Generation of Intermediate Code

Chapter 1, Section 1.2.4

Chapter 2, Section 2.8

Chapter 5, Section 5.1, 5.2, 5.3

Chapter 6, Section 6.1, 6.2, 6.4, 6.6

Outline

Program representations

- Abstract syntax trees (ASTs)
- Expression DAGs
- Three-address code

Translation (to three-address code) of

- Expressions
- Flow-of-control statements

Projects 4 & 5: translate an AST to three-address code

Abstract Syntax Trees (ASTs)

The Dragon Book calls them just "syntax trees"

- As opposed to "concrete syntax trees" = "parse trees"
- Each node represents a language construct
- Children represent the sub-constructs

Example: $E \rightarrow E$ **+** T

- Parse tree: node E with three children
- AST: **+** node with two children
- Example: Parse tree and AST for **1 + a * (2 + b) * 3**

  $E \rightarrow E$ **+** T | T

  $T \rightarrow T$ ***** F | F

  $F \rightarrow$ **(** E **)** | **const** | **id**

AST Construction

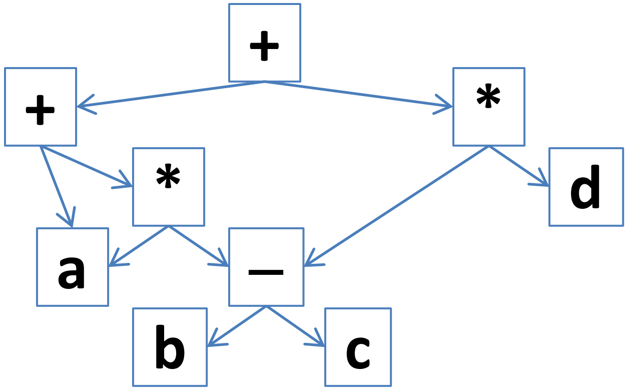
$E \rightarrow E_1 + T$	$E.node = newNode(+, E_1.node, T.node)$
$E \rightarrow T$	$E.node = T.node$
$T \rightarrow T_1 * F$	$T.node = newNode(*, T_1.node, F.node)$
$T \rightarrow F$	$T.node = F.node$
$F \rightarrow (E)$	$F.node = E.node$
$F \rightarrow$ **const**	$F.node = newLeaf($**const**, **const**.*lexval*)
$F \rightarrow$ **id**	$F.node = newLeaf($**id**, **id**.*lexval*)

AST construction can be done **during** parsing (no parse tree built) or **after** it (first build parse tree, then AST)

Expression DAGs

Directed acyclic graph: common sub-expressions are not replicated

- Example: **a + a * (b – c) + (b – c) * d**



Use a similar attribute grammar but reuse nodes

- *newNode(op, left, right)* checks if there already exists a node with label *op*, and children *left* and *right*; returns this node if it already exists
- *newLeaf* is modified in a similar way

Another Representation: Three-Address Code

AST is a high-level IR

- Close to the source language
- Suitable for tasks such as type checking

Three-address code is a lower-level IR

- Closer to the target language (i.e., assembly code)
- Suitable for tasks such as code generation/optimization

Basic ideas

- A small number of simple instructions: e.g. **x = y op z**
- A number of compiler-generated temporary variables

  a = b + c + d; in source code → **t = b + c; a = t + d;**
- Simple flow of control – conditional and unconditional jumps to labeled statements (no **while-do**, **switch**, …)

Addresses and Instructions

"Address": a program variable, a constant, or a compiler-generated temporary variable

Instructions

- **x = y op z**: binary operator **op**; y and z are variables, temporaries, or constants; x is a variable or a temporary
- **x = op y**: unary operator **op**; y is a variable, a temporary, or a constant; x is a variable or a temporary
- **x = y**: copy instruction; y is a variable, a temporary, or a constant; x is a variable or a temporary
- More later: arrays, flow-of-control
- Each instruction contains at most three "addresses"
  - Thus, three-address code

Translation of Expressions: Toy Example

A simple grammar for assignments and expressions

- Ambiguous, but it doesn't matter – parsing is finished

$S \rightarrow$ **id =** E **;**

$E \rightarrow E_1$ **+** E_2

$E \rightarrow$ **-** E_1

$E \rightarrow$ **id**

Two attributes

- Synthesized attribute *code* for S and E: sequence of three-address instructions
- Synthesized attribute *addr* for E: the "address" (program variable or temp or const) that will hold the value of E

