Intermediate Code Generation

Reading List:

Aho-Sethi-Ullman:
Chapter 2.3
Chapter 6.1 ~ 6.2
Chapter 6.3 ~ 6.10

(Note: Glance through it only for intuitive understanding.)
Component-Based Approach to Building Compilers

Source program in Language-1

Language-1 Front End

Source program in Language-2

Language-2 Front End

Non-optimized Intermediate Code

Intermediate-code Optimizer

Optimized Intermediate Code

Target-1 Code Generator

Target-1 machine code

Target-2 Code Generator

Target-2 machine code
Intermediate Representation (IR)

A kind of abstract machine language that can express the target machine operations without committing to too much machine details.

Why IR?
Without IR

- C
- Pascal
- FORTRAN
- C++

- SPARC
- HP PA
- x86
- IBM PPC
With IR

C
Pascal
FORTRAN
C++

IR

SPARC
HP PA
x86
IBM PPC
With IR

C
Pascal
FORTRAN
C++

IR → Common Backend

?
Advantages of Using an Intermediate Language

1. **Retargeting** - Build a compiler for a new machine by attaching a new code generator to an existing front-end.

2. **Optimization** - reuse intermediate code optimizers in compilers for different languages and different machines.

**Note**: the terms “intermediate code”, “intermediate language”, and “intermediate representation” are all used interchangeably.
Issues in Designing an IR

- Whether to use an existing IR
  - if target machine architecture is similar
  - if the new language is similar
- Whether the IR is appropriate for the kind of optimizations to be performed
  - e.g. speculation and predication
  - some transformations may take much longer than they would on a different IR
Issues in Designing an IR

- Designing a new IR needs to consider
  - Level (how machine dependent it is)
  - Structure
  - Expressiveness
  - Appropriateness for general and special optimizations
  - Appropriateness for code generation
  - Whether multiple IRs should be used
Multiple-Level IR

- Source Program
- High-level IR
- Low-level IR
- Target code

Semantic Check
High-level Optimization
Low-level Optimization
Using Multiple-level IR

Translating from one level to another in the compilation process

- Preserving an existing technology investment
- Some representations may be more appropriate for a particular task.
Commonly Used IR

Possible IR forms

- Graphical representations: such as syntax trees, AST (Abstract Syntax Trees), DAG
- Postfix notation
- Three address code
- SSA (Static Single Assignment) form

IR should have individual components that describe simple things
DAG Representation

A variant of syntax tree.

Example: \[ \text{D} = ((\text{A}+\text{B}^*\text{C}) + (\text{A}^*\text{B}^*\text{C}))/ -\text{C} \]

DAG: Direct Acyclic Graph
Postfix Notation (PN)

A mathematical notation wherein every operator follows all of its operands.

Examples:

- The PN of expression $9 \times (5+2)$ is $952+*$
- How about $(a+b)/(c-d)$? $ab+cd-/$
Postfix Notation (PN) – Cont’d

Form Rules:

1. If \( E \) is a variable/constant, the PN of \( E \) is \( E \) itself.

2. If \( E \) is an expression of the form \( E_1 \text{ op } E_2 \), the PN of \( E \) is \( E_1 \text{’} E_2 \text{’ op } (E_1 \text{’} \text{ and } E_2 \text{’} \text{ are the PN of } E_1 \text{ and } E_2, \text{ respectively.}) \)

3. If \( E \) is a parenthesized expression of form \( (E_1) \), the PN of \( E \) is the same as the PN of \( E_1 \).
Three-Address Statements

A popular form of intermediate code used in optimizing compilers is three-address statements.

Source statement:

\[ x = a + b \times c + d \]

Three address statements with temporaries \( t_1 \) and \( t_2 \):

\[ t_1 = b \times c \]
\[ t_2 = a + t_1 \]
\[ x = t_2 + d \]
Three Address Code

The general form

\[ x := y \ 	ext{op} \ z \]

\( x, y, \text{and } z \) are names, constants, compiler-generated temporaries

\textbf{op} stands for any operator such as +, -, ...

\( x \times 5 - y \) might be translated as

\[ t1 := x \times 5 \]

\[ t2 := t1 - y \]
Syntax-Directed Translation
Into Three-Address

Temporary

- In general, when generating three-address statements, the compiler has to create new temporary variables (temporaries) as needed.
- We use a function `newtemp()` that returns a new temporary each time it is called.
- Recall Topic-2: when talking about this topic
Syntax-Directed Translation Into Three-Address

