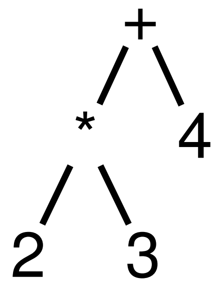
Parsing

Parsing

Objective: build an abstract syntax tree (AST) for the token sequence from the scanner.

`2 * 3 + 4` $\Rightarrow$



Goal: discard irrelevant information to make it easier for the next stage.

Parentheses and most other forms of punctuation removed.

Grammars

Most programming languages described using a *context-free grammar*.

Compared to regular languages, context-free languages add one important thing: recursion.

Recursion allows you to count, e.g., to match pairs of nested parentheses.

Which languages do humans speak? I'd say it's regular: I do not not not not not not not not not not understand this sentence.

Languages

Regular languages (t is a terminal):

$$A \to t_1 \ldots t_n B$$

$$A \to t_1 \ldots t_n$$

Context-free languages (P is terminal or a variable):

$$A \to P_1 \ldots P_n$$

Context-sensitive languages:

$$\alpha_1 A \alpha_2 \to \alpha_1 B \alpha_2$$

“$B \to A$ only in the ‘context’ of $\alpha_1 \cdots \alpha_2$”

Issues

Ambiguous grammars

Precedence of operators

Left- versus right-recursive

Top-down vs. bottom-up parsers

Parse Tree vs. Abstract Syntax Tree

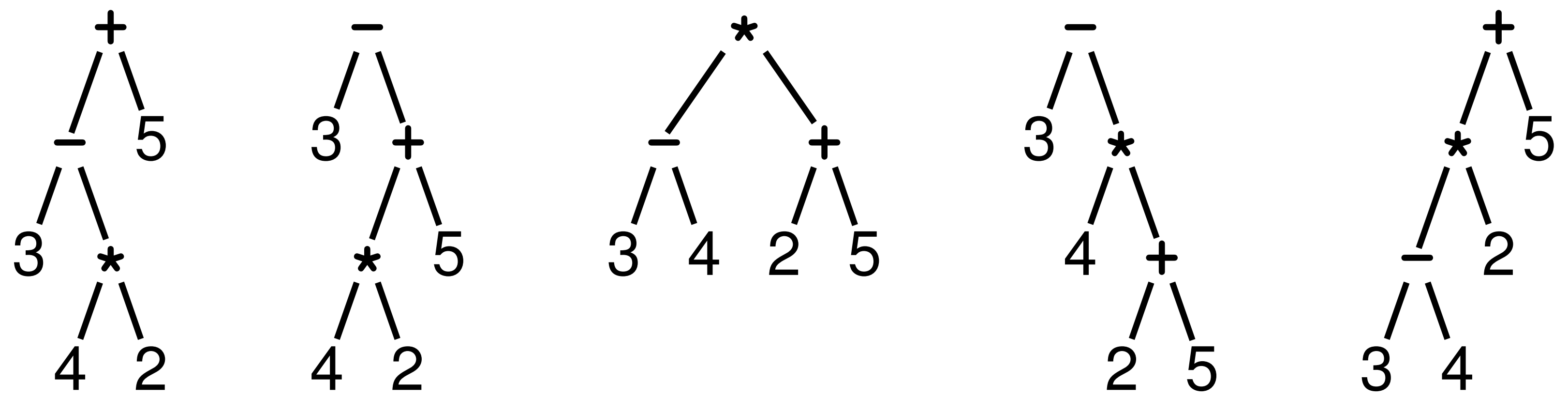
Ambiguous Grammars

A grammar can easily be ambiguous. Consider parsing

3 - 4 * 2 + 5

with the grammar

$$e \to e + e \mid e - e \mid e * e \mid e \,/\, e \mid N$$



Operator Precedence and Associativity

Usually resolve ambiguity in arithmetic expressions

Like you were taught in elementary school:

“My Dear Aunt Sally”

Mnemonic for multiplication and division before addition and subtraction.

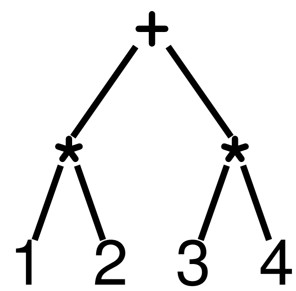
Operator Precedence

Defines how “sticky” an operator is.

`1 * 2 + 3 * 4`

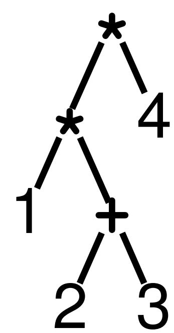
`*` at higher precedence than `+`:

`(1 * 2) + (3 * 4)`



`+` at higher precedence than `*`:

`1 * (2 + 3) * 4`



Associativity

Whether to evaluate left-to-right or right-to-left

Most operators are left-associative

`1 - 2 - 3 - 4`

```
        -                     -
       / \                   / \
      -   4                 1   -
     / \                       / \
    -   3                     2   -
   / \                           / \
  1   2                         3   4
```

((1 - 2) - 3) - 4 | 1 - (2 - (3 - 4))

left associative | right associative

Fixing Ambiguous Grammars

Original ANTLR grammar specification

```
expr
   : expr '+' expr
   | expr '-' expr
   | expr '*' expr
   | expr '/' expr
   | NUMBER
   ;
```

Ambiguous: no precedence or associativity.

