Programming assignment
Code generation

In this assignment you will implement intermediate code generation for part of the MiniP language. The ICode 3-address code described in lecture F10b will be used. You will generate code for procedure allocation and return, assignments, local variable access, and temporary variables. Optionally, you can also implement support for nested procedures, non-local variables, control-flow structures, procedure calls, parameter passing, and nested blocks. You will need all these techniques for the project later on.

There is a demo with a prettyprinter for ICode.

1 Preparations*

Read through this guide. Study the ICode example. Relevant parts of the literature

- Appel: Ch 6-6.1 (Selected parts — see reading directions for lectures on the web).
- Lecture F9 and F10

2 The demo*

The demo contains an abstract grammar for ICode and with a prettyprinter.

The demo has provisions for a target language like MiniP. Since you should not mix the jastadd files for ICode and MiniP there are separate specification directories for the two languages.

3 Intermediate code generation

3.1 Configuration*

It is convenient to use the directory structure from the ICode example as a starting point for your work. It may be useful as starting point for the project too.

3.2 Empty program*

You should start generating intermediate code for an empty program. You may adapt this and the following examples to match your grammar.

```
procedure main()
begin
end;
```

```
main:
       ENTER   vars(0) temps(0)
       RETURN
```

3.3 Allocation of variables

The variables of a procedure should be numbered starting with 0. This number will be used to find the variable inside its activation record. Use a JastAdd aspect to do the numbering and compute the total number of variables in the procedure and use this to generate a correct ENTER instruction. The numbering can be tested in the next task.

```
procedure main()
begin
  integer x;
  integer y;
end;
```

```
main:
        ENTER   vars(2) temps(0)
        RETURN
```

3.4 Accessing local variables

Next you should generate code for assignment statements where just local variables are accessed and the expressions have no operators as in

```
procedure main()
begin
  integer x;
  integer y;
  x = 1;
  y = x;
end;
```

The lecture slides 31–34 from F10 will provide many hints. Since all variables are local you can ignore block levels. All `Variable` objects should have `Levels`=0.

3.5 Expressions with temporary variables

Generate code for expressions that can be a sum of any number of constants and variables. You have to count the number of temporaries and backpatch the ENTER instruction with `Alloc.setTemps` at the end of the procedure. Don't reuse temporaries.

```
procedure main()
begin
  integer x;
  integer y;
  x = 1 + 2;
  y = x + x + 3;
end;
```

When generating temporary variables, it is suitable to start at # within each procedure, and to keep track of the maximum number used in the procedure, so that a suitable `ENTER` instruction can be generated. The `TempFactory` class can be used to generate `Temp` objects and keep track of the number of temporary variables used in the procedure. The number may be accessed using the method `TempFactory.getNumberOfTemps()`. Each temporary variable will be used twice; once when storing a value and once when it is used.

The `ENTER` will be generated before the number of temporaries is known. The instruction may be updated with this number using `Enter.setTemps(int)` provided that you keep a reference to the instruction.

The back patching technique is often used in machine code generation when generating relative forward jumps, i.e., jump instruction that do not jump to an absolute address, but a specified number of instruction further down the code. Then there is a similar problem, as it is not known how many instructions to jump over until after they have been generated. We will only use absolute jump instructions, and will thus not have this kind of back patching problems.)

A temporary value used in one statement can never be used in another statement. It is therefore possible to optimize the use of temporaries by starting with #0 in each statement. This is supported by the method `TempFactory.restartNumbering()`.

N.B! When you build the ICode AST you should make sure that you do not use the same `Temp` object more than once. To represent the a second occurrence of a temporary variable, you should copy the `Temp` object, using `(Temp)temp.copy()` before inserting it into the AST. The reason is that the ICode AST will otherwise not be a tree, but a graph, and it will not work correctly if you traverse the AST upwards, using the `getParent()` operation.

4 Optional tasks

These tasks are not reviewed in this assignment, but will have to be completed in the project.

4.1 Other arithmetic operators

Generate code for your other arithmetic operators. Avoid duplicating code by using the *Template method pattern*, i.e.

