Predictive Parsing

How to Construct Recursive-Descent Parsers

Wednesday, November 30, and Friday, December 2, 2011

CS235 Languages and Automata

Department of Computer Science
Wellesley College

Goals of This Lecture

- Introduce **predictive parsers**, efficient parsers for certain grammars in reading the first token (or first few tokens) of input is sufficient for determining which production to apply.
- Show how predictive parsers can be implemented by a **recursive descent parser** in your favorite programming language.
- Show how to construct a **predictive parsing table**, which determines whether a grammar is amenable to predictive parsing.
- Study some techniques for **transforming nonpredictive grammars** into predictive ones, including **removing ambiguity** and **left-recursion removal**.
- Learn how to use SML's **sum-of-product datatypes** to represent tokens and parse trees.
- Learn the distinction between **concrete** and **abstract** syntax.

Main Example: Intexp

As our main example, we'll use a simple integer expression language that we'll call Intexp.

The **abstract syntax**, or logical structure, of Intexp is described by these SML datatypes:

```
datatype pgm = Pgm of exp  (* a program is an expression *)

and exp = Int of int     (* an expression is either an integer *)
        | BinApp of exp * binop * exp  (* or a binary operator application *)

and binop = Add | Sub | Mul | Div  (* there are four binary operators *)
```

We'll explore several versions of Intexp's **concrete syntax** , i.e., how programs, expressions, and binary operators are written down.

We'll also consider several extensions to Intexp.

A Token Data Type for Intexp

token data type definition

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

datatype token = EOF (* special "end of input" marker *)
        |  INT of int  (* INT (3) is a token, while Int(3) is an expression *)
        |  OP of binop
        |  LPAREN |  RPAREN
```

Sample "program"

```
((3+4) * (42-17))
```

SML token list for sample program

```
- Scanner.stringToTokens "((3+4)*(42-17))";
val it =  [LPAREN, LPAREN, INT 3, OP Add, INT 4, RPAREN, OP Mul, LPAREN,
           INT 42, OP Sub, INT 17, RPAREN, RPAREN] : Token.token list
```

(* Note: EOF does *not* appear explicitly in the token list, but is implicitly at the end *)

Our First Concrete Syntax for Intexp: Explicitly Parenthesized Operations

Productions for Concrete Grammar

P → E EOF
E → INT(int) | (E B E)
B → + | - | * | /

SML Data Types for Abstract Grammar

```
datatype pgm = Pgm of exp
and exp = Int of int
        | BinApp of exp * binop * exp
and binop = Add | Sub | Mul | Div
```


An Example Intexp Program

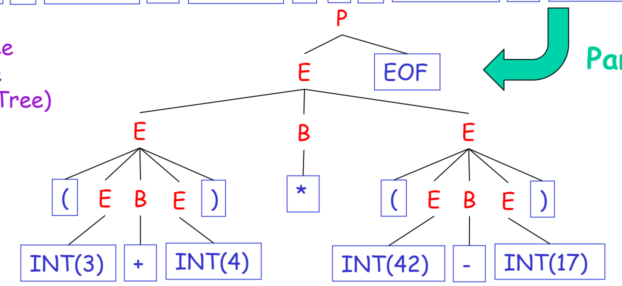
Chars: ((3+4) * (42 – 17))

Lexer (scanner)

Tokens: ((INT(3) + INT(4)) * (INT(42) - INT(17)))

Parser

Parse Tree (Concrete Syntax Tree)



Abstract Syntax Tree (AST):

```
- ParserParens.stringToPgm "((3+4)*(42-17))";
val it = Pgm (BinApp (BinApp (Int 3,Add,Int 4),
    Mul,BinApp (Int 42,Sub,Int 17))): AST.pgm
```


Predictive Parsing For Intexp

P → E EOF
E → INT(int) | (E B E)
B → + | - | * | /

Observe that:

•expressions E must begin with INT(int) or (.

•programs P must begin with an expression (E) and so must begin with INT(int) or (.

•binops B must begin with +, -, *, or /.

Predictive Parsing Table for Intexp

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.

	INT(i)	(	OP(b)	)	EOF
P	P → E EOF	P → E EOF			
E	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 Intexp.

Intexp Parser: Scanner Functions

