CSE504 Compiler Design
Implementation of a Compiler

YoungMin Kwon

Overview

- We will learn how various parts of a program are translated into assembly codes.

Definition: Program/Function/Procedure

```
program
    : program_def { Translate($1); }
    ;
program_def
    : PROGRAM ID ';'
      opt_definition_list
      BEGIN_ opt_stmt_list END      { $$ = CreateProgramDef...}
    ;
function_def
    : FUNCTION ID '(' opt_formal_param_list ')' ':' basic_type ';'
      opt_variable_def_list         /*for NESTED_FUNC, use opt_definition_list*/
      BEGIN_ opt_stmt_list END      { $$ = CreateFunctionDef...}
    ;
procedure_def
    : PROCEDURE ID '(' opt_formal_param_list ')' ';'
      opt_variable_def_list         /*for NESTED_FUNC, use opt_definition_list*/
      BEGIN_ opt_stmt_list END      { $$ = CreateFunctionDef...}
    ;
```

Definitions: Variables

```
variable_def
    : VAR
variable_type_pair ';'
    ;
variable_type_pair
    : variable_list ':'
type
    ;
variable_list
    : variable_list ',' ID
    | ID
    ;
```

```
type
    : array_type
    ;
array_type
    : ARRAY '[' NUM ']' OF
      array_type
    | ptr_type
    ;
ptr_type
    : '^' ptr_type
    | basic_type
    ;
basic_type
    : INTEGER
    | REAL
    | '(' type ')'
    ;
```

Definitions: Factor

```
factor
    : '(' expr ')'
    | NUM
    | ID
    | ID '(' opt_actual_param_list ')' { $$ = CreateFunctionCall...}
    | ID '[' actual_param_list ']'  { $$ = CreateArrayIndex... }
    ;
opt_actual_param_list
    : actual_param_list
    |
    ;
actual_param_list
    : actual_param_list ',' expr    { $$ = AddToActualParamList...}
    | expr                          { $$ = AddToActualParamList...}
    ;
```

Quiz: modify the syntax to use C-style array indexing. (e.g. a[1][2])

Symbol Table

```
struct SymTableEntry
{
    char *id;               //name of the id
    struct Type* type;      //type of the id
    bool isGlobal;          //whether id is a global var
    int offset;             //local variable offset from bp
    int size;               //size of the local variable
    int label;              //label for function
    SymTableEntry *next;

    SymTableEntry();
    SymTableEntry(char* id, bool isGlobal);
    SymTableEntry* SetId(char* id);
    SymTableEntry* SetIsGlobal(bool isGlobal);
    SymTableEntry* SetType(Type* type);
    SymTableEntry* SetOffset(int offset);
    SymTableEntry* SetLabel(int label);
    SymTableEntry* SetNext(SymTableEntry* next);
    SymTableEntry* Find(char* id);
};
```

Symbol Table

```
struct SymTable
{
    SymTableEntry head;     //link to the first entry
    int varSize;            //size of variables to allocate
    SymTable* nextScope;    //link to the next scope
    SymTable* next;         //link all tables created
    static SymTable* latestTable;

    SymTable();
    SymTableEntry* Find(char* id, int* linkCount);
    SymTableEntry* Insert(char* id);

    static SymTable* PushNewTable(SymTable* top);
    static SymTable* PopTable(SymTable* top);
};
```

SyntaxNode (Base Type)

```
struct SyntaxNode
{
    virtual int Tag();
    // Update Symbol Table
    virtual void Pretranslate(SymTable* symTable);

    // Translate this node
    virtual void Translate(SymTable* symTable);

    // For LHS, to pass function, array, struct...
    virtual void TranslateAsLValue(
                                SymTable* symTable);
};
```

Translation of Number

```
SyntaxNum::SyntaxNum(double value)
    : value(value)
{
    FUNCTION_TRACE;
}

void SyntaxNum::Translate(SymTable* symTable)
{
    FUNCTION_TRACE;

    // Saving the result at rax
    // Round to the nearest number (handling int only)
    printf("    mov     rax, %d\n", (int)(value + .5));
}
```

