Compiler Construction : Exercises on Lexing and Parsing

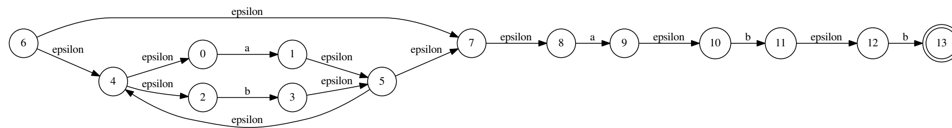
Timothy G. Griffin

January 31, 2021

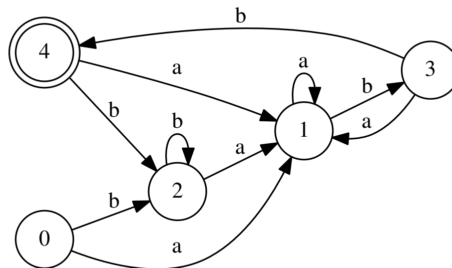
1 Regular language review

First, let's recall the NFA to DFA transformation.

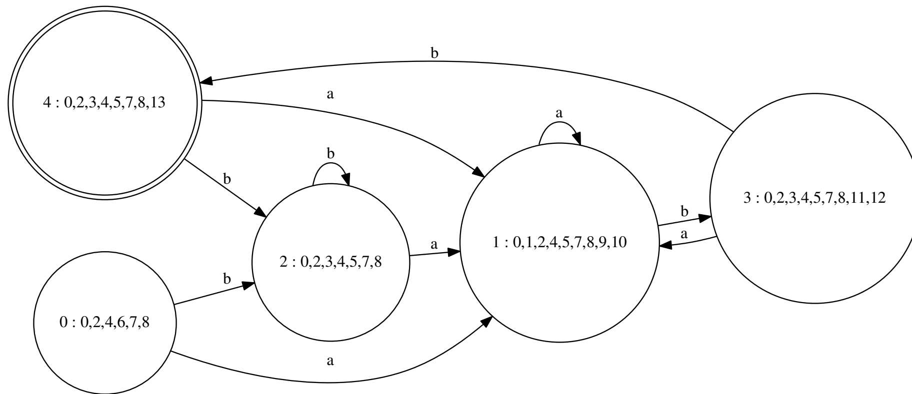
Here is an NFA for $(a+b)^*abb$. I used ϵ to connect NFAs in concatenation rather that smashing together final and start state, simply because it was easier to draw. (Here ϵ is represented as "epsilon" because I'm having trouble getting graphviz to digest latex.) The start state is state 6.



This can be transformed into the following DFA using the subset method. Here the start state is state 0.



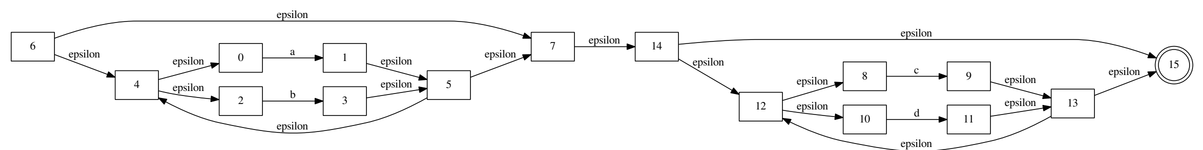
I can be easier to understand how the DFA was constructed if we label each state with the set of all NFA states used in the subset construction method:



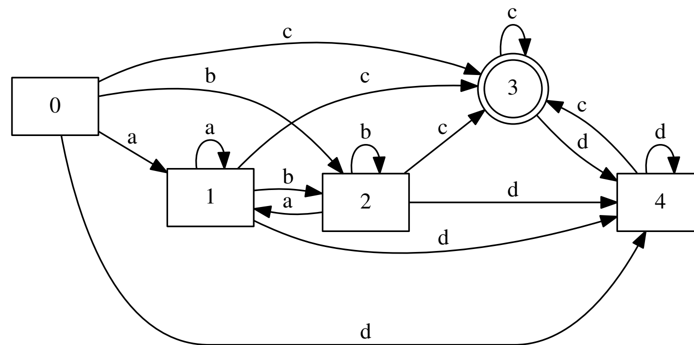
For example, let δ be the NFA transition function and δ' the derived DFA transition function. Then

$$\begin{aligned} q'_0 &= \epsilon - \text{closure}(\{6\}) &= \{0,2,4,6,7,8\} \\ \delta'(q'_0, b) &= \epsilon - \text{closure}(\{q \in \delta(q', b) | q' \in q'_0\}) &= \{0,2,3,4,5,7,8\} \end{aligned}$$

Here is another example. The NFA for $(a+b)^*(c+d)^*$:



Here is the resulting DFA:



1.1 Exercises

For each of these regular expressions

$$b(a+b)^*a$$
$$((\epsilon+a)b^*)^*$$

1.1 construct an NFA accepting the regular language.

