Example: CSX-Lite

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

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

Calls Pending	Remaining Input

`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`

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

Calls Pending	Remaining Input

`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`

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

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

Calls Pending	Remaining Input

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

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

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

`Match("}");` `Match(Eof);`	`} Eof`
`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.
- $T[A][CT] = A \rightarrow X_1 \ldots X_n$
  if CT is in Predict($A \rightarrow X_1 \ldots X_n$)
  T[A][CT] = error
  if CT predicts no production with A as its lefthand side

CSX-lite Example

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

Syntax Errors in LL(1) Parsing

In LL(1) parsing, syntax errors are automatically detected as soon as the first illegal token is seen.

How? When an illegal token is seen by the parser, either it fetches an error entry from the LL(1) parse table *or* it fails to match an expected token.

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

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

(Where should the first syntax error be detected?)

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

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

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

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

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

Parse Stack	Remaining Input

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

Current token (+) fails to match expected token (=)!	+ c = a; } Eof

How do LL(1) Parsers Build Syntax Trees?

So far our LL(1) parser has acted like a recognizer. It verifies that input token are syntactically correct, but it produces no output.

Building complete (concrete) parse trees automatically is fairly easy.

As tokens and non-terminals are matched, they are pushed onto a second stack, the *semantic stack*.

At the end of each production, an action routine pops off n items from the semantic stack (where n is the length of the production's righthand side). It then builds a syntax tree whose root is the

lefthand side, and whose children are the n items just popped off.

For example, for production

Stmt → id = Expr ;

the parser would include an action symbol after the "**;**" whose actions are:

```
P4 = pop(); // Semicolon token
P3 = pop(): // Syntax tree for Expr
P2 = pop(); // Assignment token
P1 = pop(); // Identifier token
Push(new StmtNode(P1,P2,P3,P4));
```

Creating Abstract Syntax Trees

Recall that we prefer that parsers generate abstract syntax trees, since they are simpler and more concise.

Since a parser generator can't know what tree structure we want to keep, we must allow the user to define "custom" action code, just as Java CUP does.

We allow users to include "code snippets" in Java or C. We also allow labels on symbols so that we can refer to the tokens and tress we wish to access. Our production and action code will now look like this:

Stmt → id:i = Expr:e ;

```
{: RESULT = new StmtNode(i,e); :}
```

How do We Make Grammars LL(1)?

Not all grammars are LL(1); sometimes we need to modify a grammar's productions to create the disjoint Predict sets LL1) requires.

There are two common problems in grammars that make unique prediction difficult or impossible:

1. Common prefixes.
   Two or more productions with the same lefthand side begin with the same symbol(s).
   For example,

   Stmt → id = Expr ;

   Stmt → id (Args) ;

2. Left-Recursion

A production of the form

A $\rightarrow$ **A ...**

is said to be left-recursive.

When a left-recursive production is used, a non-terminal is immediately replaced by itself (with additional symbols following).

Any grammar with a left-recursive production can *never* be LL(1).

Why?

Assume a non-terminal A reaches the top of the parse stack, with CT as the current token. The LL(1) parse table entry, T[A][CT], predicts **A** $\rightarrow$ **A ...**

We expand A again, and T[A][CT], so we predict **A** $\rightarrow$ **A ...** again. We are in an infinite prediction loop!

Eliminating Common Prefixes

Assume we have two of more productions with the same lefthand side and a common prefix on their righthand sides:

$\mathbf{A} \rightarrow \alpha\,\beta \mid \alpha\,\gamma \mid \ldots \mid \alpha\,\delta$

We create a new non-terminal, **X**.

We then rewrite the above productions into:

$\mathbf{A} \rightarrow \alpha\mathbf{X} \quad \mathbf{X} \rightarrow \beta \mid \gamma \mid \ldots \mid \delta$

For example,

Stmt → id = Expr ;

Stmt → id (Args) ;

becomes

Stmt → id StmtSuffix

StmtSuffix → = Expr ;

StmtSuffix → (Args) ;

Eliminating Left Recursion

Assume we have a non-terminal that is left recursive:

$\mathbf{A} \rightarrow \mathbf{A}\alpha \quad \mathbf{A} \rightarrow \beta \mid \gamma \mid \ldots \mid \delta$

To eliminate the left recursion, we create two new non-terminals, **N** and **T**.

We then rewrite the above productions into:

$\mathbf{A} \rightarrow \mathbf{N}\ \mathbf{T} \quad \mathbf{N} \rightarrow \beta \mid \gamma \mid \ldots \mid \delta$

$\mathbf{T} \rightarrow \alpha\ \mathbf{T} \mid \lambda$

For example,

Expr $\rightarrow$ **Expr + id**

Expr $\rightarrow$ **id**

becomes

Expr $\rightarrow$ **N T**

N $\rightarrow$ **id**

T $\rightarrow$ **+ id T** | λ

This simplifies to:

Expr $\rightarrow$ **id T**

T $\rightarrow$ **+ id T** | λ