Compiler Development (CMPSC 401)

Three Address Code

Janyl Jumadinova

April 2, 2019
High-level assembly where each operation has at most three operands (e.g., an add instruction has three operands: two for each argument and one for the result).
Three Address Code (TAC)

- High-level assembly where each operation has at most three operands (e.g., an add instruction has three operands: two for each argument and one for the result).
- The operands are called *addresses* and can be held in a *virtual register*.
- The compiler back-end will convert TAC instructions and registers into specific machine instructions and hardware registers.
Temporary Variables

- The “three” in “three-address code” refers to the number of operands in any instruction.
- Evaluating an expression with more than three subexpressions requires the introduction of temporary variables.
int x;
int y;

int x2 = x * x;
int y2 = y * y;
int r2 = x2 + y2;
int x;
in y;

int x2 = x * x;
int y2 = y * y;
int r2 = x2 + y2;

x2 = x * x;
y2 = y * y;
r2 = x2 + y2;
```c
int a;
int b;
int c;
int d;

a = b + c + d;
b = a * a + b * b;
```
```c
int a;
int b;
int c;
int d;

a = b + c + d;
b = a * a + b * b;
```

```c
_t0 = b + c;
a = _t0 + d;
_t1 = a * a;
_t2 = b * b;
b = _t1 + _t2;
```
int a;
int b;
a = 5 + 2 * b;
TAC allows for instructions with two operands.

```c
int a;
int b;

a = 5 + 2 * b;
```

```c
_t0 = 5;
_t1 = 2 * b;
a = _t0 + _t1;
```
Variable assignment allows assignments of the form:

\[ \text{var} = \text{constant}; \]
Simple TAC Instructions

- Variable assignment allows assignments of the form:
  
  \[
  \begin{align*}
  var &= constant; \\
  var_1 &= var_2;
  \end{align*}
  \]
Simple TAC Instructions

- Variable assignment allows assignments of the form:
  
  \begin{align*}
  var & = \text{constant}; \\
  var_1 & = var_2; \\
  var_1 & = var_2 \text{ op } var_3;
  \end{align*}

  \text{Permitted operators are} +, -, *, /, \%.

How would you compile $y = -x$?

$y = 0 - x$; $y = -1 * x$;
Simple TAC Instructions

- Variable assignment allows assignments of the form:

  \[ \text{var} = \text{constant}; \]
  \[ \text{var}_1 = \text{var}_2; \]
  \[ \text{var}_1 = \text{var}_2 \text{ op } \text{var}_3; \]
  \[ \text{var}_1 = \text{constant op } \text{var}_2; \]

Permitted operators are +, -, *, /, %.
Variable assignment allows assignments of the form:

\[ var = constant; \]
\[ var_1 = var_2; \]
\[ var_1 = var_2 \text{ op } var_3; \]
\[ var_1 = constant \text{ op } var_2; \]
\[ var_1 = var_2 \text{ op } constant; \]
Simple TAC Instructions

- Variable assignment allows assignments of the form:
  
  \[
  \text{var} = \text{constant}; \\
  \text{var}_1 = \text{var}_2; \\
  \text{var}_1 = \text{var}_2 \text{ op} \text{ var}_3; \\
  \text{var}_1 = \text{constant} \text{ op} \text{ var}_2; \\
  \text{var}_1 = \text{var}_2 \text{ op} \text{ constant}; \\
  \text{var} = \text{constant}_1 \text{ op} \text{ constant}_2; \\
  \]

- Permitted operators are +, -, *, /, %.
Simple TAC Instructions

- Variable assignment allows assignments of the form:
  
  $\text{var} = \text{constant};$
  
  $\text{var}_1 = \text{var}_2;$
  
  $\text{var}_1 = \text{var}_2 \text{ op } \text{var}_3;$
  
  $\text{var}_1 = \text{constant op } \text{var}_2;$
  
  $\text{var}_1 = \text{var}_2 \text{ op } \text{constant};$
  
  $\text{var} = \text{constant}_1 \text{ op } \text{constant}_2;$

- Permitted operators are $+, -, *, /, \%$.

- How would you compile $y = -x;$?
Simple TAC Instructions

- Variable assignment allows assignments of the form:
  \[ var = constant; \]
  \[ var_1 = var_2; \]
  \[ var_1 = var_2 \text{ op } var_3; \]
  \[ var_1 = constant \text{ op } var_2; \]
  \[ var_1 = var_2 \text{ op } constant; \]
  \[ var = constant_1 \text{ op } constant_2; \]

- Permitted operators are +, -, *, /, %.

