Compilers Priciple Lab5: ***Language Features Extension*** Report

BY 袁玉润 PB19111692

摘要

基于Lab1-Lab4，我在本次实验中增加了`cminusf`的语言特性，主要包括：

1. 类型拓展

   增加了丰富的指针、数组类型，支持结构体。

2. 运算符拓展

   除基本的算数运算和关系运算外，增加下标运算、调用运算、取地址运算、解引运算、成员访问运算。

3. 类

   支持部分类的特性，包括成员函数, 算数运算符重载和类模板。

在本报告中，我将展示上述内容的实现方法和实现效果。

1 新增语言特性及实现方法

1.1 类型拓展

1.1.1 指针与数组

定义. *Declaration grammar*

$$\textit{var-declaration} \ \rightarrow \ \textit{type-specifier declarator ;}$$

例. `int *a;`中 `int` 为 type-specifier, `*a` 为declarator.

1. type specifier

$$\text{type-specifier} \ \rightarrow \ \textbf{int} \mid \textbf{float}$$

2. declarator

   Declarator 分为以下3种：

   a. Pointer declarator

      `int *a;` 中 `*a` 即pointer declarator.

   b. Function declarator

      `int (*p)(int);` 中 `(*p)(int)` 即 function declarator.

   c. Array declarator

      `int a[32];` 中 `a[32]` 即 array declarator.

   各种declarator间可以嵌套，得到更加丰富的类型：

   Trial 1:

$$\begin{aligned}\text{declarator} \ \rightarrow \ & * \ \text{declarator} \\ \mid \ & \text{declarator (params)} \\ \mid \ & \text{declarator [} \textbf{int} \text{]} \\ \mid \ & \text{(declarator)}\end{aligned}$$

Problem: Ambiguity

```
int *ambiguous[42]
// Can be interpreted as either of the following
int (*ptr)[42];
int*(array[42]);
```

Solution: Precedence

根据*和[]的运算优先级修改CFG，消除二义性。

表格. Declarator CFG

Precedence	**Description**	**Context Free Grammar**
0		factor → **ID** | (declarator)
1	Function declarator Array declarator	declarator-1 → declarator-1 (params) | declarator-1 [**int**] | factor
2	Pointer declarator	declarator → * declarator | declarator-1

例. An array of function pointers.

```
int (* func_table[2])(int, int)
```

type-specifer	`int`
declarator	`(* func_table[2])(int, int)`

各层Declarators如下：

Declarators	**Type**
`(* func_table[2])(int, int)`	**int**
`(* func_table[2])`	**int**×**int**→**int**
`func_table[2]`	pointer (**int**×**int**→**int**)
`func_table`	array (2, pointer (**int**×**int**→**int**))

1.1.2 结构体

例. 结构体及结构体类型变量定义语法的多样性，使得结构体的实现看起来十分困难。

```
// Struct definitions with no variables declared
struct S
{
    ...
};

// Struct definitions with a variable declared
struct T
{
    ...
} t;

// Define a variable with struct type
struct S s;

// Anonymous structs
struct
{
```

```
    ...
} anonymous;

// Nested structs
struct U
{
    struct Nested
    {
        ...
    } n;
};
```

根据观察和调研，可知：

Struct definitions *are* type specifiers.

declarators may be omitted if the type-specifier is a struct definition.

例.

Description	整数	结构体定义+变量声明	结构体定义	变量声明
Declaration	`int a;`	`struct S{...} s;`	`struct S{...};`	`struct S s;`
Type specifier	`int`	`struct S{...}`	`struct S{...}`	`struct S`
Declarator	`a`	`s`	*empty*	`s`

修改文法如下：

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

1.2 运算拓展

支持如下运算：

1. Arithmatic operations
2. Relational operations
3. Assignment
4. Array subscript

   `array[subscript]`
5. Pointer dereference

   `*ptr`
6. Address of

   `&rval`
7. Member access

   `inst.mem`

8. Function call

   `callable(params)`

1.2.1 表达式CFG

根据运算优先级消除二义性，得到以下文法：

表格. Expression CFG

Precedence	Operator	Description	Context Free Grammar
0			expr-0 $\rightarrow$ **ID** | **Integer-literal** | **Float-literal** | (expr)
1	() [] .	Function call Array subscripting Member access	expr-1 $\rightarrow$ expr-1 (args) | expr-1 [expr] | expr-1 **. ID** | expr-0
2	$*$ &	Dereference Address of	expr-2 $\rightarrow$ ***** expr-2 | **&** expr-2 | expr-1
3	$*$ /	Multiplication, division	expr-3 $\rightarrow$ expr-3 **MulOp** expr-2 | expr-2
4	$+$ $-$	Addition, subtraction	expr-4 $\rightarrow$ expr-4 **AddOp** expr-3 | expr-3
5	$>$ $\geqslant$ $==$ $\neq$ $<$ $\leqslant$	Relational operators	expr-5 $\rightarrow$ expr-5 **RelOp** expr-4 | expr-4
6	$=$	Assignment	expr $\rightarrow$ expr-5 **Assign** expr | expr-5