Assigning Precedence Levels

Split into multiple rules, one per level

```
expr : expr '+' expr
     | expr '-' expr
     | term ;

term : term '*' term
     | term '/' term
     | atom ;

atom : NUMBER ;
```

Still ambiguous: associativity not defined

Assigning Associativity

Make one side or the other the next level of precedence

```
expr : expr '+' term
     | expr '-' term
     | term
     ;

term : term '*' atom
     | term '/' atom
     | atom
     ;

atom : NUMBER ;
```

Parsing Context-Free Grammars

There are $O(n^3)$ algorithms for parsing arbitrary CFGs, but most compilers demand $O(n)$ algorithms.

Fortunately, the LL and LR subclasses of CFGs have $O(n)$ parsing algorithms. People use these in practice.

The CYK algorithm (Cocke-Younger-Kasami)

Inputs: a string $w = a_1 a_2 \ldots a_n$ and a grammar in Chomsky Normal Form (only productions $X \to a$ and $X \to YZ$)

Construct an $n \times n$ table T, where $T[i, j]$ is the set of nonterminals that generate the substring $a_i a_{i+1} \ldots a_j$. Using dynamic programming, each table entry can be filled in $O(n)$ time. The algorithm runs in $O(n^3)$ time.

Question: how to determine that $a_1 a_2 \ldots a_n$ is in the language?

Parsing LL(k) Grammars

LL: Left-to-right, Left-most derivation

k: number of tokens to look ahead. Usually k = 1

Parsed by top-down, predictive, recursive parsers

Basic idea: look at the next token to predict which production to use

ANTLR builds recursive LL(k) parsers

Almost a direct translation from the grammar.

Implementing a Top-Down Parser

```
stmt  :  IF expr THEN expr
      |  WHILE expr DO expr
      |  expr ASSIG expr ;

expr  :  NUMBER | LPAR expr RPAR ;
```

```
stmt() {
 switch (token) {
 case IF:
     Match(IF); expr(); Match(THEN); expr();             break;
 case WHILE:
     Match(WHILE); expr(); Match(DO); expr();            break;
 case NUMBER or LPAREN:
     expr(); Match(ASSIG); expr();                       break;
 }
}
```

Writing LL(k) Grammars

Cannot have left-recursion

```
expr : expr '+' term | term ;
```

becomes

```
expr() {
  switch (token) {
  case NUMBER or LPAREN: expr(); Match('+'); term();
                  /* Infinite Recursion in expr() */
  ...
  }
}
```

Writing LL(1) Grammars

Cannot have common prefixes

```
expr : ID LPAR expr RPAR
     | ID EQUALS expr
     ;
```

becomes

```
expr() {
  switch (token) {
  case ID:
    Match(ID); Match(LPAR); expr(); Match(RPAR);  break;
  case ID:
    Match(ID); Match(EQUALS); expr();             break;
  }
}
```

Eliminating Common Prefixes

Consolidate common prefixes:

```
expr
  : expr '+' term
  | expr '-' term
  | term
  ;
```

becomes

```
expr
  : expr ('+' term | '-' term )
  | term
  ;
```

Eliminating Left Recursion

Understand the recursion and add tail rules

```
expr
  : expr ('+' term | '-' term )
  | term
  ;
```

becomes

```
expr : term exprt ;
exprt : '+' term exprt
      | '-' term exprt
      | /* nothing */
      ;
```

Using ANTLR's EBNF

ANTLR makes this easier since it supports **`*`**, **`+`** and **`?`**:

```
expr : expr '+' term
     | expr '-' term
     | term
     ;
```

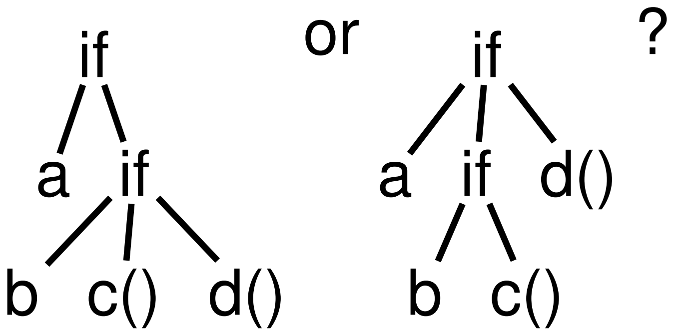
becomes

```
expr : term ('+' term | '-' term)* ;
```

The Dangling Else Problem

Who owns the *else*?

```
if (a) if (b) c(); else d();
```



Grammars are usually ambiguous; manuals give disambiguating rules such as C’s:

> As usual the “else” is resolved by connecting an else with the last encountered elseless if.

