Compiler Principle

Prof. Dongming LU

Mar. 11th, 2024

Content

3 Parsing

3.2 Predictive Parsing

Recursive-Descent Parser

Each grammar production turns into one clause of a recursive function.

- **Predictive parsing**
  - ✓ **Top-down parsing**
  - ✓ **Simple, efficient**
  - ✓ **Can be coded by hand in C quickly**

An Example I

1. S→**IF** E **THEN** S **ELSE** S
2. | **BEGIN** S L
3. | **PRINT** E
4. L → **END**
5. | **;** S L
6. E → **NUM = NUM**

```
enum token  { IF , THEN , ELSE , BEGIN , END  , PRINT , SEMI , NUM, EQ}
extern enum token  getToken(void);

enum token tok;
void advance( ) { tok = getToken( );}
void eat(enum token t) { if (tok == t ) advance( ); else error(); }
```

An Example I

1. **S→IF E THEN S ELSE S**
2. **| BEGIN S L**
3. **| PRINT E**
4. **L → END**
5. **| ; S L**
6. **E → NUM = NUM**

```
void S( ) {switch(tok) {
        case IF: eat(IF); E(); eat(THEN); S(); eat(ELSE); S(); break;
        case BEGIN: eat(BEGIN); S( ); L( ); break;
        case PRINT: eat(PRINT); E( ); break;
        default: error();
 }}
```

An Example I

1. S→IF E THEN S ELSE S
2. | BEGIN S L
3. | PRINT E

4. L → END
5. | ; S L
6. E → NUM = NUM

```
void L( ) {switch(tok) {
        case END: eat(END); break;
        case SEMI: eat(SEMI); S( ); L(); break;
        default: error();
}}

void E( ) { eat(NUM); eat(EQ); eat(NUM); }
```

An Example II

S → E $
E → E + T
  | E − T
  | T

T → T * F
  | T / F
  | F

F → id
  | num
  |(E)

```
void S() { E(); eat(EOF); }
void E() {switch (tok) {
   case ?: E(); eat(PLUS); T(); break;
   case ?: E(); eat(MINUS); T();
break;
   case ?: T(); break; default: error();
}}
```

?conflict

?predictive

```
void T() {switch (tok) {
   case ?: T(); eat(TIMES); F();
break;
   case ?: T(); eat(DIV); F(); break;
   case ?: F(); break; default: error();
```

Problem

- Predictive parsing only works for grammars where the first terminal symbol of each subexpression provides enough information to choose which production to use

How to derive conflict-free recursive-descent parsers using a simple algorithm

Nullable Sets

- **Non-terminal X is Nullable only if the following constraints are satisfied**

base case:

✓if (X :=   ) then X is Nullable

inductive case:

✓if (X := ABC...) and A, B, C, ... are all Nullable then X is Nullable

Computing Nullable Sets

- **Compute X is Nullable by iteration:**

```
Initialization:
    Nullable := { }
    if (X :=  )  then Nullable := Nullable U {X}

While Nullable different from last iteration do:
    for all X,
        if (X := ABC...) and A, B, C, ... are all Nullable then
        Nullable := Nullable U {X}
```

First Sets

- **First(X) is specified like this:**

base case:

if T is a terminal symbol then First (T) = {T}

inductive case:

if X is a non-terminal and (X:= ABC...) then

First (X) = First (ABC...)

where First(ABC...) = F1 U F2 U F3 U ... and

F1 = First (A)

F2 = First (B), if A is Nullable; emptyset otherwise

F3 = First (C), if A is Nullable & B is Nullable; emp...

...

