Objectives

- To introduce the concepts involved in intermediate code generation
- Types of intermediate code
- Three address code
- Implementation examples

Intermediate Code Generation (1)

Why?

- We could translate the source program directly into the target language
- However, there are benefits to having an intermediate, machine-independent representation
  - A clear distinction between the machine-independent and machine-dependent parts of the compiler
  - Retargeting is facilitated; the implementation of language processors for new machines will require replacing only the back-end
  - We could apply machine independent code optimisation techniques.

Intermediate Code Generation (2)

Which?

- There are several options for intermediate code
  - Specific to the language being implemented
    - P-code for Pascal
    - Bytecode for Java
  - Language independent:
    - 3-address code (we will examine this in more detail, following Aho et al)
- In all cases, the intermediate code is a linearisation of the syntax tree produced during syntax and semantic analysis
- Formed by breaking down the tree structure into sequential instructions, each of which is equivalent to a single, or small number of machine instructions.
- Machine code can then be generated (access might be required to symbol tables etc)

Three-address code (1)

Introduction

- Three address code contains statements of the form \( x := y \text{ op } z \) [three addresses]
- \( x, y, z \) are names, constants, or compiler-generated temporary variables
- \( \text{op} \) stands for any operator
  - Arithmetic operator
  - Logical operator
  - Only one operator is permitted
- Code can contain symbolic labels, statements for flow of control
Three-address code (2)

Types of statements:
- Assignment statements of the form \( x := y \)\( \text{op} \) \(z\)
- Assignment statements of the form \( x := \text{op} \) \(z\) where \(\text{op}\) is a unary operation (e.g. unary minus, logical negation, shift and convert operators)
- Copy statements of the form \(x := y\)
- Unconditional jumps of the form \(\text{goto} \ L\)
- Conditional jumps of the form \(\text{if} \ x \ \text{relop} \ y \ \text{goto} \ L\)
- \(\text{Param} \ x\) and \(\text{call} \ p.n\) for procedure calls and \(\text{return} \ y\)
- Procedure \(p(x_1,x_2,\ldots,x_n)\) will be translated into \(\text{param} \ x_1\) \(\text{param} \ x_2\).
- \(\text{call} \ p.n\)

Three-address code (3)

Implementation of three-address statements:
- A three-address statement is an abstract form of intermediate code.
- In a compiler, these statements can be implemented as records with fields for the operator and the operands.
- E.g., for the statement \(a := b \times c + b \times c\):

<table>
<thead>
<tr>
<th>Quadroples</th>
<th>Op</th>
<th>Arg1</th>
<th>Arg2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>uminus</td>
<td>c</td>
<td></td>
<td>t1</td>
</tr>
<tr>
<td>(1)</td>
<td>\times</td>
<td>b</td>
<td></td>
<td>t2</td>
</tr>
<tr>
<td>(2)</td>
<td>uminus</td>
<td>c</td>
<td></td>
<td>t3</td>
</tr>
<tr>
<td>(3)</td>
<td>\times</td>
<td>b</td>
<td></td>
<td>t4</td>
</tr>
<tr>
<td>(4)</td>
<td>\times</td>
<td>t2</td>
<td></td>
<td>t5</td>
</tr>
<tr>
<td>(5)</td>
<td>:=</td>
<td>t4</td>
<td></td>
<td>a</td>
</tr>
</tbody>
</table>

Three-address code (2)

Types of statements (2):
- Indexed assignments of the form \(x := y[i]\) and \(x[i] := y\)
- Address and pointer assignments \(x := &y, x := *y\) and \(*x := y;\)
- \(\ldots\)
- Choice of allowable operators a design decision:
  - Small operator set -> easier to port to a new machine
  - but will force the compiler front end to generate long sequences of statements

Three-address code (3)

Implementation of three-address statements (2):
- Alternatively, in order to avoid using temporaries in the symbol table, we can refer to a temporary value by the position of the statement that computes it.
**Three-address code (4)**

Example – Assignments

Example translation scheme to produce three address code for assignments [Aho et al. p479]

\[
S \rightarrow id := E
\]

\[
E \rightarrow E_1 + E_2
\]

\[
E \rightarrow E_1 \times E_2
\]

\[
E \rightarrow - E_1
\]

\[
E \rightarrow ( E_1 )
\]

\[
E \rightarrow id
\]

**Three-address code (5)**

Example (2) – Reusing temporaries (1)

- `newtemp` function returns a new temporary name every time it is called
- Temporaries used to hold intermediate values in expression calculations tend to clutter up the symbol table [space is also needed for them]
- `newtemp` can be adapted to reuse temporaries
- We can determine the lifetime of a temporary from the rules of the grammar, e.g.:
  - \[E \rightarrow E_1 + E_2\]  
    - evaluate \(E_1\) into \(t_1\)
    - evaluate \(E_2\) into \(t_2\)
    - \(t := t_1 + t_2\)
    - \(t_1\) and \(t_2\) are not used after the assignment

Example (2) – Reusing temporaries (2)

- Note that frequently temporaries are used only once – can be reused
- A simple algorithm:
  - Say we have a counter \(c\) initialized to zero
  - Whenever a temporary name is used, decrement \(c\) by 1
  - Whenever a new temporary name is created, use \(c\) and increment \(c\) by 1
- E.g.:
  - \(x := a \times b + c \div d - e^f\)
    - \(a\) incremented by 1
    - \(b\) incremented by 1
    - \(c\) incremented by 1
    - \(d\) incremented by 1
    - \(e\) incremented by 1
    - \(f\) incremented by 1
    - \(x := 0\)  
      - \(c\) incremented once
- What if temporary is used more than once?

Summary

- Intermediate code generation is concerned with the production of a simple machine-independent representation of the source program.
- We saw three-address code as an example of such intermediate code and how structures can be translated into it.

Next lecture:
- Code Generation II

Recommended Reading:
- Chapter 8 of Aho et al. [Highly recommended]
- Chapter 4 of Grune et al.
- Chapter 8 of Hunter’s “The essence of compilers”