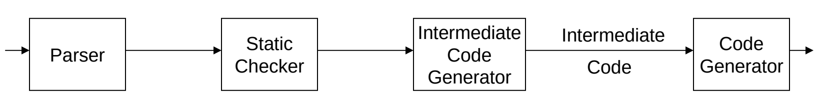
Intermediate Code

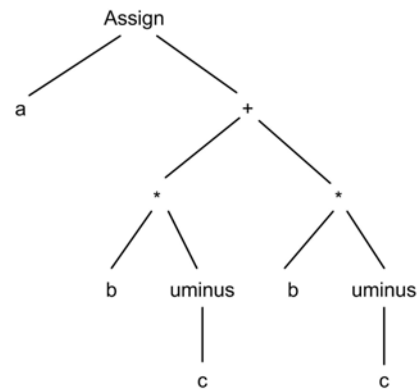
Intermediate Code Generation

- The front end of a compiler translates a source program into an intermediate representation
- Details of the back end are left to the back end
- Benefits include:
  - Retargeting
  - Machine-independent code optimization



Intermediate Languages

- Consider the code:
  a := b * -c + b * -c
- A *syntax tree* graphically depicts code
- *Postfix notation* is a linearized representation of a syntax tree:
  a b c uminus * b c uminus * + assign



Syntax Tree

Three-Address Code

- Three-address code is a sequence of statements of the general form x := y op z
- x, y, and z are names, constants, or compiler-generated temporaries
- op can be any operator
- Three-address code is a linearized representation of a syntax tree
- Explicit names correspond to interior nodes of the graph

Three-Address Code Example

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


Types of Three-Address Statements

1. Assignment statements:
   a. x := y op z, where op is a binary operator
   b. x := op y, where op is a unary operator
2. Copy statements
   a. x := y
3. The unconditional jumps:
   a. goto L
4. Conditional jumps:
   a. if x relop y goto L
5. param x and call p, n and return y relating to procedure calls
6. Assignments:
   a. x := y[i]
   b. x[i] := y
7. Address and pointer assignments:
   a. x := &y, x := *y, and *x = y

Generating Three-Address Code

- Temporary names are made up for the interior nodes of a syntax tree
- The synthesized attribute S.code represents the code for the assignment S
- The nonterminal E has attributes:
  - E.place is the name that holds the value of E
  - E.code is a sequence of three-address statements evaluating E
- The function newtemp returns a sequence of distinct names
- The function newlabel returns a sequence of distinct labels

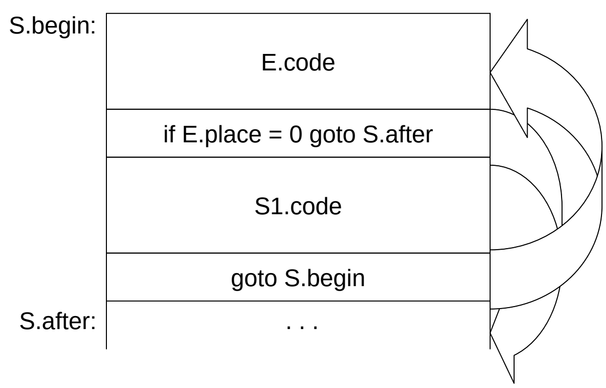
Assignments

Production	Semantic Rules
S → **id** := E	S.code := E.code || gen(**id**.place ':=' E.place)
E → E_1 + E_2	E.place := newtemp; E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '+' E_2.place)
E → E_1 * E_2	E.place := newtemp; E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '*' E_2.place)

Assignments

Production	Semantic Rules
E → -E_1	E.place := newtemp; E.code := E_1.code || gen(E.place ':=' 'uminus' E_1.place)
E → (E_1)	E.place := E_1.place; E.code := E_1.code
E → **id**	E.place := **id**.place; E.code := ''

