

Top-Down Parsing

Non-terminal as a Function

- In a top-down parser a non-terminal may be viewed as a generator of a substring of the input.
- We view a non-terminal as a function that generates the substring.
- In the expression grammar, $E \rightarrow E + T$, $E \rightarrow E - T$, $E \rightarrow T$, the function may look as follows:

Function of E

$E()$

Select an E -production p // possible choice

if $p = E \rightarrow E + T$ // and backtracking

if $E()$ then

if $yylex() = '+'$ then

if $T()$ then return OK

else ERROR

else ERROR

else ERROR

Example

- Let the input be **ic**... The parser may choose the production rule $E \rightarrow E + T^a$.
- As there is no change in the input and the leftmost non-terminal is still E . It may be expanded by the same rule again and again.
- A **left recursive** grammar leads to non-termination.

^aBut how to choose, each rule of E produces a string starting with **ic**.

Example

- The parser chooses the following sequence of rules: $E \rightarrow T$, $T \rightarrow F$ and $F \rightarrow ic$.
- The first symbol of the input matches, but the choice may be incorrect if the next input symbol is '+', as there is no rule with right hand side $F + \dots$.
- It may be necessary to **backtrack** on the choice of production rules.

Example

Consider the grammar:

$$S \rightarrow aSa \mid aTba \mid c$$

$$T \rightarrow bS$$

And let the first symbol of the input be $a \cdots$.

Example

- A parser with 1-lookahead cannot decide whether to use the **first** or the **second** rule.
- But if it is allowed to **look-ahead** one more symbol, the correct choice can be made.
- If the input is **aa**..., it selects the rule $S \rightarrow aSa$. But if it is **ab**..., the choice is $S \rightarrow aTba$.

Note

- In case of the expression grammar G , no fixed amount of look-ahead can help.
- The parser may have 5-look-ahead and the input is $ic+ic+ic\cdots$.
- The derivation sequence will be $E \rightarrow E + T \rightarrow E + T + T$. But the next step cannot be decided as the operator after the rightmost ic is not known.

Example

Consider the grammar:

$$S \rightarrow Aab$$

$$A \rightarrow a \mid \varepsilon$$

Functions of a recursive descent parser are as follows.

Function of S

$S()$

if $A()$ then

if $yylex() = 'a'$ then

if $yylex() = 'b'$ then return OK

else ERROR

else ERROR

$A()$

if $yylex() = 'a'$ return OK # incorrect

else return ERROR

Note

- The parser cannot recognize ' ab '.
- The problem is, A can produce a as the first symbol. But A also produces ε , and a can come after A .
- The parser cannot decide whether to use $A \rightarrow a$ or $A \rightarrow \varepsilon$.

Example

Consider the ambiguous grammar:

$$S \rightarrow aSa \mid bSb \mid aTba \mid c$$

$$T \rightarrow bS$$

There is no way to decide a rule entirely on the basis of the input, without removing the ambiguity.

$LL(k)$

An unambiguous context-free grammar without left recursion is called an $LL(k)$ grammar^a, if a **predictive parser** for its language can be constructed with a **look-ahead** of at most k input symbols. Often for a compiler construction we consider $k = 1$.

^aThe parser scans the input from **left-to-right** and uses the **leftmost** derivation.

Note

In an $LL(k)$ grammar

- For every non-terminal A , the right-hand side of each production rule must produce l distinct terminal symbols as prefix (first symbols), for some $l \leq k$.
- If $A \rightarrow \varepsilon$ is a rule, then l terminal symbols that can appear behind A (follow) in a sentential form should be different from the prefixes of other rules of A , for some $l \leq k$.

Note

If A is a non-terminal in a $LL(k)$ grammar and $A \rightarrow \alpha_1 \mid \cdots \mid \alpha_n \mid \varepsilon$,

- Then for some $l \leq k$, $\text{First}_l(\alpha_i) \cap \text{First}_l(\alpha_j) = \emptyset$, for $1 \leq i < j \leq n$.
- And for some $m \leq k$, $\text{First}_m(\alpha_i) \cap \text{Follow}_m(A) = \emptyset$, for $1 \leq i \leq n$.