- move the common parts to an abstract superclass (create one if needed)
- replace the differing parts with an abstract method in the superclass
- implement it in each subclass.

In the current case `Instruction instruction(Operand, Operand, Address)` is a suitable signature for the method.

For an example see slides 19–22 in `http://fileadmin.cs.lth.se/cs/Education/EDAF10/lectures/F03.pdf`.

4.2 Nested procedures

Suppose that a procedure `q` is nested within another procedure `p`. But the intermediate code for `q` should not be nested inside the intermediate code for `p`. How can this be solved?

You could introduce a method `void procCodeGen(...)` that traverses the program and generates code for each of the procedures. And make sure that the `codeGen(...)` method used for generating the intermediate code for one method should skip entering local methods. When there are nested procedures there is the possibility that several procedures have identical names (within different blocks). But in the intermediate code all the procedures are at the same (global) level, and a need unique labels. There is a need for a naming convention. For example, to prefix all labels with the name of the outer procedure, as illustrated in lecture F10.

4.3 Non-local variables

To access variables and parameters in blocks outside the current block you need to know the number of surrounding blocks up to the declaration of the variable/parameter. One way to compute this is to introduce an attribute `level` that holds the nesting level for the procedure. The outermost block should have level=0 and contain the name of the main procedure. The parameters and the body of the main procedure should have level=1, etc. `Levels` can then be calculated as the difference in level between the procedure holding the declaration and the procedure holding the use.

4.4 Control flow structures

Control flow structures, e.g., loops and if statements, can be implemented using labels and jumps. For these, one needs a mechanism to generate unique labels. One can, for example, use label names `p_0`, `p_1`, ... for labels within a procedure `p`.

A `while` statement is more flexible and easier to implement than a `for` statement.

4.5 Procedure calls

For a procedure call, one also needs to calculate `StaticLevel`, i.e. the relative block level. This can be done in a similar way as for variables.

4.6 Parameter passing

For parameter passing one needs a consistent order of allocation so that calling and called procedures address the parameters in a consistent way. One could for example decide to put the return value first (if existing) and thereafter the parameters in the order of declaration (This order is used in the lectures).

In the project you may chose to use the layout from the lectures or adapt it to the conventions used by the C compiler. In a procedure call the C compiler will push the arguments on the stack in reverse order and push the return address on top of them. You may discuss the options with your project adviser.

4.7 Nested blocks

The following example illustrates nested blocks. How can they be handled?

```
procedure nestedBlocks() {
  integer x;
  if (x < 0) {
    integer y;
    ...
  } else {
    integer z;
    integer t;
    ...
  }
};
```

There are several possibilities. One simple technique is to let the activation record for the procedure hold separate places for all variables declared in the procedure, i.e. in this case 4 variables (x,y,z,t). A more efficient solution is to let different blocks that are not active at the same time share variable space in the frame. Now one only has to allocate space for 3 variables since y cannot be used at the same time as z and t. A third solution is to represent each inner block as an anonymous nested procedure. This variant is also easy to generate code for, but will be less efficient: Procedure calls are more expensive and more accesses will be non-local.

5 Topics for discussion

1. To be added.

References

[1] A. W. Appel with J. Palsberg: *Modern Compiler Implementation in Java*, 2nd Edition, Cambridge University Press, 2002, ISBN: 0-521-82060-X.

[2] L. Andersson, Lecture slides from Lectures F05, `http://cs.lth.se/EDA180`.

[3] G. Hedin: *Quick guide - Scanning in JavaCC*, `http://cs.lth.se/eda180`, located 2009–12–07.

[4] *Apache Ant*, `http://ant.apache.org`, located 2009–12–04.

[5] *Apache Ant Manual*, `http://ant.apache.org/manual/index.html`, located 2009–12–05.

[6] *JastAdd*, `http://jastadd.org`, located 2010–02–05.

[7] *JavaCC*, `https://javacc.dev.java.net`, located 2009–12–04.