CS 346: Top Down Parser

Resource: Textbook

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, "*Compilers: Principles, Techniques, and Tools*", Addison-Wesley, 1986.

Top-Down Parsing

- Parse tree created top to bottom
- Top-down parser
  - Recursive-Descent Parsing
    - Backtracking needed: If a choice of a production rule does not work, we backtrack to try other alternatives
    - General parsing technique, but not widely used
    - Not efficient
  - Predictive Parsing
    - no backtracking
    - efficient
    - needs a special form of grammars (LL(1) grammars)
    - *Recursive predictive parsing-a special form of recursive descent parsing without backtracking*
    - *Non-Recursive* (Table Driven) predictive parser: also known as LL(1) parser

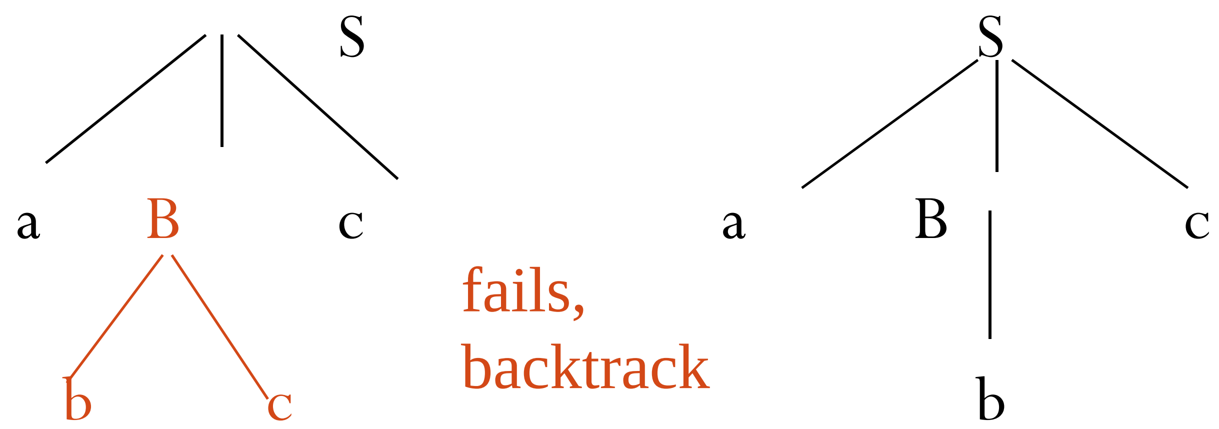
Recursive-Descent Parsing (uses Backtracking)

- Backtracking is needed
- Tries to find the left-most derivation

$S \rightarrow aBc$

$B \rightarrow bc \mid b$

input: abc



Predictive Parser

a grammar ➔ ➔ a grammar suitable for predictive parsing (a LL(1) grammar), however

eliminate left recursion — left factor — no %100 guarantee

- When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the current symbol in the input string

$A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$ input: ... a ....

current token

Predictive Parser (example)

stmt → `if` ...... |
`while` ...... |
`begin` ...... |
`for` .....

- When we are trying to write the non-terminal *stmt*, if the current token is `if` we have to choose first production rule
- When we are trying to write the non-terminal *stmt*, we can uniquely choose the production rule by just looking the current token
- Eliminate the left recursion in the grammar, and left factor it. But it may not be suitable for predictive parsing (not LL(1) grammar)

Recursive Predictive Parsing

- Each non-terminal corresponds to a procedure

Ex: $A \rightarrow aBb$ (This is only the production rule for A)

proc A {

- match the current token with a, and move to the next token;
- call 'B';
- match the current token with b, and move to the next token

}

Recursive Predictive Parsing (cont.)

$A \rightarrow aBb \mid bAB$

```
proc A {
    case of the current token {
          'a':  - match the current token with a, and move to the next token;
                - call 'B';
                - match the current token with b, and move to the next token;
          'b':  - match the current token with b, and move to the next token;
                - call 'A';
                - call 'B';
    }
}
```

Recursive Predictive Parsing (cont.)

