Outline

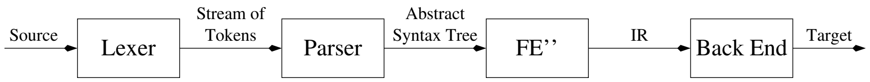
- Recursive Descent Parsing
- Shift-Reduce Parsing
- ML-Yacc
- Recursive Descent Parser Generation

Syntax Analysis

Front End:

- Lexical Analysis - Break source into *tokens*.
- Syntax Analysis - Parse phrase structure.
- Semantic Analysis - Calculate meaning.

Our Compiler:



Parser Functions:

- Verify that token stream is valid.
- If it is not valid, report syntax error and recover.
- Build Abstract Syntax Tree (AST).

Recursive Descent Parsing

- Recall discussion on Context-Free Grammars: symbols (terminal, non-terminal), productions, derivations, etc.
- Can parse many grammars using algorithm called *recursive descent* parsing.
  - A.K.A.: *predictive parsing*
  - A.K.A.: *top-down parsing*
  - A.K.A.: *LL(1)* - Left-to-right parse, Leftmost-derivation, 1-symbol lookahead.
- One recursive function for each non-terminal.
- Each production becomes clause in function.

Example

Grammar:
non-terminals: `S, L, E`

terminals: `IF`(*if*), `THEN`(*then*), `ELSE`(*else*), `BEGIN`(*begin*), `PRINT`(*print*), `END`(*end*), `SEMI`(*;*), `NUM`, `EQ`(=)

S → *if* E *then* S *else* S
S → *begin* S L
S → *print* E
L → *end*
L → *;* S L
E → *num* = *num*

Example

```
datatype token = EOF | IF | THEN | ELSE | BEGIN |
                 PRINT | END | SEMI | NUM | EQ

val tok = ref (getToken())
fun advance() = tok := getToken()
fun eat(t) = if (!tok = t) then advance() else error()

fun S() = case !tok of
            IF    => (eat(IF); E(); eat(THEN); S();
                      eat(ELSE); S())
            BEGIN => (eat(BEGIN); S(); L())
            PRINT => (eat(PRINT); E())
and L() = case !tok of
            END   => (eat(END))
            SEMI  => (eat(SEMI); S(); L())
and E() =            (eat(NUM); eat(EQ); eat(NUM))
```


Another Example

Grammar:

A → S EOF
S → *id* := E
S → *print* (L)

E → *id*
E → *num*
L → E
L → L, E

```
fun A() =                (S(); eat(EOF))
and S() = case !tok of
              ID     => (eat(ID); eat(ASSIGN); E())
              PRINT  => (eat(PRINT); eat(LPAREN);
                          L(); eat(RPAREN))
and E() = case !tok of
              ID     => (eat(ID))
              NUM    => (eat(NUM))
and L() = case !tok of
              ID     => (?????)
              NUM    => (?????)
```


Problem

- If !tok = ID, parser cannot determine which production to use:
  L → E (E could be ID)
  L → L, E (L could be ID)
- Predictive parsing only works for grammars where first terminal symbol of each subexpression provides enough information to choose which production to use.
- Can write predictive parser by eliminating *left recursion*.

L → E
L → L, E

$\Longrightarrow$

L → E M
M → , E M
M → ϵ

```
and L() = case !tok of
            ID      => (E(); M())
            NUM     => (E(); M())
and M() = case !tok of
            COMMA    => (eat(COMMA); E(); M())
            RPAREN   => ()
```


Shift-Reduce Parsing

- Given next input token, predictive parser must predict which production to use.
- *Shift-reduce parsing* delays decision until it has seen input token corresponding to entire RHS of production.
  - A.K.A.: *bottom-up parsing*
  - A.K.A.: *LR(k)* - Left-to-right parse, Rightmost derivation, k-token lookahead
- Shift-reduce parsing can parse more grammars than predictive parsing.
- Parser has *stack*.
- Based on stack contents and next input token, one of two action performed:
  1. *Shift* - push next input token onto top of stack.
  2. *Reduce* - choose production (X → ABC); pop off RHS (C, B, A); push LHS (X).
- Stack is initially empty.
- Parser points to beginning of input stream.
- If \$ is shifted, then input stream has been parsed successfully.

