UCS1602:
COMPILER DESIGN

Introduction to

Intermediate code generation

Session Outcomes

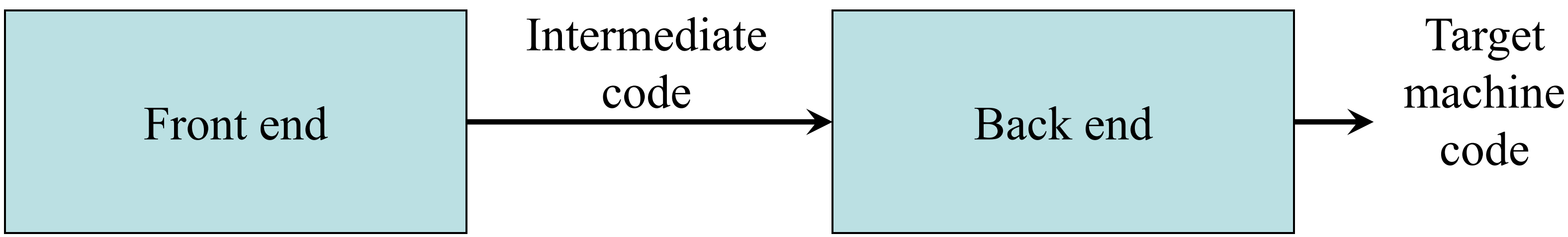
- At the end of this session, participants will be able to
  - Understand the concepts of Intermediate code
  - Study about three address code

Outline

- Intermediate code
- Abstract syntax tree
- Three address code
- Implementation of TAC

Introduction

- *Intermediate codes* are machine independent codes, but they are close to machine instructions.
- Simplifies retargeting of the compiler.
- Allows a variety of optimizations to be implemented in a machine-independent way.



- The given program in a source language is converted to an   equivalent program in an intermediate language by the intermediate code generator.

Introduction Cont...

- Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language.
  - syntax trees can be used as an intermediate language.
  - postfix notation can be used as an intermediate language.
  - three-address code can be used as an intermediate language
    - we will use quadraples to discuss intermediate code generation
    - quadraples are close to machine instructions, but they are not actual machine instructions.
- Some programming languages have well defined intermediate languages.
  - java – java virtual machine
  - prolog – warren abstract machine

Syntax-Directed Translation of Abstract Syntax Trees

Production	Semantic Rule
$S \to$ **id** **:=** E	S.nptr := *mknode*(':=', *mkleaf*(**id**, **id**.entry), E.nptr)
$E \to E_1$ **+** E_2	E.nptr := *mknode*('+', E_1.nptr, E_2.nptr)
$E \to E_1$ ***** E_2	E.nptr := *mknode*('*', E_1.nptr, E_2.nptr)
$E \to$ **-** E_1	E.nptr := *mknode*('uminus', E_1.nptr)
$E \to$ **(** E_1 **)**	E.nptr := E_1.nptr
$E \to$ **id**	E.nptr := *mkleaf*(**id**, **id**.entry)

6

Postfix Notation

a := b * -c + b * -c

a b c uminus * b c uminus * + assign

Postfix notation represents
operations on a stack

Pro: easy to generate
Cons: stack operations are more difficult to optimize

Three-Address Code

a := b * -c + b * -c

t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a := t5

8

Three-Address Statements

Binary Operator: result := y op z

where op is a binary arithmetic or logical operator. This binary operator is applied to y and z, and the result of the operation is stored in result.

Unary Operator: result := op y

where op is a unary arithmetic or logical operator. This unary operator is applied to y, and the result of the operation is stored in result.

Three-Address Statements (cont.)

Move Operator: result := y

where the content of y is copied into result.

Unconditional Jumps: goto L

We will jump to the three-address code with the label L, and the execution continues from that statement.

Three-Address Statements (cont.)

Conditional Jumps: if y ***relop*** z goto L

We will jump to the three-address code with the label L if the result of y relop z is true, and the execution continues from that statement. If the result is false, the execution continues from the statement following this conditional jump statement.

Our relational operator can also be a unary operator.

Three-Address Statements (cont.)

Procedure Parameters: param x

Procedure Calls: call p,n

where x is an actual parameter, we invoke the procedure p with n parameters.

Ex: param x_1

param x_2

$\rightarrow p(x_1,\ldots,x_n)$

param x_n

call p,n

Three-Address Statements (cont.)

Indexed Assignments:

x := y[i]

y[i] := x

Address and Pointer Assignments:

x := &y

x := *y

Implementation of Three-Address Statements

1. Quadruples
2. Triples
3. Indirect triples

Quadruples: A quadruples is a record structure with four fields: OP, arg1,arg2 and result.

OP field contains an internal code for the operator

Implementation of Three-Address Statements: Quadruple

#	*Op*	*Arg1*	*Arg2*	*Res*
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Quads (quadruples)

Pro: easy to rearrange code for global optimization
Cons: lots of temporaries

Implementation of Three-Address Statements: Triples

#	*Op*	*Arg1*	*Arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Triples

Pro: temporaries are implicit
Cons: difficult to rearrange code

16

Implementation of Three-Address Stmts: Indirect Triples

#	*Stmt*
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Program

#	*Op*	*Arg1*	*Arg2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	:=	a	(18)

Triple container

Pro: temporaries are implicit & easier to rearrange code

17

Summary

- Intermediate code
- Abstract syntax tree
- Three address code
- Implementation of TAC

Check your understanding?

1. Translate the expression –(a+b)*(c+d)+(a+b+c) into Quadruples, triple and indirect triple