Programs as data
Parsing cont’d; first-order functional language, type checking

Peter Sestoft
Monday 2012-09-17
Plan for today

• Parsing:
  – LR versus LL
  – How does an LR parser work
  – Hand-writing an LL parser

• A first-order functional language
  – Lexer and parser specifications
  – Interpretation: function closures

• Explicit types
  – a type checking function
  – type rules

• Static versus dynamic types
LR versus LL parsing

• **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. fsyacc – exist
  – No grammar transformations required

• **LL**: Read input from **Left** to right, make derivations from **Leftmost** nonterminal
  – Top-down parsing
  – Fairly easy to hand-write a parser
  – *But* requires grammar transformations, to encode associativity and precedence
An LR derivation from last week

```
Main => Expr EOF A
Main => Expr + Expr EOF H
Main => Expr + Expr * Expr EOF G
Main => Expr + Expr * wk EOF B
Main => Expr + 52 * wk EOF C
Main => x + 52 * wk EOF B
```

Derivation tree
The (fsyacc, LR) parser automaton

A parser is an automaton with a stack

state 19:
items:
Expr -> Expr . 'TIMES' Expr
Expr -> Expr . 'PLUS' Expr
Expr -> Expr 'PLUS' Expr .
Expr -> Expr . 'MINUS' Expr

actions:
action 'EOF': reduce Expr --> Expr 'PLUS' Expr
action 'LPAR': reduce Expr --> Expr 'PLUS' Expr
action 'RPAR': reduce Expr --> Expr 'PLUS' Expr
action 'END': reduce Expr --> Expr 'PLUS' Expr
action 'IN': reduce Expr --> Expr 'PLUS' Expr
action 'LET': reduce Expr --> Expr 'PLUS' Expr
action 'PLUS': reduce Expr --> Expr 'PLUS' Expr
action 'MINUS': reduce Expr --> Expr 'PLUS' Expr
action 'TIMES': reduce Expr --> Expr 'PLUS' Expr
action 'EQ': reduce Expr --> Expr 'PLUS' Expr
action 'NAME': reduce Expr --> Expr 'PLUS'Expr
action 'CSTINT': reduce Expr --> Expr 'PLUS' Expr
action 'error': reduce Expr --> Expr 'PLUS' Expr
action '#': reduce Expr --> Expr 'PLUS' Expr
action '$$': reduce Expr --> Expr 'PLUS' Expr
immediate action: <none>
gotos:

File ExprPar.fsyacc.output from fsyacc -v ExprPar.fsy
### Parser stack snapshots, example

<table>
<thead>
<tr>
<th>Input</th>
<th>Parse stack (top on right)</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>x+52*wk EOF</td>
<td>#0</td>
<td>shift #4</td>
</tr>
<tr>
<td>+52*wk EOF</td>
<td>#0 x #4</td>
<td>reduce by B</td>
</tr>
<tr>
<td>+52*wk EOF</td>
<td>#0 Expr</td>
<td>goto #2</td>
</tr>
<tr>
<td>+52*wk EOF</td>
<td>#0 Expr #2</td>
<td>shift #22</td>
</tr>
<tr>
<td>52*wk EOF</td>
<td>#0 Expr #2 + #22</td>
<td>shift #22</td>
</tr>
<tr>
<td>*wk EOF</td>
<td>#0 Expr #2 + #22 52 #5</td>
<td>reduce by C</td>
</tr>
<tr>
<td>*wk EOF</td>
<td>#0 Expr #2 + #22 Expr</td>
<td>goto #19</td>
</tr>
<tr>
<td>*wk EOF</td>
<td>#0 Expr #2 + #22 Expr #19</td>
<td>shift #21</td>
</tr>
<tr>
<td>wk EOF</td>
<td>#0 Expr #2 + #22 Expr #19 * #21</td>
<td>shift #4</td>
</tr>
<tr>
<td>EOF</td>
<td>#0 Expr #2 + #22 Expr #19 * #21 wk #4</td>
<td>reduce by B</td>
</tr>
<tr>
<td>EOF</td>
<td>#0 Expr #2 + #22 Expr #19 * #21 Expr</td>
<td>goto #18</td>
</tr>
<tr>
<td>EOF</td>
<td>#0 Expr #2 + #22 Expr #19 * #21 Expr #18</td>
<td>reduce by G</td>
</tr>
<tr>
<td>EOF</td>
<td>#0 Expr #2 + #22 Expr</td>
<td>goto #19</td>
</tr>
<tr>
<td>EOF</td>
<td>#0 Expr #2 + #22 Expr</td>
<td>reduce by H</td>
</tr>
<tr>
<td>EOF</td>
<td>#0 Expr</td>
<td>goto #2</td>
</tr>
<tr>
<td>EOF</td>
<td>#0 Expr #2</td>
<td>shift 3</td>
</tr>
<tr>
<td>#0 Expr #2 EOF #3</td>
<td></td>
<td>reduce by A</td>
</tr>
<tr>
<td>#0 Main</td>
<td></td>
<td>goto #1</td>
</tr>
<tr>
<td>#0 Main #1</td>
<td></td>
<td>accept</td>
</tr>
</tbody>
</table>

