Spis treści

6. Języki bezkontekstowe (klasa "2")

6.1. Gramatyki bezkontekstowe

- gramatyka $G_B = < N, \Sigma, P, S >$ nazywa się ***bezkontekstową*** (ang. *context-free*) wtw. każda produkcja $p \in P$ przyjmuje postać:

$$\mathbf{A} \rightarrow \mathbf{u}$$

gdzie $\mathbf{A} \in N$ i $\mathbf{u} \in (N \cup \Sigma)^*$

- ***wywód lewostronny*** (ang. *leftmost derivation*) $\Rightarrow^{*L}$ – ciąg form zdaniowych uzyskiwanych w trakcie wywodu z aksjomatu pewnego zdania x w ten sposób, że w każdym kroku stosuje się produkcję do skrajnie lewego symbolu nieterminalnego formy zdaniowej
- ***wywód prawostronny*** (ang. *rightmost-derivation*) $\Rightarrow^{*R}$ – ciąg form zdaniowych uzyskiwanych w trakcie wywodu z aksjomatu pewnego zdania x w ten sposób, że w każdym kroku stosuje się produkcję do skrajnie prawego symbolu nieterminalnego formy zdaniowej

- **przykład** (gramatyka bezkontekstowa G_{B1}; język prostych wyrażeń arytmetycznych)

$$G_{B1} = < N=\{E, T, F\}, \Sigma=\{a, b, +, *, (,)\}, P, E >$$

$$P = \{E \rightarrow E + T \mid T,\ T \rightarrow T * F \mid F,\ F \rightarrow a \mid b \mid (E)\ \}$$

– wywód lewostronny napisu **(a+b)*a**

$$E \Rightarrow^{L} T \Rightarrow^{L} T*F \Rightarrow^{L} F*F \Rightarrow^{L} (E)*F \Rightarrow^{L} (E+T)*F \Rightarrow^{L} (T+T)*F \Rightarrow^{L}$$

$$(F+T)*F \Rightarrow^{L} (a+T)*F \Rightarrow^{L} (a+F)*F \Rightarrow^{L} (a+b)*F \Rightarrow^{L} (a+b)*a$$

– wywód prawostronny napisu **(a+b)*a**

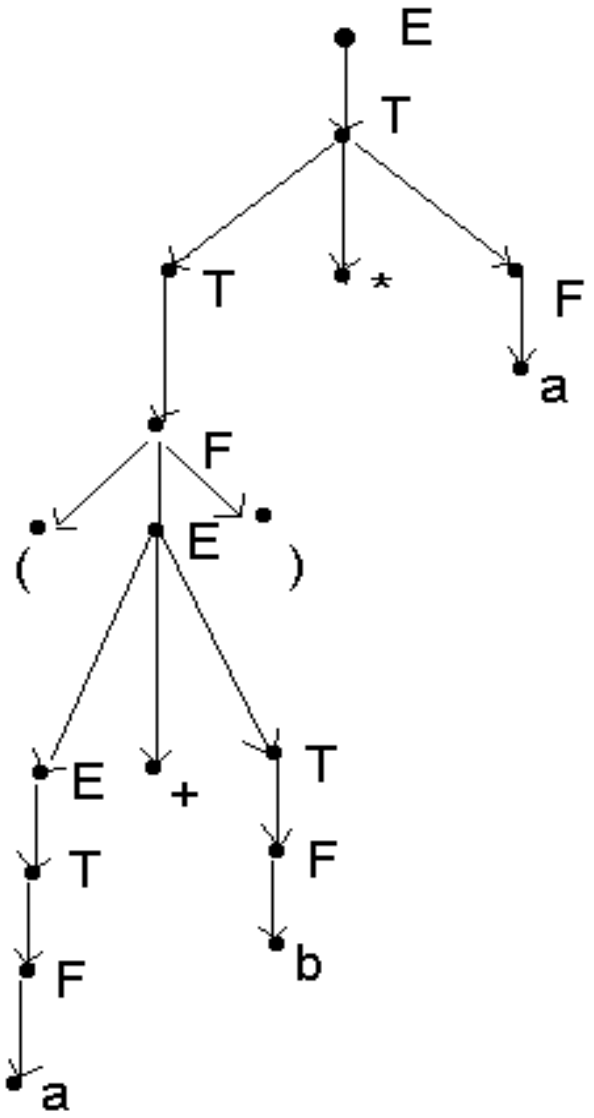
$$E \Rightarrow^{R} T \Rightarrow^{R} T*F \Rightarrow^{R} T*a \Rightarrow^{R} F*a \Rightarrow^{R} (E)*a \Rightarrow^{R} (E+T)*a \Rightarrow^{R}$$

$$(E+F)*a \Rightarrow^{R} (E+b)*a \Rightarrow^{R} (T+a)*a \Rightarrow^{R} (F+b)*a \Rightarrow^{R} (a+b)*a$$

- pojęcie ***drzewa wywodu*** T (ang. *derivation tree*):

  1) każdy wierzchołek T jest etykietowany symbolem ze zbioru $N \cup \Sigma \cup \{\varepsilon\}$
  2) etykietą korzenia jest aksjomat S
  3) każdy wierzchołek wewnętrzny jest etykietowany symbolem nieterminalnym
  4) jeżeli wierzchołek ma etykietę A, a jego wierzchołki potomne, uszeregowane od strony lewej ku prawej etykiety, odpowiednio, X_1, X_2, … X_n, to w zbiorze P musi istnieć produkcja $A \rightarrow X_1X_2 \ldots X_n$
  5) jeśli wierzchołek ma etykietę ze zbioru $\Sigma \cup \{\varepsilon\}$, to jest on liściem i jedynym potomkiem wierzchołka bezpośrednio nadrzędnego

- **<u>przykład</u>** (przedstawienie wywodu napisu **(a+b)*a** w postaci drzewa wywodu)



- jeśli ***napis x należy do L(G)***, to ma on co najmniej jedno ***drzewo wywodu***; każdemu drzewu wywodu odpowiada dokładnie jeden wywód prawostronny i dokładnie jeden wywód lewostronny

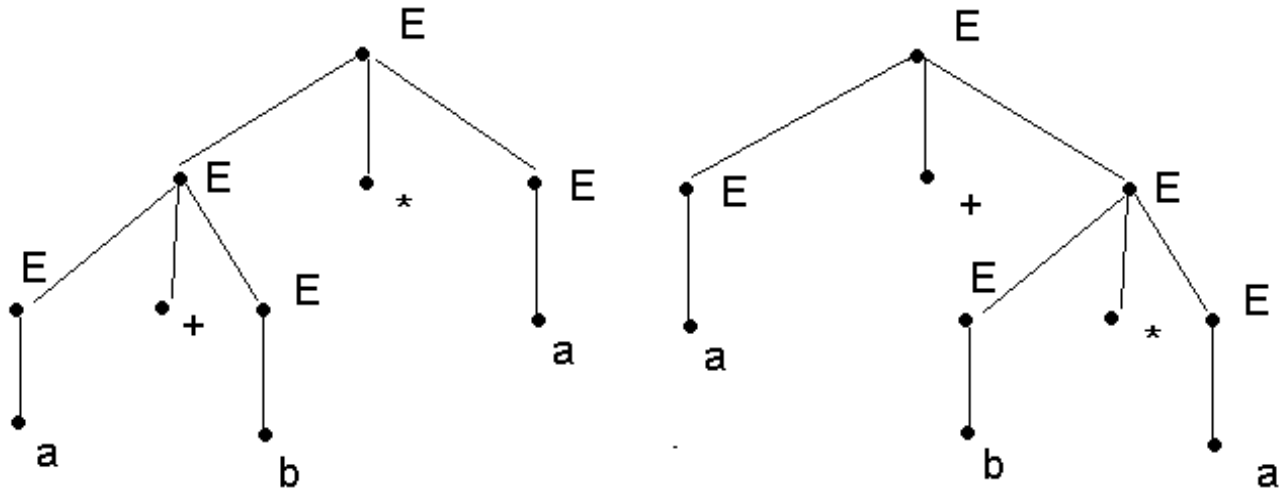
- gramatyka bezkontekstowa G_B nazywa się ***wieloznaczną*** (ang. *ambiguous*) wtw. w $L(G)$ istnieje co najmniej jedno słowo mające więcej niż jedno drzewo wywodu

- język bezkontekstowy J nazywa się ***jednoznacznym*** wtw. istnieje jednoznaczna gramatyka G_B, taka że $J = L(G_B)$ (w przeciwnym wypadku, język jest ***ściśle wieloznaczny***)

- **<u>przykład</u>** (gramatyka wieloznaczna G_{B2}, $L(G_{B2}) = L(G_{B1})$)

  $G_{B2} = < N=\{E\}, \Sigma=\{a, b, +, *, (,)\}, P, E >$

  $P = \{E \rightarrow E + E \mid E * E \mid a \mid b \mid (E) \}$



dwa drzewa wywodu tego samego napisu **a+b*a**

- ***notacja BNF*** (skr. ang. *Backus-Naur Form*) zapisu produkcji gramatyki bezkontekstowej

  <wyrażenie> ::= <wyrażenie> + <składnik> | <składnik>

  <składnik> ::= <składnik> * <czynnik> | <czynnik>

  <czynnik> ::= (<wyrażenie>) | a | b

- gramatyka bezkontekstowa jest ***ε-wolna*** wtw. :

  1) w P nie występują ε-produkcje (postaci $A \rightarrow \varepsilon$, gdzie A nie jest aksjomatem)
  2) istnieje dokładnie jedna ε-produkcja $S \rightarrow \varepsilon$, S jest aksjomatem i S nie występuje po prawej stronie jakiejkolwiek produkcji;

- gramatyka bezkontekstowa jest ***acykliczna*** wtw. gdy nie istnieją w niej wywody postaci $A \Rightarrow^{+} A$ (A jest nieterminalem)

- symbol nieterminalny A gramatyki G_B nazywamy rekurencyjnym wtw. $A \Rightarrow^{+} u\,A\,v$, gdzie $u, v \in (N \cup \Sigma)^*$; jeśli $u = \varepsilon$, to symbol jest lewostronnie rekurencyjny (i cała gramatyka jest ***lewostronnie rekurencyjna***); jeśli $v = \varepsilon$, to

symbol jest prawostronnie rekurencyjny (i cała gramatyka jest prawostronnie rekurencyjna);

- usuwanie lewostronnej rekurencyjności: jeśli w danej lewostronnie rekurencyjnej gramatyce bezkontekstowej $G_B = < N, \Sigma, P, S >$, w której istnieją produkcje postaci:

  $$\mathbf{A \rightarrow A\ x_1 \mid A\ x_2 \mid \ldots A\ x_m \mid y_1 \mid \ldots \mid y_n}$$

  gdzie $x_i , y_i \in (N \cup \Sigma)^*$ oraz y_i nie rozpoczyna się od A, zastąpimy rozważane produkcje przez:

  $$\mathbf{A \rightarrow y_1 \mid \ldots \mid y_n \mid y_1\ A' \mid y_2\ A' \mid \ldots\ y_n\ A'}$$

  $$\mathbf{A' \rightarrow x_1 \mid \ldots \mid x_m \mid x_1\ A' \mid x_2\ A' \mid \ldots\ x_m\ A'}$$

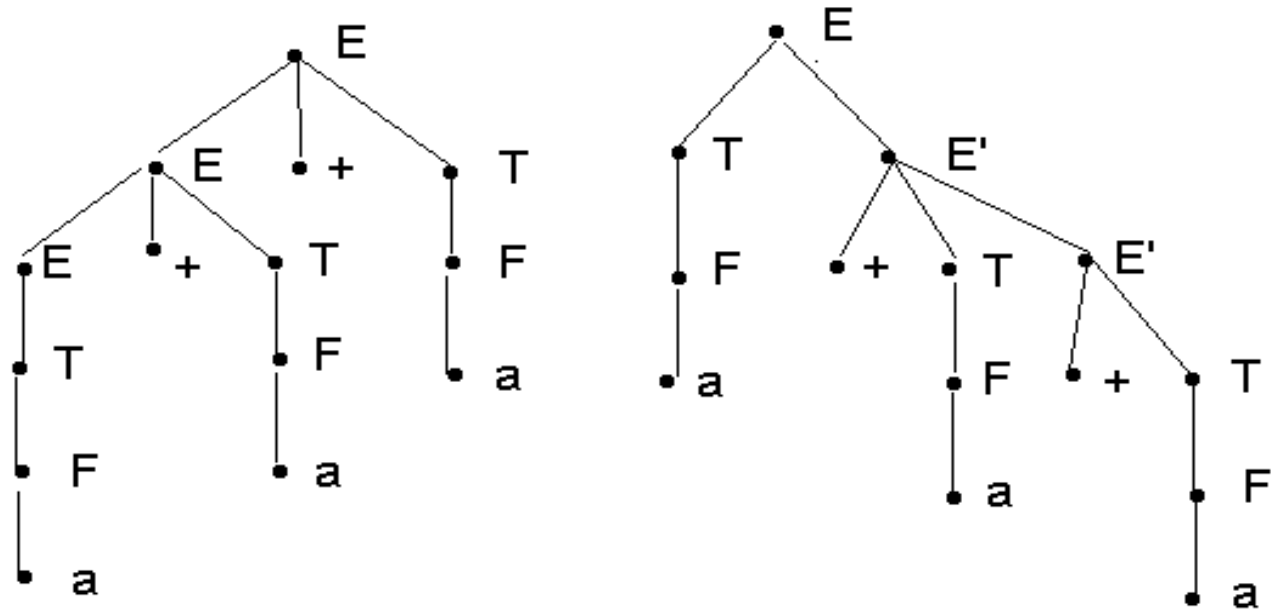
  gdzie A‘ jest nowym symbolem nieterminalnym (uzyskamy w ten sposób $G_B' = < N \cup \{A'\}, \Sigma, P', S >$), to $L(G_B) = L(G_B')$.

- **<u>przykład</u>** (utworzyć G_{B1}')

  G_{B1}' = < N={E, E', T, T', F}, Σ={a, b, +, *, (,)}, P, E >

  P = {E → T | T E', E' → + T | + T E', T → F | F T', T' →* F | * F T',

  F → a | b | (E) }



drzewa wywodu napisu **a+a+a** w G_{B1} i G_{B1}'

- Tw. (***lemat o pompowaniu dla języków bezkontekstowych***)

  Dla dowolnego języka bezkontekstowego $L(G_B)$ istnieje liczba naturalna p taka, że jeśli słowo $z \in L(G_B)$ i $|z| \geq p$, to $z = uvwxy$ oraz:

  1) $|vx| \geq 1$

  2) $|vwx| \leq p$

  3) każde słowo postaci $uv^iwx^iy \in L(G_B)$, $i \geq 0$.

6.2. Automaty ze stosem

- ***automatem niedeterministycznym ze stosem*** (ang. *pushdown automaton*) M_P nazywamy system:

$$M_P = < Q, \Sigma, \Gamma, \delta, q_0, Z_0, F >$$

Q – skończony zbiór ***stanów sterowania***

Σ – skończony alfabet wejściowy

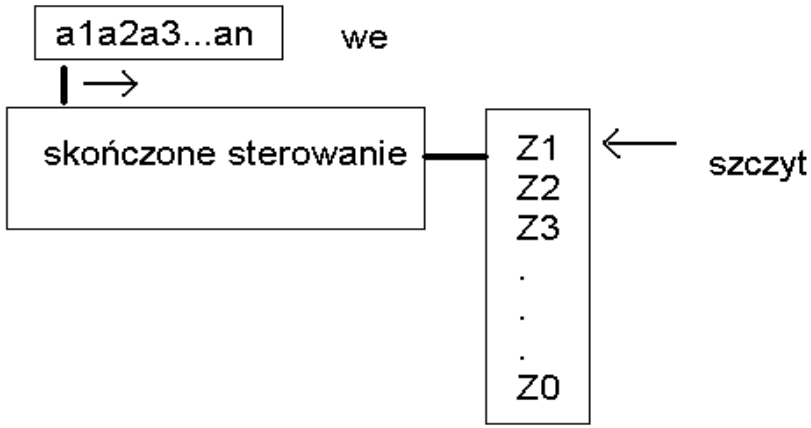
Γ – skończony zbiór ***symboli stosu***

$\delta : Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \rightarrow 2^{Q \times \Gamma^*}$ – ***funkcja przejść***

$q_0 \in Q$ – wyróżniony stan początkowy

$Z_0 \in \Gamma$ – znacznik dna stosu

$F \subseteq Q$ – zbiór stanów końcowych.



- ***konfiguracja automatu ze stosem***: $(q, w, \alpha) \in (Q \times \Sigma^* \times \Gamma^*)$, gdzie: q – bieżący stan sterowania

  w – słowo wejściowe do przeczytania

  α – słowo będące zawartością stosu

- relacja ***następstwa konfiguracji*** |– :

  $(q, aw, Z\alpha) \vdash (q', w, \gamma\alpha)$

  wtw. $(q', \gamma) \in \delta(q, a, Z)$, $\gamma \in \Gamma^*$, $a \in \Sigma \cup \{\varepsilon\}$

- ***konfiguracja początkowa*:** (q_0, w, Z_0), $w \in \Sigma^*$
- ***konfiguracja końcowa*:** (q, ε, α), $\alpha \in \Gamma^*$, $q \in F$
- słowo w jest ***akceptowane przez automat ze stosem*** M_P, wtw.

  $(q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha)$, $\alpha \in \Gamma^*$, $q \in F$

- język definiowany przez M_P, $L(M_P) = \{w |$ w jest akceptowane przez $M_P\}$

- **<u>przykład</u>** (akceptor języka 0^n1^n, $n \geq 0$)

  $M_{P1} = <\{q_0,q_1,q_2\}, \{0,1\}, \{Z_0, 0\}, \delta, q_0, Z_0, \{q_0\}>$

  $\delta(q_0, 0, Z_0) = \{(q_1, 0Z_0)\}$

  $\delta(q_1, 0, 0) = \{(q_1, 00)\}$

  $\delta(q_1, 1, 0) = \{(q_2, \varepsilon)\}$

  $\delta(q_2, 1, 0) = \{(q_2, \varepsilon)\}$

  $\delta(q_2, \varepsilon, Z_0) = \{(q_0, \varepsilon)\}$

  przejścia pomiędzy konfiguracjami:

  $(q_0, 0, Z_0) \vdash (q_1, \varepsilon, 0Z_0)$

  $(q_1, 0^i, 0Z_0) \vdash^i (q_1, \varepsilon, 0^{i+1}Z_0)$

  $(q_1, 1, 0^{i+1}Z_0) \vdash (q_2, \varepsilon, 0^iZ_0)$

  $(q_2, 1^i, 0^iZ_0) \vdash^i (q_2, \varepsilon, Z_0)$

  $(q_2, \varepsilon, Z_0) \vdash (q_2, \varepsilon, \varepsilon)$

  czyli $(q_0, 0^n1^n, Z_0) \vdash^{2n+1} (q_0, \varepsilon, \varepsilon)$ dla $n \geq 1$

  $(q_0, \varepsilon, Z_0) \vdash^0 (q_0, \varepsilon, Z_0)$ dla $n=0$;

- klasy języków generowanych gramatykami bezkontekstowymi i akceptowanych przez automaty niedeterministyczne ze stosem są sobie równoważne; przejścia pomiędzy formalizmami są efektywne
- ***automatem deterministycznym ze stosem*** (ang. *deterministic pushdown automaton*) M_{DP} nazywamy system:

$$\mathbf{M_{DP} = < Q, \Sigma, \Gamma, \delta, q_0, Z_0, F >}$$

$\mathbf{Q}$ – skończony zbiór stanów sterowania

Σ – skończony alfabet wejściowy

Γ – skończony zbiór symboli stosu

$\delta : \mathbf{Q} \times \Sigma \cup \{\varepsilon\} \times \Gamma \rightarrow \mathbf{Q} \times \Gamma^*$ – funkcja przejść

$\mathbf{q_0} \in \mathbf{Q}$ – wyróżniony stan początkowy

$\mathbf{Z_0} \in \Gamma$ – znacznik dna stosu

$\mathbf{F} \subseteq \mathbf{Q}$ – zbiór stanów końcowych.

Klasa języków akceptowanych za pomocą deterministycznych automatów ze stosem (***deterministyczne języki bezkontekstowe***) stanowi podklasę właściwą języków bezkontekstowych.

7. Deterministyczne języki bezkontekstowe i ich akceptory

7.1. Gramatyki i języki LR

- ***deterministyczne języki bezkontekstowe*** są generowane gramatykami typu LR i akceptowane ***deterministycznymi automatami ze stosem***
- gramatyki typu LR (i generowane nimi klasy języków) tworzą hierarchię (LR(0), LR(k), k ≥ 1)
- gramatyki LR(0) definiują klasę deterministycznych języków bezkontekstowych mających ***własność przedrostka***:
  a) dla dowolnego słowa **w** należącego do języka żaden przedrostek właściwy tego słowa nie należy do języka
  b) dla słowa abc, przedrostkami są: ε, a, ab i abc
  c) przedrostki właściwe, to: ε, a, ab
  d) nazwa gramatyk oznacza, że podczas wywodzenia słów w tej gramatyce, ciąg wejściowy jest przeglądany ***od strony lewej do prawej***, a tworzony wywód jest ***wywodem prawostronnym***, z podglądem wejścia na 0 symboli do przodu

- ***LR(0)-lokacją*** nazywamy produkcję z kropką wstawioną w dowolnym miejscu prawej strony, z początkiem i końcem włącznie, np.:

$$X \to \bullet Y\ Z\ A, \quad X \to Y \bullet Z\ A, \quad X \to Y\ Z \bullet A, \quad X \to Y\ Z\ A \bullet$$

są lokacjami dla produkcji $X \to Y\ Z\ A$, gdzie Y, Z i A są dowolnymi symbolami gramatyki

- ***prawostronna forma zdaniowa***, to forma zdaniowa uzyskana w efekcie wywodu prawostronnego uzyskanego za pomocą relacji $\Rightarrow^{*R}$

- ***uchwyt*** prawostronnej formy zdaniowej γ jest jej podłańcuchem takim, że $S \Rightarrow^{*R} \delta A w \Rightarrow^{R} \delta \beta w$ i $\delta \beta w = \gamma$ (uchwytem jest podłańcuch β, który można wyprowadzić w ostatnim kroku wywodu prawostronnego)

- ***istotnym przedrostkiem*** prawostronnej formy zdaniowej γ nazywamy dowolny przedrostek γ, który kończy się nie dalej na prawo niż prawy koniec uchwytu tej formy, np.:

  $$\mathbf{X} \Rightarrow^{R} \mathbf{Sc} \Rightarrow^{R} \mathbf{SAc} \Rightarrow^{R} \mathbf{SaSbc}$$

  SaSbc jest prawostronną formą zdaniową o uchwycie **aSb**; istotnymi przedrostkami są: ε, **S**, **Sa**, **SaS**, **SaSb**

- LR(0)-lokacja $\mathbf{A} \rightarrow \alpha \bullet \beta$ jest ***prawdziwa*** dla istotnego przedrostka γ, jeśli istnieje wywód prawostronny $\mathbf{S} \Rightarrow^{*R} \delta\mathbf{A}w \Rightarrow^{R} \delta\alpha\beta w$ i $\delta\alpha = \gamma$

- metoda akceptowania języka generowanego gramatyką LR(0) przez deterministyczny automat ze stosem zależy od znajomości zbioru lokacji prawdziwych dla każdego istotnego przedrostka; zbiór ten jest zbiorem regularnym, akceptowanym przez automat skończony

- niech $G_B = <N, \Sigma, P, S>$; ***automat niedeterministyczny*** rozpoznający istotne przedrostki $M = <Q, N \cup \Sigma, \delta, q_0, Q>$, gdzie Q jest zbiorem LR(0)-lokacji plus q_0; funkcja przejść:

  1) $\delta(q_0, \varepsilon) = \{S \rightarrow \bullet\ \alpha\}$

  2) $\delta(A \rightarrow \alpha \bullet B\beta, \varepsilon) = \{B \rightarrow \bullet\ \gamma$, gdzie $B \rightarrow \gamma$ jest produkcją$\}$

  3) $\delta(A \rightarrow \alpha \bullet X\beta, X) = \{ A \rightarrow \alpha X \bullet \beta$, gdzie X jest symbolem terminalnym$\}$.