```
(* We assume the existence of the following token functions,
   whose implementation details we will *not* study. *)

val initScanner : string -> unit
(* Initialize scanner from a string, creating implicit token stream *)

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

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


Intexp Parsing Functions

```
(* Collection of mutually recursive functions for recursive descent parsing *)
fun eatPgm () = ...  (* : unit -> pgm. Consume all program tokens and return pgm *)
and eatExp () = ... (* : unit -> exp. Consume expression tokens and return exp*)
and eatBinop () = ... (* : unit -> binop. 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 stringToPgm str = (initScanner(str); eatPgm()) (* Parse string into pgm *)
```


Intexp: Parsing Programs and Binops

```
fun eatPgm () =
   (* : unit -> pgm. Consume all program tokens and return pgm *)
   let val body = eatExp()
       val _ = eat EOF
   in Pgm(body)
   end

and eatExp () =  (* see next slide *)

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

Intexp: Parsing Expressions

```
and eatExp () =
   (* : unit -> exp. Consume expression tokens and return exp*)
   let val token = nextToken()
   in case token of
      INT(i) =>

     | LPAREN =>







     | _ => raise Fail ("Unexpected token begins exp: "
                        ^ (Token.toString token))
   end
```


An Extended Language: SLiP--

SLiP-- is a subset of Appel's straight-line programming language (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

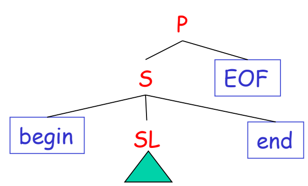
```
datatype pgm = Pgm of stm
and stm = Assign of string * exp
        |   Print of exp
        |   Seq of stm list
and exp = Id of string
        |   Int of int
        |   BinApp of exp * binop * exp
and binop = Add | Sub | Mul | Div
```


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)

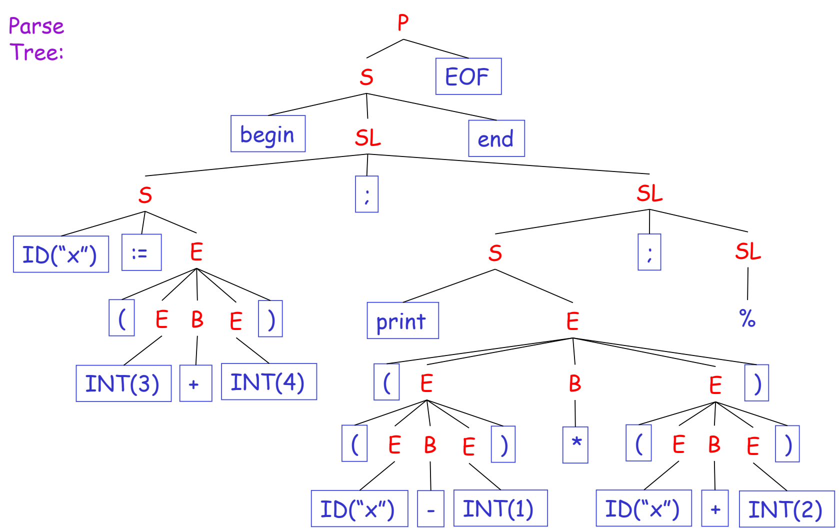


Abstract Syntax Tree (AST):

```
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)			

NULLABLE, FIRST, and FOLLOW

Predictive parsing tables like that for Slip-- are constructed using the following notions:

Let t range over terminals, V and W range over variables, α range over terminals ∪ variables, and γ range over sequences of terminals ∪ 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 → %

OR

2. There is a production V → $V_1 \ldots V_n$ 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. We've seen this already in the algorithm for converting a CFG to Chomsky Normal Form.

