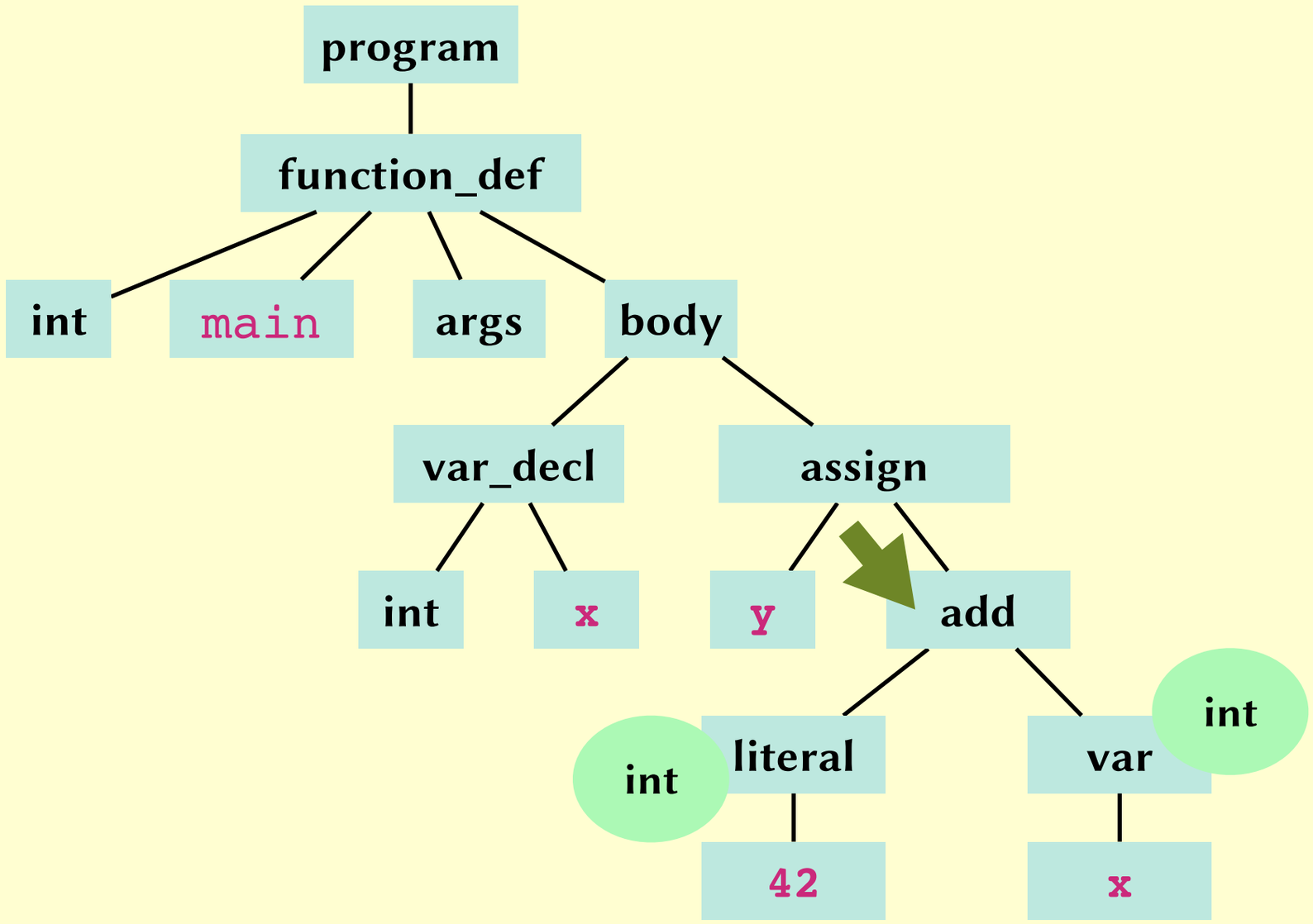


Symbol Table

Name	Type
`main`	void ⟶ int
`x`	int

Type checking in C

```
int main () { int x; y = 42+x; }
```

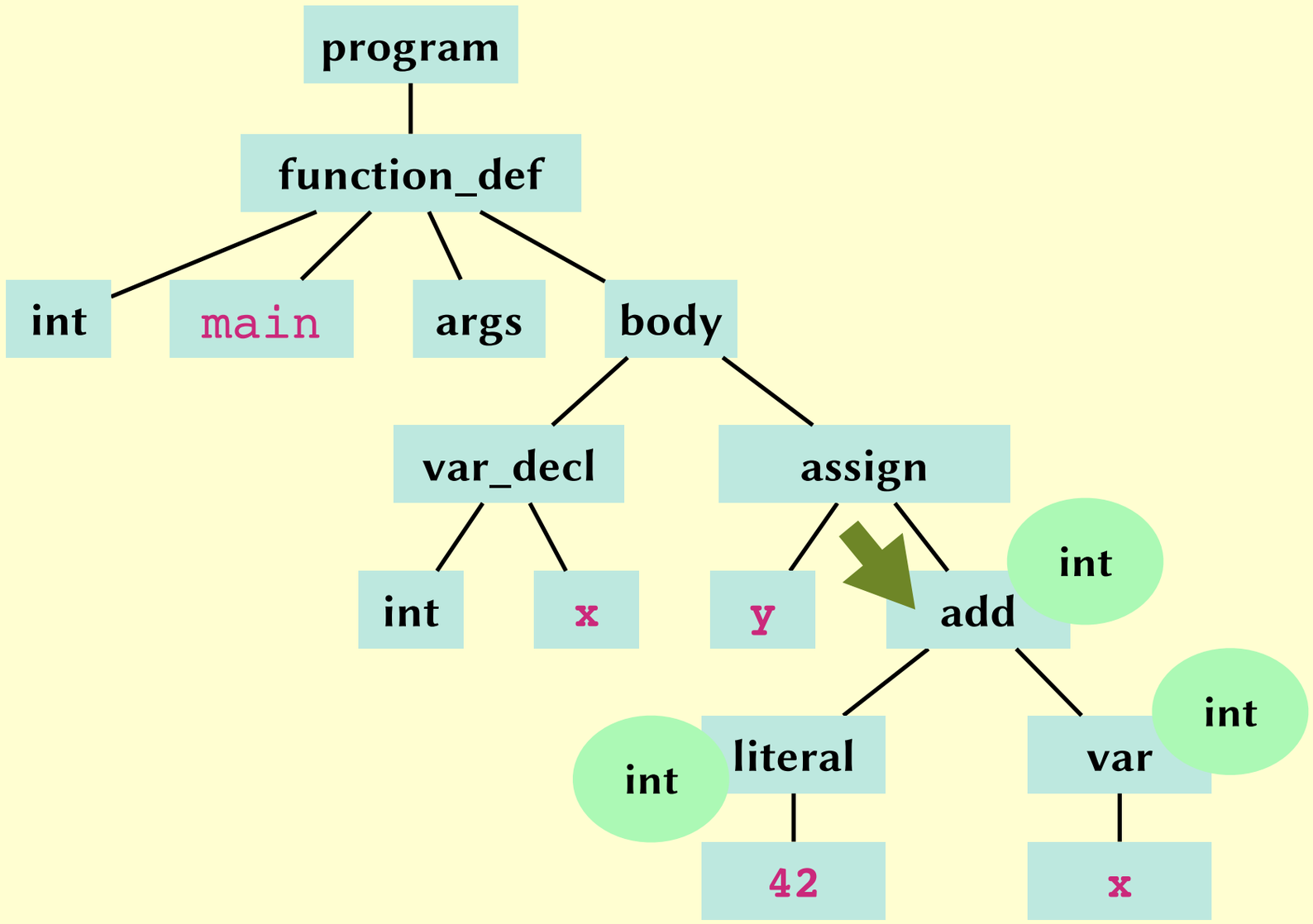


Symbol Table

Name	Type
`main`	void ⟶ int
`x`	int