- ***automat deterministyczny*** tworzymy wykorzystując dwie funkcje pomocnicze: **closure (I)** i **goto (I, X)**, gdzie **I** jest zbiorem LR(0)-lokacji, a **X –** symbolem gramatyki

- ***closure(I)*** definiujemy rekurencyjnie jako:

  a) zbiór **I** należy do closure(I)

  b) jeśli w zbiorze **I** jest lokacja z ***kropką przed nieterminalem***, to należy dołączyć lokacje dla wszystkich produkcji, których lewą stroną jest rozważany nieterminal, a kropka znajduje się na skrajnie lewej pozycji strony prawej; proces domykania stosujemy tak długo, jak długo powstają nowe lokacje

- ***goto(I, X)*** uzyskujemy poprzez domknięcie (closure) zbioru lokacji uzyskanych z **I** w ten sposób, że bierzemy pod uwagę lokacje z kropką przed symbolem **X** i tworzymy nowe lokacje, z kropką przesuniętą poza symbol **X**

- ***kolekcję zbiorów*** $\mathbf{C = \{s_0, s_1, \ldots, s_n\}}$ prawdziwych LR(0)-lokacji dla istotnych przedrostków gramatyki uzupełnionej G_B (z dodanym nowym aksjomatem S' i produkcją $S' \rightarrow S$) tworzymy zgodnie z algorytmem:

  1) $s_0 = \text{closure}(\{S' \rightarrow \bullet S\})$

  2) dla każdego s_i ze zbioru C i każdego symbolu X gramatyki takiego, że zbiór $\text{goto}(s_i, X)$ jest niepusty i nie należy jeszcze do C, dodaj $\text{closure}(\text{goto}(s_i, X))$; wykonuj tę operację tak długo, jak długo powstają nowe zbiory kolekcji C.

- **przykład** (uzupełniona gramatyka G_{B3} = <{S', S, A}, {(,),$}, P, S'>

  P = { S'→ S $, S→ S A | A, A → (S) | () })

  s_0 = { S'→ •S $, S→ •S A, S→ • A, A→ • (S), A→ • ( )}

  s_1 = goto(s_0, S)={ S'→ S • $, S→ S • A, A→ • (S), A→ • ( )}

  s_2 = goto(s_1, $)={ S'→ S $ •}

  s_3 = goto(s_0, A)={ S→ A •}

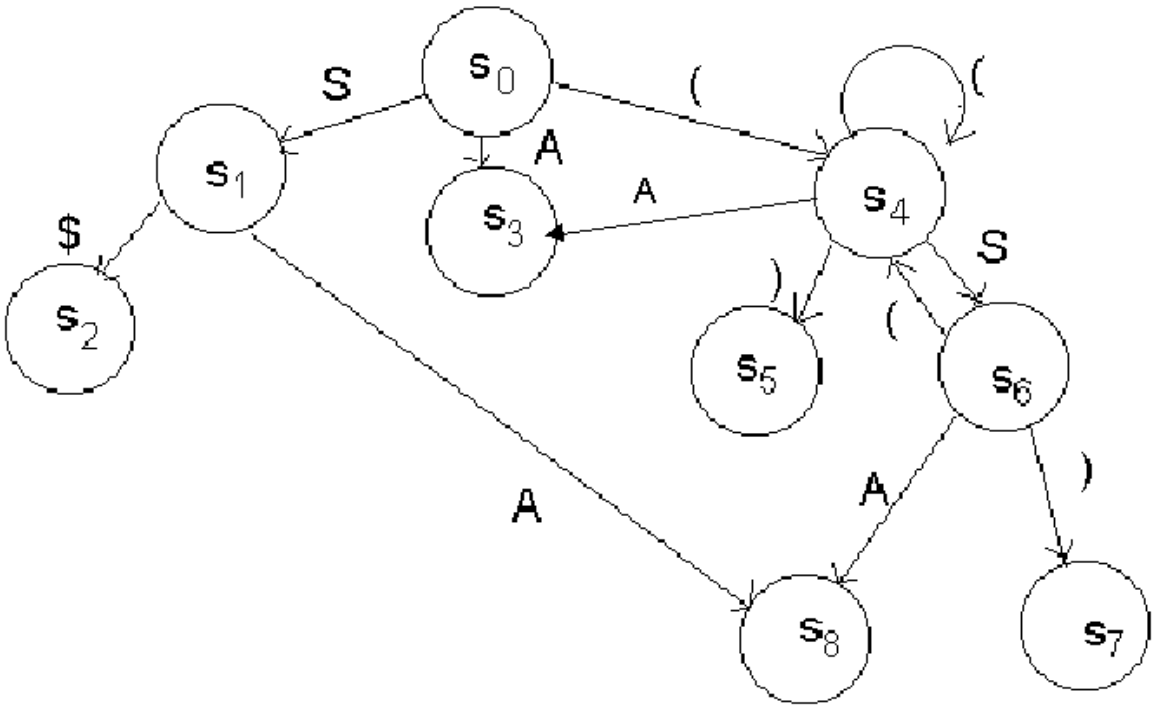
  s_4 = goto(s_0, **(**) ={ A→ (• S), A→ (•), S→ •S A, S→ • A,
  A→ • (S), A→ • ( )} = goto(s_4, **(**)

  s_5 = goto(s_4, **)**) ={ A→ ( ) •}

  s_6 = goto(s_4, S)={ A→ (S •), S→ S • A, A→ • (S), A→ • ( )}
  goto(s_4, A)=s_3
  goto(s_6, **(**) = s_4

  s_7 = goto(s_6, **)**) = { A→ (S) •}

  s_8 = goto(s_6, A) = {S→ S A •}
  goto(s_1, A) = s_8



- na podstawie kolekcji C można skonstruować ***deterministyczny automat ze stosem*** M_{DP} (automat ***redukuje słowo*** wejściowe ***do aksjomatu*** – odtwarza zatem ***odwrócony wywód prawostronny***, wykorzystując ***stos*** do przechowywania ***istotnego przedrostka*** aktualnej formy zdaniowej; strategia rozbioru zdania wejściowego typu „z dołu do góry” –„bottom-up”)

- ***konfiguracja***: **(q, α, w)**: q – stan sterowania, α – istotny przedrostek formy zdaniowej na stosie (zmodyfikowany tak, że ***nad*** każdym ***symbolem*** przedrostka znajduje się ***stan automatu*** rozpoznającego istotne przedrostki), **w** – aktualne słowo wejściowe

- jeśli przedrostek $\alpha = X_1X_2\ldots X_k$, to na stosie znajdzie się $s_0X_1s_1\ldots X_ks_k$ takie, że $s_i = \delta(q_0, X_1X_2\ldots X_i)$; stan s_k na szczycie stosu zawiera lokacje prawdziwe dla przedrostka $X_1X_2\ldots X_k$

- rozpatrywany ***automat ze stosem*** M_{DP} wykonuje akcje: ***shift*** (przesunięcie symbolu z wejścia na stos), ***reduce*** (zastąpienie ciągu symboli stosu, odpowiadającego prawej stronie produkcji gramatyki, symbolem lewej strony tej produkcji), ***accept*** (akceptacja ciągu wejściowego) i ***error*** (brak akceptacji ciągu wejściowego)

- akcja ***reduce***: jeśli stan s_k na szczycie zawiera lokację postaci $A \rightarrow \alpha \bullet$ (tzw. „lokację pełną”), to jest ona prawdziwa dla $X_1X_2\ldots X_k$, co oznacza, że α jest przyrostkiem $X_1X_2\ldots X_k$, np. postaci $X_{i+1}\ldots X_k$; istnieje także forma zdaniowa $X_1X_2\ldots X_k w$ (w – aktualne wejście), która podlega następującemu wywodowi prawostronnemu:

$$S \Rightarrow^{*R} X_1X_2\ldots X_iAw \Rightarrow^{R} X_1X_2\ldots X_{i+1}\ldots X_kw$$

Zatem, automat dokonuje przejścia:

$$(q_i, s_0X_1s_1\ldots s_{k-1}X_ks_k, w) \vdash^{*} (q_j, s_0X_1s_1\ldots s_{i-1}X_is_iAs, w)$$

gdzie $s = \delta(s_i, A)$

- akcja ***shift***: szczytowy stan s_k zawiera wyłącznie lokacje niepełne, to konieczne jest przesunięcie następnego symbolu wejściowego na stos:

$$(q, s_0X_1s_1\ldots s_{k-1}X_ks_k, ay) \vdash (q, s_0X_1s_1\ldots s_{k-1}X_ks_kas, y)$$

gdzie $s = \delta(s_k, a)$

- akcja ***accept***: jeśli szczytowy stan s_k zawiera ***jedynie*** lokację pełną **S'→ S $ •**, to automat akceptuje słowo wejściowe
- w pozostałych wypadkach – akcja ***error***
- **przykład** (ciąg konfiguracji akceptujących słowo (()())$)

LP	Stos	Wejście	Akcja wykonana
1	s_0	(()())$	
2	$s_0(s_4$	()())$	shift
3	$s_0(s_4(s_4$	)())$	shift
4	$s_0(s_4(s_4)s_5$	())$	shift
5	$s_0(s_4As_3$	())$	reduce A→()
6	$s_0(s_4Ss_6$	())$	reduce S→A
7	$s_0(s_4Ss_6(s_4$	))$	shift
8	$s_0(s_4Ss_6(s_4)s_5$	)$	shift
9	$s_0(s_4Ss_6As_8$	)$	reduce A→()
10	$s_0(s_4Ss_6$	)$	reduce S→S A
11	$s_0(s_4Ss_6)s_7$	$	shift
12	s_0As_3	$	reduce A→(S)
13	s_0Ss_1	$	reduce S→A
14	$s_0Ss_1\$s_2$		shift
15			reduce S'→S $ /accept

- ***gramatyka typu LR(0)***:

  a) aksjomat nie pojawia się po prawej stronie produkcji

  b) dla dowolnego istotnego przedrostka γ, jeśli $A \rightarrow \alpha\bullet$ jest pełną lokacją prawdziwą dla γ, to żadna inna lokacja pełna ani też żadna lokacja z symbolem terminalnym po prawej stronie kropki nie jest prawdziwa dla γ; niepełne lokacje mogą być prawdziwe, o ile nie ma prawdziwej lokacji pełnej

- ***gramatyki typu LR(k)*** – dodanie k symboli „podglądu” wejścia, w celu określenia zbioru napisów wejściowych, na których podstawie można dokonać redukcji; praktycznie $k=1$, gdyż każdy deterministyczny język bezkontekstowy ma gramatykę typu $LR(1)$; zwiększenie k nie rozszerza klasy definiowanych języków

- dla gramatyk typu LR(1) można generować automaty metodami SLR (ang. *Simple LR*), LALR (ang. *LookAhead LR*) i kanoniczną (metoda SLR wykorzystuje kolekcję LR(0)-lokacji, metody LALR i kanoniczna – kolekcje LR(1)-lokacji, w których dodatkowo uwzględnia się zbiory symboli „podglądu”)
  - **przykład** (uzupełniona gramatyka G_{B3} = <{S', S, A}, {(,),$}, P, S'>

    P = { S'→ S $, S→ S A | A, A → (S) | () })

    s_0 = { S'→ •S $, S→ •S A, S→ • A, A→ • (S), A→ • ( )}

    s_1 = goto(s_0, S)={ S'→ S • $, S→ S • A, A→ • (S), A→ • ( )}

    s_2 = goto(s_1, $)={ S'→ S $ •}

    s_3 = goto(s_0, A)={ S→ A •}

    s_4 = goto(s_0, **(**) ={ A→ (• S), A→ (•), S→ •S A, S→ • A,
    A→ • (S), A→ • ( )} = goto(s_4, **(**)