Shift-Reduce Parsing

How does parser know when to shift or reduce?

- DFA: applied to stack contents, not input stream
- Each state corresponds to contents of stack at some point in time.
- Edges labelled with terms/non-terms that can appear on stack.

Example

Grammar:

1 A $\rightarrow$ S EOF
2 S $\rightarrow$ (L)
3 S $\rightarrow$ *id* = *num*
4 L $\rightarrow$ L; S
5 L $\rightarrow$ S

Input:

$(a = 4; b = 5) \rightarrow (ID_a = NUM_4; ID_b = NUM_5)$

0
```
 input: ( ID = NUM ; ID = NUM )
        |
 stack:
action: shift
```


Example

1
```
 input: ( ID = NUM ; ID = NUM )
          |
 stack: (
action: shift
```

2
```
 input: ( ID = NUM ; ID = NUM )
             |
 stack: ( ID
action: shift
```

3
```
 input: ( ID = NUM ; ID = NUM )
               |
 stack: ( ID =
action: shift
```

4
```
 input: ( ID = NUM ; ID = NUM )
                   |
 stack: ( ID = NUM
action: reduce 3
```


Example

5
```
 input: ( ID = NUM ; ID = NUM )
                   |
 stack: ( S
action: reduce 5
```

6
```
 input: ( ID = NUM ; ID = NUM )
                   |
 stack: ( L
action: shift
```

7
```
 input: ( ID = NUM ; ID = NUM )
                     |
 stack: ( L ;
action: shift
```

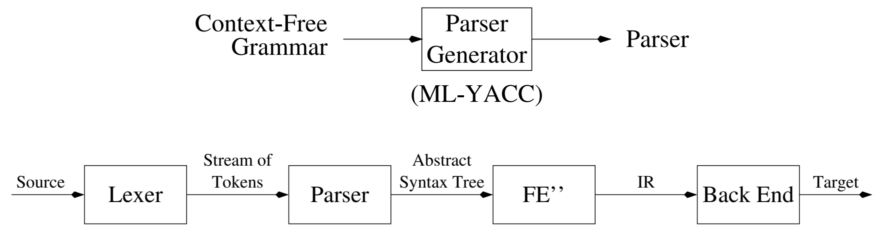
8
```
 input: ( ID = NUM ; ID = NUM )
                        |
 stack: ( L; ID
action: shift
```

Example

13
```
 input:  ( ID = NUM ; ID = NUM )
                                |
 stack:  ( L )
action:  reduce 2
```

14
```
 input:  ( ID = NUM ; ID = NUM )
                                |
 stack:  S
action:  ACCEPT
```


ML-YACC ("Yet Another Compiler-Compiler")



- Input to **ml-yacc** is a context-free grammar specification.
- Output from **ml-yacc** is a shift-reduce parser in ML.

Example

9
```
 input:  ( ID = NUM ; ID = NUM )
                       |
 stack:  ( L ; ID =
action:  shift
```

10
```
 input:  ( ID = NUM ; ID = NUM )
                              |
 stack:  ( L ; ID = NUM
action:  reduce 3
```

11
```
 input:  ( ID = NUM ; ID = NUM )
                              |
 stack:  ( L ; S
action:  reduce 4
```

12
```
 input:  ( ID = NUM ; ID = NUM )
                              |
 stack:  ( L
action:  shift
```