Type checking in C

```
int main () { int x; y = 42+x; }
```

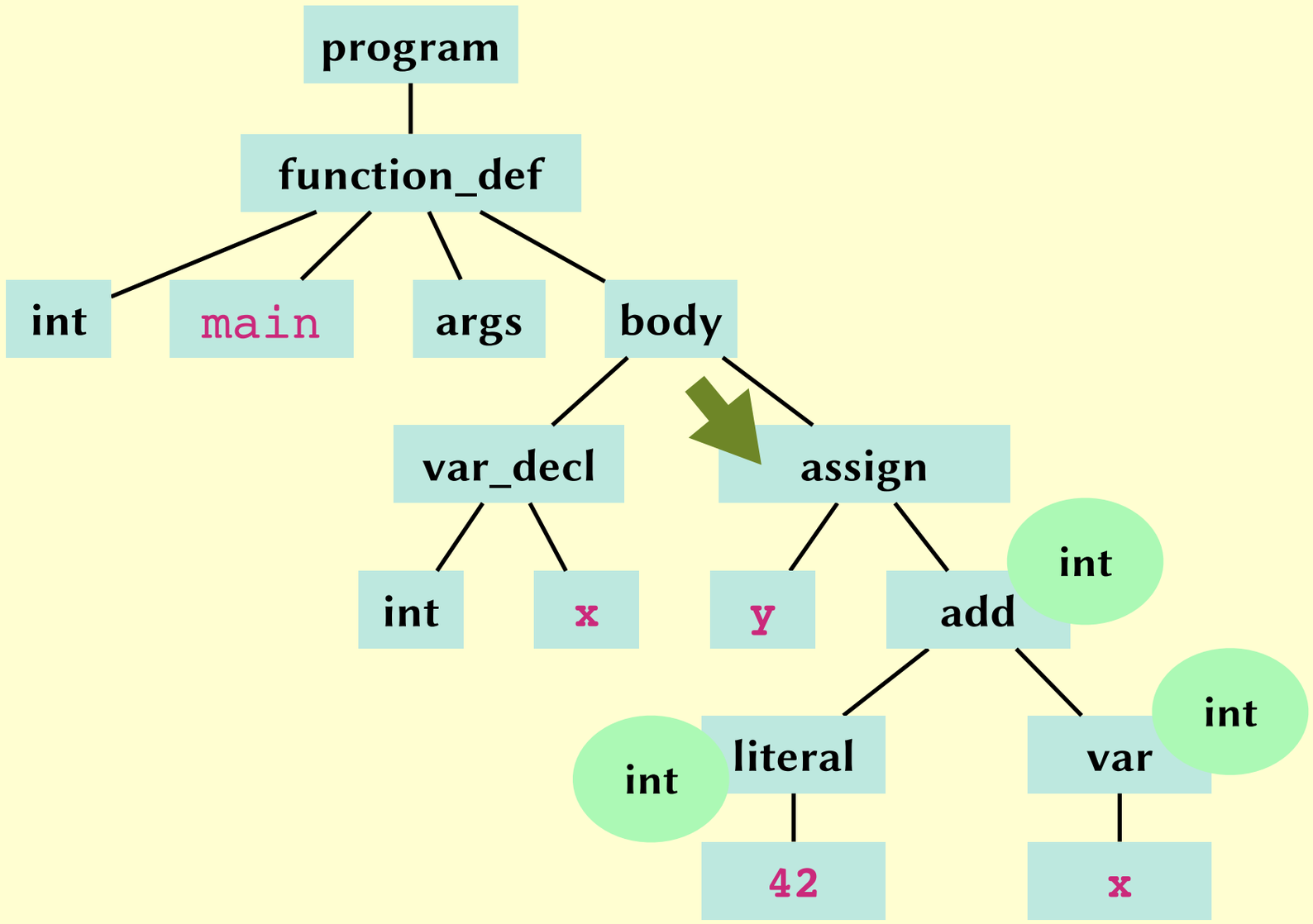


Symbol Table

Name	Type
`main`	void ⟶ int
`x`	int

Type checking in C

```
int main () { int x; y = 42+x; }
```

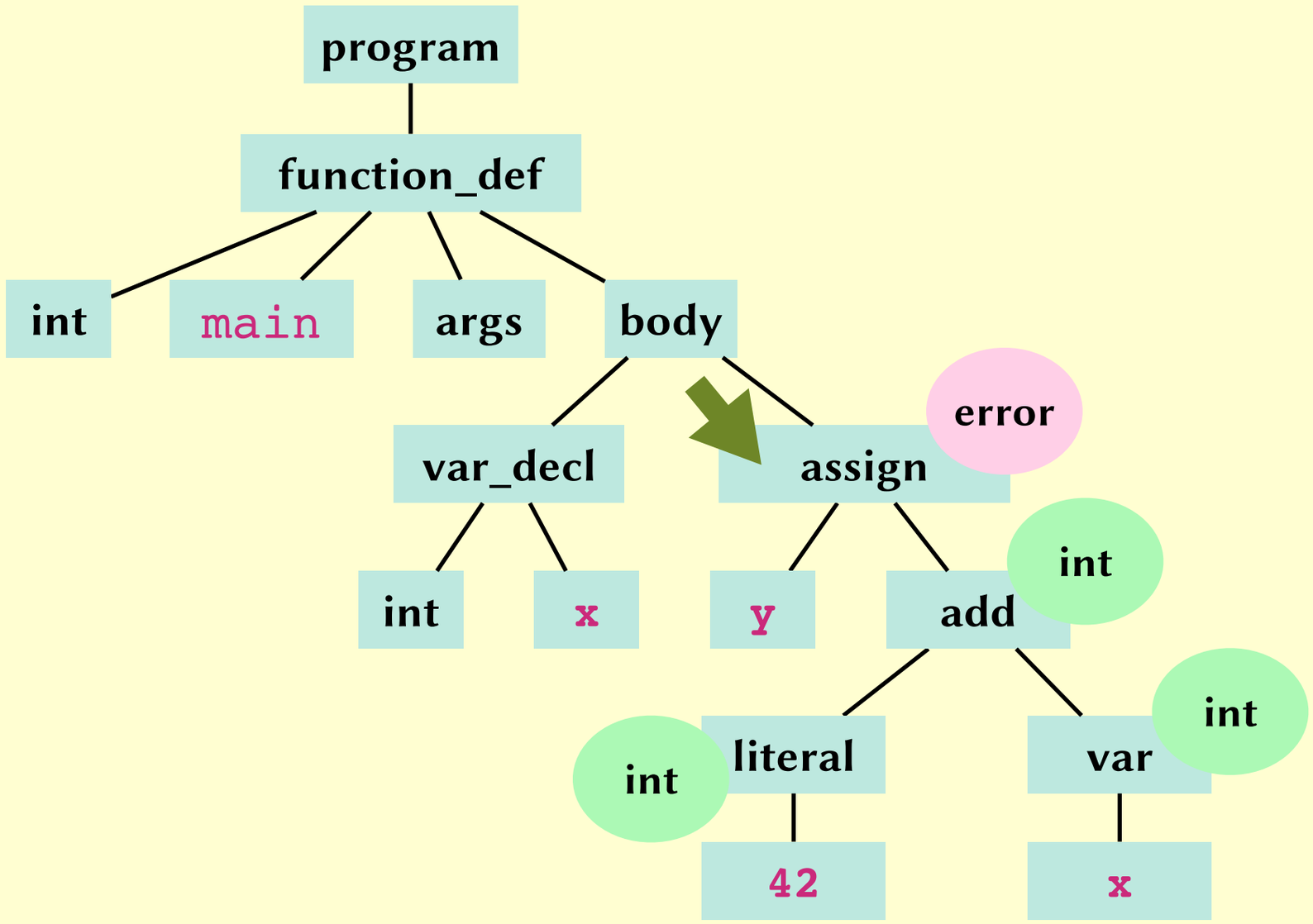


Symbol Table

Name	Type
`main`	void ⟶ int
`x`	int

Type checking in C

```
int main () { int x; y = 42+x; }
```



Symbol Table

Name	Type
`main`	void ⟶ int
`x`	int

Type checking in C

- Another example, featuring *function calls*.

```
void foo(int a) {...}

int baz(int b, char c) {...}

int main() {
  foo(42);
  return baz(17,'g');
}
```

Name	Type
`foo`	int ⟶ void
`baz`	(int × char) ⟶ int
`main`	void ⟶ int

Type checking in C

- Convenient time to check for other programming errors.

