Language processing

Course "Software Language Engineering"

University of Koblenz-Landau
Department of Computer Science
Ralf Lämmel
Software Languages Team
Some bits of language processing ontology
Language definition components

- Concrete syntax (useful for parsing and humans)
- Abstract syntax (useful for most processing)
- Type system
- Extra rules
- Dynamic semantics
- Translation semantics
- Pragmatics
Concrete versus abstract syntax

Example: IF-statement

- **Concrete syntax**
  - if-statement = „if“ expr „then“ stm („else“ stm)?

- **Abstract syntax**
  - if-statement = expr stm stm

Keywords were removed.

Optionality of „else“ removed (assuming „empty“ statement).
Concrete versus abstract syntax

Example: Arithmetic expressions

- **Concrete syntax**
  - `expr = term ("+" term)*`
  - `term = factor ("*" factor)*`
  - `factor = number`

- **Abstract syntax**
  - `expr = number ("+"|"*" number)*`

Concrete syntax models priorities. Abstract syntax doesn’t.
Language processors

- Recognizer (check concrete syntax)
- Parser (map text to trees)
  - Option 1: Return concrete parse trees
  - Option 2: Return abstract syntax trees
- Imploder (map parse tree to abstract syntax trees)
- Exploder (map abstract syntax trees to parse trees)
- Unparser (map trees to text)
- Pretty printer (unparse, „prettily“)

As discussed previously
More language processors
(Components of language processing)

- Type checker
- Interpreter
- Compiler to machine or high-level language
- Software visualizers (e.g., call graph or flow chart)
- Software analyzers (e.g., uninitialized variables)
- Software transformers (e.g., dead-code elimination)
- Software metrics tools (e.g., cyclomatic complexity)
- IDE integration (coloring, navigation, ...)

How to do language processors?

What programming techniques to use?

What programming technologies to use?

- Code generators
- APIs / combinator libraries
- Metaprogramming frameworks

How to leverage language definitions?

Let’s get some data points today.
Language processors
for a simple imperative language
begin declare input : natural,
    output : natural,
    repnr : natural,
    rep : natural;
input := 14;
output := 1;
while input - 1 do
    rep := output;
    repnr := input;
    while repnr - 1 do
        output := output + rep;
        repnr := repnr - 1
    od;
    input := input - 1
od
end

Factorial function in the simple imperative language Pico
Components for Pico

- Recognizer
- Parser
- Type checker
- Interpreter
- Pretty printer
- Virtual machine
- Compiler
- Control-flow visualizer

https://github.com/slecourse/slecourse/tree/master/sources/pico/
Haskell-based components for Pico

- Recognizer
- Parser
- Type checker
- Interpreter
- Pretty printer
- Virtual machine
- Compiler
- Control-flow visualizer

https://github.com/slecourse/slecourse/tree/master/sources/pico/
Concrete syntax to be recognized

Program = "begin" Decls Stms "end" ;
Decls = "declare" {Decl ","} * ";" ;
Decl = Id ":=" Type;
Type = "natural" | "string";
Stms = {Stm ";"} * ;

Stm
= Id ":=" Expr
| "if" Expr "then" {Stm ";"} * "else" {Stm ";"} * "fi"
| "while" Expr "do" {Stm ";"} * "od";

Expr
= Id
| String
| Natural
| "(" Expr ")"
| Expr "+" Expr
| Expr "-" Expr
| Expr "|" | " Expr ;

Grammar versus recognizer

Grammar

Program = "begin" Decls Stms "end";

Haskell

program ::Recognizer
program = do
  spaces
  special "begin"
  decls
  stms
  special "end"
  eof

This is a straightforward to transcription only adding some technology-specific details like „spaces“ and „eof“.
Syntax vs. recognizer

- Syntax definition may be declarative / technology-independent.
- Unambiguous grammar needed for recognition.
  - Think of dangling else or operator priorities.
- Recognizers may be hand-written.
- Recognizers specs may be non-declarative / technology-dependent.

The same is true for parsers.
Recognizer

- Leverage parser combinators (Parsec)
  - Recognizer = functional program.
  - Parser combinators are monadic.