The Dangling Else Problem

- Valid Program: if a then if b then S1 else S2

  1 S → *if* E *then* S *else* S
  2 S → *if* E *then* S
  3 S → OTHER

- 2 interpretations: if a then [if b then S1 else S2]
  if a then [if b then S1] else S2
- Want first behavoir, but parse will report *shift-reduce conflict* when S1 is on top stack.
- Eliminate Ambiguity by modifying grammar (matched/unmatched):

  1 S → M
  2 S → U
  3 M → *if* E *then* M *else* M
  4 M → OTHER
  5 U → *if* E *then* S
  6 U → *if* E *then* M *else* U

Context-Free Grammar Specification

- CFG specification consists of 3 parts:

User Declarations
%%
ML-YACC Definitions
%%
Rules

- **User Declarlarions:** define various values that are available to *rules* section.
- **ML-YACC Definitions:** declare terminal and non-terminal symbols; declare precedences for terminals that help resolve shift-reduce conflicts.
- **Rules:** specify productions of grammar and *semantic actions* associated with productions.

ML-YACC Declarations

- Need to specify type associated with positions of tokens in input file

```
%pos int
```

- Need to specify terminal and non-terminal symbols (no symbols can be in both lists)

```
%term IF | THEN | ELSE |...
%nonterm prog | stmt | expr |...
```

- Optionally specify end-of-parse symbol - terminals which may follow start symbol

```
%eop EOF
```

- Optionally specify start symbol - otherwise, LHS non-terminal of first rule is taken as start symbol

```
%start prog
```


Attribute Grammar

- ML-YACC employs *attribute grammar* scheme
  - Each terminal or non-terminal symbol may have associated attribute/value.
  - When parser reduces using production $A \rightarrow \alpha$, semantic action associated with production is exectued in order to compute value for A based on the values of symbols in α.
  - Parser returns value associated with start symbol. (If no attribute, `()` is returned.)
- Can specify *types* of atttributes associtated with symbols.

```
%term     ID of string | NUM of int | IF | THEN | ...
%nonterm  prgm | stmt | expr of int | ...
```


Rules

$$symbol_0 \; : \; symbol_1 \; symbol_2 \; ... \; symbol_n \; (\textit{semantic\_action})$$

- Semantic action typically builds piece of AST corresponding to derived string
- Can access attribute/value of RHS symbol X using X<n>, where n specifies a particular occurrence of X on RHS.

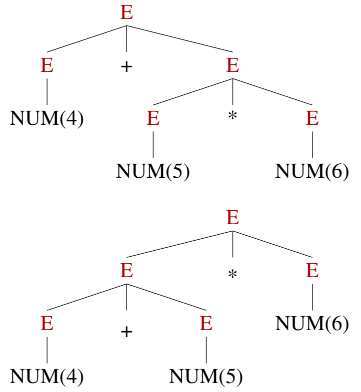
```
%term    PLUS | MINUS | NUM of int | ...
%nonterm exp of int | ...

exp: exp PLUS exp     (exp1 + exp2)
   |  exp MINUS exp   (exp1 - exp2)
   |  NUM             (NUM)
```

- Type of value computed by semantic action must match type of value associated with LHS non-terminal.

ML-YACC and Ambiguous Grammars

- A grammar is ambiguous if it can derive a string of tokens with two or more different parse trees.
- Consider: $4 + 5 * 6$, `NUM(4) PLUS NUM(5) MULT NUM(6)`

$$
\begin{aligned}
expr &\rightarrow ID\\
expr &\rightarrow NUM\\
expr &\rightarrow expr + expr\\
expr &\rightarrow expr * expr
\end{aligned}
$$



- We perfer to bind “*” tighter than “+”.

