CSE-504 : Compiler Design

E-- Compiler Design Document

16 - Apr - 2014

Sahil Parmar (109242129)

Tejal Kaple (109294247)

Sohil Gandhi (109395892)

Aniket Alshi (109114640)

I. Introduction

Most of the programming is done in high level language, while on the other hand the processor understands only machine level language. To bridge this gap we use compilers which translates high level language into machine level language. In the process of translation the compiler also checks for errors committed by users during programming. So far in our assignments we have implemented the first three phases of compilers, as follows:

Lexical Analysis: In this phase we read the input text in form of regular expressions and categorize each pattern into separate tokens. Few of these tokens such as TOK_UINTNUM, TOK_STRCONST, etc. are assigned with their respective literal values. All the tokens defined in E--_lexer.tab.h are used in writing next phase of syntax analysis.

Syntax Analysis: The tokens categorized in the lexical analysis are further processed to form the syntax rules of E-- language. Error handling is also implemented for the incorrect syntax encountered in the input program.

Abstract Syntax Tree: An abstracted representation of the input program i.e. an Abstract Syntax Tree is generated, which mainly deals with relevant information required for the next phases of compilation. Redundant information is eliminated and few semantic errors such as duplicate symbol declaration are detected.

The detailed description and implementation of the next phases of compiler are given in the following sections.

II. Type Checking

Type Checking by definition implies type verification of variables and expressions to determine whether they are compatible. To achieve this objective, we first determine types of the expressions and then check whether these types are compatible with the operators or functions it is used with.

We have implemented type checking with the help of “typeCheck()” and “typeCheckST()” functions. typeCheckST() function is used to iterate over all the declarations, i.e. global, local, param variables and function definitions, and call their subsequent typeCheck() functions. typeCheck() is a virtual function which is implemented for each subclass of AstNode and SymTabEntry. This function is used to propagate the types from classes RefExprNode, ValueNode to other classes such as VariableEntry and OpNode, where the type compatibility is determined. The typeCheck() function also prints the errors by calling errMsg() function, which is a result of incompatible types. In the process of type checking, conditions may arise such that the operands of an operator do not belong to the same type. For this we perform coercion and update the “coercedType” of the resultant operand based on the subtype of the operands.

Eg:
```
double d1, d2;
int x;
d1 = x*d2;        // Here coerced type of x i.e. coercedType of ExprNode is updated to double
```

To achieve this we first check whether one operand is a subtype of another or not. For this we have implemented “isSubType()” function in Type.C which determines whether operand1 is a subtype of operand2 or vice-a-versa. Coercion is also performed when the actual parameters of the function are subtype of the formal parameters.

III. Memory Allocation

Memory allocation is the next compiler pass over the semantically correct AST. There are two types of memory allocation: Static and Dynamic Memory Allocation. Static allocation for global, static and local variable are performed at compile time.

a. Global / Static Variable Allocation:

In static or compile time memory allocation, compiler assigns memory to global variables by knowing the exact size and type **(VarKind = GLOBAL_VAR)** of variables at compile time. These global variables are allocated memory in the .DATA section at compile time. All reference to global variables are provided with references to this space allocated from .DATA section. All addresses of global variables can be stored as an offset in the **"offset"** attribute of each **SymTabEntry** object for a global variable. This offset would be from the start of the .DATA section. In general, this runtime address of the .DATA section is defined by the linker.

In the project, all memory allocations are handled by the memAlloc() function call. We directly provide the address of the .DATA section i.e. address 0x100. Hence the first global variable will be at location 0x100. Next variable will be allocated memory (0x100 + size of first variable).

Eg: int i; double j;

In the below diagram we can see that i is allocated memory from 0x100 to 0x103 in the .DATA section as i is an int of 4 bytes. Also j is allocated memory from 0x104 to 0x107 at compile time. Hence any reference to global variables i and j are indicated by references to addresses 0x100 and 0x104 respectively.

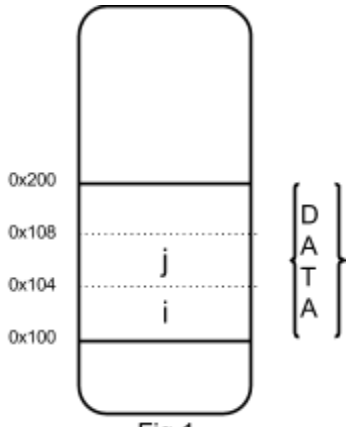

Fig 1

b. Local Variable Allocation:

All local variable allocations for functions are assigned on the stack at offset's from RSP. Hence the allocation of local variables **(VarKind = LOCAL_VAR)** will be stored as offset from RSP in the **"offset"** attribute of the **SymTabEntry** object.