Translation of Id

```
void SyntaxId::TranslateAsLValue(SymTable* symTable)
{
    int linkCount = 0;
    SymTableEntry *entry = symTable->Find(id, &linkCount);
    if(entry == NULL)
        errQuit("Unknown variable %s.\n", id);

    if(entry->type->typeTag == TypeFunction)
    {
      printf("    mov     rax, label_%03d  ;  %s\n",
             entry->label, entry->id);
    }
    else if(entry->isGlobal)
    {
      printf("    mov     rax, global_vars + %d ;  %s\n",
             entry->offset, entry->id);
    }
    else /*parameters and local variables*/
    {
      printf("    lea     rax, [rbp + %d] ;  %s\n",
             entry->offset, entry->id);
    }
}
```

Translation of Id

```
void SyntaxId::Translate(SymTable* symTable)
{
    TranslateAsLValue(symTable);

    int linkCount = 0;
    SymTableEntry *entry = symTable->Find(id, &linkCount);

    // Treat array and function names as a reference
    // Array index and Function call are handled separately.
    bool toReference = entry->type->typeTag == TypeArray ||
                       entry->type->typeTag == TypeFunction ||
                       entry->type->typeTag == TypeProduct;
    if(!toReference)
    {
        printf("    mov     rax, [rax]\n");
    }
}
```

Translation of Expr

```
// Saving the result at rax
void SyntaxExpr::Translate(SymTable* symTable)
{
...
    if(op == OpAddressOf)
        lexpr->TranslateAsLValue(symTable);
    else
        lexpr->Translate(symTable);

    // For binary operators, rax = operand 1, rbx = operand 2
    if(rexpr)
    {
        // push/pop rax is less than efficient, but we don't
        // know what registers will be used in rexpr->Translate.
        printf("    push    rax\n");
        rexpr->Translate(symTable);
        printf("    mov     rbx, rax\n");
        printf("    pop     rax\n");
    }
...
```

Translation of Expr

```
...
 switch(op)
    {
    case '+' :      printf("    add     rax, rbx\n");        break;
    case '-' :      printf("    sub     rax, rbx\n");        break;
    case '*' :      printf("    imul    rbx\n");             break;
    case '/' :      printf("    xor     rdx, rdx\n");
                    printf("    idiv    rbx\n");             break;
    case OpNeg :    printf("    neg     rax\n");             break;
    case OpAND:     printf("    and     rax, rbx\n");        break;
    case OpOR:      printf("    or      rax, rbx\n");        break;
    case OpNOT:     printf("    not     rax\n");             break;
    case OpValueOf: printf("    mov     rax, [rax]\n");      break;
...
```

```
Quiz: Write a code for OpEQ
```

Translation of Expr

```
const char* jmpFmt = NULL;
switch(op)
{
...
case OpEQ:      jmpFmt="    je      label_%03d\n";      break;
case OpNE:      jmpFmt="    jne     label_%03d\n";      break;
case OpGE:      jmpFmt="    jge     label_%03d\n";      break;
case OpGT:      jmpFmt="    jg      label_%03d\n";      break;
case OpLE:      jmpFmt="    jle     label_%03d\n";      break;
case OpLT:      jmpFmt="    jl      label_%03d\n";      break;
}

if(jmpFmt != NULL)
{
    int label = newLabel();
    printf("    mov     rcx, 1\n");
    printf("    cmp     rax, rbx\n");
    printf(jmpFmt, label);
    printf("    xor     rcx, rcx\n");
    printf("label_%03d:\n", label);
    printf("    mov     rax, rcx\n");
}
```

Assignment Statement

```
void SyntaxAssignStmt::Translate(SymTable* symTable)
{
...
    else
    {
        SyntaxExpr* lvalue = (SyntaxExpr*)lexpr;
        lvalue->TranslateAsLValue(symTable);

        printf("    push    rax\n");
        rexpr->Translate(symTable);
        printf("    pop     rbx\n");

        // copy rax to the address rbx is pointing to
        printf("    mov     [rbx], rax\n");
    }
...
```

Assignment Statement

```
void SyntaxAssignStmt::Translate(SymTable* symTable)
{
    // check if this is for the return value. e.g. foo := 1
    bool isReturnValue = false;
    if(lexpr->Tag() == TagSyntaxId)
    {
        SyntaxId* idExpr = (SyntaxId*)lexpr;
        int linkCount = 0;
        SymTableEntry* entry = symTable->Find(idExpr->id, &linkCount);
        if(entry && entry->type &&
           entry->type->typeTag == TypeFunction)
            isReturnValue = true; // TODO: check the function name
    }

    if(isReturnValue)
    {
        rexpr->Translate(symTable);
        printf("    mov     r15, rax\n");//r15 keeps the return value
    }
...
```

