Compilers Priciple Lab5: *Language Features* 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，包括成员函数和算数运算符重载。

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：

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

Problem: Ambiguity

```
int *ambiguous[42]

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])`	A function who takes 2 `int`s as parameters and returns an `int`
`func_table[2]`	A function pointer
`func_table`	An array of function pointers

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：

type-specifier → **int** | **float**
| struct-definition
struct-definition → **struct ID** { definitions }
| **struct** { definitions }
| **struct ID**
declarator → ...
| ε

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	Context Free Grammar
0		expression-0 → **ID** | **Integer-literal** | **Float-literal** | (expression)
1	Function call Array subscripting Member access	expression-1 → expression-1 (args) | expression-1 [expression] | expression-1 **. ID** | expression-0
2	Dereference Address of	expression-2 → ***** expression-2 | **&** expression-2 | expression-1
3	Multiplication, division	expression-3 → expression-3 **MulOp** expression-2 | expression-2
4	Addition, subtraction	expression-4 → expression-4 **AddOp** expression-3 | expression-3
5	Relational operators	expression-5 → expression-5 **RelOp** expression-4 | expression-4
6	Assignment	expression → expression-5 **Assign** expression | expression-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 represent
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

cminusf	Translated represent

void func(struct S s)
{
. . .
}
 	
int main()
{
func(t);
. . .
}
	void func(struct S *ptr)
	{
	struct S s;
	s = *ptr;
	. . .
	}
	
	int main()
	{
	func(&t);
	}

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

cminusf	Translated represent

struct S func()
{
. . .
return s;
}
 	
int main()
{
func();
}
	void func(struct S* ret_ptr)
	{
	. . .
	*ret_ptr = ret_value;
	}
	
	int main()
	{
	struct S ret_value;
	func(&ret_value);
	}

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.

cminusf	Translated represent

struct S
{
. . .
void member_func(){}
};
 	
int main()
{
struct S s;
s.member_func();
}
	struct S
	{
	. . .
	};
	
	void S.member_func(struct S* this){}
	
	int main()
	{
	struct S s;
	S.member_func(&s);
	}

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.

cminusf	Translated represent

struct S
{
...
// Overload operator '+'
struct S operator+(struct S rhs)
{...}
};
 	
int main()
{
struct S s;
struct S t;
...
s + t; // Call 'S::operator+'
}
	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);
	}

2 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。

3 Demos

I wrote a complex nubmer calculator to exhibit all the extended language features mentioned above. 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.

Functions that perform arithmetic operations for complex number. 4 functions that take 2 complex numbers as parameters and calculate plus, subtraction, mulpication and division respectively.

A function table containing arithmetic calculation functions. An array containing all 4 functions mentioned above.

`main` function. Takes care of IO operations and invokes other functions to do the calculation.

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);`