Advanced Models and Programs

Grammar transformations and stack machines

Peter Sestoft

Friday 2010-02-12

Grammar transformations

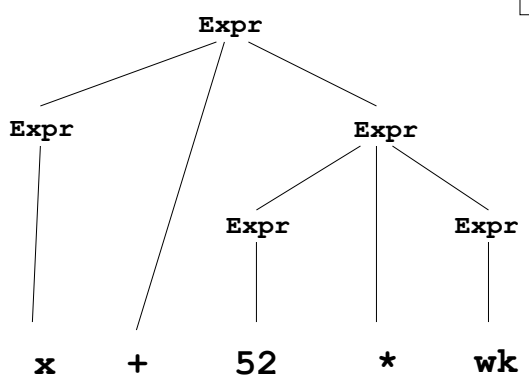
- Grammar ambiguity
- Encoding operator precedence
- Left factorization
- Left recursion elimination
- Grammar classes

Compilation for a stack machine

- Postfix notation
- Compilation schemes
- Stack machines
- Postscript

Derivation: grammar as string generator

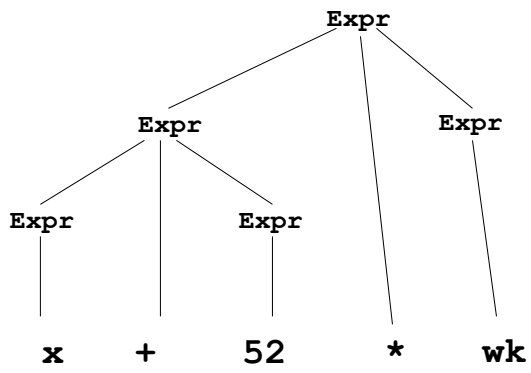
```
Expr
=> Expr + Expr
=> Expr + Expr * Expr
=> Expr + Expr * wk
=> Expr + 52 * wk
=> x + 52 * wk
```



Derivation tree

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

Ambiguous grammar:

```
Expr = Expr "+" Expr
     | Number
```

Unambiguous grammar:

```
Expr  = Number Expr'
Expr' = ε
      | "+" Number Expr'
```

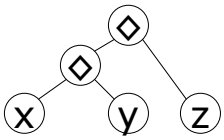
Same, compact notation:

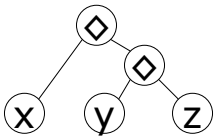
```
Expr = Number
       { "+" Number }
```


Operator associativity and precedence

- Associativity: How should we read x ◇ y ◇ z ?

◇ left-assoc.

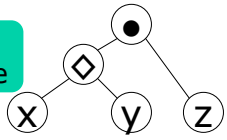


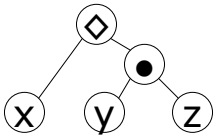


◇ right-assoc.

- 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 }
```



```
Expr<out Expression e>      (. Expression e1, e2; Operator op; e = null; .)
= SimExpr<out e1>           (. e = e1; .)
  [ RelOp<out op>
    SimExpr<out e2>         (. e = new BinOp(op, e, e2); .)
  ]
.
SimExpr<out Expression e>   (. Expression e1, e2; Operator op; .)
= Term<out e1>              (. e = e1; .)
  { AddOp<out op>
    Term<out e2>            (. e = new BinOp(op, e, e2); .)
  }
.
Term<out Expression e>      (. Operator op; Expression e1, e2; .)
= Factor<out e1>            (. e = e1;                         .)
  { MulOp<out op>
    Factor<out e2>          (. e = new BinOp(op, e, e2);       .)
  }
.
Factor<out Expression e>    (. String name; Expression e1; e = null; .)
= ( Ident<out name>         (. e = new Variable(name); .)
  | number                  (. e = new Constant(Convert.ToInt32(t.val),
                                                Type.intType); .)
  | '-' Factor<out e1>      (. e = new UnOp(Operator.Neg, e1); .)
  | '(' Expr<out e1> ')'    (. e = e1; .)
  )
.
RelOp<out Operator op>      (. op = Operator.Bad; .)
= ( "=="                    (. op = Operator.Eq;  .)
  | "!="                    (. op = Operator.Ne;  .) ...
MulOp<out Operator op>      (. op = Operator.Bad; .) = ...
AddOp<out Operator op>      (. op = Operator.Bad; .) = ...
```

Grammar classes (Chomsky hierarchy, 1956)

- Type 3: Regular grammars; same expressiveness as regular expressions

  $A \rightarrow cB$ $A \rightarrow B$ $A \rightarrow c$ $A \rightarrow \varepsilon$
- 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 * 3 + 4 * 5`
- Write it in postfix: `2 3 * 4 5 * +`
- This is code for a stack machine:

Instructions:	Stack contents:
`2 3 * 4 5 * +`	`2`
`3 * 4 5 * +`	`2 3`
`* 4 5 * +`	`6`
`4 5 * +`	`6 4`
`5 * +`	`6 4 5`
`* +`	`6 20`
`+`	`26`

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 =
```

let x=7 in x*x+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)