Directives

Three Solutions:

1. Let YACC complain, but demonstrate that its choice (to shift) was correct.
2. Rewrite grammar to eliminate ambiguity.
3. Keep grammar, but add *precedence directives* which enable conflicts to be resolved. Use `%left, %right, %nonassoc`
   - For this grammar:
     ```
     %left PLUS MINUS
     %left MULT DIV
     ```
   - PLUS, MINUS are left associative, bind equally tightly
   - MULT, DIV are left associative, bind equally tightly
   - MULT, DIV bind tighter than PLUS, MINUS

Example

```
%%

%term    ID | NUM | PLUS | MINUS | MULT | DIV | EOF
%nonterm expr
%pos int
%start expr
%eop EOF
%verbose

%%

expr : ID                ()
     | NUM               ()
     | expr PLUS expr    ()
     | expr MINUS expr   ()
     | expr MULT expr    ()
     | expr DIV expr     ()
```


ML-YACC and Ambiguous Grammars

- Similarly Consider: $4 + 5 + 6$, `NUM(4) PLUS NUM(5) PLUS NUM(6)`
- We perfer to bind left “+” first.
- ML-YACC will report *shift-reduce* conflicts when parsing strings.
  - $4 + 5 * 6$, `NUM(4) PLUS NUM(5) MULT NUM(6)`
    - At some point, E + E will be on top of stack, “*” will be the current token-type in stream.
    - Parser can reduce by rule E → E + E, or shift. Prefer *shift*.
  - $4 + 5 + 6$, `NUM(4) PLUS NUM(5) PLUS NUM(6)`
    - At some point, E + E will be on top of stack, “+” will be the current token-type in stream.
    - Parser can reduce by rule E → E + E, or shift. Prefer *reduce*.

Directives

- Given directives, ML-YACC assigns precedence to each *terminal* and *rule*
  - Precedence of terminal based on order in which associativity specified
  - Precedence of rule is the precedence of right-most terminal. For example, precedence(E $\rightarrow$ E + E) = precedence(PLUS).
- Given shift-reduce conflict, ML-YACC performs the following:
  1. Find precedence of rule to be reduced, terminal to be shifted.
  2. prec(terminal) $>$ prec(rule) $\Rightarrow$ shift.
  3. prec(rule) $>$ prec(terminal) $\Rightarrow$ reduce.
  4. prec(terminal) $=$ prec(rule), then:
     - assoc(terminal) $=$ left $\Rightarrow$ reduce.
     - assoc(terminal) $=$ right $\Rightarrow$ shift.
     - assoc(terminal) $=$ nonassoc $\Rightarrow$ report as error.

Precedence Examples

1

```
 input: 4 + 5 * 6
                |
 stack: 4 + 5
action: prec(*) > prec(+) -> shift
```

2

```
 input: 4 * 5 + 6
                |
 stack: 4 * 5
action: prec(*) > prec(+) -> reduce
```

3

```
 input: 4 + 5 + 6
                |
 stack: 4 + 5
action: assoc(+) = left -> reduce
```


Default Behavior

What if directives not specified?

- shift-reduce: report error, *shift* by default.
- reduce-reduce: report error, reduce by rule that occurs first.

What to do:

- shift-reduce: acceptable in well defined cases (dangling else).
- reduce-reduce: unacceptable. Rewrite grammar.

Direct Rule Precedence Specification

Can assign *specific* precedence to rule, rather than precedence of last terminal.

- Use the %prec directive.
- Commonly used for the *unary minus* problem.

```
%left PLUS MINUS
%left MULT DIV
```

- Consider $-4 * 6$, `MINUS NUM(4) MULT NUM(6)`
- We perfer to bind left unary minus ("-") tighter. Here, precedence of MINUS is lower than MULT, so we get $-(4 * 6)$, not $(-4) * 6$.
- Solution:

```
%left PLUS MINUS
%left MULT DIV
%left UMINUS

exp : MINUS expr %prec UMINUS ()
    | expr PLUS expr ()...
```

Syntax vs. Semantics

Consider language with two classes of expressions

- *Arithmetic* expressions (ae)

```
ae : ae PLUS ae ()
   | ID         ()
```

- *Boolean* expressions (be)

```
be : be AND be ()
   | be OR be  ()
   | be EQ be  ()
   | ID        ()
```

- Consider: `a := b`, ID(a) ASSIGN ID(b):
  - Reduce-reduce conflict - parser can't choose between be $\rightarrow$ ID or ae $\rightarrow$ ID.
  - For now ae and be should be aliased - let semantic analysis (next phase) determine that `a & b + c` is a type error.
  - Type checking cannot be done easily in context free grammars.

