Parsing III
(Top-down parsing: recursive descent & *LL(1)*)

Roadmap (Where are we?)

We set out to study parsing

- Specifying syntax
  - → Context-free grammars ✔
  - → Ambiguity ✔

- Top-down parsers
  - → Algorithm & its problem with left recursion ✔
  - → Left-recursion removal ✔

- Predictive top-down parsing
  - → The LL(1) condition today
  - → Simple recursive descent parsers today
  - → Table-driven LL(1) parsers today

Picking the “Right” Production

If it picks the wrong production, a top-down parser may backtrack

Alternative is to look ahead in input & use context to pick correctly

How much lookahead is needed?

- In general, an arbitrarily large amount
- Use the Cocke-Younger, Kasami (CYK) algorithm or Earley’s algorithm

Fortunately,

- Large subclasses of CFGs can be parsed with limited lookahead
- Most programming language constructs fall in those subclasses

Among the interesting subclasses are *LL(1)* and *LR(1)* grammars

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, *the parser should be able to choose between* α *&* β

FIRST sets

For some *rhs* $\alpha \in G$, define FIRST(α) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $\underline{x} \in$ FIRST(α) *iff* $\alpha \Rightarrow^* \underline{x}\,\gamma$, for some γ

We will defer the problem of how to compute FIRST sets until we look at the *LR(1)* table construction algorithm

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, the parser should be able to choose between α & β

FIRST sets

For some *rhs* $\alpha \in G$, define FIRST(α) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $\underline{x} \in$ FIRST(α) *iff* $\alpha \Rightarrow^* \underline{x}\,\gamma$, for some γ

The LL(1) Property

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \varnothing$$

This would allow the parser to make a correct choice with a lookahead of exactly one symbol !

This is almost correct
See the next slide

Predictive Parsing

What about ε-productions?

$\Rightarrow$ They complicate the definition of LL(1)

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\varepsilon \in$ FIRST(α), then we need to ensure that FIRST(β) is disjoint from FOLLOW(α), too

Define FIRST+(α) as

- FIRST(α) $\cup$ FOLLOW(α), if $\varepsilon \in$ FIRST(α)
- FIRST(α), otherwise

Then, a grammar is *LL(1)* iff $A \rightarrow \alpha$ and $A \rightarrow \beta$ implies

$$\text{FIRST}^+(\alpha) \cap \text{FIRST}^+(\beta) = \varnothing$$

FOLLOW(α) is the set of all words in the grammar that can legally appear immediately after an α

FIRST and FOLLOW Sets

FIRST(α)

For some $\alpha \in T \cup NT$, define FIRST(α) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $\underline{x} \in$ FIRST(α) *iff* $\alpha \Rightarrow^* \underline{x}\, \gamma$, for some γ

FOLLOW(α)

For some $\alpha \in NT$, define FOLLOW(α) as the set of symbols that can occur immediately after α in a valid sentence.

FOLLOW(S) = {EOF}, where S is the start symbol

To build FOLLOW sets, we need FIRST sets …

Computing FOLLOW Sets

$FOLLOW(S) \leftarrow \{EOF\}$

for each $A \in NT$, $FOLLOW(A) \leftarrow \emptyset$

while (FOLLOW sets are still changing)

    for each $p \in P$, of the form $A \rightarrow \beta_1 \beta_2 \ldots \beta_k$

        $FOLLOW(\beta_k) \leftarrow FOLLOW(\beta_k) \cup FOLLOW(A)$

        $TRAILER \leftarrow FOLLOW(A)$

        for $i \leftarrow k$ down to 2

            if $\varepsilon \in FIRST(\beta_i)$ then

                $FOLLOW(\beta_{i-1}) \leftarrow FOLLOW(\beta_{i-1}) \cup \{ FIRST(\beta_i) - \{ \varepsilon \}\}$
$\cup \; TRAILER$

            else

                $FOLLOW(\beta_{i-1}) \leftarrow FOLLOW(\beta_{i-1}) \cup FIRST(\beta_i)$

                $TRAILER \leftarrow \emptyset$

Computing FIRST Sets

for each $\alpha \in (T \amalg \varepsilon)$

  FIRST(α) $\leftarrow \alpha$

for each $A \in NT$

  FIRST(A) $\leftarrow \emptyset$

while (FIRST sets are still changing)

  for each $p \in P$, where p has the form $A \rightarrow \beta$

    if β is $\beta_1\beta_2...\beta_k$, where $\beta_i \in T \amalg NT$, *then*

      FIRST(A) $\leftarrow$ FIRST(A) $\amalg$ (FIRST(β_1) – $\{\varepsilon\}$)