See Haskell code online.
import Text.Parsec

-- The type of recognition
type Recognizer = Parsec String () ()

-- Top-level recognition function
recognize :: String -> IO (Either ParseError ()

{- Context-free syntax -}

program :: Recognizer
program = do
  spaces
  special "begin"
  decls
  stms
  special "end"
  eof

decls :: Recognizer
...
Parsec

(http://www.haskell.org/haskellwiki/Parsec, 23 April 2013)

“Parsec is an industrial strength, monadic parser combinator library for Haskell. It can parse context-sensitive, infinite look-ahead grammars but it performs best on predictive (LL[1]) grammars.”

See also:
ParserT monad transformer and Parser type

data ParsecT s u m a

ParsecT s u m a is a parser with stream type s, user state type u, underlying monad m and return type a. Parsec is strict in the user state. If this is undesirable, simply use a data type like data Box a = Box a and the state type Box YourStateType to add a level of indirection.

An illustrative parser combinator

choice :: Stream s m t => [ParsecT s u m a] -> ParsecT s u m a

choice ps tries to apply the parsers in the list ps in order, until one of them succeeds. Returns the value of the succeeding parser.

Haskell-based components for Pico

- Recognizer
- Parser
- Type checker
- Interpreter
- Pretty printer
- Virtual machine
- Compiler
- Control-flow visualizer

https://github.com/slecourse/slecourse/tree/master/sources/pico/
Abstract syntax

```haskell
type Name = String
data Type = NatType | StrType
type Program = ([Decl], [Stm])
type Decl = (Name, Type)
data Stm
    = Assign Name Expr
    | IfStm Expr [Stm] [Stm]
    | While Expr [Stm]
data Expr
    = Id Name
    | NatCon Int
    | StrCon String
    | Add Expr Expr
    | Sub Expr Expr
    | Conc Expr Expr

See Haskell code online.
```

Parser

Add *synthesis* of abstract syntax terms to recognizer.

Concrete and abstract syntaxes are coupled up.

See Haskell code online.
Recognition versus parsing

```haskell
type Recognizer = Parsec String () ()

program :: Recognizer
program = do
    spaces
    special "begin"
    decls
    stms
    special "end"
    eof

type Parser x = Parsec String () x

program :: Parser Program
program = do
    spaces
    special "begin"
    ds <- decls
    ss <- stms
    special "end"
    eof
    return (ds, ss)
```

Intermediate parse trees are bound and composed to bigger parse trees that are returned.

See Haskell code online.
Haskell-based components for Pico

- Recognizer
- Parser
- Type checker
- Interpreter
- Pretty printer
- Virtual machine
- Compiler
- Control-flow visualizer

https://github.com/slecoursel/slecoursel/tree/master/sources/pico/
Type checker

Check that ...

- declarations are unambiguous;
- all referenced variables are declared;
- all expected operand types agree with actual ones;
- ...
- ...
Type system vs. checker

Type systems...
- are based on formal/declarative specifications;
- they are possibly executable (perhaps inefficiently).

Type checkers...
- implement type systems efficiently;
- they provide useful error messages.
--- Error messages

type Error = ( 
    String, -- Message
    String -- Syntactical phrase
)

--- Type environments

type Env = Map Name Type

--- Haskell code online.

typeCheckProgram :: Program -> [Error]
typeCheckDecls :: [Decl] -> [Error]
typeCheckStms :: Env -> [Stm] -> [Error]
typeCheckStm :: Env -> Stm -> [Error]
typeCheckExpr :: Env -> Expr -> Type -> [Error]
Haskell-based components for Pico

- Recognizer
- Parser
- Type checker
- Interpreter
- Pretty printer
- Virtual machine
- Compiler
- Control-flow visualizer

https://github.com/slecourse/slecourse/tree/master/sources/pico/
Interpreter

Leverage natural semantics (big-step).

Define it as a total function.

- Use a special error result.

- Error messages could also be provided that way.
Architecture of the interpreter

---

Expression values

data Value
  = IntValue Int
  | StrValue String
  deriving (Eq, Show, Read)

Variables stores

type Store = Map Name Value

Execute programs
execProgram :: Program -> Maybe Store

Execute statements
execStms :: [Stm] -> Store -> Maybe Store
eexecStm :: Stm -> Store -> Maybe Store

Evaluate expressions
evalExpr :: Expr -> Store -> Maybe Value

See Haskell code online.
Haskell-based components for Pico

- Recognizer
- Parser
- Type checker
- Interpreter

- Pretty printer
- Virtual machine
- Compiler
- Control-flow visualizer

https://github.com/slecourse/slecourse/tree/master/sources/pico/
Pretty printer

- Map abstract syntax to text.
- Composable documents.
  - Vertical / horizontal composition.
  - Indentation.
- Another case of a combinator library (= DSL).

