Advanced Models and Programs

Abstract syntax, interpretation, checking; lexing and parsing

Peter Sestoft
Monday 2009-02-09

Plan for today

• Plan for the next six lectures
• Abstract syntax trees
• Interpretation of abstract syntax
• Checking of abstract syntax
• From text to abstract syntax
  1. Lexing: character stream to token stream
  2. Parsing: token stream to abstract syntax
The next six AMP lectures

- Mon 9: Abstract syntax, interpretation, lexing, parsing
- Wed 11: Parsing, grammar transformations, compilation, stack machines
- Mon 16: Micro-C and pointers
- Wed 18: Bytecode, abstract machines, real machines
- Mon 23: The Scheme programming language, program generation and multi-stage programs
- Wed 25: *(Runtime code generation in a spreadsheet implementation or Runtime bytecode generation)*

A little language of expressions

- Example expressions:

```
7
x
7 + 9 * 10
x + y * z
x * (y + z) > 42
(x > 42) * 10
```

- An expression evaluates to an integer
Abstract syntax trees

- An expression is a tree

\[
\begin{align*}
7 + 9 \times 10 \\
7 + (9 \times 10) \\
\text{(No parentheses)} \\
x \times (y + z) \times 42
\end{align*}
\]

Object-oriented representation of expressions

- A tree is an object structure
- Tree node kinds form a class hierarchy
C# classes for expressions

```csharp
public abstract class Expression { }
public class Constant : Expression {
    private readonly int value;
}
public class Variable : Expression {
    private readonly String name;
}
public class BinOp : Expression {
    private readonly Operator op;
    private readonly Expression e1, e2;
}
public class UnOp : Expression {
    private readonly Operator op;
    private readonly Expression e1;
}
public enum Operator { Add, Sub, Mul, ... }
```

Example expressions

```
9 * 10
new BinOp(Operator.MUL,
    new Constant(9),
    new Constant(10))
```
In SML we would use a datatype

datatype expr =
  Constant of int
| Variable of string
| UnOp of operator * expr
| BinOp of operator * expr * expr

datatype operator =
  Add | Sub | Mul | ...

- Expression representation is very similar

9 * 10

BinOp(MUL,
  Constant(9),
  Constant(10))

Evaluation of expressions, 1

public abstract class Expression {
  abstract public int Eval(REnv env);
}

public class Constant : Expression {
  private readonly int value;
  public override int Eval(REnv env) {
    return value;
  }
}

public class Variable : Expression {
  private readonly String name;
  public override int Eval(REnv env) {
    return env.GetVariable(name).value;
  }
}
Evaluation of expressions, 2

public class BinOp : Expression {
    ...
    public override int Eval(REnv env) {
        int v1 = e1.Eval(env);
        int v2 = e2.Eval(env);
        switch (op) {
            case Operator.Add:
                return v1 + v2;
            case Operator.Div:
                return v1 / v2;
            case Operator.Mul:
                return v1 * v2;
            ...
        }
    }
}

The runtime environment maps
variable to storage and value

public class REnv {
    private readonly Stack<Pair<String, Storage>> locals;
    public Storage GetVariable(String name) {
        ...
    }
}

public class Storage {
    public int value = 0;
}

• The storage can hold an int

    env.GetVariable("x").value = 42;

    return env.GetVariable("x").value;

    Set value

    Get value
The runtime environment is a stack of bindings

- This works well with nested scopes

```csharp
let x = 2 in let y = x+3 in y*4 end end

[$[x",2]]$  [$[y",5],[x",2]]$ 
```

```csharp
public class REnv {
    private readonly Stack<Pair<String, Storage>> locals;
    public Storage GetVariable(String name) {
        foreach (Pair<String, Storage> variable in locals)
            if (variable.Fst == name)
                return variable.Snd;
        throw new Exception("Unbound variable: " + name);
    }
}
```

Typed expressions

- We want to prohibit `(4 > 3) * 10`
- Distinguish types `bool` and `int`
- And an error type, for `(4 > 3) * 10`

```csharp
abstract public class Type {
    public static readonly Type intType = new PrimitiveType("int");
    public static readonly Type boolType = new PrimitiveType("bool");
    public static readonly Type errorType = new PrimitiveType("*ERROR*");
}
public class PrimitiveType : Type {
    public readonly String name;
}
```
Type checking of expressions

```java
public abstract class Expression {
    abstract public int Eval(REnv env);
    abstract public Type Check(TEnv env);
}
public class Constant : Expression {
    private readonly int value;
    private readonly Type type;
    public override Type Check(TEnv env) {
        return type;
    }
}
public class Variable : Expression {
    public override Type Check(TEnv env) {
        return env.GetVariable(name);
    }
}
```

Result is a Type

New

Type environment

New

Type checking of binary operators

```java
public class BinOp : Expression {
    public override Type Check(TEnv env) {
        Type t1 = e1.Check(env);
        Type t2 = e2.Check(env);
        switch (op) {
            case Operator.Add: case Operator.Div: ...
                if (t1 == Type.intType
                    && t2 == Type.intType)
                    return Type.intType;
                else
                    throw new TypeException(...);
            case Operator.Eq: case Operator.Ge: ...
                if (t1 == Type.intType
                    && t2 == Type.intType)
                    return Type.boolType;
                else
                    throw new TypeException();
        ...
    }
```
From text file to abstract syntax

- Programmers write text (source code), not abstract syntax

![Diagram showing the process from program text to abstract syntax](image)

```
7 + 9 * 10
```

Regular expressions (R.E.)