Computing Follow Sets

- **Follow(X) is computed iteratively**

base case:

✓Initially, assume nothing in particular follows X
(when computing, Follow (X) is initially { })

inductive case:

✓if (Y := s1 X s2) for any strings s1, s2 then
Follow (X) = Follow(X) U First (s2)

✓if (Y := s1 X s2) for any strings s1, s2 then
Follow (X) = Follow(x) U Follow(Y), if s2 is Nullable

Building a Predictive Parser

$Z \rightarrow X\ Y\ Z$	$Y \rightarrow c$	$X \rightarrow a$
$Z \rightarrow d$	$Y \rightarrow$	$X \rightarrow Y$

	nullable	first	follow
Z			
Y			
X			

Building a Predictive Parser

Z → X Y Z
Z → d

Y → c
Y →

X → a
X → Y

	nullable	first	follow
Z	no		
Y	yes		
X	yes		

Building a Predictive Parser

$Z \rightarrow X\ Y\ Z$	$Y \rightarrow c$	$X \rightarrow a$
$Z \rightarrow d$	$Y \rightarrow$	$X \rightarrow Y$

	nullable	first	follow
Z	no	{ }	
Y	yes	{ }	
X	yes	{ }	

Building a Predictive Parser

$Z \to X\ Y\ Z$
$Z \to d$

$Y \to c$
$Y \to$

$X \to a$
$X \to Y$

	nullable	first	follow
Z	no	d	
Y	yes	c	
X	yes	a	

Building a Predictive Parser

$Z \rightarrow X\ Y\ Z$
$Z \rightarrow d$

$Y \rightarrow c$
$Y \rightarrow$

$X \rightarrow a$
$X \rightarrow Y$

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

Building parsing table

Grammar:

Z → X Y Z	Y → c	X → a
Z → d	Y →	X → Y

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

Build parsing table where row X, col T tells parser which clause to execute in function X with next-token T:

- if T ∈ First(s) then
  enter (X → s) in row X, col T
- if s is Nullable and T ∈ Follow(X)
  enter (X → s) in row X, col T

	a	c	d
Z			
Y			
X			

Building parsing table

Grammar:

$Z \to X\ Y\ Z$	$Y \to c$	$X \to a$
$Z \to d$	$Y \to$	$X \to Y$

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

Build parsing table where row X, col T tells parser which clause to execute in function X with next-token T:

- if T ∈ First(s) then
  enter (X → s) in row X, col T
- if s is Nullable and T ∈ Follow(X)
  enter (X → s) in row X, col T

	a	c	d
Z	Z → XYZ	Z → XYZ	Z → XYZ
Y			
X			

Building parsing table

Grammar:

Z → X Y Z	Y → c	X → a
Z → d	Y →	X → Y

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

Build parsing table where row X, col T tells parser which clause to execute in function X with next-token T:

- if T ∈ First(s) then
  enter (X → s) in row X, col T
- if s is Nullable and T ∈ Follow(X)
  enter (X → s) in row X, col T

	a	c	d
Z	Z → XYZ	Z → XYZ	Z → d Z → XYZ
Y			
X			

Building parsing table

Grammar:

$Z \to X\ Y\ Z$	$Y \to c$	$X \to a$
$Z \to d$	$Y \to$	$X \to Y$

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

Build parsing table where row X, col T tells parser which clause to execute in function X with next-token T:

- if $T \in$ First(s) then
  enter ($X \to s$) in row X, col T
- if s is Nullable and $T \in$ Follow(X)
  enter ($X \to s$) in row X, col T

	a	c	d
Z	$Z \to XYZ$	$Z \to XYZ$	$Z \to d$ $Z \to XYZ$
Y		$Y \to c$	
X			

Building parsing table

Grammar:

$Z \to X\,Y\,Z$ $\quad Y \to c$ $\quad X \to a$

$Z \to d$ $\quad Y \to$ $\quad X \to Y$

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

Build parsing table where row X, col T tells parser which clause to execute in function X with next-token T:

- if $T \in$ First(s) then enter ($X \to s$) in row X, col T
- if s is Nullable and $T \in$ Follow(X) enter ($X \to s$) in row X, col T

	a	c	d
Z	$Z \to XYZ$	$Z \to XYZ$	$Z \to d$ $Z \to XYZ$
Y	$Y \to$	$Y \to$ $Y \to c$	$Y \to$
X			

Building parsing table

Grammar:

$Z \to X\ Y\ Z$ $\quad Y \to c \quad X \to a$

$Z \to d \quad Y \to \quad X \to Y$

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

Build parsing table where row X, col T tells parser which clause to execute in function X with next-token T:

- if T $\in$ First(s) then
  enter ($X \to s$) in row X, col T
- if s is Nullable and T $\in$ Follow(X)
  enter ($X \to s$) in row X, col T

	a	c	d
Z	$Z \to XYZ$	$Z \to XYZ$	$Z \to d$ $Z \to XYZ$
Y	$Y \to$	$Y \to$ $Y \to c$	$Y \to$
X	$X \to a$ $X \to Y$	$X \to Y$	$X \to Y$

Building parsing table

Grammar:

$Z \to X\,Y\,Z$	$Y \to c$	$X \to a$
$Z \to d$	$Y \to$	$X \to Y$

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

Is it possible to put 2 grammar rules in the same box?

	a	c	d
Z	$Z \to XYZ$	$Z \to XYZ$	$Z \to d$ $Z \to XYZ$
Y	$Y \to$	$Y \to$ $Y \to c$	$Y \to$
X	$X \to a$ $X \to Y$	$X \to Y$	$X \to Y$

Predictive Parsing: LL(1)

- If a predictive parsing table constructed this way contains no duplicate entries, the grammar is called LL(1)
- if not, of the grammar is not LL(1)

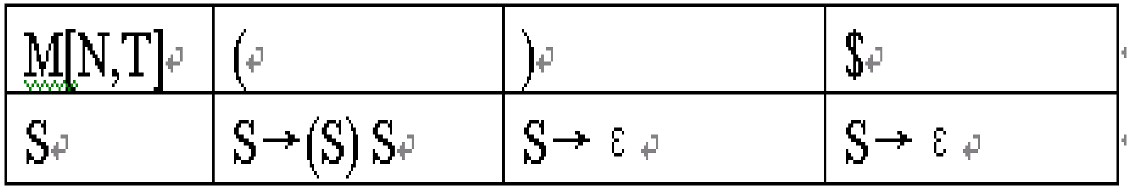
LL(1): Left-to-right parse, Left-most derivation, 1 symbol lookahead

- In LL(k) parsing table, columns include every k-length sequence of terminals:

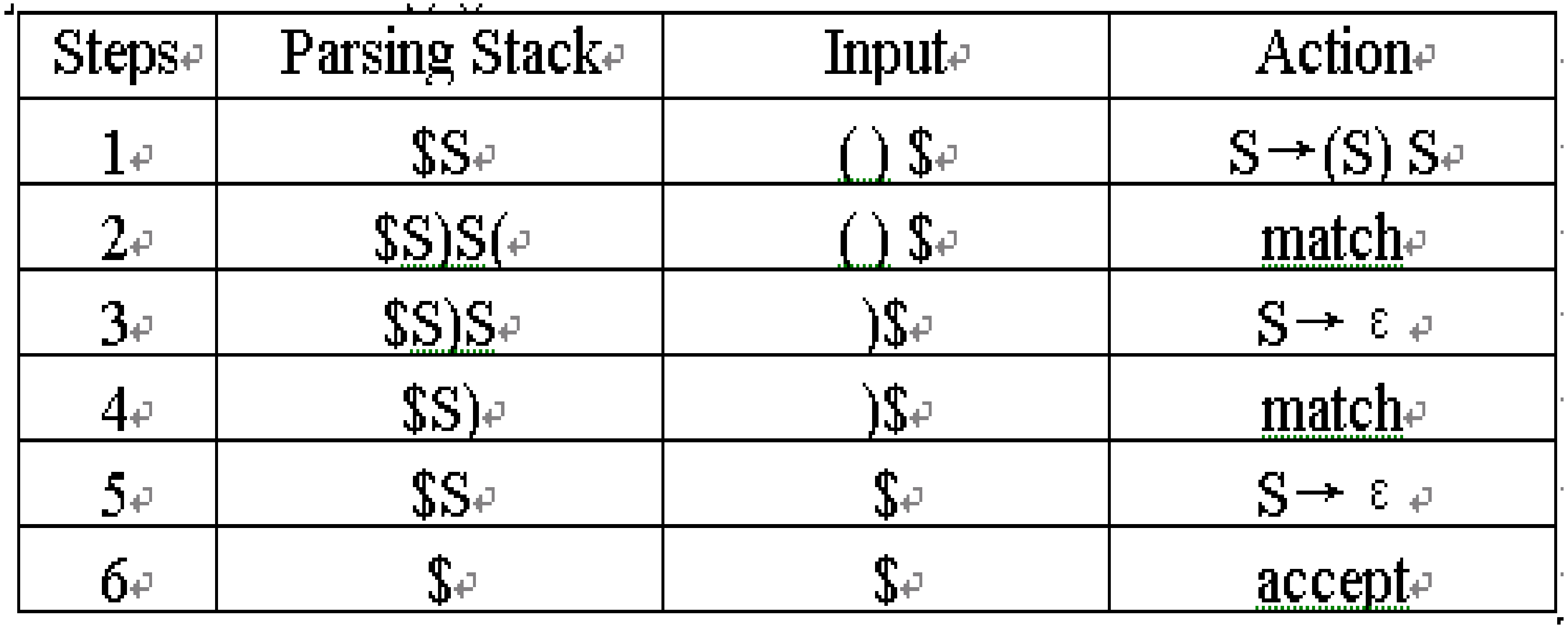
aa	ab	ba	bb	ac	ca	...