$\text{First}_l(\alpha_i)$ is the set of first l symbols produced by α_i . $\text{Follow}_m(A)$ is the set terminals of length m that can follow A in a sentential form.

Note

Let $A \rightarrow \alpha_1 \mid \cdots \mid \alpha_n$, and α_i is nullable i.e. α_i produces ε .

Then not only for some $l \leq k$, $\text{First}_l(\alpha_i) \cap \text{First}_l(\alpha_j) = \emptyset$, for $1 \leq i < j \leq k$, but also $\text{First}_m(\alpha_i)$ should be disjoint from $\text{Follow}_m(A)$ for some $m \leq k$, for $1 \leq i \leq n$.
But two such α 's cannot be nullable.

Information From Grammar

- It is necessary to extract **First** and **Follow** information from the given grammar to decide whether it is $LL(k)^a$.
- This information enables the $LL(k)$ **parser** to choose the correct action.
- We restrict our attention to $k = 1$.

^aIt may also help to transform the grammar to $LL(k)$ if possible.

FIRST(X)

Informally the FIRST(X) is a set, where X is a terminal, a non-terminal, a string over terminals and non-terminals, or even a production rule. The set is a collection of all terminal symbols (also ε) that may appear as the first (leftmost) symbol of X in the given grammar.

FIRST(X)

If $X \in \Sigma \cup N \cup \{\varepsilon\}$, then $\text{FIRST}(X) \subseteq \Sigma \cup \{\varepsilon\}$ is defined inductively as follows:

- $\text{FIRST}(X) = \{X\}$, if $X \in \Sigma \cup \{\varepsilon\}$,
- $\text{FIRST}(X)$ is $\bigcup_{X \rightarrow \alpha \in P} \text{FIRST}(\alpha)$, if $X \in N$,
- $\varepsilon \in \text{FIRST}(X)$, if X is nullable,
- $\text{FIRST}(A \rightarrow \alpha)$ is the $\text{FIRST}(\alpha)$.

FIRST(α)

If $\alpha = X_1 X_2 \cdots X_k$, then

- $\text{FIRST}(X_1) \setminus \{\varepsilon\} \subseteq \text{FIRST}(\alpha)$.
- $\text{FIRST}(X_i) \setminus \{\varepsilon\} \subseteq \text{FIRST}(\alpha)$ if $\varepsilon \in \bigcap_{j=1}^{i-1} \text{FIRST}(X_j)$, $1 < i \leq k$.
- If $\varepsilon \in \bigcap_{j=1}^k \text{FIRST}(X_j)$, then $\varepsilon \in \text{FIRST}(\alpha)$.

Example

Consider the classic expression grammar:

- $\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{\text{ic}, (\}$.
- There are two production rules for both E and T with identical $\text{FIRST}()$ sets:
 $E \rightarrow E + T, E \rightarrow T$ and $T \rightarrow T * F, T \rightarrow F$
- This makes the choice of rule impossible with finite look-ahead.

Example

The first two symbols produced by the non-terminals of the expression grammar are,

- $\text{FIRST}_2(F) = \{((, (ic\}$.
- $\text{FIRST}_2(T) = \{((, (ic, ic*\}$.
- $\text{FIRST}_2(E) = \{((, (ic, ic*, ic+\}$.

Example

Consider the grammar obtained after removing the **left-recursion** from G :

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \varepsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \varepsilon$$

$$F \rightarrow (E) \mid \text{ic}$$

Example

$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{\text{ic}, (\},$
 $\text{FIRST}(E') = \{+, \varepsilon\},$ and $\text{FIRST}(T') = \{*, \varepsilon\}.$
No non-terminal has more than one production rule with the identical **FIRST()** set.

Note

- Let $A \rightarrow \alpha$ and $A \rightarrow \beta$ be two production rules. A top-down parser can choose one of them with **one look-ahead** if $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$ and none of them contains ε .
- But what happens if one of α or β is **nullable**?

FOLLOW(X)

For every non-terminal X , the FOLLOW(X) is the collection of all terminals that can follow X in a sentential form. The set can be defined inductively as follows.

