Compilers Principles Lab5 Report

Language Feature Extension

Yurun Yuan

CS, USTC

January 21, 2022

Outline

Outline

Declarations of Variables

- Pointers
  `int *ptr;`
- Function
  `int func(int);`
- Arrays
  `int array[42];`

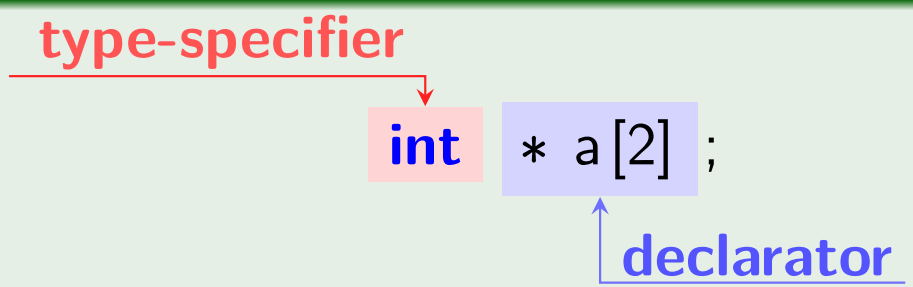
- Structures
  `struct S {...};`
  `struct S s;`
  `struct S {...} s;`
  `struct {...} anonymous;`

Declaration Grammar

Declaration grammar

$$\text{declaration} \rightarrow \quad \text{type-specifier declarator ;}$$

Example



Declaration Grammar

Declaration grammar

declaration → type-specifier declarator ;

- Pointer declarators
  int ∗ ptr ;
- Function declarators
  int func(**float**) ;
- Array declarators
  int array [42] ;

Crux: declarators can get mixed up

Example

int (∗ a [2])(**int**);

Declarators	Declarator type	Type expression
(∗ a [2])(**int**)	Function	**int**
∗ a [2]	Pointer	**int** → **int**
a [2]	Array	pointer(**int** → **int**)
a		array(2, pointer(**int** → **int**))

Trial 1: a naive solution

Example

int (* a[2])(**int**);

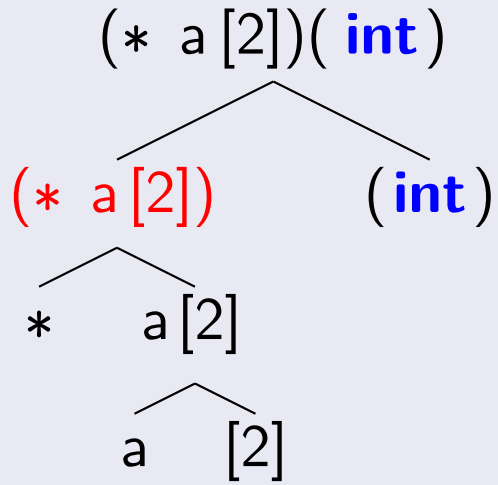
declarator	→	* declarator	***Pointer declarator***
	|	declarator (type, ...)	***Funtcion declarator***
	|	declarator [**int**]	***Array declarator***
	|	(declarator)	
	|	**ID**	

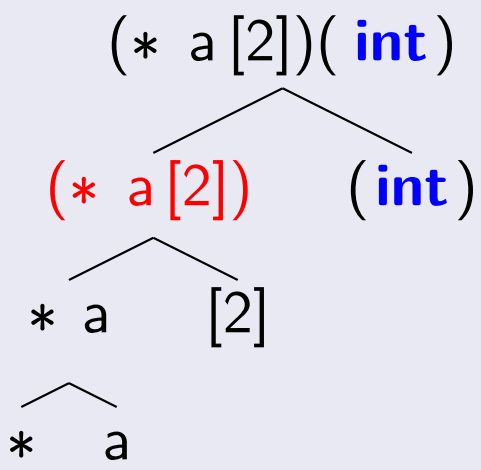
Trial 1: a naive solution

Example

int (* a[2])(**int**);

Problem: ambiguity





Trial 2: eliminating ambiguity

Preced	Operation
0	Parenthesis ()
1	Call () subscript []
2	Dereference *

- Parentheses and identifiers

$$\begin{aligned} \text{factor} &\rightarrow \textbf{ID} \\ &\mid \text{(decl)} \end{aligned}$$

- Function calls and array subscripting

$$\begin{aligned} \text{decl-1} &\rightarrow \text{decl-1(type, ...)} \\ &\mid \text{decl-1}[\textbf{int}] \\ &\mid \text{factor} \end{aligned}$$

