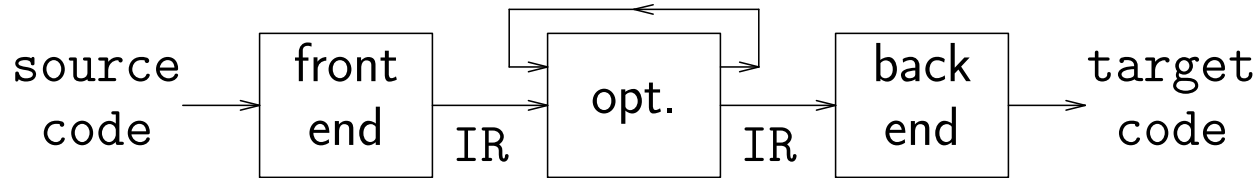


Code generation



Code generation steps

1. Source code \rightarrow intermediate representation

- generate intermediate representation during parse based on syntax, symbol tables

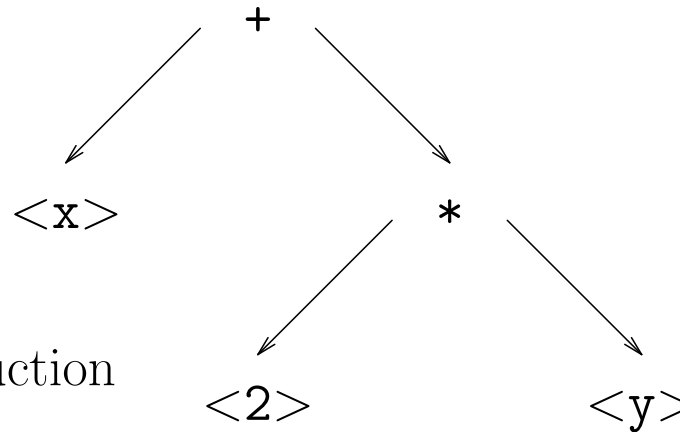
2. Intermediate representation \rightarrow target code

- instruction selection
choose instructions based on target instr set
- memory management
decide storage for variables; allocate registers
- linkage code
determine prolog/epilog code
- instruction scheduling
choose instruction execution order

Abstract syntax trees

Abstract syntax tree (AST)

- stores syntactic structure of program



Building the AST

- construct node for LHS of production
- fill in fields using RHS of production

Example

E_0	$::=$	E_1	'+'	E_2	{	$E_0.val = \text{node}('+', E_1.val, E_2.val);$	}
		E_1	'*'	E_2	{	$E_0.val = \text{node}('*', E_1.val, E_2.val);$	}
		id			{	$E_0.val = \text{node}(\mathbf{id}.val);$	}
		num			{	$E_0.val = \text{node}(\mathbf{num}.val);$	}

Instruction selection

Code templates

- template for each language construct
- ignore surrounding context
- simple recursive approach

Language constructs

1. simple expressions
2. control structures
3. procedure calls
4. complex expressions

Applying templates

- use syntax during parse
- apply tree rewriting to AST

Intermediate representation

We'll be targeting RISC-like processors

- load-store architecture
- register-transfer language
- three-address code
- explicit loads and stores

Examples

load r1, <addr>	\$ r1 ← value at <addr>
loadi r1, <const>	\$ r1 ← value of <const>
store r1, <addr>	\$ <addr> ← r1
move r1, r2	\$ r1 ← r2
add r1, r2, r3	\$ r1 ← r2 + r3
sub r1, r2, r3	\$ r1 ← r2 - r3
mult r1, r2, r3	\$ r1 ← r2 * r3
jmp <addr>	\$ jump to <addr>

Simple expressions

Expression trees:

- adopt a simple treewalk scheme
- assign a virtual register to each operator
- emit code in postorder walk

Support routines:

- `addr(str)` — returns the name of a virtual register that contains the base address for `str`
- `newtemp()` — returns a new virtual register name

Assume:

- assume tree reflects precedence, associativity
- assume all operands are integers

Simple expressions

```
expr( node )
  int result, t1, t2, t3;
  switch( type of node )

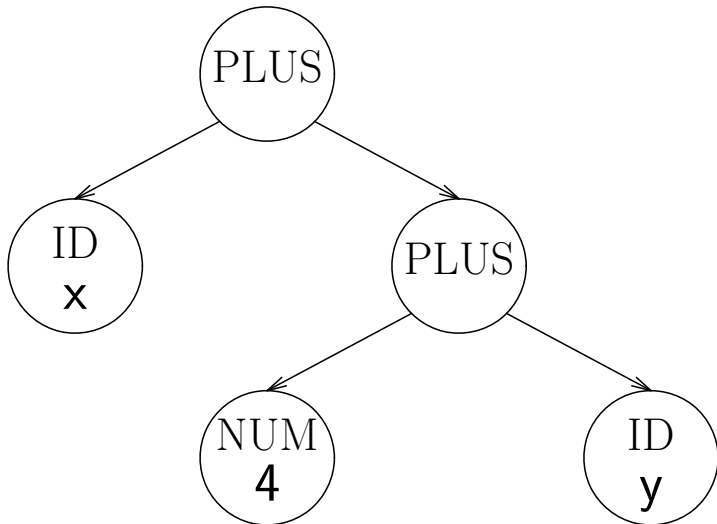
    case PLUS:
      t1 = expr( left child of node );
      t2 = expr( right child of node );
      result = newtemp();
      emit( add, result, t1, t2 );
      break;

    case ID:
      t3 = addr( node.str );
      result = newtemp();
      emit( load, result, t3 );
      break;

    case NUM:
      result = newtemp();
      emit( loadi, result, node.val );
      break;

  return result
```