Eg:

```
int foo()
{
        char k;
        double j;
        int i;
        .......
}
```

In Fig 2, offset attributes of i, j and k are stored as 0, 4, 8 respectively. This means that j is at offset 4 from RSP in the activation record for function foo().

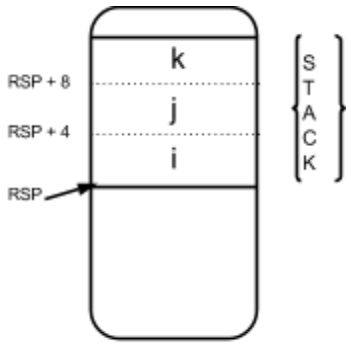

Fig 2

IV. Code Generation

Intermediate code Generation and Final Code Generation involves developing machine independent assembly code. Our compiler will be modeled after a very simple Instruction Set.

iCode generated in iCode file is stored in a word addressed memory starting at address zero. All the assembly code is generated by the **codeGen()** function.

We intend to use Quadruples form of three address intermediate code generation .

a. Statements :

```
stmt --> S1; S2;                          {   stmt.code = S1.code || S2.code;                    }
if_then_else --> if E1 then S1 else S2    {   elselabel = newlabel();
                                              endlabel = newlabel();
                                              if_then_else.code = E1.code || codeGen ( cmp E1.temp, 1)
                                                        || codeGen ( jne elselabel ) || S1.code ||
                                                        codeGen ( jmp endlabel ) || codeGen (elselabel :)
                                                        ||
                                                        S2.code || codeGen (endlabel :)
                                          }
```

User Code	**Intermediate Assembly Code**

int a = 2;
if (a < 0) then
  a = a + 1;
else
  a = a - 1;
	begin:
	MOVI 0x100 R000
	STI 2 R000 // store global variable a at address 0x100
	LDI R000 R010 // load global variable again
	JMPC GE R010 0 elselabel // check if a is less than 0
	ADD R010 1 R010 // Increment value by 1
	STI R010 R000 // store the computed value back to memory
	JMP endif
	elselabel:
	SUB R010 1 R010 // Decrement value by 1
	STI R010 R000 // store the computed value back to memory
	endif:

b. Expressions :

x = y + z;

```
bin_expr --> E1 + E2        {   bin_expr.temp = newtemp();
                                bin_expr.code = E1.code || E1.code ||
                                bin_expr.temp || ":=" || E1.temp || "+" || E2.temp;
                            }
```

User Code	Intermediate Assembly Code

float a = 2.1;
float b = 5.2;
float c = 0.0;
 	
c = a + b;
	begin:
	MOVIF 100 F000
	STF 2.1 F000 // store global float variable a at address 0x100
	FADD F000 4 F000
	STF 5.2 F000 // store global float variable b at address 0x104
	FADD F000 4 F000
	STF 0.0 F000 // store global float variable c at address 0x104
	MOVIF 100 F000
	LDI F000 F001 // load global variable a again
	FADD F000 4 F000
	LDI F000 F002 // load global variable b again
	FADD F000 4 F000
	FADD F001 F002 F003 // sum of a and b in register
	STF F003 F000 // store result back into C

c. Function Definitions and Calls:

```
int a = 2;
float b = 3.0;
foo(a, b);
```

```
ident --> TOK_IDENT         {   ident.lval = TOK_IDENT.val;
                                ident.code = " ";
                            }

FUNC_INV --> ident ( Exp1 , Exp2 )   {   FUNC_INV.rval = newtemp();
                                         FUNC_INV.code = ident.code ||
                                                         Exp1.code ||
                                                         Exp2.code ||
                                                         codeGen( push Exp1.rval )
                                                         codeGen( push Exp2.rval )
                                                         codeGen( jmp ident.offset )
                                                         codeGen( pop FUNC_INV.rval)
                                     }
```

Function/Procedure calls will involve setting up the activation record for each invocation of a function. This invocation occurs by pushing appropriate elements onto the stack by the caller and callee.

In the implementation, the **caller** modifies the stack by performing the following operations in sequence :
a.) Push actual parameters onto the stack from right to left.
b.) Allocate space for return value.
c.) Push return address onto the stack.

The **callee** then performs the following operations :
a.) Push local variables onto the stack and move RSP (decrement RSP by size of local variable).
b.) Allocate space for temporary variables and move RSP to the bottom of the activation record.

d. Event Matching:

We intend to implement Event Matching using the technique mentioned in [3]. As per this technique, the generated code will first read a single character event name and match against the existing event declarations. On a successful match, we would read the appropriate number of parameters using the INI/INF instruction.