1.2.2 结构体相关的运算

支持结构体这样的复合类型后，一些对scalar type的运算需要拓展，以支持结构体类型的运算元。最显著的变化便是，由于许多结构体类型的变量无法储存在单一寄存器中，结构体的赋值和传递需要特别的操作。

Assignment to struct-type variables 通过对成员逐个赋值的方式实现了结构体变量的默认赋值操作。

例. 考虑如下定义的结构体S:

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

对`struct S`类型的变量s的赋值操作将会被翻译为：

cminusf	Translated representation
`s=t`	`s.member1 = t.member1;` `s.member2 = t.member2;`

Struct-type parameters 当函数调用时参数为结构体类型时，传递结构体变量的指针，并由callee复制指针指向的结构体变量。

例. 考虑如下情形

<table>
<tr><th>cminusf</th><th>Translated representation</th></tr>
<tr><td>void func(struct S s)
{
    . . .
}

int main()
{
   func(t);
   . . .
}</td><td>void func(struct S *ptr)
{
    struct S s;
    s = *ptr;
    . . .
}

int main()
{
    func(&t);
}</td></tr>
</table>

Struct-type return value 当函数返回值为结构体类型时，由caller开辟返回值存放空间，并将指向这块空间的指针作为参数传递至callee

例. 考虑如下情形

<table>
<tr><th>cminusf</th><th>Translated representation</th></tr>
<tr><td>struct S func()
{
    . . .
    return s;
}

int main()
{
   func();
}</td><td>void func(struct S* ret_ptr)
{
    . . .
    *ret_ptr = ret_value;
}

int main()
{
    struct S ret_value;
    func(&ret_value);
}</td></tr>
</table>

1.3 部分类的特性

1.3.1 Member functions (Non-static)

在参数列表首位添加`this`指针参数，将成员函数翻译为普通函数。

例.

<table>
<tr><th>cminusf</th><th>Translated representation</th></tr>
<tr><td>struct S
{
    ...
    void member_func(){}
};

int main()
{
    struct S s;
    s.member_func();
}</td><td>struct S
{
    ...
};

void S.member_func(struct S* this){}

int main()
{
    struct S s;
    S.member_func(&s);
}</td></tr>
</table>

Problem: 若按顺序对结构体定义中的成员变量和成员函数的定义遍历，则在某函数定义后的成员变量对成员函数不可见。

例.

```
struct S
{
    void mem_func(); // 此处并不知道struct S有成员a
    int a;
}
```

Solution: 首先遍历结构体中所有定义，将成员函数与成员变量分离开，在得到完整的结构体成员变量定义后再处理成员函数。

1.3.2 Arithmetic Operator Overloading

识别关键字`operator`，将重载操作的函数记录于表中，在需要时调用此函数。

例.

<table>
<tr><th>cminusf</th><th>Translated representation</th></tr>
<tr><td>
<pre>
struct S
{
    ...
    // Overload operator ‘+’
    struct S operator+(struct S rhs)
    {...}
};

int main()
{
    struct S s;
    struct S t;
    ...
    s + t; // Call ‘S::operator+’
}
</pre>
</td><td>
<pre>
struct S{...};

void S.operator+(
    struct S* ret_ptr,
    struct S* this
    struct S* rhs_ptr)
{
    struct S rhs;
    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);
}
</pre>
</td></tr>
</table>

1.3.3 Class Templates

注记. 由于CFG的限制，我的实现中的语法与C++中并不完全兼容，具体表现在：在类模板中使用模板参时，需要在参数名前使用`typename`关键字。具体可见4.1。

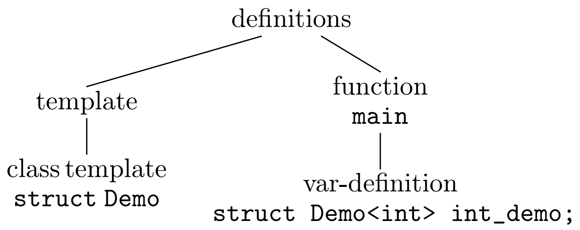
Overview

实现类模板的大体思路是：当访问到AST中类模板定义的节点时，记录此节点的地址。后续过程中如需对模板实例化，则使用类模板参数重新访问类模板对应节点，并构建相应结构体、成员函数。

例.

```
template<typename T>
struct Demo{
    ...
};

int main(){
    struct Demo<int> int_demo;
}
```