The Dangling Else Problem

```
stmt : IF expr THEN stmt iftail
     | other-statements
     ;

iftail
    : ELSE stmt
    | /* nothing */
    ;
```

Problem comes when matching **`iftail`**.

Normally, an empty choice is taken if the next token is in the “follow set” of the rule. But since **`ELSE`** can follow an **`iftail`**, the decision is ambiguous.

The Dangling Else Problem

ANTLR can resolve this problem by making certain rules “greedy.” If a conditional is marked as greedy, it will take that option even if the “nothing” option would also match:

```
stmt
  : IF expr THEN stmt
     ( options {greedy = true;}
     : ELSE stmt
     )?
  | other-statements
  ;
```

The Dangling Else Problem

Some languages resolve this problem by insisting on nesting everything.

E.g., Algol 68:

```
if a < b then a else b fi;
```

“fi” is “if” spelled backwards. The language also uses do–od and case–esac.

Statement separators/terminators

C uses `;` as a statement terminator.

```
if (a<b) printf("a less");
else {
  printf("b"); printf(" less");
}
```

Pascal uses `;` as a statement separator.

```
if a < b then writeln('a less')
else begin
  write('a'); writeln(' less')
end
```

Pascal later made a final `;` optional.

Table-driven Top-Down Parsing

Nomenclature

$a, b, c, \ldots$ represent terminal symbols

$A, B, C, \ldots$ represent nonterminal symbols

S represents the initial nonterminal symbol

$X, Y, Z, \ldots$ represent terminal or non-terminal symbols

$u, v, w, \ldots$ represent words of terminal symbols

$\alpha, \beta, \gamma, \ldots$ represent words of terminal and non-terminal symbols

$\Rightarrow_G$ represents a one-step derivation with grammar G

$\stackrel{*}{\Rightarrow}_G$ represents zero or more steps in a derivation

Nullable, first and follow

$$\text{NULLABLE}(\alpha) \Leftrightarrow$$

$$\exists\ \alpha \overset{*}{\Rightarrow} \epsilon$$

$$\text{FIRST}(\alpha) \equiv$$

$$\{a : \alpha \overset{*}{\Rightarrow} a\beta\}$$

$$\text{FOLLOW}(A) \equiv$$

$$\{a : S\,\$ \overset{*}{\Rightarrow} \alpha A a \beta\}$$

Calculating nullable

NULLABLE(X):

For all terminals a: NULLABLE$(a) = $ FALSE;

For all nonterminals A: NULLABLE$(A) = $ FALSE;

For all productions $A \to \epsilon$: NULLABLE$(A) = $ TRUE;

repeat

For all productions $A \to X_1 X_2 \ldots X_k$

if $\forall i$ in $1..k$: NULLABLE(X_i) **then**

NULLABLE(A) = TRUE

until no further progress

Note: NULLABLE$(X_1 X_2 \ldots X_k) \Leftrightarrow \forall i$ in $1..k$: NULLABLE(X_i)

Calculating first and follow

FIRST(X):

For all terminals a: FIRST$(a) = \{a\}$;

For all nonterminals A: FIRST$(A) = \emptyset$;

repeat

For all productions $A \to X_1 X_2 \ldots X_k$ ⟨outer loop⟩

For i in $1 \ldots k$ ⟨inner loop⟩

add FIRST(X_i) to FIRST(A);

if not NULLABLE(X_i) **continue outer loop**;

until no further progress

Note: FIRST$(X_1 X_2 \ldots X_k)$ is calculated in a similar way as the inner loop.

Calculating first and follow

FOLLOW(A):

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

For all other nonterminal symbols A, FOLLOW$(A) = \emptyset$

`repeat`

For all productions $A \rightarrow \alpha B \beta$

add FIRST(β) to FOLLOW(B)

