Compiler Construction - Assignment 3

LIACS, Leiden University

Fall 2008

Introduction

Again, we will study a subset Pascal compiler in this assignment. This time, you have to translate the syntax tree into an intermediate (flat) form and apply a simple optimization. The intermediate code level is used for "separation of concerns" between the translation of higher-level language constructs to constructs common in lower-level (assembly-like) languages and the translation of lower-level language constructs to the particular assembly language of the target system.

Framework

In your CVS repository, a module named "`assignment3`" is available. This module contains a framework for a subset Pascal compiler. The parsing and syntax tree construction parts are already filled in; you have to add intermediate code generation. A makefile is provided; after performing a checkout of the CVS module, first execute `make first` once. In order to build the entire compiler, simply execute `make`.

Data structures

The framework of this assignment is an extension to the framework of the previous assignment. You should already be familiar with the data structures that are used for syntax tree construction and symbol table management. If not, refer to the documentation of the previous assignment. Several new data structures that are needed during the intermediate code generation phase are already provided. In the next paragraphs, we briefly describe them. Note that you are not allowed to make any changes to any of the provided data structures, unless stated otherwise. Futhermore, you should avoid triggering warning or error messages from the different object methods.

ICGenerator

The `ICGenerator` data structure is meant to handle the intermediate code generation. In contrast to all other classes, you are allowed to modify and extend this class. By default, the class offers three (almost) empty functions: `Preprocess()`, which preprocesses the syntax tree before the intermediate code generator is called; `GenerateIntermediateCode()` which should translate a `SyntaxTree` object into an `IntermediateCode` representation; and `Postprocess()`, which postprocesses the output of the intermediate code generation step. The `Preprocess()` and `Postprocess()` methods are offered as insertion points for optimization passes. For this assignment, the `GenerateIntermediateCode()` function is likely to be your starting point.

IntermediateCode

The `IntermediateCode` data structure is the top-level container of the "flat" intermediate code. Basically, it is a one-dimensional list of `IStatements`. Various operations (`InsertStatement`, `AppendStatement` etc.) are provided to modify this list.

IStatement

The `IStatement` (or Intermediate Statement) data structure represents a single three-address statement, which is stored using a quadruple structure. Hence, an `IStatement` consists of four important fields:

Operator	Operand1	Operand2	Result

- Operator: the operation that has to be performed. This is specified using the `IOperator` enum type which is described below.
- Operand 1 & 2: the source operands. Input for the operation is provided by these fields. For unary operators (like unary minus), the second operand is left empty.
- Result operand: the destination operand. Output of the operation is written to the location specified by this operand.

IOperator

The `IOperator` enumeration type specifies the actual operation of a statement at the intermediate level. The set of operations is fixed, that is, you are not allowed to modify the `IOperator` enum type. The complete list of available operations can be found in the file `IOperator.h`. Usage instructions for each operator are also included. Operator names can be recognized by the `IOP_` prefix.

IOperand

The `IOperand` data structure is used to specify the two source operands and the destination operand of an `IStatement`. There are three different forms of the `IOperand`:

- `IOperand_Int`: the operand is an integer value. Note that this type can only be used as a source operand.
- `IOperand_Real`: the operand is a real (32 bits floating point) value. Note that this type can only be used as a source operand.
- `IOperand_Symbol`: the operand refers to a symbol in the symbol table. In general, when this type is used as a source operand, the value of the memory location identified by the symbol is read and used as input for the operation. When this type is used as a destination operand, the resulting value of the operation is written to the memory location identified by the symbol. However, this operand is also used for other purposes, like the declaration of a variable, label, procedure or function. In these cases, a reference to the symbol is added such that during the next phase the assembly code generator can access the required information.

Note that you can use a specific instance of an `IOperand` only once. If you need to use the same operand somewhere else in your intermediate code, use a copy created with the `Clone()` method.

Some examples

To give you a more concrete idea of how the intermediate code looks like, we provide some examples. Consider the integer assignment `a := b*c + 7*c`. A dump of the corresponding intermediate code could look like the following:

```
#   operator      | operand1          | operand2          | result
------------------+-------------------+-------------------+-------------------
0:  MUL_I         | sym: b            | sym: c            | sym: t1
1:  MUL_I         | int: 7            | sym: c            | sym: t2
2:  ADD_I         | sym: t1           | sym: t2           | sym: t3
3:  ASSIGN_I      | sym: t3           |                   | sym: a
------------------+-------------------+-------------------+-------------------
```

As another example, consider a function `increase` that takes a parameter x, and returns $x + 1$. A dump of the corresponding intermediate code could look like the following:

```
#   operator      | operand1          | operand2          | result
------------------+-------------------+-------------------+-------------------
0:  SUBPROG       | sym: increase     |                   |
1:  ADD_I         | sym: x            | int: 1            | sym: increase
2:  RETURN_I      | sym: increase     |                   |
```

Now we call this function, using the assignment `a := increase(41)`:

```
3:  PARAM_I       | int: 41           |                   |
4:  CALL FUNC     | sym: increase     |                   | sym: a
------------------+-------------------+-------------------+-------------------
```

Assignment

What you have to deliver: a compiler which can parse a given code file and translate this into the intermediate form that is provided by the framework. You should also provide clear documentation about your implementation decisions. This documentation should be placed in your CVS repository in the form of a file called `DOCUMENTATION.TXT`. Furthermore, you have to take a reasonable test case (i.e., a small test case, with recursion, such as recursive Fibonacci number calculation) and explain in detail why the intermediate code generated by your compiler is correct for this case.

You also have to apply a few basic optimizations to either the syntax tree or the intermediate code (you may decide for yourself where to apply them). These optimizations are activated by the commandline option `-O1`. Your compiler has to perform the following tasks when the optimizations are enabled:

- Constant folding: for example, when the expression `32 * 2 + 2` is encountered, replace it by a single constant with value `66`.
- Zero-product: for example, when the expression `a * 0` is encountered, replace it by a single constant with value `0`.
- Algebraic identity elements: for example, when the expression `1 * a + 0` is encountered, replace it by a single constant with value `a`.

In the framework for this assignment, you receive a syntax tree similar to the tree you constructed for the previous assignment. This tree does not contain any cycles. Using the methods associated with the data structures of which the tree is built, you should traverse the tree and generate intermediate code.

Some other notes:

- In this and following assignments, we use call-by-value parameter passing.
- Unlike the previous assignment, you can assume the input for the intermediate code generation phase is correct. This means that you do not have to verify that the syntax tree is correct.

Submission & Grading

Submit your work using CVS. Do not send anything by email. Make sure the latest version of your work has been committed to the CVS repository. Then, tag that version using the command:

```
cvs tag DEADLINE3
```

This tag command has to be issued *before* November 6th, 2008 at 23:59.

For this assignment, 0-10 points can be obtained, which account for 25% of your final grade. If you do not submit your work in time, it will not be graded. The results, once available, can be found on BlackBoard. Your grade for the assignment will not only depend on the functionality of your compiler, but also on the

understandability of your code and the documentation of the design decisions. Documentation should be submitted in English only.

Assistance will be given by Michiel Helvensteijn in room 306/308. A schedule for this can be found on BlackBoard. Additionally, you can go to Sven van Haastregt in room 1.22.