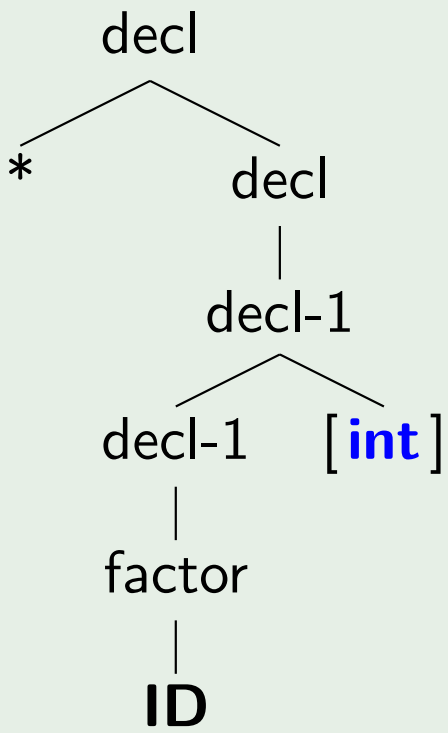
- Dereference

$$\begin{aligned} \text{decl} &\rightarrow \textbf{*} \text{ decl} \\ &\mid \text{decl-1} \end{aligned}$$

Trial 2: eliminating ambiguity

Example

int * array [2] ;



- Parentheses and identifiers

$$\begin{aligned} \text{factor} &\rightarrow \textbf{ID} \\ &\mid \text{(decl)} \end{aligned}$$

- Function calls and array subscripting

$$\begin{aligned} \text{decl-1} &\rightarrow \text{decl-1(type, ...)} \\ &\mid \text{decl-1}[\textbf{int}] \\ &\mid \text{factor} \end{aligned}$$

- Dereference

$$\begin{aligned} \text{decl} &\rightarrow * \text{ decl} \\ &\mid \text{decl-1} \end{aligned}$$

Declarations of Structures

- Structure definition
  struct S {...};
- Variable definition
  struct S s;
- Structure and variable definition
  struct S {...} s;
- Anonymous structure
  struct {...} anonymous;

Extension of the declaration grammar

Declaration grammar

declaration → type-specifier declarator ;

Declaration	Type specifier	Declarator
struct S {...} ;	**struct** S {...}	∅
struct S s ;	**struct** S	s
struct S {...} s ;	**struct** S {...}	s
struct {...} anonymous ;	**struct** {...}	anonymous

Extension of the declaration grammar

Declaration grammar

$$\text{declaration} \rightarrow \text{type-specifier declarator ;}$$

$$\begin{aligned}
\text{type-specifier} &\rightarrow \cdots \\
&\mid \textbf{struct ID}\{\text{definitions}\} \\
&\mid \textbf{struct}\{\text{definitions}\} \\
&\mid \textbf{struct ID}
\end{aligned}$$

$$\begin{aligned}
\text{declarator} &\rightarrow \cdots \\
&\mid \epsilon
\end{aligned}$$

Self reference

Example

A naive implementation of the node of a forwarding linked list

```
struct list_node{
    int elem;
    struct list_node* next;
};
```

The identifier `struct list_node` is visible within the definition of the structure, although `struct list_node` is an incomplete type. To render the identifier `struct list_node` available, construct an empty structure before parsing the body of the structure definition, and complete the type after the definition parsing is done.

Outline

Supported operators

- Arithmatic operators
  $+,-,*,/$
- Relational operators
  $>,<,>=,<=,==,!=$
- Assignment operator
  $=$
- Array subscript
  array [index]
- Pointer dereference
  $*$ ptr
- Address of
  & lvalue
- Member access
  var . member
- Function call
  callable (params)

Precedence and CFG

0. Parenthesis ()

$$\begin{aligned} \text{expr-0} \rightarrow\ & \textbf{ID} \\ \mid\ & \textbf{Integer-literal} \\ \mid\ & \textbf{Float-literal} \\ \mid\ & (\ \text{expr}\ ) \end{aligned}$$

1. Function call, array subscript, member access

$$\begin{aligned} \text{expr-1} \rightarrow\ & \text{expr-1}\ (\ \text{args}\ ) \\ \mid\ & \text{expr-1}\ [\ \text{expr}\ ] \\ \mid\ & \text{expr-1}\ .\ \textbf{ID} \\ \mid\ & \text{expr-0} \end{aligned}$$

Precedence and CFG

2. Dereference, address of

$$\begin{aligned} \text{expr-2} \rightarrow & * \text{ expr-2} \\ | & \ \& \text{ expr-2} \\ | & \ \text{expr-1} \end{aligned}$$

3. Multiplication, division

$$\begin{aligned} \text{expr-3} \rightarrow & \text{ expr-3 } \textbf{MulOp} \text{ expr-2} \\ | & \ \text{expr-2} \end{aligned}$$

4. Addition subtraction

$$\begin{aligned} \text{expr-4} \rightarrow & \text{ expr-4 } \textbf{AddOp} \text{ expr-3} \\ | & \ \text{expr-3} \end{aligned}$$

Precedence and CFG

5. Relational operations

$$\begin{aligned} \text{expr-5} &\rightarrow \text{expr-5 } \textbf{RelOp} \text{ expr-4} \\ &\mid \text{expr-4} \end{aligned}$$

6. Assignment

$$\begin{aligned} \text{expr} &\rightarrow \text{expr-5} = \text{expr} \\ &\mid \text{expr-5} \end{aligned}$$

Note

The assignment operator is right associated, so the production is expr $\rightarrow$ expr-5 = expr instead of expr $\rightarrow$ expr = expr-5.

