Predictive Parsing

Monday, November 19, 2007
and Monday November 26, 2007
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**.

Today, we will see how to determine if a grammar is amendable to predictive parsing and how to build a predictive parser by hand.

We will also learn some techniques for handling grammars for which predictive parsers do not exist.

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
```

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

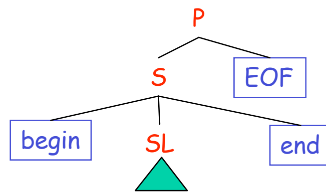
```
datatype pgm = Pgm of Stm
and stm = Assign of string * exp
        |   Print of exp
        |   Seq of stm list
and exp = Id of string
         |  Int of int
         |  BinApp of exp * binop * exp
and binop = Add | Sub | Mul | Div
```

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)

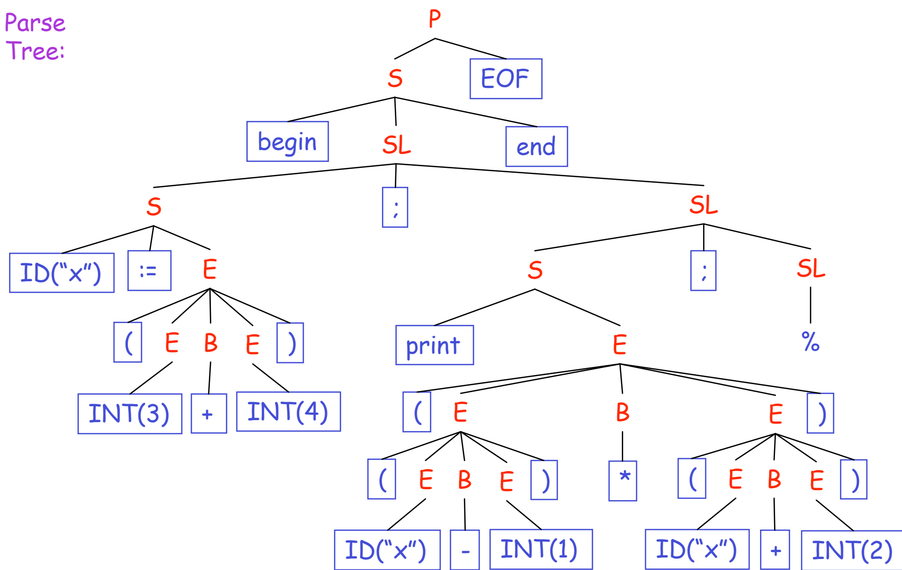


Abstract Syntax Tree (AST):

```
Pgm(Seq [Assign(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), prin, 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" at the next token.

See ParserParens.sml for a recursive descent parser for SLiP-- written in SML.

NULLABLE, FIRST, and FOLLOW

Parsing tables are constructed using the following notions:

Let t range over terminals, V and W range over variables, α range over terminals $\cup$ variables, and γ range over sequences of terminals $\cup$ variables.

- **NULLABLE(γ)** is true iff γ can derive the empty string (%)
- **FIRST(γ)** is the set of terminals that can begin strings derived from γ.
- **FOLLOW(V)** is the set of terminals that can immediately follow V in some derivation.

Computing NULLABLE For Variables

A variable V is NULLABLE iff

1. There is a production $V \to \%$

OR

2. There is a production $V \to V_1 \ldots V_n$
   and each of $V_1, \ldots, V_n$ is NULLABLE

(Case 1 is really a special case of 2 with n = 0.)

In general, it is necessary to compute an **iterative fixed point** to determine nullability of a variable.

Example (from Appel 3.2)

$X \to a \mid Y$

$Y \to \% \mid c$

$Z \to d \mid X\ Y\ Z$

Computing FIRST

FIRST(t) = t

FIRST(V) = $\cup$ {FIRST(γ) | $V \to \gamma$ is a production for V}

FIRST($\alpha_0 \ldots \alpha_i \ldots \alpha_n$) = $\cup$ {FIRST(α_i) | $\alpha_0, \ldots, \alpha_{i-1}$ are all nullable}

Again, this is determined by an **iterative fixed point** computation

$X \to a \mid Y$

$Y \to \% \mid c$

$Z \to d \mid X\ Y\ Z$

Computing FOLLOW

FOLLOW(V) =
$\cup$\{FIRST(α_j) | W → $\alpha_0 \ldots \alpha_{i-1}$ V $\alpha_{i+1} \ldots \alpha_j \ldots \alpha_n$
is a production in the grammar
and $\alpha_{i+1}, \ldots, \alpha_{j-1}$ are all nullable\}
$\cup$ $\cup$\{FOLLOW(W) | W → $\alpha_0 \ldots \alpha_{i-1}$ V$\alpha_{i+1} \ldots \alpha_n$
is a production in the grammar
and $\alpha_{i+1}, \ldots, \alpha_n$ are all nullable\}

Again, this is determined by an **iterative fixed point** computation

X → a | Y
Y → % | c
Z → d | X Y Z

Constructing Predictive Parsing Tables

To construct a predictive parsing table,
do the following for each production V → γ:

- For each t in FIRST(γ), enter V → γ in row V, column t.
- If NULLABLE(γ), for each t in FOLLOW(V), enter V → γ in row V, column t

P → S EOF
S → ID(str) := E | print E | begin SL end
SL → % | S ; SL
E → ID(str) | INT(int) | (E B E)
B → + | - | * | /

More Practice

S' → S EOF
S → T | 0S1
T → % | 10T

	NULLABLE	FIRST	FOLLOW
S'			
S			
T			

	0	1	EOF
S'			
S			
T			

Parsing not predictive since some table slots now have multiple entries!