      $i \leftarrow 1$

      while ($\varepsilon \in$ FIRST(β_i) and $i < k$)

        FIRST(A) $\leftarrow$ FIRST(A) $\amalg$ (FIRST(β_{i+1}) – $\{\varepsilon\}$)

        $i \leftarrow i+1$

    if $i = k$ and $\varepsilon \in$ FIRST(β_k), *then*

      FIRST(A) $\leftarrow$ FIRST(A) $\amalg$ $\{\varepsilon\}$

Predictive Parsing

Given a grammar that has the *LL(1)* property

- Can write a simple routine to recognize each *lhs*
- Code is both simple & fast

Consider $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, with

$$\text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_2) \cap \text{FIRST}^+(\beta_3) = \varnothing$$

```
/* find an A */
if (current_word ∈ FIRST(β1))
    find a β1 and return true
else if (current_word ∈ FIRST(β2))
    find a β2 and return true
else if (current_word ∈ FIRST(β3))
    find a β3 and return true
else
    report an error and return false
```

Grammars 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.

Of course, there is more detail to "find a β_i" (§ 3.3.4 in EAC)

Recursive Descent Parsing

Recall the expression grammar, after transformation

1	Goal	→	Expr
2	Expr	→	Term Expr'
3	Expr'	→	+ Term Expr'
4		|	- Term Expr'
5		|	ε
6	Term	→	Factor Term'
7	Term'	→	* Factor Term'
8		|	/ Factor Term'
9		|	ε
10	Factor	→	<u>number</u>
11		|	<u>id</u>

This produces a parser with six *mutually recursive* routines:

- *Goal*
- *Expr*
- *Expr'*
- *Term*
- *Term'*
- *Factor*

Each recognizes one *NT* or *T*

The term *descent* refers to the direction in which the parse tree is built.

Recursive Descent Parsing (Procedural)

A couple of routines from the expression parser

```
Goal( )
   token ← next_token( );
   if (Expr( ) = true & token = EOF)
      then next compilation step;
      else
         report syntax error;
         return false;

Expr( )
  if (Term( ) = false)
     then return false;
     else return Eprime( );
```

looking for EOF, found token

```
Factor( )
  if (token = Number) then
     token ← next_token( );
     return true;
  else if (token = Identifier) then
     token ← next_token( );
     return true;
  else
     report syntax error;
     return false;
```

EPrime, Term, & TPrime follow the same basic lines (Figure 3.7, EAC)

looking for Number or Identifier, found token instead

Recursive Descent Parsing

To build a parse tree:

- Augment parsing routines to build nodes
- Pass nodes between routines using a stack
- Node for each symbol on *rhs*
- Action is to pop *rhs* nodes, make them children of *lhs* node, and push this subtree

To build an abstract syntax tree

- Build fewer nodes
- Put them together in a different order

```
Expr( )
   result ← true;
   if (Term( ) = false)
     then return false;
     else if (EPrime( ) = false)
            then result ← false;
            else
              build an Expr node
              pop EPrime node
              pop Term node
              make EPrime & Term
                children of Expr
              push Expr node
   return result;
```

Success ⇒ build a piece of the parse tree

Left Factoring

What if my grammar does not have the LL(1) property?

$\Rightarrow$ Sometimes, we can transform the grammar

The Algorithm

$\forall$ *A* $\in$ *NT,*

find the longest prefix α *that occurs in two or more right-hand sides of A*

if $\alpha \neq \varepsilon$ *then replace all of the A productions,*

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma\ ,$$

with

$$A \rightarrow \alpha\, Z \mid \gamma$$
$$Z \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

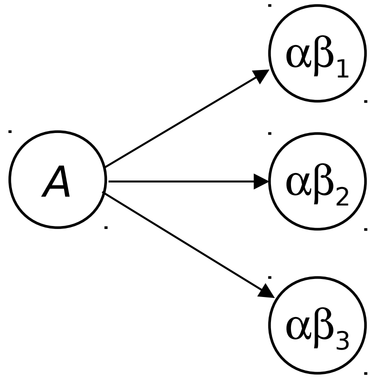
where Z is a new element of NT

Repeat until no common prefixes remain

Left Factoring

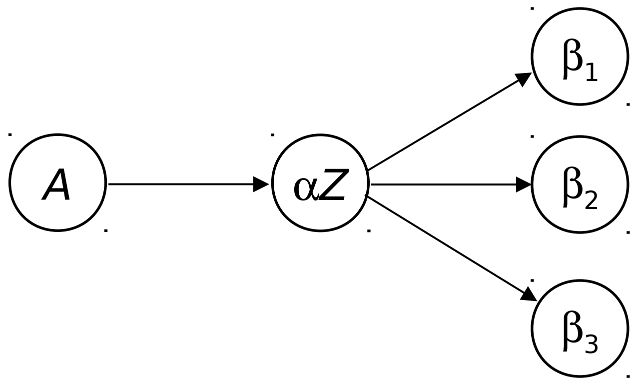
A graphical explanation for the same idea