1.2 construct a corresponding DFA.

1.3 Given any regular expression, can you produce a CFG that generates the same language? Can you do this for the above regular expressions?

2 Context-Free Grammars

Consider the following Context Free Grammar G_1 (taken from 2020 paper 4, question 4):

$$S \rightarrow Aa \mid BAb$$
$$A \rightarrow BB \mid c$$
$$B \rightarrow Sd \mid e$$

where $\{a,\ b, c,\ d,\ e\}$ is the set of terminal symbols.

Here is a left-most derivation of *ecadeb*,

$$\begin{array}{rcl} S & \Rightarrow & BAb \\ & \Rightarrow & eAb \\ & \Rightarrow & eBBb \\ & \Rightarrow & eSdBb \\ & \Rightarrow & eAadBb \\ & \Rightarrow & ecadBb \\ & \Rightarrow & ecadeb \end{array}$$

and a right-most derivation of *ecadeb*.

$$\begin{array}{rcl} S & \Rightarrow & BAb \\ & \Rightarrow & BBBb \\ & \Rightarrow & BBeb \\ & \Rightarrow & BSdeb \\ & \Rightarrow & BAadeb \\ & \Rightarrow & Bcadeb \\ & \Rightarrow & ecadeb \end{array}$$

If you draw the trees associated with these two derivations you will see they are the same. In fact, the grammar is not ambiguous, so this will always be the case.

2.1 Exercises

Consider the grammar

$$\begin{array}{rcl} T & \rightarrow & R \\ & | & aTc \\ R & \rightarrow & \epsilon \\ & | & RbR \end{array}$$

2.1 Give a leftmost derivation of *aabbbcc*.

2.2 Give a rightmost derivation of *aabbbcc*.

2.3 Is the grammar ambiguous? Justify your answer.

3 $FIRST$ and $FOLLOW$

Given a context-free grammar G we may want to automatically generate a top-down parser for G. This is not always possible, but for many simple grammars it is. Consider again grammar G_1.

Next we calculate $FIRST(\alpha)$ (for any $\alpha \in (N \cup T)^*$) and $FOLLOW(A)$ for any non-terminal A that is not S'. Recall that

$$\begin{array}{rcl} FIRST(\alpha) & = & \{a \in T | \exists \beta \in (N \cup T)^*, \alpha \Rightarrow^* a\beta\} \cup \{\epsilon | \alpha \Rightarrow^* \epsilon\} \\ FOLLOW(A) & = & \{a \in T | \exists \alpha, \beta \in (N \cup T)^*, S' \Rightarrow^* \alpha A a \beta\} \end{array}$$

Often with simple grammars, like G_1, we can easily derive $FIRST$ and $FOLLOW$ by inspection. For example,

$$\begin{aligned} FIRST(S) &= \{id,\ (\} \\ FOLLOW(S) &= \{+,\ \$\} \end{aligned}$$

Notice that if there are no ϵ productions (as is the case here), then the calculation of FIRST and FOLLOW can be simplified to the following.

FIRST: We first initialize the sets as follows.

1. For each terminal X set FIRST$(X) = \{X\}$.
2. For each production $X \to a\alpha$ where a is a terminal, add a to the set FIRST(X).

We then repeat this step until no more changes can be made:

- For each production $X \to Y\alpha$ where Y is a nonterminal, add all elements of FIRST(Y) to the set FIRST(X).

FOLLOW: We first initialize the sets as follows.

1. FOLLOW$(S) = \{\$\}$.
2. For each production $X \to \alpha Y \beta$ where Y is a nonterminal and $\beta \neq \epsilon$, add all elements of FIRST(Y') to the set FOLLOW(Y), where Y' is the first symbol of β.