Problems with operations on structures

- Assignment to structures
  struct S s = t;
- Structures as parameters
  void func(**struct** S){...};
- Structures as return values
  struct S func (){...};

Assignment to structures

- Source code

```
struct S
{
    int member1;
    float member2;
};

s = t;
```

- Transformed code

```
struct S{
    int member1;
    float member2;
};

s.member1 = t.member1;
s.member2 = t.member2;
```

Structures as parameters

- Source code

```
void func(struct S s)
{
    ...
}

int main()
{
    func(t);
}
```

- Transformed code

```
void func(struct S *ptr)
{
    struct S s = *ptr;
}

int main()
{
    func(&t);
}
```

Structures as return values

- Source code

```
struct S func()
{
    ...
    return s;
}

int main()
{
    func();
}
```

- Transformed code

```
void func(struct S* ret_ptr)
{
    ...
    *ret_ptr = ret_value;
}

int main() {
    struct S ret_value;
    func(&ret_value);
}
```

Outline

Functions declared within a structure

Example

```
struct S
{
    ...
    void func(){}
};
```

```
int main()
{
    struct S s;
    s.func();
}
```

Transformation of member functions

- Source code

```
struct S
{
    ...
    void func(){}
};

int main()
{
    struct S s;
    s.func();
}
```

- Transformed code

```
struct S
{
    ...
};
void S::func(struct S* this){}

int main()
{
    struct S s;
    S::func(&s);
}
```

Overload an operator

```
struct S {
    struct S operator+(struct S rhs)
    {...}
};

int main()
{
    struct S s;
    struct S t;
    ...
    s + t; // S::operator+ is called
}
```

Transform the overloading function

- Source code

```
struct S
{
    S operator+(S)
    {...}
};

int main() {
    struct S s;
    struct S t;
    ...
    s + t;
}
```

- Transformed code

```
void S::operator+(
    struct S* ret_ptr,
    struct S* this
    struct S* rhs_ptr){
    struct S rhs = *rhs_ptr;
    ...
    *ret_ptr = ret_value;
}
int main(){
    struct S s; struct S t;
    ...
    struct S ret_value;
    S::operator+(&ret_value, &s, &t);
}
```

Generic programing

Example

```
template<typename T>
struct stack{
    T top();
    void push(T);
};
```

```
int main()
{
    stack<int> a;
    stack<string> b;
    a.push(42);
    b.push("42");
}
```

CFG for class templates

$$\begin{aligned} \text{template-definition} \rightarrow\ & \textbf{template} < \textbf{typename ID, ...}> \\ & \text{struct-definition} \end{aligned}$$

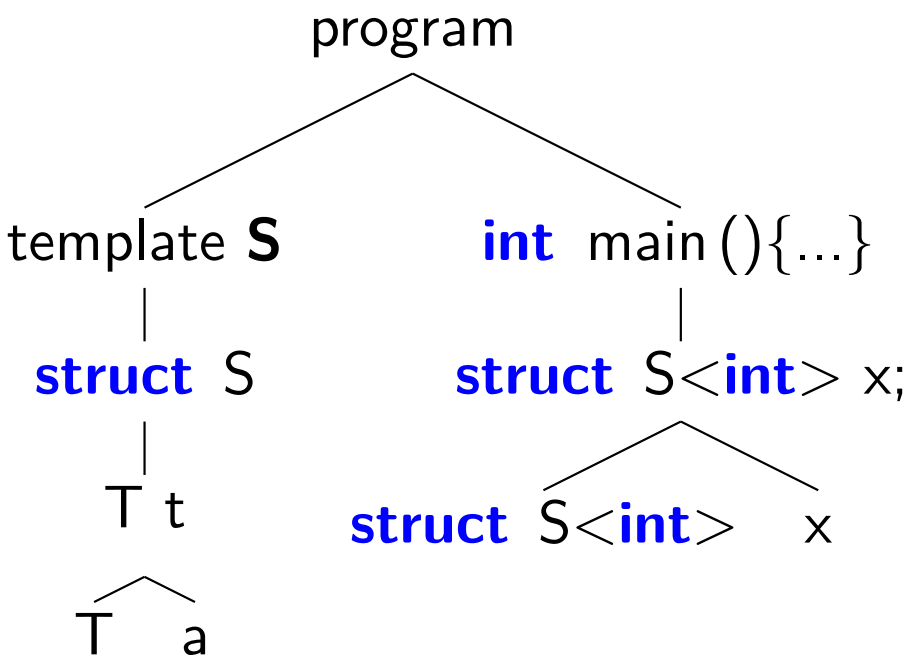
$$\begin{aligned} \text{type-specifier} \rightarrow\ & \cdots \\ \mid\ & \textbf{typename ID} \\ \mid\ & \textbf{struct ID} <\text{declaration,...}> \end{aligned}$$