- The syntax-directed definition for \( E \) in a production 
  \( \text{id} := \text{E} \) has two attributes:
  1. \textit{E.place} - the location (variable name or offset) that holds the value corresponding to the nonterminal
  2. \textit{E.code} - the sequence of three-address statements representing the code for the nonterminal
Example Syntax-Directed Definition

term ::= ID
  { term.place := ID.place ; term.code = "" }

term₁ ::= term₂ * ID
  {term₁.place := newtemp( );
   term₁.code := term₂.code || ID.code ||*
   gen(term₁.place ':=' term₂.place '*' ID.place}

expr ::= term
  { expr.place := term.place ; expr.code := term.code }

expr₁ ::= expr₂ + term
  { expr₁.place := newtemp( )
    expr₁.code := expr₂.code || term.code ||+
    gen(expr₁.place ':=' expr₂.place '+' term.place
  }

Syntax tree vs. Three address code

Expression: \((A+B*C) + (-B*A) - B\)

Three address code is a linearized representation of a syntax tree (or a DAG) in which explicit names (temporaries) correspond to the interior nodes of the graph.
DAG vs. Three address code

Expression: $D = ((A+B\times C) + (A\times B\times C)) / -C$

Question: Which IR code sequence is better?
Implementation of Three Address Code

**Quadruples**

- Four fields: op, arg1, arg2, result
  - Array of struct `{op, *arg1, *arg2, *result}`
- `x:=y op z` is represented as `op y, z, x`
- `arg1, arg2` and `result` are usually pointers to symbol table entries.
- May need to use many temporary names.
- Many assembly instructions are like quadruple, but `arg1, arg2,` and `result` are real registers.
Implementation of Three Address Code (Con’t)

*Triples*

- Three fields: op, arg1, and arg2. Result is implicit.
- arg1 and arg2 are either pointers to the symbol table or index/pointers to the triple structure.

Example: \( d = a + (b*c) \)

1. \( * \)   \( b, c \)
2. \( + \)   \( a, (1) \)
3. assign \( d, (2) \)

- No explicit temporary names used.
- Need more than one entries for ternary operations such as \( x := y[i], a = b + c, x[i] = y, \) ... etc.
IR Example in Open64 — WHIRL

The Open64 uses a tree-based intermediate representation called WHIRL, which stands for Winning Hierarchical Intermediate Representation Language.
WHIRL

- Abstract syntax tree based
- Symbol table links, map annotations
- Base representation is simple and efficient
- Used through several phases with lowering
- Designed for multiple target architectures
From WHIRL to CGIR
An Example

```c
int *a;
int i;
int aa;

aa = a[i];
```

(a) Source

```
U4U4LDID 0 <2,1,a> T<47,anon_ptr.,4>
U4U4LDID 0 <2,2,i> T<8,.predef_U4,4>
U4INTCONST 4 (0x4)
U4MPY
U4ADD
I4I4LOAD 0 T<4,.predef_I4,4> T<47,anon_ptr.,4>
I4STID 0 <2,3,aa> T<4,.predef_I4,4>
```

(b) Whirl
From WHIRL to CGIR
An Example

(c) WHIRL

(d) CGIR

\[ T_1 = \text{sp + &a}; \]
\[ T_2 = \text{ld } T_1 \]
\[ T_3 = \text{sp + &i}; \]
\[ T_4 = \text{ld } T_3 \]
\[ T_6 = T_4 \ll 2 \]
\[ T_7 = T_6 \]
\[ T_8 = T_2 + T_7 \]
\[ T_9 = \text{ld } T_8 \]
\[ T_{10} = \text{sp + &aa} \]
\[ := \text{st } T_{10} T_9 \]
U4U4LDID 0 <2,1,a> T<47,anon_ptr,.4>
U4U4LDID 0 <2,2,i> T<8,.predef_U4,4>
U4INTCONST 4 (0x4)
U4MPY
U4ADD
I4I4LOAD 0 T<4,.predef_I4,4> T<47,anon_ptr,.4>
I4STID 0 <2,3,aa> T<4,.predef_I4,4>
Differences

- gcc rtl describes more details than whirl
- gcc rtl already assigns variables to stack
- Actually, WHIRL needs other symbol tables to describe the properties of each variable. Separating IR and symbol tables makes WHIRL simpler.
- WHIRL contains multiple levels of program constructs representation, so it has more opportunities for optimization.
Summary of Front End

Lexical Analyzer (Scanner) + Syntax Analyzer (Parser) + Semantic Analyzer

Abstract Syntax Tree w/Attributes

Intermediate-code Generator

Non-optimized Intermediate Code

Error Message

Front End
The Phases of a Compiler

Position := initial + rate * 60

id₁ := id₂ + id₃ * 60

id₁ := id₂ + id₃ * 60

id₁ := id₂ + id₃ * inttoreal (60)

temp1 := inttoreal (60)
temp2 := id₃ * temp1
temp3 := id₂ + temp2
id1 := temp3

id1 := id₂ + temp1

MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1

2/19/2008
Summary

1. Why IR
2. Commonly used IR
3. IRs of Open64 and GCC