Toy Example: Translation

$S \rightarrow$ **id = E ;**

$S.code = E.code \,||\, \textbf{id}.lexval$ "**=**" $E.addr$ || is concatenation

$E \rightarrow E_1$ **+** E_2

$E.addr = newTemp()$

$E.code = E_1.code \,||\, E_2.code \,||$

$E.addr$ "**=**" $E_1.addr$ "**+**" $E_2.addr$

$E \rightarrow$ **-** E_1

$E.addr = newTemp()$

$E.code = E_1.code \,||\, E.addr$ "**=**" "**-**" $E_1.addr$

$E \rightarrow$ **id**

$E.addr = \textbf{id}.lexval$ $E.code =$ " "

Examples of Code Generation

x = y; produces three-address instruction **x = y;**

In a real compiler, **x** and **y** are pointers to rows in the symbol table; here we will pretend they are just strings (provided by **id**.*lexval*)

x = - y; produces **t1 = - y**; **x = t1**;

x = y + z; produces **t1 = y + z**; **x = t1**;

x = y + z + w; produces **t1 = y + z**; **t2 = t1 + w**; **x = t2**;

x = y + - z; produces **t1 = - z**; **t2 = y + t1**; **x = t2**;

More Complex Expressions & Assignments

All binary & unary operators are handled similarly

We run into more interesting issues with

- Expressions that have side effects
- Arrays

Example: $E \rightarrow \ldots \mid E_1$ **=** $E_2 \mid E_1$ **+=** $E_2 \mid$ **id [** E_1 **]**

- In C, we can write **x = y = z + z**: maybe it should be translated to **t1 = z + z**; **y = t1**; **x = t1**; ?
- How should we translate **x = y += w**? How about **a[v = x += 1] = y = z += 2 + w**? How about …

Language Features for Project 4

Will only consider expression statements and return statements

$S \rightarrow E$ **;** | **return** E **;**

$E \rightarrow$ **id** | **intconst** | **doubleconst**

$E \rightarrow$ **id [**E_1**]** (discuss 1-dim arrays; implement multi-dim arrays)

$E \rightarrow E_1$**+**E_2 | E_1**==**E_2 | … (+,-,*,/,%,==,!=,<,<=,>,>=)

$E \rightarrow E_1$ **=** E_2 | E_1 **+=** E_2 | … (=,+=,-=,*=,/=,%=)

L-values of Expressions

An expression *E* has an l-value if this expression can appear on the left-hand-side of an assignment

- The type of an l-value is always "a chunk of memory"
- E.g. **x** is an **int** variable
  - the *value* (called *r-value*) of expr **x** is some integer
  - the *l-value* of expression **x** is the "chunk of memory" (typically, 4 bytes) in which the integer resides

L-values: only for $E \rightarrow$ **id** | **id[**E_1**]**

The semantic analyzer guarantees that the left operand of an assignment operator has an l-value

- i.e. Project 3 has done the checking successfully

Modified Grammar for Project 4

$E \rightarrow E_1$ **=** E_2 | E_1 **+=** E_2 | …

becomes

$E \rightarrow$ **id =** E_1 | **id +=** E_1 | **id[**E_1**] =** E_2 | **id[**E_1**] +=** E_2 | …

Semantics of assignment operators

- **id =** E_1: result value is the new value of **id**
- **id +=** E_1 is equivalent to **(id = id +** E_1**)**
- **id[**E_1**] =** E_2: evaluate E_1 and E_2 (in some unspecified order); modify the array element; result is the new value
- **id[**E_1**] +=** E_2 is equivalent to **(id[**E_1**] = id[**E_1**] +** E_2**)**, except that the evaluation of E_1 happens only once

Translation

$S \rightarrow E$ **;**

$S.code = E.code$

$E \rightarrow E_1 + E_2$ (and similar binary operators -,*,/,%)

$E.addr = newTemp()$ and $E.code = E_1.code \,||\, E_2.code \,||$

$E.addr$ "**=**" $E_1.addr$ "**+**" $E_2.addr$ *But C semantics defines no order*

$E \rightarrow E_1 < E_2$ (and similar binary operators <=,>,>=,==,!=)