    s_5 = goto(s_4, **)**) ={ A→ ( ) •}

    s_6 = goto(s_4, S)={ A→ (S •), S→ S • A, A→ • (S), A→ • ( )}

goto(s_4, A)=s_3

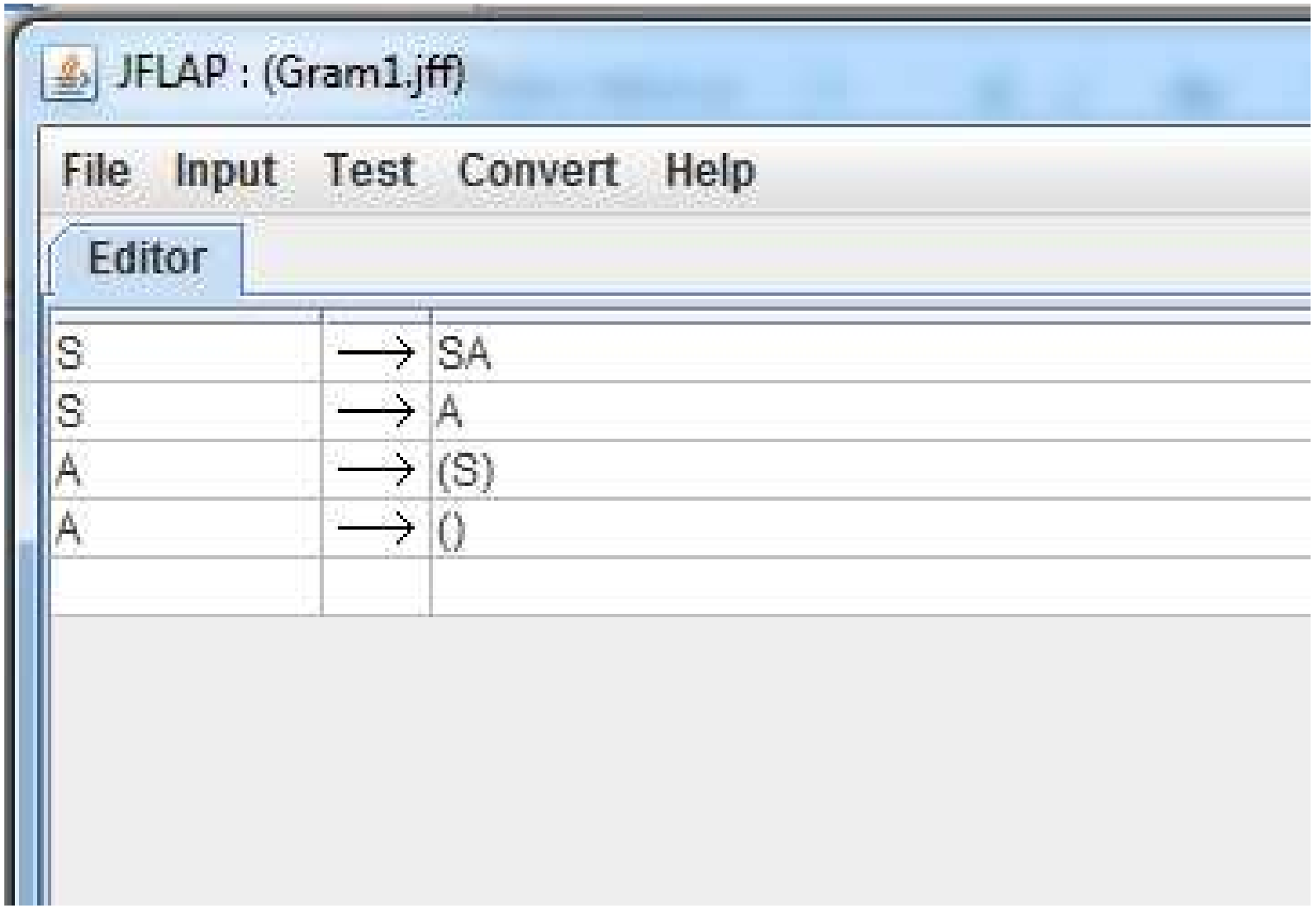
goto(s_6, **(**) = s_4

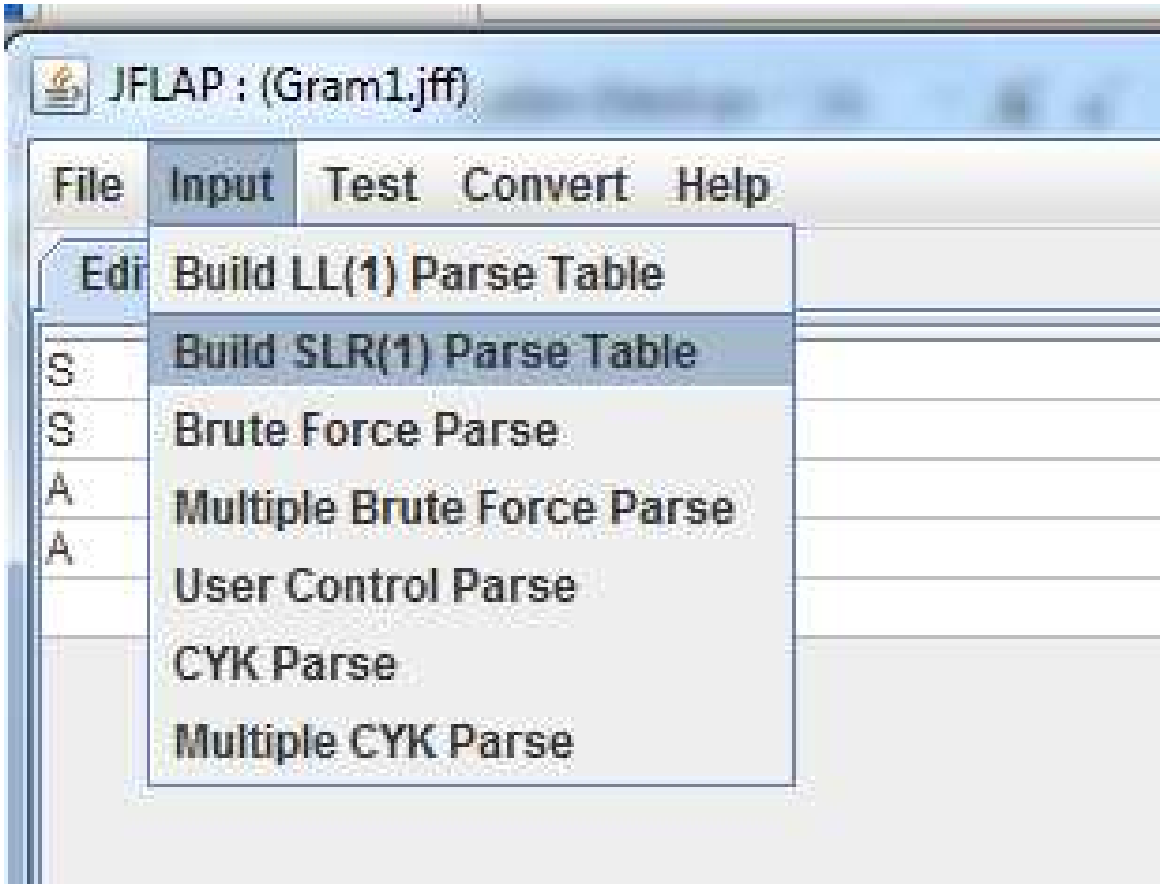
s_7 = goto(s_6, **)**) = { A→ (S) •}

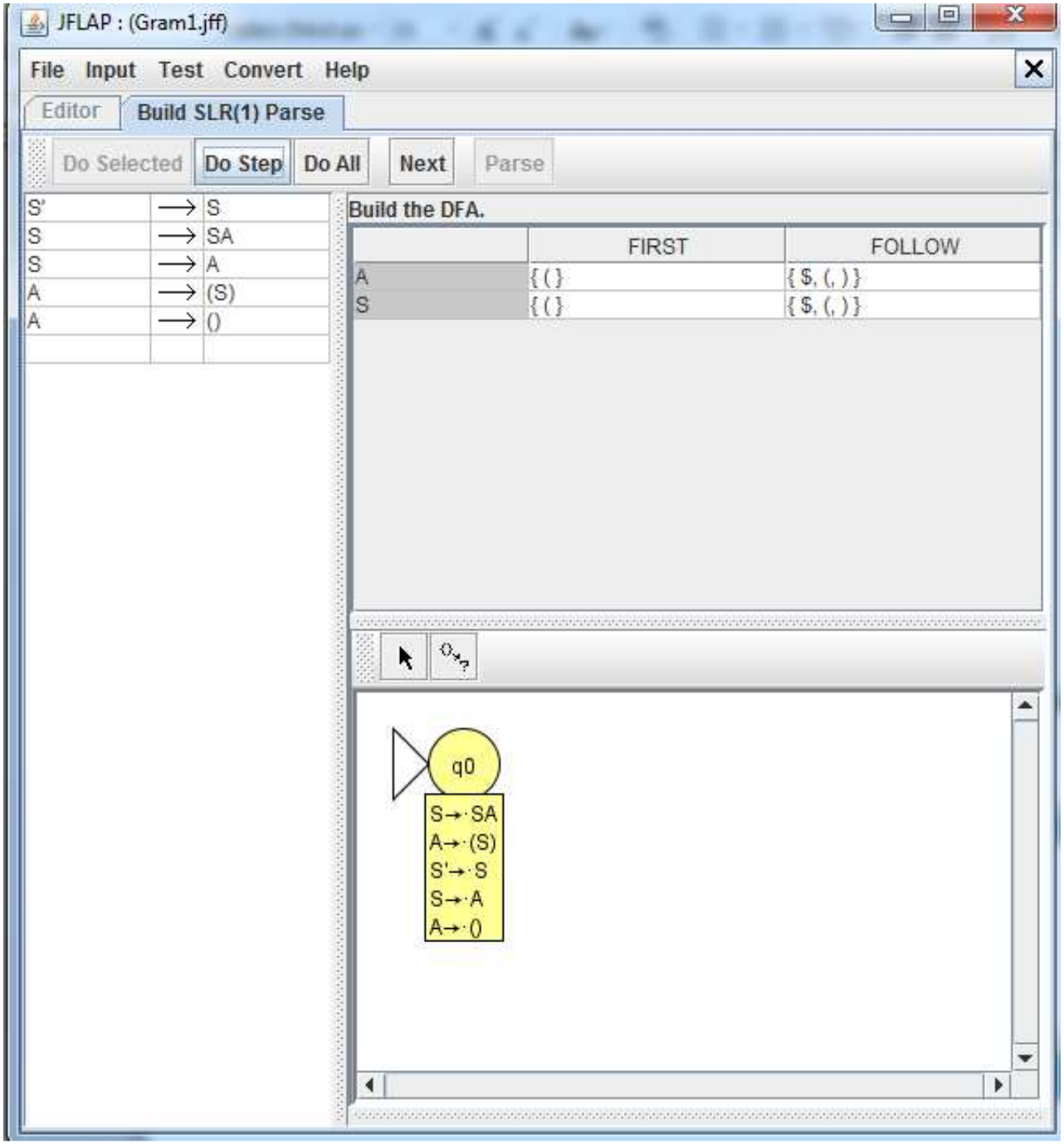
s_8 = goto(s_6, A) = {S→ S A •}

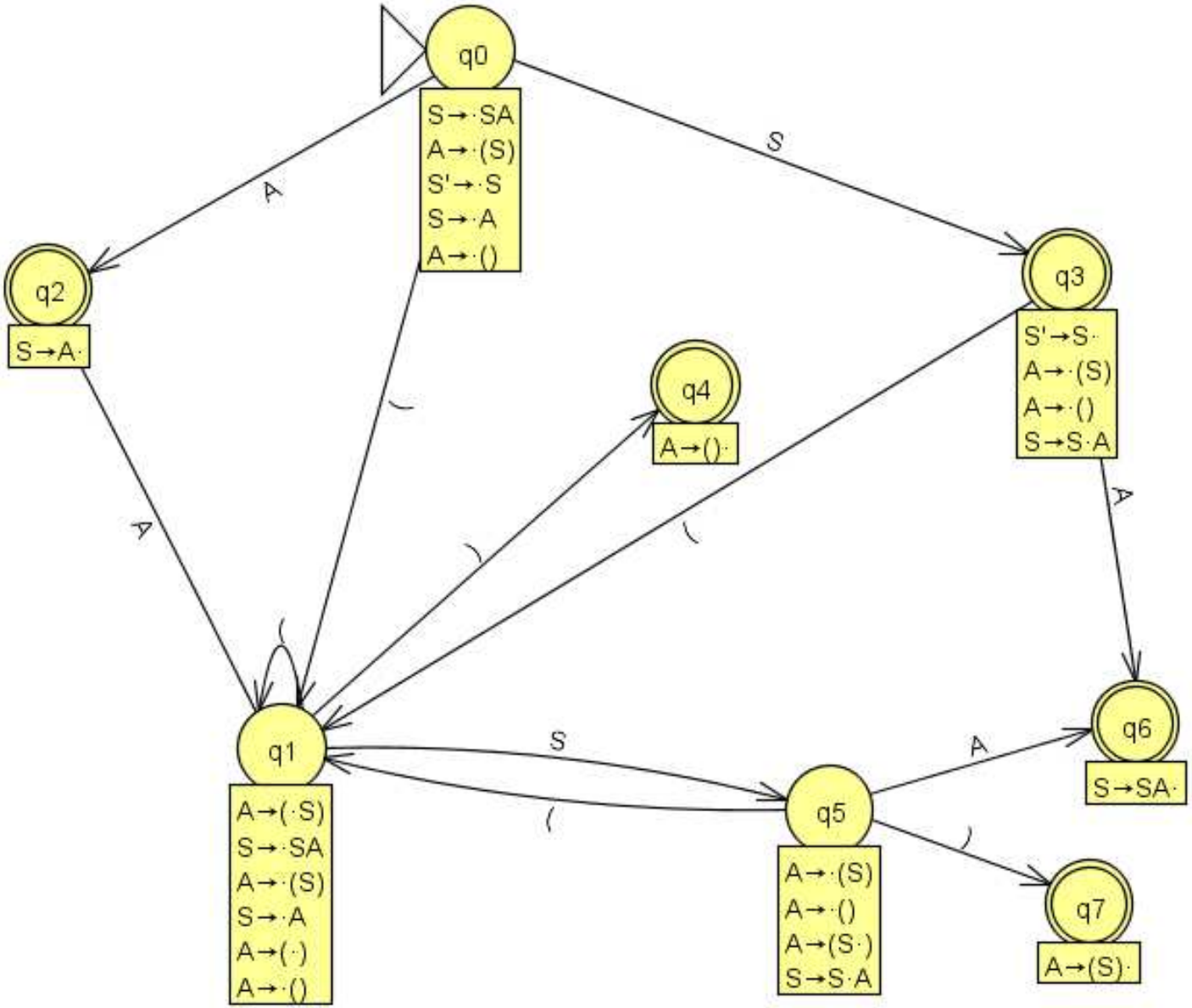
goto(s_1, A) = s_8

	ACTION			GOTO	
	(	)	$	S	A
s_0	s_4			1	3
s_1	s_4				8
s_2			acc		
s_3	r3	r3	r3		
s_4	s_4	s_5		6	3
s_5	r5	r5	r5		
s_6	s_4	s_7			8
s_7	r4	r4	r4		
s_8	r2	r2	r2		

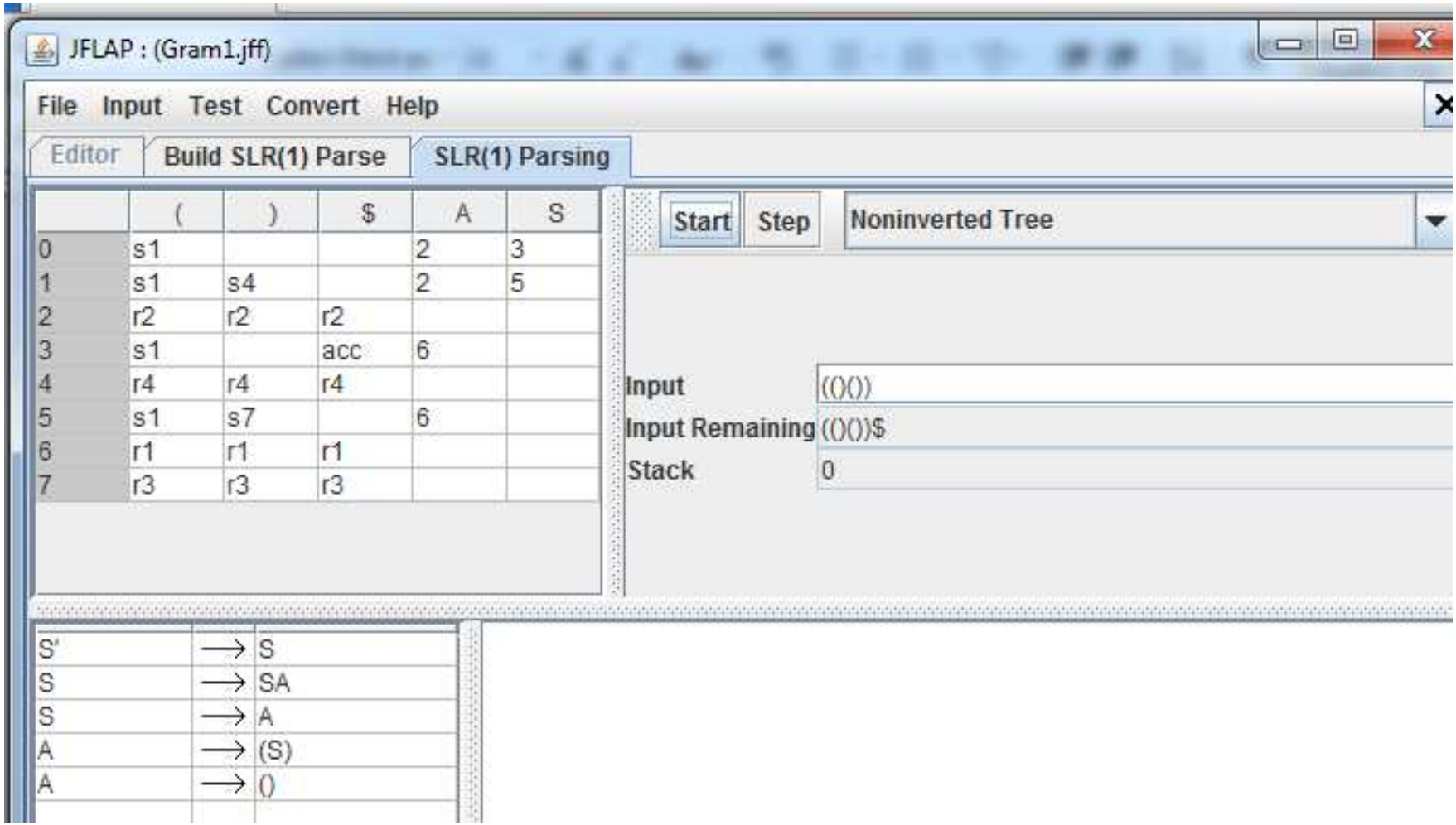

JFLAP : (Gram1.jff)
File Input Test Convert Help
Editor
S → SA
S → A
A → (S)
A → ()

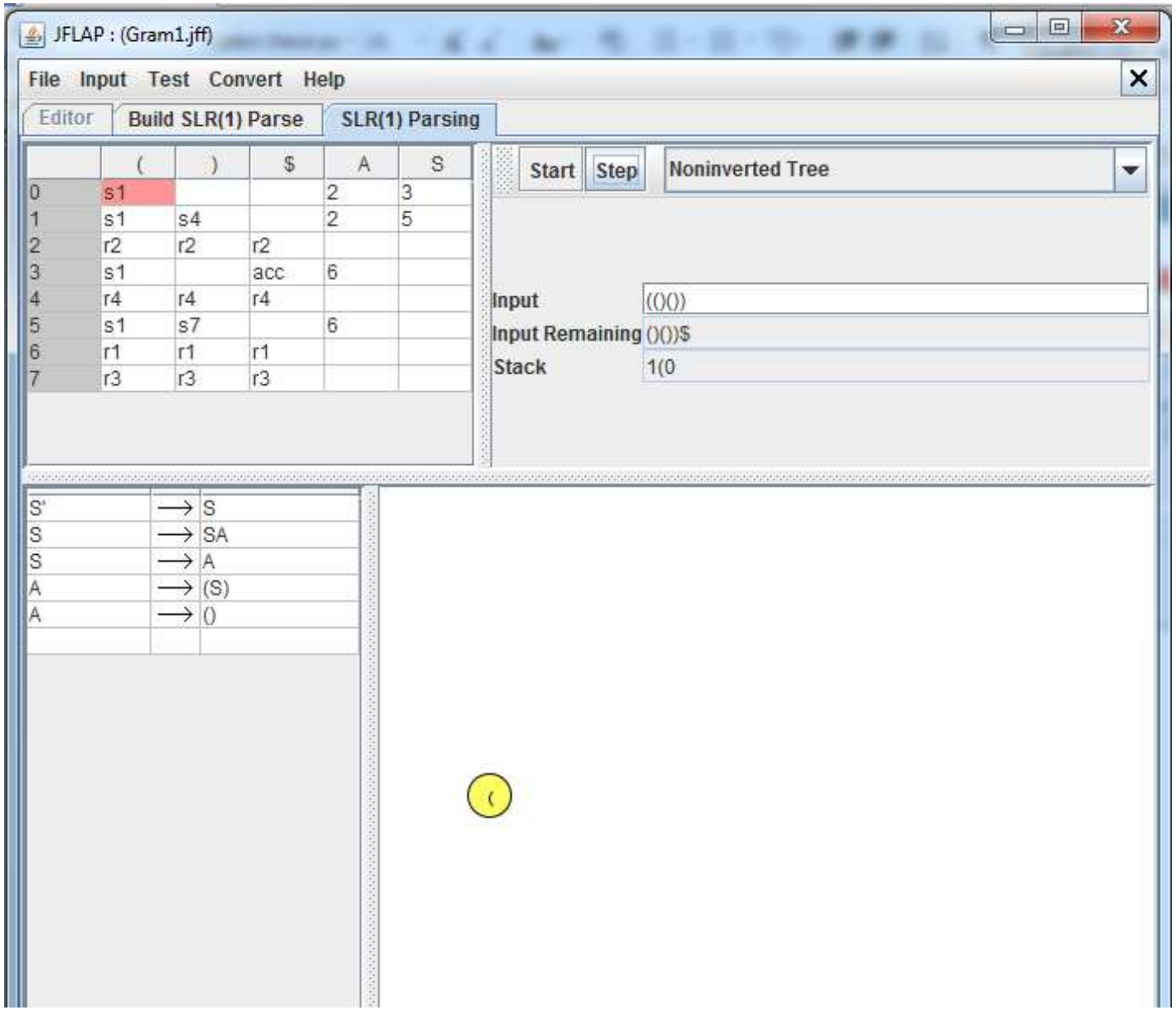

JFLAP : (Gram1.jff)
File
Input
Test
Convert
Help
Edi
Build LL(1) Parse Table
Build SLR(1) Parse Table
Brute Force Parse
Multiple Brute Force Parse
User Control Parse
CYK Parse
Multiple CYK Parse
S
S
A
A

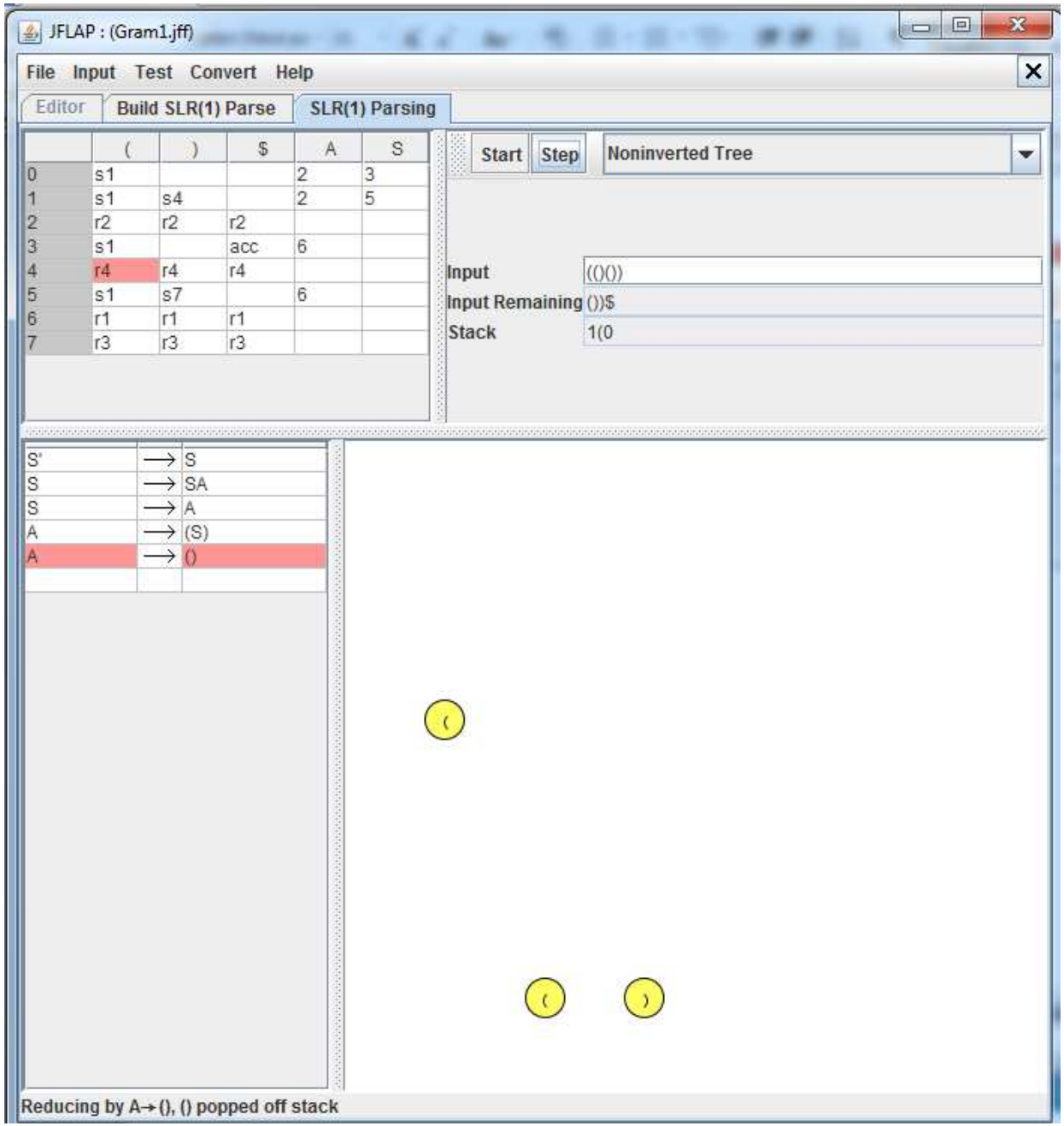

JFLAP : (Gram1.jff)
File Input Test Convert Help
Editor
Build SLR(1) Parse
Do Selected
Do Step
Do All
Next
Parse
S' ⟶ S
S ⟶ SA
S ⟶ A
A ⟶ (S)
A ⟶ ()
Build the DFA.
FIRST
FOLLOW
A { (} { $, (,) }
S { (} { $, (,) }
q0
S→·SA
A→·(S)
S'→·S
S→·A
A→·()

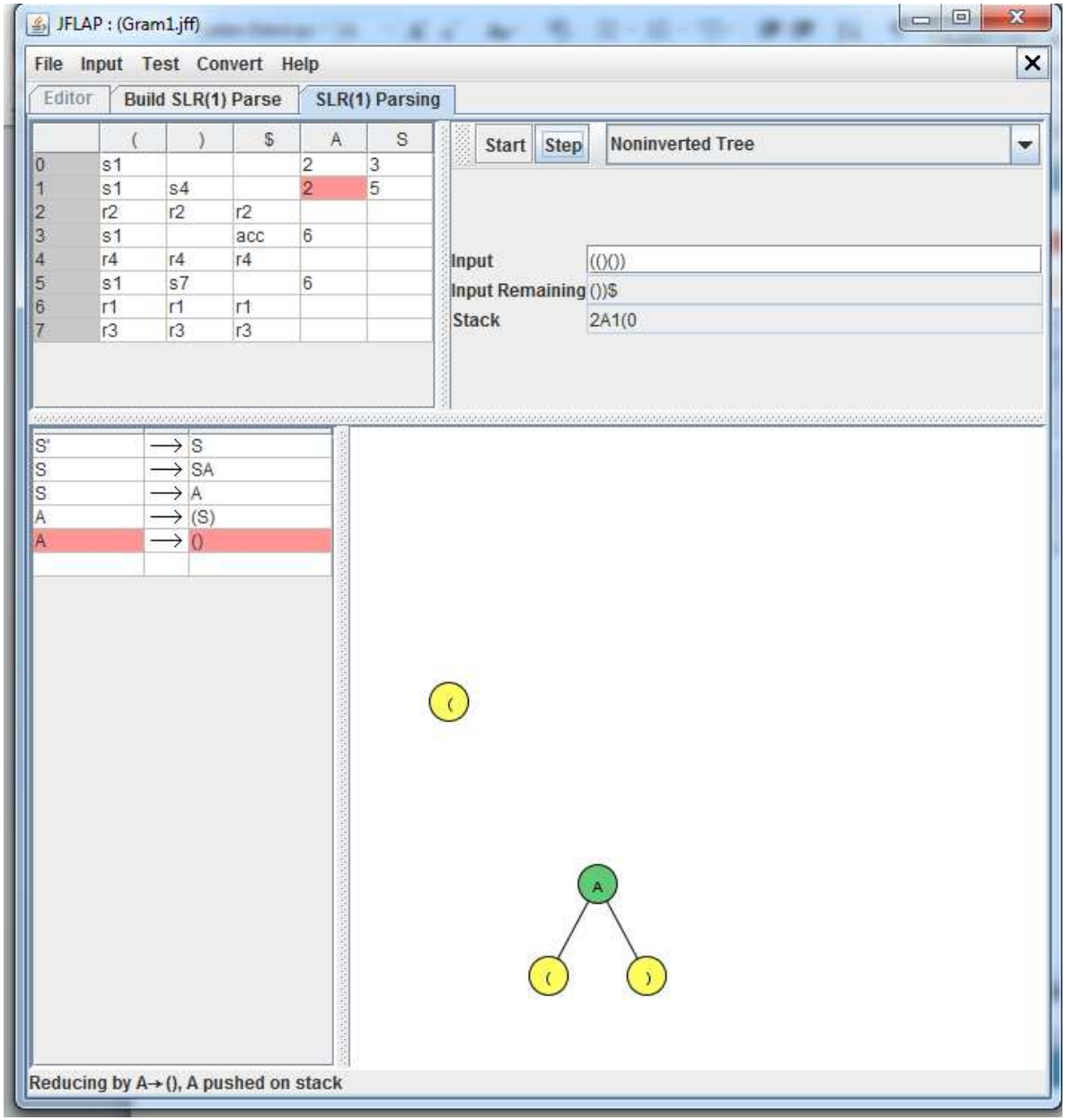

q0
S→·SA
A→·(S)
S'→·S
S→·A
A→·()
A
S
q2
S→A·
q3
S'→S·
A→·(S)
A→·()
S→S·A
q4
A→()·
(
A
)
(
A
q1
A→(·S)
S→·SA
A→·(S)
S→·A
A→(·)
A→·()
S
(
q5
A→·(S)
A→·()
A→(S·)
S→S·A
A
)
q6
S→SA·
q7
A→(S)·

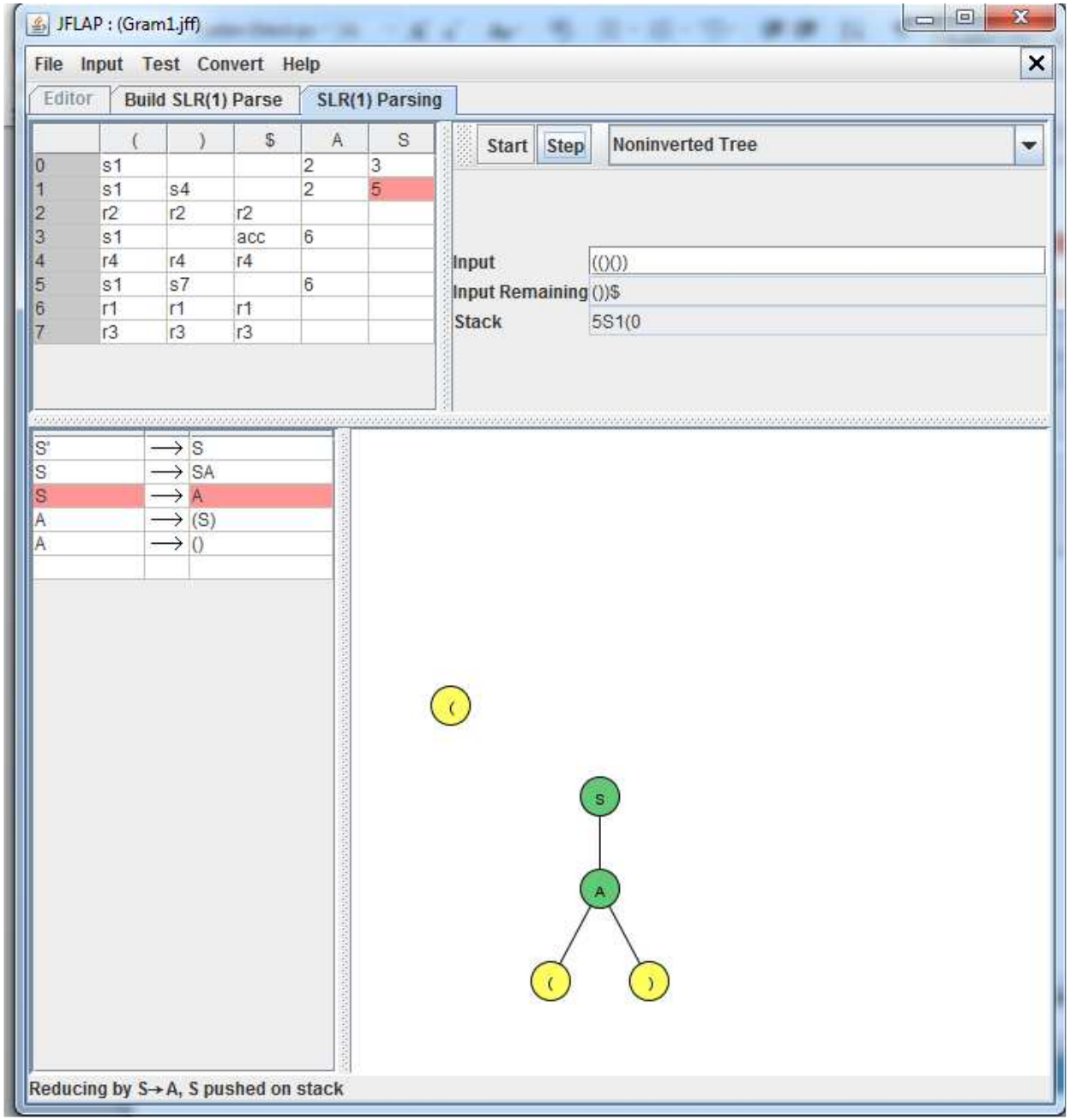
	(	)	$	A	S
0	s1			2	3
1	s1	s4		2	5
2	r2	r2	r2		
3	s1		acc	6	
4	r4	r4	r4		
5	s1	s7		6	
6	r1	r1	r1		
7	r3	r3	r3		