See Haskell code online.
Haskell-based components for Pico

- Recognizer
- Parser
- Type checker
- Interpreter
- Pretty printer
- Virtual machine
- Compiler
- Control-flow visualizer

https://github.com/slecourse/slecourse/tree/master/sources/pico/
Assembly code

- Stack-based assembly language.
- Explicit notion of label and gotos.
- Could be translated to Bytecode or x86 or ....
type Label = String

type Name = String

data Instr
  = DclInt Name
    -- Reserve a memory location for an integer variable
  | DclStr Name
    -- Reserve a memory location for a string variable
  | PushNat Int
    -- Push integer constant on the stack
  | PushStr String
    -- Push string constant on the stack
  | Rvalue Name
    -- Push the value of a variable on the stack
  | Lvalue Name
    -- Push the address of a variable on the stack
  | AssignOp
    -- Assign value on top, to variable at address top-1
  | AddOp
    -- Replace top two stack values by their sum
  | SubOp
    -- Replace top two stack values by their difference
  | ConcOp
    -- Replace top two stack values by their concatenation
  | Label Label
    -- Associate a label with the next instruction
  | Go Label
    -- Go to instruction with given label
  | GoZero Label
    -- Go to instruction with given label, if top equals 0
  | GoNonZero Label
    -- Go to instruction with given label, if top not equal to 0

  deriving (Eq, Show, Read)
-- Values are either ints or strings

```haskell
type Value = Either Int String
```

-- Memory for variables

```haskell
data Memory = Memory {
    getNext :: Int, -- Next cell to allocate
    getCell :: [((Name, Int))], -- Memory cells for variables
    getValue :: [((Int, Value))] -- Actual memory
}
```

--Operand stack

```haskell
type Stack = [Value]
```

-- Interpretation of instruction sequences

```haskell
run :: [Instr] -> Maybe Memory
```

-- Interpretation of instructions

```haskell
step :: [Instr] -- The complete instruction sequence
     -> Int -- The instruction pointer (before)
     -> Memory -- The memory (before)
     -> Stack -- The stack (before)
     -> Maybe (Int, -- The instruction pointer (after)
               Memory, -- The memory (after)
               Stack -- The stack (after)
)
Haskell-based components for Pico

- Recognizer
- Parser
- Type checker
- Interpreter
- Pretty printer
- Virtual machine
- Compiler
- Control-flow visualizer

https://github.com/slecourse/slecourse/tree/master/sources/pico/
Compiler

- Use state to keep track of label generator.
- Generate machine instructions compositionally.
- A stream could also be used for linear output.
-- Instruction sequences with labeling

```
type LabeledInstrs = State Int [Instr]
```

-- Generate next jump label

```
nextLabel :: State Int String
```

-- Compile the program to instructions

```
compileProgram :: Program -> [Instr]
```

-- Compile declarations to instructions

```
compileDecls :: [Decl] -> LabeledInstrs
```

-- Compile statements to instructions

```
compileStms :: [Stm] -> LabeledInstrs
```

-- Compile expressions to instructions

```
compileExpr :: Expr -> [Instr]
```
Haskell-based components for Pico

- Recognizer
- Parser
- Type checker
- Interpreter
- Pretty printer
- Virtual machine
- Compiler
- Control-flow visualizer

https://github.com/slecourse/slecourse/tree/master/sources/pico/
Wanted: Flow charts

```
begin declare input : natural,
  output : natural,
  repnr : natural,
  rep : natural;
  input := 14;
  output := 1;
  while input - 1 do
    rep := output;
    repnr := input;
    while repnr - 1 do
      output := output + rep;
      repnr := repnr - 1
    od;
  input := input - 1
od
end
```
Flow charts

- Define an abstract syntax of flow charts.
- No concrete syntax needed.
- Abstract syntax is mapped to “dot” (graphviz).
Flow charts

```haskell
type FlowChart = ([Box], [Arrow])
type Box = (Id, BoxType)
type Id = String -- Identifier for boxes
data BoxType
    = Start
    | End
    | Decision Text
    | Activity Text

type Text = String -- Text to show
type Arrow = ((Id, FromType), Id)
data FromType
    = FromStart
    | FromActivity
    | FromYes
    | FromNo

See Haskell code online.

