A few notes on grammars, language specifications, and interpreters

Keunwoo Lee
CSE 341 : Programming Languages
University of Washington
Winter 2004

Outline

- **Language syntax**
  - BNF grammars
  - Regular expressions
- **Operational semantics revisited**
- **Case study: MicroC**
  - **Syntax**
  - **Semantics**
  - **Interpreter**

So you want to design a language...

What do you do?

- Specify syntax
- Specify semantics
- Write a prototype interpreter

Goal of this lecture-and-a-half: whirlwind overview of each step, and conceptual tools

Syntax: terminology

- a **language** is a subset of the finite-length strings over some **alphabet** of symbols
- a **grammar** specifies which strings belong in teh alphabet
- a **parser** turns a string in a language into a more structured representation (usually a **syntax tree**) relevant to that language

Backus-Naur form

- named for **John Backus & Peter Naur**
- Used to specify **context-free grammars** (correspond to **context-free languages**)
- CFGs have four parts:
  1. **terminals**: "atomic" textual entities, e.g. identifiers, keywords, literals
  2. **nonterminals**: "structured" textual entities, e.g. if/then/else or val declarations
  3. **productions**: rules for building nonterminals
  4. **a unique "start" production**

Productions

- Nonterminal on left, sequence of terminals and nonterminals on right, separated by ::=

ifExpr ::= **if** *expr* **then** *expr* **else** *expr*

- May have cases separated by bars:

expr ::= *binOpExpr* | *unaryOpExpr* | *ifExpr* | ...

Example

constLiteral ::= **BOOL_LITERAL** | **INT_LITERAL**

expr ::= *constLiteral*
| **IDENTIFIER**
| *expr* + *expr*
| **not** *expr*
| *expr* = *expr*
| **let** *decl* **in** *expr* **end**
| **if** *expr* **then** *expr* **else** *expr*
| **(** *expr* **)**

decl ::= **val** **IDENTIFIER** = *expr*

BNF "alphabet"?

- BNF grammar alphabet = terminals = *tokens*, not individual characters
- Can specify format of tokens using BNF too, but cumbersome
- Usually use *regular expressions* instead

Regular expressions

- **regexp = an expression that may "match" a string; recursively defined as:**
  - base case: **a character**; `a` matches "a"
  - inductive cases:
    - **concatenation**: `aa` matches "aa"
    - **repetition (Kleene star)**: `a*` matches "a", "aaaa", "aaaaaaaa", ""
    - **union (alternation)**: `a|b` matches "a" or "b
- Inductive cases may use parens to enforce order of operators

Regexp examples

`(a*b)|c`

matches: "b", "c", "aaaab"

`a((b*c*)|d*)`

matches: "a", "abbbbc", "ac", "abbbcccccc", "ad"

`a|b|c|d`

matches: "a", "b", "c", "d"

RE syntactic sugars

- **[ABCDEFG] = (A|B|C|D|E|F|G)**
- **[A-Z] = [ABC....XYZ]**
- **A? = "optional A"**
  - e.g., AB?C = (AC|ABC)
- **A+ = "one or more A"**
  - e.g., A+B = AA*B
- **. = "any character"**
  - e.g., .* = "any string, including empty string"
- **[^A] = "Any character but A."**

Regexps for tokens

- Integer literals:

  INTEGER = `[0-9]+`

- Identifiers:

  IDENTIFER = `[A-Za-z][A-Za-z0-9]+`

- Keywords (easy):

  IF = `if`

  THEN = `then`

Extended BNF

- **Sequences:**

  tupleExpr ::= **(** *expr** **)**

- **Delimited sequences:**

  tupleExpr ::= **(** *expr** \ **,** **)**

- **At-least-one sequences:**

  tupleExpr ::= **(** *expr* **,** *expr*+\ **,** **)**

- **Optional sequences:**

  valDecl ::= **val IDENTIFIER** = *expr* [**;**]

Parsing, lexing

- **Lexer: string -> token stream**
- **Parser: token stream -> syntax tree**
- Usually build using **lexer generator** and **parser generator:**
  - lexer gen and parser gen can take syntax specification and automatically generate code for each
  - bison, yacc, javacc, **sablecc, ml-yacc**

Semantics revisited

- **Review: "Language Construct X in a Nutshell"**
  - **Syntax:**
    - **if** *expr1* **then** *expr2* ***else*** *expr3*
  - **Dynamic semantics**
    - eval *expr1* to *v*
    - if *v* is true, eval *expr2* to *v2* and return
    - else eval *expr3* to *v3* and return
  - **Static semantics**
    - constraints: *expr1* boolean, *expr2* and *expr3* agree
    - result type: type of *expr2* and *expr3*

Formal notation

$$\frac{\Gamma \vdash e \Downarrow \texttt{true} \qquad \Gamma \vdash e_1 \Downarrow v}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v}$$

$$\frac{\Gamma \vdash e \Downarrow \texttt{false} \qquad \Gamma \vdash e_2 \Downarrow v}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v}$$

(dynamic semantics only)