- The special symbol eof or \$ is in FOLLOW(S), where S is the start symbol.
- If $A \rightarrow \alpha B \beta$ be a production rule, $\text{FIRST}(\beta) \setminus \{\varepsilon\} \subseteq \text{FOLLOW}(B)$.

FOLLOW(X)

- If $A \rightarrow \alpha B \beta$, where $\beta = \varepsilon$ or $\beta \Rightarrow^* \varepsilon$, then $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$.

The reason is simple:

$S \Rightarrow uAv \Rightarrow u\alpha B\beta v \Rightarrow u\alpha Bv$, naturally $\text{FIRST}(v) \subseteq \text{FOLLOW}(A), \text{FOLLOW}(B)$.

Computation of FOLLOW() Sets

for each $A \in N$

$\text{FOLLOW}(A) \leftarrow \emptyset$

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

Computation of FOLLOW() Sets

while (FOLLOW sets are not fixed points)

for each $A \rightarrow \alpha B \beta \in P$

$\text{FOLLOW}(B) \leftarrow \text{FOLLOW}(B) \cup (\text{FIRST}(\beta) \setminus \{\varepsilon\})$

if $\beta = \varepsilon$ or $\varepsilon \in \text{FIRST}(\beta)$

$\text{FOLLOW}(B) \leftarrow \text{FOLLOW}(B) \cup \text{Follow}(A)$

Example

In the expression grammar G :

$\text{FOLLOW}(E) = \{\$, +,)\}$, $\text{FOLLOW}(T) = \text{FOLLOW}(E) \cup \{*\} = \{\$, +,), *\}$ and $\text{FOLLOW}(F) = \{\$, +,), *\}$.

In the transformed grammar:

$\text{FOLLOW}(E) = \text{FOLLOW}(E') = \{\$,)\}$,
 $\text{FOLLOW}(T) = \text{FOLLOW}(T') = \{\$,), +\}$ and
 $\text{FOLLOW}(F) = \{\$,), +, *\}$.

Note

- Let $A \rightarrow \alpha$ and $A \rightarrow \varepsilon$ be two production rules. A top-down parser can choose a rule if $\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \emptyset$.
- The first rule is chosen if the next symbol is from the $\text{FIRST}(\alpha)$.
- The second rule is chosen if the next symbol is from the $\text{FOLLOW}(A)$.

Note

- Let $A \rightarrow \alpha$ and $A \rightarrow \beta$ be two production rules such that β is nullable. A top-down parser can still choose a rule if $\text{FIRST}(\alpha) \cap (\text{FIRST}(\beta) \cup \text{FOLLOW}(A)) = \emptyset$.
- The first rule is chosen if the next symbol is from the $\text{FIRST}(\alpha)$.
- The second rule is chosen if the next symbol is from the $\text{FIRST}(\beta) \cup \text{FOLLOW}(A)$.

$LL(1)$ Grammar

A context-free grammar G is $LL(1)$ iff for any pair of distinct productions $A \rightarrow \alpha$, $A \rightarrow \beta$, the following conditions are satisfied.

- $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$ i.e. no $a \in \Sigma \cup \{\varepsilon\}$ can belong to both^a.
- If $\alpha \rightarrow \varepsilon$ or $\alpha = \varepsilon$, then $\text{FIRST}(\beta) \cap (\text{FOLLOW}(A) \cup \text{FIRST}(\alpha)) = \emptyset$.

^aBoth cannot be nullable.

Example

Consider the following grammar with the set of terminals,

$\Sigma = \{\text{id} \ ; \ := \ \text{int} \ \text{float} \ \text{main} \ \text{do} \ \text{else} \ \text{end} \ \text{if} \ \text{print} \ \text{scan} \ \text{then} \ \text{while}\} \cup \{\text{E} \ \text{BE}\}^{\text{a}};$

the set of non-terminals,

$N = \{\text{P} \ \text{DL} \ \text{D} \ \text{VL} \ \text{T} \ \text{SL} \ \text{S} \ \text{ES} \ \text{IS} \ \text{WS} \ \text{IOS}\};$

the start symbol is **P** and the set of production rules are:

^aE and BE, corresponds to expression and boolean expressions, are actually non-terminals. But here we treat them as terminals.

