Course: Compiler Construction

Week 9: Intermediate Code Generation

Lecturer: Martha Gichuki

Learning outcomes Week 9: Intermediate Code Generation

At the end of the lecture, you will be able to: -

i. Differentiate Source code from intermediate output/code and Target Code
ii. Differentiate High Level and Low-Level Intermediate Representation
iii. Describe the Intermediate Code Generation methods

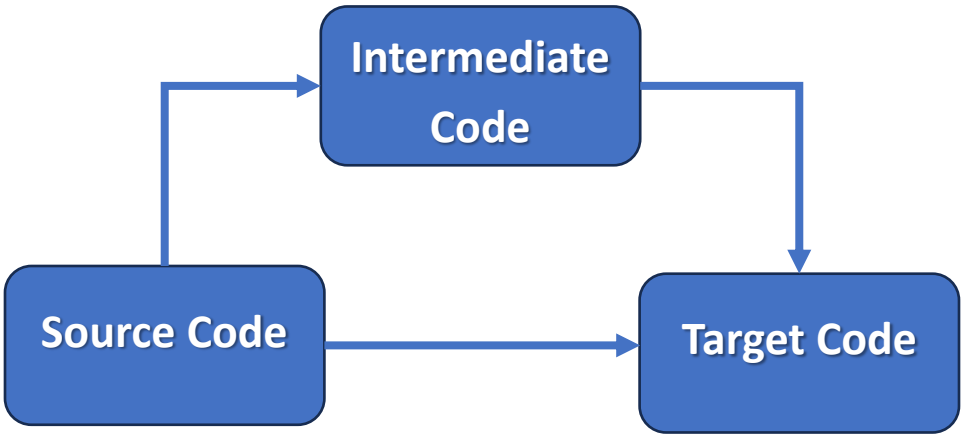
1. Introduction:

1.1. INTERMEDIATE CODE GENERATION

- ✓ Compilers are designed to produce target code after working on some source code.
- ✓ When the input program is represented in some hypothetical language or data structure known as intermediate code an intermediate output is generated[1].

[1] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 382

- ✓ This representation between the source language and the target machine language program is depicted below[2]



- ✓ Generally, *while it is possible to generate machine code directly from source code*, it entails two problems: -

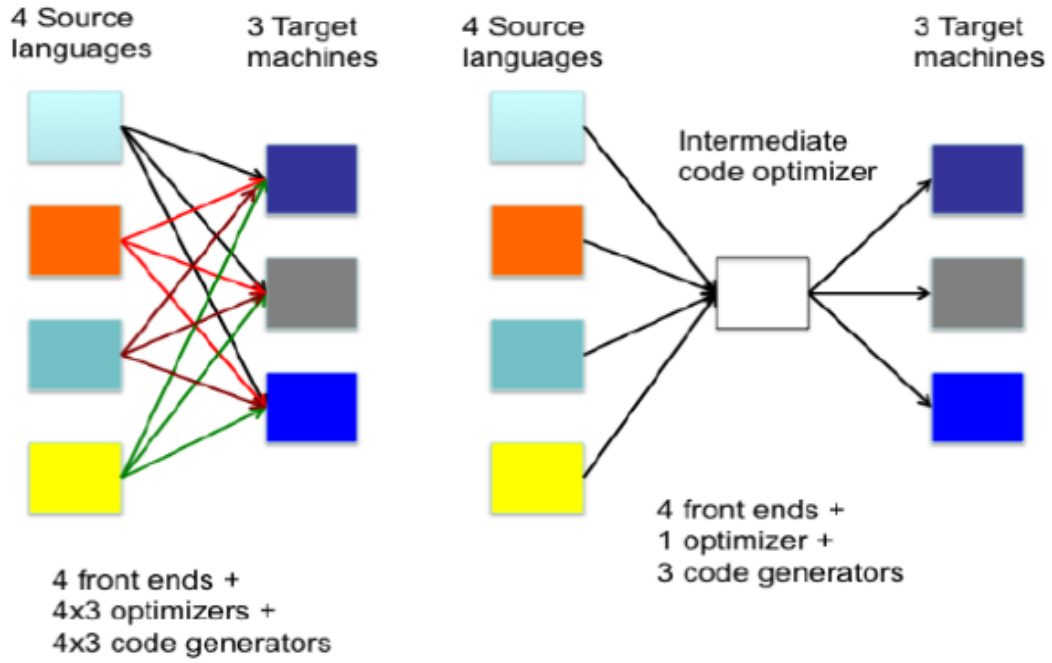
i. With m languages and n target machines, we need to write ***m front ends***, ***m× n optimizers***, and ***m × n code generators***[3]

ii. The code optimizer which is one of the largest and very-difficult-to-write components of a compiler, cannot be reused[4]

- ✓ By converting source code to an intermediate code, a machine-independent code optimizer may be written
- ✓ This means just ***m front ends***, ***n code generators*** and ***1 optimizer***[5]

[2] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 383
[3] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 383
[4] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 383
[5] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 382

Example of a machine-independent code optimizer[6]