PREDICTIVE PARSING:LL(1)

- $S \rightarrow (S)\ S \mid \varepsilon$



M[N,T]	(	)	$
S	$S \rightarrow (S)\ S$	$S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$



Steps	Parsing Stack	Input	Action
1	$S	( ) $	$S \rightarrow (S)\ S$
2	$S)S(	( ) $	match
3	$S)S	)$	$S \rightarrow \varepsilon$
4	$S)	)$	match
5	$S	$	$S \rightarrow \varepsilon$
6	$	$	accept

Eliminate left-recursion

- **Rewrite the grammar so it parses the same language but the rules are different:**

S $\rightarrow$ A
A $\rightarrow$ ID := E
| PRINT (L)

E $\rightarrow$ ID
| NUM

L $\rightarrow$ L, E
| E

S $\rightarrow$ A
A $\rightarrow$ ID := E
| PRINT (L)

E $\rightarrow$ ID
| NUM

L $\rightarrow$ E M

M $\rightarrow$, E M
| ε

Eliminate left-recursion

$E \rightarrow E + T$
$E \rightarrow T$

$\Longrightarrow$

$E \rightarrow T E'$
$E' \rightarrow +T E' | \varepsilon$

$A \rightarrow A \alpha | \beta$

$\Longrightarrow$

$A \rightarrow \beta A'$ **(To generate β first)   )**
$A' \rightarrow \alpha A' | \varepsilon$

(To generate the repetitions of α, using right recursion.)

Eliminate left-recursion

- An Example

$S \rightarrow E\ \$$
$E \rightarrow E + T$
$E \rightarrow E - T$
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T / F$