- Order of reduce actions is reverse LR derivation!
Parser state and actions

• Parser state = parser stack, containing:
  – Parser state numbers: \#n
  – Grammar symbols: terminals and nonterminals

Parser actions:

• *Shift*: read a symbol from input onto the stack, and go to new state

• *Reduce*: take grammar rule rhs symbols off the stack and replace them by its lhs nonterminal, and evaluate a semantic action

• *Goto*: go to a new parser state (after reduce)
Shift/reduce conflicts

- Sometimes the parser generator does not know whether to shift or to reduce
- Especially if the grammar is ambiguous
- Then warnings are issued by \texttt{fsyacc}

- To resolve shift/reduce conflicts, change the parser specification
- To understand how, study the parser automaton in \texttt{ExprPar.fsyacc.output}
LL parsing, recursive descent

• Example: Scheme terms, “S-expressions”
  – symbols such as foo, bar, b52, +, *
  – numbers such as 117, -4
  – nested lists such as (foo (+ n 1))

• Grammar:

```
sexp ::= symbol
| number
| ( sexp* )
```
Hand-written lexer and parser in C#

- A token is an object implementing IToken

```csharp
interface IToken { }
class Lpar : IToken { ... }
class Rpar : IToken { ... }
class Symbol : IToken {
    public readonly String name;
    ...
}
class NumberCst : IToken {
    public readonly int val;
    ...
}
```

```
IEnumerator<IToken>
```

![Diagram](image_url)
public static IEnumerator<IToken> Tokenize(TextReader rd) {
    for (;;) {
        int raw = rd.Read();
        char ch = (char)raw;
        if (raw == -1)
            yield break;
        else if (Char.IsWhiteSpace(ch))
            { }
        else if (Char.IsDigit(ch))
            yield return new NumberCst(ScanNumber(ch, rd));
        else switch (ch) {
            case '(': 
                yield return Lpar.LPAR; break;
            case ')':
                yield return Rpar.RPAR; break;
            case '-': // negative number, or symbol
                ...
            default:
                yield return ScanSymbol(ch, rd);
                break;
        } } } 

Helper method to make a number token
Parsing S-expressions top-down

- To parse S-expression:
- If next token is Symbol, then success
- If next token is NumberCst, then success
- If next token is Lpar, then
  - read that token
  - while next token is not Rpar
    - parse an S-expression
- If next token is anything else, then error
Handwritten recursive descent parser

```csharp
public static void ParseSexp(IEnumerator<IToken> ts) {
    if (ts.Current is Symbol) {
        Console.WriteLine("Parsed symbol " + ts.Current);
    } else if (ts.Current is NumberCst) {
        Console.WriteLine("Parsed number " + ts.Current);
    } else if (ts.Current is Lpar) {
        Console.WriteLine("Started parsing list");
        Advance(ts);
        while (!(ts.Current is Rpar)) {
            ParseSexp(ts);
            Advance(ts);
        }
        Console.WriteLine("Ended parsing list");
    } else
    throw new ArgumentException("Parse error");
}

private static void Advance(IEnumerator<IToken> ts) {
    if (!ts.MoveNext())
        throw new ArgumentException("Unexpected eof");
}
```
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
  \[ aAb \rightarrow acAdb \]

- **Type 0**: Unrestricted grammars; same as term rewrite systems
  \[ 0Ay \rightarrow 0 \]
Micro-ML: A small functional language

- First-order: A value cannot be a function
- Dynamically typed, so this is OK:
  
  ```
  if true then 1+2 else 1+false
  ```
- Eager, or call-by-value: In a call $f(e)$ the argument $e$ is evaluated before $f$ is called
- Example Micro-ML programs (an F# subset):

  ```
  5+7

  let f x = x + 7 in f 2 end

  let fac x = if x=0 then 1 else x * fac(x - 1) in fac 10 end
  ```
Abstract syntax of Micro-ML

type expr =
  | CstI of int
  | CstB of bool
  | Var of string
  | Let of string * expr * expr
  | Prim of string * expr * expr
  | If of expr * expr * expr
  | Letfun of string * string * expr * expr
  | Call of expr * expr

let f x = x + 7 in f 2 end

Letfun ("f", "x", Prim ("+",Var "x",CstI 7), Call (Var "f",CstI 2))
Runtime values, function closures