Note

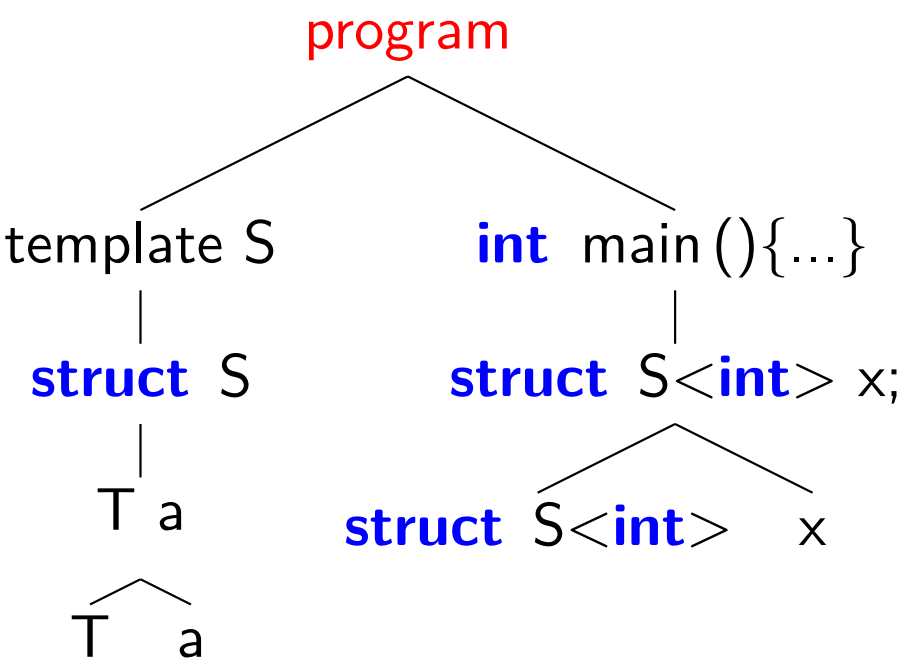
Due to the limitation of CFG, in the specification of a class template, the keyword *typename* is required before a template parameter.

Translation process of a class template

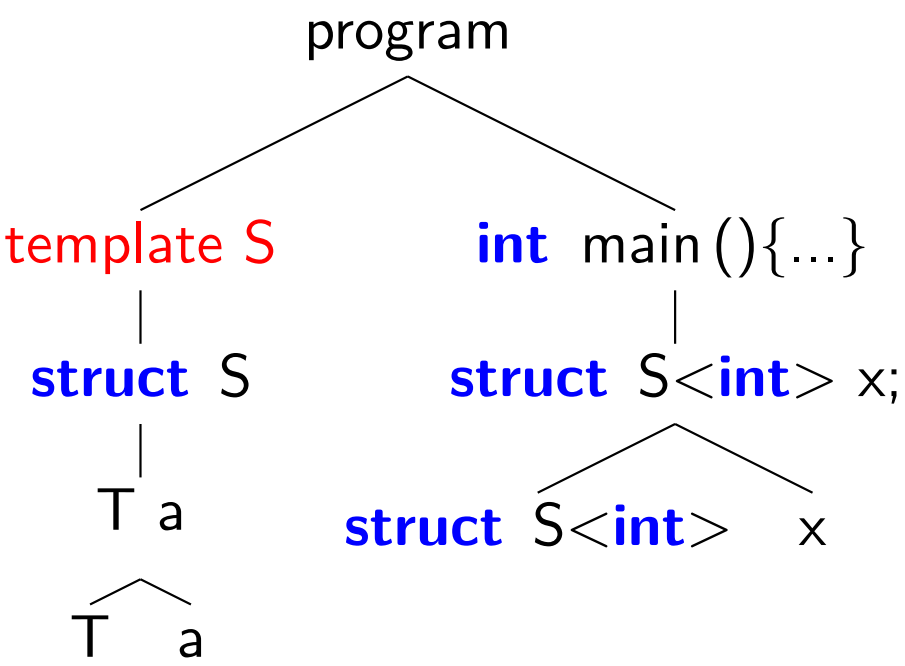
```
template<typename T>
struct S{
    typename T a;
};

int main(){
    struct S<int> x;
}
```



