Advanced Models and Programs

Grammar transformations
and stack machines

Peter Sestoft
Friday 2010-02-12

Grammar transformations

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

Compilation for a stack machine

• Postfix notation
• Compilation schemes
• Stack machines
• Postscript
### Derivation: grammar as string generator

- $\text{Expr} 
  \Rightarrow \text{Expr} + \text{Expr} 
  \Rightarrow \text{Expr} + \text{Expr} * \text{Expr} 
  \Rightarrow \text{Expr} + \text{Expr} * \text{wk} 
  \Rightarrow \text{Expr} + 52 * \text{wk} 
  \Rightarrow x + 52 * \text{wk}$

#### Grammar ambiguity

- A grammar is *ambiguous* if there is a string that has more than one derivation tree
- Example:
**Eliminating ambiguity**

- Often possible to find a corresponding unambiguous grammar
- Note: Even an unambiguous grammar may not be LL(1)-parsable

<table>
<thead>
<tr>
<th>Ambiguous grammar:</th>
<th>Unambiguous grammar:</th>
</tr>
</thead>
<tbody>
<tr>
<td>`Expr = Expr “+” Expr</td>
<td>`Expr = Number Expr’</td>
</tr>
<tr>
<td></td>
<td>`Expr’ = ε</td>
</tr>
<tr>
<td></td>
<td>`</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Same, compact notation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>`Expr = Number</td>
</tr>
<tr>
<td>`{ “+” Number }</td>
</tr>
</tbody>
</table>

**Operator associativity and precedence**

- Associativity: How should we read x ◊ y ◊ z?
  - ◊ left-assoc.
  - ◊ right-assoc.

- Precedence: How should we read x ◊ y ● z?
  - ◊ higher precedence
  - ● higher precedence

- What Java/C# operators
  - are left-associative?
  - are right-associative?
  - have higher precedence than others?
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

```
Expr = Ident
    | Ident "(" ")"
```

Left factorization:

Replace

```
A = a b c
    | a b d
```

with

```
A  = a b A'
A' = c | 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.
- Top-down/LL(1) parsers do not work on left recursive rules.
- Left recursion can be eliminated systematically.

Example: C types

```
Type = Ident
    | Type "[" "]"
```

Left recursion elimination:

Replace

```
A = A b | c
```

with

```
A = c { b }
```
Grammar classes
(Chomsky hierarchy, 1956)

• Type 3: Regular grammars; same expressiveness as regular expressions
  \[ A \rightarrow cB \quad A \rightarrow B \quad A \rightarrow c \quad A \rightarrow \epsilon \]

• Type 2: Context-free grammars (CFG)
  \[ A \rightarrow cBd \]

• Type 1: Context-sensitive grammars, non-abbreviating rules
  \[ aAb \rightarrow acAdb \]

• Type 0: Unrestricted grammars; same as term rewrite systems
  \[ 0Ay \rightarrow 0 \]

LL versus LR parsing (within CFG)

• LL: Read input from Left to right, make derivations from Leftmost nonterminal
  – Predictive parsing
  – Top-down parsing
  – Easy to hand-write a parser
  – Requires grammar transformations (associativity)

• LR: Read input from Left to right, make derivations from Rightmost nonterminal
  – Bottom-up parsing
  – Difficult to hand-write parsers, but excellent parser generator tools – e.g. yacc, bison, fsyacc
  – No grammar transformations required
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 = $(4 + 5) \times 8$

/x 7 def
x x mul 9 add = let $x=7$ in $x^2 + 9$

/fac { dup 0 eq
{ pop 1 }
{ dup 1 sub fac mul }
ifelse } def

n!, factorial function

gs on ssh.itu.dk
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

```java
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) *

<table>
<thead>
<tr>
<th>Instructions:</th>
<th>Stack:</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 1 LDI 4 + SWAP INCSP(-1) *</td>
<td>2</td>
</tr>
<tr>
<td>3 1 LDI 4 + SWAP INCSP(-1) *</td>
<td>2 3</td>
</tr>
<tr>
<td>1 LDI 4 + SWAP INCSP(-1) *</td>
<td>2 3 1</td>
</tr>
<tr>
<td>LDI 4 + SWAP INCSP(-1) *</td>
<td>2 3 3</td>
</tr>
<tr>
<td>4 + SWAP INCSP(-1) *</td>
<td>2 3 3 4</td>
</tr>
<tr>
<td>+ SWAP INCSP(-1) *</td>
<td>2 3 7</td>
</tr>
<tr>
<td>SWAP INCSP(-1) *</td>
<td>2 7</td>
</tr>
<tr>
<td>INCSP(-1) *</td>
<td>14</td>
</tr>
</tbody>
</table>
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

