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

COMP 412
Fall 2005

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 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(A), too, where

FOLLOW(A) = the set of terminal symbols that can immediately follow A in a sentential form

Define FIRST$^+$($A\rightarrow\alpha$) as

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

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

FIRST$^+$($A\rightarrow\alpha$) $\cap$ FIRST$^+$($A\rightarrow\beta$) = $\varnothing$

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

$$\text{FIRST}^+(A \rightarrow \beta_i) \cap \text{FIRST}^+(A \rightarrow \beta_j) = \varnothing \text{ if } i \neq j$$

```
/* find an A */
if (current_word ∈ FIRST(A→β1))
   find a β1 and return true
else if (current_word ∈ FIRST(A→β2))
   find a β2 and return true
else if (current_word ∈ FIRST(A→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" **(p. 103 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*	→	number
11		|	id
12		|	(Expr)

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

- *Goal*
- *Expr*
- *EPrime*
- *Term*
- *TPrime*
- *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 Number, Identifier, or "(", found token instead, or failed to find Expr or ")" after "("

```
Factor( )
  if (token = Number) then
     token ← next_token( );
     return true;
  else if (token = Identifier) then
     token ← next_token( );
     return true;
  else if (token = Lparen)
     token ← next_token( );
     if (Expr( ) = true & token = Rparen) then
        token ← next_token( );
        return true;
  // fall out of if statement
  report syntax error;
     return false;
```

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

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

This is a preview of Chapter 4

Left Factoring

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

⇒ 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,

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

with

$$A \rightarrow \alpha\, Z \mid \gamma$$

$$Z \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

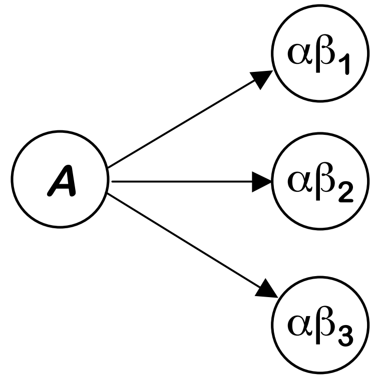
where Z is a new element of NT

Repeat until no common prefixes remain

Left Factoring

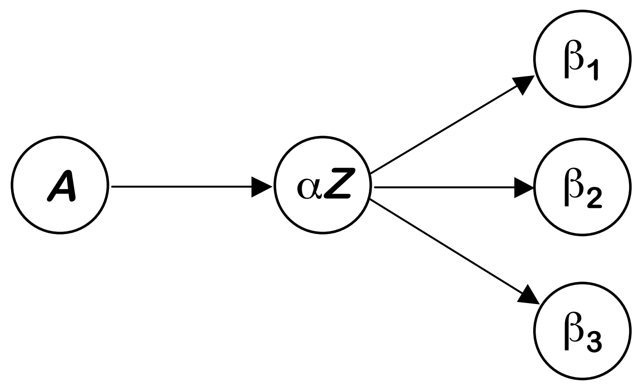
A graphical explanation for the same idea

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



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

Factor	→	Identifier
	|	Identifier [*ExprList*]
	|	Identifier (*ExprList*)

FIRST(rhs_1) = { Identifier }
FIRST(rhs_2) = { Identifier }
FIRST(rhs_3) = { Identifier }

After left factoring, it becomes

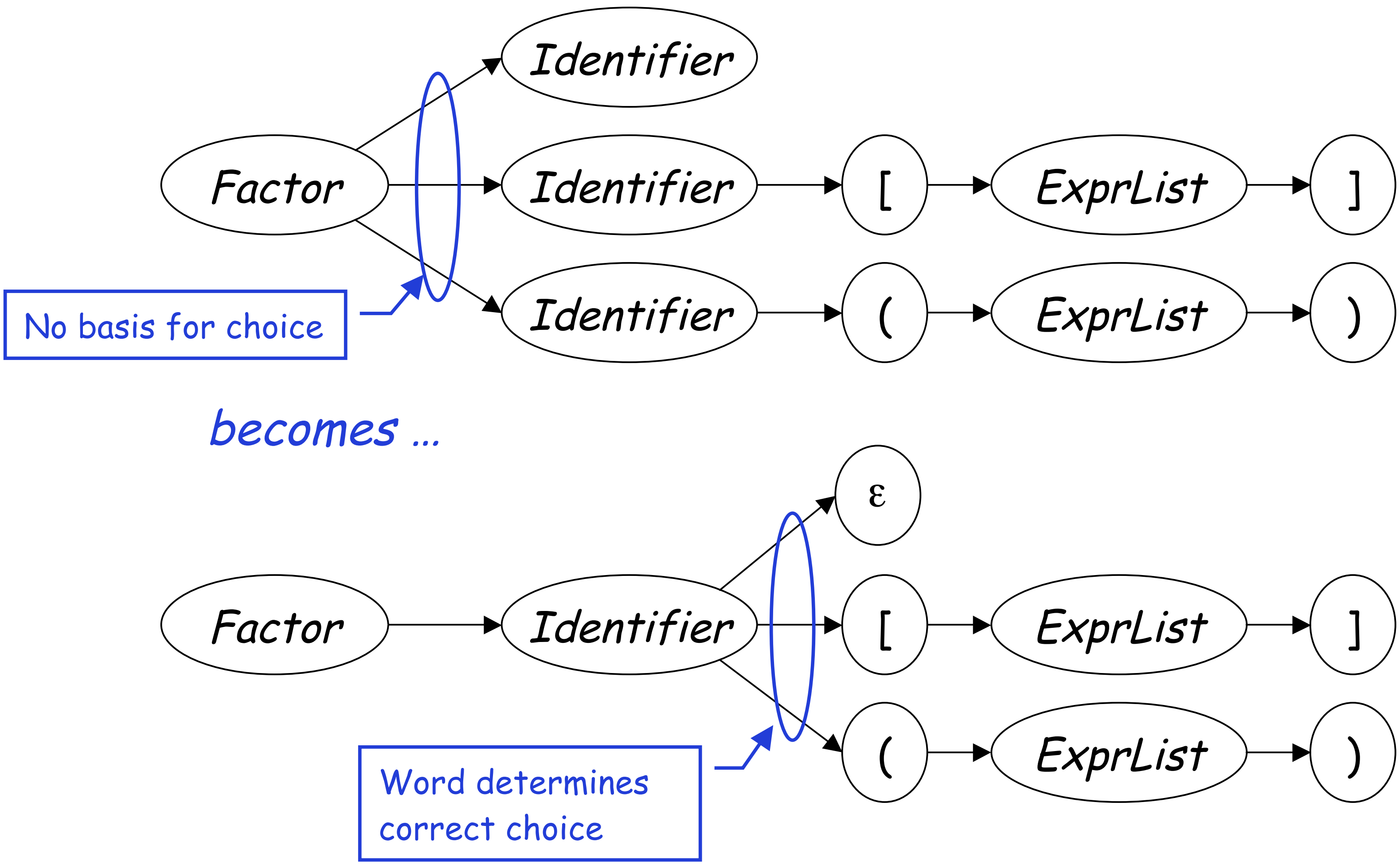
Factor	→	Identifier *Arguments*
Arguments	→	[*ExprList*]
	|	(*ExprList*)
	|	ε

FIRST(rhs_1) = { Identifier }
FIRST(rhs_2) = { [}
FIRST(rhs_3) = { (}
FIRST(rhs_4) = FOLLOW(*Factor*)
⇒ 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

$$
\begin{aligned}
G &\rightarrow \underline{a}A\underline{b} \\
&\mid \underline{a}B\underline{bb} \\
A &\rightarrow \underline{a}A\underline{b} \\
&\mid \underline{0} \\
B &\rightarrow \underline{a}B\underline{bb} \\
&\mid \underline{1}
\end{aligned}
$$

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?

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(A)

For some $A \in NT$, define FOLLOW(A) as the set of symbols that can occur immediately after A in a valid sentential form

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

To build FIRST sets, we need FOLLOW sets …

Next lecture, we'll look at how to compute these sets …

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

I don't know of a system that does this …

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	12	—	—

Error on `+'

Reduce by rule 10 on `id'

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          exit on 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→ B_1B_2…B_k then
        pop Stack                   // get rid of A
        push B_k, B_k-1, …, B_1     // in that order
     else report error expanding TOS
  TOS ← top of Stack
```