While Statement



Example: S → while E do S_1

```
S.begin := newlabel;
S.after  := newlabel;
S.code  := gen(S.begin ':') ||
             E.code ||
             gen('if' E.place '=' '0' 'goto' S.after) ||
             S1.code ||
             gen('goto' S.begin) ||
             gen(S.after ':')
```


Quadruples

- A quadruple is a record structure with four fields: op, arg_1, arg_2, and result
  - The op field contains an internal code for an operator
  - Statements with unary operators do not use arg_2
  - Operators like param use neither arg_2 nor result
  - The target label for conditional and unconditional jumps are in result
- The contents of fields arg_1, arg_2, and result are typically pointers to symbol table entries
  - If so, temporaries must be entered into the symbol table as they are created
  - Obviously, constants need to be handled differently

Quadruples Example

	op	arg1	arg2	result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

Triples

- Triples refer to a temporary value by the position of the statement that computes it
  - Statements can be represented by a record with only three fields: op, arg_1, and arg_2
  - Avoids the need to enter temporary names into the symbol table
- Contents of arg_1 and arg_2:
  - Pointer into symbol table (for programmer defined names)
  - Pointer into triple structure (for temporaries)
  - Of course, still need to handle constants differently

Triples Example

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

Declarations

- A symbol table entry is created for every declared name
- Information includes name, type, relative address of storage, etc.
- Relative address consists of an offset:
  - Offset is from the base of the static data area for globals
  - Offset is from the field for local data in an activation record for locals to procedures
- Types are assigned attributes type and width (size)
- Becomes more complex if we need to deal with nested procedures or records

Declarations

Production	Semantic Rules
P → D	offset := 0
D → D ; D	
D → id : T	enter(id.name, T.type, offset); offset := offset + T.width
T → integer	T.type := integer; T.width := 4
T → real	T.type := real T.width := 8
T → array[num] of T_1	T.type := array(num, T_1.type); T.width := num * T_1.width
T → ↑T_1	T.type := pointer(T_1.type); T.width := 4

Translating Assignments

Production	Semantic Rules
S → id := E	p := lookup(id.name); if p != NULL then emit(p ':=' E.place) else error
E → E_1 + E_2	E.place := newtemp; emit(E.place ':=' E_1.place '+' E_2.place)
E → E_1 * E_2	E.place := newtemp; emit(E.place ':=' E_1.place '*' E_2.place)

Translating Assignments

Production	Semantic Rules
E → -E_1	E.place := newtemp; emit(E.place ':=' 'uminus' E_1.place)
E → (E_1)	E.place := E_1.place
E → id	p := lookup(id.name); if p != NULL then E.place := p else error

Addressing Array Elements

- The location of the i^{th} element of array A is:

  base + (i – low) * w

  – w is the width of each element
  – low is the lower bound of the subscript
  – base is the relative address of A[low]

- The expression for the location can be rewritten as: i * w + (base – low * w)
  – The subexpression in parentheses is a constant
  – That subexpression can be evaluated at compile time

Two-Dimensional Arrays

- Stored in row-major form
- The address of $A[i_1,i_2]$ is:

  $$base+((i_1-low_1)\times n_2+i_2-low_2)\times w$$

  – Where $n_2 = high_2 - low_2 + 1$
- We can rewrite the above as:

  $$((i_1\times n_2)+i_2)\times w+(base-((low_1\times n_2)+low_2)\times w)$$

  – The last term can be computed at compile time

Type Conversions

- There are multiple types (e.g. integer, real) for variables and constants
  – Compiler may need to reject certain mixed-type operations
  – At times, a compiler needs to general type conversion instructions
- An attribute E.type holds the type of an expression

Semantic Action: $E \rightarrow E_1 + E_2$

E.place := newtemp;
if E_1.type = integer and E_2.type = integer then
begin
  emit(E.place ':=' E_1.place 'int+' E_2.place);
  E.type := integer
end
else if E_1.type = real and E2.type = real then
  …
else if E_1.type = integer and E_2.type = real then
begin
  u := newtemp;
  emit(u ':=' 'inttoreal' E_1.place);
  emit(E.place ':=' u 'real+' E_2.place);
  E.type := real
end
else if E_1.type = real and E_2.type = integer then
  …
else E.type := type_error;