- Run-time values: integers and functions

```ocaml
type value =
  | Int of int
  | Closure of string * string * expr * value env
```

- **Closure**: a package of a function’s body and its declaration environment

- A name should refer to a *statically* enclosing binding:

```ocaml
let y = 11
in let f x = x + y
  in let y = 22 in f 3 end
end
```

Evaluate as

```
3 + y
```

Should always have value 11

```
(f, x, x+y, [(y,11)])
```
Interpretation of Micro-ML

- Constants, variables, primitives, let, if: as for expressions
- Letfun: Create function closure and bind f to it
- Function call f(e):
  - Look up f, it must be a closure
  - Evaluate e
  - Create environment and evaluate the function’s body

```ocaml
let rec eval (e : expr) (env : value env) : int =
  match e with
  | ... |
  | Letfun(f, x, fBody, letBody) ->
    let bodyEnv = (f, Closure(f, x, fBody, env)) :: env
    in eval letBody bodyEnv
  | Call(Var f, eArg) ->
    let fClosure = lookup env f
    in match fClosure with
        | Closure (f, x, fBody, fDeclEnv) ->
          let xVal = Int(eval eArg env)
          let fBodyEnv = (x, xVal) :: (f, fClosure) :: fDeclEnv
          in eval fBody fBodyEnv
        | _ -> failwith "eval Call: not a function"
```
Dynamic scope (instead of static)

• With static scope, a variable refers to the lexically, or statically, most recent binding.

• With **dynamic scope**, a variable refers to the dynamically most recent binding:

```haskell
let y = 11
in let f x = x + y
    in let y = 22 in f 3 end
end
end
```

Evaluate as:

```
3 + y
```
A dynamic scope variant of Micro-ML

- Very minimal change in interpreter:
  ```ml
  let rec eval (e : expr) (env : value env) : int =
    ...
    | Call(Var f, eArg) ->
      let fClosure = lookup env f
      in match fClosure with
        | Closure (f, x, fBody, fDeclEnv) ->
          let xVal = Int(eval eArg env)
          let fBodyEnv = (x, xVal) :: (f, fClosure) :: env
          in eval fBody fBodyEnv
  ```
  Evaluate fBody in call environment

- fDeclEnv is ignored; function is just \((f, x, fBody)\)
- Good and bad:
  - simple to implement (no closures needed)
  - makes type checking difficult
  - makes efficient implementation difficult
- Used in macro languages, and Lisp, Perl, Clojure
Lexer and parser for Micro-ML

• Lexer:
  – Nested comments, as in F#, Standard ML

1 + (* 33 (* was 44 *) *) 22

• Parser:
  – To parse applications e1 e2 e3 correctly, distinguish atomic expressions from others

• Problem: f(x-1) parses as f(x(-1))

• Solution:
  – FunLex.fsl: make CSTINT just [0-9]+ without sign
  – FunPar.fsy: add rule Expr := MINUS Expr
An explicitly typed functional language

```ocaml
let f (x : int) : int = x+1
in f 12 end

letfun("f", "x", TypI,
  Prim("+", Var "x", CstI 1), TypI,
  Call(Var "f", CstI 12));;

type typ =
  | TypI
  | TypB
  | TypF of typ * typ

(f, x, xTyp, fBody, rTyp, letBody)
```

```
type tyexpr =
  | CstI of int
  | CstB of bool
  | Var of string
  | Let of string * tyexpr * tyexpr
  | Prim of string * tyexpr * tyexpr
  | If of tyexpr * tyexpr * tyexpr
  | Letfun of string * string * typ * tyexpr * typ * tyexpr
  | Call of tyexpr * tyexpr

(TypF(TypI, TypI))
```
Type checking by recursive function

- Using a type environment [\(\text{"x", TypI}\)]:

```ml
let rec typ (e : tyexpr) (env : typ env) : typ =
  match e with
  | CstI i -> TypI
  | CstB b -> TypB
  | Var x -> lookup env x
  | Prim(ope, e1, e2) ->
    let t1 = typ e1 env
    let t2 = typ e2 env
    in match (ope, t1, t2) with
      | ("*", TypI, TypI) -> TypI
      | ("+", TypI, TypI) -> TypI
      | ("-", TypI, TypI) -> TypI
      | ("=", TypI, TypI) -> TypB
      | ("<", TypI, TypI) -> TypB
      | ("&&", TypB, TypB) -> TypB
      | _    -> failwith "unknown primitive, or type error"
      | ...  
```
Type checking, part 2