- When to apply ε-productions?

  e.g., $A \rightarrow aA \mid bB \mid \varepsilon$

- Apply ε-production if all other productions fail
  - For example, if the current token is not a or b, we may apply the ε-production

- *Most correct choice*: Apply an ε-production for a non-terminal A when the current token is in the follow set of A
  - terminals that can follow A in the sentential forms (*coming later*)

Recursive Predictive Parsing (Example)

$A \rightarrow aBe \mid cBd \mid C$
$B \rightarrow bB \mid \varepsilon$
$C \rightarrow f$

```
proc A {
    case of the current token {
        a:    - match the current token with a,
                and move to the next token;
              - call B;
              - match the current token with e,
                and move to the next token;
        c:    - match the current token with c,
                and move to the next token;
              - call B;
              - match the current token with d,
                and move to the next token;
        f:    - call C
    }
}
```

first set of C

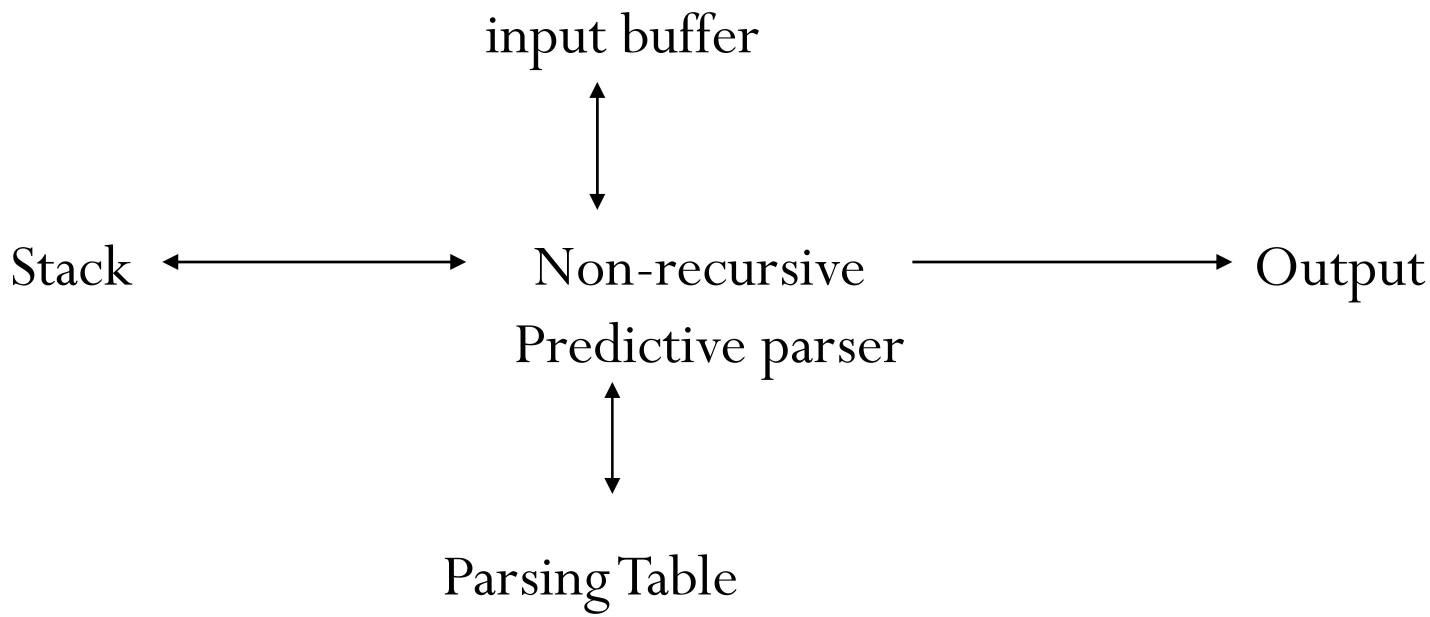
```
proc C {    match the current token with f,
            and move to the next token; }
```

```
proc B {
    case of the current token {
        b:    - match the current token with b,
                and move to the next token;
              - call B
        e,d:  do nothing
    }
}
```

follow set of B

Non-Recursive Predictive Parsing -- LL(1) Parser

- Non-Recursive predictive parsing
  - table-driven parser
  - top-down parser
  - also known as LL(1) Parser



