

Programming Assignment 4

CS 322 Compiler Construction
Winter Quarter 2006

Due: March 10, 11:59pm

–Objective

Generate code (through the evaluation of the `.code` attribute for expressions, statements and function calls without parameters. The generated code will be for the stack machine described in the GIRA manual (provided with the code).

–Before you start

Read this handout and the GIRA manual and study the provided code. Look at the test cases and the standard output and make sure you understand what you need to do.

–Changes in the code from project 3

A new method, `EmitStrings()`, has been added to the `Symbol Table` class. It generates code related to strings and should be called at the very end of `Program::EvalCode()`. If there is a string “blah” in the source code, then this method will generate the line:

```
S_x: .str "blah"
```

where `x` is the `val` attribute of the string constant (i.e. its index in the string table). There should be only one copy per string. See `test.3` for more examples.

To avoid the conflict between the global variable `scopeNumber` and the `IdentInfo` member of the same name, the latter has been changed to `scope`. Modify your `symtab.cpp` accordingly.

New methods have been added to the `IdentInfo` and `TypeDescr` classes to aid in the evaluation of the code attribute for expressions and statements. They are helper methods, and you are not required to use them. You may come up with your own helper functions, as needed.

–Introduction

In project 3, you evaluated the type attributes for variables and parameters and generated code to allocate the `Display`, make the call to the main program, allocate local variables on entry to a `Block` and deallocate them on exit. However, no code was generated for the list of statements in each block. This is what you will do now.

For example, generating code for a statement such as an assignment involves generating code to **load the address of the left hand side**, generating code to **compute the value of right hand side** and then **storing the result** of the right hand side to the address on the left hand side.

In the process, you will perform some minor type checking: Is the type of the expression to the right the same as that to the left? Is the left hand side a valid l-value? Implement structural equivalence. This means that the following program should pass the type checker:

```
program test;
type ptr=^integer;
var x:^integer;
    y:ptr;
begin
    y:=nil;
    x:=y
end.
```

In order to do type checking, expressions must have types. This is the reason why the `EvalCode` methods for expressions have a return type of `TypeDescr*`.

The code generated for some operations depends on the type of the operands. For example, the expression `write(x)` will generate a `WriteInt` instruction if `x` is an integer, a `WriteBool` if it is boolean, etc. Therefore, a method for generating code for a `write` must be defined for each type descriptor. If the method is called on an invalid type, an error message must be printed.

The following sections describe most (but not all) of the translation scheme that generates code for statements, expressions etc. Read the Gira manual and see the provided test cases for more.

.code represents the code attribute, evaluated through the `EvalCode()` functions, .type represents the type attribute that has already been evaluated, .index represents an index in the lexemes array and || represents concatenation.

–Generating code for simple statements

```
statement : expression
{statement.code = expression.code
    || unlink 1}
```

```
statement : variable TASN expression
{statement.code = variable.code
    || expression.code
    || variable.type->AssignCode(expression.type) }
/* AssignCode generates:
    st @
*/
```

```
statement : TWRITELN
{statement.code = sys wln}
```

```
statement : TWRITE ( expression )
{statement.code = expression.code
  || expression.type->WriteCode()}
/* WriteCode generates calls to WriteInt, WriteBool, etc */
```

–Generating code for simple expressions

```
variable : ident
{variable.code = ST::GetInfo(ident.index)->GetAddressCode()}
/* GetAddressCode generates code to obtain the address of ident.
```

ident may be a variable (in this case), or a function name or a parameter.

If it is a local name, then all you need to do is load the value located at the appropriate offset from the frame pointer, #mp.

If it is a non-local name, then you must use the Display to find it.

```
*/
```

```
expression : TUINT
{expression.code = lda <val>}
```

```
expression : variable
{expression.code = variable.code
  || variable.type->VarExprCode()}
/* VarExprCode() generates:
  ld @
*/
```

```
expression : expression1 addop expression2
{expression.code = expression1.code
                || expression2.code
                || expression1.type->DyadicExprCode(addop.val, expression2.type)}
/* addop.val can be VPLUS, VOR, etc.
   DyadicExprCode generates code appropriate to the operator and the types of
   the operands. For example, if the operands are integers and the operator is
   VPLUS, then it will generate "add".

*/
```

```
variable : variable1 TCARET
{variable.code = variable1.code
                || variable1.type->DerefCode()}
/* DerefCode generates:
   ld @
*/
```

```
variable : variable1 [subscript_list]
{variable.code = variable1.code
                || subscript_list.code
                || variable1.type->SubscriptCode()}
/* SubscriptCode generates code to evaluate the subscript */
```

–Generating code for function calls (with no parameters)

```
expression : ident()
{expression.code = <code to allocate return slot>
                || <code to save and grow the display>
                || <code to invoke ident.label>
                || <code to restore the display>}
```

–Submitting your code

Email your files to c22@cs.northwestern.edu