Example (from Appel 3.2)

X → a | Y
Y → % | c
Z → d | X Y Z

Another example:

S' → S EOF
S → T | 0S1
T → % | 10T

Computing FIRST

$FIRST_0(V) = \{\}$ for every variable V
For all i > 0:
•$FIRST_i(t) = \{t\}$
•$FIRST_i(V) = \cup \{FIRST_{i-1}(\gamma) \mid V \to \gamma \text{ is a production for } V\}$
•$FIRST(\alpha_1 \ldots \alpha_j \ldots \alpha_n) = \cup_{1 \le j \le n} \{FIRST_{i-1}(\alpha_j) \mid \alpha_1, \ldots, \alpha_{j-1} \text{ are all nullable}\}$

Again, this is determined by an **iterative fixed point** computation.
For the following grammars (1) write the $FIRST_i$ equations and (2) for each var V use them to find the smallest k s.t. $FIRST_k(V) = FIRST_{k-1}(V)$.

X → a | Y
Y → % | c
Z → d | X Y Z

S' → S EOF
S → T | 0S1
T → % | 10T

Computing FOLLOW

$FOLLOW_0(V) = \{\}$ for every variable V
For all i > 0:
$FOLLOW_i(V) =$
$\cup \{FIRST(\alpha_j) \mid W \to \gamma V \alpha_1 \ldots \alpha_j \ldots \alpha_n$ is a production in the grammar and $\alpha_1, \ldots, \alpha_{j-1}$ are all nullable variables$\}$
$\cup$ $\cup \{FOLLOW_{i-1}(W) \mid W \to \gamma V \alpha_1 \ldots \alpha_n$ is a production in the grammar and $\alpha_1, \ldots, \alpha_n$ are all nullable variables$\}$

Again, this is determined by an **iterative fixed point** computation:
For the following grammars (1) write the $FIRST_i$ equations and (2) for each var V use them to find the smallest k s.t. $FOLLOW_k(V) = FOLLOW_{k-1}(V)$.

X → a | Y
Y → % | c
Z → d | X Y Z

S' → S EOF
S → T | 0S1
T → % | 10T

Example: Slip--

Calculate NULLABLE, FIRST, and FOLLOW for the variables in the Slip-- 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 → + | - | * | /

Constructing Predictive Parsing Tables

A predictive parsing table has rows labeled by variables and columns labeled by terminals.

To construct a predictive parsing table for a given grammar, do the following for each production $V \to \gamma$:

- For each t in $FIRST(\gamma)$, enter $V \to \gamma$ in row V, column t.
- If $NULLABLE(\gamma)$, for each t in FOLLOW(V), enter $V \to \gamma$ in row V, column t

	ID(s)	INT(i)	(	OP(b)	print	begin	end
P							
S							
SL							
E							
B							

Slip--: Recursive Descent Parser

```
(* 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: Programs and Binops

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

```
and eatStm () =
    let val token = nextToken()
     in case token of
          ID(str) =>



        | PRINT =>


        | BEGIN =>



        | _ => raise Fail ("Unexpected token begins stm: "
                           ^ (Token.toString token))
    end
```

Slip-- Parsing: Statement Lists

```
and eatStmList () =
    let val token = peekToken()   (* Must peek rather than eat
                                     (the essence of FOLLOW!) *)
     in case token of
          END =>

        | _ =>




    end
```


Slip-- Parsing: Expressions

```
and eatExp () =
    let val token = nextToken()
      in case token of
           ID(str) =>

