

Combinatory parsing

Fritz Henglein
DIKU

March 31, 2003

DIKU Project Seminar

The basic idea

For each language (identified by grammar symbol S), define a function f_S that, given an input stream i , recognizes a prefix of i that is an element of $L(S)$ and returns a result for it, together with the remainder of the input stream:

type 'a parser = istream -> ('a, istream)

March 31, 2003

DIKU Project Seminar

Sources

- W.H. Burge, "Recursive Programming Techniques", Addison-Wesley, 1975
- Hutton, "Higher-order functions for parsing", J. Functional Programming 2(3):323-343.
- L.C. Paulson, "ML for the Working Programmer, 2d ed.", Cambridge University Press, 1996**

March 31, 2003

DIKU Project Seminar

The basic idea (more general)

Define a function that returns the list of all possible parses and their results of some prefix of a given input stream.

*type 'a Parser = istream -> ('a * istream) list*

March 31, 2003

DIKU Project Seminar

Combinatory parsing: Basics

- The basic idea
- Primitive parsers
- Fundamental parser combinators
- Useful parser combinators
- Pragmatics of functional parsing
- Monadic parser combinators

March 31, 2003

DIKU Project Seminar

Primitive parsers

- Recognize a specific character
- Recognize a character with a given property
- Recognize a specific string
- Recognize everything until some string

March 31, 2003

DIKU Project Seminar

Recognize a specific character

```
fun $ c charStream =
  case get charStream of
  SOME (res as (c', charStream')) =>
    if c = c'
    then res
    else raise SyntaxError
         (String.str c ^ " expected", charStream)
  | NONE =>
    raise SyntaxError
      ("Unexpected end of file ($)", charStream)
```

March 31, 2003

DIKU Project Seminar

Fundamental parser combinators

- Sequencing
- Alternation
- Empty
- Process parse result

March 31, 2003

DIKU Project Seminar

Recognize a character with a given property

```
fun getIf charPred charStream =
  case get charStream of
  SOME (res as (c, charStream')) =>
    if charPred c
    then res
    else raise SyntaxError
         (String.str c ^ " unexpected", charStream)
  | NONE =>
    raise SyntaxError
      ("Unexpected end of file", charStream)
```

March 31, 2003

DIKU Project Seminar

Sequencing

```
fun (pf1 -- pf2) charStream =
  let val (res1, charStream1) = pf1 charStream
      val (res2, charStream2) = pf2 charStream1
  in ((res1, res2), charStream2)
  end
```

March 31, 2003

DIKU Project Seminar

Recognize a specific string

```
fun $$ s charStream =
  if isPrefix s charStream
  then (s, triml (size s) charStream)
  else raise SyntaxError (s ^ " expected", charStream)
```

March 31, 2003

DIKU Project Seminar

Alternation

```
fun (pf1 || pf2) charStream =
  pf1 charStream
  handle SyntaxError _ =>
    pf2 charStream
```

March 31, 2003

DIKU Project Seminar

Empty

```
fun empty charStream = (nil, charStream)
```

March 31, 2003

DIKU Project Seminar

Example

- ✧ XML parsing
 - ✧ XML grammar
 - ✧ Example XML parser (context-free part only)

March 31, 2003

DIKU Project Seminar

Process parse result

```
fun (pf >> f) charStream =  
  let val (res, charStream') = pf charStream  
  in (f res, charStream')  
  end
```

March 31, 2003

DIKU Project Seminar

Pros

- ✧ Define many parsers, not just one parser; phrase parsers can be applied and reused independently of 'global' parser
- ✧ Parsers can be put in general purpose library and (re)used in any context (necessary requirement: functional interface, no implicit state, all information passed by argument)

March 31, 2003

DIKU Project Seminar

EBNF parser combinators

```
fun const x _ = x  
fun middle (_, x, _) = x  
  
fun repeat p = p -- repeat p >> op:: || empty  
fun repeatOne = p -- repeat p >> op::  
fun optional p = p >> SOME || empty >> const NONE  
fun pack (p1, p2) p = p1 -- p -- p2 >> middle  
fun listOf sp p = p -- repeat (sp -- p >> #2) >> op::
```

March 31, 2003

DIKU Project Seminar

Pros...

- ✧ Monadic (left-to-right propagation) combinatory parsing allows passing semantic information to parser (e.g. "DECIMAL-POINT IS COMMA" in COBOL)
- ✧ Allows parametric definitions; that is, parsers parameterized by other parsers (listOf, etc.); supports building of and (re)use of libraries

March 31, 2003

DIKU Project Seminar

Pros...