Example: x := y + i * j

- In this example, x and y have type real
- i and j have type integer
- The intermediate code is shown below:

  t_1 := i int* j
  t_3 := inttoreal t_1
  t_2 := y real+ t_3
  x := t_2

Boolean Expressions

- Boolean expressions compute logical values
- Often used with flow-of-control statements
- Methods of translating boolean expression:
  - Numerical methods:
    - True is represented as 1 and false is represented as 0
    - Nonzero values are considered true and zero values are considered false
  - Flow-of-control methods:
    - Represent the value of a boolean by the position reached in a program
    - Often not necessary to evaluate entire expression

Numerical Representation

- Expressions evaluated left to right using 1 to denote true and 0 to donate false
- Example: a or b and not c

  t_1 := not c
  t_2 := b and t_1
  t_3 := a or t_2
- Another example: a < b

```
100:  if a < b goto 103
101:  t : = 0
102:  goto 104
103:  t : = 1
104:  …
```

Numerical Representation

Production	Semantic Rules
E → E_1 or E_2	E.place := newtemp; emit(E.place ':=' E_1.place 'or' E_2.place)
E → E_1 and E_2	E.place := newtemp; emit(E.place ':=' E_1.place 'and' E_2.place)
E → not E_1	E.place := newtemp; emit(E.place ':=' 'not' E_1.place)
E → (E_1)	E.place := E1.place;

Numerical Representation

Production	Semantic Rules

E → id_1 relop id_2	E.place := newtemp;
	emit('if' id_1.place relop.op id_2.place 'goto' nextstat+3);
	emit(E.place ':=' '0');
	emit('goto' nextstat+2);
	emit(E.place ':=' '1');

E → true	E.place := newtemp; emit(E.place ':=' '1')
E → false	E.place := newtemp; emit(E.place ':=' '0')

Example: $a<b$ or $c<d$ and $e<f$

100: if $a < b$ goto 103
101: $t_1 := 0$
102: goto 104
103: $t_1 := 1$
104: if $c < d$ goto 107
105: $t_2 := 0$
106: goto 108
107: $t_2 := 1$
108: if $e < f$ goto 111
109: $t_3 := 0$
110: goto 112
111: $t_3 := 1$
112: $t_4 := t_2$ and t_3
113: $t_5 := t_1$ or t_4

slt t_1, a, b
slt t_2, c, d
slt t_3, e, f
and t_4, t_2, t_3
or t_5, t_1, t_4

MIPS code

Flow-of-Control

- The function newlabel will return a new symbolic label each time it is called
- Each boolean expression will have two new attributes:
  - E.true is the label to which control flows if E is true
  - E.false is the label to which control flows if E is false
- Attribute S.next of a statement S:
  - Inherited attribute whose value is the label attached to the first instruction to be executed after the code for S
  - Used to avoid jumps to jumps

Flow-of-Control

Production	Semantic Rules
S → if E then S_1	E.true := newlabel; E.false := S.next; S_1.next := S.next; S.code := E.code || gen(E.true ':') || S_1.code

Flow-of-Control

Production	Semantic Rules
S → if E then S_1 else S_2	E.true := newlabel; E.false := newlabel; 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

Flow-of-Control

Production	Semantic Rules
S → while E do S_1	S.begin := newlabel; E.true := newlabel; E.false := S.next; S1.next := S.begin; S.code := gen(S.begin ':') || E.code || gen(E.true ':') || S_1.code || gen('goto' S.begin)

Boolean Expressions Revisited

Production	Semantic Rules
E → E_1 or E_2	E_1.true := E.true; E_1.false := newlabel; E_2.true := E.true; E_2.false := E.false; E.code := E_1.code || gen(E_1.false ':') || E_2.code
E → E_1 and E_2	E_1.true := newlabel; E_1.false := E.false; E_2.true := E.true; E_2.false := E.false; E.code := E_1.code || gen(E_1.true ':') || E_2.code

Boolean Expressions Revisited

Production	Semantic Rules
E → not E_1	E_1.true := E.false; E_1.false := E.true; E.code := E_1.code
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.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)

Revisited: a<b or c<d and e<f

```
       if a < b goto Ltrue
       goto L1
L1:    if c < d goto L2
       goto Lfalse
L2:    if e < f goto Ltrue
       goto Lfalse
```

- The code generated is inefficient
- What is the problem?
  - Why was the code generated that way?