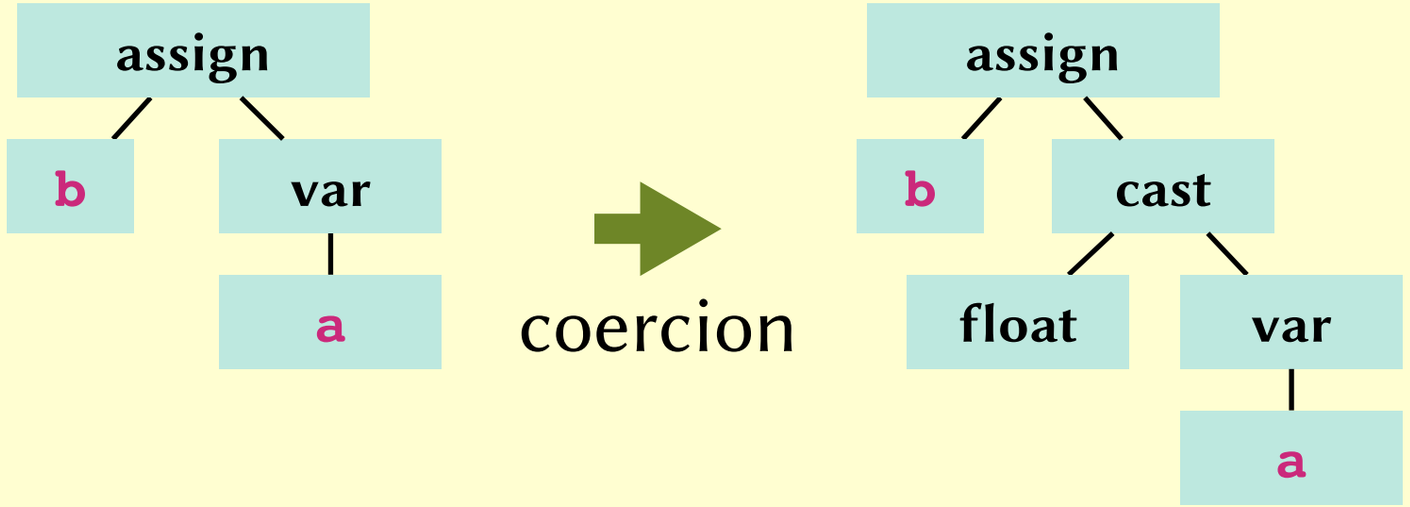
```
switch(x) {
case 1: y=42;
case 3: y=45;
case 1: y=0;
}
```

```
int main() {
  break;
  return 0;
}
```

```
int main() {
  int x;
  int x;
  return 0;
}
```

Type checking in C

- Types don't always need to match *exactly*.



Name	Type
a	int
b	float

Type systems

✔Type checking

- Type inference
- Polymorphic typing
- Subtyping
- Even fancier type systems

A little language

Type ::= `int` | `bool` | Type → Type

`int → bool`

`bool → (int → bool)`

`(int → int) → int`

A little language

Type ::= `int` | `bool` | Type → Type

Expr ::= **x** *// variables*

```
foo   baz
```

A little language

Type ::= `int` | `bool` | Type → Type

Expr ::= **X** *// variables*
| **N** *// integer literals*

`2` `42`

A little language

Type ::= `int` | `bool` | Type → Type

Expr ::= **X** *// variables*
| **N** *// integer literals*
| Expr + Expr *// integer addition*

```
foo + 42
```

A little language

Type ::= `int` | `bool` | Type → Type

Expr ::= **X** *// variables*
| **N** *// integer literals*
| Expr + Expr *// integer addition*
| Expr > Expr *// integer comparison*

```
(foo + 42) > 59
```

A little language

Type ::= `int` | `bool` | Type → Type

Expr ::= **X** *// variables*
| **N** *// integer literals*
| Expr + Expr *// integer addition*
| Expr > Expr *// integer comparison*
| **if** Expr **then** Expr **else** Expr *// if-expressions*

```
if (foo + 42) > 59 then 1 else 0
```

A little language

Type ::= `int` | `bool` | Type → Type

```
Expr ::= X                                  // variables
       | N                                  // integer literals
       | Expr + Expr                        // integer addition
       | Expr > Expr                        // integer comparison
       | if Expr then Expr else Expr        // if-expressions
       | true                               // boolean literal
       | false                              // boolean literal
```

A little language

Type ::= `int` | `bool` | Type → Type

```
Expr ::= X                                    // variables
       | N                                    // integer literals
       | Expr + Expr                          // integer addition
       | Expr > Expr                          // integer comparison
       | if Expr then Expr else Expr          // if-expressions
       | true                                 // boolean literal
       | false                                // boolean literal
       | let X = Expr in Expr                 // assignment
```

```
let a = 42 in
let b = 17+a in
a+b
```