在访问节点template时，并不访问节点 class template `struct Demo` ，而是只记录下此节点的位置。

在访问 var-definition `struct Demo<int> int_demo;` 时，使用模板参`int`访问节点 class template `struct Demo` ，此过程中会产生`struct Demo<int>`的实例。

Modification of CFG

class-template-declaration	→	**template** ⟨**typename ID, typename ID...**⟩
type-specifier	→	**int** | **float**
	|	struct-definition
	|	**struct ID**
	|	class-template-type
	|	**typename ID**
class-template-type	→	**struct ID** ⟨type-specifier,type-specifier...⟩

Modification of AST visitors

类型等价判断. 当首次遇到`struct Demo<int>`时， 编译器会生成结构体`struct.Demo.int`. 在之后的过程中遇到`struct Demo<int>`时，编译器应当识别出此类型，而非构建新的结构体。

在我的实现中，使用类模板名和模板参数的值为键，记录已生成的结构体。

2 Demos

The source code can be found at `CMINUS/Reports/5-bonus/demos/complex_num_cal.cminus`.

I wrote a complex nubmer calculator to exhibit all the extended language features mentioned above. It may seem a little verbose because I am trying to exert all the features my compiler supports. The program has following components:

Struct `complext_num`. A struct representing a complex number, which simply contains 2 `float`s as real component and imaginary component.

This struct has several member functions and overloads 4 arithmetic operators, that is, addition, subtraction, multiplication and division.

The prototype of the struct is shown as follows (not in `cminusf` syntax):

```
struct complex_num{
    float real, imaginary;
    complex_num operator+(complex_num);
    complex_num operator-(complex_num);
    complex_num operator*(complex_num);
    complex_num operator/(complex_num);
};
```

Functions performing arithmetic operations for complex number. 4 functions that take 2 complex numbers's addresses as parameters and calculate plus(function `plus`), subtraction(function `subtract`), multiplication(function `multiply`) and division(function `divide`) respectively. For example, the function `plus` is defined as:

```
struct complex_num plus(struct complex_num *lhs, struct complex_num *rhs)
{
    return *lhs + *rhs;
}
```

An array `function_table`. An array containing all 4 functions mentioned above, i.e., `function_table = {plus,subtract,multiply,divide}`.

Function `get_function`. It takes an operator as input and returns a function pointer in the function table that performs the corresponding operation. For example, `get_function('+')` woud returns the first element in the `function_table`, i.e. the function pointer `plus`.

A class template `stack`. The template has one template parameter `T`, which is the type of the element in the stack. The template has several member functions relevent to stack data structure. The compact prototype is as follows:

```
template<typename T>
struct stack{
    T *memory;  // To store the elements in stack
    int
    void init();// Constructor
    void drop();// Destructor

    void push(T elem);
    T top();
    T pop();
};
```

The `main` function. To implement the calculator, I used 2 stacks:

1. one for the operands, whose type is `struct stack<struct complex_num>`, and
2. one for the operators, whose type is `struct stack<int>`, containing operators like `+`,`-`, `*`, `/`, `(` and `)`.

To parse an algebraic expression like `1i * 2 * (3 + 4i)`, the program would push the operands and operators into stack repectively, and do the calculation at the right time.

3 实现时的阻碍

实现上述特性时遇到许多耗时耗力的工作，主要的困难来自以下几点：

1. Lab1-Lab3中代码的大规模重新设计及重写

   因拓展点较多，原先的结构无法支持增加的语言特性，故对词法分析、语法分析、AST的设计及构建、IR的翻译部分均有较大的删改。

2. LightIR对一些特性支持的缺失

   需自行修改LightIR使之支持结构体类型。

   此外，LightIR中只能对函数实施调用操作，但在IR中允许对函数指针实施调用。

4 Unsolved Problems

4.1 Omitting Keyword **struct** when Refering to a Struct Type

在C++中，指代类类型时可忽略`struct`或`class`关键字，如

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

但使用CFG似乎很难区分开以下两种情况：

变量声明	函数调用
`S (s);`	`func(s);`

因此在我的实现中要求类型名前使用`struct`关键字。

同样的问题也出现在类模板内。在C++中使用类模板参数时可直接以参数名表示，例如：

```
template<typename T>
class Demo{
    T mem;
};
```

在我的实现中，基于上述原因，要求用户在参数名前使用`typename`关键字，如：

```
template<typename T>
class Demo{
    // 'typename' is compulsory in my cminusf
    typename T mem;
};
```

4.2 Recursive Type

链表节点的常见实现如下：

```
struct Node{
    T elem;
    Node* next;
};
```

在Node节点未完整定义(但已声明)的情况下，可使用`Node*`类型。

我的实现中未考虑此情况。