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

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 $y \in \text{FIRST}^{+}(X \rightarrow \beta)$
2. entry is error if rule 1 does not define

If any entry has more than one rule, *G* is not *LL(1)*

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

LL(1) Expression Parser

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*	→	id
11		|	number
12		|	(Expr)

FIRST(*Goal*) = FIRST(*Expr*) =
FIRST(*Term*) = FIRST(*Factor*) =
{ id, number, (}

FIRST(*Expr′*) = { +, -, ε }

FIRST(*Term′*) = { *, /, ε }

FOLLOW(*Goal*) = { EOF }

FOLLOW(*Expr*) = {), EOF }

FOLLOW(*Expr′*) = {), EOF }

FOLLOW(*Term*) = { +, -,), EOF }

FOLLOW(*Term′*) = { +, -,), EOF }

FOLLOW(*Factor*) =
{ +, -, *, /,), EOF }

LL(1) Expression Parsing Table

	+	-	*	/	Id	Num	(	)	EOF
Goal	—	—	—	—	1	1	1	—	—
Expr	—	—	—	—	2	2	2	—	—
Expr'	3	4	—	—	—	—	—	5	5
Term	—	—	—	—	6	6	6	—	—
Term'	9	9	7	8	—	—	—	9	9
Factor	—	—	—	—	10	11	12	—	—