1.2. Intermediate Representation (IR)

- ✓ Intermediate codes can be represented in a variety of ways; each of which have their own benefits.
- ✓ In this lecture we look at two ways of representing Intermediate Code[7]

I. High Level IR

II. Low Level IR

I. High Level IR

- ✓ High-level intermediate code representation is *very close to the source language itself*.
- ✓ They can easily be generated from the source code and we can easily apply *code modifications to enhance performance.*
- ✓ But for target machine optimization, *it is less preferred*[8]**.**

---

[6] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 32

[7] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 3

[8] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 116

II. Low Level IR

- ✓ Low lever Intermediate Representation is *close to the target machine*, which makes it suitable for register and memory allocation, inst4uction set selection, etc.
- ✓ It is good for machine-dependency[9].
- ✓ Intermediate code can either be:
  a) Language-specific (e.g., Byte Code for Java) or
  b) Language-independent (three- address code)[10].

1.3. Intermediate Representation Techniques

a) High level representation

- ✓ The following are the various ways of representing the input program in high level intermediate representation[11]:
  i) *Abstract Syntax Trees (AST)*
  ii) *Directed Acyclic Graphs (DAG)*
  iii) *P-code*

i) Abstract Syntax Trees (Syntax Trees)

- ✓ An *abstract syntax tree* is a *compacted form of a parse tree* that represents the hierarchical structure of the program.
- ✓ The *nodes* represent *operators*
- ✓ The *children of a node* represent the *operands* on which the operators operate[12].

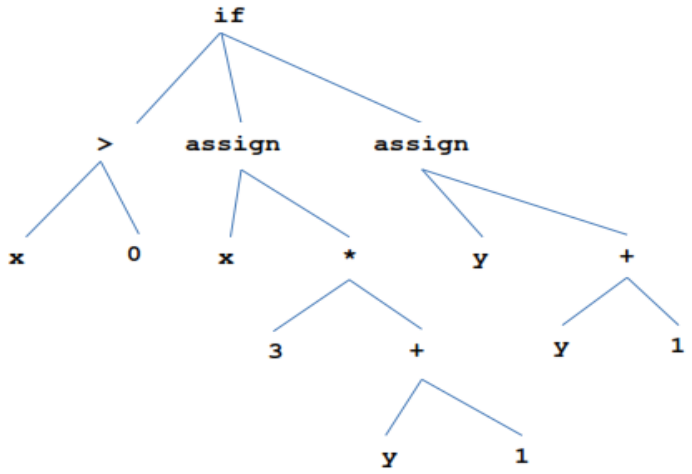
[9] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 117
[10] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 118
[11] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 384
[12] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 384

Example: Consider the following piece of code in the source language[13]:

```
if x>0
        then x=3*(y+1)
                else y=y+1
```

✓ The Abstract Syntax Trees (AST) is as shown below[14]:

if x>0 then x=3*(y+1) else y=y+1



ii) Directed Acyclic Graphs (DAG)

✓ These are similar to AST except that common sub-expressions are represented by a single node[15].

Example: Consider the following piece of code whose DAG is described below[16]

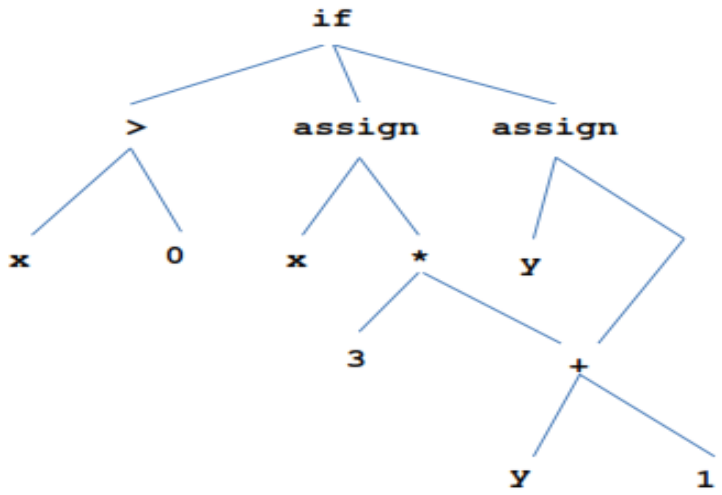
if x>0 then x=3(y+1) else y=y+1*

[13] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 383

[14] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 384. 385

[15] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 386

[16] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 386



iii) P-code

- ✓ This representation is used for stack based virtual machines.
- ✓ In this machine, the operands for the operators are always found on top of the stack[17].

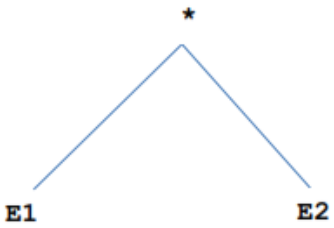
Example:

- ✓ For the following AST, the code is as follows[18]:

Code to evaluate E1

Code to evaluate E2

Mult



[17] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 388

[18] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 388

b) Low Level Representation

i) Three Address Code

- ✓ This is a form of low-level intermediate representation that permits only one operator to the Right-Hand Side (RHS)[19]
- ✓ Low Level Representation Example:

Consider the following source language statement[20]:-

x=y*z+w*a

is translated as follows:-

t1=y*z

t2=w*a

x=t1+t2

2. Type Checking

- ✓ Type checking is verifying that each operation executed in a program respects the type system of the language, i.e., that all operands in any expression are of appropriate types and number.
- ✓ It is used in the following ways:

i. Allow programmers to limit what types may be used in certain circumstances.
ii. Assign type to values.
iii. Determine whether these values are used in an appropriate manner[21].

