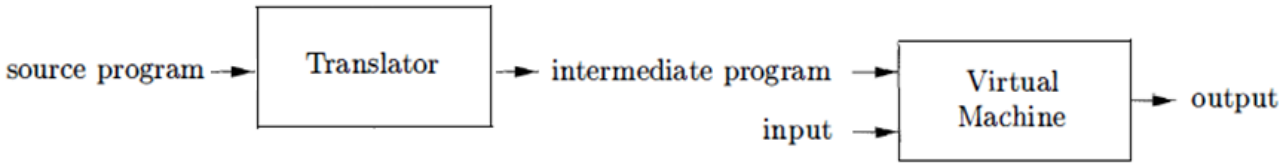
CSE 504 Compiler Design

Abstract Stack Machine, Simple Compiler

YoungMin Kwon

Simple Compiler for an Abstract Stack Machine

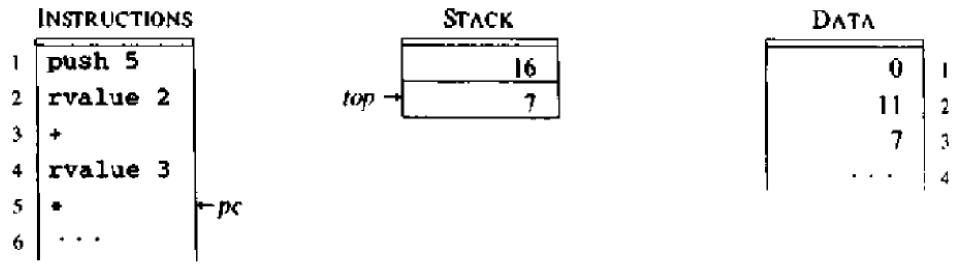
- We will build a Simple Compiler to translate a program into a target program for an abstract stack machine
- It is the hybrid model of the language processors, where the abstract stack machine is the virtual machine.



- We will produce the target machine code using the translation scheme during the semantic analysis phase.

Abstract Stack Machine

- The machine has separate instruction and data memories and all arithmetic operations are performed on values on a stack.
- The picture below shows a snapshot of an abstract stack machine, where pc (program counter) is the instruction to be executed.



- The instructions are arithmetic, stack manipulation, and control flow.

Arithmetic Instructions

- In abstract stack machine, arithmetic expressions simulate the evaluation of the postfix representation.
- Example
  - The following operations can compute 1 3 + 5 *
  - push 1
    push 3
    pop the two elements, add them, and push the result 4
    push 5
    pop the two elements, multiply them, and push the result 20
- +, -, *, / instructions pop two elements from the stack, perform their usual operations, and push the result to the stack.
- Similarly, ==, !=, <=, <, >=, >, and, or, not instructions pop one or two elements from the stack, perform their logical operations and push the result to the stack, where 0 is regarded as false and other values are regarded as true.

L-values and R-values

- There is a distinction between the meaning of the identifies on the left and on the right sides of an assignment.
- Example: i = i + 5 ;
  - The i on LHS means the location where i is stored (l-value).
  - The i on RHS means the value of i (r-value).

Stack Manipulation

- Besides the arithmetic instructions, the stack machine supports the following stack manipulations instructions.
  - push v push v onto the stack
  - rvalue l push the contents of data location l
  - lvalue l push the address of data location l
  - pop throw away the value on top of the stack
  - := the r-value on top is placed in the l-value below it
both elements are poped.
  - copy push a copy of the top value on the stack

Control Flow

- The control-flow instructions for the stack machine are:
  - label l target of jumps to l; has no other effect
  - goto l pc moves to the location where label l is.
  - gofalse l pop a value; jump if it is zero.
  - gotrue l pop a value; jump if it is non-zero.
  - halt stop execution.