A little language

Type ::= `int` | `bool` | Type → Type

```
Expr ::= X                                  // variables
       | N                                  // integer literals
       | Expr + Expr                        // integer addition
       | Expr > Expr                        // integer comparison
       | if Expr then Expr else Expr        // if-expressions
       | true                               // boolean literal
       | false                              // boolean literal
       | let X = Expr in Expr               // assignment
       | fun X => Expr                      // anonymous function
```

```
(fun a => a + 1)
```

A little language

Type ::= `int` | `bool` | Type → Type

Expr ::=	**X**	*// variables*
|	**N**	*// integer literals*
|	Expr `+` Expr	*// integer addition*
|	Expr `>` Expr	*// integer comparison*
|	`if` Expr `then` Expr `else` Expr	*// if-expressions*
|	`true`	*// boolean literal*
|	`false`	*// boolean literal*
|	`let` **X** `=` Expr `in` Expr	*// assignment*
|	`fun` **X** `=>` Expr	*// anonymous function*
|	Expr `(` Expr `)`	*// function call*

```
(fun a => a + 1)(2)
```

A little language

Type ::= `int` | `bool` | Type → Type

```
Expr ::= X                                  // variables
       | N                                  // integer literals
       | Expr + Expr                        // integer addition
       | Expr > Expr                        // integer comparison
       | if Expr then Expr else Expr        // if-expressions
       | true                               // boolean literal
       | false                              // boolean literal
       | let X = Expr in Expr               // assignment
       | fun X => Expr                      // anonymous function
       | Expr (Expr)                        // function call
```

```
let i = (fun a => a) in
let d = (fun a => a+a) in
d(i(2))
```

Inference rules

$$\frac{\text{If I'm a man, then I'm mortal} \qquad \text{I'm a man}}{\text{I'm mortal}}\ \textit{(Modus Ponens)}$$

Theophrastus
371BC – 287BC

John Wickerson
1987–

$$\frac{\text{All mortals are green} \qquad \text{Socrates is mortal}}{\text{Socrates is green}}$$

$$\frac{\text{likes(X, Z)} \qquad \text{cancook(Y, Z)}}{\text{wouldgetonwith(X, Y)}}$$

$$\dfrac{\dfrac{}{a\times(b+c) = a\times b + a\times c}\ \textit{(distrib)} \qquad \dfrac{\dfrac{}{a\times b = b\times a}\ \textit{(commut)} \qquad \dfrac{}{a\times c = c\times a}\ \textit{(commut)}}{a\times b + a\times c = b\times a + c\times a}\ \textit{(congr)}}{a\times(b+c) = b\times a + c\times a}\ \textit{(transitivity)}$$

Type inference

$$\frac{}{\texttt{n} \textit{ has type } \texttt{int}} \qquad \frac{\texttt{e1} \textit{ has type } \texttt{int} \quad \texttt{e2} \textit{ has type } \texttt{int}}{\texttt{e1} + \texttt{e2} \textit{ has type } \texttt{int}}$$

$$\frac{\texttt{e1} \textit{ has type } \texttt{int} \quad \texttt{e2} \textit{ has type } \texttt{int}}{\texttt{e1} < \texttt{e2} \textit{ has type } \texttt{bool}}$$

Type inference

$$\frac{}{A \vdash n : \texttt{int}} \qquad \frac{A \vdash e1 : \texttt{int} \quad A \vdash e2 : \texttt{int}}{A \vdash e1 \texttt{ + } e2 : \texttt{int}} \qquad \frac{(x,\tau) \in A}{A \vdash x : \tau}$$

$$\frac{A \vdash e1 : \texttt{int} \quad A \vdash e2 : \texttt{int}}{A \vdash e1 \texttt{ < } e2 : \texttt{bool}} \qquad \frac{A \vdash e1 : \texttt{bool} \quad A \vdash e2 : \tau \quad A \vdash e3 : \tau}{A \vdash \texttt{if } e1 \texttt{ then } e2 \texttt{ else } e3 : \tau}$$