Simple expressions



load r2, r1	\$ r2 ← addr(x)
loadi r3, 4	\$ constant
load r5, r4	\$ r5 ← addr(y)
add r6, r3, r5	\$ r6 ← 4 + y
add r7, r2, r6	\$ r7 ← x + (4 + y)

Control structures

Assignment statement

$lhs \leftarrow rhs$

Strategy

- evaluate rhs to a value (*an rvalue*)
- evaluate lhs to an address (*an lvalue*)
 - i) *lvalue* is register \Rightarrow **move** it
 - ii) *lvalue* is address \Rightarrow **store** it

Registers versus memory

- non-aliased scalars \Rightarrow can go in a register
- aggregate or potentially aliased \Rightarrow in memory

Control structures

if-then-else

1. evaluate the expression to **true** or **false**
2. if **true**, fall through to **then** part
branch around **else** part
3. if **false**, branch to **else** part
fall through to next statement

Example

<code>r1 ← expr</code>	<i>evaluate the expression</i>
<code>if not(r1) br L1</code>	<i>compare and branch</i>
<code>...</code>	<i>stmts for then part</i>
<code>br L2</code>	<i>branch to exit</i>
<code>L1: ...</code>	<i>stmts for else part</i>
<code>L2: ...</code>	<i>following stmt</i>

Control structures

while loop

1. evaluate the control expression
2. if **false**, branch beyond end of loop
if **true**, fall through into loop body
3. at end, re-evaluate the control expression
4. if **true**, branch to top of loop body
if **false**, fall through

Example

<code>r1 ← expr</code>		<i>evaluate the expression</i>
<code>if not(r1) br L2</code>		<i>compare and branch</i>
<code>L1: ...</code>		<i>loop body</i>
<code> r1 ← expr</code>		
<code> if r1, br L1</code>		
<code>L2: ...</code>		<i>following stmt</i>

Test at end ⇒ simple loop is one block

Control structures

case statement

1. evaluate the controlling expression
2. branch to the selected case
3. execute its code
4. branch to the following statement

Key Issue:

⇒ *finding the right case*

<i>Method</i>	<i>When</i>	<i>Cost</i>
<i>linear search</i>	few cases	$\mathbf{O}(\text{cases})$
<i>binary search</i>	sparse	$\mathbf{O}(\log_2(\text{cases}))$
<i>jump table</i>	dense	$\mathbf{O}(1)$

Boolean expressions

Most languages include boolean expressions.

Sample Grammar

```
<expr> ::= <expr> or <expr>
          | <expr> and <expr>
          | not <expr>
          | ( <expr> )
          | id <relop> id
          | true
          | false
```

```
<relop> ::= <
           | ≤
           | =
           | ≠
           | ≥
           | >
```

Used for logical values and to alter control flow

Relational versus logical operators

Boolean expressions

Two schools of thought on representation:

Numerical Values

- assign numerical values to true and false
- evaluate booleans like arithmetic expressions

Control Flow

- represent boolean value by location in code
- convert to numerical value when stored

Neither representation dominates the other.

Boolean expressions

Numerical Values

- assign a value to true (say 1)
- assign a value to false (say 0)
- use hardware — `and`, `or`, `not`, `xor`

Choose values that make the hardware work.

Boolean expressions

Numerical Values

<i>Source Expression</i>	<i>Generated Code</i>
$b \text{ or } (c \text{ and not } d)$	$t1 \leftarrow \text{not } d$ $t2 \leftarrow c \text{ and } t1$ $t3 \leftarrow b \text{ or } t2$
$a < b$	$\text{if } (a < b) \text{ br L1}$ $t1 \leftarrow 0$ br L2 L1: $t1 \leftarrow 1$ L2: nop

A numerical representation handles logic well.

Boolean expressions

Control Flow

- use conditional branches and comparator
- chain of branches to evaluate expression
- code looks terrible

Control flow representation works well for expressions in conditional statements.

Clean up:

- branch to next statement
- branch to branch

Boolean expressions

Control Flow

<i>Source Expression</i>	<i>Generated Code</i>
$a < b \text{ or } (c < d \text{ and } e < f)$	if a<b br LT br L1 L1: if c<d br L2 br LF L2: if e<f br LT br LF LF: <i>code under false</i> or t1 ← false br LEXIT LT: <i>code under true</i> or t1 ← true br LEXIT LEXIT:

This works well when the expression's value is tested but not preserved for later reuse.

Boolean expressions

What about "short circuiting"

Do the semantics require evaluating all terms of an expression?

- once value established, stop evaluating
- (`true or <expr>`) is `true`
- (`false and <expr>`) is `false`
- save cycles in evaluation

Order of evaluation

- if specified, must be observed
- if not, reorder by cost and short-circuit

Boolean expressions

Reality

- either approach works fairly well
- numerical code reflects logical constructs
- control flow code works well for relations
- compiler can choose based on context

Control flow

- accounting nightmare — tracking labels
- backpatching is the right answer