$E.addr = newTemp()$ and $E.code = E_1.code \,||\, E_2.code \,||$

$E.addr$ "**=**" $E_1.addr$ “**<**" $E_2.addr$ *But C semantics defines no order*

$E \rightarrow$ **id**

$E.addr =$ **id**.$lexval$ $E.code =$ " "

Note: for the project, we will assume **<**, **>**, etc. produce integer values: 0 is false, not 0 is true (C semantics)

Translation

$E \rightarrow$ **intconst**

$E.addr$ = **intconst**.*lexval* and $E.code$ = " " (same for **doubleconst**)

$E \rightarrow$ **id[**E_1**]**

$E.addr = newTemp()$

$E.code = E_1.code \;||\; E.addr$ "**=**" **id**.*lexval* "**[**" $E_1.addr$ "**]**"

- Here we use **x = y[z]** instructions in the three-address code
  - y is an array-typed variable
  - z is a variable, a temporary, or a constant
  - x is a variable or a temporary
- Multi-dim arrays: **x = y[u][v]…[w]**
  - In real compilers, need to use several instructions

Translation

$E \rightarrow$ **id =** E_1

$E.addr = E_1.addr$ *Here we* ***do not*** *need a new temp*

$E.code = E_1.code$ || **id**.*lexval* "**=**" $E_1.addr$

$E \rightarrow$ **id[**E_1**] =** E_2

$E.addr = E_2.addr$ *Here we* ***do not*** *need a new temp*

$E.code = E_2.code$ || $E_1.code$ || *But C semantics defines no order*

id.*lexval* "**[**" $E_1.addr$ "**]**" "**=**" $E_2.addr$

- Here we use **x[y] = z** instructions
  - x is an array variable
  - y and z are variables, temporaries, or constants

Example

```
int a[10][20];
int x; int y; int z;
x = 1;
y = 2;
z = 3;
a[y-x][y+x] = z + 2*y;
```

```
int a[10][20];
int x; int y; int z;
int _t1; int _t2; int _t3;
int _t4; int _t5;
x = 1;
y = 2;
z = 3;
_t4 = 2 * y;
_t5 = z + _t4;
_t1 = y - x;
_t2 = y + x;
a[_t1][_t2] = _t5;
```

Example

```
int x; int y; int z; int w;
w = z = (x = 1) + (y = x+2);
```

```
int x; int y; int z; int w;
int _t1; int _t2;
x = 1;
_t1 = x + 2;
y = _t1;
_t2 = 1 + _t1;
z = _t2;
w = _t2;
```

Translation

$E \rightarrow$ **id +=** E_1

Treat this exactly as **id = id +** E_1 (i.e., combination of the rules for $E \rightarrow E_1$ **+** E_2 and $E \rightarrow$ **id =** E_1)

E.addr = *newTemp*() *Here we* ***do*** *need a new temp*

E.code = E_1.*code* || *E.addr* "**=**" **id**.*lexval* "**+**" E_1.*addr* ||
id.*lexval* "**=**" *E.addr*

Example

```
int x; int y; int z;
x = 1;
z = (x += 1) + (y += x+2);
```

```
int x; int y; int z;
int _t1; int _t2;
int _t3; int _t4;
x = 1;
_t1 = x + 1;
x = _t1;
_t2 = x + 2;
_t3 = y + _t2;
y = _t3;
_t4 = _t1 + _t3;
z = _t4;
```

Translation

$E \rightarrow \textbf{id}[E_1] \mathrel{\textbf{+=}} E_2$

$E.addr = newTemp()$

$E.code = E_1.code \;||$

$E.addr$ "**=**" **id**.*lexval* "**[**" $E_1.addr$ "**]**" ||

$E_2.code$ || $E.addr$ "**=**" $E.addr$ "**+**" $E_2.addr$ ||

id.*lexval* "**[**" $E_1.addr$ "**]**" "**=**" $E.addr$

Example

```
int a[10][20];
int x; int y; int z;
x = 1;
y = 2;
z = 3;
a[y-x][y+x] += z + 2*y;
```

```
int a[10][20];
int x; int y; int z;
int _t1; int _t2; int _t3;
int _t4; int _t5; int _t6;
x = 1; y = 2; z = 3;
_t1 = y - x;
_t2 = y + x;
_t6 = a[_t1][_t2];
_t4 = 2 * y;
_t5 = z + _t4;
_t6 = _t6 + _t5;
a[_t1][_t2] = _t6;
```