**Position in expression:**

2*
2*let x=3 in
2*let x=3 in x+
2*let x=3 in x+4 end

**Compile-time env:**

```
TEMP
TEMP x
TEMP x TEMP
TEMP
```

---

**Compile-time environment CEnv**

```java
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 (PLCSD 8.2.2)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack before</th>
<th>Stack after</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 PSW i</td>
<td>s</td>
<td>s, i</td>
<td>Push constant i</td>
</tr>
<tr>
<td>1 ADD</td>
<td>s, i, j + k</td>
<td>s, (i + j)</td>
<td>Add</td>
</tr>
<tr>
<td>2 SUB</td>
<td>s, i, j + k</td>
<td>s, (i - j)</td>
<td>Subtract</td>
</tr>
<tr>
<td>3 MUL</td>
<td>s, i, j + k</td>
<td>s, (i * j)</td>
<td>Multiply</td>
</tr>
<tr>
<td>4 DIV</td>
<td>s, i, j + k</td>
<td>s, (i / j)</td>
<td>Divide</td>
</tr>
<tr>
<td>5 MOD</td>
<td>s, i, j + k</td>
<td>s, (i % j)</td>
<td>Modulo</td>
</tr>
<tr>
<td>6 EQ</td>
<td>s, i, j + k</td>
<td>s, (i = j)</td>
<td>Equality (0 or 1)</td>
</tr>
<tr>
<td>7 LT</td>
<td>s, i, j + k</td>
<td>s, (i &lt; j)</td>
<td>Less-than (0 or 1)</td>
</tr>
<tr>
<td>8 NEC</td>
<td>s, v</td>
<td>s, !v</td>
<td>Negation (0 or 1)</td>
</tr>
<tr>
<td>9 DUP</td>
<td>s, v</td>
<td>s, v, v</td>
<td>Duplicate</td>
</tr>
<tr>
<td>10 SWAP</td>
<td>s, v1, v2</td>
<td>s, v1, v2</td>
<td>Swap</td>
</tr>
<tr>
<td>11 LDS</td>
<td>s, i</td>
<td>s, s[i]</td>
<td>Load indirect</td>
</tr>
<tr>
<td>12 STD</td>
<td>s, i, v</td>
<td>s, v</td>
<td>Store indirect s[i] = v</td>
</tr>
<tr>
<td>13 GMP</td>
<td>s</td>
<td>s, bp</td>
<td>Load base ptr bp</td>
</tr>
<tr>
<td>14 GMR</td>
<td>s</td>
<td>s, sp</td>
<td>Load stack ptr sp</td>
</tr>
<tr>
<td>15 INCP m</td>
<td>s</td>
<td>s, v1, ..., v m</td>
<td>Grow stack (m ≥ 0)</td>
</tr>
<tr>
<td>16 DEC m</td>
<td>s</td>
<td>s, v1, ..., v m</td>
<td>Shrink stack (m &lt; 0)</td>
</tr>
<tr>
<td>17 INP m a</td>
<td>s, v</td>
<td>s, a</td>
<td>Jump to a</td>
</tr>
<tr>
<td>18 INP m d</td>
<td>s, v</td>
<td>s, d</td>
<td>Jump to d if v = 0</td>
</tr>
<tr>
<td>19 CALL m a</td>
<td>s, v1, ..., v na</td>
<td>s, r, h_p, v1, ..., v na</td>
<td>Call function at a</td>
</tr>
<tr>
<td>20 CALL m n a</td>
<td>s, r, h_p, v1, ..., v na</td>
<td>s, r, d, v1, ..., v na</td>
<td>Tailcall function at a</td>
</tr>
<tr>
<td>21 LET m</td>
<td>s, v</td>
<td>s, v, v1, ..., v m</td>
<td>Return bp = b, pc = r</td>
</tr>
<tr>
<td>22 PRINT t</td>
<td>s, v</td>
<td>s, v</td>
<td>Print integer v</td>
</tr>
<tr>
<td>23 PRINTC t</td>
<td>s, v</td>
<td>s, v</td>
<td>Print character v</td>
</tr>
<tr>
<td>24 LDAPS</td>
<td>s</td>
<td>s, i1, ..., i t</td>
<td>Command line args</td>
</tr>
<tr>
<td>25 STOP</td>
<td>s</td>
<td></td>
<td>Halt the machine</td>
</tr>
</tbody>
</table>

What’s next

- Monday lecture:
  - C concepts, lvalue, rvalue, pointer arithmetics
  - Micro-C parsing challenges
  - Micro-C abstract syntax
  - Compilation of Micro-C to stack machine code

- Exercises week 3:
  - Extend the expression language
  - Extend the lexer and parser specifications
  - Extend the interpreter, checker and compiler