We then repeat this step until no more changes can be made:

- For each production $X \to \alpha Y$ where Y is a nonterminal, add all elements of FOLLOW(Y) to the set FOLLOW(X).

Assume that we have added a new start production $S' \to S\$$ to the grammar. We can then calculate

$$\begin{aligned} \text{FIRST}(S) &= \{c,\ e\} \\ \text{FIRST}(A) &= \{c,\ e\} \\ \text{FIRST}(B) &= \{c,\ e\} \\ \\ \text{FOLLOW}(S) &= \{\$,\ d\} \\ \text{FOLLOW}(A) &= \{a,\ b\} \\ \text{FOLLOW}(B) &= \{a, b, c, e\} \text{(Note this is corrected from the original solution notes for this question)} \end{aligned}$$

3.1 Exercises

Consider this grammar G_2:

$$\begin{aligned} S &\to id \\ &\mid (S + id) \end{aligned}$$

3.1 Compute FIRST for the non-terminals of this grammar.

3.2 Compute FOLLOW for the non-terminals of this grammar.

4 $LL(1)$ Parsing

Given a context-free grammar G we may want to automatically generate a top-down parser for G. This is not always possible, but for many simple grammars it is.

Consider again grammar G_1. The grammar itself does not admit the top-down approach since it contains (indirect) left recursion:

$$S \rightarrow BAb \rightarrow SdAb.$$

and

$$S \rightarrow Aa \rightarrow BBa \rightarrow SdBa.$$

Note that this type of "indirect" left recursion was not discussed in lecture. In general, it is more difficult eliminating this type of left recursion and we will not go into this topic. Our conclusion is that the grammar G_1 cannot be parsed using the $LL(1)$ approach (however, the language generated by G_1 might be if we could eliminate the left recursion).

Consider the grammar of slide 22 of lecture 3:

```
S ::= if E then S else S
    | begin S L
    | print E
E ::= NUM = NUM
L ::= end
   | ; S L
```

First, let's translate this into more a more abstract specification:

$$\begin{array}{rcl} S & \rightarrow & iEtSeS \\ & | & bSL \\ & | & pE \\ E & \rightarrow & n = n \\ L & \rightarrow & t \\ & | & ;SL \end{array}$$

We can then compute $FIRST$ and $FOLLOW$:

$$\begin{array}{rcl} \text{FIRST}(S') & = & \{p,\ b,\ i\} \\ \text{FIRST}(S) & = & \{p,\ b,\ i\} \\ \text{FIRST}(E) & = & \{n\} \\ \text{FIRST}(L) & = & \{;,\ t\} \\ \text{FOLLOW}(S) & = & \{\$,\ ;,\ t,\ e\} \\ \text{FOLLOW}(E) & = & \{\$,\ ;,\ t,\ e\} \\ \text{FOLLOW}(L) & = & \{\$,\ ;,\ t,\ e\} \end{array}$$

The LL(1) parsing table M is computed as follows.

for all $A \in N,\ a \in T,\ M[A,a] \leftarrow$
for each $A \in N$
  for each production $A \rightarrow \alpha$
    if $a \in FIRST(\alpha)$ and $a \neq \epsilon$
    then $M[A,a] \leftarrow M[A,a] \cup \{A \rightarrow \alpha\}$
    else if $\epsilon \in FIRST(\alpha)$
      then for each $b \in FOLLOW(A)$
        $M[A,b] \leftarrow M[A,b] \cup \{A \rightarrow \alpha\}$

Here are the non-empty entries in table M for our example grammar:

$$\begin{array}{rcl} M[S,i] & = & \{S \rightarrow iEtSeS\} \\ M[S,p] & = & \{S \rightarrow pE\} \\ M[S,b] & = & \{S \rightarrow bSL\} \\ M[E,n] & = & \{E \rightarrow n = n\} \\ M[L,t] & = & \{L \rightarrow t\} \\ M[L,;] & = & \{L \rightarrow ; SL\} \end{array}$$

Suppose we are going to parse the following program fragment.

```
begin print 9=17; if 3=2 then print 4=3 else print 2=1 end
```