For all productions $A \rightarrow \alpha B$

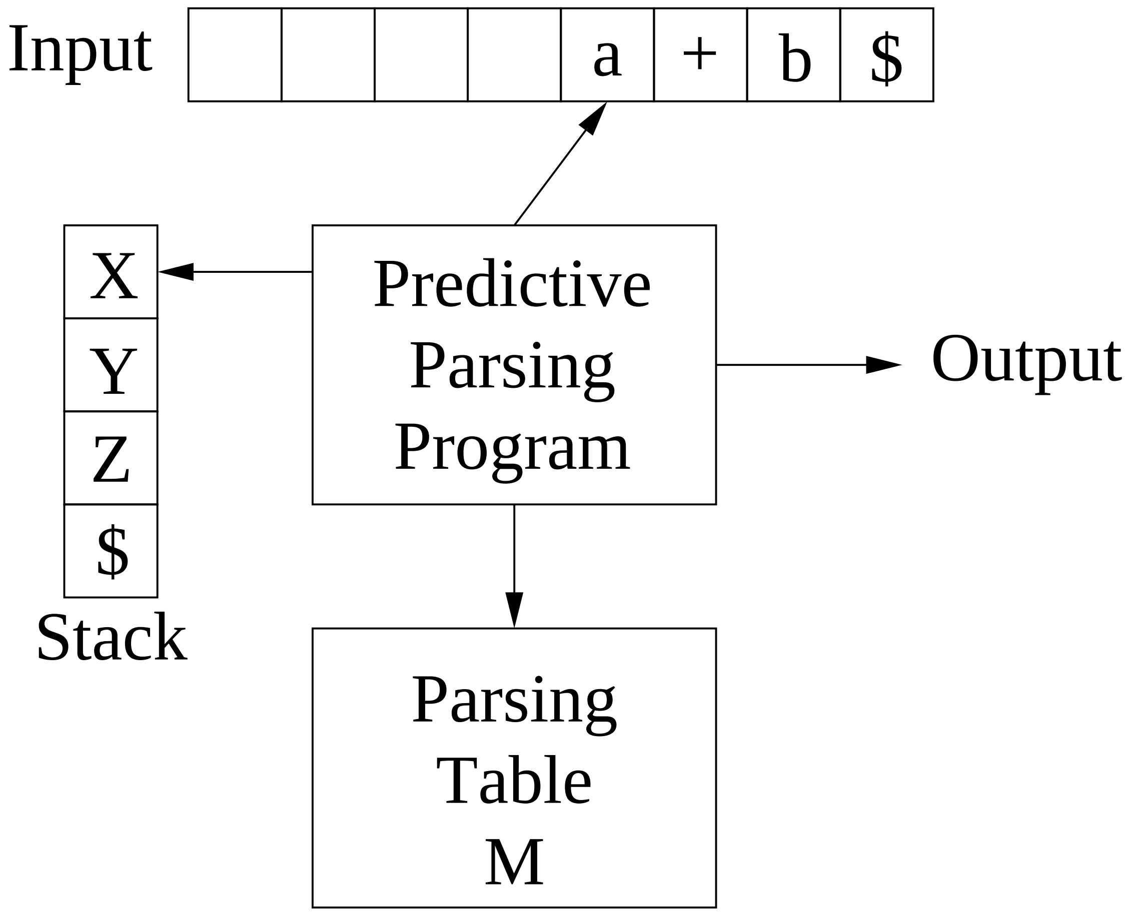
or $A \rightarrow \alpha B \beta$, where NULLABLE$(\beta)$

add FOLLOW(A) to FOLLOW(B)

`until` no further progress

Note: FIRST(β) is calculated in a similar way as the inner loop of FIRST.

Table-driven predictive parser



Parsing table

For each production $A \to \alpha$ of the grammar do:

1. For each terminal $a \in \textsc{First}(\alpha)$, add $A \to \alpha$ to $M[A, a]$.

2. If $\textsc{Nullable}(\alpha)$, then for each terminal b in $\textsc{Follow}(A)$, add $A \to \alpha$ to $M[A, b]$.

All empty entries must be set to *error*.

Parsing table

PRODUCTION $A \to \alpha$	NULLABLE(α)	FIRST(α)	FOLLOW(A)
$E \to TE'$	FALSE	$\{(, \textbf{id}\}$	$\{), \$\}$
$E' \to +TE'$	FALSE	$\{+\}$	$\{), \$\}$
$E' \to \varepsilon$	TRUE	$\emptyset$	
$T \to FT'$	FALSE	$\{(, \textbf{id}\}$	$\{+,), \$\}$
$T' \to *FT'$	FALSE	$\{*\}$	$\{+,), \$\}$
$T' \to \varepsilon$	TRUE	$\emptyset$	
$F \to (E)$	FALSE	$\{(\}$	$\{+, *,), \$\}$
$F \to \textbf{id}$	FALSE	$\{\textbf{id}\}$	

	id	$+$	$*$	(	), \$
E	$E \to TE'$			$E \to TE'$	
E'		$E' \to +TE'$			$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$	
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$
F	$F \to \textbf{id}$			$F \to (E)$	

LL(1) grammars

A grammar is LL(1) if and only if for every pair of rules $A \to \alpha$ and $A \to \beta$:

- $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$.
- $\text{NULLABLE}(\alpha) \wedge \text{NULLABLE}(\beta) = \text{FALSE}$.
- If $\text{NULLABLE}(\alpha)$, then $\text{FIRST}(\beta) \cap \text{FOLLOW}(A) = \emptyset$.

If a grammar is LL(1), then no conflicts appear in the parsing table.

Predicting parsing algorithm

ip is the pointer to the input (initially pointing at the first token);
The stack initially contains $S\$$, with S on the top;
Set X to the top of the stack symbol;

while $(X \neq \$)$ **do**
  a = INPUT[*ip*];
  if $(X = a)$ Pop the stack and Advance *ip*;
  else if (X is a terminal $\neq a$) **SYNTAXERROR()**;
  else if ($M[X, a] =$ *error*) **SYNTAXERROR()**;
  else if $(M[X, a] = X \to Y_1 Y_2 \cdots Y_k)$
    Output the production $X \to Y_1 Y_2 \cdots Y_k$;
    Pop the stack;
    Push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with Y_1 on top;
  Set X to the top of the stack symbol;

Exercise: Execute the parsing algorithm with the input $z * (x + y) \$$.