Translation process of a class template

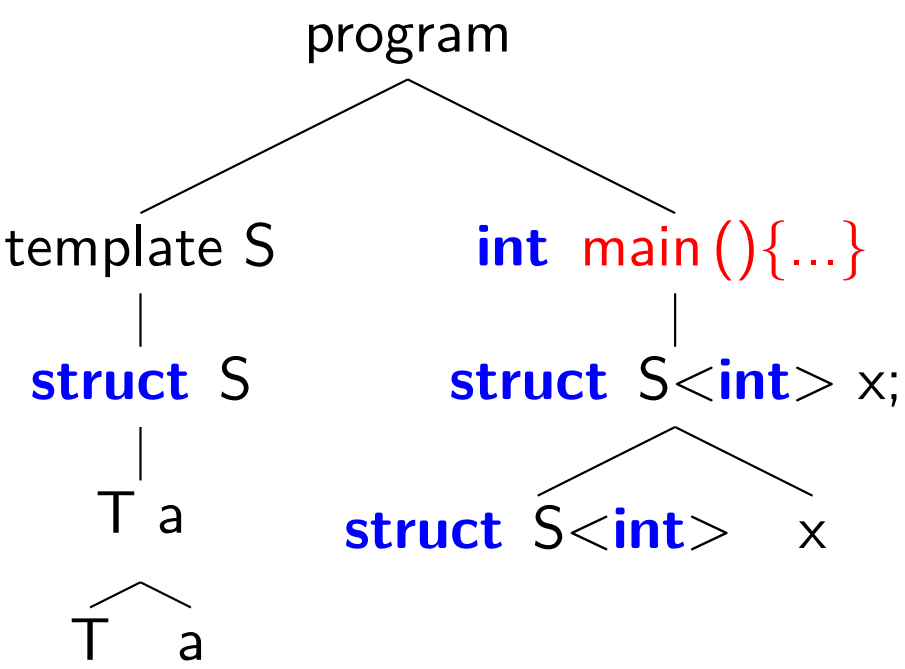


Translation process of a class template



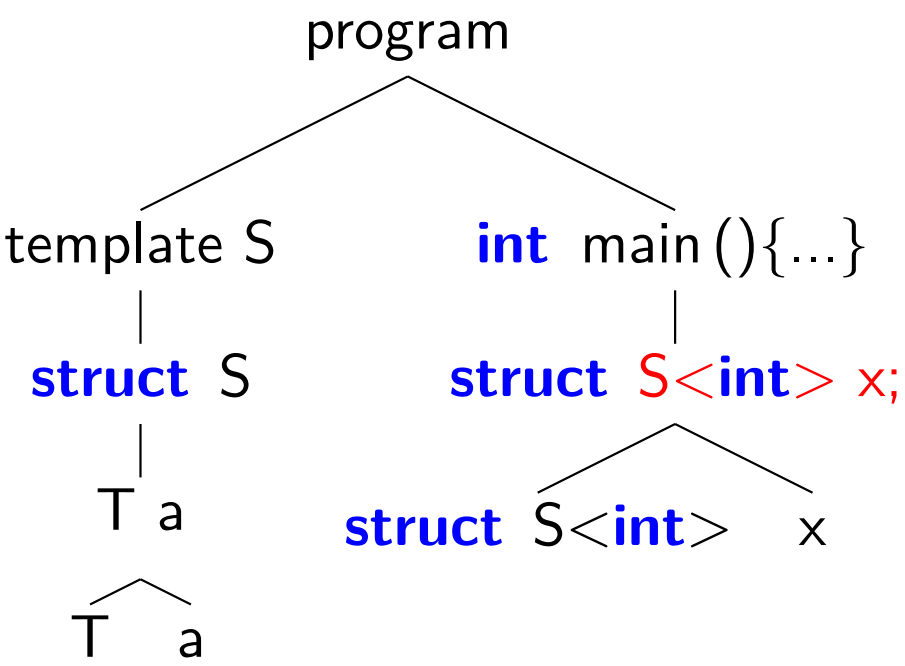
- **Record the pointer of node template S in AST, do not analyse subtrees.**

Translation process of a class template



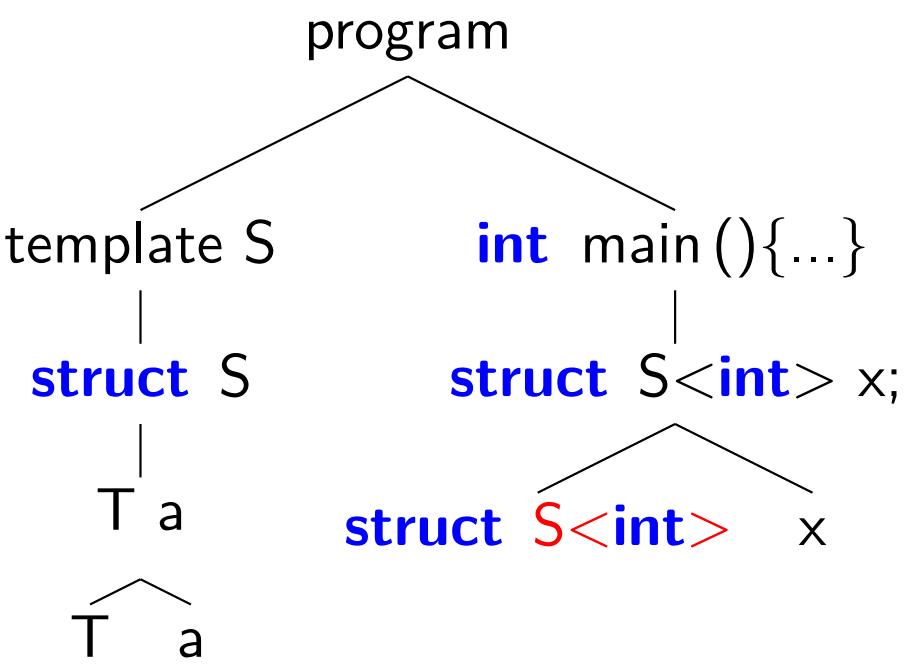
- Record the pointer of node template S in AST, do not analyse subtrees.