Building a Syntax Tree (example data structure)

```
struct SyntaxNode                    /*the base type for the nodes of a syntax tree*/
{
    int tag;                             /*a tag indicating what type of node it is*/
    virtual void Translate() {};
};
struct SyntaxCondStmt                    /*a syntax node for conditional statements*/
    : public SyntaxNode
{
    SyntaxNode* testExpr;                                    /*the test expression*/
    SyntaxNode* thenStmt;                                     /*the then statement*/
    SyntaxNode* elseStmt;                                     /*the else statement*/

    SyntaxCondStmt(SyntaxNode* testExpr, SyntaxNode* thenStmt, SyntaxNode* elseStmt)
        : testExpr(testExpr), thenStmt(thenStmt), elseStmt(elseStmt)
    {
        tag = TagCondStmt; /*indicating that this node is a conditional statement*/
    }

    virtual void Translate();  /*the translate method for a conditional statement*/
};
```

Syntax Tree (Working with Symbol Table)

```
struct SyntaxAssignStmt                    /*a syntax node for the assignment statement*/
    : public SyntaxNode
{
    int indexId;                                          /*for the LHS of Id := expr ;*/
    SyntaxNode* expr;                                     /*for the RHS of Id := expr ;*/

    SyntaxAssignStmt(char*id, SyntaxNode* expr)
        : expr(expr)
    {
        tag = TagAssignStmt;

              /*for now all variables are global. Scoping rules will be handled later*/
        indexId = symTable.Find(id);          /*try to find the id in the symbol table*/
        if(indexId < 0)
        {
            indexId = symTable.Push(id);                  /*if not found add an entry*/
        }
    }

    virtual void Translate();
};
```

Syntax Tree (building from the yacc program)

```
stmt
    : ID ASSIGN expr ';'            { $$ = CreateSyntaxAssignStmt($1, $3); free($1); }
    | IF expr THEN stmt             { $$ = CreateSyntaxCondStmt($2, $4, NULL); }
    | IF expr THEN stmt ELSE stmt   { $$ = CreateSyntaxCondStmt($2, $4, $6); }
    | WHILE expr DO stmt            { $$ = CreateSyntaxWhileStmt($2, $4); }
    | BEGIN_ opt_stmt_list END      { $$ = $2; }
    ;
opt_stmt_list
    : opt_stmt_list stmt            { $$ = CreateSyntaxStmtList($1, $2); }
    |                               { $$ = CreateSyntaxStmtList(NULL, NULL); }
    ;
…
$$ is the value of the LHS of the production.
$1, $2, … are the values of the RHS symbols of the production.

extern "C" /*these wrapper methods are for the interaction between C and C++ */
void* CreateSyntaxAssignStmt(void* id, void* expr)
{
    return (void*) new SyntaxAssignStmt((char*)id, (SyntaxNode*)expr);
}
…
```

Simple Compiler: translation of expressions

```
/*translate to the postfix notation*/
void SyntaxExpr::Translate()
{
    lexpr->Translate();
    if(rexpr)
    {
        rexpr->Translate();
    }
    printf("%c\n", op);
}

void SyntaxId::Translate()
{   /*offset[index] is the location of the Id*/
    printf("rvalue %d\n", symTable.offset[index]);
}

void SyntaxNum::Translate()
{   /*value is the value of the number*/
    printf("push %lf\n", value);
}
```

Example

```
1 + 2 * y
```

The expression above will be translated to

```
push 1
push 2
rvalue y (y's address actually)
*
+
```

Simple Compiler: translation of assignment stmt

```
void SyntaxAssignStmt::Translate()
{                                          /*offset[index] is the location of the Id*/
    printf("lvalue %d\n", symTable.offset[indexId]);       /*lvalue Id instruction*/
    expr->Translate();           /*top of the stack will be the evaluation of expr*/
    printf(":=\n");                                          /*:= instruction*/
}
```

- Translation of `ID := expr ;`
- `expr->Translate()` will print the stack machine codes for expr.
- Once the stack machine codes are executed, it will push the evaluated value of expr to the stack