- ⌘ Allows lookahead through (limited) backtracking: in some cases relatively efficiently:
 - ⌘ few alternatives, the first one mostly taken;
 - ⌘ few alternatives, predictive (LL(1)) grammar (failure detected rapidly)
- ⌘ Allows interesting extensions to conventional parsing (e.g. requirement to parse in two different ways)

March 31, 2003

DIKU Project Seminar

Cons

- ⌘ Lack of efficiency (parsing in general, and lexing in particular, are often on the critical performance path!)
- ⌘ Have to be careful with imperative semantic actions.
- ⌘ Error reporting and recovery difficult due to backtracking

March 31, 2003

DIKU Project Seminar

Pros...

- ⌘ Grammar development and testing can go hand-in-hand (phrase parsing supported, no requirement to invoke parser generator, no explicit hand-coding of look-ahead sets as in predictive [recursive-descent] parsing).
- ⌘ Lexers treated by combinatory parsing have potential advantages:
 - ⌘ parser state (and, in the case of monadic combinators, semantic state) is passed to lexer;
 - ⌘ parser state specific lexers are often very small, mostly deterministic (e.g., "look for symbol XYZ").

March 31, 2003

DIKU Project Seminar

Cons...

- ⌘ Conventional lexers support efficient lexical debugging since they can be used to parse the program as

```
p ::= t*
t ::= <any lexical token>
```

(Indeed, this is the canonical program, with only one parse state, lexers are basically "made" for.) This is practically impossible by functional parsing because of large degree of alternation.

March 31, 2003

DIKU Project Seminar

Pros...

- ⌘ Parsers are typically very small, good for mobile code, small languages.
- ⌘ Parsers can be generated rapidly and lazily ("just-in-time"); good for universal parsing, low upstart costs; e.g. validating XML-parsers.
- ⌘ Universal parsing supported.
- ⌘ Behaves in practice "almost" like predictive parser, without requiring grammar limitations and need to explicate lookahead set.

March 31, 2003

DIKU Project Seminar

Cons...

- ⌘ Immature technology:
 - ⌘ Subpar performance
 - ⌘ Often not taught
 - ⌘ Research focus on foundations and generalizations, not applications (with notable exceptions)

March 31, 2003

DIKU Project Seminar

Research and development suggestions

- ✍ Comments on current parsing technology
- ✍ Parsing methods and tools for improved software development

March 31, 2003

DIKU Project Seminar

Parsing methods and tools for improved software development

- ✍ Improve performance of functional parsers by:
 - ✍ abandoning full backtracking;
 - ✍ adding caching and dynamic dispatch generation
 - ✍ just-in-time table generation (e.g. Swierstra, Duponcheel; Chakravarty)
 - ✍ run-time code generation

March 31, 2003

DIKU Project Seminar

Comments on current parsing technology

- ✍ There is good support for specifying regular expressions and generating processors (lexers) for those (e.g. `{[a-z-]*}` lex, perl). Those processors, however:
 - ✍ are restricted to processing specific character sets;
 - ✍ have a narrow interface to parser/semantic processors;
 - ✍ suffer from slow generation and state explosion (partially due to their use as 'universal' lexers for a given language);
 - ✍ cannot be used directly for (nonregular) context-free grammars; doing so requires explicit programming and is error-prone (e.g. XML-parsing using a SAX-lexer).

March 31, 2003

DIKU Project Seminar

Better tools...

- ✍ Classical parser/lexer architecture should be reconsidered wrt. division of labor between parser and lexer:
 - ✍ parallel processing of multiple characters (e.g. 32-bit entities);
 - ✍ flexible, specialized parser data structures for 0-lookahead, 1-lookahead, dynamic-lookahead cases (e.g. Jikes technology, ANTLR, GLR, functional parsers); dynamic rearrangement of data structures (e.g. splay trees) for common cases, dynamic specialization of functional parsers.

March 31, 2003

DIKU Project Seminar

Comments...

- ✍ Lexers support (arbitrary) look-ahead, but resort to relexing from scratch from last success prefix upon failure.
- ✍ Most parser generators require resolution of parser actions based on one-token lookahead. This requires both a lexer (so that one token extends as far into the right context as possible) and carefully engineered grammars (which spoils the semantic structure of the language).

March 31, 2003

DIKU Project Seminar

Better tools...

