Building A Recursive Descent Parser

We start with a procedure **`Match`**, that matches the current input token against a predicted token:

```
void Match(Terminal a) {
  if (a == currentToken)
     currentToken = Scanner();
  else SyntaxErrror();}
```

To build a parsing procedure for a non-terminal A, we look at all productions with A on the lefthand side:

$A \rightarrow X_1 \ldots X_n \mid A \rightarrow Y_1 \ldots Y_m \mid \ldots$

We use predict sets to decide which production to match (LL(1) grammars always have disjoint predict sets).

We match a production's righthand side by calling **`Match`** to match terminals, and calling parsing procedures to match non-terminals.

The general form of a parsing procedure for

$A \to X_1 \ldots X_n \mid A \to Y_1 \ldots Y_m \mid \ldots$

is

```
void A() {
 if (currentToken in Predict(A→X1...Xn))
   for(i=1;i<=n;i++)
     if (X[i] is a terminal)
         Match(X[i]);
     else X[i]();
 else
  if (currentToken in Predict(A→Y1...Ym))
   for(i=1;i<=m;i++)
     if (Y[i] is a terminal)
         Match(Y[i]);
     else Y[i]();
 else
      // Handle other A →... productions
 else // No production predicted
     SyntaxError();
}
```

Usually this general form isn't used.

Instead, each production is "macro-expanded" into a sequence of **`Match`** and parsing procedure calls.

Example: CSX-Lite

Production	Predict Set
Prog $\rightarrow$ { Stmts } Eof	{
Stmts $\rightarrow$ Stmt Stmts	id if
Stmts $\rightarrow$ λ	}
Stmt $\rightarrow$ id = Expr ;	id
Stmt $\rightarrow$ if (Expr) Stmt	if
Expr $\rightarrow$ id Etail	id
Etail $\rightarrow$ + Expr	+
Etail $\rightarrow$ - Expr	-
Etail $\rightarrow$ λ	) ;

CSX-Lite Parsing Procedures

```
void Prog() {
  Match("{");
  Stmts();
  Match("}");
  Match(Eof);
}

void Stmts() {
 if (currentToken == id ||
     currentToken == if){
     Stmt();
     Stmts();
 } else {
     /* null */
}}
```

```
void Stmt() {
 if (currentToken == id){
     Match(id);
     Match("=");
     Expr();
     Match(";");
 } else {
     Match(if);
     Match("(");
     Expr();
     Match(")");
     Stmt();
}}

void Expr() {
  Match(id);
  Etail();
}


void Etail() {
 if (currentToken == "+") {
     Match("+");
     Expr();
 } else if (currentToken == "-"){
     Match("-");
     Expr();
 } else {
     /* null */
}}
```

Let’s use recursive descent to parse
{ a = b + c; } Eof
We start by calling **`Prog()`** since this represents the start symbol.

Calls Pending	**Remaining Input**
`Prog()`	{ a = b + c; } Eof

`Match("{");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	{ a = b + c; } Eof

`Stmts();`
`Match("}");`
`Match(Eof);`
	a = b + c; } Eof

`Stmt();`
`Stmts();`
`Match("}");`
`Match(Eof);`
	a = b + c; } Eof

Calls Pending	Remaining Input

`Match(id);`
`Match("=");`
`Expr();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	a = b + c; } Eof

`Match("=");`
`Expr();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	= b + c; } Eof

`Expr();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	b + c; } Eof

`Match(id);`
`Etail();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	b + c; } Eof

Calls Pending	Remaining Input

`Etail();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	+ c; } Eof

`Match("+");`
`Expr();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	+ c; } Eof

`Expr();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	c; } Eof

`Match(id);`
`Etail();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	c; } Eof

Calls Pending	Remaining Input

`Etail();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	; } Eof

`/* null */`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	; } Eof

`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	; } Eof

`Stmts();`
`Match("}");`
`Match(Eof);`
	} Eof

`/* null */`
`Match("}");`
`Match(Eof);`
	} Eof

`Match("}");`
`Match(Eof);`
	} Eof

Calls Pending	**Remaining Input**
`Match(Eof);`	Eof
Done!	**All input matched**

Syntax Errors in Recursive Descent Parsing

In recursive descent parsing, syntax errors are automatically detected. In fact, they are detected *as soon as possible* (as soon as the first illegal token is seen).

How? When an illegal token is seen by the parser, either it fails to predict any valid production or it fails to match an expected token in a call to **`Match`**.

Let’s see how the following illegal CSX-lite program is parsed:

{ b + c = a; } Eof

(Where should the first syntax error be detected?)

