C/0 Language Design

Na Luo

ln931022@mail.ustc.edu.cn

Outline

C/0 is a subset of Programming Language C. In our specific design and implementation, we trying to utilize the codes in PL/0 to present a clear, simple, and representative language to demonstrate the basic concept and implementation of Parser, Symbol Table, and Intermediate representation.

The assignment P3.3.B,C is also include in this document as part of the "Code Generation"

Language Features in C/0

- Type
    - Only 32-bit Integer is support. (in keyword "int")
- Function
    - Function without parameters and return value
- No function declaration
    - Recursive function calls
- Variable and Constant Declaration
    - Declaration in Global scope
    - Declaration in Local Scope

      *Definition Statement is a legal statement may appears in any position of the program
- Statement
    - While Statement
    - If … else Statement
    - Compound Statement
    - Assignment Statement
    - Function Call Statement
    - Object Declaration Statement
- Operators and Expression
    - Basic Arithmetic Operators: + - * /
    - Parentheses: ()
    - Relation Operators: > < == != <= >=

- *Boolean expression can only be used in if() or while() condition

Lexicon

We will first present the lexical design of C/0

Word	Definition
whitespace	(/t | ' ')*
digit	0-9
number	(digit)+ | 0[x|X](digit)+ | 0[0-7]+
alpha	A-Za-z
keyword	while | if | int | else | const | void | float
identifier	(alpha | _)(number | alpha | _)*
operator	+ | - | * | / | > | < | = | <= | >= | != | == | "||" | && | | | & | << | >>
LBracket	[
RBracket	]
LParenthese	(
RParenthese	)
LBrace	{
RBrace	}
Coma	,
Semicolon	;
comment	/* [[^*]|([*]+ [^*/])]* [*]* */

The lexicon definition is in C-style, since it simply replace the corresponding words.

Grammar

Here we do some change to the existed PL/0 grammer to make it into C-style.

The most significant change is that we don't allow "Block" Nesting anymore , as well as bring variable and constant declaration as a formal statement.

Nonterminal	Definition

Program	Block MainDef
Block	Block Decls
Decls	Vdecl | Cdecl | FunctionDef
Vdecl	int ident {, ident} ;
Cdecl	const int ident = number {, ident = number} ;
FunctionDef	void ident () CompoundStatment
MainDef	Void main() CompoundStatment
Stat	Assign | Condition | Wlop | CompoundStatment | FunctionCall | Vdecl | Cdecl | **Functioncall**
FunctionCall	ident ();
CompoundStatment	'{' {Stat} '}'
Wlop	while (Relation) Stat
Condition	if (Relation) Stat [else Stat]
Assign	ident = Expr ;
Relation	Expr Relopr Expr
Relopr	>|<|==|!=|>=|<=
Expr	Term {Lopr Term}
Term	Factor {Hopr Factor}
Factor	ident | number | (Expr)
Lopr	+|-
Hopr	*|/

Symbol Lookup

In PL/0 , The symbol lookup is divide into two function : position(id) and table(index) , the symbol table is organized into a stack structure , where the most recent appeared symbol is on the top part of stack. The lookup algorithm will always look for the latest decelerated symbol from the top of stack to the bottom of stack. The symbol insertion should first try to find if there is a symbol shares the same id with the insertion command in the same scope ("Block" in PL/0), and then push the symbol in the top of stack. (But the PL0 implementation actually didn't perform the check)The symbol delete is processed in the end edge of a scope ("Block"). Directly modify the stack top position to where before enter the scope to remove the "upper" symbols.

In our C/0 design, we will inherit this symbol lookup mechanism it works great in C/0 concept.

Code Generation

Since the statement design in PL/0 and C/0 is functional similar , we direct inherit it's code gen method here :

All the variable is running and processing in a "Execution Stack", each variable maps to a address in the stack.

Gen (instruction, operand A, operand B) will generate the code into the code buffer (in Pl0.c, is code [], which buffer position indicated by global variable cx)

And for each kind of statement , it generate specific codes:

Variable declaration

Actually this is done in the "Block" processing

It will generate IR "Int 0 dx" where dx = 3 + (number of variable in current block)

The constant 3 is reserved for procedure calling information

This revealed the proper IR generate for variable declaration is

```
"Int 0 1"
```

work perfect with such statement like : Int a;

Constant declaration and immediate number

Is pretty similar to variable , but with a initial value , so it is

```
"Lit 0 val"
```

Operators and Expression

First gen the code of it's operands , then generate :

```
"Opr 0 op"
```

The operand is implicit defined as the top two number on the top of the stack.

Variable reference

This code is generated for variable reference

```
"Lod LevelDistance VariableRelativeAddress"
```

LevelDistance is represented by "lev – var.level" , where the global "lev" shows the current Depth of procedure nesting. And the var.level store the variable 's definition 's depth.

The absolute address is computed by the runtime with such equation:

Variable Absolute Address = Level Base Address + Variable Relative Address

In C0 , since you cannot define nested procedure or function , so the callee function cannot access the variable defined in caller function , the variable address can be simplified into two mode : "Local or Global" where the LevelDistance in gen function should be 0 or level.

```
"Lod 0 RelativeAddress" for local variable
"Lod lev RelativeAddress" for global variable
```

Assignment

The instruction generated for assignment is similar to reference , the only difference is the IR changed from "Lod" to "Sto"

```
"Sto 0 RelativeAddress" for local variable
"Sto level RelativeAddress" for global variable
```

If … else … Statement

The code block below is generated for if (Condition) then Action Statement:

Code Number	Instruction
…Cx0-1	Code(Condition)
Cx0	Jpc 0 Cx1
Cx0…Cx1-1	Code(Action)
Cx1	Code(Next)

Similar structure could be use there to add the "else Action" support

Code Number	Instruction
…Cx0-1	Code(Condition)
Cx0	Jpc 0 Cx1+1
Cx0…Cx1-1	Code(TrueAction)
Cx1	Jpc 0 Cx2
Cx1+1…Cx2-1	Code(FalseAction)
Cx2	Code(Next)

While Statement

While statement could direct use:

Code Number	Instruction
Cx0…Cx1-1	Code(Condition)
Cx1	Jpc 0 Cx2
Cx1+1…Cx2-1	Code(Action)
Cx2	Jpc 0 Cx0

Function Call Statement

This code is generated for pl0 in similar fashion with variable:

```
"Cal LevelDistance ProcedureEntryAddress"
```

Since the nested function is not allowed in C/0 , all function object's "level" must be 0, as a "global function" , so the code can be change to :

```
"Cal lev FunctionEntryAddress"
```