

Lecture 14

The LL(1) Property

If $A \rightarrow a$ and $A \rightarrow b$ both appear in the grammar, we would like

$$\text{FIRST}(a) \cap \text{FIRST}(b) = \emptyset$$

Predictive parsers accept LL(k) grammars. The two LL stand for Left-to-right scan of input, left-most derivation. The k stands for number of look-ahead tokens of input. The LL(1) Property allows the parser to make a correct choice with a look-ahead of exactly *one symbol*! What about ϵ -productions? They complicate the definition of LL(1).

If $A \rightarrow a$ and $A \rightarrow b$ and $\epsilon \in \text{FIRST}(a)$, then we need to ensure that $\text{FIRST}(b)$ is disjoint from $\text{FOLLOW}(a)$, too.

Definition: $\text{FOLLOW}(a)$ is the set of all words in the grammar that can legally appear after an a .

For a non-terminal X , $\text{FOLLOW}(X)$ is the set of symbols that might follow the derivation of X . Define $\text{FIRST}^+(a)$ as $\text{FIRST}(a) \cup \text{FOLLOW}(a)$, if $\epsilon \in \text{FIRST}(a)$, $\text{FIRST}(a)$, otherwise. Then a grammar is LL(1) iff $A \rightarrow a$ and $A \rightarrow b$ implies

$$\text{FIRST}^+(a) \cap \text{FIRST}^+(b) = \emptyset$$

Given a grammar that has the LL(1) property, we can write a simple routine to recognize each lhs. The code is simple and fast. Consider

$A \rightarrow b_1 \mid b_2 \mid b_3$,

which satisfies the LL(1) property $\text{FIRST}^+(a) \cap \text{FIRST}^+(b) = \emptyset$

```
/* find an A */
if(token ∈ FIRST(b1))
    find a b1 and return true
else if(token ∈ FIRST(b2))
    find a b2 and return true
if(token ∈ FIRST(b3))
    find a b3 and return true
else error and return false
```

Grammar with the LL(1) property are called *predictive grammars* because the parser can “predict” the correct expansion at each point in the parse. Parsers that capitalize on the LL(1) property are called *predictive parsers*. One kind of predictive parser is the *recursive descent* parser.

Recursive Descent Parsing

Consider the right-recursive expression grammar

1	<i>Goal</i>	?	<i>expr</i>
2	<i>expr</i>	?	<i>term expr'</i>
3	<i>expr'</i>	?	<i>+ term expr'</i>
4			<i>- term expr'</i>
5			<i>e</i>
6	<i>term</i>	?	<i>factor term'</i>
7	<i>term'</i>	?	<i>* factor term'</i>
8			<i>/ factor term'</i>
9			<i>e</i>
10	<i>factor</i>	?	<u><i>number</i></u>
11			<u><i>id</i></u>
12			<i>(expr)</i>

This leads to a parser with six mutually recursive routines: *goal*, *expr*, *exprime*, *term*, *termprime* and *factor*. Each recognizes one non-terminal (NT) or terminal (T). The term *descent* refers to the direction in which the *parse tree* is built. Here are some of these routines written as functions:

```
Goal() {
    token = next_token();
    if(Expr() == true && token == EOF)
        next compilation step
    else {
        report syntax error;
        return false;
    }
}
Expr()
{
    if(Term() == false)
        return false;
    else
        return Exprime();
}
Exprime() {
    token_type op = next_token();
    if( op == PLUS || op == MINUS ) {
        if(Term() == false)
            return false;
        else
            return Exprime();
    }
}
```

Functions for other non-terminals Term, Factor and Tprime follow the same pattern.

Recursive Descent in C++

This form of the routines is too procedural. Moreover, there is no convenient way to build the parse tree. We can use C++ to code the recursive descent parser in an object oriented manner. We associate a C++ class with each non-terminal symbol. An instantiated object of a non-terminal class contains pointer to the parse tree. Here are the C++ code for the non-terminal classes:

```
class NonTerminal {
public:
    NonTerminal(Scanner* sc)
    {
        s = sc; tree = NULL;
    }
    virtual ~NonTerminal(){}
    virtual bool isPresent()=0;
    TreeNode* AST(){
        return tree;
    }

protected:
    Scanner* s;
    TreeNode* tree;
}

class Expr:public NonTerminal
{
public:
    Expr(Scanner* sc): NonTerminal(sc){ }
    virtual bool isPresent();
}

class Eprime:public NonTerminal {
public:
    Eprime(Scanner* sc, TreeNode* t): NonTerminal(sc)
    {
        exprSofar = t;
    }
    virtual bool isPresent();
protected:
    TreeNode* exprSofar;
}
```

```

class Term:public NonTerminal
{
public:
    Term(Scanner* sc): NonTerminal(sc){ }
    virtual bool isPresent();
}

class Tprime:public NonTerminal {
public:
    Tprime(Scanner* sc, TreeNode* t): NonTerminal(sc)
    {
        exprSofar = t;
    }
    virtual bool isPresent();
protected:
    TreeNode* exprSofar;
}

class Factor:public NonTerminal {
public:
    Factor(Scanner* sc, TreeNode* t): NonTerminal(sc){ };

    virtual bool isPresent();
}

```