- ✍ Improvements that keep the compositional, and context-independent nature of functional parsers:
 - ✍ For each choice operator choice $[p_1, \dots, p_n]$ maintain an array which maps a character c (lookahead symbol) to an index i , such that p_j is guaranteed to fail on any input starting with c , for $1 \leq j < i$. This array is maintained dynamically during execution. (NB: This optimization is only valid for (purely) functional parsers that have a "prefix failure property".)
 - ✍ Load and save the array, as defined above at parser start; better yet, serialize parser after each run (the parser 'learns' from experience) and becomes increasingly more efficient.

March 31, 2003

DIKU Project Seminar

Parsing methods and tools for improved software development

- Parser/lexer architecture should be reconsidered wrt.: parallel processing of multiple characters (e.g. 32-bit entities); flexible
- parser data structures for 0-lookahead, 1-lookahead,
- dynamic-lookahead cases (Jikes); dynamic rearrangement of data
- structure (e.g. splay trees) for common cases, dynamic

March 31, 2003 DIKU Project Seminar

Literature...

- Partridge, Wright, "Parser combinators need four values to report errors", TR, DCS, U. Tasmania, 1994
- Røjemo, "Garbage collection, and memory efficiency, in lazy functional languages", Ph.D. thesis, Chalmers, 1995
- Chakravarty, "Lazy lexing is fast", 1999, <http://www.score.is.tsukuba.ac.jp/~chak/ctk/>
- Swierstra, Duponcheel, "Deterministic, Error-Correcting Combinator Parsers", ...

March 31, 2003 DIKU Project Seminar

Literature

- Burge, "Recursive Programming Techniques", Addison-Wesley, 1975
- Hutton, "Higher-Order Functions for Parsing", JFP 2(3):323-343, July 1992
- Hutton, Meijer, "Monadic Parser Combinators", TR NOTTCS-TR-96-4
- Hutton, Meijer, "Monadic Parsing in Haskell", JFP 1993

March 31, 2003 DIKU Project Seminar

Literature...

- Gill, Marlow, "Happy: the parser generator for Haskell", U. Glasgow, 1995

March 31, 2003 DIKU Project Seminar

Literature...

- Fokker, "Functional Parsers", summer school lecture notes, 1995
- Jeuring, Swierstra, "Grammars and parsing", lecture notes, 1999
- Paulson, "ML for the Working Programmer", 1996, chapter 10

March 31, 2003 DIKU Project Seminar

Conclusion

- Functional parsing:
 - supports incremental development of language processing, effective reuse, high quality/safety;
 - is presently interesting technology for rapid prototyping and rapid development of 'small' applications;
 - appears improvable for industrial-strength applications.

March 31, 2003 DIKU Project Seminar

What is software design? Definition of...

- ∞ **Organization** of a software system into modules/components/classes or other units
- ∞ **Behavior** and **responsibilities** of units
- ∞ **Interactions** and **collaborations** between units

March 31, 2003

DIKU Project Seminar

UML Notations

- ∞ Use cases
- ∞ Class diagrams
- ∞ Interaction diagrams:
 - ∞ Sequence diagrams
 - ∞ Collaboration diagrams
- ∞ State diagrams
- ∞ Activity diagrams
- ∞ Physical diagrams (no more on those)

March 31, 2003

DIKU Project Seminar

UML: Unified Modeling Language

- ∞ Drawing notations for supporting software design process and conveying designs
- ∞ “Bubbles and arcs”:
 - ∞ Meant to be used informally and selectively;
 - ∞ No unique, definite semantics.
- ∞ CASE tool: drawing tool, plus more; e.g., automatic code generation (Fowler: Visio + MS Word = gets you far)

March 31, 2003

DIKU Project Seminar

General design techniques

- ∞ Design **testability** and systematic test into the software (“self-testing software”)
- ∞ **Refactoring**: redesigning (“cleaning up”) existing code without adding functionality
- ∞ Use **design patterns**: Fosters reuse, communication and process and result predictability
- ∞ Recommendations:
 - ∞ Don’t combine refactoring with adding functionality
 - ∞ Refactor rather often and in small doses

March 31, 2003

DIKU Project Seminar

Rational Unified Process

1. **Inception**: Business rationale, scope
2. **Elaboration**: Requirements, high-level analysis and design of baseline architecture, construction plan, risk management
3. **Construction**
 - ∞ iteration 1: use cases A,B,C implemented
 - ∞ iteration 2: use cases D,E implemented
 - ∞ ...
4. **Transition**: Beta testing, performance tuning, user training.

