E- - Compiler

Implementation Report

5/18/2010

CSE 504, Stony Brook University
Navdeep Bhatia
Pritam Damania
Raunak Kumar
Vaibhav Shrivastava
Subramanian Arumugam

Table of Contents

Table of Figures

Introduction

This report is being submitted in fulfillment of project for course work CSE 504 for spring 2010. The compiler designed and implemented is for E- - language, the grammar for which is defined as part of appendix A.

Stages of Compilation

Following figure depicts the stages of compilation that the compiler goes through to generate the executable.

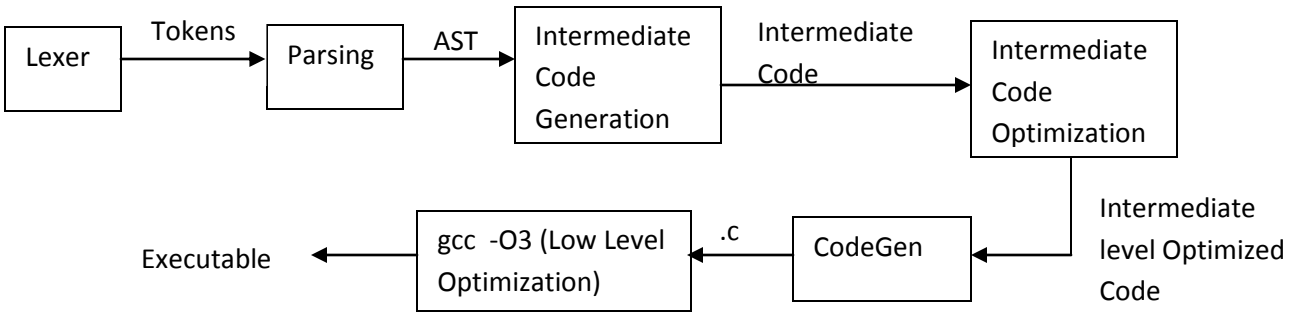

Figure 1 Stages of Compilation

Design

The compiler consists of following components:

1) Lexer
2) Parser
3) Type Checker
4) Intermediate Code Generation
5) Pattern Matching
6) Optimization

Lexer

The lexer is responsible for generating tokens to be used by the parser to parse the grammar for E - -. It was implemented using flex tool. The lexer was used as was developed for earlier Assignment with two additions. One to support while loops and another to extend the grammar to print messages. For this purpose three tokens "TOK_WHILE", "TOK_BREAK" and "TOK_PRINT" were defined.

Parser

To parse the grammar for E - -, productions were written using Bison, a LR parser. The parser was also used from earlier assignment and was extended to include print as well as while as part of the statement nodes. The corresponding definitions were also propagated to Abstract Syntax Tree to create statement nodes for while statement as well as print statement. For this, two new classes were added namely, WhileNode and PrintStmtNode.

The parser parses the input file written as per the syntax of E - - language. In case of syntax errors, it reports to the user specifying the error and the corresponding file name and line number. e.g.

```
test:13:Error: Syntax error while parsing a statement
```

While parsing, it allocates appropriate nodes (VariableEntry for variable declarations, FunctionEntry for function declarations and definitions etc) and inserts those into the symbol table. Parser also checks whether variables are defined or not while they are being used in expressions by getting their entries from the symbol table. After the grammar has been parsed, an abstract syntax tree is produced for every statement/expression that is used by succeeding components to generate intermediate code.

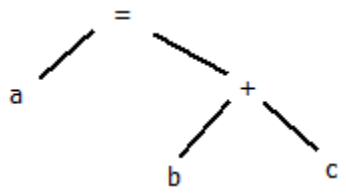

Figure 2 Abstract Syntax Tree for expression "a=b+c"

Type Checker

After AST is created; we verify types for each of the Symbol table Entries and each node of the abstract syntax tree by traversing through the tree. The type verification is done based on the fact that entry on the left hand side should be the sub type of the entry on the right hand side. e.g.

Expressions:

a = b + c

i) If a is of INT type the RefExpr Nodes on the right hand side should be subtype of INT otherwise following error is generated:

```
test:2: Error: Assignment between incompatible type
```

If and While Node:

- The expression node for condition in If and While Nodes should result in boolean values otherwise following error is generated:

```
test:30: Error: While conditional Expr is not Boolean
```

Function Invocation:

- The return type of function invocation node should be subtype of the entity on the right hand side.

