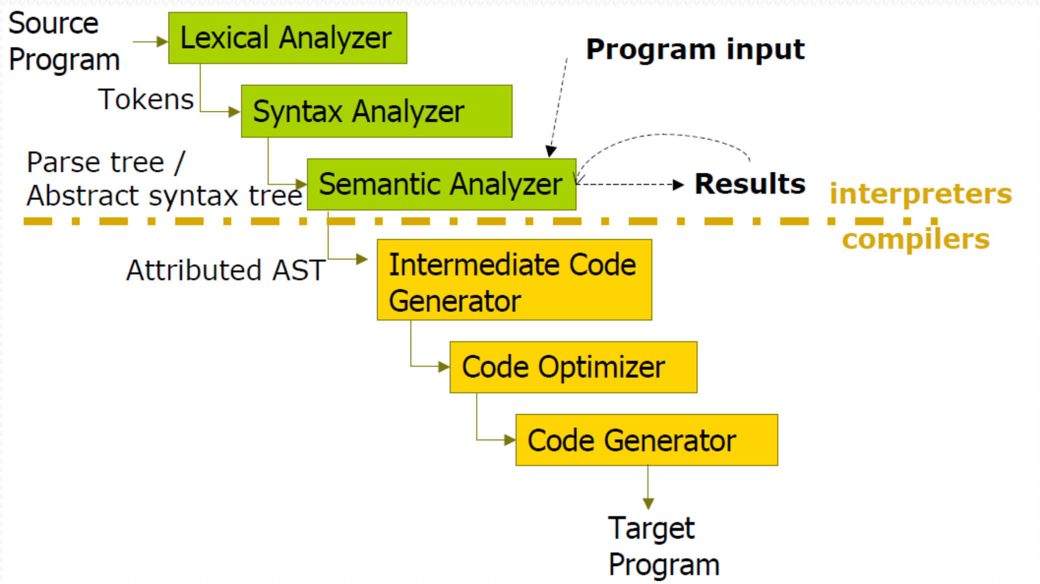
Intermediate Code Generation

Dr. Nguyễn Văn Vinh
UET

Intermediate Code Generation

- ***Intermediate codes*** are machine independent codes, but they are close to machine instructions
- The given program in a source language is converted to an **equivalent program** in an intermediate language by the **intermediate code generator**

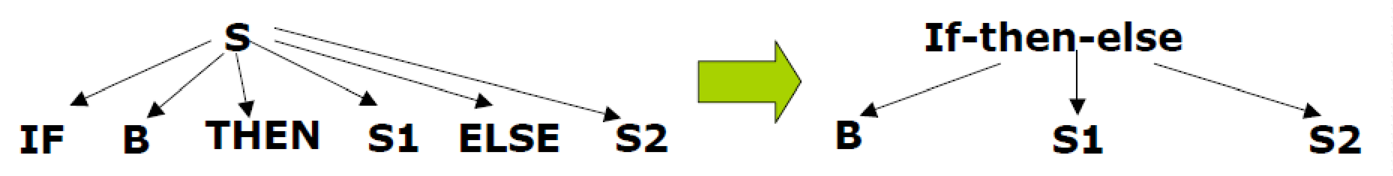


Why do we need intermediate code?

- Convenience for module design
- Machine independence, reducing code generation complexity
- Easier to optimize code

Abstract syntax tree -AST

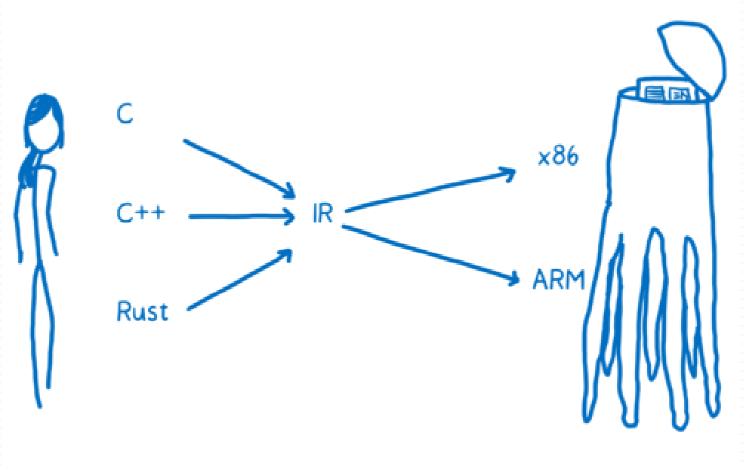
- **Short form of parse tree to represent language constructs**
  - Operators and keywords do not appear as leaves
  - ASTs represent only semantically meaningful aspects of input program