- ✓ Apart from *verifying the code to be correct*, like checking if a function call has the correct number and types of parameters, type checking also helps in *deciding which code is generated* as in the case of arithmetic expressions.
- ✓ There are many variants of type checking, it may be used to check the type of objects and report a type error in case of violation or the incorrect type may be corrected (coercing)[22].

[19] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 388
[20] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 385
[21] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 395
[22] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 411

2.1. Static Checking and Dynamic Checking

2.1.1. Static Checking

- ✓ In modern languages, type checking is done before execution (compile-time). It is known as static checking since properties are verified before the program is run.
- ✓ Two important checks are: -

i. ***Scope checking*** *- ensures that all variables and functions used within a given scope are declared correctly.*

ii. ***Type checking*** *- ensures that an operator or function is applied to the correct number of arguments of the correct types*[23]

Advantages of Static Checking[24]**:**

a) Can catch many common errors.
b) It is desirable when speed is important i.e. it can result in faster code that does not perform any type checking during execution.

2.1.2. Dynamic checking (run-time checking)

This is performed during program execution.

Advantages of Dynamic Checking:

i) It permits programmers to be less concerned with types

ii) It may be required in some cases like array bound checks which can only be performed during execution.

iii) May generate more robust code by ensuring thorough checking of values for program identifiers during execution[25]

[23] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 122, 382
[24] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 382
[25] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 412

3. Type systems: Strong Typing and Weakly typed Languages

3.1. Strongly typed language

- ✓ Language implementation ensures that whenever an operation is performed, the arguments to the operation are of a type that the operation is defined,
- ✓ **Example**:- do not concatenate a string and a floating-point number. This is independent of whether this is ensured statically (*prior to execution*) or dynamically (*during execution*)[26] .

3.2. Weakly typed language

- ✓ There is no guarantee that operations are performed on arguments that make sense for the operation.
- ✓ The archetypical weakly typed language is machine code where operations are just performed with no checks.

Weakly typed languages are mostly used for system programming, where data needs to be manipulated, moved, copied, encrypted or compressed without regard to what the data represents[27]

	Static	Dynamic
Strong	**Java**	**Lisp**
Weak	**C**	**Perl(1-5)**

4. Type Expressions

- ✓ These are used to represent the type of language constructs.
- ✓ Here, we discuss *SIX different kinds of type expressions*:-

4.1. Basic type which includes: integer, real, character, Boolean, and other atomic types with no internal structures

4.2. Arrays which are specified as array(I,T) where T is a type expression; I is an integer or a range of integers e.g. array(I,T) is an array of type T with T elements. Int a [100] is array type a with 100 elements[28]

---

[26] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 398

[27] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 398

[28] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 371

4.3. If T1 and T2 are two typed expressions, then the **Cartesian product** *T1* ×*T2* is also a type expression. Products denote anonymous records (i.e. field names are absent) and functions argument lists. E.g. functions argument list passed to *function funct* with first argument as *integer*, second as *real* has the associated type as integer × real.

4.4. **Named records** are products, but with named elements e.g. a record structure with two named fields: length, an integer and word which is type array (10, character)[29]

4.5. If T is a type expression, then pointer(T) is a type expression representing objects which are **pointers to objects** of type T e.g. int *p; may be represented as pointer(integer)

4.6. **Functions** map a collection of types to another.

- ✓ They are represented by the type expression ***D*** → ***R,*** where ***D*** is the **domain** and ***R*** is the **range.**
- ✓ ***D*** and ***R*** are ***type expressions***
- ✓ The type ***expression integer*** × ***integer*** → ***character*** represents a function that takes two integers as arguments and returns a character value[30]

[29] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 396

[30] Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 397

Content Covered in Week 9: Intermediate Code Generation

At the end of the lecture, we were able to:

i. Differentiate Source code from intermediate output/code and Target Code
ii. Differentiate High Level and Low-Level Intermediate Representation
iii. Describe the Intermediate Code Generation methods

Course Text Books

1. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007))

2. Compiler Construction: Principles and practices; Kenneth C. Louden; Cengage Learning; 1st edition, ISBN-10: 0534939724 (1997)

3. Basics of Compiler Design: Torben Mogensen; DIKU University of Copenhagen Universitetsparken 1 DK-2100 Copenhagen DENMARK (2007)

4. Engineering a Compiler: 2nd edition; Keith D. Cooper and Linda Torczon; Morgan Kaufmann Publishers ISBN: 978-0-12-088478-0 (2003)

5. Compiler Design: Santanu Chattopadhyay; PHI Learning Publishers, ISBN 812032725X, 9788120327252 (2005)