Advanced Models and Programs

Grammar transformations and stack machines

Peter Sestoft
Wed 2009-02-11

Grammar transformations

• Grammar ambiguity
• Encoding operator precedence
• Left factorization
• Left recursion elimination

Compilation for a stack machine

• Postfix notation
• Compilation schemes
• Stack machines
• Postscript
### Grammar ambiguity

- A grammar is ambiguous if the same string can be derived two ways
- Example: Derive $1 + 2 + 3$
- Find a corresponding unambiguous grammar
- Note: Even an unambiguous grammar may not be LL(1)-parsable

#### Ambiguous grammar:

```
Expr = Expr \+\ Expr \\
    | Number
```

#### Unambiguous grammar:

```
Expr = Number Expr' \\
Expr' = \epsilon \\
    | \+\ Number Expr'
```

#### Same, compact notation:

```
Expr = Number \{ \+\ Number \}
```

### Operator precedence with LL

- Should "2 * 3 + 4" parse as $(2 * 3) + 4$ or as $2 * (3 + 4)$?
- In LL-parsing, operator precedence is encoded using extra non-terminals

#### Grammar w/o precedence:

```
Expr = Expr \+\ Expr \\
    | Expr \*\ Expr \\
    | Number | ...
```

#### New grammar with precedence:

```
Expr = Term \{ \+\ Term \} \\
Term  = Factor \{ \*\ Factor \} \\
Factor = Number | ...
```
Left factorization in LL

- Alternatives must be distinguishable on their possible first symbols
- If two alternatives start with the same terminal or non-terminal, join them and create new non-terminals

Example: Var and Call

\[ \text{Expr} = \text{Ident} \]
\[ \quad | \quad \text{Ident "( " )"} \]

Left factorization:

Replace
\[
A = a \ b \ c \\
\quad | \quad a \ b \ d
\]
with
\[
A = a \ b \ A' \\
A' = c \quad | \quad d
\]

Left recursion elimination

- If an alternative starts with the non-terminal itself, the rule is left recursive
- Many grammar rules are naturally left recursive
- Left recursion can be eliminated systematically

Example: C types

\[ \text{Type} = \text{Ident} \]
\[ \quad | \quad \text{Type "[ " "]"} \]

Left recursion elimination:

Replace
\[
A = A \ b \ | \ c
\]
with
\[
A = c \ \{ \ b \ \}
\]
Compilation of expressions

- Consider expression \(2 \times 3 + 4 \times 5\)
- Write it in postfix: \(2 \ 3 \times 4 \ 5 \times +\)
- This is code for a stack machine:

<table>
<thead>
<tr>
<th>Instructions:</th>
<th>Stack contents:</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 * 4 5 * +</td>
<td>2</td>
</tr>
<tr>
<td>3 * 4 5 * +</td>
<td>2 3</td>
</tr>
<tr>
<td>* 4 5 * +</td>
<td>6</td>
</tr>
<tr>
<td>4 5 * +</td>
<td>6 4</td>
</tr>
<tr>
<td>5 * +</td>
<td>6 4 5</td>
</tr>
<tr>
<td>* +</td>
<td>6 20</td>
</tr>
<tr>
<td>+</td>
<td>26</td>
</tr>
</tbody>
</table>
Postscript is a postfix stack-based language

- A Postscript printer is an interpreter

\[
4 \ 5 \ add \ 8 \ mul = (4 + 5) * 8
\]

\[
/x \ 7 \ def
x \ x \ mul \ 9 \ add = \text{let } x=7 \text{ in } x*x+9
\]

\[
/fac \{ \ dup \ 0 \ eq
{ \ pop \ 1 } \{ \ dup \ 1 \ sub \ fac \ mul \} \text{ifelse } \} \ def
\text{n!, factorial function}
\]

