Chapter 11: Compiler II: Code Generation

Usage and Copyright Notice:

Copyright 2005 © Noam Nisan and Shimon Schocken


The book web site, www.idc.ac.il/tecs, features 13 such presentations, one for each book chapter. Each presentation is designed to support about 3 hours of classroom or self-study instruction.

You are welcome to use or edit this presentation for instructional and non-commercial purposes.

If you use our materials, we will appreciate it if you will include in them a reference to the book’s web site.

And, if you have any comments, you can reach us at tecs.ta@gmail.com
The big picture

- **Syntax analysis**: understanding the code
- **Code generation**: constructing semantics

Syntax analysis: understanding the code

Code generation: constructing semantics
Syntax analysis (review)

The code generation challenge:
- Extend the syntax analyzer into a full-blown compiler
- Program = a series of operations that manipulate data
- The compiler should convert each “understood” (parsed) source operation and data item into corresponding operations and data items in the target language
- So we have to generate code for
  - handling data
  - handling operations.
Handling data

When dealing with a variable, say x, we have to know:

- **What is x's data type?**
  - Primitive, or ADT (class name)?
  - (Need to know in order to properly allocate to it RAM resources)

- **What kind of variable is x?**
  - local, static, field, argument?
  - (Need to know in order to properly manage its life cycle)
Symbol table

```
class BankAccount {
    // Class variables
    static int nAccounts;
    static int bankCommission;
    // account properties
    field int id;
    field String owner;
    field int balance;

    method int commission(int x) { /* Code omitted */ }

    method void transfer(int sum, BankAccount from, Date when) {
        var int i, j; // Some local variables
        var Date due; // Date is a user-defined type
        let balance = (balance + sum) - commission(sum * 5);
        // More code ...
    }
}
```

Classical implementation:
- A list of hash tables, each reflecting a single scope nested within the next one in the list.
- The identifier lookup works from the current table upwards.
Life cycle

- **Static**: single copy must be kept alive throughout the program duration
- **Field**: different copies must be kept for each object
- **Local**: created on subroutine entry, killed on exit
- **Argument**: similar to local
- **Good news**: the VM handles all these details !!! Hurray!!!
Handling arrays

Java code

class Complex {
    ...
    void foo(int k) {
        int x, y;
        int[] bar; // declare an array
        ...
        // Construct the array:
        bar = new int[10];
        ...
        bar[k]=19;
    }
    ...
    Main.foo(2); // Call the foo method
    ...
}

VM Code (pseudo)

// bar[k]=19, or *(bar+k)=19
push bar
push k
add
  // Use a pointer to access x[k]
pop addr // addr points to bar[k]
push 19
pop *addr // Set bar[k] to 19

VM Code (final)

// bar[k]=19, or *(bar+k)=19
push local 2
push argument 0
add
  // Use the that segment to access x[k]
pop pointer 1
push constant 19
pop that 0

Following compilation:

RAM state, just after executing bar[k]=19

Bar = new int(n)

Is typically handled by causing the compiler to generate code affecting:

bar = Mem.alloc(n)

275  x (local 0)
276  y (local 1)
277  4315  bar (local 2)
504  2  k (argument 0)
4315  (bar array)
4316
4317  19
4318
4324

Handling objects: memory allocation

Java code

```java
class Complex {
    // Properties (fields):
    int re;  // Real part
    int im;  // Imaginary part
    ...
    /** Constructs a new Complex object. */
    public Complex(int aRe, int aIm) {
        re = aRe;
        im = aIm;
    }
    ...
}
// The following code can be in any class:
public void bla() {
    Complex a, b, c;
    ...
    a = new Complex(5,17);
    b = new Complex(12,192);
    ...
    c = a;  // Only the reference is copied
    ...
}
```

Following compilation:

```
0  ...
326  6712  a
327  7002  b
328  6712  c
    ...
6712  5
6713  17
    ...
7002  12 
7003  192 
    ...
```

`foo = new ClassName(...)`

Is typically handled by causing the compiler to generate code affecting:

`foo = Mem.alloc(n)`
Handling objects: operations

Java code

```java
class Complex {
    // Properties (fields):
    int re;  // Real part
    int im;  // Imaginary part
    ...
    /** Constructs a new Complex object. */
    public Complex(int aRe, int aIm) {
        re = aRe;
        im = aIm;
    }
    ...
    // Multiplication:
    public void mult (int c) {
        re = re * c;
        im = im * c;
    }
}
```

Translating $\text{im} = \text{im} \times \text{c}$:

- Look up the symbol table
- Resulting semantics:

```
// im = im * c :
*(this+1) = *(this+1) times (argument 0)
```

- Of course this should be written in the target language.
Handling objects: method calls

Java code

```java
class Complex {
    // Properties (fields):
    int re;  // Real part
    int im;  // Imaginary part
    ...
    /** Constructs a new Complex object. */
    public Complex(int aRe, int aIm) {
        re = aRe;
        im = aIm;
    }
    ...
}

class Foo {
    ...
    public void foo() {
        Complex x;
        ...
        x = new Complex(1,2);
        x.mult(5);
        ...
    }
    }
```

Translating `x.mult(5)`:

- Can also be viewed as `mult(x,5)`
- Generated code:

```
// x.mult(5):
push x
push 5
call mult
```

General rule: each method call `foo.bar(v1,v2,...)`

- can be translated into

```
push foo
push v1
push v2
...
call bar
```
Generating code for expressions

The `codeWrite(exp)` algorithm:

- if `exp` is a number `n` then output “push n”;
- if `exp` is a variable `v` then output “push v”;
- if `exp = (exp1 op exp2)` then `codeWrite(exp1); codeWrite(exp2);` output “op”;
- if `exp = op(exp1)` then `codeWrite(exp1);` output “op”;
- if `exp = f(exp1 ... expN)` then `codeWrite(exp1) ... codeWrite(expN);` output “call f”.