$$A = \{ (\texttt{foo}, \texttt{int}), (\texttt{baz}, \texttt{bool}) \}$$

Type inference

$$\frac{}{A \vdash n : \texttt{int}} \qquad \frac{A \vdash e1 : \texttt{int} \quad A \vdash e2 : \texttt{int}}{A \vdash e1 + e2 : \texttt{int}} \qquad \frac{(x,\tau) \in A}{A \vdash x : \tau}$$

$$\frac{A \vdash e1 : \texttt{int} \quad A \vdash e2 : \texttt{int}}{A \vdash e1 < e2 : \texttt{bool}} \qquad \frac{A \vdash e1 : \texttt{bool} \quad A \vdash e2 : \tau \quad A \vdash e3 : \tau}{A \vdash \textbf{if } e1 \textbf{ then } e2 \textbf{ else } e3 : \tau}$$

$$\frac{}{A \vdash \textbf{true} : \texttt{bool}} \qquad \frac{}{A \vdash \textbf{false} : \texttt{bool}}$$

$$\frac{A \vdash e1 : \tau' \qquad A+(x,\tau') \vdash e2 : \tau}{A \vdash \textbf{let } x = e1 \textbf{ in } e2 : \tau}$$

$$\frac{\emptyset \vdash 5 : \texttt{int} \quad \{\texttt{a} : \texttt{int}\} \vdash \texttt{a} > 3 : \texttt{bool}}{\emptyset \vdash \textbf{let } \texttt{a} = 5 \textbf{ in } \texttt{a} > 3 : \texttt{bool}}$$

Type inference

$$\frac{}{A \vdash n : \texttt{int}} \qquad \frac{A \vdash e1 : \texttt{int} \quad A \vdash e2 : \texttt{int}}{A \vdash e1 + e2 : \texttt{int}} \qquad \frac{(x,\tau) \in A}{A \vdash x : \tau}$$

$$\frac{A \vdash e1 : \texttt{int} \quad A \vdash e2 : \texttt{int}}{A \vdash e1 < e2 : \texttt{bool}} \qquad \frac{A \vdash e1 : \texttt{bool} \quad A \vdash e2 : \tau \quad A \vdash e3 : \tau}{A \vdash \textbf{if } e1 \textbf{ then } e2 \textbf{ else } e3 : \tau}$$

$$\frac{}{A \vdash \textbf{true} : \texttt{bool}} \qquad \frac{}{A \vdash \textbf{false} : \texttt{bool}} \qquad \frac{A+(x,\tau) \vdash e : \tau'}{A \vdash \textbf{fun } x \Rightarrow e : \tau \to \tau'}$$

$$\frac{A \vdash e1 : \tau' \qquad A+(x,\tau') \vdash e2 : \tau}{A \vdash \textbf{let } x = e1 \textbf{ in } e2 : \tau} \qquad \frac{A \vdash e1 : \tau \to \tau' \qquad A \vdash e2 : \tau}{A \vdash e1\ (e2) : \tau'}$$

An interesting connection

$$\frac{A \vdash e1 : \tau \to \tau' \qquad A \vdash e2 : \tau}{A \vdash e1\ (e2) : \tau'}$$

$$\frac{\tau \to \tau' \qquad \tau}{\tau'}$$

$$\frac{\text{man} \to \text{mortal} \qquad \text{man}}{\text{mortal}}$$

$$\frac{\text{If I'm a man, then I'm mortal} \qquad \text{I'm a man}}{\text{I'm mortal}}\ \textit{(Modus Ponens)}$$

Haskell B. Curry
1900–1982

William Howard
1926–

Theophrastus
371BC – 287BC

Type systems

✔Type checking

✔Type inference

- Polymorphic typing
- Subtyping
- Even fancier type systems

Polymorphic typing

- The above approach to type-inference fails if given:

```
let i = (fun a => a) in
let d = (fun a => a+a) in
i(d)(i(2))
```

because the type system does not support *polymorphism*.