Intermediate Code Generation

The Intermediate code component of the compiler generates intermediate code from Abstract Syntax Tree. This component is further divided into two components:

1) Intermediate Code Generation Framework

2) Intermediate code Generator

Intermediate Code Generation Framework

This component provides the basic framework (or utilities) used by the intermediate code generator. It consists of following utilities:

1) Register Manager
2) Memory Manager
3) Label Generator
4) Quadruple Manager

Register Manager

Register Manager maintains map for all the allocated registers and also keeps track of free and used registers for both integer as well as float registers. The map is maintained to keep track of variables and to retrieve them while generating intermediate code for expressions. The free and allocated lists are maintained to serve the requesting intermediate code generator with the available registers and delete them from the used list when they are done. Register Manager is implemented in `class RegTable` and exposes following methods for the above mentioned functionality:

```
RegTable::getFreeIntReg()
string RegTable::getFreeFloatReg()
int RegTable::del(string key)
```

Memory Manager

Memory Manager keeps track of the memory assigned to global variables. As these need to be assigned memory locations at compile time, Memory Manager provides the next available memory location. Since, memory was not the constraint, memory manager keeps on incrementing the memory location counter to provide the next available memory location. This is implemented by `class MemManage`

Label Generator

Label Generator generates labels by appending an incremented integer to string "L" as "L1", "L2" and so on. These labels are used by intermediate code generator for specifying locations to jump in case of function calls or jump statements.

Quadruple Manager

Quadruple Manager provides an interface for every statement of intermediate code. Intermediate code generator calls quadruple manager to generate a new statement in form of a quadruple. This provides a uniform interface that can be used by various components of the intermediate code generator to generate quadruples. Quadruple manager also maintains list of quadruples (statements of intermediate code) and inserts a new quadruple into the list whenever it creates one.

Intermediate code Generator

Intermediate Code generator generates intermediate code using the framework (utilities) by traversing through the AST. For each and every node of AST it follows different rules to generate intermediate code. There are following types of nodes in the AST:

1. ExprNode
    a. RefExprNode
    b. OpNode
    c. ValueNode
    d. InvocationNode
2. BasePatNode
    a. PrimitivePatNode
    b. PatNode
3. StmtNode
    a. WhileNode
    b. BreakNode
    c. ReturnStmtNode
    d. ExprStmtNode
    e. CompoundStmtNode
    f. PrintStmtNode
    g. IfNode
4. RuleNode

Computing Expressions

Expressions are computed in a bottom up manner in the AST. Each of ValueNode, RefExprNode, OpNode and InvocationNode return a register to its parent. The parent node now uses this returned register to compute the necessary operation. For example suppose we have E1 + E2, then code for E1 is generated , which stores the result of E1 in a register R1 and this register is returned back, same goes for E2. Then the current OpNode basically generates code as follows:

ADD R1 R2 R3

Now, the current OpNode returns the register R3 indicating that the computed value is stored in R3.

Function Invocation and Return

Function Invocation is implemented by considering part of the memory as a stack. When a function is invoked, the return label and the parameters of the function call are pushed onto the stack. Now in the code generated for Function Definition, a Stack Register and a Base Register are used. The Base Register points to the memory location onwards which the local variables of the function are stored. All the local variables of the function are now given memory locations relative to the Base Pointer. Therefore, to load a variable from memory, the base register contents are added to the relative memory address to get the actual memory address.

To improve efficiency, the code was modified to store local variables always in registers and keep a track of active registers for each function definition. When there is a function call within a function, the list of active registers are pushed onto the stack and when the function returns, these registers are restored from the stack. This enables reuse of registers and easy implementation of recursive functions. Also, the stack pointer and base pointers are accordingly updated as we invoke a function and return from it. The stack pointer points

to the top of the stack indicating the next free word in memory which can be used for allocation. Also a particular register is assigned to hold the return value of a function. Specific Registers have been assigned as follows:

R999: Stack Register
R998: Return address Register
R997: Base Register
R996: Return value Register

Pattern Nodes

Pattern Nodes do not generate code but they play their part in constructing DFA used by the Rule node.

While and Break

While starts with generating code for its condition which is a ExprNode and then the ExprNode's generateCode returns while the label where the true statements are to be placed. Immediately after generating code for condition, we generate a JMP statement which jumps to the statement next to the while, this is for handling false condition. Now the code for true statements gets generated at label returned by the ExprNode. Altogether fours Labels gets created during the code generation of while

=> While Start label
=> True Statement Label
=> While End Label

As soon as we create the end labels before we generate code for true statements we push the label into a stack which will be used if the true statement contains break statements in it. Depending upon the parameter 'n' of break we pop out n labels from stack and create a JMP instruction for that label.