         | INT(i) =>

         | LPAREN =>




         | _ => raise Fail ("Unexpected token begins exp: "
                            ^ (Token.toString token))
    end
```


More Practice with NULLABLE, FIRST, & FOLLOW

$S' \rightarrow S$ EOF
$S \rightarrow T \mid 0S1$
$T \rightarrow \% \mid 10T$

	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

$S' \rightarrow S$ EOF
$S \rightarrow T \mid 0S1$
$T \rightarrow \% \mid 10T$

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.

Expressions with Prefix Syntax

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

E → ID(str) | INT(int) | B E E

E.g. , * - x 1 + y 2

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 Intexp/Slip-- expressions to use postfix syntax:

E → ID(str) | INT(int) | E E B

E.g. , x 1 – y 2 + *

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), so there's nothing we can do to parse them predictively.

It turns out that 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)

E.g. x - 1 * y + 2

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.

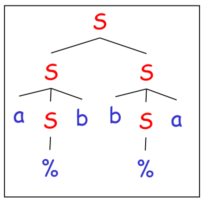
Digression: Ambiguity (Lec #24 Review)

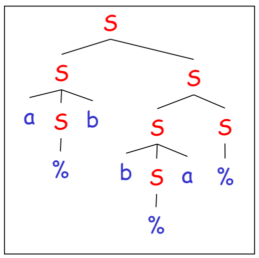
A CFG is **ambiguous** if there is more than one parse tree for a string that it generates.

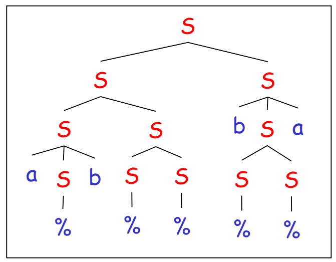
S → %
S → SS
S → aSb
S → bSa

This is an example of an ambiguous grammar.
The string abba has an infinite number of parse trees!

Here are a few of them:







Ambiguity Can Affect Meaning

Ambiguity can affect the meaning of a phrase in both natural languages and programming languages.

Here's are some natural language examples:

High school principal

Fruit flies like bananas.

A woman without her man is nothing.

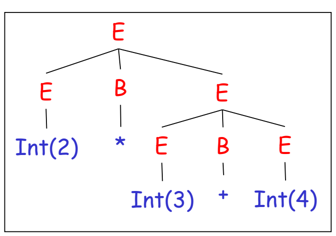
A classic example in programming languages is arithmetic expressions:

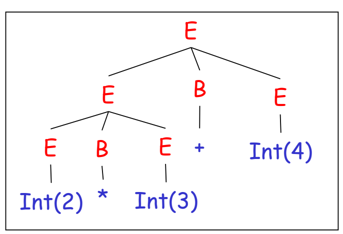
E → ID(str) | INT(int) | E B E | (E)
B → + | - | * | /

Arithmetic Expressions: Precedence

E → ID(str) | INT(int) | E B E | (E)
B → + | - | * | /

What does 2 * 3 + 4 mean?

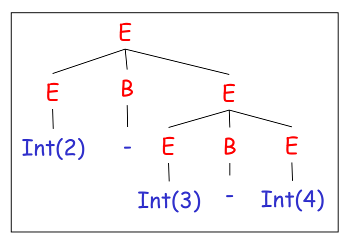


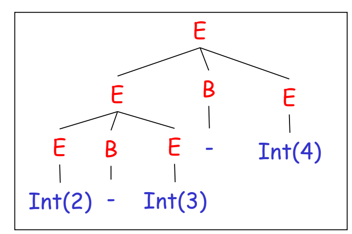


Arithmetic Expressions: Associativity

E → ID(str) | INT(int) | E B E | (E)
B → + | - | * | /

What does 2 - 3 - 4 mean?





Precedence Levels

We can transform the grammar to express precedence levels:

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

Now there is only one parse tree for 2 * 3 + 4. Why? What is it?

Specifying Left Associativity

We can further transform the grammar to express left associativity.

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

Now there is only one parse tree for 2 - 3 - 4. Why? What is it?

How would we specify right associativity?

Another Classic Example: Dangling **Else**

Stm → if Exp then Stm else Stm
Stm → if Exp then Stm
Stm → *… other productions for statements …*

There are two parse trees for the following statement.
What are they?

if Exp_1 then if Exp_2 then Stm_1 else Stm_2

Fixing the Dangling **Else**

Stm → MaybeElseAfter | NoElseAfter
MaybeElseAfter → if Exp then MaybeElseAfter else MaybeElseAfter
MaybeElseAfter → *… other productions for statements…*
NoElseAfter → if Exp then Stm
NoElseAfter → if Exp then MaybeElseAfter else NoElseAfter

Now there is only one parse tree for the following statement.
What is it?

if Exp_1 then if Exp_2 then Stm_1 else Stm_2

Back to Predictive Parsing: Removing Ambiguity May not Help

Suppose 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).

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

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

See Appel 3.2 for a general description of this transformation.
You will use this transformation in PS10.

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. (It's hard to parse linear sequence of tokens into trees, but it's easy to transform trees!)
E.g. 2 * 3 + 4