The expression `x + g(2, y, -z) * 5` is parsed and translated into code.
### Handling control flow (e.g. IF, WHILE)

<table>
<thead>
<tr>
<th>Source code</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (cond)</td>
</tr>
<tr>
<td>s1</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>s2</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Generated code</th>
</tr>
</thead>
<tbody>
<tr>
<td>code for computing ~cond</td>
</tr>
<tr>
<td>if-goto L1</td>
</tr>
<tr>
<td>code for executing s1</td>
</tr>
<tr>
<td>goto L2</td>
</tr>
<tr>
<td>label L1</td>
</tr>
<tr>
<td>code for executing s2</td>
</tr>
<tr>
<td>label L2</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source code</th>
</tr>
</thead>
<tbody>
<tr>
<td>while (cond)</td>
</tr>
<tr>
<td>s1</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Generated code</th>
</tr>
</thead>
<tbody>
<tr>
<td>label L1</td>
</tr>
<tr>
<td>code for computing ~cond</td>
</tr>
<tr>
<td>if-goto L2</td>
</tr>
<tr>
<td>code for executing s1</td>
</tr>
<tr>
<td>goto L1</td>
</tr>
<tr>
<td>label L2</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Program flow

### Flow of control structure

if (cond)  
s1  
else  
s2  
...

### VM pseudo code

VM code for computing ~(cond)  
if-goto L1  
VM code for executing s1  
goto L2  
label L1  
VM code for executing s2  
label L2  
...

while (cond)  
s1  
...

VM code for computing ~(cond)  
if-goto L2  
VM code for executing s1  
goto L1  
label L2  
...
High level code (BankAccount.java class file)

```java
/* Some common sense was sacrificed in this banking example in order
to create a non trivial and easy-to-follow compilation example. */
class BankAccount {
    // Class variables
    static int nAccounts;
    static int bankCommission; // As a percentage, e.g., 10 for 10 percent
    // account properties
    field int id;
    field String owner;
    field int balance;

    method int commission(int x) { /* Code omitted */ }

    method void transfer(int sum, BankAccount from, Date when) {
        var int i, j; // Some local variables
        var Date due; // Date is a user-defined type
        let balance = (balance + sum) - commission(sum * 5);
        // More code ...
        return;
    } // More methods ... }
```

Class scope symbol table

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Kind</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>nAccounts</td>
<td>int</td>
<td>static</td>
<td>0</td>
</tr>
<tr>
<td>bankCommission</td>
<td>int</td>
<td>static</td>
<td>1</td>
</tr>
<tr>
<td>id</td>
<td>int</td>
<td>field</td>
<td>0</td>
</tr>
<tr>
<td>owner</td>
<td>String</td>
<td>field</td>
<td>1</td>
</tr>
<tr>
<td>balance</td>
<td>int</td>
<td>field</td>
<td>2</td>
</tr>
</tbody>
</table>

Method scope (transfer) symbol table

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Kind</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>this</td>
<td>BankAccount</td>
<td>argument</td>
<td>0</td>
</tr>
<tr>
<td>sum</td>
<td>int</td>
<td>argument</td>
<td>1</td>
</tr>
<tr>
<td>from</td>
<td>BankAccount</td>
<td>argument</td>
<td>2</td>
</tr>
<tr>
<td>when</td>
<td>Date</td>
<td>argument</td>
<td>3</td>
</tr>
<tr>
<td>i</td>
<td>int</td>
<td>var</td>
<td>0</td>
</tr>
<tr>
<td>j</td>
<td>int</td>
<td>var</td>
<td>1</td>
</tr>
<tr>
<td>due</td>
<td>Date</td>
<td>var</td>
<td>2</td>
</tr>
</tbody>
</table>

Pseudo VM code

```java
function BankAccount.commission
    // Code omitted
function BankAccount.transfer
    // Code for setting "this" to point
    // to the passed object (omitted)
    push balance
    push sum
    add
    push this
    push sum
    push 5
    call multiply
    call commission
    sub
    pop balance
    // More code ...
    push 0
    return
```

Final VM code

```java
function BankAccount.commission 0
    // Code omitted
function BankAccount.transfer 3
    push argument 0
    pop pointer 0
    push this 2
    push argument 1
    add
    push argument 0
    push argument 1
    push constant 5
    call Math.multiply 2
    call BankAccount.commission 2
    sub
    pop this 2
    // More code ...
    push 0
    return
```
Perspective

“Hard” Jack simplifications:

- Primitive type system
- No inheritance
- No public class fields (e.g. must use `r=c.getRadius()` rather than `r=c.radius`)

“Soft” Jack simplifications:

- Limited control structures (no `for`, `switch`, …)
- Cumbersome handling of char types (cannot use `let x='c'`)

Optimization

- For example, `c++` will be translated into `push c, push 1, add, pop c`.
- Parallel processing
- Many other examples of possible improvements …