Compilers Principles Lab5 Report

Language Feature Extension

Yurun Yuan

CS, USTC

January 13, 2022

Outline

1. Type System Extension
   - Pointers and Arrays
   - Structures
2. Operator Extension
   - Context Free Grammar of Expressions
   - Operations on Structures
3. Classes and Templates
   - Non-Static Member Functions
   - Operator Overloading
   - Class Templates

Outline

1. Type System Extension
   - Pointers and Arrays
   - Structures
2. Operator Extension
   - Context Free Grammar of Expressions
   - Operations on Structures
3. Classes and Templates
   - Non-Static Member Functions
   - Operator Overloading
   - Class Templates

Goals

- 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

Variable declaration	Type specifier	Declarator
int ∗ptr;	**int**	∗ptr
int array [42];	**int**	array [42]

Declaration Grammar

Declaration grammar

declaration → type-specifier declarator ;

For type specifiers, cminuf supports type **int**, **float** and **void**.

Type specifiers

type-specifier → **int** |**float**|**void**

Declaration Grammar

Declaration grammar

declaration → type-specifier declarator ;

For declarators, cminuf supports pointer declarators, function declarators and array declarators.

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

Crux: declarators can get mixed up

Example

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

Nested declarators	Type expression
(* a[2])(int)	**int**
* a[2]	**int**→**int**
a[2]	pointer(**int**→**int**)
a	array(2, pointer(**int**→**int**))

Trial 1: a naive solution

Example

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

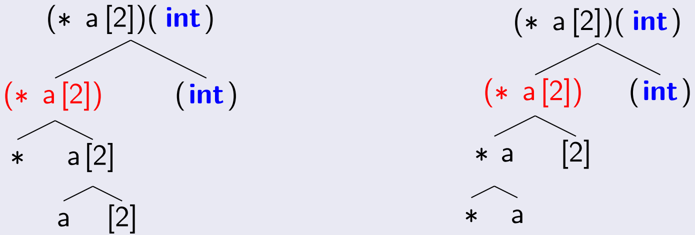
$$
\begin{array}{lcl}
\text{declarator} & \rightarrow & \text{* declarator} \\
 & | & \text{declarator (type, ...)} \\
 & | & \text{declarator [} \textbf{int} \text{]} \\
 & | & \text{(declarator)} \\
 & | & \textbf{ID}
\end{array}
$$

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

$$\begin{array}{lcl} \text{factor} & \rightarrow & \textbf{ID} \\ & | & (\ \text{decl}\ ) \end{array}$$

- Function calls and array subscripting

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

- Dereference

$$\begin{array}{lcl} \text{decl} & \rightarrow & \textbf{*}\ \text{decl} \\ & | & \text{decl-1} \end{array}$$

Trial 2: eliminating ambiguity

- Parentheses and identifiers

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

- Function calls and array subscripting

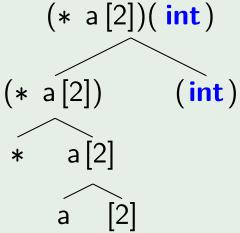
$$\begin{aligned} \text{decl-1} \quad &\rightarrow \quad \text{decl-1(type, ...)} \\ &\mid \quad \text{decl-1}[\textbf{int}] \\ &\mid \quad \text{factor} \end{aligned}$$

- Dereference

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

Example

int (∗ a [2])(**int**);



Goals

- 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

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

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

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

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

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.

Self reference

Example

A naive implementation of the node of a forwarding linked list

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

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

1. Type System Extension
   - Pointers and Arrays
   - Structures
2. Operator Extension
   - Context Free Grammar of Expressions
   - Operations on Structures
3. Classes and Templates
   - Non-Static Member Functions
   - Operator Overloading
   - Class Templates

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 ()

$$\begin{array}{lcl} \text{expr-0} & \rightarrow & \textbf{ID} \\ & | & \textbf{Integer-literal} \\ & | & \textbf{Float-literal} \\ & | & (\ \text{expr}\ ) \end{array}$$

1. Function call, array subscript, member access

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

Precedence and CFG

2. Dereference, address of

$$
\begin{array}{rcl}
\text{expr-2} & \rightarrow & \textbf{*}\ \text{expr-2} \\
 & | & \textbf{\&}\ \text{expr-2} \\
 & | & \text{expr-1}
\end{array}
$$