Example

```
x := 1 + 2 * y ;
```

The expression above will be translated to

```
lvalue x (x's address actually)
push 1
push 2
rvalue y (y's address actually)
*
+
:=
```

Simple Compiler: translation of conditional stmts

```
void SyntaxCondStmt::Translate()
{
    if(elseStmt)
    {
        int labelElse = newLabel();
        int labelExit = newLabel();
        testExpr->Translate();
        printf("gofalse %d\n", labelElse);
        thenStmt->Translate();
        printf("goto %d\n", labelExit);
        printf("label %d\n", labelElse);
        elseStmt->Translate();
        printf("label %d\n", labelExit);
    }
    else
    {
        int labelExit = newLabel();
        testExpr->Translate();
        printf("gofalse %d\n", labelExit);
        thenStmt->Translate();
        printf("label %d\n", labelExit);
   }
}
```

Example

```
if x then y := 1 ;
     else y := 2 ;
```

The code above will be translated to

```
rvalue x
gofalse adrs1
lvalue y
push 1
:=
goto adrs2
label adrs1
lvalue y
push 2
:=
label adrs2
```

Simple Compiler: translation of while stmt

```
void SyntaxWhileStmt::Translate()
{
    int labelTest = newLabel();
    int labelExit = newLabel();
    printf("label %d\n", labelTest);
    testExpr->Translate();
    printf("gofalse %d\n", labelExit);
    stmt->Translate();
    printf("goto %d\n", labelTest);
    printf("label %d\n", labelExit);
}
```

Example

```
while x do
    x := x - 1 ;
```

The code above will be translated to

```
label adrs1
rvalue x
gofalse adrs2
lvalue x
rvalue x
push 1
-
:=
goto adrs1
label adrs2
```

Simple Compiler: translation of optional statements

```
void SyntaxStmtList::Translate()
{
    if(stmt)
    {
        stmt->Translate();
    }
    if(nextStmt)
    {
        nextStmt->Translate();
    }
}
```

Example

```
if x then
  begin
    x := 0 ;
    y := 1 ;
  end
```

The code above will be translated to

```
rvalue x
gofalse adrs1
lvalue x
push 0
:=
lvalue y
push 1
:=
label adrs1
```

Simple Compiler: translation of the program

```
program
    : stmt { Translate($1); } /*once the syntax tree is built, translate the program*/
    ;


extern "C"                              /*for the interaction between C and C++*/
void Translate(void* pgm)               /*like simple.tab.c and SyntaxTree.cpp*/
{
    ((SyntaxNode*)pgm)->Translate();    /*translate the whole program then*/
    printf("halt\n");                   /*print "halt" to stop the execution*/
}
```

Compile with the Simple Compiler

- Compile the simple compiler (sc) you wrote.
- Compile the abstract stack machine (sm) in the course web site.
- Write a sample test program (sourceprogram.txt).
- To generate the translation (using the redirection)
  - .\sc < sourceprogram.txt > targetprogram.txt
- To run the targetprogram.txt on the stack machine
  - .\sm < targetprogram.txt

Programming Assignment 1

- Extend the simple compiler
  1. Add unary minus operator
     - E.g. `x := -1 ; y := 1 + -2 ;`
  2. Add arithmetic comparison operators to the expression
     - `= (equals), <> (not equal), > (greater than), >= (greater than or equal to), < (less than), <= (less than or equal to)`
     - E.g. `if x > 1 then x := x <> 2 ;`
  3. Add Boolean operators
     - `and, or, not`
     - E.g. `if 0 < x and x < 3 then y := x < 2 and x <> 1 ;`
  4. Add `for` statement
     - `FOR ID ASSIGN expr TO expr DO stmt`
     - E.g.
```
for i := 1 to n do
    for j := i to n do
        k := k + 1 ;
```