Another Example

```
while a < b do
  if c < d then
    x := y + z
  else
    x := y - z
```

⇒

```
L1:     if a < b goto L2
        goto Lnext
L2:     if c < d goto L3
        goto L4
L3:     t1 := y + z
        x:= t1
        goto L1
L4:     t2 := y – z
        x := t2
        goto L1
Lnext:
```


Mixed-Mode Expressions

- Boolean expressions often have arithmetic subexpressions, e.g. (a + b) < c
- If false has the value 0 and true has the value 1
  - arithmetic expressions can have boolean subexpressions
  - Example: (a < b) + (b < a) has value 0 if a and b are equal and 1 otherwise
- Some operators may require both operands to be boolean
- Other operators may take both types of arguments, including mixed arguments

Revisited: $E \rightarrow E_1 + E_2$

```
E.type := arith;
if E1.type = arith and E2.type = arith then
begin
  /* normal arithmetic add */
  E.place := newtemp;
  E.code := E1.code || E2.code ||
    gen(E.place ':=' E1.place '+' E2.place)
end
else if E1.type := arith and E2.type = bool then
begin
  E2.place := newtemp;
  E2.true := newlabel;
  E2.flase := newlabel;
  E.code := E1.code || E2.code ||
    gen(E2.true ':' E.place ':=' E1.place + 1) ||
    gen('goto' nextstat+1) ||
    gen(E2.false ':' E.place ':=' E1.place)
else if …
```


Case (Switch) Statements

- Implemented as:
  – Sequence of if statements
  – Jump table

Labels and Goto Statements

- The definition of a label is treated as a declaration of the label
- Labels are typically entered into the symbol table
  - Entry is created the first time the label is seen
  - This may be before the definition of the label if it is the target of any forward goto
- When a compiler encounters a goto statement:
  - It must ensure that there is exactly one appropriate label in the current scope
  - If so, it must generate the appropriate code; otherwise, an error should be indicated

Procedures

- The procedure is an extremely important, very commonly used construct
- Imperative that a compiler generates good calls and returns
- Much of the support for procedure calls is provided by a run-time support package

S → call id (Elist)
Elist → Elist, E
Elist → E

Calling Sequence

- Calling sequences can differ for different implementations of the same language
- Certain actions typically take place:
  - Space must be allocated for the activation record of the called procedure
  - The passed arguments must be evaluated and made available to the called procedure
  - Environment pointers must be established to enable the called procedure to access appropriate data
  - The state of the calling procedure must be saved
  - The return address must be stored in a known place
  - An appropriate jump statement must be generated

Return Statements

- Several actions must also take place when a procedure terminates
  - If the called procedure is a function, the result must be stored in a known place
  - The activation record of the calling procedure must be restored
  - A jump to the calling procedure's return address must be generated
- No exact division of run-time tasks between the calling and called procedure

Pass by Reference

- The param statements can be used as placeholders for arguments
- The called procedure is passed a pointer to the first of the param statements
- Any argument can by obtained by using the proper offset from the base pointer
- Arguments other than simple names:
  - First generate three-address statements needed to evaluate these arguments
  - Follow this by a list of param three-address statements

Using a Queue

Production	Semantic Rules
S → call id (Elist)	for each item p on queue do emit('param' p); emit('call' id.place)
Elist → Elist, E	push E.place to queue
Elist → E	initialize queue to contain E

- The code to evaluate arguments is emitted first, followed by param statements and then a call
- If desired, could augment rules to count the number of parameters