- Even polymorphic type inference would fail if given:

```
if false then 5 else true
```

Type systems

✔Type checking

✔Type inference

✔Polymorphic typing

- Subtyping

- Even fancier type systems

Subtyping

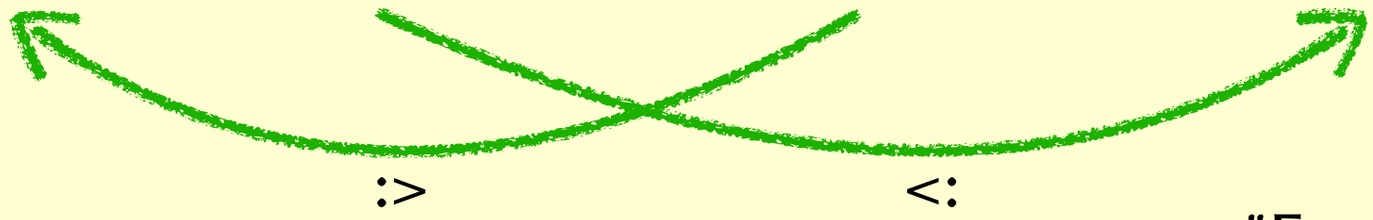
- `int` <: `float`
- `Labrador` <: `Dog` <: `Animal`

$$\frac{A \vdash e : \tau' \qquad \tau' <: \tau}{A \vdash e : \tau}$$

- `Tshirt` <: `Clothing`
- `struct {int a; int b;}` <: `struct {int a;}`
- `Dog → Tshirt`

Subtyping

- `int` <: `float`
- `Labrador` <: `Dog` <: `Animal`
- `Tshirt` <: `Clothing`
- **`struct`** `{int a; int b;}` <: **`struct`** `{int a;}`
- `Dog → Tshirt` <: `Labrador → Clothing`



$$\frac{A \vdash e : \tau' \qquad \tau' <: \tau}{A \vdash e : \tau}$$

"Functions are *contravariant* in the input type and *covariant* in the output type."

- `in`
- `Lab`
- `Tsh`
- `str` `}`
- `Dog`

$$\frac{\tau' <: \tau}{\quad : \tau}$$

travariant in
covariant in

output type.

Type systems

✔Type checking

✔Type inference

✔Polymorphic typing

✔Subtyping

- Even fancier type systems

Units of measure

- ```
  float<m> distance;
  float<s> time;
  float<m/s> speed;
  ```

- System is implemented in the F# language.

- Would have been handy for the Mars Climate Orbiter in 1999.


  ```

Dependent types

- `int[][] mult (int[][] A, int[][] B);`
- `int[n][p] mult (int[n][m] A, int[m][p] B);`
- `int[len] makeArray(int len);`
- Type-checking now gives much stronger guarantees.
- But type-checking becomes much more complicated.

Summary

- Designing type systems involves a three-way trade-off:
  - Type system **should not restrict** programmers.
  - Type system **should detect many errors**.
  - Type checking/inference **should run quickly**.
- Some key phrases:

Summary

- Designing type systems involves a three-way trade-off:
  - Type system **should not restrict** programmers.
  - Type system **should detect many errors**.
  - Type checking/inference **should run quickly**.
- Some key phrases: type checking

Summary

- Designing type systems involves a three-way trade-off:
  - Type system **should not restrict** programmers.
  - Type system **should detect many errors**.
  - Type checking/inference **should run quickly**.
- Some key phrases: type checking, type inference

Summary

- Designing type systems involves a three-way trade-off:
  - Type system **should not restrict** programmers.
  - Type system **should detect many errors**.
  - Type checking/inference **should run quickly**.
- Some key phrases: type checking, type inference, coercion

Summary

- Designing type systems involves a three-way trade-off:
  - Type system **should not restrict** programmers.
  - Type system **should detect many errors**.
  - Type checking/inference **should run quickly**.
- Some key phrases: type checking, type inference, coercion, polymorphism

Summary

- Designing type systems involves a three-way trade-off:
  - Type system **should not restrict** programmers.
  - Type system **should detect many errors**.
  - Type checking/inference **should run quickly**.
- Some key phrases: type checking, type inference, coercion, polymorphism, subtype

Summary

- Designing type systems involves a three-way trade-off:
  - Type system **should not restrict** programmers.
  - Type system **should detect many errors**.
  - Type checking/inference **should run quickly**.
- Some key phrases: type checking, type inference, coercion, polymorphism, subtype, covariance and contravariance

Summary

- Designing type systems involves a three-way trade-off:
  - Type system **should not restrict** programmers.
  - Type system **should detect many errors**.
  - Type checking/inference **should run quickly**.
- Some key phrases: type checking, type inference, coercion, polymorphism, subtype, covariance and contravariance, dependent type.