Recursive Descent/Predictive/LL(1) Parser Generation

Grammar:

A $\rightarrow$ S EOF
S $\rightarrow$ *id* := E
S $\rightarrow$ *print* (L)

E $\rightarrow$ *id*
E $\rightarrow$ *num*
L $\rightarrow$ E
L $\rightarrow$ L, E

```
fun A() =             (S(); eat(EOF))
and S() = case !tok of
              ID     => (eat(ID); eat(ASSIGN); E())
              PRINT  => (eat(PRINT); eat(LPAREN);
                          L(); eat(RPAREN))
and E() = case !tok of
              ID     => (eat(ID))
              NUM    => (eat(NUM))
and L() = case !tok of
              ID     => (?????)
              NUM    => (?????)
```


Problem

- Based on current function and next token-type in input stream, parser must predict which production to use.
- If !tok = ID, parser cannot determine which production to use:
  L $\rightarrow$ E (E could be ID)
  L $\rightarrow$ L, E (L could be ID)
- Predictive parsing only works for grammars where first terminal symbol of each subexpression provides enough information to choose which production to use.

Formal Techniques

Can use formal techniques to determine whether or not a predictive parser can be built for a particular grammar.

- Let γ be a string of terminal and non-terminal symbols
- Need to compute 3 values:
  1. For each γ corresponding to RHS of production, must determine if γ can derive empty string (ϵ) $\Rightarrow$ **nullable**.
  2. For each γ corresponding to RHS of production, must determine set of all terminal symbols that can begin any string derived from γ $\Rightarrow$ **first**(γ).
  3. For each non-terminal X in grammar, must determine set of all terminal symbols that can immediately follow X in a derivation $\Rightarrow$ **follow**(X).

  Computation of Nullable:
  - γ is nullable if every symbol S $\in \gamma$ is nullable.
  - Check if every S can derive ϵ.

Computation of First

- If T is a terminal symbol, then first(T) = $\{T\}$.
- If X is a non-terminal and $X \rightarrow Y_1 Y_2 Y_3 ... Y_n$, then

  first(Y_1) $\in$ first(X)
  first(Y_2) $\in$ first(X), if Y_1 is nullable
  first(Y_3) $\in$ first(X), if Y_1, Y_2 is nullable
  $\vdots$
  first(Y_n) $\in$ first(X), if Y_1, Y_2, ... Y_{n-1} is nullable

- Let $\gamma = S_1 S_2 ... S_k$. Then,

$$\text{first}(\gamma) = \begin{cases} \text{first}(S_1) \\ \text{first}(S_2), \text{ if } S_1 \text{ is nullable} \\ \text{first}(S_3), \text{ if } S_1, S_2 \text{ is nullable} \\ \vdots \\ \text{first}(S_k), \text{ if } S_1, S_2, ..., S_{k-1} \text{ is nullable} \end{cases}$$

Computation of Follow

Let X, Y be non-terminals; γ, γ_1, and γ_2 be strings of terminals and non-terminals

- if grammar includes production: $X \rightarrow \gamma Y$
  $\Rightarrow$ follow(X) $\in$ follow (Y).
- if grammar includes production: $X \rightarrow \gamma_1 Y \gamma_2$
  $\Rightarrow$ first (γ_2) $\in$ follow (Y)
  $\Rightarrow$ follow (X) $\in$ follow (Y), if γ_2 is nullable.