Translation process of a class template



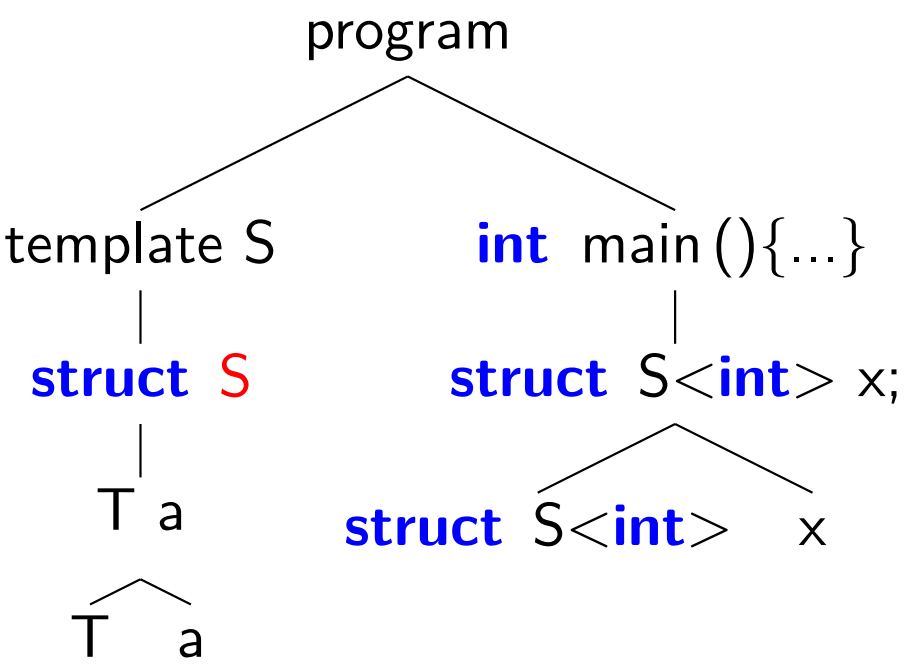
- Record the pointer of node template S in AST, do not analyse subtrees.

Translation process of a class template



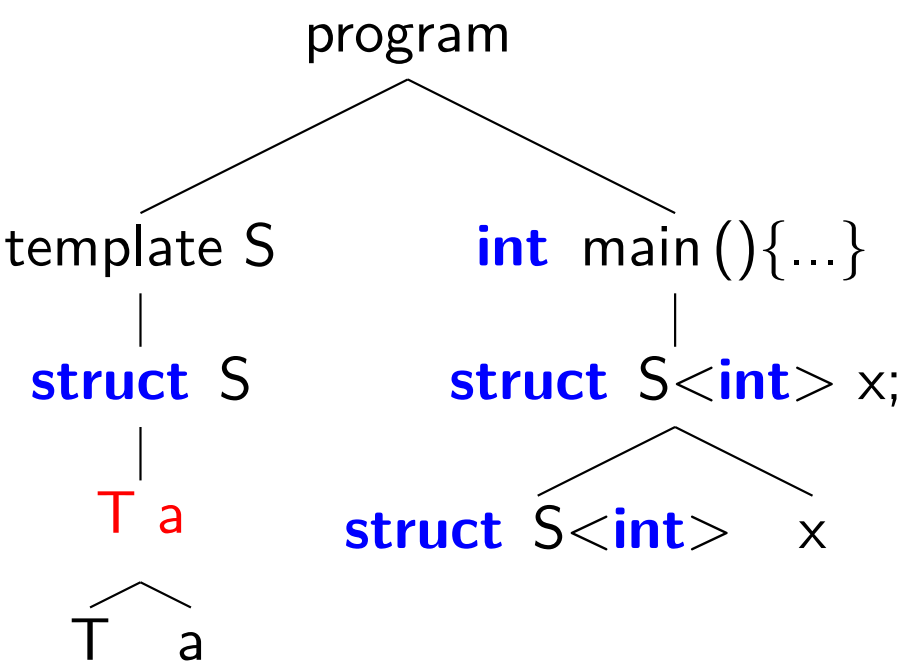
- Record the pointer of node template S in AST, do not analyse subtrees.
- **S<int> does not exist. Map parameter T to int, analyse class template**