A Few Examples to Try at Home

```
x = y+z;

w = x = y+z;

a[x=y+z] = x;

a[x] = x = y+z;

x += y+z;

x += x = y+z;

a[v = x += 1] = y = z += 2 + w;
```

Flow of Control – Expressions & Statements

Boolean expressions – in C, any expression of scalar type (in our subset of C, any int/double expr)

- Role 1: conditions of ifs and loops
- Role 2: assign to a variable

$E \rightarrow E_1$ **<** E_2 | … <, <=, ==, !=, >, >=

$S \rightarrow E$ **;** | **return** E**;** | **;**

$S \rightarrow$ **if (**E**)** S_1 | **if (**E**)** S_1 **else** S_2

$S \rightarrow$ **while (**E**)** S_1 | **for (**E_1**;** E_2 **;** E_3**)** S_1

$S \rightarrow$ **{** S_1 … S_n **}** *similarly for the whole program*

Three-Address Instructions

New instructions

- **goto L**: unconditional jump to the three-address instruction with label L
- **if (address) goto L**: the address contains a “boolean” value
- But: for convenience we will use **if (!address) goto L**: jump if the address contains the *false* “boolean” value

The labels are symbolic names

- We will just generate label names L1, L2, … using a helper function *newLabel*(), in the same way we generate temporaries with names t1, t2, … using a helper function *newTemp*()

Translation

$S \rightarrow E$ **;**

$S.code = E.code$

$S \rightarrow$ **{** $S_1 \ldots S_n$ **}** *similarly for the whole program*

$S.code = S_1.code \mid\mid \ldots \mid\mid S_n.code$

$S \rightarrow$ **if (**E**)** S_1

$S.exitLabel = newLabel()$

$S.code =$

$E.code \mid\mid$

"**if (!** " $E.addr$ "**) goto** " $S.exitLabel \mid\mid$

$S_1.code \mid\mid$

$S.exitLabel$

Example

```
int x; int y; int z;
x = 1;
y = 2;
z = 3;
if (x+y > z) z=x+y;
```

```
int x; int y; int z;
int _t1; int _t2; int _t3;
x = 1; y = 2; z = 3;
_t1 = x + y;
_t2 = _t1 > z;
if (!_t2) goto _l1;
_t3 = x + y;
z = _t3;
_l1:
;
```

Translation

$S \rightarrow$ **if (** E **)** S_1 **else** S_2

$S.exitLabel = newLabel()$

$S.elseLabel = newLabel()$

$S.code =$

$E.code$ ||

"**if (!** " $E.addr$ "**) goto** " $S.elseLabel$ ||

$S_1.code$ ||

"**goto** " $S.exitLabel$ ||

$S.elseLabel$ ||

$S_2.code$ ||

$S.exitLabel$

Example

```
int x; int y; int z;
x = 1;
y = 2;
z = 3;
if (x+y > z) z=x+y;
else         z=x-y;
```

```
int x; int y; int z;
int _t1; int _t2; int _t3; int_t4;
x = 1; y = 2; z = 3;
_t1 = x + y;
_t2 = _t1 > z;
if (!_t2) goto _l2;
 _t3 = x + y;
  z = _t3;
  goto _l1;
  _l2:
  _t4 = x - y;
  z = _t4;
  _l1:
  ;
```

Translation

$S \rightarrow$ **while (**E**)** S_1

$S.startLabel = newLabel()$

$S.exitLabel = newLabel()$

$S.code =$

$S.startLabel$ ||

$E.code$ ||

"**if (!** " $E.addr$ "**) goto** " $S.exitLabel$ ||

$S_1.code$ ||

"**goto** " $S.startLabel$ ||

$S.exitLabel$

Example

```
int n; int i; int res;
n = 10;
i = 1;
res = 1;
while (i <= n) {
  res *= i;
  i += 1;
}
```

```
int n; int i; int res;
int _t1; int _t2; int _t3;
n = 10; i = 1; res = 1;
_l1:
_t1 = i <= n;
if (!_t1) goto _l2;
_t2 = res * i;
res = _t2;
_t3 = i + 1;
i = _t3;
goto _l1;
_l2:
;
```