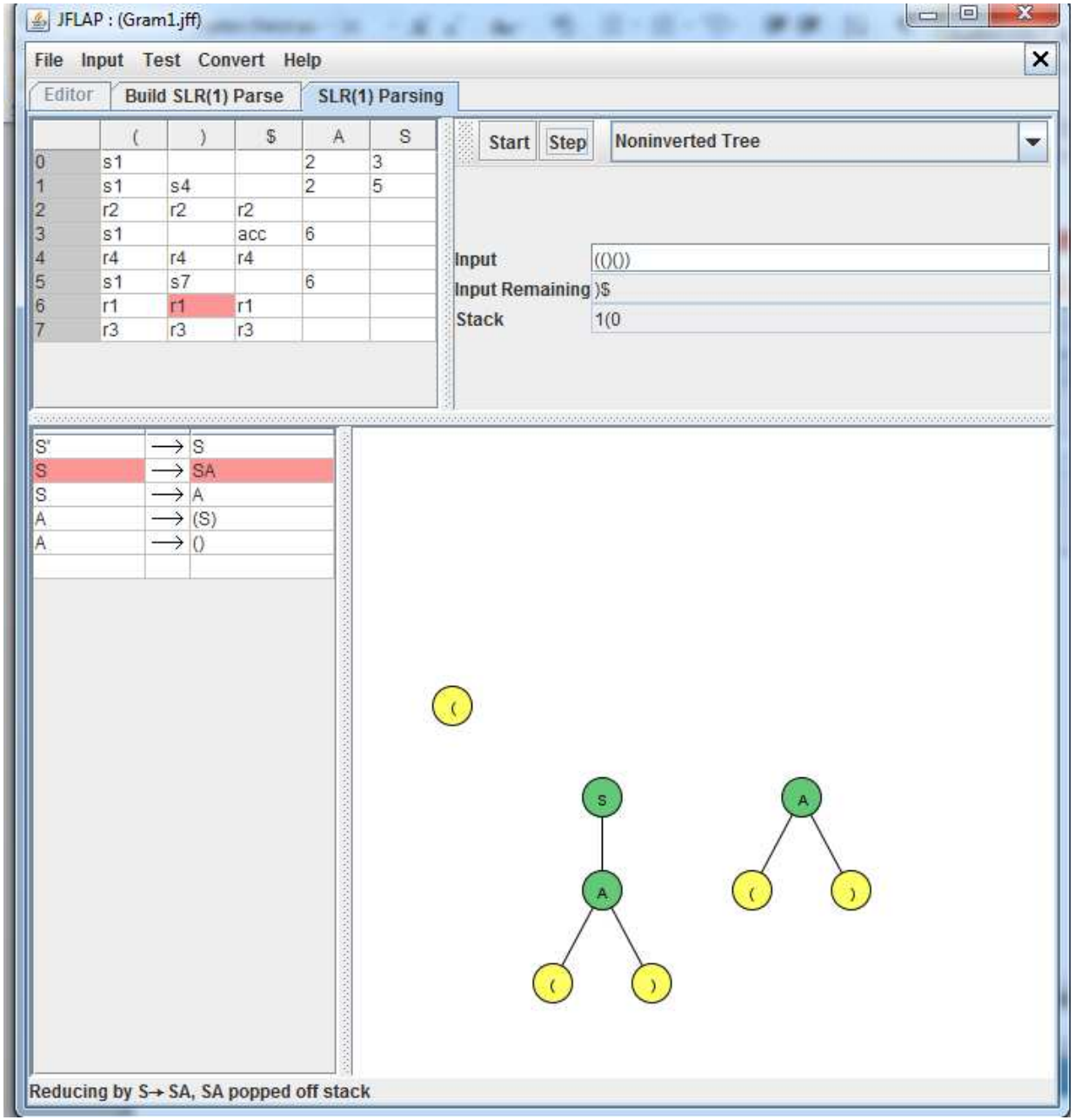

JFLAP : (Gram1.jff)
File Input Test Convert Help
Editor
Build SLR(1) Parse
SLR(1) Parsing
( ) $ A S
0 s1 2 3
1 s1 s4 2 5
2 r2 r2 r2
3 s1 acc 6
4 r4 r4 r4
5 s1 s7 6
6 r1 r1 r1
7 r3 r3 r3
Start
Step
Noninverted Tree
Input (()())
Input Remaining (()())$
Stack 0
S' → S
S → SA
S → A
A → (S)
A → ()

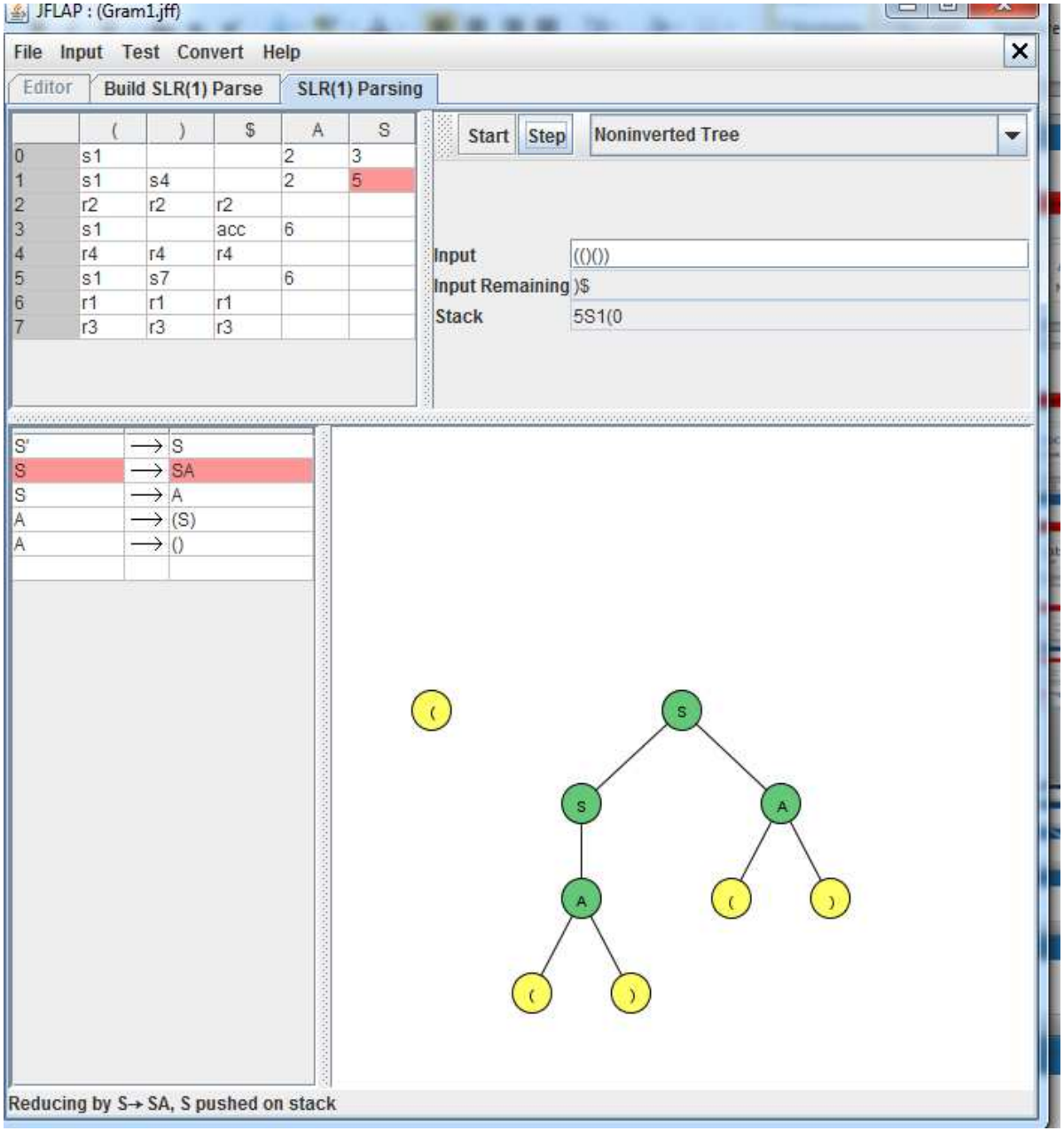

JFLAP : (Gram1.jff)
File Input Test Convert Help
Editor
Build SLR(1) Parse
SLR(1) Parsing
( ) $ A S
0 s1 2 3
1 s1 s4 2 5
2 r2 r2 r2
3 s1 acc 6
4 r4 r4 r4
5 s1 s7 6
6 r1 r1 r1
7 r3 r3 r3
Start
Step
Noninverted Tree
Input (()())
Input Remaining ()())$
Stack 1(0
S' → S
S → SA
S → A
A → (S)
A → ()
(

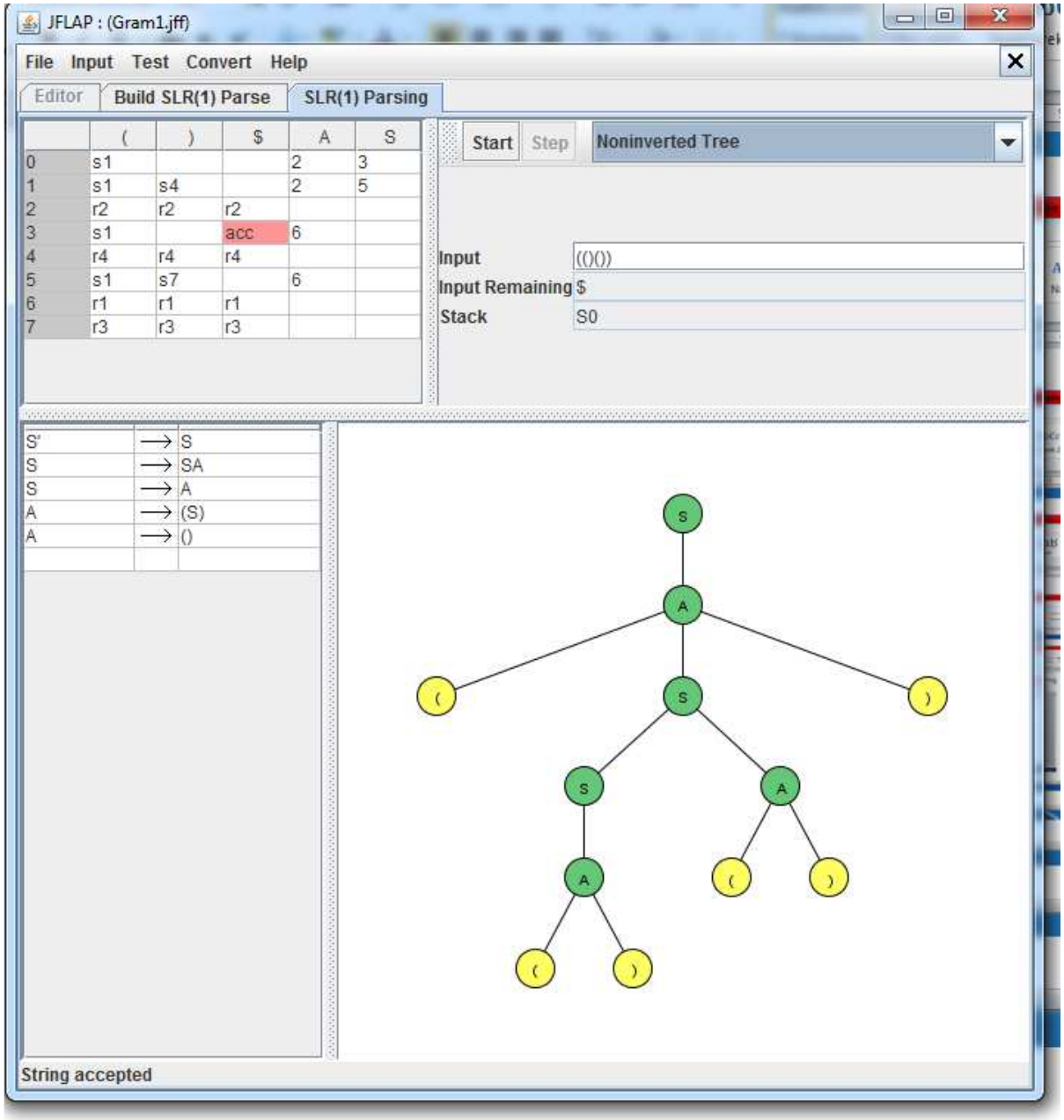

JFLAP : (Gram1.jff)
File Input Test Convert Help
Editor
Build SLR(1) Parse
SLR(1) Parsing
( ) $ A S
0 s1 2 3
1 s1 s4 2 5
2 r2 r2 r2
3 s1 acc 6
4 r4 r4 r4
5 s1 s7 6
6 r1 r1 r1
7 r3 r3 r3
Start
Step
Noninverted Tree
Input (()())
Input Remaining ())$
Stack 1(0
S' → S
S → SA
S → A
A → (S)
A → ()
(
(
)
Reducing by A→(), () popped off stack


JFLAP : (Gram1.jff)
File Input Test Convert Help
Editor
Build SLR(1) Parse
SLR(1) Parsing
( ) $ A S
0 s1 2 3
1 s1 s4 2 5
2 r2 r2 r2
3 s1 acc 6
4 r4 r4 r4
5 s1 s7 6
6 r1 r1 r1
7 r3 r3 r3
Start
Step
Noninverted Tree
Input (()())
Input Remaining ())$
Stack 2A1(0
S' → S
S → SA
S → A
A → (S)
A → ()
(
A
( )
Reducing by A→(), A pushed on stack


JFLAP : (Gram1.jff)
File Input Test Convert Help
Editor Build SLR(1) Parse SLR(1) Parsing
( ) $ A S
0 s1 2 3
1 s1 s4 2 5
2 r2 r2 r2
3 s1 acc 6
4 r4 r4 r4
5 s1 s7 6
6 r1 r1 r1
7 r3 r3 r3
Start Step Noninverted Tree
Input (()())
Input Remaining ())$
Stack 5S1(0
S' ⟶ S
S ⟶ SA
S ⟶ A
A ⟶ (S)
A ⟶ ()
(S A ( )
Reducing by S→A, S pushed on stack


JFLAP : (Gram1.jff)
File Input Test Convert Help
Editor Build SLR(1) Parse SLR(1) Parsing
( ) $ A S
0 s1 2 3
1 s1 s4 2 5
2 r2 r2 r2
3 s1 acc 6
4 r4 r4 r4
5 s1 s7 6
6 r1 r1 r1
7 r3 r3 r3
Start Step Noninverted Tree
Input (()())
Input Remaining)$
Stack 1(0
S' → S
S → SA
S → A
A → (S)
A → ()
S
A
(A) ( )
Reducing by S→ SA, SA popped off stack


JFLAP : (Gram1.jff)
File Input Test Convert Help
Editor
Build SLR(1) Parse
SLR(1) Parsing
( ) $ A S
0 s1 2 3
1 s1 s4 2 5
2 r2 r2 r2
3 s1 acc 6
4 r4 r4 r4
5 s1 s7 6
6 r1 r1 r1
7 r3 r3 r3
Start
Step
Noninverted Tree
Input (()())
Input Remaining)$
Stack 5S1(0
S' → S
S → SA
S → A
A → (S)
A → ()
S
A
Reducing by S→ SA, S pushed on stack

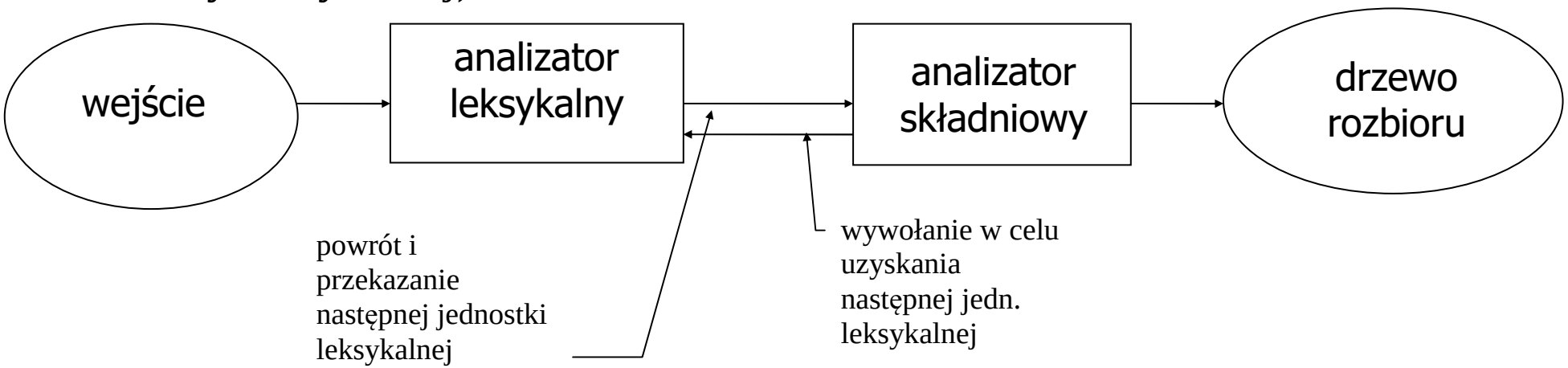

JFLAP : (Gram1.jff)
File Input Test Convert Help
Editor Build SLR(1) Parse SLR(1) Parsing
( ) $ A S
0 s1 2 3
1 s1 s4 2 5
2 r2 r2 r2
3 s1 acc 6
4 r4 r4 r4
5 s1 s7 6
6 r1 r1 r1
7 r3 r3 r3
Start Step Noninverted Tree
Input (()())
Input Remaining $
Stack S0
S' ⟶ S
S ⟶ SA
S ⟶ A
A ⟶ (S)
A ⟶ ()
String accepted

8. Zagadnienie analizy składniowej i generator parserów YACC

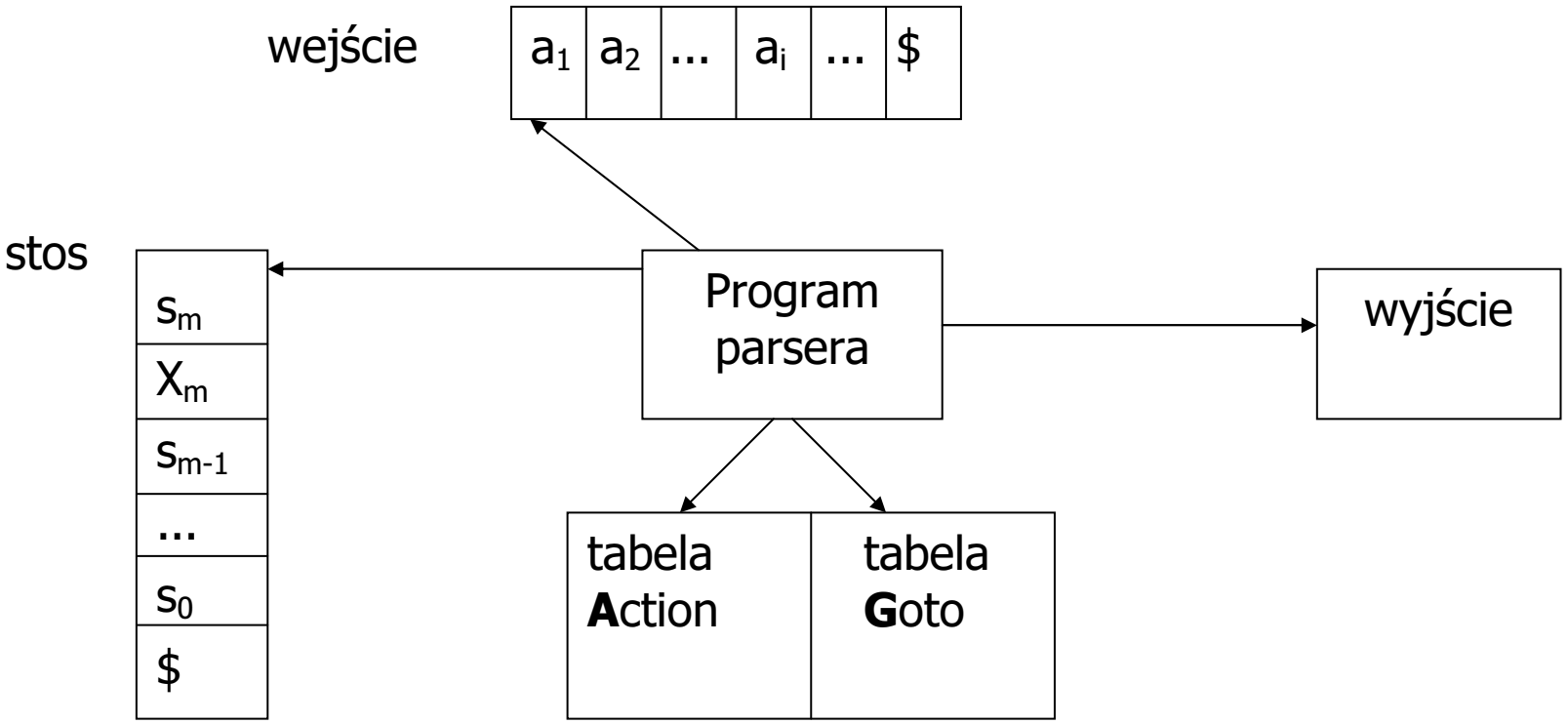
- ***analiza składniowa*** to proces sprawdzania poprawności budowy napisów pewnego języka w odniesieniu do ***reguł składniowych***, zdefiniowanych zazwyczaj za pomocą ***gramatyki bezkontekstowej***

- ***analizator składniowy*** (parser) jest programem realizującym proces analizy składniowej, który symuluje działanie automatu ze stosem (najczęściej deterministycznego), będącego akceptorem napisów języka