Generation of top-down parsers (ANTLR style)

A simple example

Grammar

```
instruction_list: ( instruction )* ;

instruction: IDENT ASSIG expr
           | IF expr THEN instruction_list ;

expr: (IDENT | NUM) (PLUS (IDENT | NUM))* ;
```

Parser

```
void instruction_list() {
  while (Token==IDENT or Token==IF) {
    instruction();
  }
}
```

A simple example

```
instruction: IDENT ASSIG expr |
             IF expr THEN instruction_list ;
```

```
void instruction() {
  if (Token==IDENT) {
    Token = nexttoken();
    if (Token== ASSIG) {
      Token = nexttoken(); expr();
    } else SYNTAXERROR();
  } else if (Token==IF) {
    Token = nexttoken(); expr();
    if (Token==THEN) {
      Token = nextoken(); instruction_list();
    } else SYNTAXERROR();
  } else SYNTAXERROR();
}
```

A simple example

```
expr: (IDENT|NUM) (PLUS (IDENT|NUM))* ;
```

```
void expr() {
  if (Token==IDENT or Token==NUM) {
    Token = nexttoken();
    while (Token==PLUS) {
      Token = nexttoken();
      if (Token==IDENT or Token==NUM){
        Token = nexttoken();
      } else SYNTAXERROR();
    }
  } else SYNTAXERROR();
}
```

ANTLR

- ANTLR uses the EBFN notation
- NULLABLE, FIRST and FOLLOW must be extended to the EBNF notation.
- A recursive top-down parser is generated.

Nullable

- $\textsc{Nullable}(\varepsilon) = \textit{true}$.
- $\textsc{Nullable}(E*) = \textit{true}$.
- If $\textsc{Nullable}(E)$, then $\textsc{Nullable}(E+) = \textit{true}$.
- If $V \rightarrow E \in G$ and $\textsc{Nullable}(E)$, then $\textsc{Nullable}(V) = \textit{true}$.
- If $\textsc{Nullable}(E_1)$ and $\textsc{Nullable}(E_2)$, then $\textsc{Nullable}(E_1 E_2) = \textit{true}$.
- If $\textsc{Nullable}(E_1)$ or $\textsc{Nullable}(E_2)$, then $\textsc{Nullable}(E_1 | E_2) = \textit{true}$.
- Nothing else is $\textsc{Nullable}$.

First

- If c is a terminal, $\text{FIRST}(c) = \{c\}$.
- If $V \to E \in G$, then $\text{FIRST}(V)$ contains $\text{FIRST}(E)$.
- $\text{FIRST}(E_1 E_2)$ contains $\text{FIRST}(E_1)$.
- If $\text{NULLABLE}(E_1)$, then $\text{FIRST}(E_1 E_2)$ contains $\text{FIRST}(E_2)$.
- $\text{FIRST}(E_1 | E_2)$ contains $\text{FIRST}(E_1)$ and $\text{FIRST}(E_2)$.
- $\text{FIRST}(E*)$ and $\text{FIRST}(E+)$ contain $\text{FIRST}(E)$.
- Nothing else belongs to FIRST.

Follow

- If $E_1 E_2$ is and expression of the grammar, then
  - $\textsc{Follow}(E_1)$ contains $\textsc{First}(E_2)$.
  - $\textsc{Follow}(E_2)$ contains $\textsc{Follow}(E_1 E_2)$.
  - if $\textsc{Nullable}(E_2)$, then $\textsc{Follow}(E_1)$ contains $\textsc{Follow}(E_1 E_2)$.
- If $E_1 | E_2$ is and expression of the grammar, then $\textsc{Follow}(E_1)$ and $\textsc{Follow}(E_2)$ contain $\textsc{Follow}(E_1 | E_2)$.
- If $E*$ is an expression of the grammar, then $\textsc{Follow}(E)$ contains $\textsc{Follow}(E*) \cup \textsc{First}(E)$.
- The previous rule also applies for $E+$.
- If $V \to E \in G$, then $\textsc{Follow}(E)$ contains $\textsc{Follow}(V)$.
- Nothing else belongs to FOLLOW.

Generating an LL(1) recursive-descent predictive parser

For every rule $A \rightarrow E$, a function is generated:

void A() $\{$
    Parse$(E,$ ***Follow***$(A))$
$\}$

where **Parse**(E, F) is the code generated to recognize the EBNF expression E followed by the tokens in F.

Token is a variable that represents the current token.