Following code gives a sample of Intermediate code for while statement

```
while(d>=10)
    {
            print(d,"\\n");
            if(d==15)
                    break 1;
            d=d+1;
    }
```

Output:

```
L14:  JMPC GE R000 10 L16
JMP L15
L16:  PRTI R000
PRTS "\n"
JMPC EQ R000 15 L19
JMP L17
L19:  JMP L15
L17:  ADD R000 1 R012
MOVI R012 R000
```

```
JMP L14
L15:  JMP L13
```

If Statement Node

If starts with generating code for its condition which is a ExprNode. Two labels are generated:
1. For the false condition when the code would go to the else part.
2. At the end of the If-else block where the then_ part of the loop would jump after its execution and also the else_ part would jump after its execution.

The flow of the IfNode is as follows:

```
If Condition is false jump to ElseLabel //else continue
                  Stmt1 (then_)
                 Jump to End Label
ElseLabel :
             Stmt2 (else_)
             Jump to Endlabel
Endlabel:
```

Print Statement

A print statement is added into the grammar specification and it has the following syntax: print(arg1,arg2,.....,argn), where 'n' is variable and each arg can be of any type. The print statement node generates intermediate code based on the type of the argument. E.g. to print integer, it generates intermediate code that uses "PRTI" as its command and similarly "PRTS" for strings.

Pattern Matching

In case of pattern matching, first of all a graph is built from the rules defined in the grammar file. The graph is then processed with the set of inputs to construct a DFA. The DFA generated is stored in the form of an adjacency matrix with rows depicting event name and column depicting next state. We also store final state in a register so that after every state transition we can check if we have reached the final state or not. If we reach the final state we execute the action corresponding to that rule.

e.g. the adjacency matrix for rule "a() \/ b() : c()" would be

Input(in ASCII)/States	**0**	**1**	**2**	**3**
0	-1	-1	-1	-1
1	-1	-1	-1	-1
…	…	…	…	…
97 (a)	1	-1	-1	-1
98 (b)	1	-1	-1	-1
99 (c)	-1	2	-1	-1
…	…	…	…	…

127	-1	-1	-1	-1

DFA for this would be

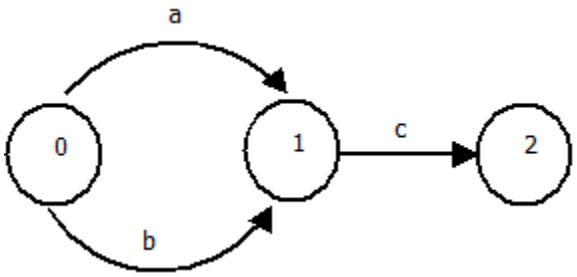

Figure 3 An Example DFA

This is stored in memory in consecutive memory locations so that we can access it directly. Compiler assigns a register to every pattern to store the current state and the final state. The current state is updated every time on receiving an input.

*Currently pattern matching for rules of the form "a()** \/ b()" is not supported.*

Optimization

Following set of optimizations are being performed by the compiler on intermediate code:

1) Constant Propagation
2) Jump Threading
3) Strength Reduction
4) Common Sub-Expression Elimination
5) Dead Code Elimination

For Low level optimizations we rely on gcc -O3 to do the job.

Creation of Blocks

To divide the Intermediate code into blocks labels associated with functions are used. These blocks have within them smaller blocks which correspond to the labels produced by if and while statements. Control Flow within these blocks is handled later. For variables in if and else statements a safe approach has been taken. All the uses are considered.

Induction Variables

Within a function block, there could exist variables which are used within loops. These variables are thus computed more than once, even though a single instruction may exist. To incorporate this aspect of control flow, the variables and labels within a block are stored in a list. When a jump back to a previously defined label is encountered, we realize this label is a part of a loop. By maintaining the earliest use and the latest definition of all the variables, one can verify if the definition of a variable is used in computation of the variable again in the next loop. By identifying all the variables as induction variables and preventing constant propagation, we take care of possible error cases.

Constant Propagation

The value of the constants is propagated down the block so that an additional Register operation can be saved. It starts traversal from the beginning and propagates the constants down the block.

When a constant is declared, it uses instruction of the form:

MOVI const, Rx

This value of const. is propagated down the block and replaced by every occurrence of Rx unless Rx is a destination Register. The Register Rx is freed at the end.
The constant is not propagated if it is a loop variable. The value of loop variable would always change for every occurrence of the loop. Therefore a list of loop-variables is maintained while scanning the block. Whenever this variable is encountered it is skipped.