Production Rules

- 1 $P \rightarrow \text{main DL SL end}$
- 2 $DL \rightarrow D DL \mid D$
- 4 $D \rightarrow T VL ;$
- 5 $VL \rightarrow \text{id VL} \mid \text{id}$
- 7 $T \rightarrow \text{int} \mid \text{float}$
- 9 $SL \rightarrow S SL \mid \varepsilon$
- 11 $S \rightarrow \text{ES} \mid \text{IS} \mid \text{WS} \mid \text{IOS}$

Production Rules

15 $ES \rightarrow id := E ;$

16 $IS \rightarrow \text{if } BE \text{ then } SL \text{ end } |$

$\text{if } BE \text{ then } SL \text{ else } SL \text{ end}$

18 $WS \rightarrow \text{while } BE \text{ do } SL \text{ end}$

19 $IOS \rightarrow \text{scan } id ; | \text{print } E ;$

Note

There is no production rule with **left-recursion**. But the rules $\{2, 3\}$, $\{5, 6\}$, and $\{16, 17\}$ needs **left-factoring** as the **FIRST()** sets are not disjoint. The transformed grammar after factoring is:

New Production Rules

1 $P \rightarrow \text{main DL SL end}$

2 $DL \rightarrow D DO$

3 $DO \rightarrow DL \mid \varepsilon$

4 $D \rightarrow T VL ;$

5 $VL \rightarrow \text{id } VO$

6 $VO \rightarrow VL \mid \varepsilon$

7 $T \rightarrow \text{int} \mid \text{float}$

Production Rules

- 9 SL $\rightarrow$ S SL | ε
- 11 S $\rightarrow$ ES | IS | WS | IOS
- 15 ES $\rightarrow$ id := E ;
- 16 IS $\rightarrow$ if BE then SL EO
- 17 EO $\rightarrow$ end | else SL end
- 18 WS $\rightarrow$ while BE do SL end
- 19 IOS $\rightarrow$ scan id ; | print E ;

FIRST()

The next step is to calculate the **FIRST()** sets of different rules.

NT/Rule	FIRST()
P (1)	main
DL (2)	int float
D0 (3)	int float
D0 (3a)	ϵ
D (4)	int float

FIRST()

NT/Rule	FIRST()
VL (5)	id
V0 (6)	id
V0 (6a)	ϵ
T (7)	int
T (8)	float
SL (9)	id if while scan print

FIRST()

NT/Rule	FIRST()
SL (10)	ε
S (11)	id
S (12)	if
S (13)	while
S (14)	scan print

FIRST()

NT/Rule	FIRST()
ES (15)	id
IS (16)	if
EO (17)	end
EO (17a)	else

FIRST()

NT/Rule	FIRST()
WS (18)	while
IOS (19)	scan
IOS (20)	print

Note

Three rules have ε -productions. Their applications in a predictive parser depends on what can follow the corresponding non-terminals. So it is necessary to compute the **FOLLOW()** sets corresponding to these non-terminals. The rules are:

$DO \rightarrow \varepsilon(3a)$, $VO \rightarrow \varepsilon(6a)$, $SL \rightarrow \varepsilon(10)$.

FOLLOW()

NT	FOLLOW()
DO	id if while scan print end
VO	;
SL	end else

Note

$\text{FOLLOW}(\text{DO}) = \text{FOLLOW}(\text{DL})$ (rule 2). The
 $\text{FOLLOW}(\text{DL}) = \text{FIRST}(\text{SL}) \setminus \{\varepsilon\} \cup$
 $\text{FOLLOW}(P)$ (rule 1) as SL is nullable (rule
10). Now $\text{FOLLOW}(P) = \{\text{end}\}$.

Note

It is clear from the previous computation that no two production rules of the form $A \rightarrow \alpha_1 \mid \alpha_2$ have common elements in their **FIRST()** sets. There is also no common element in the **FIRST()** set of the production rule $A \rightarrow \alpha$ and the **FOLLOW()** set of A in cases $A \rightarrow \varepsilon$. So the grammar is ***LL*(1)** and a **predictive parser** can be constructed.