March 31, 2003

DIKU Project Seminar

Use cases

- ∞ **Scenario**: sequence of steps describing interaction between user and system
- ∞ **Use case**: set of scenarios tied together by a common user goal
- ∞ **Use case notation**:
 - ∞ Name (user goal)
 - ∞ Sequence of numbered steps (main scenario)
 - ∞ Collection of alternatives (alternative scenarios)
- ∞ **Use case diagram**: stick man drawings

March 31, 2003

DIKU Project Seminar

Use cases...

- ∞ Present *external view* of a system ('what', not 'how')
- ∞ Used for:
 - ∞ Requirements capture (during elaboration)
 - ∞ Planning and controlling iterative construction phase

March 31, 2003

DIKU Project Seminar

Constraint rules

- ∞ Express class invariants (invariants that must always hold: after initialization, and after each method call)
- ∞ Design by Contract:
 - ∞ Pre- and postconditions for method invocations
 - ∞ Class invariants
 - ∞ Specify important semantic properties, not expressible in underlying programming language type system

March 31, 2003

DIKU Project Seminar

Class Diagrams

- ∞ **Class diagram:** describes types of *objects* in a system and their static *relationships*
- ∞ **Class:** node (box) showing attributes and operations of a class
- ∞ Relationships:
 - ∞ **Association:** analogous to relationships in entity-relationship model. ("A customer rents videos.")
 - ∞ **Subtype:** conceptual subset relation ("A nurse is a person.")

March 31, 2003

DIKU Project Seminar

Class diagrams...

- ∞ Used for:
 - ∞ Domain modeling (elaboration)
 - ∞ Construction planning (detailed design)
 - ∞ Coding (implementation description)
- ∞ CRC cards:
 - ∞ class-responsibility-collaboration
 - ∞ used for domain modeling (arriving at a conceptual class model)

March 31, 2003

DIKU Project Seminar

Perspectives

- ∞ Class diagrams are used in different stages and with different *perspectives*:
 - ∞ **Conceptual** (for domain modeling): Concepts in the domain under study and their relations (think 'things out there', not software)
 - ∞ **Specification** (for software architecture): Description of class interfaces and class relationships (interfaces and constraints only, no concrete code)
 - ∞ **Implementation** (for code description): Description of implementation (think code).

March 31, 2003

DIKU Project Seminar

Sequence diagrams

- ∞ Description of execution behavior (message sends and receives) among a (fixed, finite) collection of objects
- ∞ Each object is shown with its own thread of control, even if execution is sequential (with send-receive = call-return)

March 31, 2003

DIKU Project Seminar

Collaboration diagrams

- ≈ Different way of presenting sequence diagrams

March 31, 2003

DIKU Project Seminar

How to use UML

- ≈ Use to support software development process for communication across teams (same time) and across time (same person)
- ≈ “A picture is worth a thousand words.”

March 31, 2003

DIKU Project Seminar

State diagrams

- ≈ **State diagram:**
 - ≈ shows state transitions of single object as (finite) state machine
 - ≈ transitions may be conditional
- ≈ Used to describe ‘life cycle’ of single object for objects with complex stateful behavior

March 31, 2003

DIKU Project Seminar

Patterns

- ≈ *“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution a million times over, without ever doing it the same way twice”*
Christopher Alexander, 1977

March 31, 2003

DIKU Project Seminar

Activity diagrams

- ≈ Graphical notation for join/fork calculus
- ≈ Used for:
 - ≈ displaying (concurrent) execution of a collection of objects,
 - ≈ “orchestration” of objects (workflow, mediator pattern)
 - ≈ flow chart representation of sequential program
- ≈ Activity diagrams with swimlanes: activities are mapped to objects (object = particular swimlane)

March 31, 2003

DIKU Project Seminar

Design pattern (GoF)

- ≈ Description of communicating objects and classes that are customized to solve a general design problem in a particular context
- ≈ Names, abstracts, identifies key aspects of a common design structure that makes it useful for creating a reusable object-oriented design.
- ≈ Bigger than single class, smaller than whole framework/application

March 31, 2003

DIKU Project Seminar

Design pattern elements

- ⌘ **Pattern name:** Handle for describing design problem
- ⌘ **Problem:** When to apply pattern. Explains problem and its context.
- ⌘ **Solution:** Elements making up design (relationships, responsibilities, collaborations)
- ⌘ **Consequences:** Results and trade-offs of applying pattern (implementation issues, performance, reuse etc.)