Perform *iterative* technique in order to compute nullable, first, and follow sets for each non-terminal in grammar.

Building Predictive Parser

$Z \rightarrow XYZ$ $\quad Y \rightarrow$ c $\quad X \rightarrow$ a
$Z \rightarrow$ d $\quad Y \rightarrow \epsilon$ $\quad X \rightarrow$ b Y e

Initial:

	nullable	first	follow
Z	no		
Y	no		
X	no		

Examine each production in grammar, modifying nullable and adding to first and follow sets, until no more changes can be made.

Iteration 1:

	nullable	first	follow
Z	no		
Y	no		
X	no		

Building Predictive Parser

$Z \rightarrow XYZ$ $\quad Y \rightarrow$ c $\quad X \rightarrow$ a
$Z \rightarrow$ d $\quad Y \rightarrow \epsilon$ $\quad X \rightarrow$ b Y e

Iteration 1:

	nullable	first	follow
Z	no		
Y	yes		
X	no		

Iteration 2:

	nullable	first	follow
Z	no		
Y	yes		
X	no		

Iteration 3:

	nullable	first	follow
Z	no	d,a,b	
Y	yes	c	e,d,a,b
X	no	a,b	c,d,a,b

No Changes

Predictive Parsing Table

If the predictive parsing table contains *no* duplicate entries, can build predictive parser for grammar.

- Grammar is LL(1) (left-to-right parse, left-most derivation, 1 symbol lookahead).
- Grammar is LL(k) if its LL(k) predictive parsing table has no duplicate entries.
  - Rows correspond to non-terminals, columns correspond to every possible sequence of k terminals.
  - The first(γ) = set of all k-length terminal sequences that can begin any string derived from γ.
  - LL(k) paring tables can be too large.
  - Ambiguous grammars are not LL(k), $\forall$ k.

Example

$S' \rightarrow S\$$	$S \rightarrow$ IF E THEN A ELSE A	$T \rightarrow$ NUM
$S \rightarrow E$	$E \rightarrow E + T$	$A \rightarrow$ ID = NUM
$S \rightarrow$ IF E THEN A	$E \rightarrow T$	

Iteration 3:

	nullable	first	follow
S'	no	IF	
S	no	IF, NUM	$
E	no	NUM	$, THEN, +
T	no	NUM	$, THEN, +
A	no	ID	$, ELSE

Iteration 4:

	nullable	first	follow
S'	no	IF, NUM	
S	no	IF, NUM	$
E	no	NUM	$, THEN, +
T	no	NUM	$, THEN, +
A	no	ID	$, ELSE

No futher changes

Predictive Parsing Table

	nullable	first	follow
Z	no	d,a,b	
Y	yes	c	e,d,a,b
X	no	a,b	c,d,a,b

Build *predictive parsing table* from nullable, first, and follow sets.

	a	b	c	d	e
Z	$Z \rightarrow XYZ$	$Z \rightarrow XYZ$		$Z \rightarrow d$	
Y	$Y \rightarrow \epsilon$	$Y \rightarrow \epsilon$	$Y \rightarrow c$	$Y \rightarrow \epsilon$	$Y \rightarrow \epsilon$
X	$X \rightarrow a$	$X \rightarrow bYe$			

- Enter $S \rightarrow \gamma$ in row S, column T: for each $T \in$ first(γ).
- If γ is nullable, enter $S \rightarrow \gamma$ in row S, column T: for each $T \in$ follow(S).
- Entry in row S, column T tells parser which clause to execute if current function is $S()$ and next token-type is T
- Blank entries are syntax errors.