LL(1) Parser

input buffer

- Contains the string to be parsed
- End is marked with a special symbol $

output

- a production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer

stack

- contains the grammar symbols
- at the bottom of the stack, there is a special end marker symbol $
- initially the stack contains only the symbol $ and the starting symbol S
- $S 🡐 initial stack
- Parsing completes when the stack becomes empty (i.e. only $ left in the stack)

LL(1) Parser

parsing table

- a two-dimensional array M[A, a]
- each row is a non-terminal symbol
- each column is a terminal symbol or the special symbol $
- each entry holds a production rule

LL(1) Parser – Parser Actions

- Parser action: determined by the symbol at the top of the stack (say X) and the current symbol in the input string (say a)

- Four possible parser actions:

1. If X and a are \$ ➔ parser halts (successful completion)

2. If X and a are the same terminal symbol (different from \$)

   ➔ parser pops X from the stack, and moves to the next symbol in the input buffer

LL(1) Parser–Parser Actions

3. If X is a non-terminal
   - ➔ parser looks at the parsing table entry M[X ,a]
   - ➔ If M[X, a] holds a production rule $X \rightarrow Y_1 Y_2 \ldots Y_k$
     - ➔ pop X from the stack
     - ➔ push $Y_k, Y_{k-1}, \ldots, Y_1$ into the stack
   - ➔ Output the production rule $X \rightarrow Y_1 Y_2 \ldots Y_k$ to represent a step of the derivation

4. none of the above ➔ error
   - all empty entries in the parsing table are errors
   - If X is a terminal symbol different from a, this is also an error case

LL(1) Parser – Example1

$S \rightarrow aBa$

$B \rightarrow bB \mid \varepsilon$

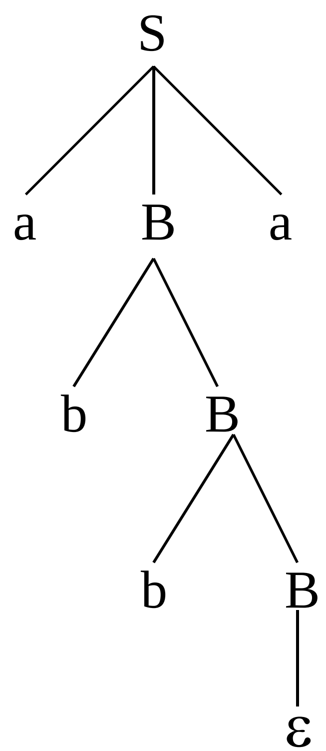
	a	b	$
S	$S \rightarrow aBa$		
B	$B \rightarrow \varepsilon$	$B \rightarrow bB$	

stack	input	output
$S	abba$	$S \rightarrow aBa$
$aBa	abba$	
$aB	bba$	$B \rightarrow bB$
$aBb	bba$	
$aB	ba$	$B \rightarrow bB$
$aBb	ba$	
$aB	a$	$B \rightarrow \varepsilon$
$a	a$	
$	$	accept, successful completion

LL(1) Parser – Example1 (cont.)