  - Chains of single productions are collapsed

Intermediate language

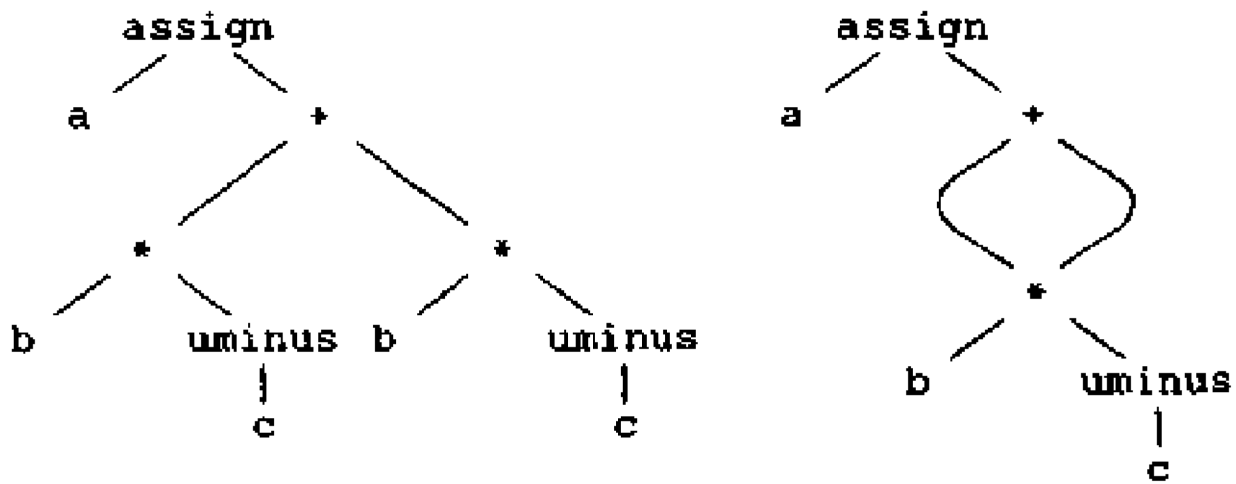
- Goal: Translate AST to low-level machine-independent 3-address IR
- Two alternative ways:
  - Bottom-up tree-walk on AST
  - Syntax-Directed Translation



Intermediate representations

- Graph

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



(a) Syntax tree. (b) Dag.

Intermediate representations (cont)

- Three-address code

  x := y op z

t_1 := − c
t_2 := b * t_1
t_3 := − c
t_4 := b * t_3
t_5 := t_2 + t_4
a := t_5

(a) Code for the syntax tree.

t_1 := − c
t_2 := b * t_1
t_5 := t_2 + t_2
a := t_5

(b) Code for the dag.

Three address statements

- IR (low level) before final code generation
  - Linear representation of AST
  - Every statement that manipulates at most two operands and a result
- Assignment statements $x := y \; op \; z$, where
  - *op: arithmetic or logic operator*
  - *x, y, z*: identifier, constant, or temporary symbol (generated by compilers)
- Assignment statements $x := op \; y$, where
  - *op*: operator
    - minus, logic negation, type casting, bit shift
- Copy statements $x := y$
- Unconditional jump statements *goto L*
  - *L label* locates the next statement

Three address statements (cont)

- Conditional jump statements *if x relop y goto L*.
- Function call statement *param x* and *call p,n*. Return statement *return y*
  - For example, to call *p(x1,x2,...,xn)*

    param x1
    param x2
    . . .
    param xn
    call p, n
- Indexed assignments *x := y[i]* and *x[i] := y*

Three address statements (cont)

- Pointer and address assignments: $x := \&y$, $y := *x$, và $*x := y$

- Note: The choice of operators is important.