Stack machines everywhere

- Burroughs B5000 hardware (1961)
- Forth virtual machine (1970)
- P-code, UCSD Pascal (1977)
- Western Digital Pascal microEngine
- Postscript (1984)
- Java Virtual Machine (1994)
- picoJava JVM core
- .NET runtime (1999)
- ARM Jazelle instructions (2005)
- ... zillions of others
The microC stack machine (part)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack before</th>
<th>Stack after</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSTI n</td>
<td>s</td>
<td>s, n</td>
<td>Push const</td>
</tr>
<tr>
<td>ADD</td>
<td>s, n1, n2</td>
<td>s, n1+n2</td>
<td>Add</td>
</tr>
<tr>
<td>SUB</td>
<td>s, n1, n2</td>
<td>s, n1-n2</td>
<td>Subtract</td>
</tr>
<tr>
<td>MUL</td>
<td>s, n1, n2</td>
<td>s, n1*n2</td>
<td>Multiply</td>
</tr>
<tr>
<td>DIV</td>
<td>s, n1, n2</td>
<td>s, n1/n2</td>
<td>Divide</td>
</tr>
<tr>
<td>MOD</td>
<td>s, n1, n2</td>
<td>s, n1%n2</td>
<td>Modulo</td>
</tr>
<tr>
<td>EQ</td>
<td>s, n1, n2</td>
<td>s, n1==n2</td>
<td>Equality (0/1)</td>
</tr>
<tr>
<td>LT</td>
<td>s, n1, n2</td>
<td>s, n1&lt;n2</td>
<td>Less than (0/1)</td>
</tr>
<tr>
<td>NOT</td>
<td>s, v</td>
<td>s, !v</td>
<td>Logical not (0/1)</td>
</tr>
<tr>
<td>SWAP</td>
<td>s, v1, v2</td>
<td>s, v2, v1</td>
<td>Swap</td>
</tr>
<tr>
<td>LDI</td>
<td>s, n</td>
<td>s, s[n]</td>
<td>Load indirect</td>
</tr>
<tr>
<td>INCSP -n</td>
<td>s, v1, ..., vn</td>
<td>s</td>
<td>Pop n items</td>
</tr>
</tbody>
</table>

Compilation schemes, part 1

- To compile constant n:
  - Emit: CSTI n

- To compile e1 + e2:
  - Compile e1
  - Compile e2
  - Emit: ADD

- To compile e1 < e2:
  - Compile e1
  - Compile e2
  - Emit: LT
Net effect principle

- An expression e is compiled to a sequence of instructions
- The net effect of executing the instructions must be to leave the expression’s value on the stack

Compilation of constants and variables

```java
public abstract class Expression {
    abstract public void Compile(CEnv env, Generator gen);
}

public class Constant : Expression {
    private readonly int value;
    public override void Compile(CEnv env, Generator gen) {
        gen.Emit(new CSTI(value));
    }
}

public class Variable : Expression {
    private readonly String name;
    public override void Compile(CEnv env, Generator gen) {
        env.CompileVariable(gen, name);
        gen.Emit(Instruction.LDI);
    }
}
```
Compilation of binary operators

```csharp
public class BinOp : Expression {
    private readonly Operator op;
    private readonly Expression e1, e2;
    public override void Compile(CEnv env, Generator gen) {
        e1.Compile(env, gen);
        env.PushTemporary();
        e2.Compile(env, gen);
        switch (op) {
            case Operator.Add:  
                gen.Emit(Instruction.ADD);
                break;
            case Operator.Div:  
                gen.Emit(Instruction.DIV);
                break;
            case Operator.Mul:  
                ...
        }
    }
}
```

Synthesizing semantics (more compilation schemes)

- To compile e1 > e2:
  - Compile e1
  - Compile e2
  - Emit: SWAP, LT
- To compile e1 >= e2:
  - Compile e1
  - Compile e2
  - Emit: LT, NOT
Compiling variable accesses

- Variables are stored in the stack
- Classical Algol/Pascal/C implementation
- Example: 2 * let x=3 in x+4 end
- Code: 2 3 1 LDI 4 + SWAP INCSP(-1) *

The compile-time environment

- The compile-time environment keeps track of variable positions in the stack
- The compile-time environment is a stack; an abstraction of the run-time stack
Compile-time environment CEnv

public class CEnv {
    private readonly Stack<String> locals;
    public void DeclareLocal(String name) {
        locals.Push(name);
    }
    public void PushTemporary() {
        locals.Push("_ temporary ");
    }
    public void CompileVariable(Generator gen, String name) {
        int offset = 0;
        foreach (String variableName in locals) {
            if (variableName == name) {
                gen.Emit(Instruction.GETSP);
                gen.Emit(new CSTI(offset));
                gen.Emit(Instruction.SUB);
                return;
            } else
                offset++;
        }
    }
}