Outputs: $S \rightarrow aBa$ $\quad B \rightarrow bB$ $\quad B \rightarrow bB$ $\quad B \rightarrow \varepsilon$

Derivation(left-most): $S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$



parse tree

LL(1) Parser – Example2

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

	id	**+**	*****	**(**	**)**	**$**
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

LL(1) Parser – Example2

stack	input	output
$E	id+id$	$E \rightarrow TE'$
$E'T	id+id$	$T \rightarrow FT'$
$E'T'F	id+id$	$F \rightarrow id$
$ E'T'id	id+id$	
$ E'T'	+id$	$T' \rightarrow \varepsilon$
$ E'	+id$	$E' \rightarrow +TE'$
$ E'T+	+id$	
$ E'T	id$	$T \rightarrow FT'$
$ E'T' F	id$	$F \rightarrow id$
$ E'T'id	id$	
$ E'T'	$	$T' \rightarrow \varepsilon$
$ E'	$	$E' \rightarrow \varepsilon$
$	$	accept

Constructing LL(1) Parsing Tables

- Two functions used in the construction of LL(1) parsing tables:
  - FIRST FOLLOW

- **FIRST(α):** Set of the terminal symbols which occur as first symbols in strings derived from α where α is any string of grammar symbols
  - if α derives to ε, then ε is also in FIRST(α)

- **FOLLOW(A):** Set of the terminals which occur immediately after (follow) the *non-terminal A* in the strings derived from the starting symbol
  - a terminal a is in FOLLOW(A) if $S \overset{*}{\Rightarrow} \alpha A a \beta$
  - $ is in FOLLOW(A) if $S \overset{*}{\Rightarrow} \alpha A$

Compute FIRST for Any String X

- If X is a terminal symbol ➔ FIRST(X)={X}
- If X is a non-terminal symbol and $X \rightarrow \varepsilon$ is a production rule
  ➔ ε is in FIRST(X)
- If X is a non-terminal symbol and $X \rightarrow Y_1 Y_2 .. Y_n$ is a production rule
  - ➔ if a terminal **a** in FIRST(Y_i) and ε is in all FIRST(Y_j) for j=1,...,i-1 then **a** is in FIRST(X)
  - ➔ if ε is in all FIRST(Y_j) for j=1,...,n then ε is in FIRST(X)
- If X is ε ➔ FIRST(X)={ε}
- If X is $Y_1 Y_2 .. Y_n$
  - ➔ if a terminal **a** in FIRST(Y_i) and ε is in all FIRST(Y_j) for j=1,...,i-1 then **a** is in FIRST(X)
  - ➔ if ε is in all FIRST(Y_j) for j=1,...,n then ε is in FIRST(X)

FIRST Example

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid id$$

$FIRST(F) = \{(, id\}$
$FIRST(T') = \{*, \varepsilon\}$
$FIRST(T) = \{(, id\}$
$FIRST(E') = \{+, \varepsilon\}$
$FIRST(E) = \{(, id\}$

$FIRST(TE') = \{(, id\}$
$FIRST(+TE') = \{+\}$
$FIRST(\varepsilon) = \{\varepsilon\}$
$FIRST(FT') = \{(, id\}$
$FIRST(*FT') = \{*\}$
$FIRST(\varepsilon) = \{\varepsilon\}$
$FIRST((E)) = \{(\}$
$FIRST(id) = \{id\}$

Compute FOLLOW (for non-terminals)

- If S is the start symbol ➔ \$ is in FOLLOW(S)

- if $A \rightarrow \alpha B \beta$ is a production rule
  ➔ everything in FIRST(β) is FOLLOW(B) except ε

- If ($A \rightarrow \alpha B$ is a production rule) or
  ($A \rightarrow \alpha B \beta$ is a production rule and ε is in FIRST(β)) ➔ everything in FOLLOW(A) is in FOLLOW(B)

We apply these rules until nothing more can be added to any follow set

FOLLOW Example

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

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

Constructing LL(1) Parsing Table -- Algorithm

- for each production rule $A \rightarrow \alpha$ of a grammar G
  - for each terminal a in FIRST(α)

    ➔ add $A \rightarrow \alpha$ to M [A, a]

  - If ε is in FIRST(α)

  ➔ for each terminal a in FOLLOW(A) add $A \rightarrow \alpha$ to M [A, a]

  - If ε in FIRST(α) and \$ in FOLLOW(A)