Syntax-directed three address code generation

- The synthesized attribute **S.code** represents the code for the production S
- The nonterminal E has attributes:
  - *E.place* is the name that holds the value of E
  - *newtemp* function: generating a temporary variable, assigned to *E.place*.
  - *E.code* is a sequence of three-address statements evaluating E
  - *gen* function: generating a three address statement

Syntax-directed thee address code generation

- For convinience, we use *gen(x ':=' y '+' z)* to represent three address statement $x := y + z$

Arithmetic expressions

Syntax rules	**Semantic rules**
$S \text{ -> } id := E$	
$E \text{ -> } E_1 + E_2$	
$E \text{ -> } E_1 * E_2$	
$E \text{ -> } - E_1$	
$E \text{ -> } (E_1)$	
$E \text{ -> } id$	

Arithmetic expressions

Syntac rules	**Semantic rules**
$S \text{ -> } id := E$	$S.code := E.code \mid\mid gen(id.place \text{ ':=' } E.place)$
$E \text{ -> } E_1 + E_2$	$E.place := newtemp;$ $E.code := E_1.code \mid\mid E_2.code \mid\mid gen(E.place \text{ ':=' } E_1.place \text{ '+' } E_2.place)$
$E \text{ -> } E_1 * E_2$	$E.place := newtemp;$ $E.code := E_1.code \mid\mid E_2.code \mid\mid gen(E.place \text{ ':=' } E_1.place \text{ '*' } E_2.place)$

$E \rightarrow - E_1$	$E.place$:= newtemp; $E.code$:= $E_1.code$ || gen($E.place$ ':=' 'uminus' $E_1.place$)
$E \rightarrow (E_1)$	$E.place := E_1.place$ $E.code := E_1.code$
$E \rightarrow id$	$E.place := id.place$ $E.code$:= ''

Logic expressions

- The major purpose of logic expression
  - Conditional expression
- Logic expression
  - Logic operators
  - Logic variables or relation expressions

Logic expressions (cont)

- Two strategies
  - Encoding true and false as numerical values (for example *true - 1* and *false - 0*) and computing logic expressions similar to arithmetic expressions
  - Representing logic expression value as the position can be reached in the program

Logic expressions (cont)

- Semantics of a programming language decides whether all componets of a logic expression must be computed
  - For example, in *C/C++*, if *E1* is true then it is not necessary to evaluate *E2* in *E1 or E2*

Logic expressions (cont)

- A logic expression E is translated into a sequence of three address statements including jumps to
  - *E.true*: the next statement if E is true
  - *E.false*: the next statement if E is false
- To generate a new label, we use *newlable*
- Example: *a<b*

  if a<b goto E.true

  goto E.false

Logic expressions (cont)

Syntax rules	
$E \text{ -> } E_1 \text{ or } E_2$	
$E \text{ -> } E_1 \text{ and } E_2$	
$E \text{ -> not } E_1$	
$E \text{ -> } (E_1)$	
$E \text{ -> } id_1 \text{ relop } id_2$	
$E \text{ -> true}$	
$E \text{ -> false}$	

Logic expressions (cont)

Syntax rules	Semantic rules
$E \text{ -> } E_1 \text{ or } E_2$	$E_1.true := E.true;$ $E_1.false := newlable;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code \mid\mid gen(E_1.false \text{ ‘:’}) \mid\mid E_2.code$
$E \text{ -> } E_1 \text{ and } E_2$	$E_1.true := newlable;$ $E_1.false := E.false;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code \mid\mid gen(E_1.true \text{ ‘:’}) \mid\mid E_2.code$
$E \text{ -> not } E_1$	$E_1.true := E.false;$ $E_1.false := E.true;$ $E.code := E_1.code;$

Logic expressions (cont)

E -> (E_1)	*E_1.true := E.true;* *E_1.false := E.false;* *E.code := E_1.code;*
E -> id_1 relop id_2	*E.code := gen('if' id_1.place relop.op id_2.place 'goto' E.true) || gen('goto' E.false)*
E -> true	*E.code := gen('goto' E.true)*
E -> false	*E.code := gen('goto' E.false)*

Control statements

- Based on logic expression E to control the action
- *E.true*: the position when *E* is *true*
- *E.false*: the position when *E* is *false*
- *S.begin*: begin position of the statement block
- *S.next*: the position after S's statements

Control statements (cont)

Syntax rules	Semantic rules
$S \text{ -> } if\ E\ then\ S_1$	
$S \text{ -> } if\ E\ then\ S_1\ else\ S_2$	
$S \text{ -> } while\ E\ do\ S_1$	

Control statements (cont)

Syntax rules	Semantic rules
S -> if E then S_1	*E.true := newlable;* *E.false := S.next;* S_1*.next := S.next;* *S.code := E.code || gen(E.true ':') ||* S_1*.code*
S -> if E then S_1 *else* S_2	*E.true := newlable;* *E.false := newlable;* S_1*.next := S.next;* S_2*.next := S.next;* *S.code := E.code || gen(E.true ':') ||* S_1*.code ||* *gen('goto' S.next) || gen(E.false ':') ||* S_2*.code*