$$
\begin{aligned}
A \rightarrow\ & \alpha\beta_1 \\
\mid\ & \alpha\beta_2 \\
\mid\ & \alpha\beta 3
\end{aligned}
$$



becomes …

$$
\begin{aligned}
A &\rightarrow \alpha\ Z \\
Z &\rightarrow \beta_1 \\
&\mid\ \ \beta_2 \\
&\mid\ \ \beta_n
\end{aligned}
$$



Left Factoring (An example)

Consider the following fragment of the expression grammar

1	Factor	$\rightarrow$	<u>Identifier</u>
2		|	<u>Identifier</u> <u>[</u> ExprList <u>]</u>
3		|	<u>Identifier</u> <u>(</u> ExprList <u>)</u>

FIRST(rhs_1) = { <u>Identifier</u> }
FIRST(rhs_2) = { <u>Identifier</u> }
FIRST(rhs_3) = { <u>Identifier</u> }

After left factoring, it becomes

1	Factor	$\rightarrow$	<u>Identifier</u> Arguments
2	Arguments	$\rightarrow$	<u>[</u> ExprList <u>]</u>
3		|	<u>(</u> ExprList <u>)</u>
4		|	ε

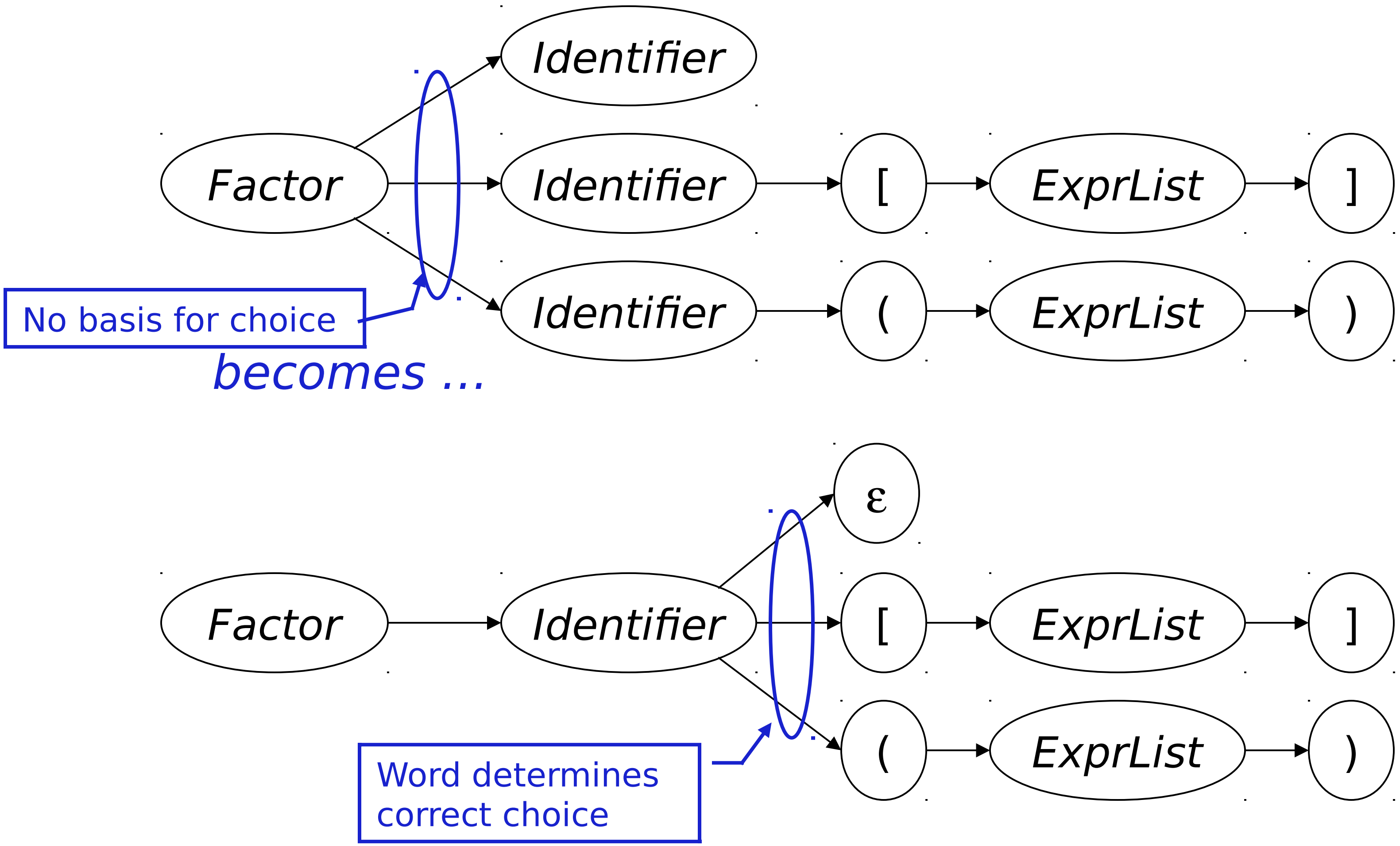
FIRST(rhs_1) = { <u>Identifier</u> }
FIRST(rhs_2) = { <u>[</u> }
FIRST(rhs_3) = { <u>(</u> }
FIRST(rhs_4) = FOLLOW(*Factor*)