Translation process of a class template



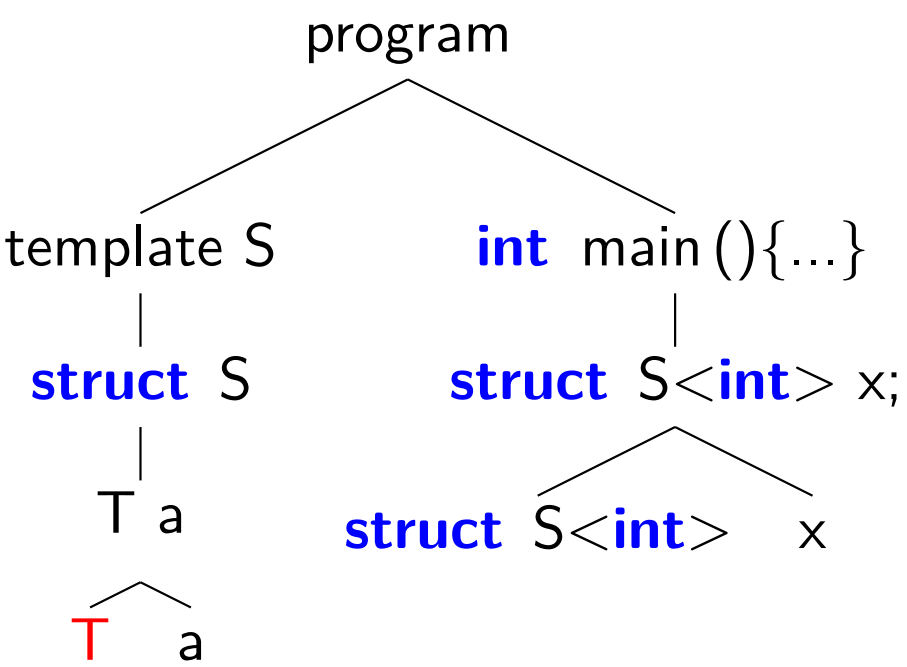
- Record the pointer of node template S in AST, do not analyse subtrees.
- S<**int**> does not exist. Map parameter T to **int**, analyse class template
- **Construct structure S<int>**

Translation process of a class template



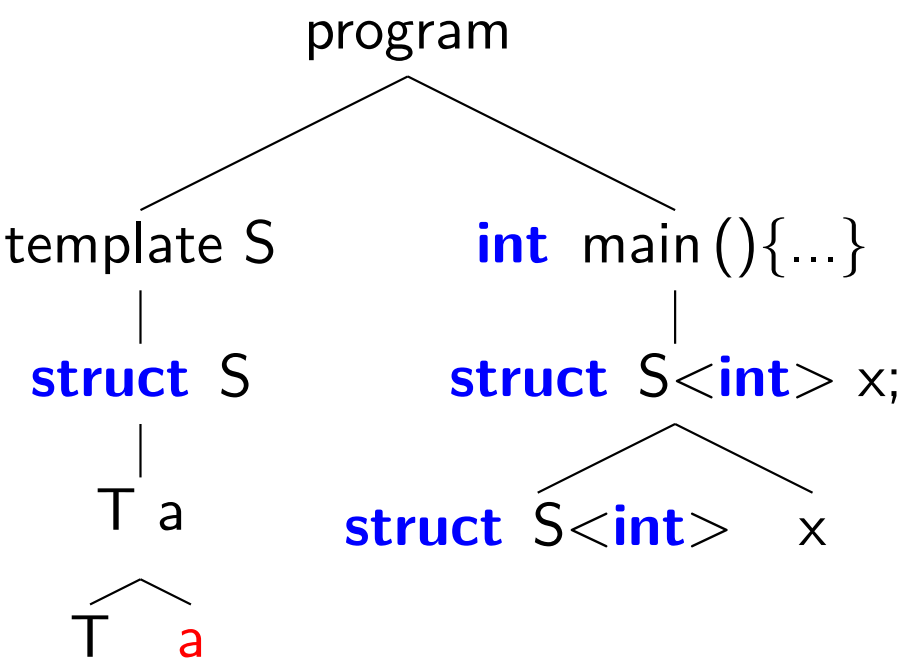
- Record the pointer of node template S in AST, do not analyse subtrees.
- S<**int**> does not exist. Map parameter T to **int**, analyse class template
- Construct structure S<**int**>