Control statements (cont)

S -> while E do S_1	*S.begin := newlable;* *E.true := newlable;* *E.false := S.next* S_1*.next := S.begin;* *S.code := gen(S.begin ':') || E.code || gen(E.true ':') ||* S_1*.code || gen('goto' S.begin)*

Example

Example 1:

```
if a>b then
    a:=a-b;
else
    b:=b-a;
```

Example 2:

```
if a>b and c>d then
    x:=y+z
else
    x:=y-z
```

Example (cont)

```
if a>b goto L1
        goto L2
L1:     t1 := a –b
        a := t1
        goto Lnext
L2:     t2 := b-a
        b := t2
Lnext:
```

```
if a>b  goto L1
goto L3
L1:  if c>d goto L2
     goto L3
L2:  t1 := y+z
     x := t1
     goto L4
L3:  t2 := y-z
     x := t2
L4:
```

Example (cont)

```
while a<>b do
  if a>b then
    a:=a-b
  else
    b:=b-a
```

Example (cont)

```
L1:  if a<>b goto L2
     goto Lnext
L2:  if a>b goto L3
     goto L4
L3:  t1 := a-b
     a := t1
     goto L1
L4:  t2 := b-a
     b := t2
     goto L1
Lnext:
```

Function

```
n = f(a[i]);
```

might translate into the following three-address code:

1) t_1 = i * 4
2) t_2 = a [t_1]
3) param t_2
4) t_3 = call f, 1
5) n = t_3

Exercise

Convert the following statementss or programs into three-address code:

1) *a * - (b+c)*

1) Segment of program in C

   int i; int a[100];

   i = 1;

   while(i ≤ 10)

   {

   a[i] = 0;

   i = i + 1;

   }

Declarations

- There are no corresponding three address statements
- Using symbol table to store
- Intermediate code generation phase results in
  - A set of three address statements
  - Updated symbol table

Declarations (cont)

- For each identifier, we store:
  - type
  - relative address to store value
- So, we can
  - Retrieve the information about type, address to use in statements
  - Replace an identifier by its index in the symbol table
- Note: for an array element, for example *x[i]*, its address is the sum of x's address and i times the length of each element

Declarations (cont)

- Considering an example, with the following conventions:
  - *offset*: storing relative addresses of identifiers
  - An interger number occupies 4 bytes
  - A real number occupies 8 bytes
  - A pointer has a size of 4 byte
  - *enter* function: entering the information about type and relative address for an identifier

Declarations (cont)

Syntax rules	**Semantic rules**
P -> D	*offset := 0*
D -> D ; D	
D -> id : T	*enter(id.name,T.type, offset) ;* *offset := offset + T. width*
T -> interger	*T.type := interger;* *T. width := 4*
T -> real	*T.type := real; T. width := 8*
T -> array [num] of T_1	*T.type := array(num.val,*T_1*.type);* *T.width := num.val ** T_1*. width*
T -> ^T_1	*T.type := pointer(*T_1*.type)* *T. width := 4*

Three address code implementations

- Using records in which
  - Fields representing operators and operations
- Records with four and three fields are enough (***quadruple*** and ***triple***)

Quadruple

- Quadruple is a record structure with four fields *op, arg1, arg2* and *result*.
- Example, $x := y + z$
  - *op* is +, *arg1* is *y*, *arg2* is *z* and *result* is *x*.
- For unary operators, *arg2* is not necessary
- Example, $a := -b * (c+d)$

  $t1 := - b$

  $t2 := c+d$

  $t3 := t1 * t2$

  $a := t3$

  and can be represented by the following quadruple:

Quadruple (cont)

	op	arg1	arg2	result
0	uminus	b		t1
1	+	c	d	t2
2	*	t1	t2	t3
3	assign	t3		a

Triple

- To avoid the use of many temporary names
  - Replacing a temporary name by the position of the statement computing it
  - Such reference is the pointer containing triple of the statement
- We need only three fields *op, arg1,* and *arg2.*

Triple (cont)

	op	arg1	arg2
0	uminus	b	
1	+	c	d
2	*	(0)	(1)
3	assign	a	(2)

Triple (cont)

- Note, *x[i] := y* needs two records:

	op	arg1	arg2
(0) (1)	[]= assign	x (0)	i y

- Similarly: *x := y[i]*

	op	arg1	arg2
(0) (1)	[]= assign	y x	i (0)