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

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

```
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);
  }
}
```

Compile-time environment

Compilation of binary operators

```
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) *`**

Instructions:

```
2 3 1 LDI 4 + SWAP INCSP(-1) *
  3 1 LDI 4 + SWAP INCSP(-1) *
    1 LDI 4 + SWAP INCSP(-1) *
      LDI 4 + SWAP INCSP(-1) *
          4 + SWAP INCSP(-1) *
            + SWAP INCSP(-1) *
              SWAP INCSP(-1) *
                   INCSP(-1) *
                             *
```

Stack:

```
2
2 3
2 3 1
2 3 3
2 3 3 4
2 3 7
2 7 3
2 7
14
```

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:	Compile-time env:
`2*`	`TEMP`
`2*let x=3 in`	`TEMP x`
`2*let x=3 in x+`	`TEMP x TEMP`
`2*let x=3 in x+4 end`	`TEMP`

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 (PLCSD 8.2.2)

	Instruction	Stack before		Stack after	Effect
0	CSTI i	s	$\Rightarrow$	s, i	Push constant i
1	ADD	s, i_1, i_2	$\Rightarrow$	$s, (i_1 + i_2)$	Add
2	SUB	s, i_1, i_2	$\Rightarrow$	$s, (i_1 - i_2)$	Subtract
3	MUL	s, i_1, i_2	$\Rightarrow$	$s, (i_1 * i_2)$	Multiply
4	DIV	s, i_1, i_2	$\Rightarrow$	$s, (i_1 / i_2)$	Divide
5	MOD	s, i_1, i_2	$\Rightarrow$	$s, (i_1 \% i_2)$	Modulo
6	EQ	s, i_1, i_2	$\Rightarrow$	$s, (i_1 = i_2)$	Equality (0 or 1)
7	LT	s, i_1, i_2	$\Rightarrow$	$s, (i_1 < i_2)$	Less-than (0 or 1)
8	NOT	s, v	$\Rightarrow$	$s, !v$	Negation (0 or 1)
9	DUP	s, v	$\Rightarrow$	s, v, v	Duplicate
10	SWAP	s, v_1, v_2	$\Rightarrow$	s, v_2, v_1	Swap
11	LDI	s, i	$\Rightarrow$	$s, s[i]$	Load indirect
12	STI	s, i, v	$\Rightarrow$	s, v	Store indirect $s[i] = v$
13	GETBP	s	$\Rightarrow$	s, bp	Load base ptr bp
14	GETSP	s	$\Rightarrow$	s, sp	Load stack ptr sp
15	INCSP m	s	$\Rightarrow$	$s, v_1, .., v_m$	Grow stack ($m \geq 0$)
15	INCSP m	$s, v_1, .., v_{-m}$	$\Rightarrow$	s	Shrink stack ($m < 0$)
16	GOTO a	s	$\Rightarrow$	s	Jump to a
17	IFZERO a	s, v	$\Rightarrow$	s	Jump to a if $v = 0$
18	IFNZRO a	s, v	$\Rightarrow$	s	Jump to a if $v \neq 0$
19	CALL m a	$s, v_1, .., v_m$	$\Rightarrow$	$s, r, bp, v_1, .., v_m$	Call function at a
20	TCALL m n a	$s, r, b, u_1, .., u_n, v_1, .., v_m$	$\Rightarrow$	$s, r, b, v_1, .., v_m$	Tailcall function at a
21	RET m	$s, r, b, v_1, .., v_m, v$	$\Rightarrow$	s, v	Return $bp = b$, $pc = r$
22	PRINTI	s, v	$\Rightarrow$	s, v	Print integer v
23	PRINTC	s, v	$\Rightarrow$	s, v	Print character v
24	LDARGS	s	$\Rightarrow$	$s, i_1, .., i_n$	Command line args
25	STOP	s	$\Rightarrow$	_	Halt the machine

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