Jump Threading

Jump threading refers to the process of removing unnecessary jumps. If a label l1 has an immediate Jump to another label l2; then one can replace all occurrences of l1 with l2 and remove the l1 label all together. This prevents unnecessary jumps.

Strength Reduction

Expensive instructions involving multiplication are replaced by move instruction if the multiplicative factor is 0 or 1. This was further extended to instructions containing addition by 0. To further optimize the code, if the source and destination registers are same for multiplicative factor of 1 or addition by 0, the instruction is deleted.

Common Sub-Expression Elimination

A simple technique has been used to perform Common sub-expression elimination. Once an expression is evaluated, it is stored in a list of expressions already computed. Further down the line when an expression is encountered, this list is looked up and appropriately the register holding the value of the queried expression is returned. If a variable is assigned, this list is searched and all expressions consisting this variable are removed from the list, since now these expression are no longer valid as the value of the variables have changed.

This technique is very effective and serves the purpose of optimizing the code via common sub-expression elimination.

Dead Code Elimination

Dead code elimination starts from the last instruction of the block and remove those statements which write to a local variable register and the register is not being used in any of the subsequent statements. This is achieved by maintaining two lists, one for maintaining the global variables and one for used registers. After processing an instruction its arguments are scanned and are put into the used registers list which helps in processing the instructions above it. E.g. Assume the following are the last few instructions of a function

MOVI 100 R0100 => this instruction writes to R0100 register which belongs to a local variable and the register is not being used anywhere below it. So we can delete this statement.
JMP R999

```
a()-> {
```

```
                int i=8;
                int k;
                while(b)
                {
                        i=i+1;
                }
                k=10;
      };
```

```
Without Dead Code Elimination:

RL6:  MOVI 1 R901
MOVI 8 R006
L8:  JMPC EQ R002 1 L10
JMP L9
L10:  ADD R006 1 R009
MOVI R009 R006
JMP L8
L9:  MOVI 10 R008
JMP L7


With Dead Code Elimination

RL6:  MOVI 1 R901
MOVI 8 R006
L8:  JMPC EQ R002 1 L10
JMP L9
L10:  ADD R006 1 R009
MOVI R009 R006
JMP L8
L9:  JMP L7
```

Compiler doesn't delete those instructions which are written into a global variable register because it is possible that this register may be used in some other part of the code.

Performance

The performance of the compiler vis-à-vis gcc is shown in the following table for different scenarios:

Test Case/Compiler	**E - - without Optimization (Time in sec)**	**E- - with Optimization (Time in sec)**	**gcc (Time in sec)**
Dead Code Elimination	8.21 s	5.5 s	4.22 s

Dead Code + CSE	22 s	5.5 s	12.66 s
Recursive Functions	11 s	11 s	4.5 s
CSE	21 s	5 s	13 s
Strength Reduction	24 s	13 s	8.42 s

Time command was used to calculate the time. The test cases used for this are in appendix B. Note that if we add –O3 to generate the executable, the times are exactly similar to gcc –O3.

Limitations

The compiler currently performs well for the numerous scenarios but suffers from following limitations:

1) Pattern matching for rules of the form "a()** \/ b()" is not supported.
2) Function invocation (recursive) is taking longer than gcc. This can be further optimized.
3) Register optimizations are not being done currently.
4) Event parameters are not supported.

Appendix A

E-- Syntax

1. A Specification consists of a (possibly empty) list of bf Declarations and a non-empty list of Rules.

2. Statements, declarations, and rules are terminated by a semicolon. Semicolons are optional following a closing brace.

3. A Declaration declares a Class, Function, Event or a Variable.

4. A class declaration consists of the keyword class and a name for the class. Classes represent an abstract, external datatype in E--. Class objects can be parameters to external functions or events, but not variables.

5. A function declaration consists of a Type, function name, and zero or more comma-separated FormalParam's enclosed by a parenthesis. Optionally, a function may have a body, which consists of a (possibly empty) sequence of variable declarations and one or more statements, all of them enclosed by braces.

6. A FormalParam consists of a Type followed by a variable name.

7. A Variable declaration consists of a Type, followed by a comma-separated list of variable names. Each variable name may have an optional initialization, which consists of TOK ASSIGN and an expr.

8. An Event declaration consists of the keyword event followed by an event name and comma-separated list of FormalParam's enclosed by a parenthesis. A special event any matches any event, and need not be followed by parenthesis.