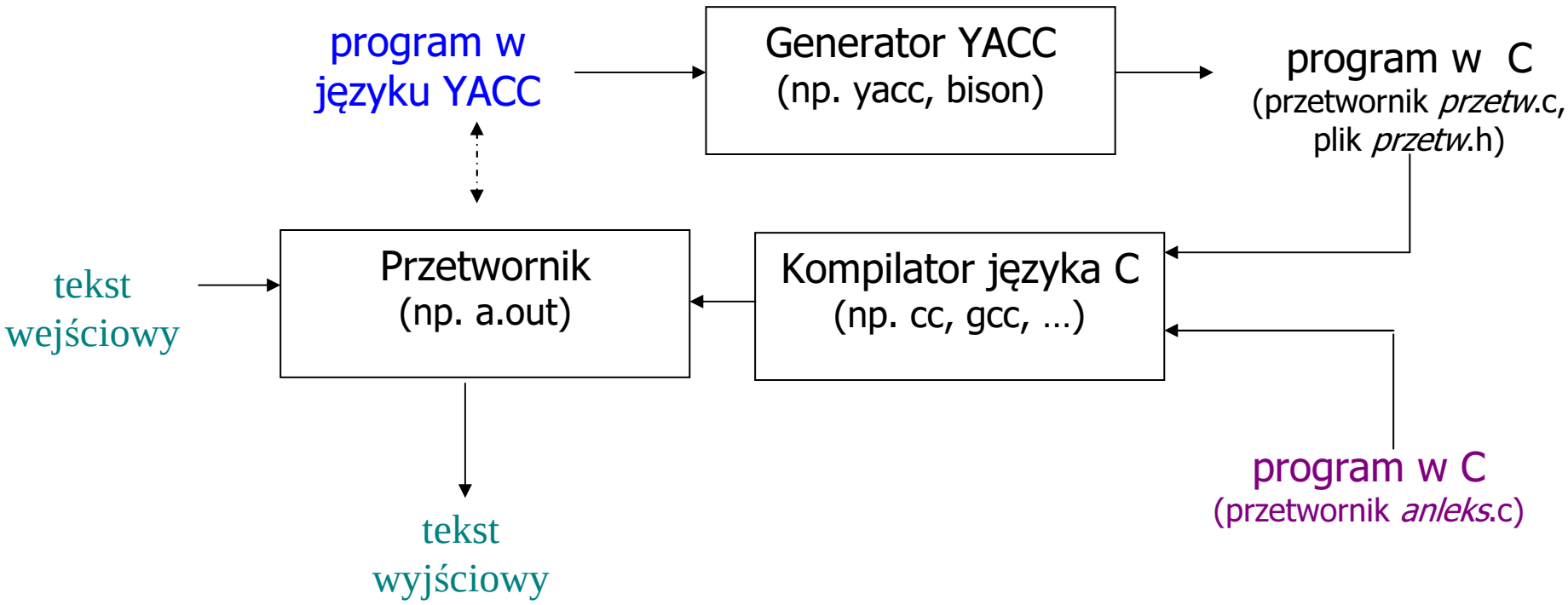
- daną wejściową analizatora składniowego jest reprezentacja napisu w postaci ciągu ***jednostek leksykalnych*** (ang. *token*) (wyodrębnionych w procesie analizy leksykalnej):



- struktura analizatora składniowego działającego metodą **LR**:



- schemat działania generatora parserów (i translatorów) YACC (*Yet Another Compiler-Compiler*, 1970, S.C. Johnson):



- struktura programu w języku YACC:

  Deklaracje
  %%
  Reguły_przetwarzania
  %%
  Podprogramy

- ***deklaracje*** są opcjonalne; są to deklaracje, definicje w sensie języka C, ujęte w nawiasy

```
%{
   …
%}
```

- ***definicje jednostek leksykalnych*** w linii (najprostsza postać):

  `%token` *nazwa_1 nazwa_2 ... nazwa_n*

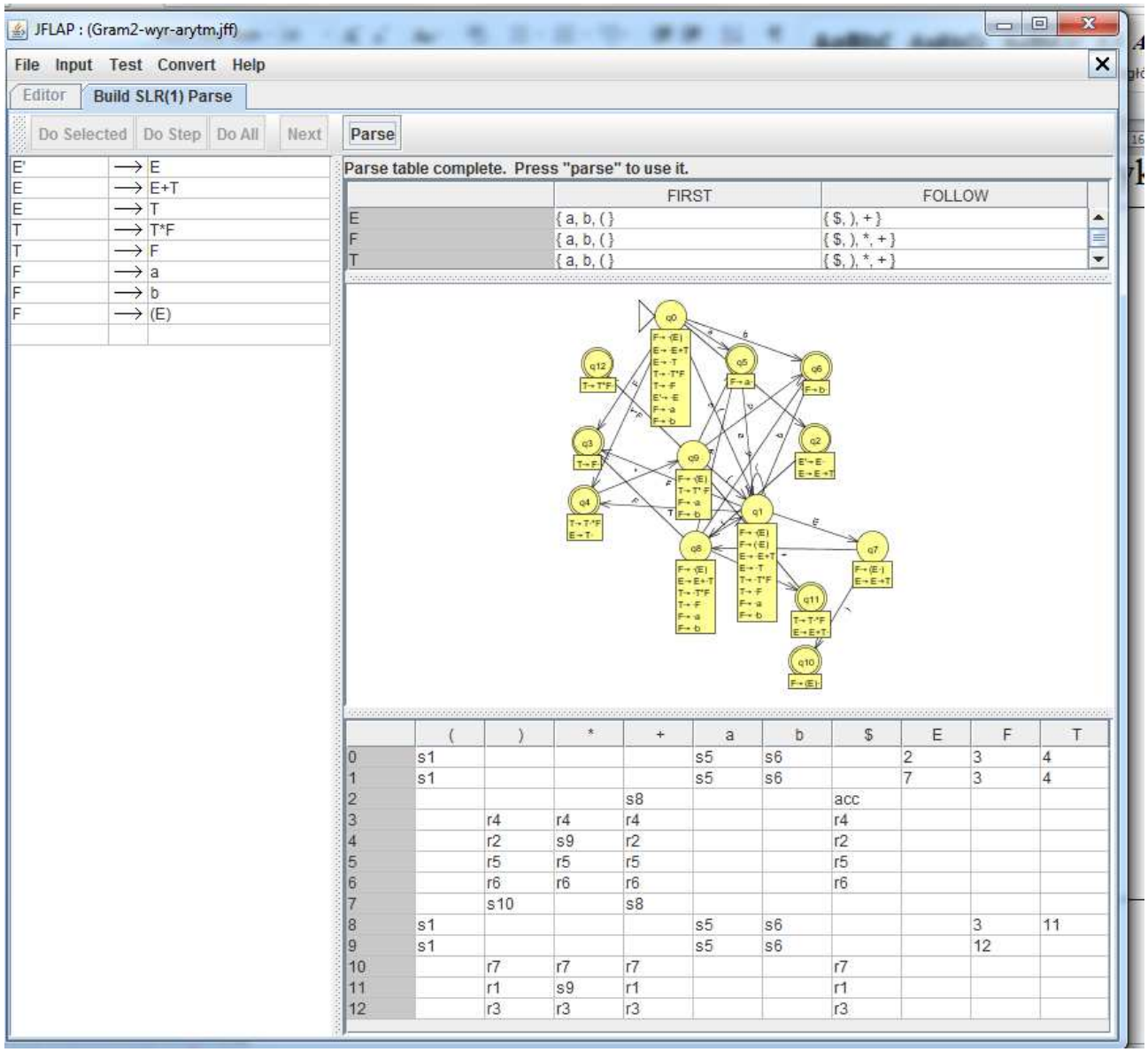
- ***reguły_przetwarzania*** występują zawsze i definiują produkcje gramatyki wraz z (ewentualnymi) akcjami (w języku C) opisującymi sposób przetwarzania napisów języka; schemat zapisu ciągu produkcji (dla jednego nieterminala):

```
nieterminal : prawa_1     {akcja_1}
            | prawa_2     {akcja_2}
            ...
            | prawa_n     {akcja_n}
            ;
```

- ***podprogramy*** występują opcjonalnie i zawierają dowolne funkcje pomocnicze w języku C, które mogą być wykorzystywane w akcjach

- **<u>przykład</u>** (analizator składniowy prostego języka wyrażeń arytmetycznych):

```
%token letter digit
%start Expr
%%
Expr  : Expr '+' Term
      | Term
      ;
Term  : Term  '*' Factor
      | Factor
      ;
Factor : letter
      | digit
      | '(' Expr ')'
      ;
%%
#include <stdio.h>
yyerror(char *s)
{     printf("Syntax error!!!\n");
}
```



E'	⟶ E
E	⟶ E+T
E	⟶ T
T	⟶ T*F
T	⟶ F
F	⟶ a
F	⟶ b
F	⟶ (E)

	FIRST	FOLLOW
E	{ a, b, (}	{ $,), + }
F	{ a, b, (}	{ $,), *, + }
T	{ a, b, (}	{ $,), *, + }

	(	)	*	+	a	b	$	E	F	T
0	s1				s5	s6		2	3	4
1	s1				s5	s6		7	3	4
2				s8			acc			
3		r4	r4	r4			r4			
4		r2	s9	r2			r2			
5		r5	r5	r5			r5			
6		r6	r6	r6			r6			
7		s10		s8						
8	s1				s5	s6			3	11
9	s1				s5	s6			12	
10		r7	r7	r7			r7			
11		r1	s9	r1			r1			
12		r3	r3	r3			r3			

- definicja analizatora leksykalnego (w języku Lex):

```
%{
#include "przetw.h"
%}
%%
[a-z]           { return(letter);}
[0-9]           { return(digit);}
"+"             { return('+');}
"*"             { return('*');}
"("             { return('('); }
")"             { return(')'); }
\ |\n|\t        ;
.               {printf(" Unexpected character!");
                 yyterminate();}
```

8.1. Uwagi o uruchamianiu generatora YACC

- jeśli plik z programem w języku YACC nazywa się *przetw*`.y`, a współpracujący z nim analizator leksykalny, zdefiniowany w języku Lex znajduje się w pliku *anleks*`.l`, to para poleceń:

  `yacc -d -o` *przetw*`.c` *przetw*`.y`

  `lex -o` *anleks*`.c` *anleks*`.l`

- spowoduje utworzenie dwóch przetworników w języku C: analizatora składniowego w pliku *przetw*`.c` i analizatora leksykalnego w pliku *anleks*`.c`, a także w pliku *przetw*`.h` umieszczone zostaną nazwy jednostek leksykalnych, rozpoznawanych przez analizator leksykalny i wczytywanych przez analizator składniowy

- przetworniki kompilujemy i konsolidujemy, korzystając z kompilatora języka C (np. `cc` lub `gcc`), podając wygenerowane pliki w C oraz wskazując pliki biblioteczne: systemu YACC `-ly` i systemu Lex `-ll`:

  `gcc` *anleks*`.c` *przetw*`.c` *przetw*`.h -ly -ll`

  powstanie plik `a.out`:

  `a.out` `<` *test.in* `>` *test.out*

```
root@boromir:/home/lab/labadm/jcyb# cat gram.y
%token letter digit
%start Expr
%%
Expr   : Expr '+' Term
       | Term
       ;
Term   : Term  '*' Factor
       | Factor
       ;
Factor : letter
       | digit
       | '(' Expr ')'
       ;
%%
#include <stdio.h>
yyerror(char *s)
{
printf("Syntax error!!!\n");
}
```

```
cat: sca.l: No such file or directory
root@boromir:/home/lab/labadm/jcyb# cat scan.l
%{
#include "gram.h"
%}
%%
[a-z]    { return(letter);}
[0-9]    { return(digit);}
"+"      { return('+');}
"*"      { return('*');}
"("      { return('('); }
")"      { return(')'); }
\ |\n|\t  ;
.        {printf(" %c - Unexpected character!\n", yytext[0]); exit(0);}
```

```
root@boromir:/home/lab/labadm/jcyb#
root@boromir:/home/lab/labadm/jcyb# yacc -d -o gram.c gram.y
root@boromir:/home/lab/labadm/jcyb# lex -o scan.c scan.l
root@boromir:/home/lab/labadm/jcyb# cc gram.c scan.c gram.h -ly -ll
root@boromir:/home/lab/labadm/jcyb# ./a.out<we
root@boromir:/home/lab/labadm/jcyb# cat we
1+2+3*(4+a)
root@boromir:/home/lab/labadm/jcyb# ./a.out<we
root@boromir:/home/lab/labadm/jcyb# cat we1
2+4*
root@boromir:/home/lab/labadm/jcyb# ./a.out<we1
Syntax error!!!
root@boromir:/home/lab/labadm/jcyb# cat we3
2+&
root@boromir:/home/lab/labadm/jcyb# ./a.out<we3
 & - Unexpected character!
```

8.2. Przykłady działania LR parserów

Przykład 1

– rozważmy język zawierający skończone, niepuste ciągi liczb całkowitych ujęte w nawiasy, w których liczby są oddzielone od siebie przecinkami. Oto przykładowe ciągi:

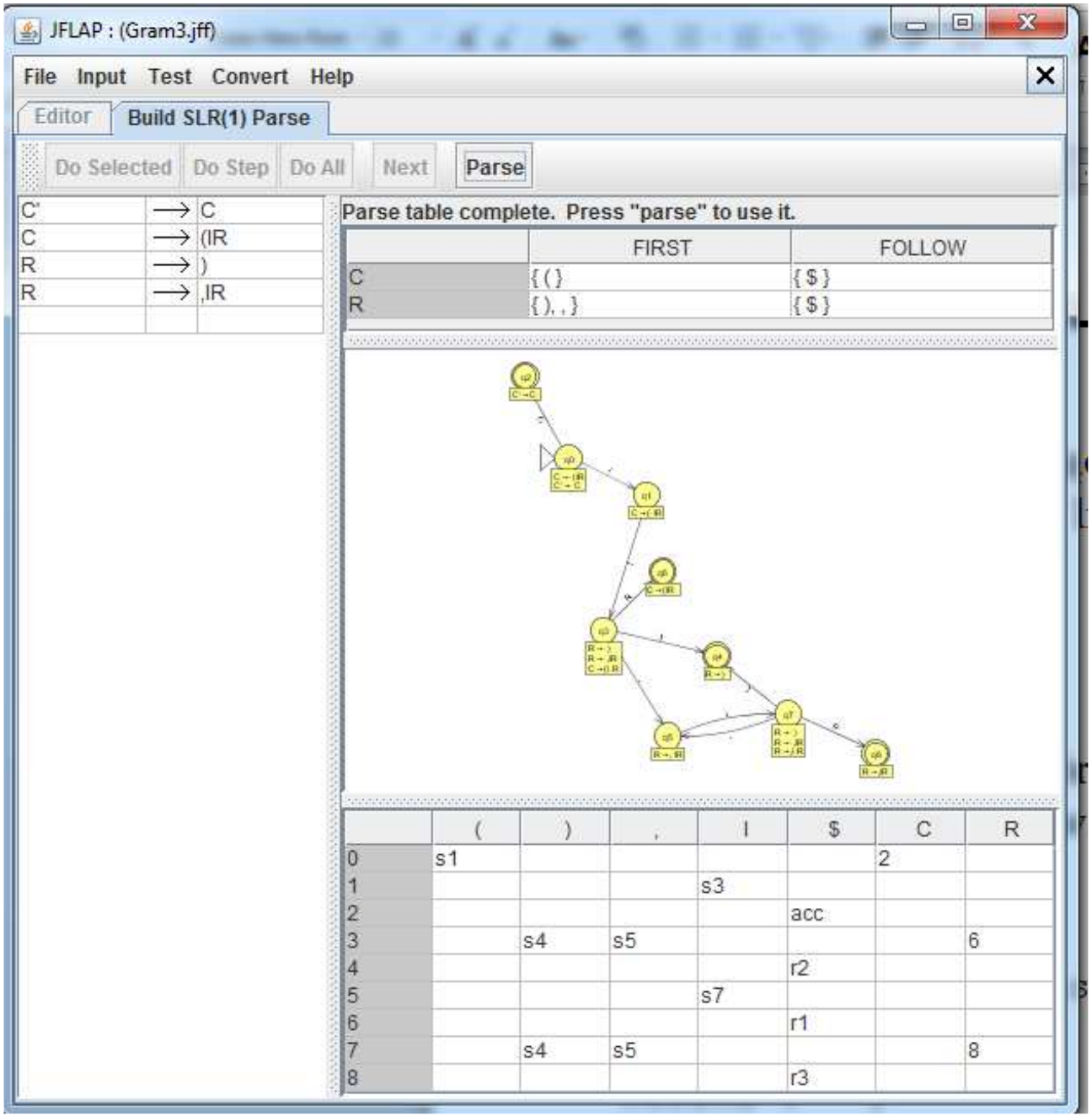
```
(44)
(19, 44)
(1, 22, 333)
```

– program w YACCu, w którym z każdą produkcją związana jest akcja drukująca na wyjściu numer zastosowanej produkcji (1., 2. lub 3.):

```
%token Liczba
%%
Ciag : '(' Liczba Reszta     {printf("1.");}
     ;
Reszta :  ')'                {printf("2. ");}
       |  ',' Liczba Reszta  {printf("3.");}
       ;
```

– analizator leksykalny rozpoznający słowa:

```
%{
#include "przetw.h"
%}
%%
[0-9]+   {return Liczba;}
"("      {return '(' ;}
")"      {return ')' ;}
","      {return ',' ;}
" "      {;}
.        {yyerror(""); exit(0);}
```


JFLAP : (Gram3.jff)
File Input Test Convert Help
Editor
Build SLR(1) Parse
Do Selected
Do Step
Do All
Next
Parse
C' ⟶ C
C ⟶ (IR
R ⟶)
R ⟶ ,IR
Parse table complete. Press "parse" to use it.
FIRST
FOLLOW
C { (} { $ }
R {), , } { $ }
( ) , I $ C R
0 s1 2
1 s3
2 acc
3 s4 s5 6
4 r2
5 s7
6 r1
7 s4 s5 8
8 r3

– rozważmy wywód ciągu:

```
(1, 22, 333)
```

– analizując produkcje gramatyki („z góry na dół”), spodziewamy się wyniku:

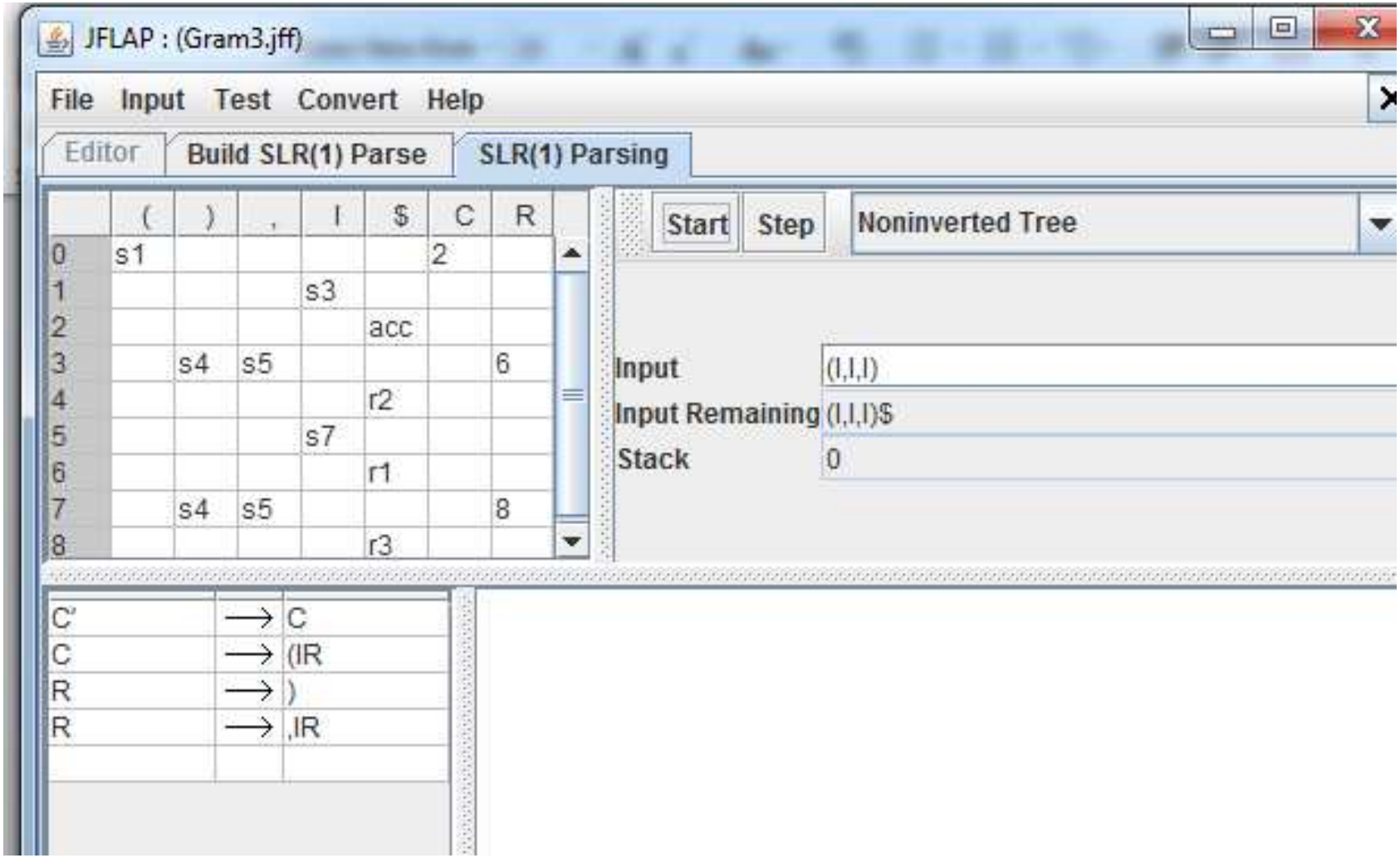
```
1. 3. 3. 2.
```

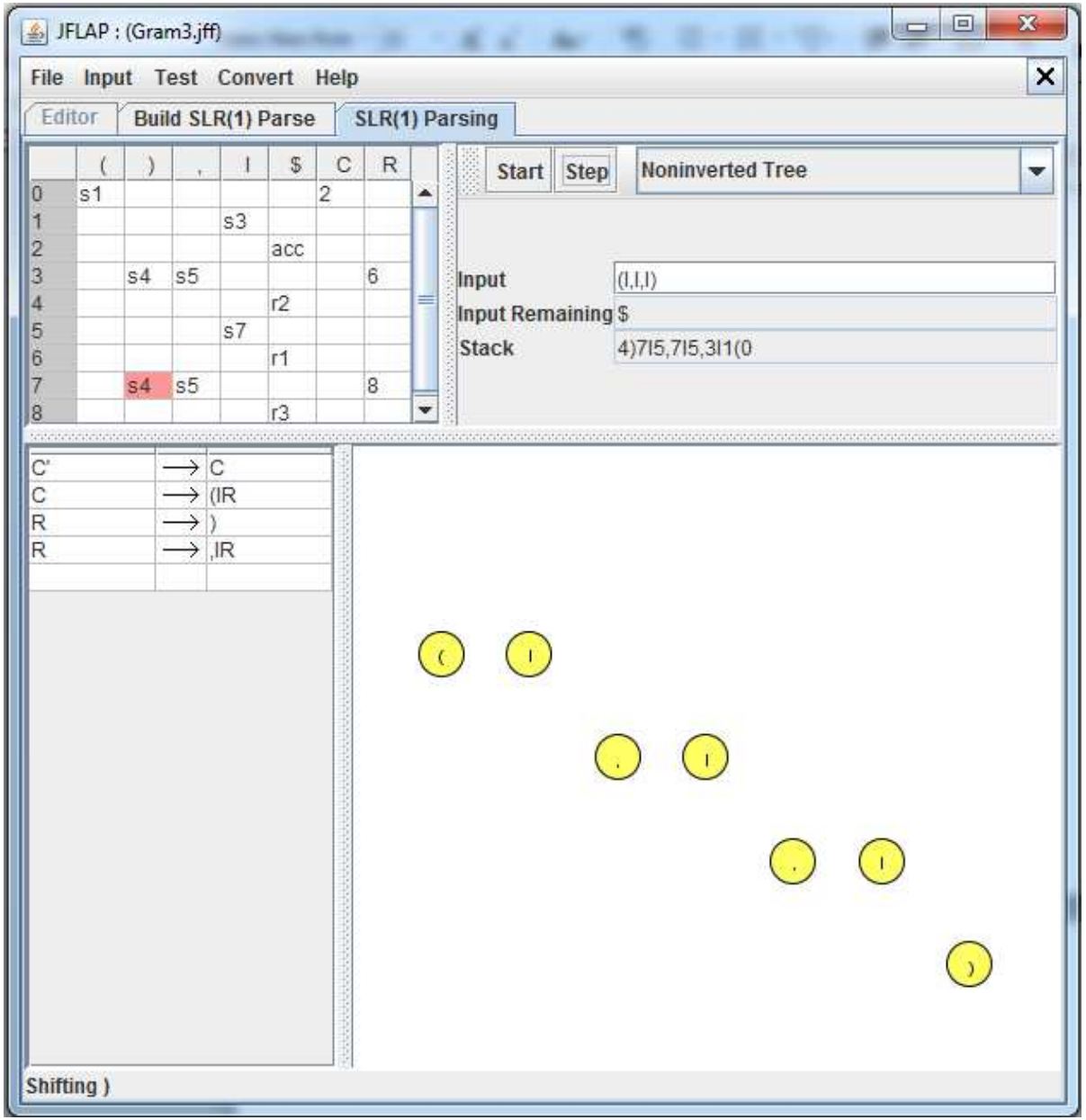
natomiast uzyskujemy:

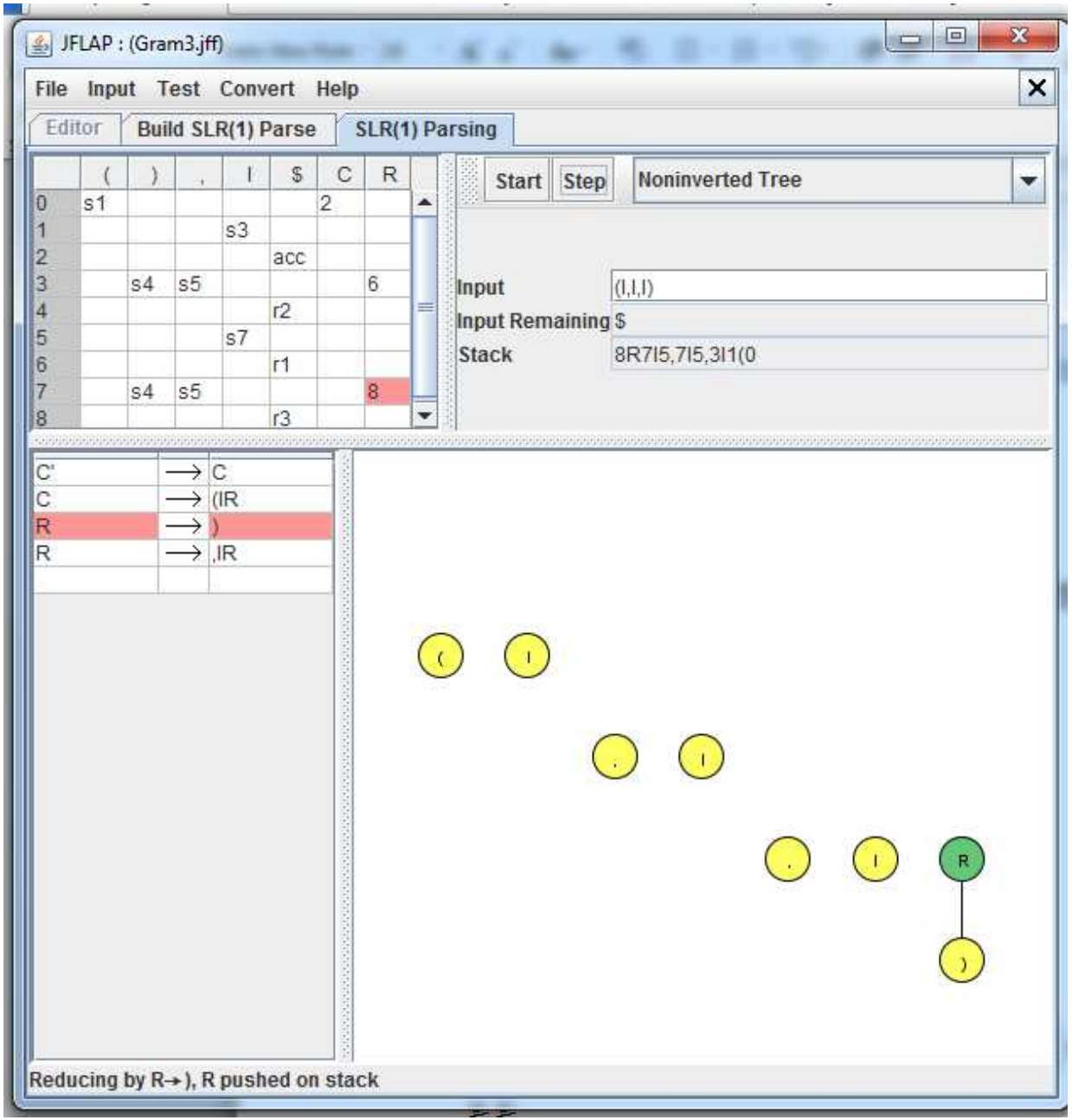
```
2. 3. 3. 1.
```

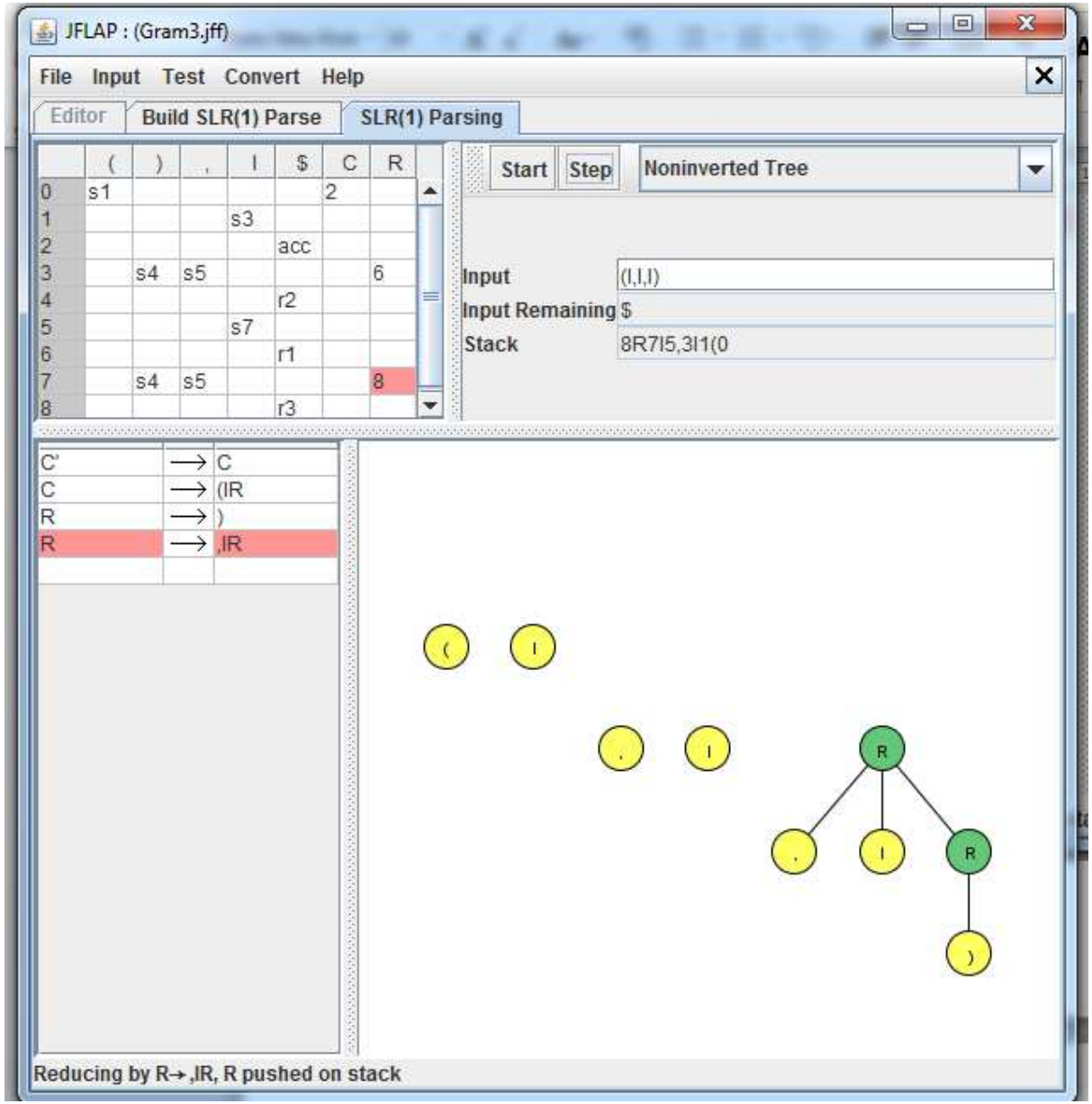
– sytuacja taka wynika z algorytmu działania parsera, którego efektem jest następujący ciąg konfiguracji (dla uproszczenia, na stosie nie umieszczono stanów), wg zasady: „wykonuj redukcję tak szybko, jak to możliwe”

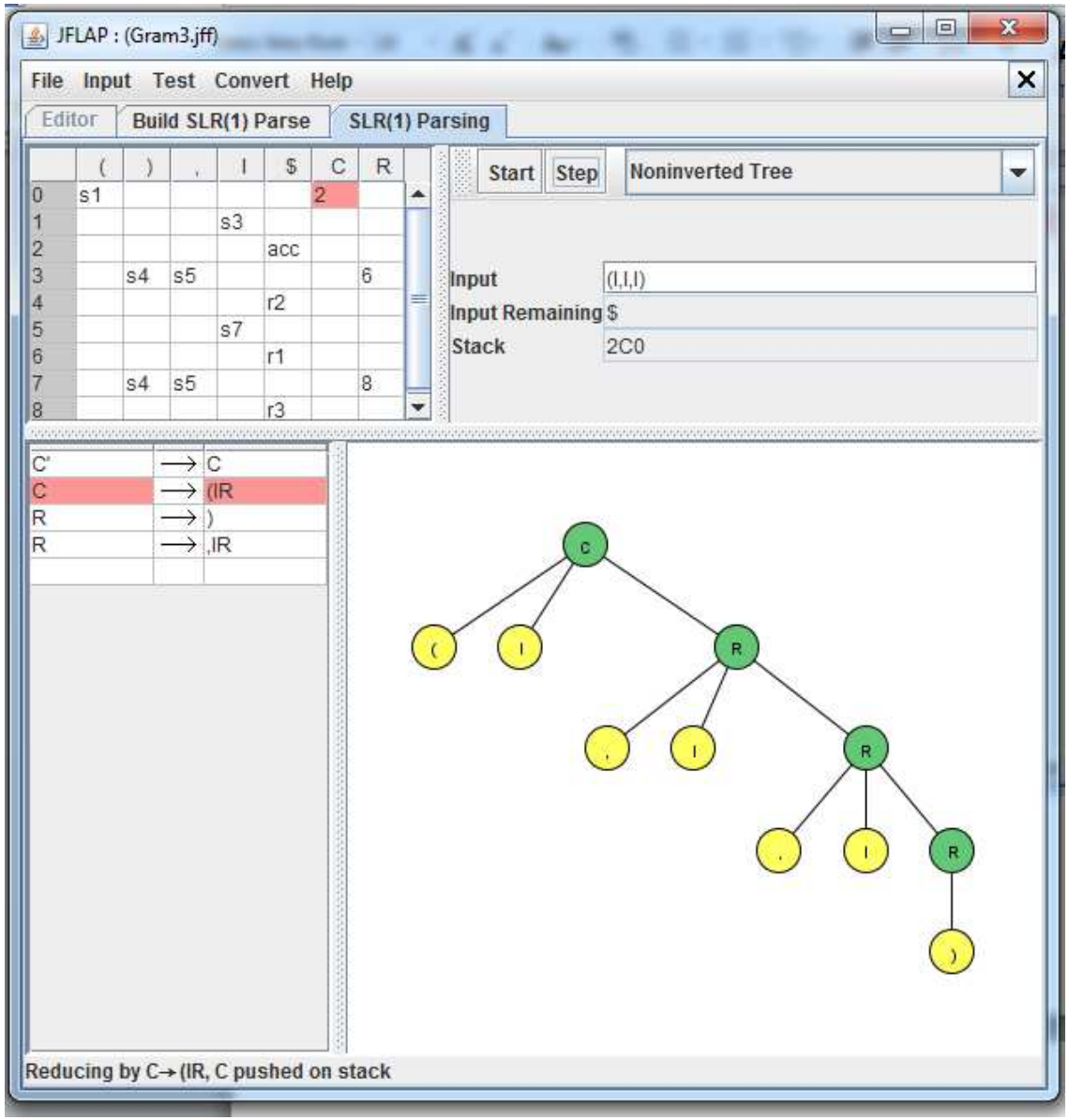
Stos	*Wejście*	*Akcja*	*Wyjście*
$	(L , L , L) $	shift	
$ (	L , L , L) $	shift	
$ (L	, L , L) $	shift	
$ (L ,	L , L) $	shift	
$ (L , L	, L) $	shift	
$ (L , L ,	L) $	shift	
$ (L , L , L	) $	shift	
$ (L , L , L)	$	reduce R→)	2.
$ (L , L , L R	$	reduce R→ , L R	3.
$ (L , L R	$	reduce R→ , L R	3.
$ (L R	$	reduce C→ (L R	1.
$C	$	accept	
	$		


JFLAP : (Gram3.jff)
File Input Test Convert Help
Editor
Build SLR(1) Parse
SLR(1) Parsing
( ) , I $ C R
0 s1 2
1 s3
2 acc
3 s4 s5 6
4 r2
5 s7
6 r1
7 s4 s5 8
8 r3
Start
Step
Noninverted Tree
Input (I,I,I)
Input Remaining (I,I,I)$
Stack 0
C' → C
C → (IR
R →)
R → ,IR


JFLAP : (Gram3.jff)
File Input Test Convert Help
Editor
Build SLR(1) Parse
SLR(1) Parsing
Start
Step
Noninverted Tree
Input
(I,I,I)
Input Remaining
$
Stack
4)7I5,7I5,3I1(0
C' ⟶ C
C ⟶ (IR
R ⟶)
R ⟶ ,IR
Shifting)


JFLAP : (Gram3.jff)
File Input Test Convert Help
Editor
Build SLR(1) Parse
SLR(1) Parsing
( ) , I $ C R
0 s1 2
1 s3
2 acc
3 s4 s5 6
4 r2
5 s7
6 r1
7 s4 s5 8
8 r3
Start
Step
Noninverted Tree
Input (I,I,I)
Input Remaining $
Stack 8R7I5,7I5,3I1(0
C' → C
C → (IR
R →)
R → ,IR
(I , I , I R)
Reducing by R→), R pushed on stack


JFLAP : (Gram3.jff)
File Input Test Convert Help
Editor
Build SLR(1) Parse
SLR(1) Parsing
( ) , I $ C R
0 s1 2
1 s3
2 acc
3 s4 s5 6
4 r2
5 s7
6 r1
7 s4 s5 8
8 r3
Start
Step
Noninverted Tree
Input (I,I,I)
Input Remaining $
Stack 8R7I5,3I1(0
C' → C
C → (IR
R →)
R → ,IR
(I , I R , I R)
Reducing by R→ ,IR, R pushed on stack


JFLAP : (Gram3.jff)
File Input Test Convert Help
Editor
Build SLR(1) Parse
SLR(1) Parsing
( ) , I $ C R
0 s1 2
1 s3
2 acc
3 s4 s5 6
4 r2
5 s7
6 r1
7 s4 s5 8
8 r3
Start
Step
Noninverted Tree
Input (I,I,I)
Input Remaining $
Stack 2C0
C' → C
C → (IR
R →)
R → ,IR
C
(I R
, I R
, I)
Reducing by C→ (IR, C pushed on stack

Przykład 2

– rozważmy prosty język L= {ca, cb}. Język ten można opisać następująco:

```
%%
S :  X  'a'
  | 'c' 'b'
  ;
X : 'c'
  ;
```

zastosowanie przytoczonej zasady analizy dla ciągu cb daje rezultat:

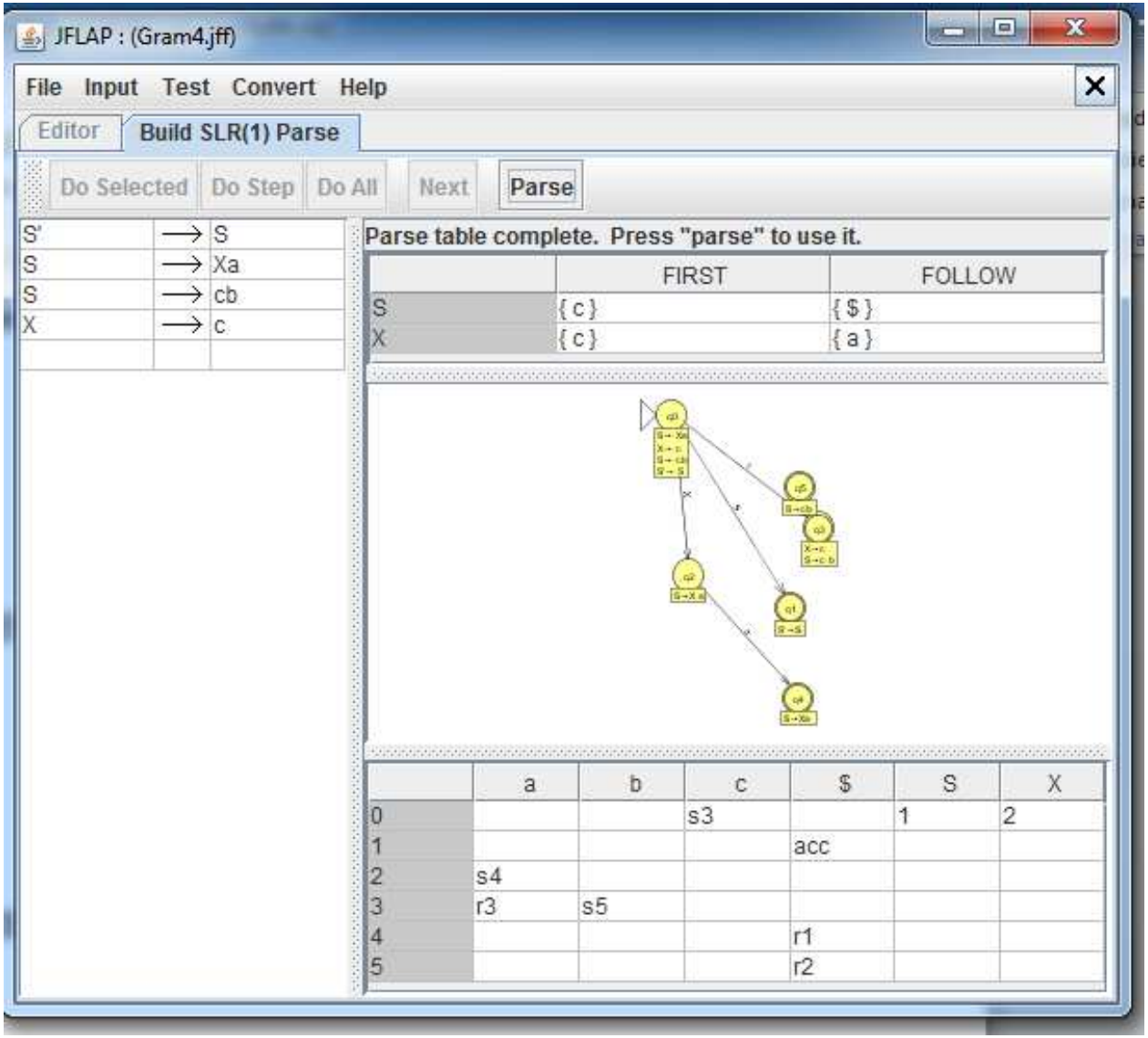
Stos	*Wejście*	*Akcja*
$	c b $	shift
$ c	b $	reduce X→ c
$ X	b $	shift
$ X b	$	error

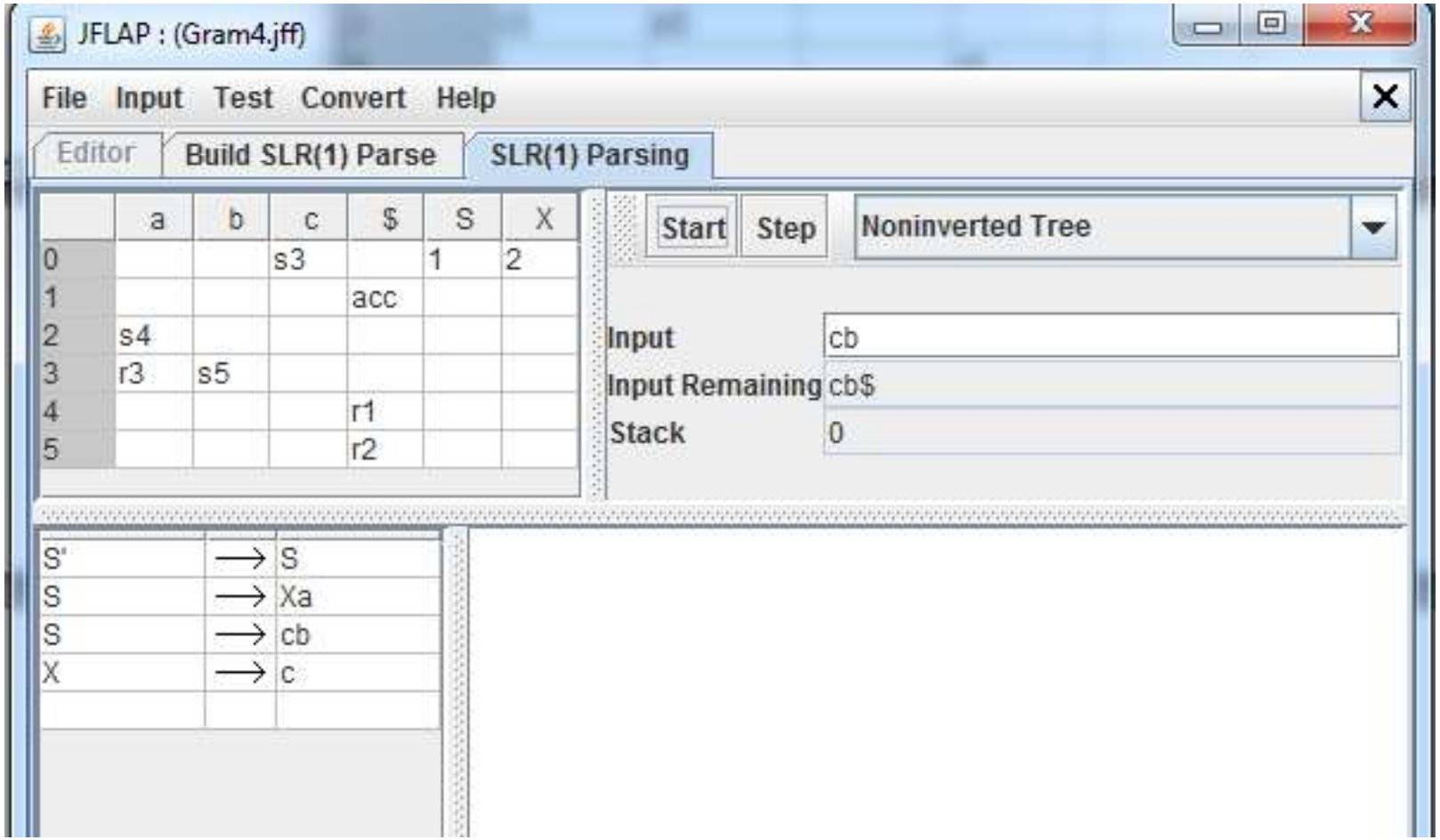
zatem, zasada powinna być zmodyfikowana do postaci: „wykonuj redukcję produkcją A → α tylko wtedy, gdy symbol terminalny widoczny na wejściu należy do zbioru symboli dopuszczalnych dla symbolu A (tzw. zbiór