Homework 3 eval...

```
| evalExp env (If (e, e1, e2)) =
  case evalExp env e of
     Integer i =>
        raise Fail "Type error: non-boolean used for test."
    | Boolean b =>
if b then evalExp env e1 else evalExp env e2
```

$$\frac{\Gamma \vdash e \Downarrow \texttt{true} \qquad \Gamma \vdash e_1 \Downarrow v}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v}$$

$$\frac{\Gamma \vdash e \Downarrow \texttt{false} \qquad \Gamma \vdash e_2 \Downarrow v}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v}$$

Summary

- **One way to do language design:**
  - **Specify syntax**
    - Use BNF, regexps
  - **Specify semantics**
    - Use operational semantics (in English, or in)
  - **Implement prototype interpreter**
    - You basically know how...
    - **Operational semantics of language closely related to *actual code* for interpreter!**

Break: ML project overview

Case study: MicroC

```
program ::= decl* stmt*
decl ::= typeName IDENTIFIER [= expr] ;
typeName ::= int | float

stmt ::= expr ;
     | if ( expr ) stmt else stmt
     | while ( expr ) stmt
     | { stmt*\; }

expr ::= INT_LITERAL | FLOAT_LITERAL
     | IDENTIFIER
     | IDENTIFIER = expr
     | expr + expr
     | expr == expr
```

A MicroC program

```
int x = -10;
int y = 0;
while (x = x + 1) {
    y = y + x;
}
```

MicroC semantics

- **We'll just do dynamic semantics of assignment exprs:**

  IDENTIFIER = *expr*

  If an expression *e* evaluates to a value *v* in an environment *Env*, then

  the expression *varName* = *e* evaluates to *v* and produces the updated environment *Env'*, where *Env'* is *Env* with (*varName, v)* substituted for the old binding of *varName*

- Useful insight: expressions produce both *value* and *new environment* (due to side effects!)

Interpreter HOWTO

1. Data type for each syntactic form
2. Lexer & parser
3. Data type for values and environments
4. Implement dynamic semantics for each syntactic form
5. (For statically typed languages) implement static semantics of each syntactic form

Syntax datatypes

```
datatype expr = IntExpr of int
          | FloatExpr of real
          | AssignExpr of string * expr
          | VarExpr of string
          | AddExpr of expr * expr
          | EqExpr of expr * expr

datatype stmt = EvalStmt of expr
          | IfStmt of expr * stmt * stmt
          | WhileStmt of expr * stmt
          | BlockStmt of stmt list

datatype typeName = TInt | TFloat
datatype decl = Decl of typeName * string * expr
datatype program = Program of decl list * stmt list
```

```
expr ::= INT_LITERAL
     | FLOAT_LITERAL
     | IDENTIFIER
     | IDENTIFIER = expr
     | expr + expr
     | expr == expr

stmt ::= expr ;
     | if ( expr ) stmt else stmt
     | while ( expr ) stmt
     | { stmt*\; }
```

Values and envs.

```
datatype value = IntVal of int | FloatVal of real

type environment = (string * value) list

(* returns the value bound to name in e *)
fun lookupEnv(e:environment, name:string) = ...

(* returns the environment e with name's binding updated
      to newValue *)
fun updateEnv(e:environment, name:string, newValue:value) =
      ...
```

Evaluator (exprs)

evalExpr has type:

environment * expr -> value * environment

```
fun evalExpr(env, IntExpr i) = (IntVal i, env)
  | evalExpr(env, VarExpr s) = lookupEnv(env, s)
  | evalExpr(env, AssignExpr(id, exp)) =
    let
       val (v, newEnv) = evalExpr(env, exp)
    in
       (v, updateEnv(newEnv, id, v))
    end
  | ...
```

Evaluator (stmts)

- evalStmt has type:

  environment * stmt -> environment

```
fun evalStmt(env, EvalStmt e) =
    let val (_, newEnv) = evalExpr(env, e) in newEnv end
  | evalStmt(env, IfStmt(e, s1, s2)) =
    let
       val (v, newEnv) = evalExpr(env, e)
       val vIsZero = case v of IntVal i => i = 0
                                         | FloatVal f => f <= 0.0 andalso f >= 0.0
    in
       if vIsZero then evalStmt(newEnv, s1)
       else evalStmt(newEnv, s2)
    end
  | ...
```

Evaluator (decls)

- evalDecl has type:

environment * decl -> environment

```
fun evalDecl(env, Decl(t, name, exp)) =
  let
    val (v, newEnv) = evalExpr(env, exp)
  in
    updateEnv(newEnv, name, v)
  end
```

Evaluator (program)

```
fun eval(Program(decls, stmts)) =
   let
      fun evalDeclList(env, nil) = env
        | evalDeclList(env, d::ds) = evalDeclList(evalDecl(env, d), ds)

      fun evalStmtList(env, nil) = env
        | evalStmtList(env, s::ss) = evalStmtList(evalStmt(env, s), ss)

      val declsEnv = evalDeclList(nil, decls)
   in
      evalStmtList(declsEnv, stmts)
   end
```