Recursive-Descent Parser

We write a function (may be recursive) for every non-terminal. The function corresponding to a non-terminal A returns **ACCEPT** if the corresponding portion of the input can be generated by A . Otherwise it returns a **REJECT** with proper error message.

Example

Consider the production rule

$$P \rightarrow \text{main DL SL end}$$

The function corresponding to the non-terminal **P** is as follows:

```
int P()
```

```
int P(){
    getNextToken();
    if(token == MAIN){ // MAIN for "main"
        if(DL() == ACCEPT)
            if(SL() == ACCEPT) {
                getNextToken();
                if(token == END){ // END is the token
                    return ACCEPT; // for "end"
                }
            }
        else {
            printf("end missing (1)\n");
            return REJECT;
        }
    }
}
```

```
        }  
    }  
    else {  
        printf("SL mismatch (1)\n");  
        return REJECT;  
    }  
    else {  
        printf("DL mismatch (1)\n");  
        return REJECT;  
    }  
}  
else {  
    printf("main missing (1)\n");  
    return REJECT;  
}
```

}

}

Note

The global variable **token** stores the **next token**. The function **getNextToken()** is called once the token is consumed.

The **stack** of the **PDA** is the stack of the **recursive call**. The body of the function corresponding to a non-terminal corresponds to all its production rules.

Example

We now consider a non-terminal with ε -production.

$$D0 \rightarrow DL \mid \varepsilon$$

The members of $\text{FIRST}(DL)$ are `{int float}` and the elements of $\text{FOLLOW}(D0)$ are `{id if while scan print end}`.

```
int DO()
```

```
int DO(){  
    getNextToken();  
    if(token == INT || token == FLOAT)  
        // token is not consumed  
        if(DL() == ACCEPT) {  
            return ACCEPT;  
        }  
        else {  
            printf("DL mismatch (3)\n");  
            return REJECT;  
        }  
    else {
```

```
    if(token == IDENTIFIER ||
        token == IF ||
        token == WHILE ||
        token == SCAN ||
        token == PRINT ||
        token == END) // token not consumed
        return ACCEPT;
    else {
        putBackToken();
        printf("DO follow mismatch (3)\n");
        return REJECT;
    }
}
```

Table Driven Predictive Parser

A non-recursive predictive parser can be constructed that maintains a stack (explicitly) and a table to select the appropriate production rule.

Parsing Table

1. The **rows** of the predictive parser table are indexed by the **non-terminals**.
2. The **columns** are indexed by the **terminals** including the **end-of-input marker (\$)**.
3. The content of the table are production rules or error situations. The table cannot have multiple entries corresponding to a **(row, column)**.

Parsing Stack

The parsing stack can hold both **terminals** and **non-terminals**. At the beginning, the stack contains the **end-of-stack marker** (\$) and the **start symbol** on top of it.

Parsing Table Construction

- If $A \rightarrow \alpha$ is a production rule and $a \in \text{FIRST}(\alpha)$, then $P[A][a] = A \rightarrow \alpha$.
- If $A \rightarrow \varepsilon$ is a production rule and $a \in \text{FOLLOW}(A)$, then $P[A][a] = A \rightarrow \varepsilon$.

Actions

- If the **top-of-stack** is a **terminal symbol** (token) and matches with **input token**, both are **consumed**. A mismatch is an **error**.
- If the **top-of-stack** is a **non-terminal** A , the **input token** is a , and $P[A][a]$ has the entry $A \rightarrow \alpha$, then A on the stack is replaced by α , with the **head** of α on the top of the stack.

Example

Consider the production rules of the non-terminal **SL**.

$$\text{SL} \rightarrow \text{S SL} \mid \varepsilon$$

The $\text{FIRST}(\text{SL} \rightarrow \text{S SL}) = \{\text{id if while scan print}\}$ and $\text{FOLLOW}(\text{SL}) = \{\text{end else}\}$. So,
 $P[\text{SL}][\text{IDENTIFIER}] = P[\text{SL}][\text{IF}] = P[\text{SL}][\text{WHILE}] = P[\text{SL}][\text{SCAN}] = P[\text{SL}][\text{PRINT}] = \text{SL} \rightarrow \text{S SL}$ and
 $P[\text{SL}][\text{END}] = P[\text{SL}][\text{ELSE}] = \text{SL} \rightarrow \varepsilon$.