  ➔ add $A \rightarrow \alpha$ to M [A, \$]

- All other undefined entries of the parsing table are error entries

Constructing LL(1) Parsing Table -- Example

$E \rightarrow TE'$	$FIRST(TE')=\{(,id\}$	➔ $E \rightarrow TE'$ into $M[E,(]$ and $M[E,id]$
$E' \rightarrow +TE'$	$FIRST(+TE')=\{+\}$	➔ $E' \rightarrow +TE'$ into $M[E',+]$
$E' \rightarrow \varepsilon$	$FIRST(\varepsilon)=\{\varepsilon\}$	➔ none
	but since ε in $FIRST(\varepsilon)$ and $FOLLOW(E')=\{\$,)\}$	➔ $E' \rightarrow \varepsilon$ into $M[E',\$]$ and $M[E',)]$
$T \rightarrow FT'$	$FIRST(FT')=\{(,id\}$	➔ $T \rightarrow FT'$ into $M[T,(]$ and $M[T,id]$
$T' \rightarrow *FT'$	$FIRST(*FT')=\{*\}$	➔ $T' \rightarrow *FT'$ into $M[T',*]$
$T' \rightarrow \varepsilon$	$FIRST(\varepsilon)=\{\varepsilon\}$	➔ none
	but since ε in $FIRST(\varepsilon)$ and $FOLLOW(T')=\{\$,),+\}$	➔ $T' \rightarrow \varepsilon$ into $M[T',\$]$, $M[T',)]$ and $M[T',+]$
$F \rightarrow (E)$	$FIRST((E))=\{(\}$	➔ $F \rightarrow (E)$ into $M[F,(]$
$F \rightarrow id$	$FIRST(id)=\{id\}$	➔ $F \rightarrow id$ into $M[F, id]$

LL(1) Grammars

- A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar

one input symbol used as a look-ahead symbol to determine parser action

LL(1) left most derivation

input scanned from left to right

- Parsing table of a grammar may contain more than one production rule in case of non- LL(1) grammar

A Grammar which is not LL(1)

$S \rightarrow i\,C\,t\,S\,E \;\;|\;\; a$
$E \rightarrow e\,S \;\;|\;\; \varepsilon$
$C \rightarrow b$

FOLLOW(S) = { \$,e }
FOLLOW(E) = { \$,e }
FOLLOW(C) = { t }

FIRST(iCtSE) = {i}
FIRST(a) = {a}
FIRST(eS) = {e}
FIRST(ε) = {ε}
FIRST(b) = {b}

	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iCtSE$		
E			$E \rightarrow e\,S$ $E \rightarrow \varepsilon$			$E \rightarrow \varepsilon$
C		$C \rightarrow b$				

two production rules for M[E,e]

Problem → ambiguity

A Grammar which is not LL(1) (cont.)

- Necessary steps that should be taken if the resulting parsing table contains multiply defined entries
  - Eliminate left recursion if it is not already done
  - Remove left factor if it is not already done
  - *If its (new grammar's) parsing table still contains multiply defined entries, that grammar is ambiguous or it is inherently not a LL(1) grammar*
- A left recursive grammar cannot be a LL(1) grammar
  - $A \rightarrow A\alpha \mid \beta$
    - ➔ any terminal that appears in FIRST(β) also appears in FIRST($A\alpha$) because $A\alpha \Rightarrow \beta\alpha$.
    - ➔ If β is ε, any terminal that appears in FIRST(α) also appears in FIRST($A\alpha$) and FOLLOW(A)
- A grammar that is not left factored cannot be a LL(1) grammar
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
    - ➔ any terminal that appears in FIRST($\alpha\beta_1$) also appears in FIRST($\alpha\beta_2$)
- An ambiguous grammar cannot be a LL(1) grammar

Properties of LL(1) Grammars

- A grammar G is LL(1) if and only if the following conditions hold for two distinctive production rules $A \rightarrow \alpha$ and $A \rightarrow \beta$