Adding Extra Lookahead

S' → S EOF
S → T | 0S1
T → % | 10T

Sometimes predictivity can be re-established by adding extra **lookahead**:

	0	10	11	1 EOF	EOF
S'					
S					
T					

LL(k) Grammars

An LL(k) grammar is one that has a predictive parsing table with k symbols of lookahead.

- The SLiP-- grammar is LL(1).
- The S'/S/T grammar is LL(2) but not LL(1).

In LL,

- the first L means the tokens are consumed left-to-right.
- the second L means that the parse tree is constructed in the manner of a leftmost derivation.

SLiP-- Expressions with Prefix Syntax

Suppose we change Slip-- expressions to use prefix syntax:

E → ID(str) | INT(int) | B E E

Parsing is still predictive:

	ID(s)	INT(i)	OP(b)	print	begin	end
E	E → ID(str)	E → INT(num)	E → B E E			
B			B → OP(b)			

Postfix Syntax for Expressions

Suppose we change Slip-- expressions to use postfix syntax:

E → ID(str) | INT(int) | E E B

Parsing is no longer predictive since some table slots now have multiple entries:

	ID(s)	INT(i)	OP(b)	print	begin	end
E	E → ID(str) E → E E B	E → INT(num) E → E E B				
B			B → OP(b)			

Postfix expressions are fundamentally not predictive (not LL(k) for any k) there's nothing we can do to parse them predictively.

But we'll soon see we *can* parse them with a shift/reduce parser.

Infix Syntax for Expressions

Suppose we change Slip-- expressions to use infix syntax without required parens (but with optional ones)

E → ID(str) | INT(int) | E B E | (E)

Parsing is no longer predictive:

	ID(s)	INT(i)	OP(b)	(	print	begin	end
E	E → ID(str) E → E B E	E → INT(num) E → E B E		E → (E)			
B			B → OP(b)				

This is not surprising: this grammar is ambiguous, and *no* ambiguous grammar can be uniquely parsed with *any* deterministic parsing algorithm.

Removing Ambiguity May not Help

Supppose we use an unambiguous infix grammar for arithmetic:

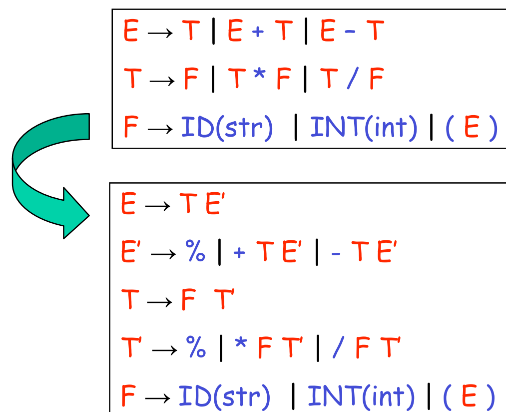
E → T | E + T | E - T	*Expressions*
T → F | T * F | T / F	*Terms*
F → ID(str) | INT(int) | (E)	*Factors*

Parsing is *still* not predictive due to **left recursion** in E and T:

	ID(s)	INT(i)	OP(b)	(	print	begin	end
E	E → T E → E + T E → E - T	E → T E → E + T E → E - T		E → T E → E + T E → E - T			
T	T → F T → T * F T → T / F	T → F T → T * F T → T / F		T → F T → T * F T → T / F			
F	F → ID(str)	F → INT(num)		F → (E)			

Left Recursion Removal

Sometimes we can transform a grammar to remove left recursion (parse trees are transformed correspondingly).



See Appel 3.2 for a general description of this transformation.
You will see an example of this in PS5.

The Transformed Grammar is Predictive!

E → T E'
E' → % | + T E' | - T E'
T → F T'
T' → % | * F T' | / F T'
F → ID(str) | INT(int) | (E)

	ID(s)	INT(i)	+	*	(	)	;	EOF
E	E → T E'	E → T E'			E → T E'			
E'			E' → + T E'			E' → %	E' → %	E' → %
T	T → F T'	T → F T'			T → F T'			
T'			T' → %	T' → * F T'		T' → %	T' → %	T' → %
F	F → ID(str)	F → INT(num)			F → (E)			

Transforming Parse Trees

The parse tree from the transformed grammar can be transformed back to the untransformed grammar.
E.g. 2 * 3 + 4