Match(T**)** $\equiv$
    if (Token == T**) Token = nexttoken();**
    else SYNTAXERROR();

Parse(E, F)

```
Parse(E₁ | E₂ | ... | Eₙ, F) ≡
    if (Token ∈ First(E₁)) Parse(E₁,F);
    else if (Token ∈ First(E₂)) Parse(E₂,F);
    ...
    else if (Token ∈ First(Eₙ)) Parse(Eₙ,F);
    else if (no Eᵢ is nullable) SYNTAXERROR();
    else if (Token ∉ F) SYNTAXERROR();
```

If the BNF version of the grammar is LL(1) then there are no conflicts between the different branches (including the case of a nullable E_i).

Parse(E, F)

Parse($E_1 E_2 \ldots E_n$**,** F**)** $\equiv$

  Parse(E_1**,** ***First***$(E_2 \ldots E_n \cdot F)$**);**

  Parse(E_2**,** ***First***$(E_3 \ldots E_n \cdot F)$**);**

  $\ldots$

  Parse(E_n**,** F**);**

where ***First***$(E \cdot F)$ **is**

- ***First***(E) **if** E **is not nullable**
- ***First***$(E) \cup F$**, otherwise**

Parse(E, F)

Parse($E*$**,** F**)** $\equiv$

  while (Token $\in$ ***First***(E)**) Parse(**E**,**F**);**

Parse($E+$**,**F**)** $\equiv$

  do Parse(E**,** F**); while (Token** $\in$ ***First***(E)**);**

Parse($E?$**,** F**)** $\equiv$

  if (Token $\in$ ***First***(E)**) Parse(**E**,**F**);**

Parse(E, F)

Parse((E)**,** F**)** $\equiv$ **Parse(**E**,**F**);**

Parse(ϵ**,** F**)** $\equiv$ **;**

Parse(a**,**F**)** $\equiv$ **Match(**a**);** // a is a terminal symbol

Parse(A**,**F**)** $\equiv$ A**();** // A is a non-terminal symbol

LL(1) example

Grammar:

$$E \rightarrow T \; (\,\texttt{'+'}\; T \mid \texttt{'-'}\; T\,)*$$

$$T \rightarrow F \; (\,\texttt{'*'}\; F \mid \texttt{'/'}\; F\,)*$$

$$F \rightarrow \texttt{ID} \mid \texttt{NUM} \mid \texttt{'('}\; E \;\texttt{')'}$$

```
void E() {
    T();
    while (Token == '+' or Token == '-') {
        if (Token == '+') { Match('+'); T(); }
        else if (Token == '-') { Match('-'); T(); }
        else SYNTAXERROR();        // redundant
    }
}
```

LL(1) example

```
void T() {
    F();
    while (Token == '*' or Token == '/') {
        if (Token == '*') { Match('*'); F(); }
        else if (Token == '/') { Match('/'); F(); }
        else SYNTAXERROR();        // redundant
    }
}

void F() {
    if (Token == ID) Match(ID);
    else if (Token == NUM) Match(NUM);
    else if (Token == '(') {
        Match('('); E(); Match(')');
    } else SYNTAXERROR();
}
```

Bottom-up Parsing

Rightmost Derivation

$$\begin{array}{rrcl} 1: & e & \rightarrow & t + e \\ 2: & e & \rightarrow & t \\ 3: & t & \rightarrow & \textbf{id} * t \\ 4: & t & \rightarrow & \textbf{id} \end{array}$$

A rightmost derivation for $\textbf{id} * \textbf{id} + \textbf{id}$:

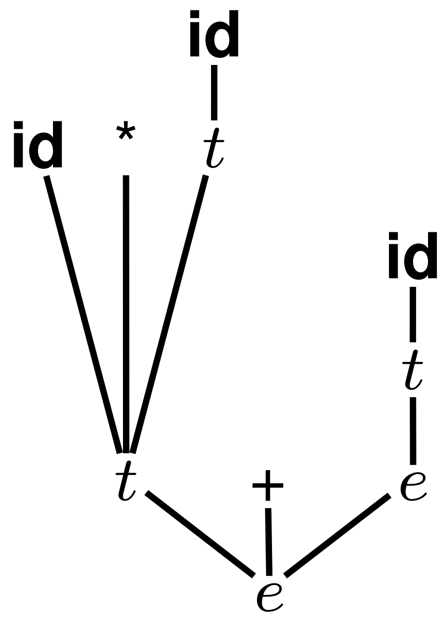
$$\begin{array}{c} \boxed{e} \\ t + \boxed{e} \\ t + \boxed{t} \\ \boxed{t} + \textbf{id} \\ \textbf{id} * \boxed{t} + \textbf{id} \\ \textbf{id} * \textbf{id} + \textbf{id} \end{array}$$

Basic idea of bottom-up parsing: construct this rightmost derivation backward.