3. Multiplication, division

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

4. Addition subtraction

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

Precedence and CFG

5. Relational operations

$$\begin{array}{lcl} \text{expr-5} & \rightarrow & \text{expr-5 } \textbf{RelOp} \text{ expr-4} \\ & | & \text{expr-4} \end{array}$$

6. Assignment

$$\begin{array}{lcl} \text{expr} & \rightarrow & \text{expr-5} = \text{expr} \\ & | & \text{expr-5} \end{array}$$

Note

The assignment operator is right associated, so the production is expr → expr-5 = expr instead of expr → 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

1. Type System Extension
   - Pointers and Arrays
   - Structures
2. Operator Extension
   - Context Free Grammar of Expressions
   - Operations on Structures
3. Classes and Templates
   - Non-Static Member Functions
   - Operator Overloading
   - Class Templates

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);
}
```

Some details about my implementation

Function tables for operators

The supported operators are arithmatic operators $+, -, *, /$.
A function table for each of these operators is maintained, and is looked up every time the operation is performed.

Generic programing

Example

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

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

Note

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

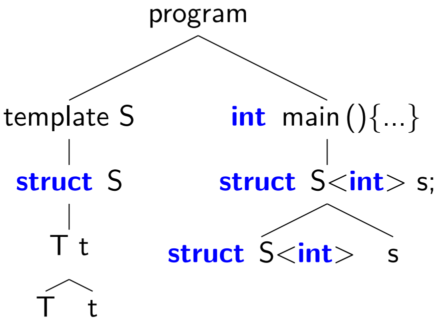
CFG for class templates

$$
\begin{aligned}
\text{template-definition} &\rightarrow \textbf{template} < \textbf{typename ID, ...} > \\
& \text{struct-definition} \\[1em]
\text{type-specifier} &\rightarrow \textbf{typename ID} \\
&\mid \textbf{struct ID} <\text{declaration,...}>
\end{aligned}
$$

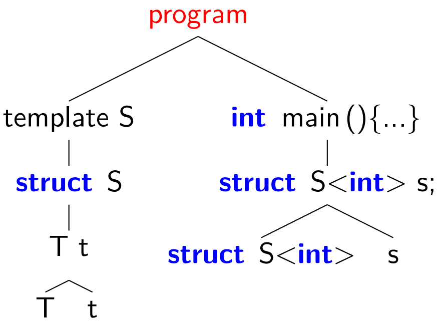
Translation process of a class template

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

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



Translation process of a class template



program

template S

int main (){...}

struct S

struct S<int> s;

T t