First, let's translated this into an abstract string looks more like a sequence of tokens produced by a lexer (for example, b for `begin` and n for an integer constant):

$$w = bpn = n; in = ntpn = nepn = nt$$

Now, using M we can now parse w:

input	stack	action
$bpn = n; in = ntpn = nepn = nt\$$	S'	$M[S', b] = \{S' \to S\$\}$
$bpn = n; in = ntpn = nepn = nt\$$	$S\$$	$M[S, b] = \{S \to bSL\}$
$bpn = n; in = ntpn = nepn = nt\$$	$bSL\$$	match
$pn = n; in = ntpn = nepn = nt\$$	$SL\$$	$M[S, p] = \{S \to pE\}$
$pn = n; in = ntpn = nepn = nt\$$	pEL	match
$n = n; in = ntpn = nepn = nt\$$	$EL\$$	$M[E, n] = \{E \to n = n\}$
$n = n; in = ntpn = nepn = nt\$$	$n = nL\$$	match
$= n; in = ntpn = nepn = nt\$$	$= nL\$$	match
$n; in = ntpn = nepn = nt\$$	$nL\$$	match
$; in = ntpn = nepn = nt\$$	$L\$$	$M[L, ;] = \{L \to ; SL\}$
$; in = ntpn = nepn = nt\$$	$; SL\$$	match
$in = ntpn = nepn = nt\$$	$SL\$$	$M[S, b] = \{S \to iEtSeS\}$
$in = ntpn = nepn = nt\$$	$iEtSeSL\$$	match
$n = ntpn = nepn = nt\$$	$EtSeSL\$$	$M[E, n] = \{E \to n = n\}$
$n = ntpn = nepn = nt\$$	$n = ntSeSL\$$	match
$= ntpn = nepn = nt\$$	$= ntSeSL\$$	match
$ntpn = nepn = nt\$$	$ntSeSL\$$	match
$tpn = nepn = nt\$$	$tSeSL\$$	match
$pn = nepn = nt\$$	$SeSL\$$	$M[S, p] = \{S \to pE\}$
$pn = nepn = nt\$$	$pEeSL\$$	match
$n = nepn = nt\$$	$EeSL\$$	$M[E, n] = \{E \to n = n\}$
$n = nepn = nt\$$	$n = neSL\$$	match
$= nepn = nt\$$	$= neSL\$$	match
$nepn = nt\$$	$neSL\$$	match
$epn = nt\$$	$eSL\$$	match
$pn = nt\$$	$SL\$$	$M[S, p] = \{S \to pE\}$
$pn = nt\$$	$pEL\$$	match
$n = nt\$$	$EL\$$	$M[E, n] = \{E \to n = n\}$
$n = nt\$$	$n = nL\$$	match
$= nt\$$	$= nL\$$	match
$nt\$$	$nL\$$	match
$t\$$	$L\$$	$M[L, t] = \{L \to t\}$
$t\$$	$t\$$	match
$\$$	$\$$	accept!

4.1 Exercises

Consider again the grammar G_2:

$$\begin{array}{rcl} S & \to & id \\ & \mid & (S + id) \end{array}$$

4.1 Construct the predictive parsing table using FIRST and FOLLOW.

4.2 Trace a parsing of $(((z + u) + y) + x)$.

5 SLR(1) Parsing

Consider the grammar G_3 for balanced parenthesis:

$$\begin{array}{rcl} S & \rightarrow & () \\ & | & (S) \\ & | & SS \end{array}$$

We can compute $FIRST$ and $FOLLOW$ for the (augmented) grammar as

$$\begin{array}{rcl} \text{FIRST}(S') & = & \{(\} \\ \text{FIRST}(S) & = & \{(\} \\ \text{FOLLOW}(S) & = & \{\$,\ (,\ )\} \end{array}$$

From this we can see that the LL(1) table M has a reduce-reduce-reduce conflict!