- How would you compile \( y = -x; \)?
  \[ y = 0 - x; \]
  \[ y = -1 * x; \]
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
TAC with **bools**

```c
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y

_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;

_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;

_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```
Boolean variables are represented as integers that have zero or nonzero values.

In addition to the arithmetic operator, TAC supports:
\begin{itemize}
  \item \texttt{<},
  \item \texttt{==},
  \item \texttt{||}, and
  \item \texttt{&&}.
\end{itemize}
Boolean variables are represented as integers that have zero or nonzero values.

In addition to the arithmetic operator, TAC supports: 

- `<`,
- `==`,
- `||`, and
- `&&`.

How might you compile $b = (x \leq y)$?
int x;
int y;
int z;

if (x < y)
    z = x;
else
    z = y;

z = z * z;
Control Flow Statements

```c
int x;
int y;
int z;

if (x < y)
    z = x;
else
    z = y;

z = z * z;
```

```c
_t0 = x < y;
IfZ _t0 Goto _L0;
    z = x;
    Goto _L1;
_L0:
    z = y;
_L1:
    z = z * z;
```
Labels

- TAC allows for named labels indicating particular points in the code that can be jumped to.
- There are two control flow instructions:
  1. Goto label;
  2. IfZ value Goto label;
- Note that IfZ is always paired with Goto.
int x;
int y;

while (x < y) {
    x = x * 2;
}

y = x;
int x;
int y;

while (x < y) {
    x = x * 2;
}

y = x;
Decaf functions consist of four pieces:

1. A label identifying the start of the function.
2. A `BeginFunc N;` instruction reserving N bytes of space for locals and temporaries.
3. The body of the function.
4. An `EndFunc;` instruction marking the end of the function.
   - When reached, cleans up stack frame and returns.
void main() {
    int x, y;
    int m2 = x * x + y * y;

    while (m2 > 5) {
        m2 = m2 - x;
    }
}

main:
    BeginFunc 24;
    _t0 = x * x;
    _t1 = y * y;
    m2 = _t0 + _t1;
_L0:
    _t2 = 5 < m2;
    IfZ _t2 Goto _L1;
    m2 = m2 - x;
    Goto _L0;
_L1:
    EndFunc;
A Logical Decaf Stack Frame

Stack frame for function  
f(a, ..., n)

- Param N
- Param N - 1
- ...
- Param 1
- Storage for Locals and Temporaries
A Logical Decaf Stack Frame

Stack frame for function \( f(a, \ldots, n) \)

- Param \( N \)
- Param \( N - 1 \)
- \( \ldots \)
- Param \( 1 \)
- Storage for Locals and Temporaries

Stack frame for function \( g(a, \ldots, m) \)

- Param \( M \)
- \( \ldots \)
- Param \( 1 \)
- Storage for Locals and Temporaries
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
    SimpleFunction(137);
}
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
    SimpleFunction(137);
}

_SimpleFn:
    BeginFunc 16;
    _t0 = x * y;
    _t1 = _t0 * z;
    x = _t1;
    EndFunc;
Stack Management in TAC

- The **BeginFunc N;** instruction only needs to reserve room for local variables and temporaries.
- The **EndFunc;** instruction reclaims the room allocated with **BeginFunc N;**
- A single parameter is pushed onto the stack by the caller using the **PushParam var** instruction.
- Space for parameters is reclaimed by the caller using the **PopParams N;** instruction.
- **N** is measured in bytes, not number of arguments.
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
    SimpleFunction(137);
}

_SimpleFn:
    BeginFunc 16;
    _t0 = x * y;
    _t1 = _t0 * z;
    x = _t1;
    EndFunc;

main:
    BeginFunc 4;
    _t0 = 137;
    PushParam _t0;
    LCall _SimpleFn;
    PopParams 4;
    EndFunc;
Compiling Function Calls

Stack frame for function \( f(a, \ldots, n) \)

- Param \( N \)
- Param \( N - 1 \)
- \( \ldots \)
- Param \( 1 \)
- Storage for Locals and Temporaries

Stack frame for function \( g(a, \ldots, m) \)

- Param \( M \)
- \( \ldots \)
- Param \( 1 \)
- Storage for Locals and Temporaries

\[
\begin{align*}
\text{PushParam } & \text{ var;} \\
\text{PushParam } & \text{ var;} \\
\text{BeginFunc } & \text{ N;} 
\end{align*}
\]
Compiling Function Calls

Stack frame for function f(a, ..., n)

- Param N
- Param N - 1
- ...
- Param 1
- Storage for Locals and Temporaries
- Param M
- ...
- Param 1
- Storage for Locals and Temporaries

EndFunc;
Compiling Function Calls

Stack frame for function f(a, ..., n)

Param N
Param N - 1
...
Param 1
Storage for Locals and Temporaries
Param M
...
Param 1

PushParams N;
### Instruction Formats

<table>
<thead>
<tr>
<th>Type</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary operation</td>
<td>$x = y \text{ op } z$</td>
</tr>
<tr>
<td>Unary operation</td>
<td>$x = \text{ op } y$</td>
</tr>
<tr>
<td>Copy</td>
<td>$x = y$</td>
</tr>
<tr>
<td>Unconditional Jump</td>
<td>$\text{goto } L$</td>
</tr>
</tbody>
</table>
| Conditional Jump            | $\text{if } x \text{ goto } L$
|                             | $\text{ifFalse } x \text{ goto } L$ |
| Compare and Jump            | $\text{if } x \text{ relop } y \text{ goto } L$ |
| Procedure Call              | $\text{param } x_1$
|                             | $\ldots$
|                             | $\text{param } x_n$
|                             | $\text{call } p, n$ |
| Function Call               | $\text{param } x_1$
|                             | $\ldots$
|                             | $\text{param } x_n$
|                             | $y = \text{call } p, n$ |
| Indexed Copy                | $x = y[i]$ |
|                             | $x[i] = y$ |
| Address/Pointer assignment  | $x = \&y$
|                             | $x = \ast y$
|                             | $\ast x = y$ |