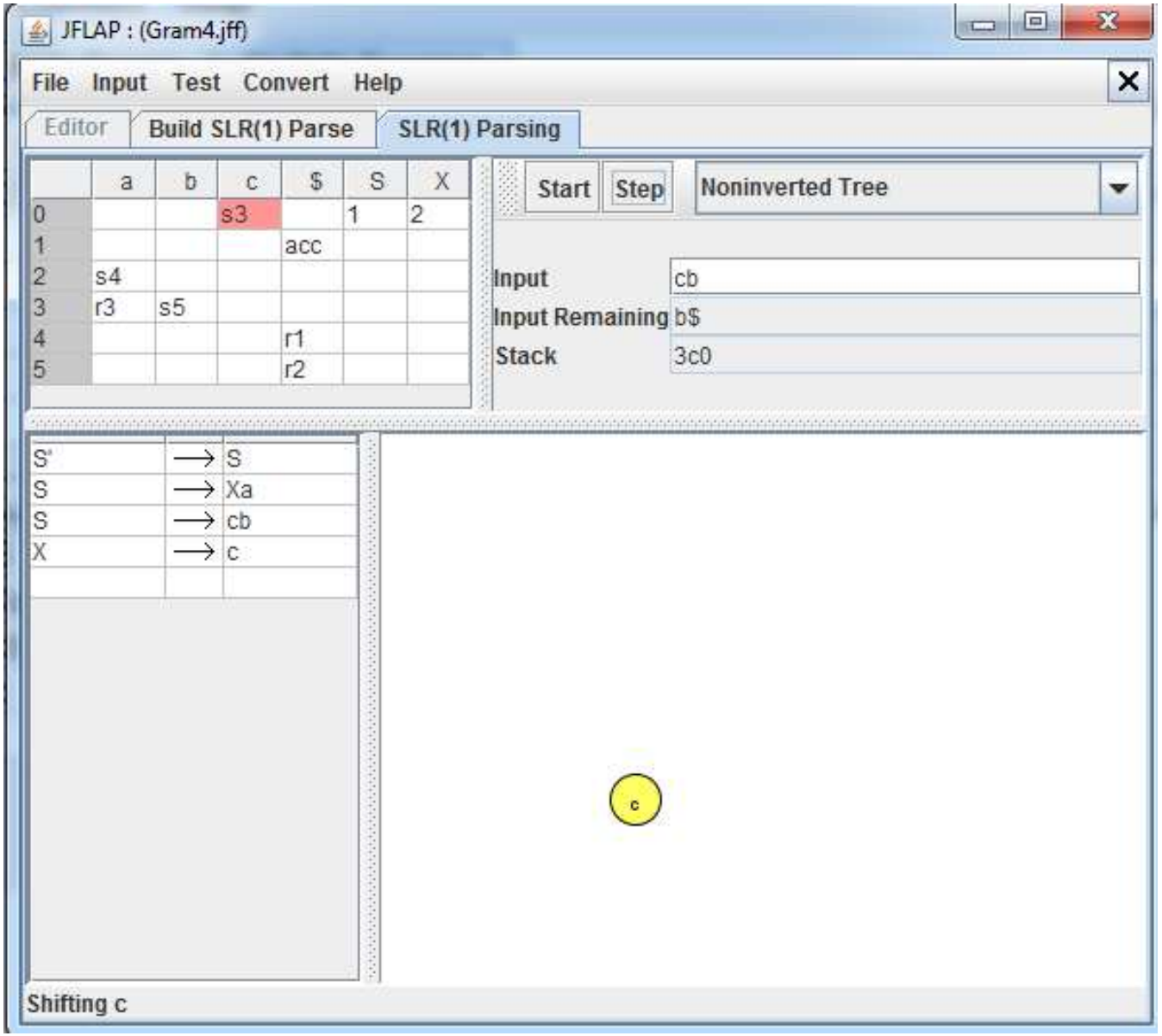
FOLLOW(A)). Jeżeli dana decyzja prowadzi do akcji *error*, to wybierz alternatywną akcję, czyli gdy trzeba wykonuj nawroty”

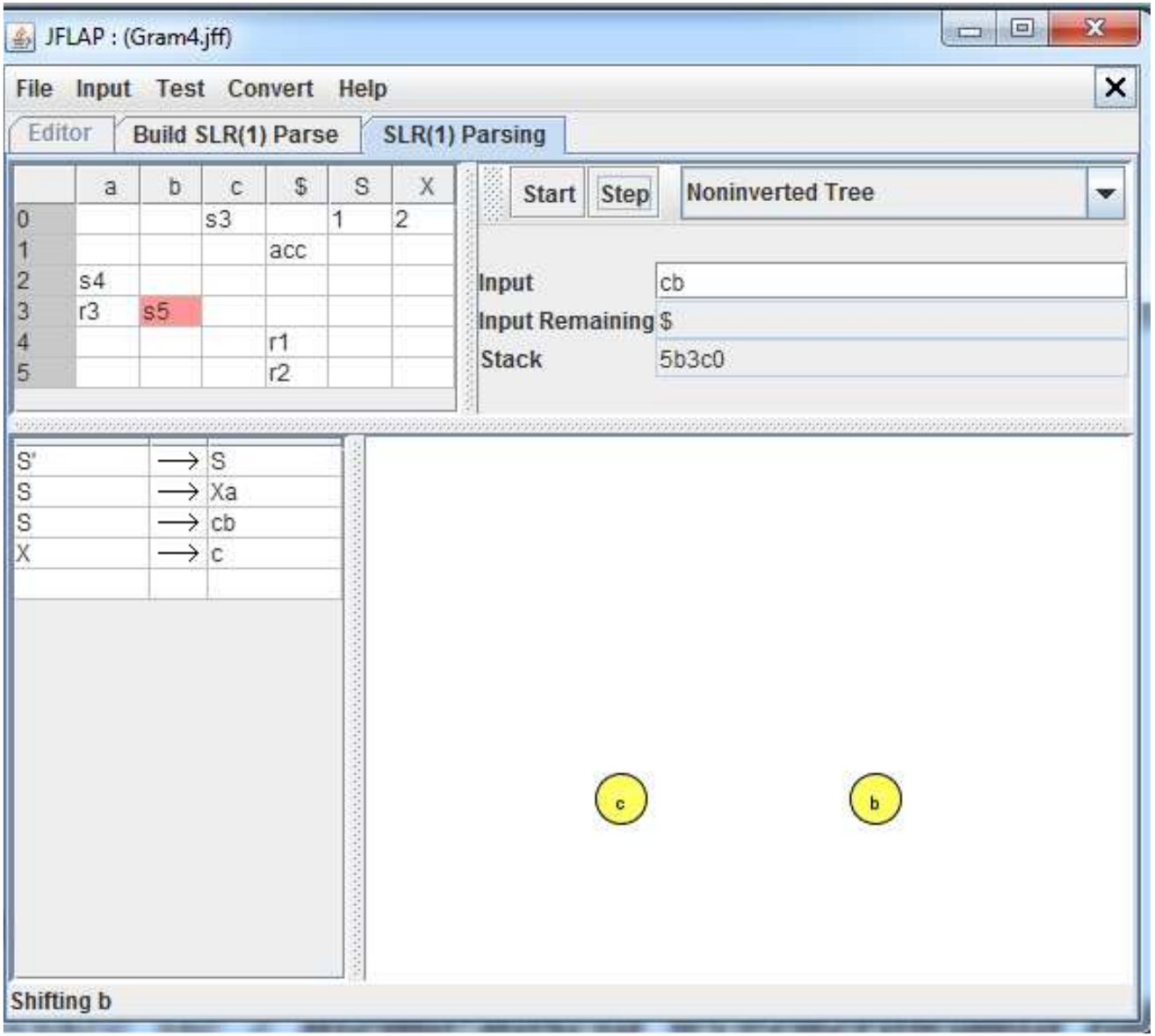
– zastosowawszy tę zasadę, uzyskamy ciąg konfiguracji:

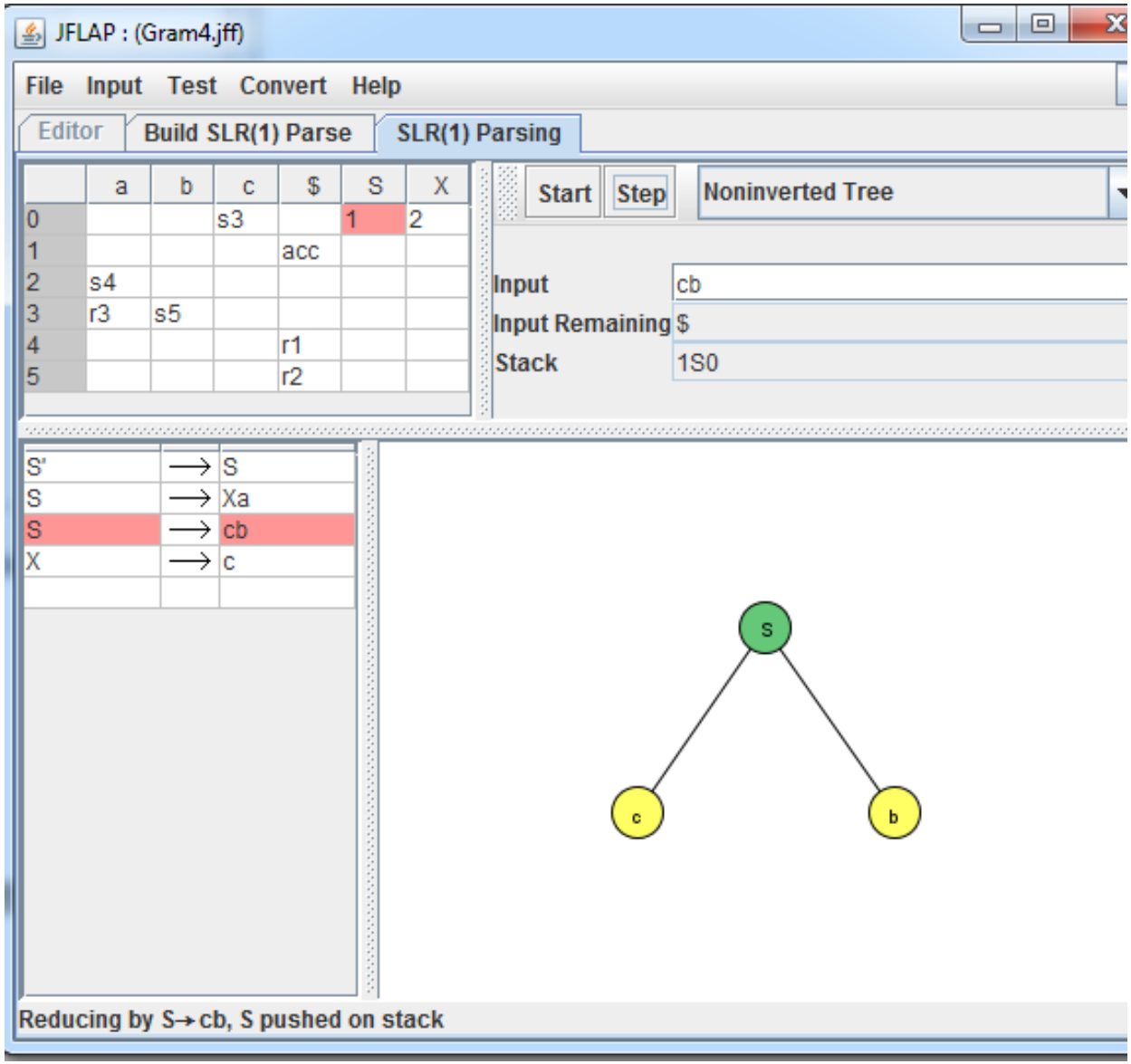
Stos	*Wejście*	*Akcja*
$	c b $	shift
$ c	b $	shift
$ c b	$	reduce S $\rightarrow$ c b
$ S	$	accept


JFLAP : (Gram4.jff)
File Input Test Convert Help
Editor
Build SLR(1) Parse
Do Selected
Do Step
Do All
Next
Parse
S' → S
S → Xa
S → cb
X → c
Parse table complete. Press "parse" to use it.
FIRST
FOLLOW
S { c } { $ }
X { c } { a }
a b c $ S X
0 s3 1 2
1 acc
2 s4
3 r3 s5
4 r1
5 r2


JFLAP : (Gram4.jff)
File Input Test Convert Help
Editor
Build SLR(1) Parse
SLR(1) Parsing
a b c $ S X
0 s3 1 2
1 acc
2 s4
3 r3 s5
4 r1
5 r2
Start
Step
Noninverted Tree
Input cb
Input Remaining cb$
Stack 0
S' → S
S → Xa
S → cb
X → c


JFLAP : (Gram4.jff)
File Input Test Convert Help
Editor
Build SLR(1) Parse
SLR(1) Parsing
Start
Step
Noninverted Tree
Input
cb
Input Remaining
b$
Stack
3c0
Shifting c


JFLAP : (Gram4.jff)
File Input Test Convert Help
Editor
Build SLR(1) Parse
SLR(1) Parsing
a b c $ S X
0 s3 1 2
1 acc
2 s4
3 r3 s5
4 r1
5 r2
Start
Step
Noninverted Tree
Input cb
Input Remaining $
Stack 5b3c0
S' → S
S → Xa
S → cb
X → c
c
b
Shifting b


JFLAP : (Gram4.jff)
File Input Test Convert Help
Editor
Build SLR(1) Parse
SLR(1) Parsing
a b c $ S X
0 s3 1 2
1 acc
2 s4
3 r3 s5
4 r1
5 r2
Start
Step
Noninverted Tree
Input cb
Input Remaining $
Stack 1S0
S' → S
S → Xa
S → cb
X → c
S
c
b
Reducing by S→cb, S pushed on stack

8.3. Powstawanie i rozstrzyganie konfliktów

– pojęcie ***konfliktu*** wiąże się z ***niemożnością wygenerowania tablic LR parsera*** dla języka opisanego przedstawioną ***gramatyką***

– ***konflikty*** objawiają się istnieniem w pewnej konfiguracji parsera ***kilku możliwych*** do wykonania ***akcji***: albo ***shift-reduce*** albo ***reduce-reduce*** (różne produkcje)

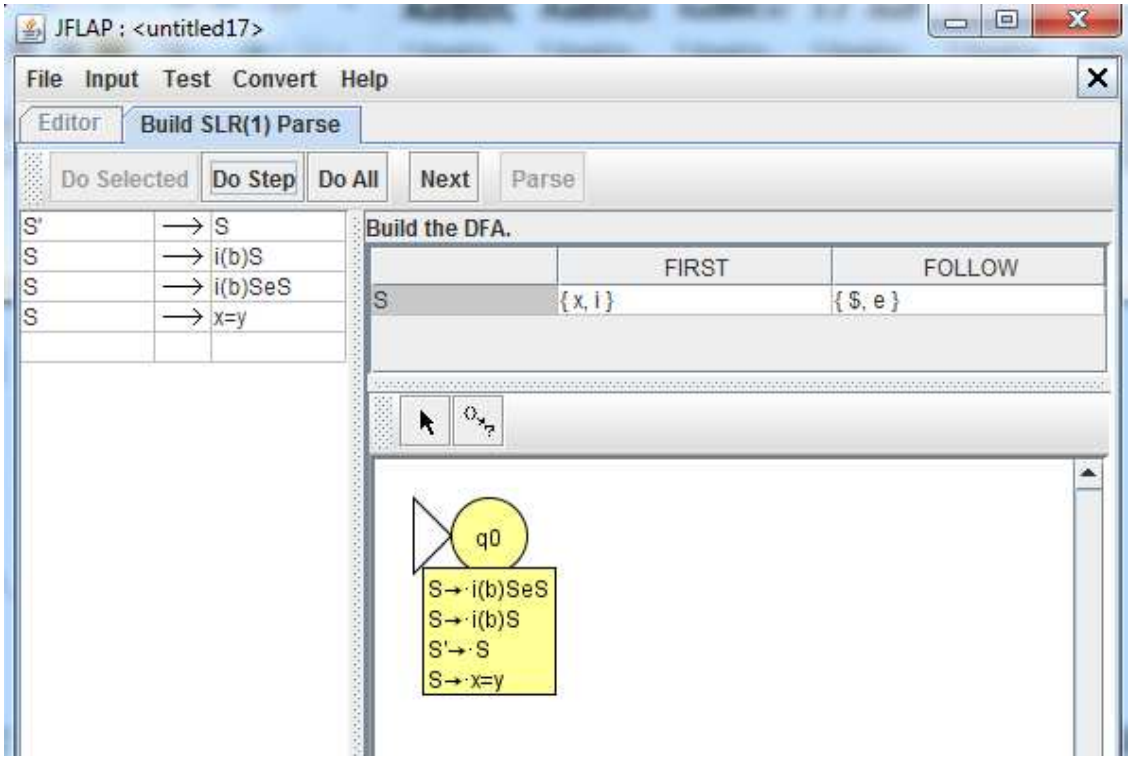
– ***źródłem*** powstania ***konfliktu*** może być ***niejednoznaczność*** gramatyki (nie jest wtedy LR(k) dla żadnego k) lub to, że ***gramatyka nie jest LR(k)*** lecz ***LR(k+1)***

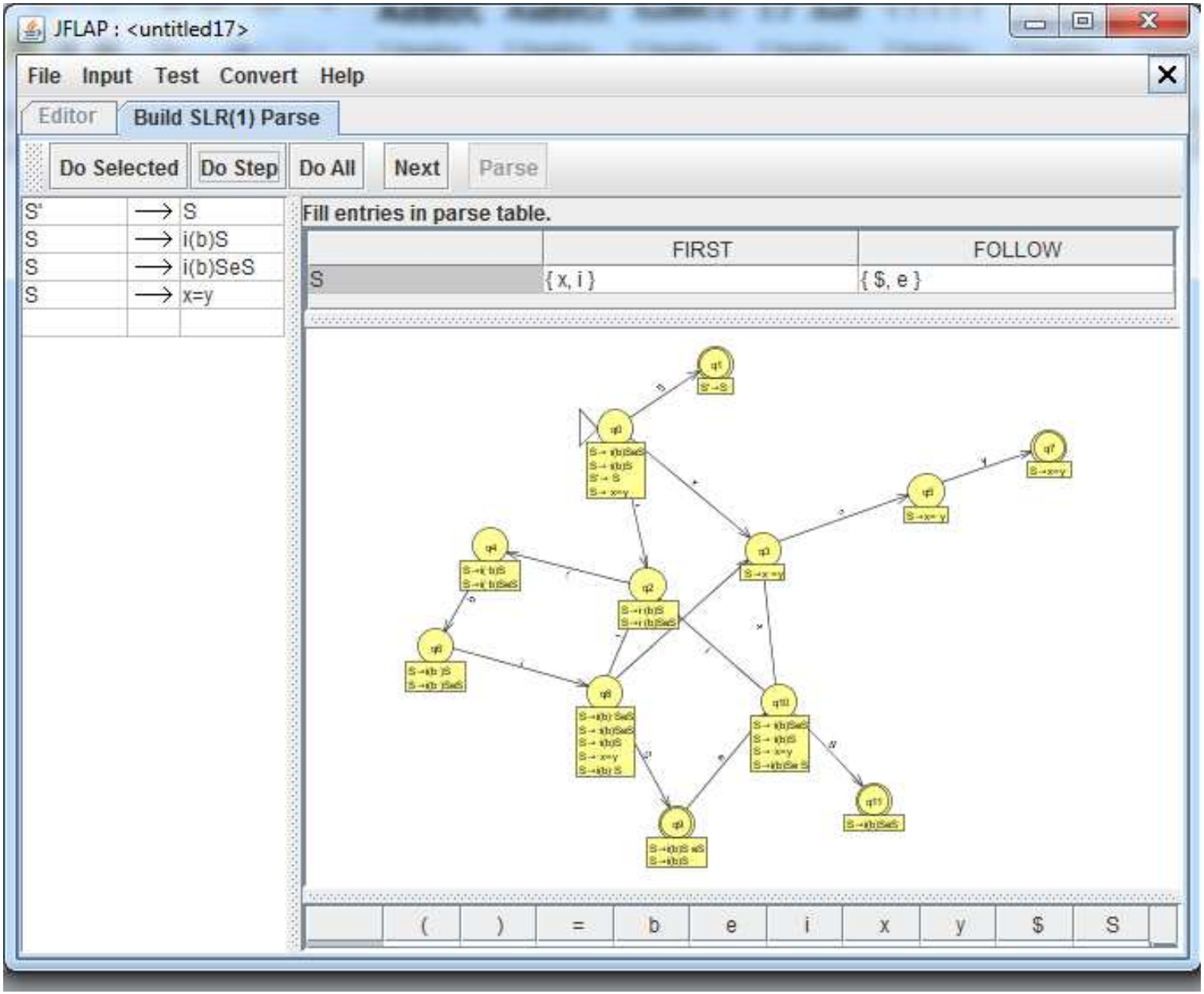
– wykorzystywanie gramatyk ***niejednoznacznych*** ma ***zalety***: ***prostota*** opisu języka, ***mniejsza***, niż w przypadku gramatyk jednoznacznych ***złożoność czasowa*** i ***pamięciowa*** wygenerowanego ***parsera***