- A regular expression describes a set of strings

<table>
<thead>
<tr>
<th>R.E. r</th>
<th>Meaning</th>
<th>Language L(r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>symbol a</td>
<td>{ “a” }</td>
</tr>
<tr>
<td>ε</td>
<td>empty sequence</td>
<td>{ “” }</td>
</tr>
<tr>
<td>r₁ r₂</td>
<td>r₁ followed by r₂</td>
<td>{ s₁s₂</td>
</tr>
<tr>
<td>r *</td>
<td>zero or more r</td>
<td>{ s₁...sₙ</td>
</tr>
<tr>
<td>r₁</td>
<td>r₂ or else r₂</td>
<td>L(r₁) ∪ L(r₂)</td>
</tr>
</tbody>
</table>

- ab* represents { “a”, “ab”, “abb”, ... }
- (ab)* represents { “”, “ab”, “abab”, ... }
- (a|b)* represents { “”, “a”, “b”, “ab”, “ba” ... }
- (a|b)c* represents ?
### Regular expression abbreviations

<table>
<thead>
<tr>
<th>Abbrev</th>
<th>Meaning</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>[aeiou]</td>
<td>set</td>
<td>a</td>
</tr>
<tr>
<td>[0-9]</td>
<td>range</td>
<td>0</td>
</tr>
<tr>
<td>[a-zA-Z]</td>
<td>ranges</td>
<td>a</td>
</tr>
<tr>
<td>r?</td>
<td>zero or one r</td>
<td>ε</td>
</tr>
<tr>
<td>r+</td>
<td>one or more r</td>
<td>r r*</td>
</tr>
</tbody>
</table>

- Alternative syntax (used in Coco/R):
  "aeiou" instead of "{aeiou}"
  "{ r }" instead of "r*"

### Examples

- Non-negative integer constants
- Integer constants
- Java variable names
- Java floating-point constants
- Pre-1990 Danish car license plates
- Internet domains (e.g. www.pol.dk)
- Email addresses
Lexers and lexer generators

• A lexer transforms a character stream into a token stream
• A lexer generator takes as input a lexer specification, and generates a lexer
• A lexer specification is a collection of regular expressions

• Coco/R is a simple combined lexer generator and parser generator

Example lexer specification (expression language, Coco/R)

CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabc...xyz".
  digit = "0123456789".
  cr = ''.
  lf = '
'.
  tab = '	'.

TOKENS
  ident = letter {letter | digit}.
  number = digit {digit}.

IGNORE cr + lf + tab
From regular expression to finite automaton

- For every regular expression $r$ there is a finite automaton that recognizes exactly the strings described by $r$
- The converse is also true
- Construction:
  Regular expression
  $\Rightarrow$ Nondeterministic finite automaton (NFA)
  $\Rightarrow$ Deterministic finite automaton (DFA)
- This gives a very efficient way of determining whether a string is described by a regular expression

Parsers and parser generators

- A parser transforms a token stream into an abstract syntax tree
- A parser generators turns a parser specification into a parser
- A parser specification is a grammar equipped with semantic actions, that is, instructions for AST building
Grammar for expressions

Expr ::= 
  n // constant
  | x // variable
  | UnOper Expr
  | Expr BinOper Expr
  | ( Expr )

UnOper ::= - | !

BinOper ::= + | - | * | /
  | == | != | < | <= | > | >=

Example parser specification

Expr<out Expression e> (. Expression e1, e2; Operator op; .)
  = 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); .)
      }
  .
RelOp<out Operator op> (. op = Operator.Bad; .)
  = ( "==" (. op = Operator.Eq; .)
       | "!=" (. op = Operator.Ne; .)
       | "<" (. op = Operator.Lt; .)
       | "<=" (. op = Operator.Le; .)
       | ">" (. op = Operator.Gt; .)
       | ">=" (. op = Operator.Ge; .)
      )
  . // more rules
Grammar classes
(Chomsky hierarchy, 1956)

- Type 3: Regular grammars; same expressiveness as regular expressions
  \[ A \rightarrow cB \quad A \rightarrow B \quad A \rightarrow c \quad A \rightarrow \varepsilon \]
- Type 2: Context-free grammars (CFG)
  \[ A \rightarrow cBd \]
- Type 1: Context-sensitive grammars, non-abbreviating rules
  \[ aAb \rightarrow acAd \]
- Type 0: Unrestricted grammars; same as term rewrite systems
  \[ 0Ay \rightarrow 0 \]

The language of a grammar

- \( L(G) = \{ \text{strings derivable from } G \} \)
- \( = \{ s \mid S \rightarrow s \} \)
  where \( S \) is the start symbol of \( G \)
- Parsing is inverse derivation:
  - Could this string be derived from \( S \)?
  - And if so, using which rules?
- There are CFG parsers, but they're slow
- Subclasses of CFG can be parsed fast:
  - LL, top-down parsers: Coco/R, ANTLR, ...
  - LR, bottom-up parsers: Yacc, Bison, Javacup, ...
  (Knuth 1968)
Wednesday

• Lecture:
  – Grammars often must be rewritten to be used as parser specifications
  – Stack machines
  – Compilation of expressions

• Exercises for Wednesday:
  – Extend the expression language
  – Extend the lexer and parser specifications
  – Extend the interpreter, checker and compiler