Conditional Statement

```
void SyntaxCondStmt::Translate(SymTable* symTable)
{
    if(elseStmt)
       ...
    else
    {
        int labelExit = newLabel();

        PRNCMT("    ;; if then\n");
        test->Translate(symTable);
        printf("    cmp     rax, 0\n");
        printf("    jz      label_%03d\n", labelExit);

        PRNCMT("    ;; then\n");
        thenStmt->Translate(symTable);
        printf("label_%03d:\n", labelExit);
        PRNCMT("    ;; end if then\n");
    }


Quiz: Write the if-then-else case
```

While Statement

```
void SyntaxWhileStmt::Translate(SymTable* symTable)
{
    ...
    int labelTest = newLabel();
    int labelExit = newLabel();

    printf("label_%03d:\n", labelTest);
    // result of test expr will be saved at rax
    test->Translate(symTable);
    printf("    cmp     rax, 0\n");
    printf("    jz      label_%03d\n", labelExit);

    // translate the body
    stmt->Translate(symTable);

    printf("    jmp     label_%03d\n", labelTest);
    printf("label_%03d:\n", labelExit);
}
```

Function Call

```
void SyntaxFunctionCall::Translate(SymTable* symTable)
{
    int linkCount = 0;
    SymTableEntry* entry = symTable->Find(id, &linkCount);

    // push actual parameters
    if(params)
        params->Translate(symTable);

    // Call the function
    printf("    call    label_%03d ;  %s\n",
               entry->label, id);

    // Pop the parameters
    int actualParamSize = params?
                          params->TotalParamSize(): 0;
    printf("    add     rsp, %d\n", actualParamSize);
}
```

Actual Parameters

```
*In our implementation, SyntaxTree for formal and Actual Parameter lists are
reversed
         f(1, 2, 3) : 3 -> 2 -> 1

void SyntaxActualParamList::Translate(SymTable* symTable)
{
    if(next)
        next->Translate(symTable);
    expr->Translate(symTable);
    printf("    push    rax\n");
}

int SyntaxActualParamList::TotalParamSize()
{
    int exprSize = 8;  // TODO: use the actual param size
    if(next)
        return exprSize + next->TotalParamSize();
    return exprSize;
}

Quiz: describe the stack for the actual parameters when foo(1, 2, 3) is called.
```

Array Index

```
int Type::Size()
{
    FUNCTION_TRACE;

    switch(typeTag)
    {
    case TypeVoid:      return 0;
    case TypeInteger:   return 8;
    case TypeReal:      return 8;
    case TypePointer:   return 8;
    case TypeArray:     return arraySize * left->Size();
    case TypeProduct:   return (left? left->Size(): 0) +
                               (right? right->Size(): 0);
    case TypeFunction:  return 0;
    case TypeReference: return 8;
    }
    return 0;
}
```

```
Quiz: find the offset of a[1][2][3] from a when a is defined as
      var a : array[10] of array[20] of array [30] of integer;
```

Array Index

```
void SyntaxArrayIndex::TranslateAsLValue(SymTable* symTable)
{
    int linkCount = 0;
    SymTableEntry* entry = symTable->Find(id, &linkCount);
    // Evaluate the indexes
    params->Translate(symTable);

    // Get the array size of each level from the type
    int arraySize[50], nas = 0; // 50: up to 50 dimensional array
    for(Type *t = arrayType; t; t = t->left)
        arraySize[nas++] = t->Size();

    // Compute the index
    printf("    xor     rbx, rbx\n");
    while(--nas >= 1)
    {
        printf("    pop     rax\n");
        printf("    mov     rcx, %d\n", arraySize[nas]);
        printf("    imul    rcx\n");
        printf("    add     rbx, rax\n");
    }
```

