Predictive Parsing, Part 1

How to Construct Recursive-Descent Parsers

Monday, November 17, 2009
Reading: Appel 3.2

CS235 Languages and Automata

Department of Computer Science
Wellesley College

Overview

For some grammars, reading the first token (or first few tokens) of input is sufficient for determining which production to apply.

For such grammars, we can write a so-called **predictive parser**. One way to implement a predictive parser is to write a **recursive descent parser**.

In this lecture, we'll study a grammar amenable to predictive parsing and show how to construct a recursive descent parser for it by hand in SML. Along the way we'll learn how to use sum-of-product datatypes to represent tokens and parse trees.

In the next lecture, we will see how to determine if a grammar is amenable to predictive parsing. We will also learn some techniques for handling grammars for which predictive parsers do not exist.

SML Digression: Sum-of-Product Data Types

```
(* contents of the file figure.sml *)
datatype figure =
    Square of int (* <constructor function> of <components> *)
  | Rectangle of int * int
  | Triangle of int * int * int

fun perimeter (Square side) = 4*side
  | perimeter (Rectangle(w,h)) = 2*(w+h)
  | perimeter (Triangle(s1,s2,s3)) = s1+s2+s3

fun scale c (Square side) = Square(c*side)
  | scale c (Rectangle(w,h)) = Rectangle(c*w,c*h)
  | scale c (Triangle(s1,s2,s3)) = Triangle(c*s1,c*s2,c*s3)
```

```
- use "figure.sml";
[opening figure.sml]
datatype figure
  = Rectangle of int * int | Square of int | Triangle of int * int * int
val perimeter = fn : figure -> int
val scale = fn : int -> figure -> figure
val it = () : unit

- map perimeter [Square 1, Rectangle(2,3), Triangle(4,5,6)];
val it = [4,10,15] : int list

- map (scale 10) [Square 1, Rectangle(2,3), Triangle(4,5,6)];
val it = [Square 10,Rectangle (20,30),Triangle (40,50,60)] : figure list
```


We Can Define our Own List Data Type

```
(* contents of the file mylist.sml *)
datatype 'a mylist = Nil | Cons of 'a * ('a mylist)

fun sum Nil = 0
  | sum (Cons(n,ns)) = n + (sum ns)

fun map f Nil = Nil
  | map f (Cons(x,xs)) = Cons(f x, map f xs)
```

```
- use "mylist.sml";
[opening mylist.sml]
datatype 'a mylist = Cons of 'a * 'a mylist | Nil
val sum = fn : int mylist -> int
val map = fn : ('a -> 'b) -> 'a mylist -> 'b mylist
val it = () : unit

- sum (Cons(1, Cons(2, Cons(3, Nil))));
val it = 6 : int

- map (fn x => x*2) (Cons(1, Cons(2, Cons(3, Nil))));
val it = Cons (2,Cons (4,Cons (6,Nil))) : int mylist
```

A Token Data Type

```
datatype binop = Add | Mul | Sub | Div

datatype token = EOF
        |  ID of string
        |  INT of int
        |  OP of binop
        |  PRINT
        |  BEGIN   |  END
        |  LPAREN |  RPAREN  | SEMI | GETS
```

token data type definition

```
begin x := (3+4); print ((x-1)*(x+2)); end
```

Sample program

SML token list for sample program

```
[BEGIN, ID "x", GETS, LPAREN, INT 3, OP Add, INT 4, RPAREN, SEMI,
 PRINT, LPAREN, LPAREN, ID "x", OP Sub, INT 1, RPAREN, OP Mul,
 LPAREN, ID "x", OP Add, INT 2, RPAREN, RPAREN, SEMI, END] : token list
```


Some Token Operations

```
fun eof() = EOF

fun isEof(EOF) = true
   | isEof(_) = false

fun binopToString(Add) = "+"
   | binopToString(Sub) = "-"
   | binopToString(Mul) = "*"
   | binopToString(Div) = "/"

 fun toString(EOF) = "[EOF]"
   | toString(ID(s)) = "[" ^ s ^ "]"
   | toString(INT(i)) = "[" ^ (Int.toString(i)) ^ "]"
   | toString(OP(opr)) = "[" ^ (binopToString(opr)) ^ "]"
   | toString(PRINT) = "[PRINT]"
   | toString(BEGIN) = "[BEGIN]"
   | toString(END) = "[END]"
   | toString(LPAREN) = "[(]"
   | toString(RPAREN) = "[)]"
   | toString(SEMI) = "[;]"
   | toString(GETS) = "[:=]"
```

Running Example: SLiP--

As our running example, we will use SLiP--, a subset of Appel's straight-line programming language (SLiP).