Handles

$$
\begin{array}{rlcl}
1: & e & \rightarrow & t + e \\
2: & e & \rightarrow & t \\
3: & t & \rightarrow & \textbf{id} * t \\
4: & t & \rightarrow & \textbf{id}
\end{array}
$$

$$
\begin{array}{l}
\textbf{id} * \boxed{\textbf{id}} + \textbf{id} \\
\boxed{\textbf{id} * t} + \textbf{id} \\
t + \boxed{\textbf{id}} \\
t + \boxed{t} \\
\boxed{t + e} \\
e
\end{array}
$$



This is a reverse rightmost derivation for **id** $*$ **id** $+$ **id**.

Each highlighted section is a handle handle.

Taken in order, the handles build the tree from the leaves to the root.

Shift-reduce Parsing

$$
\begin{aligned}
1: &\quad e \rightarrow t + e \\
2: &\quad e \rightarrow t \\
3: &\quad t \rightarrow \mathbf{id} * t \\
4: &\quad t \rightarrow \mathbf{id}
\end{aligned}
$$

stack	input	action
	$\mathbf{id} * \mathbf{id} + \mathbf{id}$	shift
$\mathbf{id}$	$* \mathbf{id} + \mathbf{id}$	shift
$\mathbf{id}*$	$\mathbf{id} + \mathbf{id}$	shift
$\mathbf{id} * \boxed{\mathbf{id}}$	$+ \mathbf{id}$	reduce (4)
$\boxed{\mathbf{id} * t}$	$+ \mathbf{id}$	reduce (3)
t	$+ \mathbf{id}$	shift
$t+$	$\mathbf{id}$	shift
$t + \boxed{\mathbf{id}}$		reduce (4)
$t + \boxed{t}$		reduce (2)
$\boxed{t + e}$		reduce (1)
e		accept

Scan input left-to-right, looking for handles.

An oracle tells what to do

LR Parsing

$1: \quad e \rightarrow t + e$

$2: \quad e \rightarrow t$

$3: \quad t \rightarrow \textbf{id} * t$

$4: \quad t \rightarrow \textbf{id}$

	action				goto	
	id	$+$	$*$	$\$$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				acc		

stack	input	action
$\boxed{0}$	**id** * **id** + **id** $\$$	shift, goto 1

1. Look at state on top of stack
2. and the next input token
3. to find the next action
4. In this case, shift the token onto the stack and go to state 1.

LR Parsing

$$1: \quad e \rightarrow t + e$$
$$2: \quad e \rightarrow t$$
$$3: \quad t \rightarrow \textbf{id} * t$$
$$4: \quad t \rightarrow \textbf{id}$$

	action				goto	
	id	$+$	$*$	$\$$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				acc		

stack	input	action
$\boxed{0}$	**id** * **id** + **id** $\$$	shift, goto 1
$\boxed{0}\ \boxed{\textbf{id}\ 1}$	* **id** + **id** $\$$	shift, goto 3
$\boxed{0}\ \boxed{\textbf{id}\ 1}\ \boxed{*\ 3}$	**id** + **id** $\$$	shift, goto 1
$\boxed{0}\ \boxed{\textbf{id}\ 1}\ \boxed{*\ 3}\ \boxed{\textbf{id}\ 1}$	+ **id** $\$$	reduce w/ 4

Action is reduce with rule 4 ($t \rightarrow$ **id**). The right side is removed from the stack to reveal state 3. The goto table in state 3 tells us to go to state 5 when we reduce a t:

stack	input	action
$\boxed{0}\ \boxed{\textbf{id}\ 1}\ \boxed{*\ 3}\ \boxed{t\ 5}$	+ **id** $\$$	

LR Parsing

$$1: \quad e \rightarrow t + e$$
$$2: \quad e \rightarrow t$$
$$3: \quad t \rightarrow \mathbf{id} * t$$
$$4: \quad t \rightarrow \mathbf{id}$$

	action				goto	
	id	$+$	$*$	$\$$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				acc		

stack	input	action
[0]	**id** * **id** + **id** $	shift, goto 1
[0] [id 1]	* **id** + **id** $	shift, goto 3
[0] [id 1] [* 3]	**id** + **id** $	shift, goto 1
[0] [id 1] [* 3] [id 1]	+ **id** $	reduce w/ 4
[0] [id 1] [* 3] [t 5]	+ **id** $	reduce w/ 3
[0] [t 2]	+ **id** $	shift, goto 4
[0] [t 2] [+ 4]	**id** $	shift, goto 1
[0] [t 2] [+ 4] [id 1]	$	reduce w/ 4
[0] [t 2] [+ 4] [t 2]	$	reduce w/ 2
[0] [t 2] [+ 4] [e 6]	$	reduce w/ 1
[0] [e 7]	$	accept

Constructing the SLR Parse Table

The states are places we could be in a reverse-rightmost derivation. Let's represent such a place with a dot.