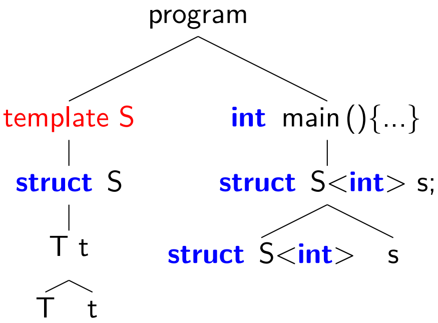
struct S<int>

s

T

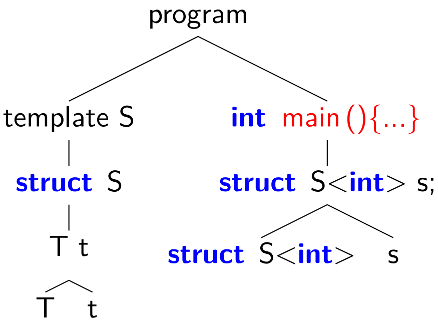
t

Translation process of a class template



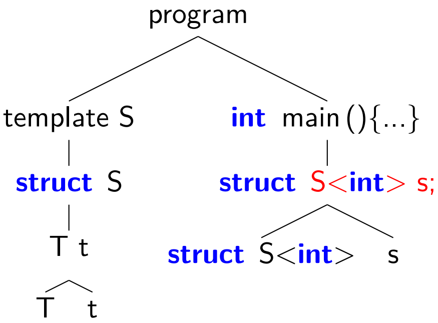
- **Record the pointer of node template S in AST, do not analyse subtrees.**

Translation process of a class template



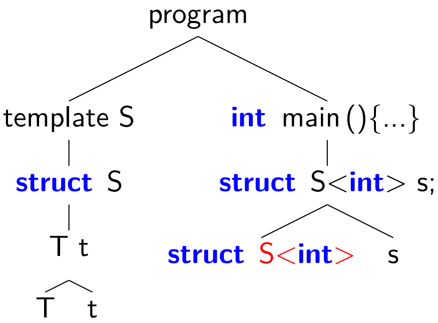
- Record the pointer of node template S in AST, do not analyse subtrees.

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



program

template S

struct S

T t

T t

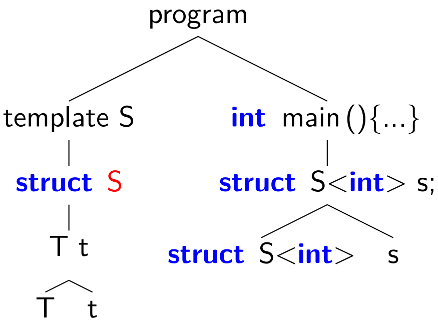
int main(){...}

struct S<int> s;

struct S<int> s

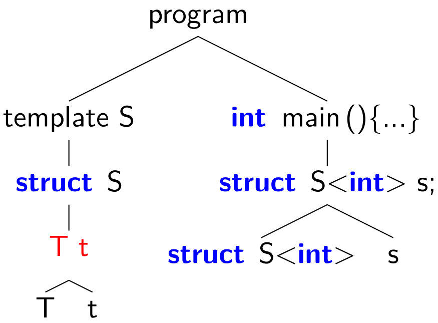
- Record the pointer of node template S in AST, do not analyse subtrees.
- **Map template T to int, go analyse class template**

Translation process of a class template



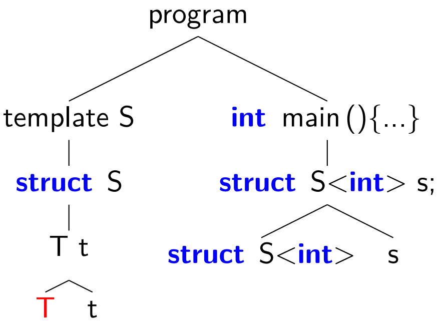
- Record the pointer of node template S in AST, do not analyse subtrees.
- Map template T to **int**, go analyse class template
- **Construct structure S<int>**

Translation process of a class template



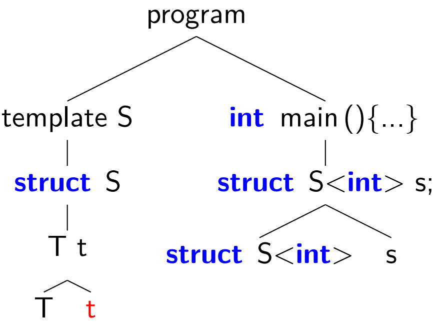
- Record the pointer of node template S in AST, do not analyse subtrees.
- Map template T to **int**, go analyse class template
- Construct structure S<**int**>

Translation process of a class template



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

Translation process of a class template



program

template S

struct S

T t

T t

int main (){...}

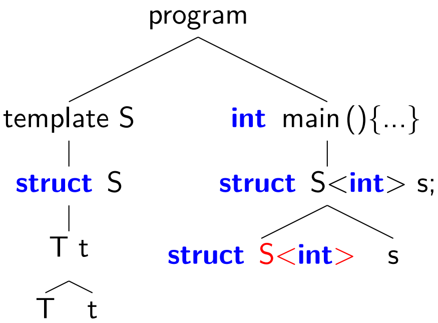
struct S<**int**> s;

struct S<**int**>

s

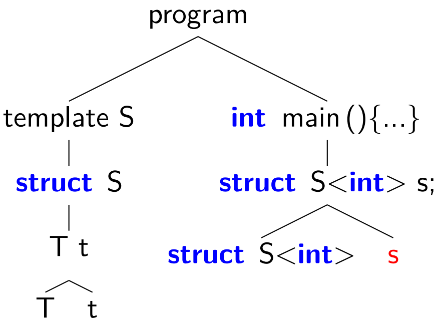
- Record the pointer of node template S in AST, do not analyse subtrees.
- Map template T to **int**, go analyse class template
- Construct structure S<**int**>
- Use mapping to replace T with **int**

Translation process of a class template



- Record the pointer of node template S in AST, do not analyse subtrees.
- Map template T to **int**, go 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.
- Map template T to **int**, go 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.