• Checking `let x=eRhs in letBody end`
• Checking `if e1 then e2 else e3`

```plaintext
let rec typ (e : tyexpr) (env : typ env) : typ =
  match e with
  | Let(x, eRhs, letBody) ->
    let xTyp = typ eRhs env
    let letBodyEnv = (x, xTyp) :: env
    in typ letBody letBodyEnv
  | If(e1, e2, e3) ->
    match typ e1 env with
    | TypB -> let t2 = typ e2 env
     let t3 = typ e3 env
     in if t2 = t3 then t2
     else failwith "If: branch types differ"
    | _    -> failwith "If: condition not boolean"
    | ...  
```
Type checking, part 3

• Checking `let f x=eBody in letBody end`

• Checking `f eArg`

```ml
let rec typ (e : typexpr) (env : typ env) : typ =
    match e with
    | ... |
    | Letfun(f, x, xTyp, fBody, rTyp, letBody) ->
      let fTyp = TypF(xTyp, rTyp)
      let fBodyEnv = (x, xTyp) :: (f, fTyp) :: env
      let letBodyEnv = (f, fTyp) :: env
      if typ fBody fBodyEnv = rTyp then typ letBody letBodyEnv
      else failwith "Letfun: wrong return type in function"
    | Call(Var f, eArg) ->
      match lookup env f with
      | TypF(xTyp, rTyp) ->
        if typ eArg env = xTyp then rTyp
        else failwith "Call: wrong argument type"
      | _ -> failwith "Call: unknown function"
    | Call(_, eArg) -> failwith "Call: illegal function in call"
```
Type checking versus evaluation

- The type checker \texttt{typ} and the interpreter \texttt{eval} have similar structure.
- Type checking can be thought of as \textit{abstract interpretation} of the program.
- We calculate “TypI + TypI gives TypI” instead of “Int 3 + Int 5 gives Int 8”.
- One major difference:
  - Type checking a function call \( f(e) \) does not require type checking the function’s body again.
  - Interpreting a function call \( f(e) \) does require interpreting the function’s body.
- Type checking always terminates.
Type checking by logical rules

\[
\begin{align*}
\rho \vdash i : \text{int} \\
\rho \vdash b : \text{bool} \\
\rho(x) = t \\
\rho \vdash x : t \\
\rho \vdash e_1 : \text{int} \quad \rho \vdash e_2 : \text{int} \\
\rho \vdash e_1 + e_2 : \text{int} \\
\rho \vdash e_1 < e_2 : \text{bool} \\
\rho \vdash e_r : t_r \quad \rho[x \mapsto t_r] \vdash e_b : t \\
\rho \vdash \text{let } x = e_r \text{ in } e_b \text{ end } : t \\
\rho \vdash e_1 : \text{bool} \quad \rho \vdash e_2 : t \quad \rho \vdash e_3 : t \\
\rho \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \\
\rho[x \mapsto t_x, f \mapsto t_x \rightarrow t_r] \vdash e_r : t_r \quad \rho[f \mapsto t_x \rightarrow t_r] \vdash e_b : t \\
\rho \vdash \text{let } f(x : t_x) = e_r : t_r \text{ in } e_b : t \\
\rho(f) = t_x \rightarrow t_r \quad \rho \vdash e : t_x \\
\rho \vdash fe : t_r
\end{align*}
\]
How to read a type rule

- IF
  - in environment $\rho$, expression $e_1$ has type int, and
  - in environment $\rho$, expression $e_2$ has type int
- THEN
  - environment $\rho$, expression $e_1 < e_2$ has type bool
Joint exercise: How read these?

\[ \rho \vdash i : \text{int} \]

\[ \rho(x) = t \]
\[ \rho \vdash x : t \]

\[ \rho \vdash e_1 : \text{bool} \quad \rho \vdash e_2 : t \quad \rho \vdash e_3 : t \]
\[ \rho \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \]

\[ \rho \vdash e_r : t_r \quad \rho[x \mapsto t_r] \vdash e_b : t \]
\[ \rho \vdash \text{let } x = e_r \text{ in } e_b \text{ end} : t \]
Combining type rules to trees