1. Both α and β cannot derive strings starting with same terminals
2. At most one of α and β can derive to ε
3. If β can derive to ε, then α cannot derive to any string starting with a terminal in FOLLOW(A)

Error Recovery in Predictive Parsing

- An error may occur in the predictive parsing (LL(1) parsing) due to
  - terminal symbol on the top of stack does not match with the current input symbol
  - top of stack is a non-terminal A, the current input symbol is a, and the parsing table entry M[A, a] is empty
- What should the parser do in an error case?
  - parser should be able to give an error message (as much meaningful as possible)
  - error should be recoverable, and it should be able to continue the parsing with the rest of the input

Error Recovery Techniques

- Panic-Mode Error Recovery
  - Skipping the input symbols until a synchronizing token is found
- Phrase-Level Error Recovery
  - Each empty entry in the parsing table is filled with a pointer to a specific error routine to take care that error case
- Error-Productions
  - If we have a good idea of the common errors that might be encountered, we can augment the grammar with productions that generate erroneous constructs
  - When an error production is used by the parser, we can generate appropriate *error diagnostics*
  - Since it is almost impossible to know all the errors that can be made by the programmers, this *method is not practical*
- Global-Correction
  - Ideally, we would like a compiler to make as few changes as possible in processing incorrect inputs
  - We have to globally analyze the input s to find the error
  - This is an expensive method, and it is not in practice

Panic-Mode Error Recovery in LL(1) Parsing

- Skip all the input symbols until a synchronizing token is found

- What is the synchronizing token?
  - All the terminal-symbols in the follow set of a non-terminal can be used as a synchronizing token set for that non-terminal

- Simple panic-mode error recovery for the LL(1) parsing:
  - For the empty entries
    - All the empty entries are marked as ***synch*** to indicate that the parser will skip all the input symbols until a symbol in the follow set of the non-terminal A appears in the input
    - Parser will pop the non-terminal A from the stack
    - Parsing continues from the state A

Panic-Mode Error Recovery in LL(1) Parsing

- For unmatched terminal symbols, parser
  - pops the unmatched terminal symbol from the stack
  - issues an error message saying that unmatched terminal is inserted

Panic-Mode Error Recovery - Example

$S \to AbS \ | \ e \ | \ \varepsilon$

$A \to a \ | \ cAd$

FOLLOW(S)={\$}

FOLLOW(A)={b,d}

	a	**b**	**c**	**d**	**e**	**\$**
S	$S \to AbS$	***sync***	$S \to AbS$	***sync***	$S \to e$	$S \to \varepsilon$
A	$A \to a$	***sync***	$A \to cAd$	***sync***	***sync***	***sync***

<u>**stack**</u>	<u>**input**</u>	<u>**output**</u>
\$S	aab\$	$S \to AbS$
\$SbA	aab\$	$A \to a$
\$Sba	aab\$	
\$Sb	ab\$	Error: missing b, inserted
\$S	ab\$	$S \to AbS$
\$SbA	ab\$	$A \to a$
\$Sba	ab\$	
\$Sb	b\$	
\$S	\$	$S \to \varepsilon$
\$	\$	accept

<u>**stack**</u>	<u>**input**</u>	<u>**output**</u>
\$S	ceadb\$	$S \to AbS$
\$SbA	ceadb\$	$A \to cAd$
\$SbdAc	ceadb\$	
\$SbdA	eadb\$	Error:unexpected e (illegal A)

(Remove all input tokens until first b or d, pop A)

\$Sbd	db\$	
\$Sb	b\$	
\$S	\$	$S \to \varepsilon$
\$	\$	accept

Phrase-Level Error Recovery

- Each empty entry in the parsing table is filled with a pointer to a special error routine that takes care of the error case

- These error routines may:
  - change, insert, or delete input symbols
  - issue appropriate error messages
  - pop items from the stack

- We should be careful when we design these error routines, because we may put the parser into an infinite loop