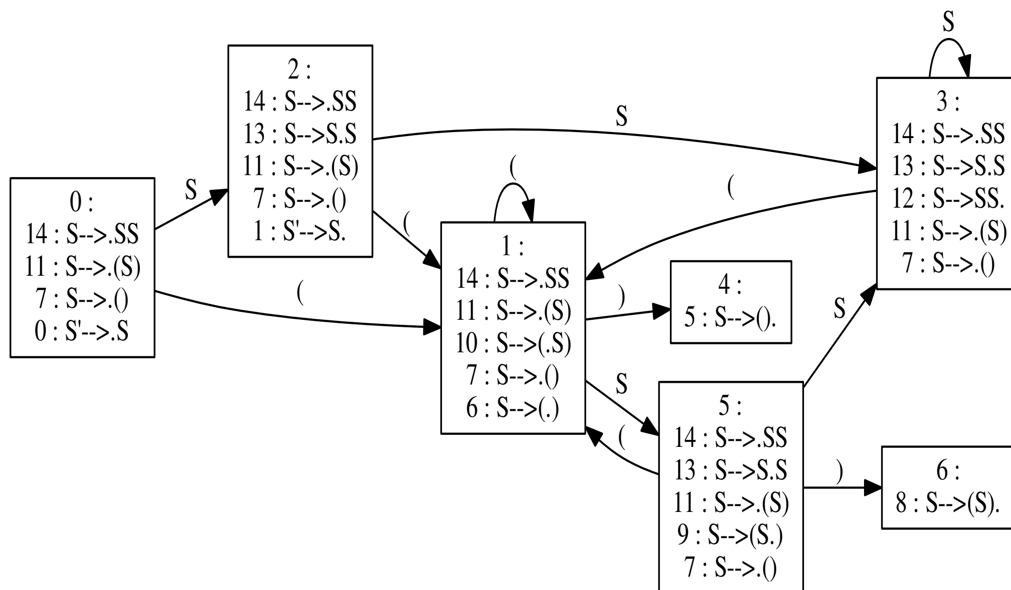
$$M[S, (] = \{S \rightarrow (),\ S \rightarrow (S),\ S \rightarrow SS\}$$

(This is left as an exercise.) So, top-down parsing won't work here.

Let's try SLR(1) parsing! Here are the steps in constructing a parser:

- Construct the NFA of LR(0) items.
- Convert the NFA into a DFA.
- Construct the SLR(1) $ACTION$ table (check for possible conflicts!).
- Construct the SLR(1) $GOTO$ table.
- Use the $ACTION$ and $GOTO$ tables to drive the generic LR(1) parser.

The DFA for this grammar is



The $ACTION$ table is computed as

$$\begin{array}{l} \text{If } A \rightarrow \alpha \bullet a\beta \in I_i \text{ and } \delta'(I_i, a) = I_j \\ \text{then } ACTION[i, a] = \text{shift } j \\ \\ \text{If } A \rightarrow \alpha \bullet \in I_i \text{ and } A \neq S' \\ \text{then for all } a \in FOLLOW(A) \\ \quad ACTION[i, a] = \text{reduce } A \rightarrow \alpha \end{array}$$

and the $GOTO$ table as $GOTO(i, A) = j$ whenever $\delta'(I_i, A) = I_j$.

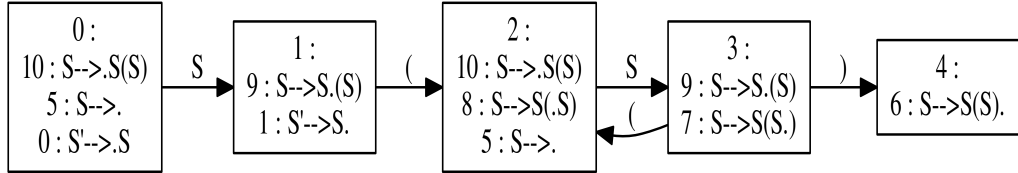
When we compute the $ACTION$ table for G_3 (left as an exercises) we see a problem: $ACTION[3, "("]'$ can be shift 1 or reduce $S \rightarrow SS$. So, we are confronted with a shift-reduce conflict! This is correct, since when seeing a left parenthesis we don't know which item to use as a guide: $S \rightarrow \bullet(S)$ or $S \rightarrow \bullet SS$.

It turns out that in this case it is the grammar that needs to be changed! We can write the grammar for balanced parenthesis as G_4:

$$\begin{array}{rcl} S & \rightarrow & \epsilon \\ & | & S(S) \end{array}$$

The language of this G_4 is the same as that of G_3 but in this case it can be parsed using the SLR(1) approach.