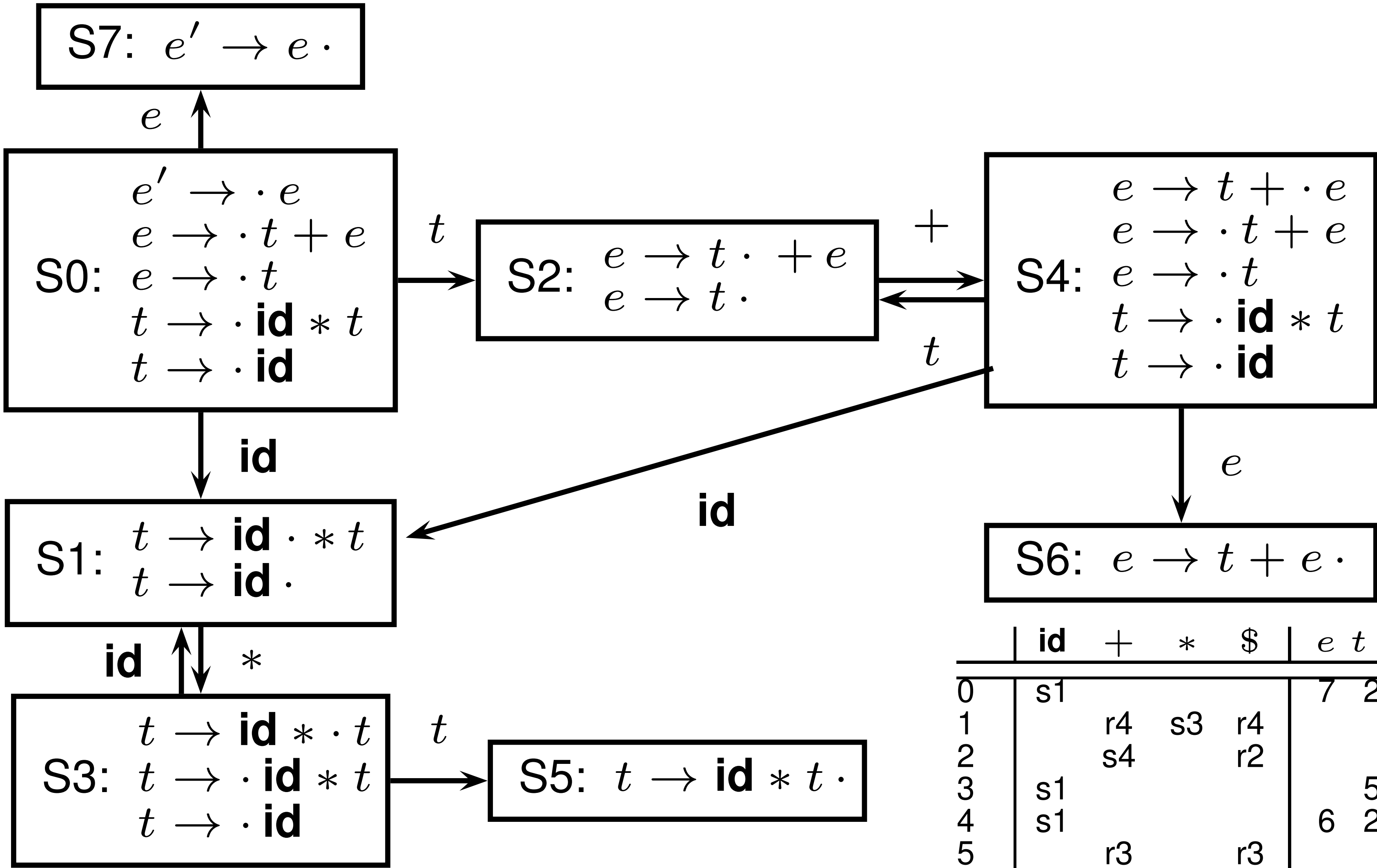
$$
\begin{aligned}
0: \quad e' &\rightarrow e \\
1: \quad e &\rightarrow t + e \\
2: \quad e &\rightarrow t \\
3: \quad t &\rightarrow \textbf{id} * t \\
4: \quad t &\rightarrow \textbf{id}
\end{aligned}
$$

Say we were at the beginning ($\cdot\, e$). This corresponds to

$$
\begin{aligned}
&e' \rightarrow \cdot\, e \\
&e \rightarrow \cdot\, t + e \\
&e \rightarrow \cdot\, t \\
&t \rightarrow \cdot\, \textbf{id} * t \\
&t \rightarrow \cdot\, \textbf{id}
\end{aligned}
$$

The first is a placeholder. The second are the two possibilities when we're just before e. The last two are the two possibilities when we're just before t.

Constructing the SLR Parsing Table



	id	$+$	$*$	\$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				acc		

The Punchline

This is a tricky, but mechanical procedure. The parser generators YACC, Bison, Cup, and others (but not ANTLR) use a modified version of this technique to generate fast bottom-up parsers.

You need to understand it to comprehend error messages:

Shift/reduce conflicts are caused by a state like

$t \to \textbf{id} \cdot * t$

$t \to \textbf{id} \cdot$

Reduce/reduce conflicts are caused by a state like

$t \to \textbf{id} * t \cdot$

$e \to t \cdot$