The **abstract syntax** of SLiP-- is described by these SML datatypes:

```
datatype pgm = Pgm of stm
and  stm = Assign of string * exp
         |  Print of exp
         |  Seq of stm list
and exp = Id of string
         |  Int of int
         |  BinApp of exp * binop * exp
and binop = Add | Sub | Mul | Div
```

Over the next two lectures, we will explore several versions of **concrete syntax** for SLiP--

Our First Concrete Syntax for SLiP--

Productions for Concrete Grammar

P → S EOF

S → ID(str) := E | print E | begin SL end

SL → % | S ; SL

E → ID(str) | INT(int) | (E B E)

B → + | - | * | /

SML Data Types for Abstract Grammar

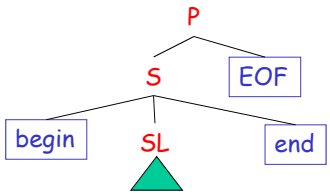
```
datatype pgm = Pgm of stm
and stm = Assign of string * exp
        |  Print of exp
        |  Seq of stm list
and exp = Id of string
        |  Int of int
        |  BinApp of exp * binop * exp
and binop = Add | Sub | Mul | Div
```

An Example SLiP-- Program

Chars: begin x := (3+4); print ((x-1)*(x+2)); end

Tokens: begin | ID("x") | := | (| INT(3) | + | INT(4) |) | ;
print | (| (| ID("x") | - | INT(1) |) | * | (| ID("x") | +
INT(2) |) |) | ; | end | EOF

Parse Tree: (see full tree on next slide)

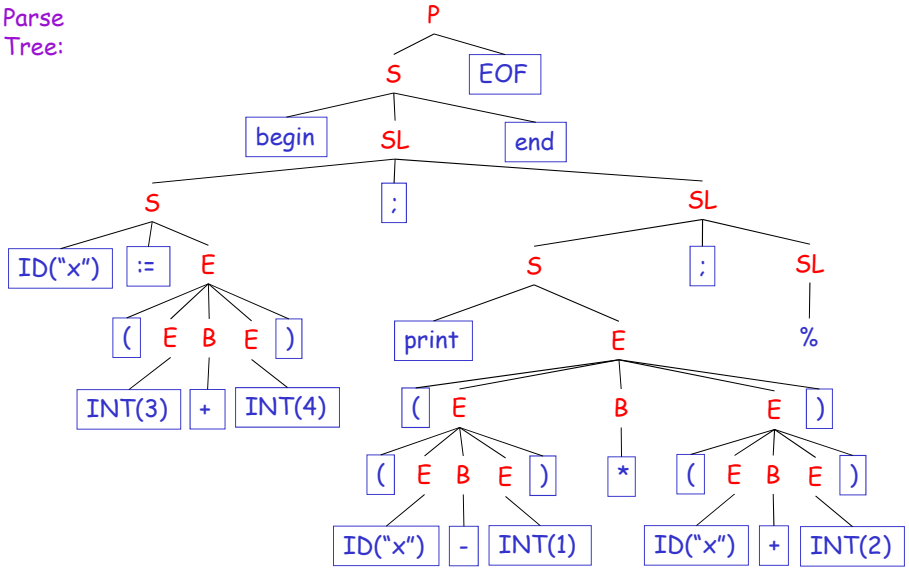


Abstract Syntax Tree (AST):

```
Pgm(Seq [Assign("x", BinApp(Int(3),Add,Int(4))),
         Print(BinApp(BinApp(Id("x"),Sub,Int(1)),
                      Mul,
                      BinApp(Id("x"),Add,Int(2))))])
```


An Example SLiP-- Program

Parse Tree:



Predictive Parsing For SLiP--

P → S EOF

S → ID(str) := E | print E | begin SL end

SL → % | S ; SL

E → ID(str) | INT(int) | (E B E)

B → + | - | * | /

Observe that:

- expressions (E) must begin with ID(str), INT(int), or (.
- statements (S) must begin with ID(str), print, or begin.
- statement lists (SL) must begin with a statement (S) and so must begin with ID(str), print, or begin . They must end with end (a token that is not part of the SL tree but one immediately following it).
- programs (P) must begin with a statement (S) and so must begin with ID(str) , print , or begin.

Predictive Parsing Table for SLiP--

Can summarize observations on previous slide with a **predictive parsing table** of variables x tokens in which at most one production is valid per entry.

Empty slots in the table indicate parsing errors.