The microC machine (PLC 7.2.2)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack before</th>
<th>Stack after</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSTI i</td>
<td>s,i</td>
<td>s,i</td>
<td>Push constant/i</td>
</tr>
<tr>
<td>ADD</td>
<td>s,i,i;</td>
<td>s,(i1+i2)</td>
<td>Add</td>
</tr>
<tr>
<td>SUB</td>
<td>s,i,i;</td>
<td>s,(i1-i2)</td>
<td>Subtract</td>
</tr>
<tr>
<td>MUL</td>
<td>s,i,i;</td>
<td>s,(i1*i2)</td>
<td>Multiply</td>
</tr>
<tr>
<td>DIV</td>
<td>s,i,i;</td>
<td>s,(i1/i2)</td>
<td>Divide</td>
</tr>
<tr>
<td>MOD</td>
<td>s,i,i;</td>
<td>s,(i1%i2)</td>
<td>Modulo</td>
</tr>
<tr>
<td>EQ</td>
<td>s,i,i;</td>
<td>s,(i1==i2)</td>
<td>Equality (0 or 1)</td>
</tr>
<tr>
<td>LT</td>
<td>s,i,i;</td>
<td>s,(i1&lt;i2)</td>
<td>Less-than (0 or 1)</td>
</tr>
<tr>
<td>NOT</td>
<td>s,v</td>
<td>s,~v</td>
<td>Negation (0 or 1)</td>
</tr>
<tr>
<td>DMP</td>
<td>s,v</td>
<td>s,v,v</td>
<td>Duplicate</td>
</tr>
<tr>
<td>SWAP</td>
<td>s,v1,v2</td>
<td>s,v2,v1</td>
<td>Swap</td>
</tr>
<tr>
<td>LDI</td>
<td>s,i</td>
<td>s,add[i]</td>
<td>Load indirect</td>
</tr>
<tr>
<td>STI</td>
<td>s,i,v</td>
<td>s,v</td>
<td>Store indirect; s[i]=v</td>
</tr>
<tr>
<td>GETSP</td>
<td>s,bp</td>
<td>s,sp</td>
<td>Load base pointer bp</td>
</tr>
<tr>
<td>GESP</td>
<td>s</td>
<td>s,sp</td>
<td>Load stack pointer sp</td>
</tr>
<tr>
<td>INSP m</td>
<td>s,v1,...,vm</td>
<td>s,v1,...,vm</td>
<td>Grow stack (m ≥ 0)</td>
</tr>
<tr>
<td>INCSP m</td>
<td>s,v1,...,vm</td>
<td>s</td>
<td>Shrink stack (m &lt; 0)</td>
</tr>
<tr>
<td>QOT a</td>
<td>s</td>
<td>s</td>
<td>Jump to a</td>
</tr>
<tr>
<td>IPERRO a</td>
<td>s,v</td>
<td>s,x</td>
<td>Jump to a if v ≠ 0</td>
</tr>
<tr>
<td>CALL m a</td>
<td>s,v1,...,vm</td>
<td>s,r,bp,v1,...,vm</td>
<td>Call function at a</td>
</tr>
<tr>
<td>TCALL m n a</td>
<td>s,r,b,v1,...,vm</td>
<td>s,r,b,v1,...,vm</td>
<td>Tail-call function at a</td>
</tr>
<tr>
<td>RET m</td>
<td>s,r,b,v1,...,vm,v</td>
<td>s,v</td>
<td>Return: bp ← b; pc ← r</td>
</tr>
<tr>
<td>PRINT m</td>
<td>s,v</td>
<td>s,v</td>
<td>Print integer v</td>
</tr>
<tr>
<td>PRINTC m</td>
<td>s,v</td>
<td>s,v</td>
<td>Print character v</td>
</tr>
<tr>
<td>LOADS s</td>
<td>s,h1,...,hn</td>
<td>s,(h1,...,hn)</td>
<td>Command line args</td>
</tr>
<tr>
<td>STOP s</td>
<td>s</td>
<td></td>
<td>Halt the machine</td>
</tr>
</tbody>
</table>