The corresponding DFA for G_4 is



Here is the $ACTION$ table for G_4

	\$	(	)
0	reduce $S \to \epsilon$	reduce $S \to \epsilon$	reduce $S \to \epsilon$
1	accept	shift 2	
2	reduce $S \to \epsilon$	reduce $S \to \epsilon$	reduce $S \to \epsilon$
3		shift 2	shift 4
4	reduce $S \to S(S)$	reduce $S \to S(S)$	reduce $S \to S(S)$

The $GOTO$ function can be read right from the DFA.

We are now ready to use the generic LR(1) parsing algorithm:

$a \leftarrow$ first symbol of input w
while(true)
  $s \leftarrow$ the state at top of stack
  if $ACTION[s, a] =$ shift t
  then push t on stack
    $a \leftarrow$ next symbol of input
  else if $ACTION[s, a] =$ reduce $A \to \beta$
    then pop $|\beta|$ states off the stack
      $t =$ state at top of stack
      push $GOTO[t, A]$ onto the stack
    else if $ACTION[s, a] =$ accept
      then accept and exit
      else ERROR

Here is a parse of $w = ((())())$. I've made the $GOTO$ steps explicit.

symbol stack	state stack	input	action
$\$$	0	$((())())\$$	reduce $S \to \epsilon$
$\$S$	0	$((())())\$$	push $GOTO[0, S] = 1$
$\$S$	$0, 1$	$((())())\$$	shift 2
$\$S($	$0, 1, 2$	$(())())\$$	reduce $S \to \epsilon$
$\$S(S$	$0, 1, 2$	$(())())\$$	push $GOTO[2, S] = 3$
$\$S(S$	$0, 1, 2, 3$	$(())())\$$	shift 2
$\$S(S($	$0, 1, 2, 3, 2$	$())())\$$	reduce $S \to \epsilon$
$\$S(S(S$	$0, 1, 2, 3, 2$	$())())\$$	push $GOTO[2, S] = 3$
$\$S(S(S$	$0, 1, 2, 3, 2, 3$	$())())\$$	shift 2
$\$S(S(S($	$0, 1, 2, 3, 2, 3, 2$	$))())\$$	reduce $S \to \epsilon$
$\$S(S(S(S$	$0, 1, 2, 3, 2, 3, 2$	$))())\$$	push $GOTO[2, S] = 3$
$\$S(S(S(S$	$0, 1, 2, 3, 2, 3, 2, 3$	$))())\$$	shift 4
$\$S(S(S(S)$	$0, 1, 2, 3, 2, 3, 2, 3, 4$	$)())\$$	reduce $S \to S(S)$
$\$S(S(S$	$0, 1, 2, 3, 2$	$)())\$$	push $GOTO[2, S] = 3$
$\$S(S(S$	$0, 1, 2, 3, 2, 3$	$)())\$$	shift 4
$\$S(S(S)$	$0, 1, 2, 3, 2, 3, 4$	$())\$$	reduce $S \to S(S)$
$\$S(S$	$0, 1, 2$	$())\$$	reduce $S \to S(S)$
$\$S(S$	$0, 1, 2$	$())\$$	push $GOTO[2, S] = 3$
$\$S(S$	$0, 1, 2, 3$	$())\$$	shift 2
$\$S(S($	$0, 1, 2, 3, 2$	$))\$$	Reduce $S \to \epsilon$
$\$S(S(S$	$0, 1, 2, 3, 2$	$))\$$	push $GOTO[2, S] = 3$
$\$S(S(S$	$0, 1, 2, 3, 2, 3$	$))\$$	Shift 4
$\$S(S(S)$	$0, 1, 2, 3, 2, 3, 4$	$)\$$	Reduce $S \to S(S)$
$\$S(S$	$0, 1, 2$	$)\$$	push $GOTO[2, S] = 3$
$\$S(S$	$0, 1, 2, 3$	$)\$$	Shift 4
$\$S(S)$	$0, 1, 2, 3, 4$	$\$$	reduce $S \to S(S)$
$\$S$	0	$\$$	push $GOTO[0, S] = 1$
$\$S$	$0, 1$	$\$$	Accept

5.1 Exercises

Consider the grammar G_5:

$$S \to (S)|()$$

4.1 Construct the LR(0) items for this grammar.

4.2 Construct the NFA with LR(0) items as states.

4.3 Construct the corresponding DFA.

4.4 Compute FIRST and FOLLOW for this grammar.

4.5 Construct the SLR(1) versions of ACTION and GOTO.

4.6 Trace the parsing of $((()))$.