Note

Multiple entries in a table indicates that the grammar is not $LL(1)$. But it is interesting to note that in some cases we can drop (with proper consideration) some of these entries and construct a parser.

Example

Consider the ambiguous grammar G_1 for expressions.

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid ic$$

After the removal of **left-recursion** we get the following ambiguous, non-left-recursive grammar:

Example

$$E \rightarrow (E)E' \mid icE'$$

$$E' \rightarrow +EE' \mid -EE' \mid *EE' \mid /EE' \mid \varepsilon$$

We calculate $\text{FIRST}(E') = \{+ \ - \ * \ / \ \varepsilon \}$ and the $\text{FOLLOW}(E') = \text{FOLLOW}(E) = \{\$ \) \ + \ - \ * \ /\}$.

Example

Naturally,

$P[E'][\pm] = \{E' \rightarrow +EE', E' \rightarrow \varepsilon\}$ and

$P[E'][* /] = \{E' \rightarrow *EE', E' \rightarrow \varepsilon\}.$

We may drop the ε -productions from these four places and get a nice parsing table^a.

^aBut it does not work for all grammars. Consider $S \rightarrow aSa \mid bSb \mid \varepsilon$.

Note

It seems that the removal of two ε -production disambiguates the grammar. The corresponding unambiguous grammar G_2 is as follows:

$$E \rightarrow (E)E' \mid icE' \mid (E) \mid ic$$

$$E' \rightarrow +E \mid -E \mid *E \mid /E \mid \varepsilon$$

We have $L(G_1) = L(G_2)$ and $\text{FOLLOW}(E') = \{\$, \text{)}\}$, so there is no multiple entries in the table^a.

^aHow to maintain operator precedence?

Error Recovery

There are two possibilities.

- The token on the top of stack does not match with the token in the input stream.
- The entry in the parsing table corresponding to the non-terminal on the top of stack and the current input token is empty, i.e. there is no prediction for a production rule of the non-terminal.

Error Recovery

There are two main concerns:

- Avoidance of **infinite loop** during error handling.
- Avoidance of the **construction of corrupted syntax tree**.

An Example

- Consider an example where the non-terminal A is on the top of the stack, where its production rules are $A \rightarrow aA \mid bc$ (The non-terminal A produces a^*bc .), and
- ‘ c ’ is input look-ahead.
- No prediction is possible due to error.

An Example

- We cannot **remove** A from the stack. That changes a part of already constructed tree.
- Forcing a prediction $A \rightarrow aA$ by inserting an ‘ a ’ will lead to an **infinite loop**.
- Tokens may be discarded from the input to get a match. But how far can we skip.
- In this case of course inserting a ‘ b ’ may solve the problem.

Panic Mode

- Remove sequence of **tokens** from the input until a **synchronizing** token appears.
- The success of the algorithm depends on the **set of synchronizing tokens**.

Synchronizing Tokens

- For a non-terminal A , the $\text{Follow}(A)$ may be the set of synchronizing tokens.
- Tokens are removed until an element of $\text{Follow}(A)$ is found. Then pop A from the stack and try to continue with parsing.

Synchronizing Tokens

- An **expression** becomes a statement when followed by a **semicolon**.
- If a semicolon (‘;’) is missing, the **follow set of expression** will not be of help as synchronizing symbol.
- We need to include possible **first symbols** of next statement or even higher level constructs.

References

- [ASRJ] Compilers Principles, Techniques, and Tools, by A. V. Aho, Monica S. Lam, R. Sethi, & J. D. Ullman, 2nd ed., ISBN 978-81317-2101-8, Pearson Ed., 2008.
- [DKHJK] Modern Compiler Design, by Dick Grune, Kees van Reeuwijk, Henri E. Bal, Criel J. H. Jacobs, Koen Langendoen, 2nd ed., ISBN 978 1461 446989, Springer (2012).
- [KL] Engineering a Compiler, by Keith D. Cooper & Linda Troczon, (2nd ed.), ISBN 978-93-80931-87-6, Morgan Kaufmann, Elsevier, 2012.