$T \rightarrow F$

$F \rightarrow$ id
$F \rightarrow$ num
$F \rightarrow (E)$

Eliminate left-recursion

$S \rightarrow E\ \$$

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E'$

$E' \rightarrow -\ T\ E'$

$E' \rightarrow$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T'$

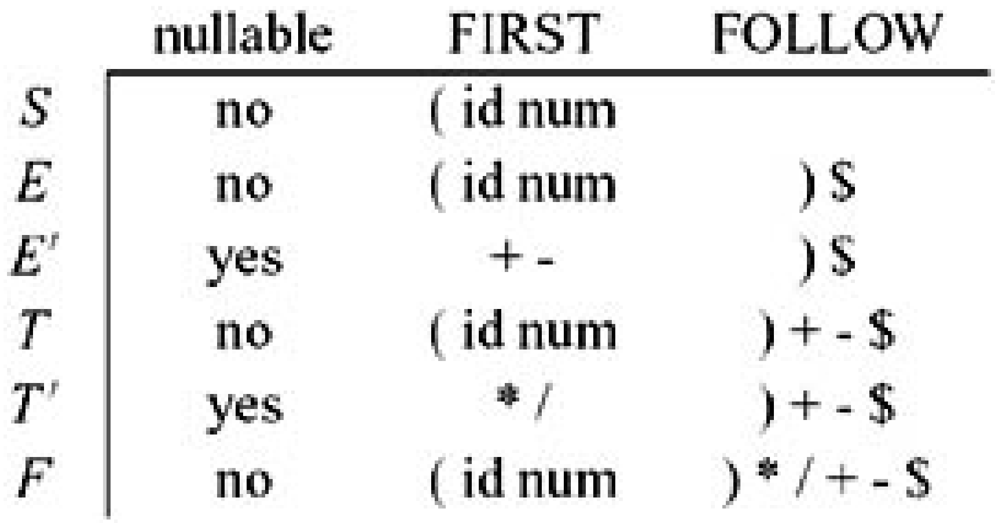
$T' \rightarrow /\ F\ T'$

$T' \rightarrow$

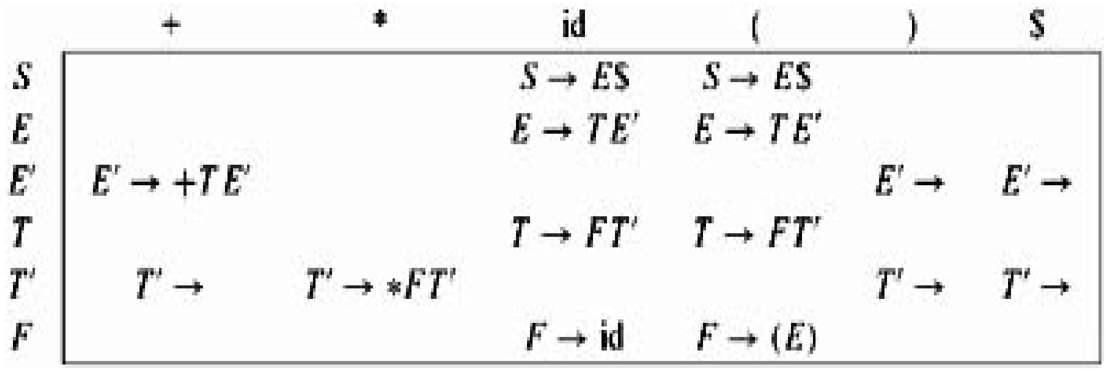
$F \rightarrow \text{id}$

$F \rightarrow \text{num}$

$F \rightarrow (E)$



	nullable	FIRST	FOLLOW
S	no	(id num	
E	no	(id num	) \$
E'	yes	+ -	) \$
T	no	(id num	) + - \$
T'	yes	* /	) + - \$
F	no	(id num	) * / + - \$



	+	*	id	(	)	\$
S			$S \rightarrow E\$$	$S \rightarrow E\$$		
E			$E \rightarrow TE'$	$E \rightarrow TE'$		
E'	$E' \rightarrow +TE'$				$E' \rightarrow$	$E' \rightarrow$
T			$T \rightarrow FT'$	$T \rightarrow FT'$		
T'	$T' \rightarrow$	$T' \rightarrow *FT'$			$T' \rightarrow$	$T' \rightarrow$
F			$F \rightarrow \text{id}$	$F \rightarrow (E)$		

Left Factoring

S → IF *E* THEN *S* ELSE *S*

S → IF *E* THEN *S*

$\Longrightarrow$

S → IF *E* THEN *S* *X*

X → ELSE *S* | ε

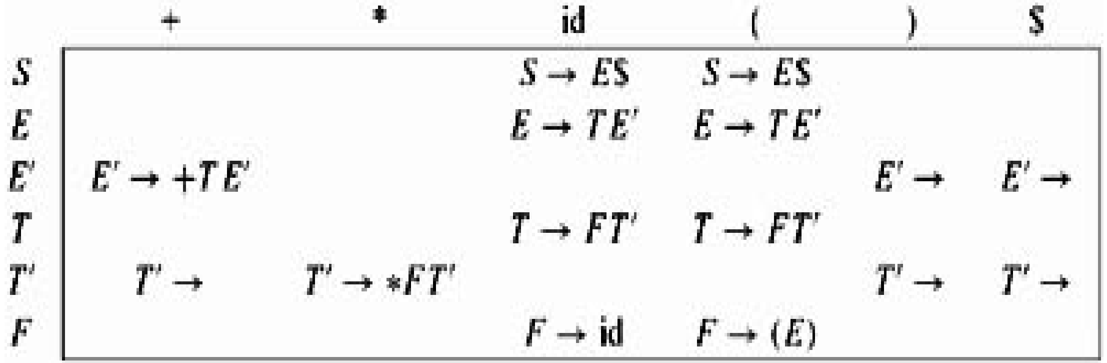
ERROR RECOVERY

How should *error* be handled?