Calls Pending	Remaining Input
`Prog()`	{ b + c = a; } Eof

`Match("{");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	{ b + c = a; } Eof

`Stmts();`
`Match("}");`
`Match(Eof);`
	b + c = a; } Eof

`Stmt();`
`Stmts();`
`Match("}");`
`Match(Eof);`
	b + c = a; } Eof

`Match(id);`
`Match("=");`
`Expr();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	b + c = a; } Eof

Calls Pending	**Remaining Input**

`Match("=");`
`Expr();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	+ c = a; } Eof

Call to Match fails!	+ c = a; } Eof

Table-Driven Top-Down Parsers

Recursive descent parsers have many attractive features. They are actual pieces of code that can be read by programmers and extended.

This makes it fairly easy to understand how parsing is done.

Parsing procedures are also convenient places to add code to build ASTs, or to do type-checking, or to generate code.

A major drawback of recursive descent is that it is quite inconvenient to change the grammar being parsed. Any change, even a minor one, may force parsing procedures to be reprogrammed, as productions and predict sets are modified.

To a less extent, recursive descent parsing is less efficient than it might be, since subprograms are called just to match a single token or to recognize a righthand side.

An alternative to parsing procedures is to encode all prediction in a parsing table. A pre-programed driver program can use a parse table (and list of productions) to parse any LL(1) grammar.

If a grammar is changed, the parse table and list of productions will change, but the driver need not be changed.

LL(1) Parse Tables

An LL(1) parse table, T, is a two-dimensional array. Entries in T are production numbers or blank (error) entries.

T is indexed by:

- A, a non-terminal. A is the non-terminal we want to expand.
- CT, the current token that is to be matched.
- T[A][CT] = $A \to X_1 \ldots X_n$
  if CT is in Predict($A \to X_1 \ldots X_n$)
  T[A][CT] = error
  if CT predicts no production with A as its lefthand side

CSX-lite Example

	Production	Predict Set
1	Prog $\rightarrow$ { Stmts } Eof	{
2	Stmts $\rightarrow$ Stmt Stmts	id if
3	Stmts $\rightarrow$ λ	}
4	Stmt $\rightarrow$ id = Expr ;	id
5	Stmt $\rightarrow$ if (Expr) Stmt	if
6	Expr $\rightarrow$ id Etail	id
7	Etail $\rightarrow$ + Expr	+
8	Etail $\rightarrow$ - Expr	-
9	Etail $\rightarrow$ λ	) ;

	{	}	if	(	)	id	=	+	-	;	eof
Prog	1										
Stmts		3	2			2					
Stmt			5			4					
Expr						6					
Etail					9			7	8	9	

LL(1) Parser Driver

Here is the driver we'll use with the LL(1) parse table. We'll also use a *parse stack* that remembers symbols we have yet to match.

```
void LLDriver(){
  Push(StartSymbol);
  while(! stackEmpty()){
  //Let X=Top symbol on parse stack
  //Let CT = current token to match
    if (isTerminal(X)) {
      match(X); //CT is updated
      pop();    //X is updated
    } else if (T[X][CT] != Error){
      //Let T[X][CT] = X→Y1...Ym
      Replace X with
         Y1...Ym on parse stack
    } else SyntaxError(CT);
  }
}
```

Example of LL(1) Parsing

We'll again parse
{ a = b + c; } Eof
We start by placing Prog (the start symbol) on the parse stack.

Parse Stack	**Remaining Input**
`Prog`	**{ a = b + c; } Eof**

`{`
`Stmts`
`}`
`Eof`
	{ a = b + c; } Eof

`Stmts`
`}`
`Eof`
	a = b + c; } Eof

`Stmt`
`Stmts`
`}`
`Eof`
	a = b + c; } Eof

Parse Stack	Remaining Input

`id`
`=`
`Expr`
`;`
`Stmts`
`}`
`Eof`
	a = b + c; } Eof

`=`
`Expr`
`;`
`Stmts`
`}`
`Eof`
	= b + c; } Eof

`Expr`
`;`
`Stmts`
`}`
`Eof`
	b + c; } Eof

`id`
`Etail`
`;`
`Stmts`
`}`
`Eof`
	b + c; } Eof

Parse Stack	Remaining Input

`Etail`
`;`
`Stmts`
`}`
`Eof`
	+ c; } Eof

`+`
`Expr`
`;`
`Stmts`
`}`
`Eof`
	+ c; } Eof

`Expr`
`;`
`Stmts`
`}`
`Eof`
	c; } Eof

`id`
`Etail`
`;`
`Stmts`
`}`
`Eof`
	c; } Eof

Parse Stack	Remaining Input

`Etail`
`;`
`Stmts`
`}`
`Eof`
	; } Eof

`;`
`Stmts`
`}`
`Eof`
	; } Eof

`Stmts`
`}`
`Eof`
	} Eof

`}`
`Eof`
	} Eof

`Eof`	Eof
Done!	All input matched