Translation process of a class template



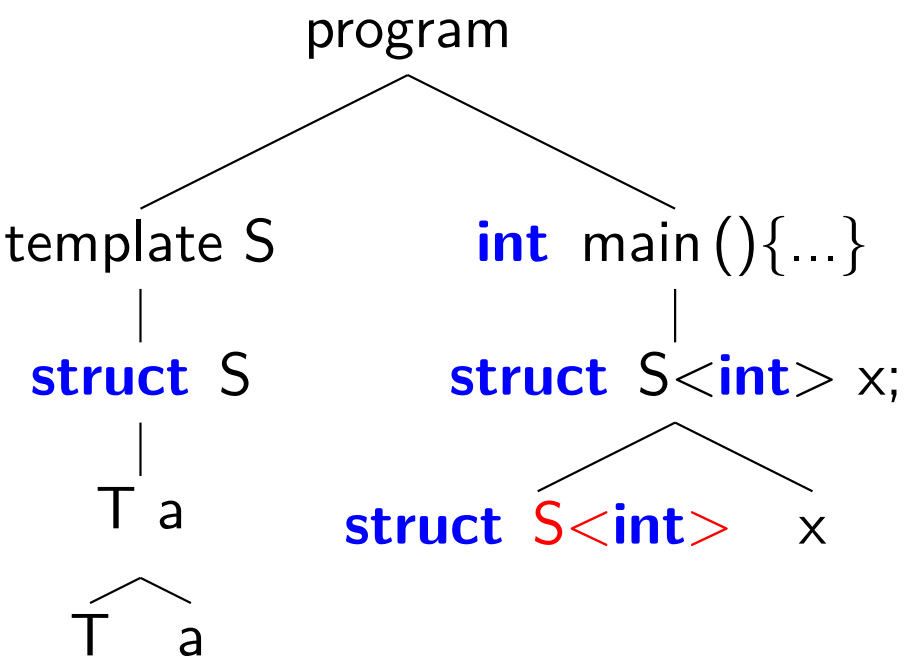
- Record the pointer of node template S in AST, do not analyse subtrees.
- S<**int**> does not exist. Map parameter T to **int**, analyse class template
- Construct structure S<**int**>
- **Use mapping to replace T with int**

Translation process of a class template



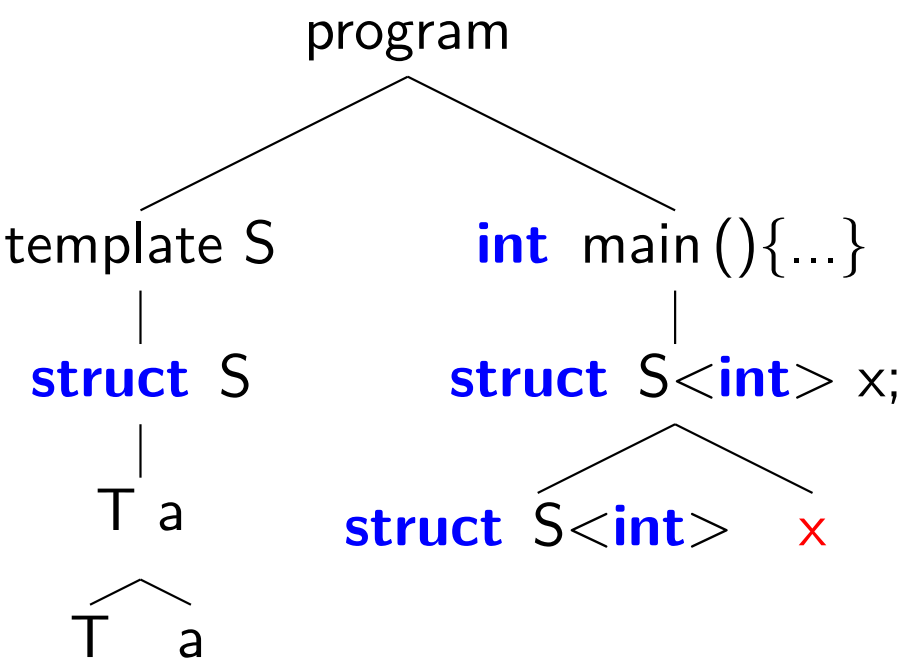
- Record the pointer of node template S in AST, do not analyse subtrees.
- S<**int**> does not exist. Map parameter T to **int**, analyse class template
- Construct structure S<**int**>
- Use mapping to replace T with **int**

Translation process of a class template



- Record the pointer of node template S in AST, do not analyse subtrees.
- S<**int**> does not exist. Map parameter T to **int**, analyse class template
- Construct structure S<**int**>
- Use mapping to replace T with **int**
- **S<int> now is a complete type**

Translation process of a class template



- Record the pointer of node template S in AST, do not analyse subtrees.
- S<**int**> does not exist. Map parameter T to **int**, analyse class template
- Construct structure S<**int**>
- Use mapping to replace T with **int**
- S<**int**> now is a complete type
- **Declare a variable of type S<int>**

Thank You.