- Raise an exception and quit parsing
- Print an error message and recover from the error

This can proceed by deleting, replacing, or inserting tokens.

```
void T( ) { switch (tok) {
        case ID:
        case NUM:
        case LPAREN: F( ); Tprime(  ); break;
        default: error!
    }}
```



	+	*	id	(	)	$
S			$S \to ES$	$S \to ES$		
E			$E \to TE'$	$E \to TE'$		
E'	$E' \to +TE'$				$E' \to$	$E' \to$
T			$T \to FT'$	$T \to FT'$		
T'	$T' \to$	$T' \to *FT'$			$T' \to$	$T' \to$
F			$F \to \text{id}$	$F \to (E)$		

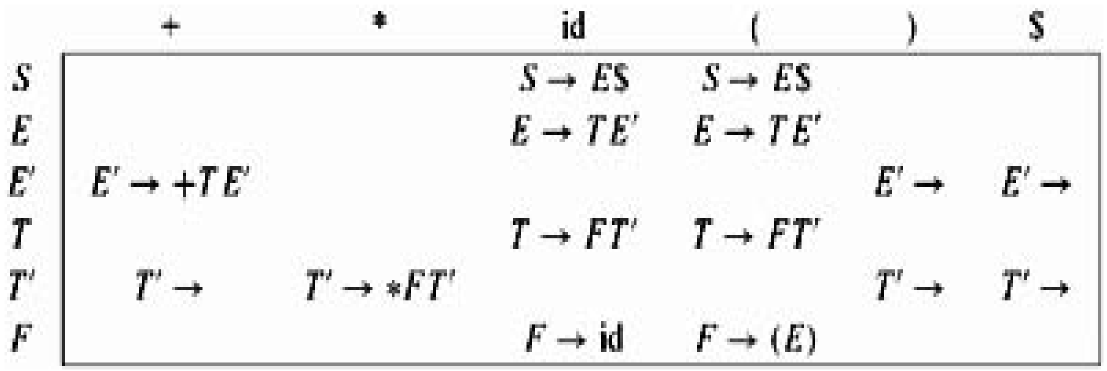
ERROR RECOVERY

```
void T( ) { switch (tok) {
        case ID:
        case NUM:
        case LPAREN: F( ); Tprime(  ); break;
        default: print("expected id, num, or left-paren");
      }}
```

- Error recovery by deletion is safer, because the loop must eventually terminate when end-of-file is reached.
- Simple recovery by deletion works by skipping tokens until a token in the FOLLOW set is reached.

ERROR RECOVERY

```
int Tprime_follow [ ] = {PLUS, RPAREN, EOF};
void Tprime( )  { switch (tok) {
     case PLUS: break;
     case TIMES: eat(TIMES); F(); Tprime(); break;
     case RPAREN: break;
     case EOF: break;
     default:  print("expected +, *, right-paren, or end-of-file");
                  skipto(Tprime_follow);
   }}
```



	+	*	id	(	)	$
S			$S \rightarrow E\$$	$S \rightarrow E\$$		
E			$E \rightarrow TE'$	$E \rightarrow TE'$		
E'	$E' \rightarrow +TE'$				$E' \rightarrow$	$E' \rightarrow$
T			$T \rightarrow FT'$	$T \rightarrow FT'$		
T'	$T' \rightarrow$	$T' \rightarrow *FT'$			$T' \rightarrow$	$T' \rightarrow$
F			$F \rightarrow \text{id}$	$F \rightarrow (E)$		

The end of Chapter 3(2)