March 31, 2003

DIKU Project Seminar

General themes

- ⌘ **Loose coupling:** Add indirection to achieve decoupling, flexibility and reuse
- ⌘ **Abstraction:** Naming, encapsulation, parametrization (same purpose as above)
- ⌘ **Dynamic :** Delegation instead of implementation inheritance
- ⌘ **Object-orientation:** object-oriented (re)design
- ⌘ **Principles of object-oriented design:**
 - ⌘ "Program to an interface, not an implementation"
 - ⌘ "Favor object composition over class inheritance."

March 31, 2003

DIKU Project Seminar

Design pattern description

- ⌘ **Name:** Essence in one word
- ⌘ **Intent:** Short statement
- ⌘ **Also known as:** Alias
- ⌘ **Motivation:** concrete scenario
- ⌘ **Applicability:** Situations for application
- ⌘ **Structure:** Graphical representation of design (UML)
- ⌘ **Participants:** Classes and objects
- ⌘ **Collaborations:** How participants collaborate
- ⌘ **Consequences:** Results, trade-offs of use
- ⌘ **Implementation:** Pitfalls, hints, variations
- ⌘ **Sample code**
- ⌘ **Known uses**
- ⌘ **Related patterns**

March 31, 2003

DIKU Project Seminar

Reasons for redesign

- ⌘ Creating objects by specifying class explicitly
- ⌘ Dependence on specific operations
- ⌘ Dependence on hardware/software platform
- ⌘ Dependence on object representations and implementations
- ⌘ Algorithmic dependencies
- ⌘ Tight coupling
- ⌘ Extending functionality by subclassing
- ⌘ Inability to alter classes conveniently

March 31, 2003

DIKU Project Seminar

Gang of Four patterns

- ⌘ 23 design patterns
- ⌘ Organized in 2 dimensions:
 - ⌘ **Scope:**
 - ⌘ class
 - ⌘ object
 - ⌘ **Purpose:**
 - ⌘ creational (how to create objects)
 - ⌘ structural (composition of classes and objects)
 - ⌘ behavioral (ways of interaction and distribution of responsibility)

March 31, 2003

DIKU Project Seminar

GoF patterns

- ⌘ Abstract factory
- ⌘ Builder
- ⌘ Factory method
- ⌘ Prototype
- ⌘ Singleton
- ⌘ Adapter
- ⌘ Bridge
- ⌘ Composite
- ⌘ Decorator
- ⌘ Facade
- ⌘ Flyweight
- ⌘ Proxy
- ⌘ Chain of Responsibility
- ⌘ Command
- ⌘ Interpreter
- ⌘ Iterator
- ⌘ Mediator
- ⌘ Memento
- ⌘ Observer
- ⌘ State
- ⌘ Strategy
- ⌘ Template Method
- ⌘ Visitor

March 31, 2003

DIKU Project Seminar

Structural and functional composition

- ⌘ **Composite:** Define tree-like structure, where internal and leaf nodes have same interface.
- ⌘ **Decorator:** Functional composition of objects with same interface.
- ⌘ **Chain of responsibility:** Chain objects and delegate requests along the chain.
- ⌘ **Proxy:** Define local object that delegates (most) functionality that other objects.

March 31, 2003

DIKU Project Seminar

More design patterns...

- ⌘ **Facade:** Centralize interface for collection of classes
- ⌘ **Mediator:** Localize control, make centralized controller for orchestrating collaborating objects
- ⌘ **Memento:** Checkpoint object (save state)
- ⌘ **Observer:** Publish-subscribe interface for “pushing” state changes to arbitrary number of clients
- ⌘ **State:** Object-oriented state representation for finite state machines

March 31, 2003

DIKU Project Seminar

From built-in methods to higher-order methods

- ⌘ **Command:** Turn requests into objects
- ⌘ **Interpreter:** Define (programming) language of requests
- ⌘ **Strategy:** Parameterize over different algorithms
- ⌘ **Visitor:** Receive and invoke any programmable operation on object (not just built-in methods)

March 31, 2003

DIKU Project Seminar

More design patterns...

- ⌘ **Singleton:** Make sure only one instance of a class exists during any program run (shared state)
- ⌘ **Adapter:** Connect two (existing) interfaces (have class with interface I1, and another one that needs I2)
- ⌘ **Flyweight:** Split object state into mutable (context) and immutable (flyweight) and share flyweights.
- ⌘ **Iterator:** Iterate over collection

March 31, 2003

DIKU Project Seminar