- Stacking type rules on top of each other
- One rule’s conclusion is another’s premise
- Checking `let x=1 in x<2 end : bool` in some environment $\rho$:

$$
\begin{align*}
\rho[x \mapsto \text{int}] & \vdash x : \text{int} \\
\rho[x \mapsto \text{int}] & \vdash 2 : \text{int}
\end{align*}
$$

$$
\begin{align*}
\rho & \vdash 1 : \text{int} \\
\rho[x \mapsto \text{int}] & \vdash x < 2 : \text{bool}
\end{align*}
$$

$$
\rho \vdash \text{let } x = 1 \text{ in } x < 2 \text{ end : bool}
$$

- The `typ` function implements the rules, from conclusion to premise!
Joint exercises: Invent type rules

- For $e_1 \ & \ & e_2$ (logical and)
- For $e_1 :: e_2$ (list cons operator)
- For \texttt{match e with} \ [\] $\rightarrow e_1$ | $x::xr$ $\rightarrow e_2$
Evaluation by logical rules

\[
\begin{align*}
\frac{}{\rho \vdash i \Rightarrow i} & \quad (e1) \\
\frac{}{\rho \vdash b \Rightarrow b} & \quad (e2) \\
\frac{}{\rho(x) = v} & \quad (e3) \\
\frac{\rho \vdash e_1 \Rightarrow v_1 \quad \rho \vdash e_2 \Rightarrow v_2 \quad v = v_1 + v_2}{\rho \vdash e_1 + e_2 \Rightarrow v} & \quad (e4) \\
\frac{\rho \vdash e_1 \Rightarrow v_1 \quad \rho \vdash e_2 \Rightarrow v_2 \quad b = (v_1 < v_2)}{\rho \vdash e_1 < e_2 \Rightarrow b} & \quad (e5) \\
\frac{\rho \vdash e_r \Rightarrow v_r \quad \rho[x \mapsto v_r] \vdash e_b \Rightarrow v}{\rho \vdash \text{let } x = e_r \text{ in } e_b \text{ end} \Rightarrow v} & \quad (e6) \\
\frac{\rho \vdash e_1 \Rightarrow \text{true} \quad \rho \vdash e_2 \Rightarrow v}{\rho \vdash \text{if } \text{true} \text{ then } e_2 \text{ else } e_3 \Rightarrow v} & \quad (e7t) \\
\frac{\rho \vdash e_1 \Rightarrow \text{false} \quad \rho \vdash e_3 \Rightarrow v}{\rho \vdash \text{if } \text{false} \text{ then } e_2 \text{ else } e_3 \Rightarrow v} & \quad (e7f)
\end{align*}
\]

In environment $\rho$, expression $x$ evaluates to $v$
Evaluation by logical rules: Function declaration and call

• Compare these with the eval interpreter:

\[
\frac{\rho[f \mapsto (f, x, e_r, \rho)] \vdash e_b \Rightarrow v}{\rho \vdash \text{let } f(x) = e_r \text{ in } e_b \text{ end} \Rightarrow v} (e8)
\]

\[
\rho(f) = (f, x, e_r, \rho_{f\text{decl}}) \quad \rho \vdash e \Rightarrow v_x \quad \rho_{f\text{decl}}[x \mapsto v_x, f \mapsto (f, x, e_r, \rho_{f\text{decl}})] \vdash e_r \Rightarrow v \tag{e9}
\]

• Also, note recursive evaluation of f's body; no such thing in the type rules
Dynamically or statically typed

• Dynamically typed:
  – Types are checked during evaluation (micro-ML, Postscript, JavaScript, Python, Ruby, Scheme, ...)

```plaintext
true { 11 } { 22 false add } ifelse =
```

• Statically typed:
  – Types are checked before evaluation (our typed fun. language, F#, most of Java and C#)

```plaintext
if true then 11 else 22+false
true ? 11 : (22 + false)
```

OK, gives 11

Type error

Type error
Dynamic typing in Java/C# arrays

• For a Java/C# array whose element type is a reference type, all assignments are type-checked at runtime

```java
void M(Object[] arr, Object x) {
    arr[0] = x;
}
```

• Why is that necessary?

```java
String[] ss = new String[1];
M(ss, new Object());
String s0 = ss[0];
```
Reading and homework

• This week’s lecture:
  – PLCSD chapter 4
  – Mogensen ICD 2011 section 2.11, 2.12, 2.16
    (or Mogensen 2010 sections 3.12, 3.17)
  – Exercises 4.1, 4.2, 4.3, 4.4, 4.5 for Wed 26 Sep

• Next week’s lecture:
  – PLCSD chapter 5.1-5.4 and chapter 6