	ID(s)	INT(i)	(	OP(b)	print	begin	end
P	P → S EOF				P → S EOF	P → S EOF	
S	S → ID(str) := E				S → print E	S → begin SL end	
SL	SL → S ; SL				SL → S ; SL	SL → S ; SL	SL → %
E	E → ID(str)	E → INT(num)	E → (E B E)				
B				B → OP(b)			

Recursive Descent Parsing

From a predictive parsing table, it is possible to construct a **recursive descent parser** that parses tokens according to productions in the table.

Such a parser can "eat" (consume) or "peek" (look at without consuming) the next token.

For each variable X in the grammar, the parser has a function, eatX, that is responsible for consuming tokens matched by the RHS of a production for X and returning an abstract syntax tree for the consumed tokens. Since the RHS of a production may contain other variables, the eat… functions can call each other recursively.

We will now study the SML code for a recursive descent parser for Slip--.

Slip-- Parser: Scanner Functions

```
val nextToken : unit -> token
(* Remove and return next token from token stream *)

val peekToken: unit -> token
 (* Return next token from token stream without removing it *)

val initScanner : string -> unit
(* Initialize scanner from a string *)
```

Slip-- Parsing Functions

```
(* Collection of mutually recursive functions for recursive descent parsing *)
fun eatPgm () = …  (* : unit -> pgm. Consume all program tokens and return pgm *)
and eatStm () = … (* : unit -> stm. Consume all statement tokens and return stm *)
and eatStm List() = … (* : unit -> stm list. Consume all tokens for a sequence of
                                  statements and return stm list *)
and eatExp () = … (* : unit -> exp. Consume expression tokens and return exp*)
and eatBinop () = … (* : unit -> exp. Consume binop token and return binop *)
and eat token =  (* token -> unit.  Consume next token and succeed without complaint
                        if it's equal to the given token.  Otherwise complain w/error. *)
   let val token' = nextToken()
    in if token = token' then ()
      else raise Fail ("Unexpected token: wanted " ^ (Token.toString token)
                  ^ " but got " ^ (Token.toString token'))
   end

fun stringToExp str = (initScanner(str); eatExp())  (* Parse string into exp *)
fun stringToStm str = (initScanner(str); eatStm()) (* Parse string into stm *)
fun stringToPgm str = (initScanner(str); eatPgm()) (* Parse string into pgm *)
```


Slip-- Parsing: Top-Level Function Examples

```
- stringToExp "((1+2)*(3-4))";
val it = BinApp (BinApp (Int 1,Add,Int 2),Mul,BinApp (Int 3,Sub,Int 4)) : exp

- stringToStm "x := (1+2);";
val it = Assign ("x",BinApp (Int 1,Add,Int 2)) : stm

- stringToPgm "begin x := (3+4); print ((x-1)*(x+2)); end";
val it =
  Pgm
    (Seq
      [Assign ("x",BinApp (Int 3,Add,Int 4)),
       Print
         (BinApp (BinApp (Id "x",Sub,Int 1),Mul,BinApp (Id "x",Add,Int 2)))])
  : pgm
```

Slip-- Parsing: Parsing Expressions

```
and eatExp () =
   let val token = nextToken()
    in case token of
         ID(str) => Id(str)
       | INT(i) => Int(i)
       | LPAREN => let val exp1 = eatExp()
                       val bin = eatBinop()
                       val exp2 = eatExp()
                       val _ = eat RPAREN
                   in BinApp(exp1,bin,exp2)
                  end
       | _ => raise Fail ("Unexpected token begins exp: " ^ (Token.toString token))
   end

 and eatBinop () =
   let val token = nextToken()
    in case token of
         OP(binop) => binop
       | _ => raise Fail ("Expect a binop token but got: " ^ (Token.toString token))
   end
```


Slip-- Parsing: Parsing Statements

```
and eatStm () =
   let val token = nextToken()
    in case token of
        ID(str) => let val _ = eat GETS
                       val rhs = eatExp()
                   in Assign(str,rhs)
                  end
      | PRINT => let val arg = eatExp()
                  in Print(arg)
                 end
      | BEGIN => let val stms = eatStmList()
                     val _ = eat END
                 in Seq(stms)
                 end
      | _ => raise Fail ("Unexpected token begins stm: " ^ (Token.toString token))
   end
```

Slip-- Parsing: Parsing Statement Lists

```
and eatStmList () =
   let val token = peekToken() (* Must peek rather than eat *)
    in case token of
           END => []
         | _ => let val stm = eatStm()
                    val _ = eat SEMI
                    val stms = eatStmList()
                in stm::stms
               end
   end
```


Slip-- Parsing: Parsing Programs

```
fun eatPgm () =
   let val body = eatStm()
       val _ = eat EOF
    in Pgm(body)
   end
```