Illustration of the **dot** sublanguage for flowcharts

digraph FlowChart {
  id1 [label="Start", shape=box, style=bold];
  id2 [label="End", shape=box, style=bold];
  id3 [label=" input := 14 ", shape=box];
  id4 [label=" output := 1 ", shape=box];
  id5 [label=" input - 1 ", shape=diamond];
  id6 [label=" rep := output ", shape=box];
  id7 [label=" repnr := input ", shape=box];
  id8 [label=" repnr - 1 ", shape=diamond];
  id9 [label=" output := output + rep ", shape=box];
  id10 [label=" repnr := repnr - 1 ", shape=box];
  id1 -> id3 [label=" ", headport= n , tailport= s ]
  id3 -> id4 [label=" ", headport= n , tailport= s ]
  id4 -> id5 [label=" ", headport= n , tailport= s ]
  id5 -> id6 [label=" Yes ", headport= n , tailport= sw ]
  id6 -> id7 [label=" ", headport= n , tailport= s ]
  id7 -> id8 [label=" ", headport= n , tailport= s ]
  id8 -> id9 [label=" Yes ", headport= n , tailport= sw ]
  id9 -> id10 [label=" ", headport= n , tailport= s ]
  id10 -> id7 [label=" ", headport= n , tailport= s ]
  id8 -> id4 [label=" No ", headport= n , tailport= se ]
  id5 -> id2 [label=" No ", headport= n , tailport= se ]
}
A Graph File Grammar

The following is an abstract grammar for the DOT language. Terminals are shown in bold font and nonterminals in italics. Literal characters are given in single quotes. Parentheses ) and [ indicate grouping when needed. Square brackets [ and ] enclose optional items. Vertical bars | separate alternatives.

### Grammar

- **graph** → ([**strict**] (digraph | graph) id ’{’ stmt-list ’}’)
- **stmt-list** → [stmt [’;’] [stmt-list ] ]
- **stmt** → attr-stmt | node-stmt | edge-stmt | subgraph | id ’=’ id
- **attr-stmt** → (graph | node | edge) attr-list
- **attr-list** → ’[’ [a-list ] ’]’ [attr-list]
- **a-list** → id ’=’ id [’,’] [a-list]
- **node-stmt** → node-id [attr-list]
- **node-id** → id [port]
- **port** → port-location [port-angle] | port-angle [port-location]
- **port-location** → ’:’ id | ’?’ (’ id ’, id ’)
- **port-angle** → ’@’ id
- **edge-stmt** → (node-id | subgraph) edgeRHS [attr-list]
- **edgeRHS** → edgeop (node-id | subgraph) [edgeRHS]
- **subgraph** → [subgraph id] ’{’ stmt-list ’}’ | subgraph id

---

An id is any alphanumeric string not beginning with a digit+ but possibly including underscores; or a number; or any quoted string possibly containing escaped quotes.

An edgeop is -> in directed graphs and -- in undirected graphs.

The language supports C@@=style comments: // **/ and //.

Semicolons aid readability but are not required except in the rare case that a named subgraph with no body immediate precedes an anonymous subgraph because under precedence rules this sequence is parsed as a subgraph with a heading and a body.

Complex attribute values may contain characters such as commas and white space which are used in parsing the DOT language. To avoid getting a parsing error such values need to be enclosed in double quotes.
Visualizer

- Very much like the compiler.
- Generate low-level graph representation.
- Also very much like the pretty printer.
- Pretty print expressions and statement in boxes.

See Haskell code online.
Concluding remarks
Issues with Haskell study

- Source-code file locations missing in error messages
- No/primitive handling of operator priorities in parsing
- Potential interest in further program analyses
- Lack of IDE integration
Planning ahead

We need to look at proper metaprogramming systems.

Let’s also look at domain-specific programming languages.

We need deeper understanding of some topics:

- Intermediate representations
- Program analysis (e.g., for optimization)
- Code generation
- ...