$\Rightarrow$ It has the *LL(1)* property

This form has the same syntax, with the *LL(1)* property

Left Factoring

Graphically



Left Factoring (Generality)

Question

By *eliminating left recursion* and *left factoring*, can we transform an arbitrary CFG to a form where it meets the *LL(1)* condition? (and can be parsed predictively with a single token lookahead?)

Answer

Given a CFG that doesn't meet the *LL(1)* condition, it is undecidable whether or not an equivalent *LL(1)* grammar exists.

Example

$\{a^n \, 0 \, b^n \mid n \geq 1\} \cup \{a^n \, 1 \, b^{2n} \mid n \geq 1\}$ has no *LL(1)* grammar

Language that Cannot Be LL(1)

Example

$\{a^n 0 b^n \mid n \geq 1\} \cup \{a^n 1 b^{2n} \mid n \geq 1\}$ has no *LL(1)* grammar

$G \rightarrow \underline{a}A\underline{b}$

$\quad \mid \underline{a}B\underline{bb}$

$A \rightarrow \underline{a}A\underline{b}$

$\quad \mid \underline{0}$

$B \rightarrow \underline{a}B\underline{bb}$

$\quad \mid \underline{1}$

Problem: need an unbounded number of a characters before you can determine whether you are in the A group or the B group.

Recursive Descent (Summary)

1. Build FIRST (and FOLLOW) sets
2. Massage grammar to have *LL(1)* condition
   a. Remove left recursion
   b. Left factor it
3. Define a procedure for each non-terminal
   a. Implement a case for each right-hand side
   b. Call procedures as needed for non-terminals
4. Add extra code, as needed
   a. Perform context-sensitive checking
   b. Build an IR to record the code

Can we automate this process?

Building Top-down Parsers

Given an *LL(1)* grammar, and its FIRST & FOLLOW sets …

- Emit a routine for each non-terminal
  - → Nest of if-then-else statements to check alternate rhs's
  - → Each returns true on success and throws an error on false
  - → Simple, working *(, perhaps ugly,)* code
- This automatically constructs a recursive-descent parser

Improving matters

- Nest of if-then-else statements may be slow
  - → Good case statement implementation would be better
- What about a table to encode the options?
  - → Interpret the table with a skeleton, as we did in scanning

Building Top-down Parsers

Strategy

- Encode knowledge in a table
- Use a standard “skeleton” parser to interpret the table

Example

- The non-terminal *Factor* has three expansions
  - → (*Expr*) or Identifier or Number
- Table might look like:

Terminal Symbols

Non-terminal Symbols

	+	-	*	/	Id.	Num.	EOF
Factor	—	—	—	—	10	11	—

Error on \` + ’

Reduce by rule 10 on \` + ’

Building Top Down Parsers

Building the complete table

- Need a row for every *NT* & a column for every *T*
- Need a table-driven interpreter for the table