9. A Rule consists of an EventPattern and a Statement separated by TOK ARROW.

10. An EventPattern can be a PrimitivePat or be obtained from other event patterns using one of the operators
! (negation), : (concatenation), \/ (alternation) or ** (closure). These operators have the usual precedences and associativities of regular expression operators. Parentheses may be used to override these precedences and associativities.

11. A PrimitivePat consists of an event name, followed by event formal parameters (variable names, i.e., identifiers) enclosed within parentheses and separated by commas. A primitive pattern may be optionally followed a condition, which consists of a TOK BITOR followed by an expr.

12. A Statement is one of the following
- IfStatment, of the form if expr Statement, followed optionally by else Statement
- EmptyStmt, which is empty.
- FunctionInvocation, which consists of a function name followed by a comma-separated, parenthesisenclosed, possibly empty list of Expr
- Assignment of the form RefExpr = Expr
- ReturnStmt of the form return Expr
- CompoundStatement that consists of a sequence of one or more Statement's that is enclosed in braces.

- Print Statement that can print variable number of arguments.
- While Statement to define a loop which can also contain break statement consisting of "break n" where n denotes the number of loops to break out of.

13. The RefExpr is simply a variable name

14. An Expr is one of:
- Literal
- RefExpr
- Assignment
- FunctionInvocation
- op Expr, for unary operator op
- Expr op Expr, for binary operator op

The precedence and associativity of different operators were specified earlier. Parentheses could be used to override these precedences and associtivity.

15. A Type is a base type (one of void, bool, string byte, int, or double), or a type name. A base type could be preceded by the keyword unsigned.

Appendix B

Test Cases

Strength Reduction:

```
int n=100000;
int m=1000;

event a();

a()->
{
        int a=1,b=1,c=1,d=1,e=1,f=1,g=1,h=1;
        int count=0;
        while(n>0)
        {
                m=10000;
                //print(n,"\\n");
                while(m>0)
                {
                //print("A : ",m,"\\n");
                a=b+c;
                d=(b+c)*1*(b+c);
                 d = d + 0;
                 d = d*1;
                e=(a+d);
                f=(a+d)*1;
                g=g*1;
                h=(a+0)+(b+c)+(0);
                //print("B : ",m,"\\n");
                  m=m-1;
                //print("C : ",m,"\\n");
                count=count+1;
                }
                n=n-1;
        }
        print("\\n",count,"\\n");
}
```

Common Sub-Expression Elimination

```
int n=100000;
int m=1000;

event a();

a()->
{
        int a=1,b=1,c=1,d=1,e=1,f=1,g=1,h=1;
        int count=0;
        while(n>0)
        {
```

```
                m=10000;
                //print(n,"\\n");
                while(m>0)
                {
                a=b+c;
                d=(b+c)*(b+c)*(b+c);
                e=(a+d);
                f=(a+d)*(b+c);
                g=(f+e);
                h=(a+d)+(b+c)+(f+e);
                m=m-1;
                count=count+1;
                }
                n=n-1;
        }
        print("\\n",count,"\\n");
}
```

Recursive Functions

```
int f(int n)
{
        if(n==0)
                return 0;
if(n==1)
return 1;
return f(n-1)+f(n-2);
}

int fiter(int n)
{
        int i=1;
        int ret=1;
        while(i<=n)
        {
        ret=ret*i;
        i=i+1;
        }
return ret;
}
int fact(int n)
{
if(n==0)
return 1;
return n*fact(n-1);
}

event a();

int n=37;
int m=1000000000;
int k=100000;
a()->
{
int l=10;
```

```
print("\\n",f(n),"\\n");
print(fiter(m),"\\n");
print(fact(k),"\\n");
print(fact(l));
}
```

Dead Code – CSE

```
int n=10000;
int m=10000;
event a();
a()->
{
        int a=1,b=1,c=1,d=1,e=1,f=1,g=1,h=1,i=1;
        int count=0;
while(n>0)
{
        m=100000;
        while(m>0)
        {
                a=(b+c)*(b+c)*(b+c)*(b+c);
                d=(e+f)*(e+f)*(e+f)*(e+f);
                g=(h+i)*(h+i)*(h+i)*(h+i);
                m=m-1;
        }
        n=n-1;
}
print("\\n","Done","\\n");
}
```

Dead Code Elimination

```
int n=10000;
int m=10000;
event a();
a()->
{
        int a=1,b=1,c=1,d=1,e=1,f=1,g=1,h=1,i=1;
        int count=0;
while(n>0)
{
        m=100000;
        while(m>0)
        {
                a=(b+c);
                d=(e+f);
                g=(h+i);
                m=m-1;
        }
        n=n-1;
}
print("\\n","Done","\\n");
}
```