Array Index

```
    // Add the index of id
    if(entry->isGlobal)
        printf("    mov     rax, global_vars + %d ;  %s\n",
               entry->offset, entry->id);
    else
    {
        printf("    lea     rax, [rbp + %d] ;  %s\n",
               entry->offset, entry->id);
    }

    // Get the value of address if the entry is a reference
    if(entry->type->typeTag == TypeReference)
        printf("    mov     rax, [rax]\n");

    printf("    add     rax, rbx\n");
}
```

Function Definition

```
void SyntaxFunctionDef::Pretranslate(
                             SymTable* symTableBase)
{
    // For the new scope, create a new symbol table
    symTable = SymTable::PushNewTable(symTableBase);

    // Update the BASE symbol table for ID
    idEntry = symTableBase->Insert(id);
    int label = newLabel();
    idEntry->SetLabel(label);

    // Prepare this function's type.
    Type* type = new Type(TypeFunction,
         paramList->ToProductType(), returnType, 0);
    idEntry->SetType(type);
```

Function Definition

```
...
    // Update symbol table for formal params
    // (actual params, return address, bp, local variables)
    int offset = 8 + 8;     // 8 + 8: return address + push rbp
    for(VarList *p = paramList; p; p = p->next)
    {
        SymTableEntry* entry = symTable->Insert(p->id);
        Type *paramType = p->type;
        if(paramType->typeTag == TypeFunction ||
           paramType->typeTag == TypeArray ||
           paramType->typeTag == TypeProduct)
        {
            paramType = new Type(TypeReference, paramType, NULL, 0);
        }
        entry->SetType(paramType);
        entry->SetOffset(offset);
        offset += entry->size;
    }

    // Update symbol table for local variables
    defList->Pretranslate(symTable);
}
```

Function Definition

```
void SyntaxFunctionDef::Translate(SymTable* symTableBase)
{
    int offset = symTable->varSize;
    int label = idEntry->label;

    printf("label_%03d:\n", label);
    printf("    push    rbp\n");         // save and update rbp
    printf("    mov     rbp, rsp\n");
    printf("    sub     rsp, %d\n", offset); // space for local variables
    printf("    push    rbx\n");         // save registers
    printf("    push    rcx\n");
    printf("    push    rdx\n");
    printf("    push    r15\n");         // store for the return value

    bodyStmt->Translate(symTable);       // translate the body
    printf("    mov     rax, r15\n");    // copy the return value to rax

    printf("    pop     r15\n");         // restore registers
    printf("    pop     rdx\n");
    printf("    pop     rcx\n");
    printf("    pop     rbx\n");
    printf("    add     rsp, %d\n", offset);  // reclaim the local var space
    printf("    pop     rbp\n");         // restore rbp
    printf("    ret\n\n");               // return to the caller
}
```

Program Definition

```
void SyntaxProgramDef::Pretranslate(
                                   SymTable* symTableBase)
{
    // For the new scope, create a new symbol table
    symTable = SymTable::PushNewTable(symTableBase);

    // Update the symbol table with ID as a function
    idEntry = symTable->Insert(id);
    idEntry->SetType(new Type(TypeFunction, NULL,
                   new Type(TypeVoid, NULL, NULL, 0), 0));
    int label = newLabel();
    idEntry->SetLabel(label);

    // Output the section before Pretranslate the defList
    printf("    global _start\n\n");
    printf("section .text\n\n");

    // Update symbol table for global variables
    defList->Pretranslate(symTable);
}
```

Program Definition

```
void SyntaxProgramDef::Translate(SymTable* symTableBase)
{
    // global variable size
    int varSize = symTable->varSize;
    int label = idEntry->label;

    printf("_start:\n");
    printf("label_%03d:\n", label);

    bodyStmt->Translate(symTable);

    // exit(0)
    printf("    mov     eax, 60\n");
    printf("    xor     rdi, rdi\n");
    printf("    syscall\n\n");

    // Translate function/procedures
    defList->Translate(symTable);

    printf("section .bss\n\n");
    printf("global_vars:    resb    %d\n\n", varSize);
}
```

Programming Assignment #2

- Modify the compiler code such that
    1. Nested functions are supported
    2. Static scoping rule is used for the variable look up.

```
program nested_function;
    var a : integer;
    procedure foo(b : integer);
        function bar(c : integer) : integer;
        begin
            b := b + c;
            bar := b + c;
        end
    begin
        a := bar(200);
        a := b;
    end
begin
    foo(100);
end
```