LL(1) Skeleton Parser

```
token ← next_token()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
   if TOS = EOF and token = EOF then
      break & report success
   else if TOS is a terminal then
      if TOS matches token then
         pop Stack                    // recognized TOS
         token ← next_token()
      else report error looking for TOS
   else                               // TOS is a non-terminal
      if TABLE[TOS,token] is A→ B1B2…Bk then
         pop Stack                    // get rid of A
         push Bk, Bk-1, …, B1         // in that order
      else report error expanding TOS
   TOS ← top of Stack
```

exit on success

Building Top Down Parsers

Building the complete table

- Need a row for every *NT* & a column for every *T*
- Need an algorithm to build the table

Filling in TABLE[X,y], X $\in$ *NT*, y $\in$ *T*

1. entry is the rule $X \rightarrow \beta$, if y $\in$ FIRST(β)
2. entry is the rule $X \rightarrow \varepsilon$ if y $\in$ FOLLOW(X) and $X \rightarrow \varepsilon \in$ G
3. entry is error if neither 1 nor 2 define it

If any entry is defined multiple times, G is not *LL(1)*

This is the *LL(1)* table construction algorithm

Extra Slides Start Here

Recursive Descent in Object-Oriented Languages

- Shortcomings of Recursive Descent
  - → Too procedural
  - → No convenient way to build parse tree
- Solution
  - → Associate a class with each non-terminal symbol
    - Allocated object contains pointer to the parse tree

```
Class NonTerminal {

public:
      NonTerminal(Scanner & scnr) { s = &scnr; tree = NULL; }
      virtual ~NonTerminal() { }
      virtual bool isPresent() = 0;
      TreeNode * abSynTree() { return tree; }

protected:
      Scanner * s;
      TreeNode * tree;
}
```

Non-terminal Classes

```
Class Expr : public NonTerminal {
public:
        Expr(Scanner & scnr) : NonTerminal(scnr) { }
        virtual bool isPresent();
}

Class EPrime : public NonTerminal {
public:
        EPrime(Scanner & scnr, TreeNode * p) :
                NonTerminal(scnr) { exprSofar = p; }
        virtual bool isPresent();
protected:
        TreeNode * exprSofar;
}

…  // definitions for Term and TPrime

Class Factor : public NonTerminal {
public:
        Factor(Scanner & scnr) : NonTerminal(scnr) { };
        virtual bool isPresent();
}
```

Implementation of isPresent

```
bool Expr::isPresent() {

    Term * operand1 = new Term(*s);
    if (!operand1->isPresent()) return FALSE;

    Eprime * operand2 = new EPrime(*s, NULL);
    if (!operand2->isPresent()) // do nothing;

    return TRUE;
}
```

Implementation of isPresent

```
bool EPrime::isPresent() {

    token_type op = s->nextToken();
    if (op == PLUS || op == MINUS) {
        s->advance();

        Term * operand2 = new Term(*s);
        if (!operand2->isPresent()) throw SyntaxError(*s);

        Eprime * operand3 = new EPrime(*s, NULL);
        if (operand3->isPresent()); //do nothing

        return TRUE;
    }
    else return FALSE;
}
```

Abstract Syntax Tree Construction

```
bool Expr::isPresent() { // with semantic processing

      Term * operand1 = new Term(*s);
      if (!operand1->isPresent()) return FALSE;
      tree = operand1->abSynTree();

      EPrime * operand2 = new EPrime(*s, tree);
      if (operand2->isPresent())
            tree  = operand2->absSynTree();

      // here tree is either the tree for the Term
      //           or the tree for Term followed by EPrime
      return TRUE;
}
```

Abstract Syntax Tree Construction

```
bool EPrime::isPresent() { // with semantic processing
    token_type op = s->nextToken();
    if (op == PLUS || op == MINUS) {
        s->advance();

        Term * operand2 = new Term(*s);
        if (!operand2->isPresent()) throw SyntaxError(*s);

        TreeNode * t2 = operand2->absSynTree();
        tree = new TreeNode(op, exprSofar, t2);

        Eprime * operand3 = new Eprime(*s, tree);
        if (operand3->isPresent())
            tree = operand3->absSynTree();
        return TRUE;
    }
    else return FALSE;
}
```

Factor

```
bool Factor::isPresent() { // with semantic processing
    token_type op = s->nextToken();

    if (op == IDENTIFIER | op == NUMBER) {
        tree = new TreeNode(op, s->tokenValue());
        s->advance();
        return TRUE;
    }
    else if (op == LPAREN) {
        s->advance();
        Expr * operand = new Expr(*s);
        if (!operand->isPresent()) throw SyntaxError(*s);
        if (s->nextToken() != RPAREN) throw SyntaxError(*s);
        s->advance();
        tree = operand->absSynTree();
        return TRUE;
    }
    else return FALSE;
}
```