Example

$S' \rightarrow S\$$	$S \rightarrow$ IF E THEN A ELSE A	$T \rightarrow$ NUM
$S \rightarrow E$	$E \rightarrow E + T$	$A \rightarrow$ ID = NUM
$S \rightarrow$ IF E THEN A	$E \rightarrow T$	

Iteration 1:

	nullable	first	follow
S'	no		
S	no	IF	$
E	no		$, THEN, +
T	no	NUM	$, THEN, +
A	no	ID	$, ELSE

Iteration 2:

	nullable	first	follow
S'	no	IF	
S	no	IF	$
E	no	NUM	$, THEN, +
T	no	NUM	$, THEN, +
A	no	ID	$, ELSE

Problems

1. $E \rightarrow E + T$
   $E \rightarrow T$

- first(E+T) = first(T)
- When in function `E()`, if next token is NUM, parser will get stuck.
- Grammar is *left-recursive* - left-recursive grammars cannot be LL(1).
- Solution: rewrite grammar so that it is *right-recursive*.
  $E \rightarrow TE'$
  $E' \rightarrow \epsilon$
  $E' \rightarrow \ +TE'$
- In general, $\begin{array}{l} X \rightarrow X\gamma \\ X \rightarrow \alpha \end{array}$ derives strings of form $\alpha\gamma*$ (α doesn't start with X).
  These two productions can be rewritten as follows:
  $X \rightarrow \alpha X'$
  $X' \rightarrow \epsilon$
  $X' \rightarrow \gamma X'$

Example

Show that modified grammar is LL(1).

$S' \rightarrow S\$$
$S \rightarrow E$
$S \rightarrow$ IF E THEN $A\ V$
$V \rightarrow \epsilon$
$V \rightarrow$ ELSE A
$E \rightarrow TE'$
$E' \rightarrow \epsilon$
$E' \rightarrow \ +TE$
$T \rightarrow$ NUM
$A \rightarrow$ ID = NUM

Predictive Parsing Table

	nullable	first	follow
S'	no	IF, NUM	
S	no	IF, NUM	$
E	no	NUM	$, THEN, +
T	no	NUM	$, THEN, +
A	no	ID	$, ELSE

Build *predictive parsing table* from nullable, first, and follow sets.

	IF	THEN	ELSE	+	NUM	ID	=	$
S'	$S' \rightarrow S$				$S' \rightarrow S$			
S	$S \rightarrow$ IF E THEN A $S \rightarrow$ IF E THEN A ELSE A				$S \rightarrow E$			
E					$E \rightarrow E + T$ $E \rightarrow T$			
T					$T \rightarrow$ NUM			
A						$A \rightarrow$ ID = NUM		

Table has duplicate entries $\Rightarrow$ grammar is not LL(1)!

Problems

2. $S \rightarrow$ IF E THEN A
   $S \rightarrow$ IF E THEN A ELSE A

- Two productions begin with same symbol.
- first(IF E THEN A) = first(IF E THEN A ELSE A)
- Solution: use *left-factoring*
  $S \rightarrow$ IF E THEN $A\ V$
  $V \rightarrow \epsilon$
  $V \rightarrow$ ELSE A

Example

Show that modified grammar is LL(1). Build predictive parsing table.

	nullable	first	follow
S'	no	IF,NUM	
S	no	IF,NUM	\$
V	yes	ELSE	\$
E	no	NUM	\$, THEN
E'	yes	+	\$, THEN
T	no	NUM	\$, THEN, +
A	no	ID	\$, ELSE

	IF	THEN	ELSE	+	NUM	ID	=	\$
S'	$S' \rightarrow S$				$S' \rightarrow S$			
S	$S \rightarrow$ IF E THEN A V				$S \rightarrow E$			
V			$V \rightarrow$ ELSE A					$V \rightarrow \epsilon$
E					$E \rightarrow TE'$			
E'		$E' \rightarrow \epsilon$		$E' \rightarrow +TE'$				$E' \rightarrow \epsilon$
T					$T \rightarrow$ NUM			
A						$A \rightarrow$ ID = NUM		

Table does not have duplicate entries $\Rightarrow$ modified grammar is LL(1)!

Example

Show that the grammar is LL(1).