Eg:

```
IN R100        // Read event name. Say "a"
  ...          // Match the event name with event declarations.  Say, "event a(int x1, float x2)"
INI R101       // Read the integer parameter
INF F101       // Read the floating point parameter
```

V. Code Optimizations

Code Optimization is one of the most important phases of compiler design, which will improve the runtime performance of the final generated code. Following are different code optimization techniques which we intend to implement:

a. Function Inlining:

The overhead associated with calling and returning from a function can be eliminated by expanding the body of the function inline. Plus additional opportunities for optimization may be exposed as well.
Eg:

```
int sub (int x, int y) { return add (x, -y); }   =>   int sub (int x, int y) { return x - y; }
```

b. Constant folding:

Expressions with constant operands can be evaluated at compile time, thus improving runtime performance and reducing code size by avoiding evaluation at compile-time.
Eg:

```
int f (void) { return 3 + 5; }                   =>   int f (void) { return 8; }
```

c. Constant propagation:

Constants assigned to a variable can be propagated through the flow graph and substituted at the use of the variable.
Eg:

```
{ x = 3; y = x + 4; }                            =>   { x = 3; y = 7; }
```

d. Strength Reduction:

Replace expensive operations with equivalent cheaper (more efficient) ones.
Eg:

```
{ y = 2; z = x^y; }                              =>   { y = 2; z = x*x; }
{ MOV x, 0; }                                    =>   { XOR x, x; }
```

e. Loop-Invariant Code Motion:

Loop-invariant code motion is the most common form of code motion optimization that moves statements that evaluate to the same value every iteration of the loop to somewhere outside the loop.
Eg:

```
while (i) { j = a+b; a[i] = j; i++; }            =>   j = a+b; while (i) { a[i] = j; i++; }
```

f. Dead Code Elimination:

Code that is unreachable or that does not affect the program (e.g. dead stores) can be eliminated.

Eg:

```
void f() {int i = 10; return a+b;}        =>    void f() { return a+b;}
```

g. Common Subexpression Elimination:

An expression is a Common Subexpression (CSE) if the expression was previously computed and the values of the operands have not changed since the previous computation. Recomputing the expression can be eliminated by using the value of the previous computation.

Eg:

```
{ i = x + y + 1; j = x + y; }        =>    { t1 = x + y; i = t1 + 1; j = t1; }
```

h. Integer Divide Optimization:

Integer divide instructions are usually slower than other instructions such as integer addition and shift. Divide expressions with power-of-two denominators or other special bit patterns can be replaced with a faster instructions.

Eg:

```
int f (int i) { return i / 2; }        =>    int f (int i) { return i >> 1; }
```

i. Integer Multiply Optimization:

On many architectures, integer multiply instructions are slower than other instructions such as integer add and shift, and multiply expressions with power-of-two constant multiplicands and other bit patterns can be replaced with faster instructions.

Eg:

```
int f (int i) { return i * 2; }        =>    int f (int i) { return i << 1; }
```

j. Integer Mod Optimization:

On most architectures, integer divide is a relatively expensive instruction. Power-of-two integer modulus expressions can be replaced with conditional and shift instructions to avoid the divide and multiply and increase run-time performance.

Eg:

```
int f (int x) { return x % 8; }        =>    int f (int x) { int temp = x & 7;
                                             return (x < 0) ? ((temp == 0) ? 0 : (temp | ~7)) : temp; }
```

VI. Conclusion

So far we have successfully implemented the front-end of the compiler i.e. lexical analysis, syntax analysis, type checking phases of a compiler. Currently we are working on the back-end of the the compiler that deals with allocation of memory for global and local variables in data section and stack respectively using memAlloc() function. Once we allocate the lvalues of the variables, we shall generate assembly code using codeGen() function of all AstNode subclasses. Finally different code optimizations as mentioned in the above section will be performed on the generated assembly code. This optimization phase can be however disabled via an user controlled compiler option “-noopt”, so as to obtain the compile time benefits as opposed to the runtime performance.

We intend to distribute the work amongst the team members equally. Two team members will simultaneously work upon a single module as illustrated below:

memalloc() function for all subclasses => Tejal Kaple and Sahil Parmar
codeGen() function for all subclasses => Sohil Gandhi and Aniket Alshi

On the successful implementation of the above phases, the code optimization phase will be distributed among all the four team members.

VII. References

[1] Jeffery Ullman and Alfred Aho, *Principles of Compiler Design*
[2] Lecture Notes by Prof. R. Sekar, *http://seclab.cs.sunysb.edu/sekar/cse504/#notes*
[3] R. Sekar and P. Uppuluri, *Synthesizing Fast Instrusion Prevention/Detection Systems from High-level Specifications*
[4] Code Optimization Techniques: *http://www.compileroptimizations.com/index.html*