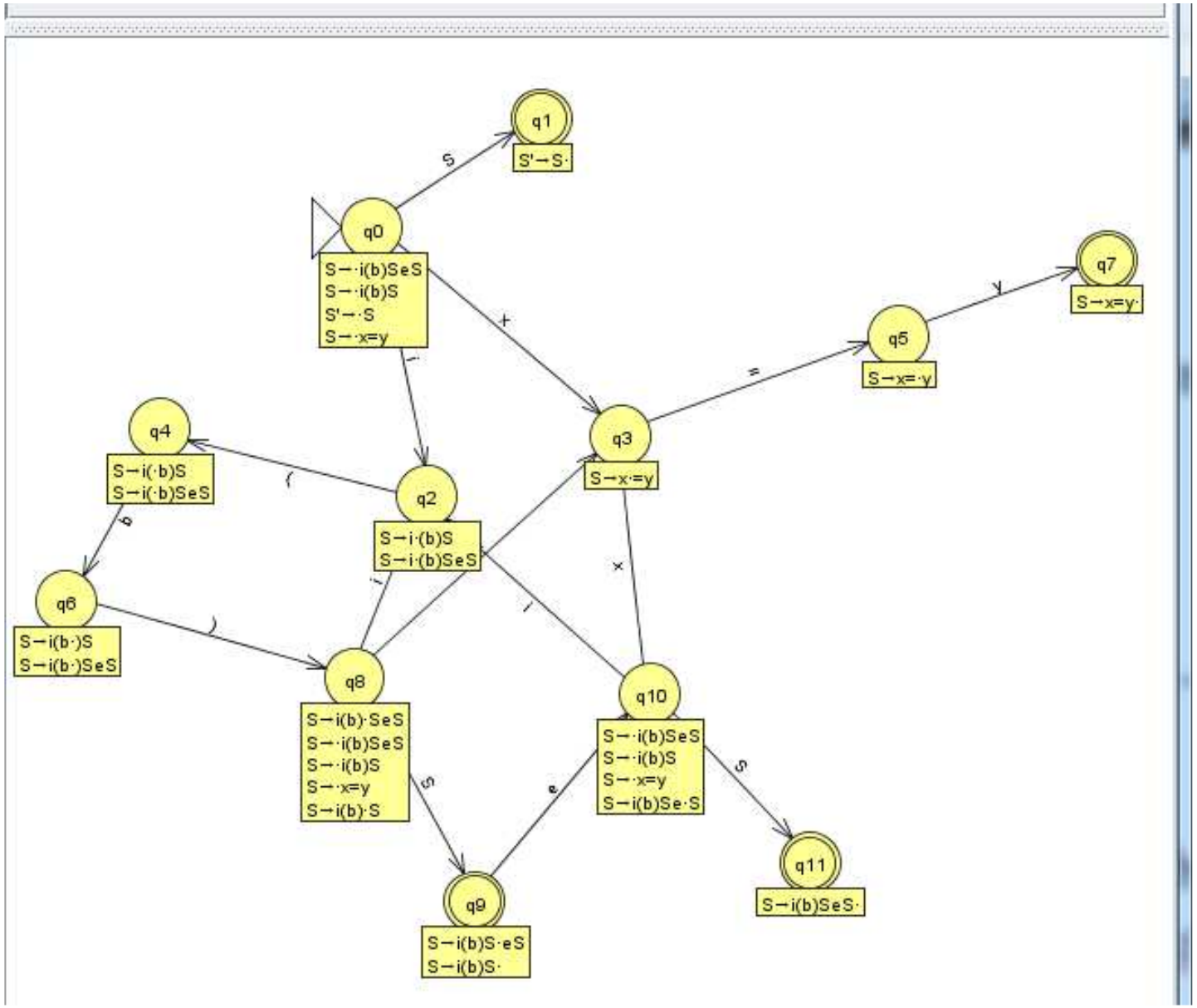
– YACC ***rozstrzyga konflikty automatycznie*** w następujący sposób: ***shift-reduce*** na korzyść ***shift***, ***reduce-reduce*** na korzyść tej produkcji, którą zdefiniowano wcześniej w specyfikacji

– przykład gramatyki niejednoznacznej (problem „*dangling-else*”):

```
%token id if_ else_
%%
S   : if_ '(' id ')' S
    | if_ '(' id ')' S else_ S
    | id '=' id
    ;
```

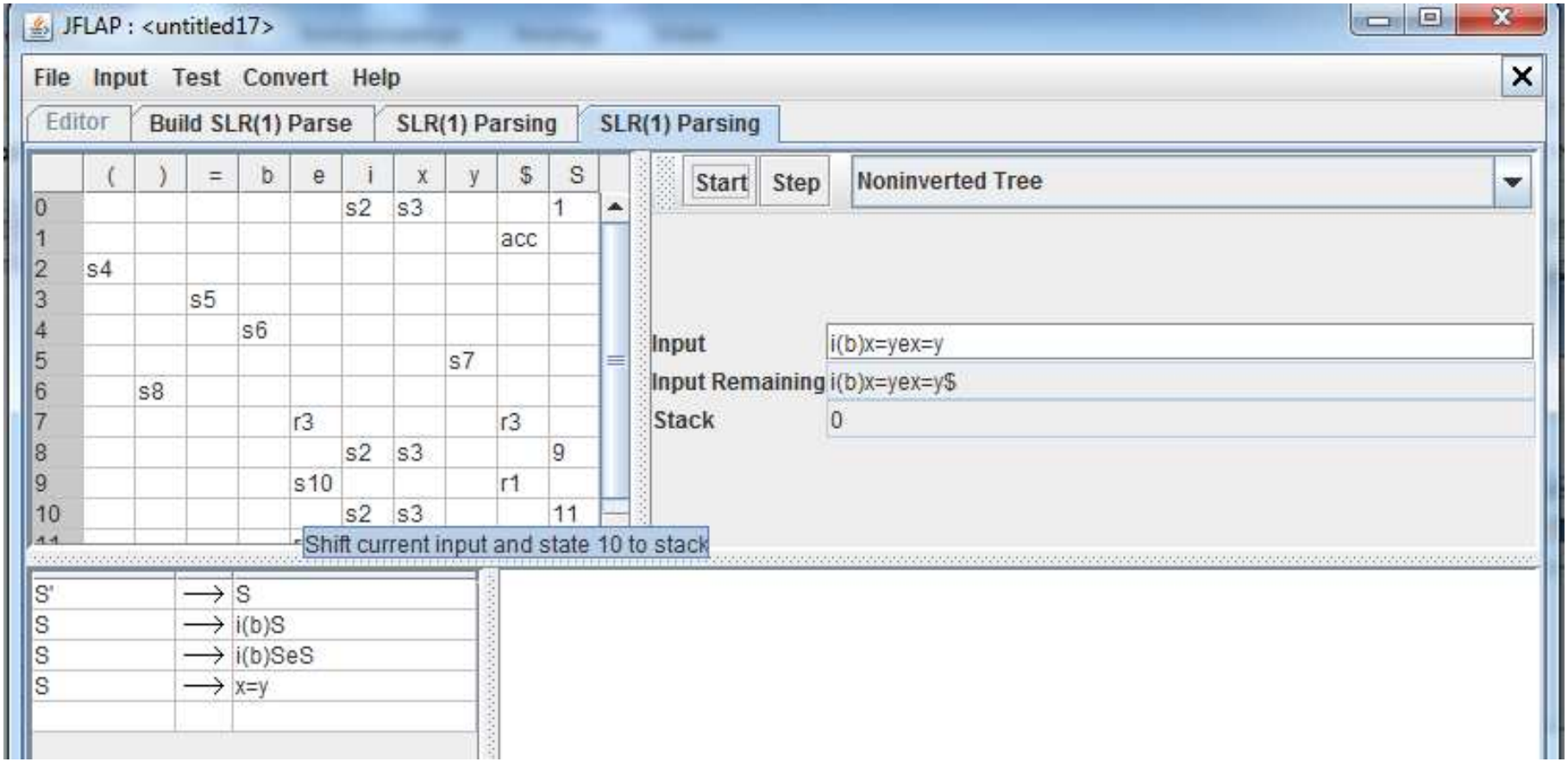


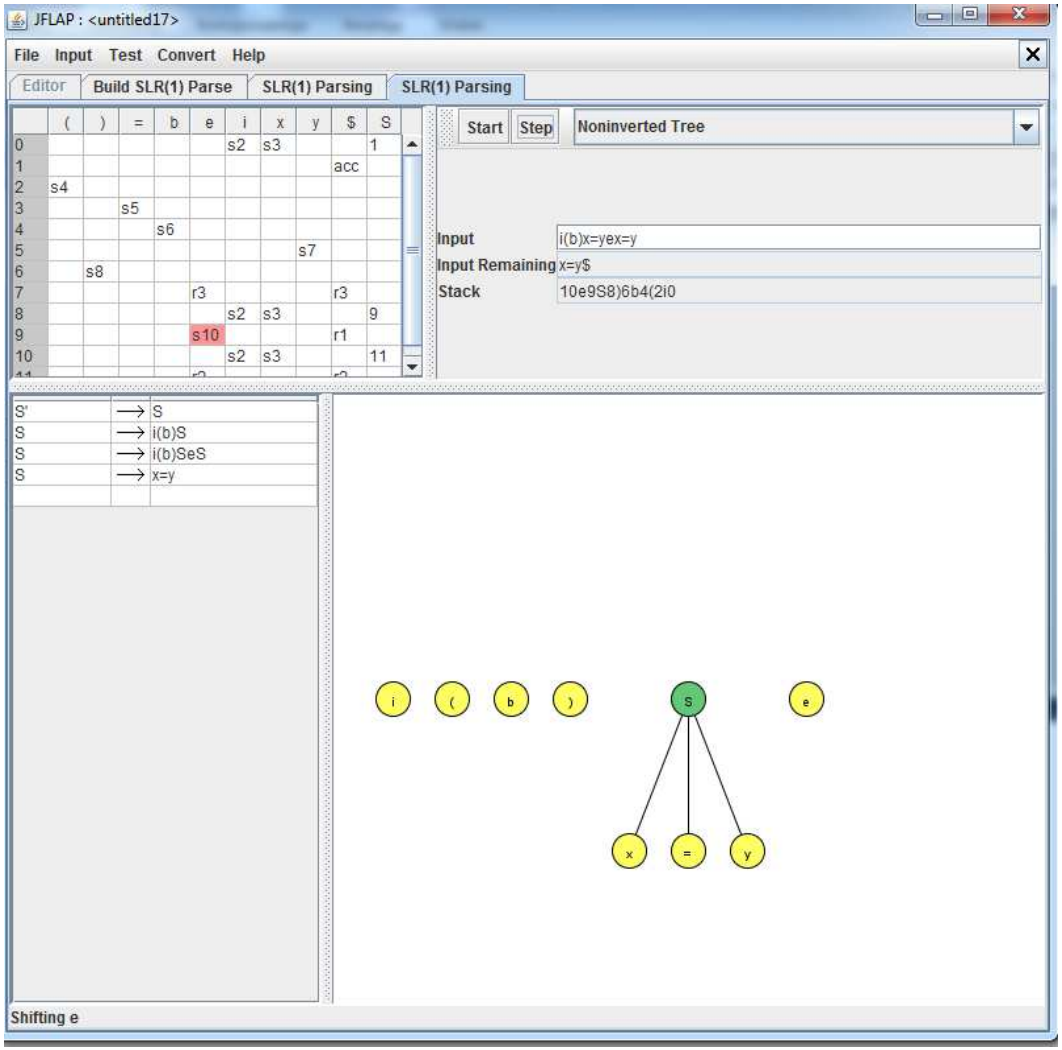

JFLAP : <untitled17>
File Input Test Convert Help
Editor
Build SLR(1) Parse
Do Selected
Do Step
Do All
Next
Parse
S' → S
S → i(b)S
S → i(b)SeS
S → x=y
Fill entries in parse table.
FIRST
FOLLOW
S
{ x, i }
{ $, e }
( ) = b e i x y $ S

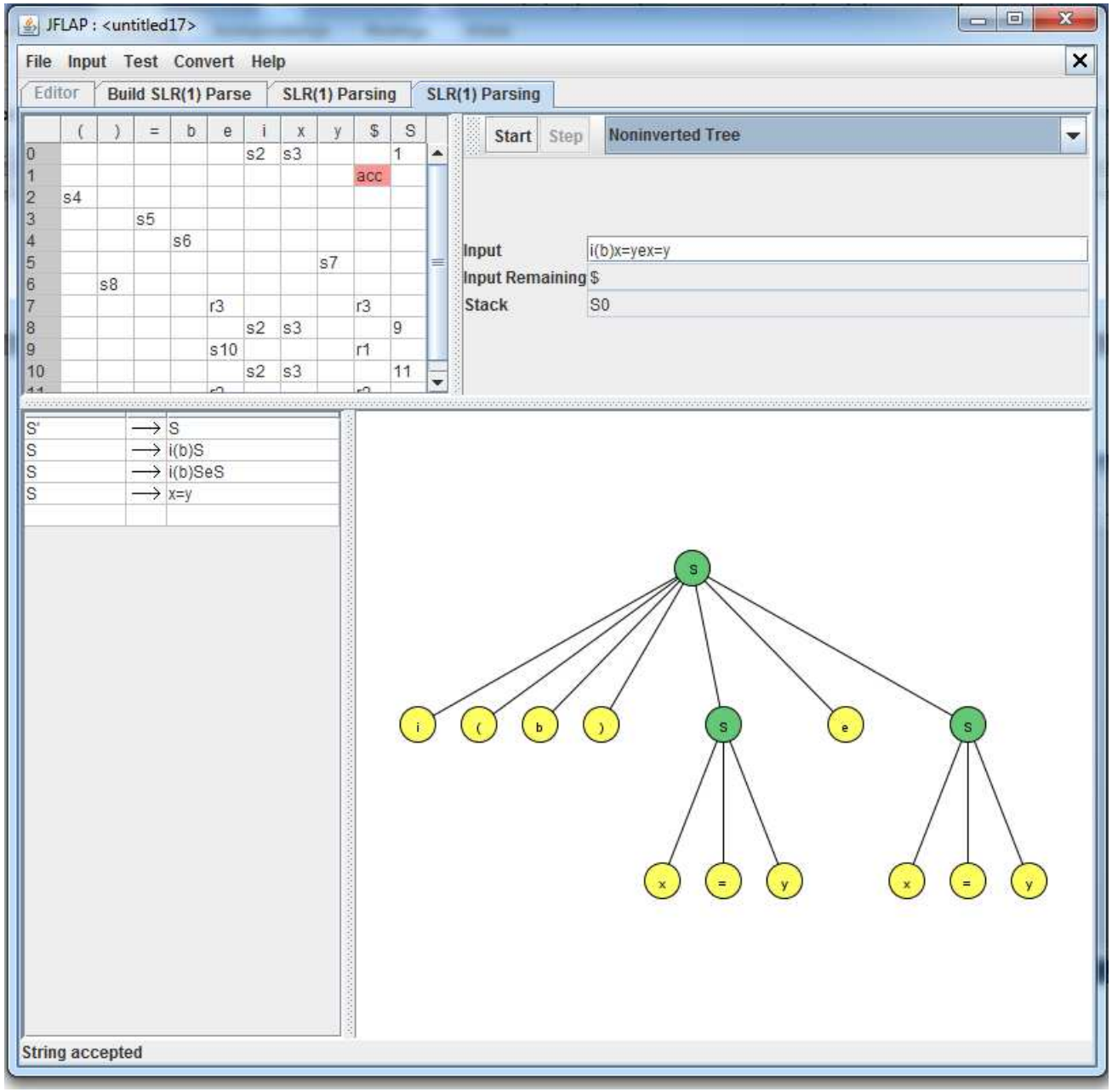

q1
S'→S·
S
q0
S→·i(b)SeS
S→·i(b)S
S'→·S
S→·x=y
i
x
q7
S→x=y·
y
q5
S→x=·y
=
q3
S→x·=y
q4
S→i(·b)S
S→i(·b)SeS
(
q2
S→i·(b)S
S→i·(b)SeS
b
q6
S→i(b·)S
S→i(b·)SeS
)
i
x
q8
S→i(b)·SeS
S→·i(b)SeS
S→·i(b)S
S→·x=y
S→i(b)·S
S
e
q10
S→·i(b)SeS
S→·i(b)S
S→·x=y
S→i(b)Se·S
S
q11
S→i(b)SeS·
q9
S→i(b)S·eS
S→i(b)S·

	(	)	=	b	e	i	x	y	$	S
0						s2	s3			1
1									acc	
2	s4									
3			s5							
4				s6						
5								s7		
6		s8								
7					r3				r3	
8						s2	s3			9
9					r1				r1	
10					r1	s2	s3			11
11					s10				r2	

– standardowe rozwiązanie konfliktów powoduje „naturalne” zachowanie się parsera


JFLAP : <untitled17>
File Input Test Convert Help
Editor
Build SLR(1) Parse
SLR(1) Parsing
SLR(1) Parsing
( ) = b e i x y $ S
0 s2 s3 1
1 acc
2 s4
3 s5
4 s6
5 s7
6 s8
7 r3 r3
8 s2 s3 9
9 s10 r1
10 s2 s3 11
Start
Step
Noninverted Tree
Input i(b)x=yex=y
Input Remaining i(b)x=yex=y$
Stack 0
Shift current input and state 10 to stack
S' → S
S → i(b)S
S → i(b)SeS
S → x=y


JFLAP : <untitled17>
File Input Test Convert Help
Editor
Build SLR(1) Parse
SLR(1) Parsing
SLR(1) Parsing
( ) = b e i x y $ S
0 s2 s3 1
1 acc
2 s4
3 s5
4 s6
5 s7
6 s8
7 r3 r3
8 s2 s3 9
9 s10 r1
10 s2 s3 11
Start
Step
Noninverted Tree
Input i(b)x=yex=y
Input Remaining x=y$
Stack 10e9S8)6b4(2i0
S' ⟶ S
S ⟶ i(b)S
S ⟶ i(b)SeS
S ⟶ x=y
i (b) S e x = y
Shifting e


JFLAP : <untitled17>
File Input Test Convert Help
Editor
Build SLR(1) Parse
SLR(1) Parsing
SLR(1) Parsing
( ) = b e i x y $ S
0 s2 s3 1
1 acc
2 s4
3 s5
4 s6
5 s7
6 s8
7 r3 r3
8 s2 s3 9
9 s10 r1
10 s2 s3 11
Start
Step
Noninverted Tree
Input i(b)x=yex=y
Input Remaining $
Stack S0
S' ⟶ S
S ⟶ i(b)S
S ⟶ i(b)SeS
S ⟶ x=y
S
i (b) S e S
x = y x = y
String accepted

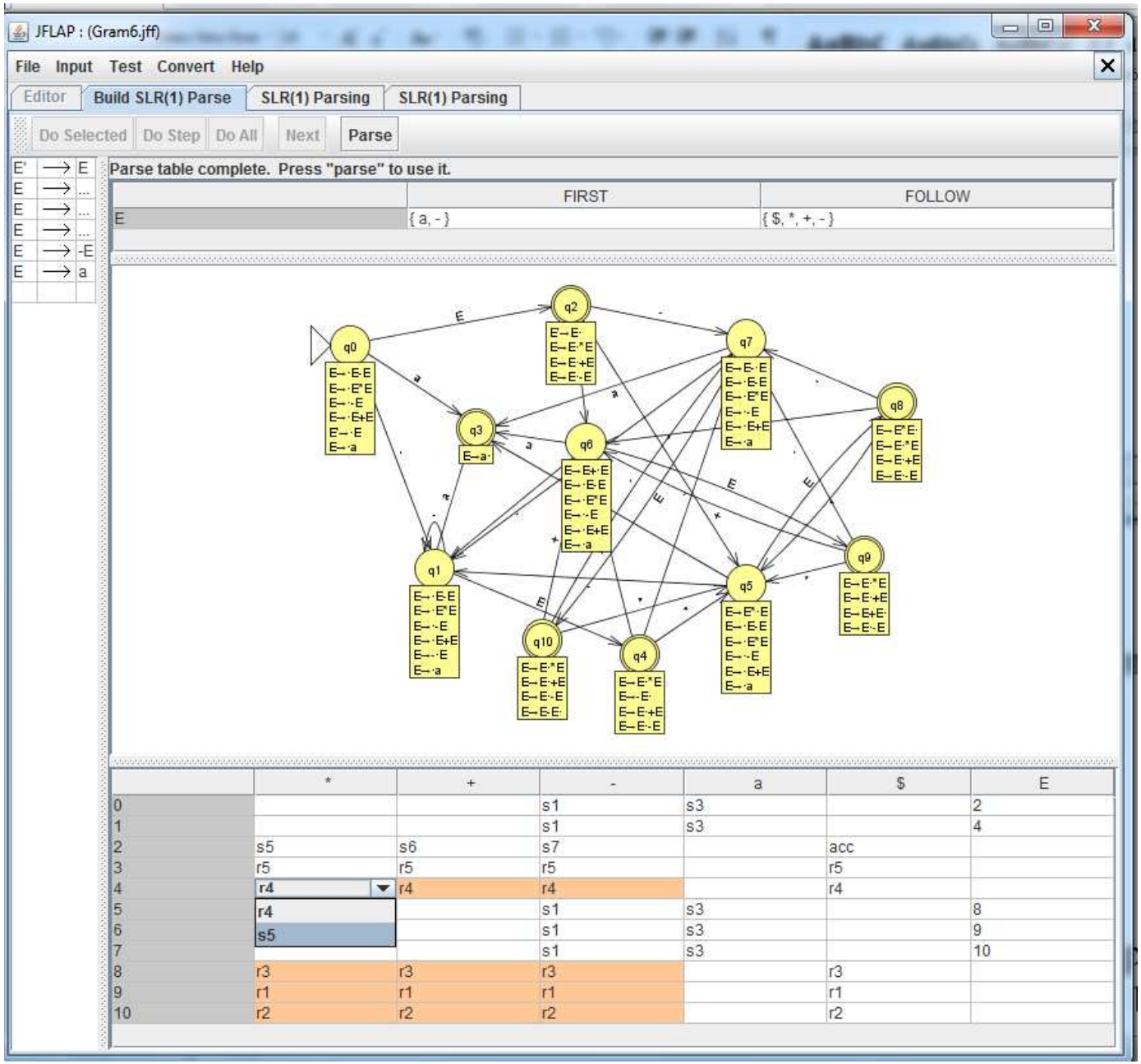
– w YACCu programista może wpływać na ***zmianę standardowego*** sposobu ***rozwiązywania konfliktów*** poprzez zastosowanie ***deklaracji***: `%left`, `%right`, `%nonassoc`, `%prec`

– przykład niejednoznacznej gramatyki wyrażeń arytmetycznych:

```
%token id
%left '+' '-'
%left '*'
%left UMINUS
%%
E  : E '+' E
   | E '-' E
   | E '*' E
   | '-' E %prec UMINUS
   | id
   ;
```

```
root@boromir:/home/lab/labadm/jcyb# cat gram2.y
%token id
%%
E: E '+' E
 | E '-' E
 | E '*' E
 | '-' E
 | id
;

root@boromir:/home/lab/labadm/jcyb# cat scan2.l
%{
#include "gram2.h"
%}
%%
[a-z][a-z0-9]*  { return(id);}
"+"             { return('+');}
"-"             {return('-');}
"*"             {return('*');}
\ |\n|\t         ;
.               {printf("%c - Unexpected character!\n",yytext[0]); exit(0);}

root@boromir:/home/lab/labadm/jcyb# yacc -d -o gram2.c gram2.y
gram2.y: warning: 12 shift/reduce conflicts [-Wconflicts-sr]
root@boromir:/home/lab/labadm/jcyb# lex -o scan2.c scan2.l
root@boromir:/home/lab/labadm/jcyb# cc gram2.c scan2.c gram2.h -ly -ll
root@boromir:/home/lab/labadm/jcyb# cat we
a1+b2+c3*d4+a
root@boromir:/home/lab/labadm/jcyb# ./a.out<we
root@boromir:/home/lab/labadm/jcyb# cat we1
aad2+d4*
root@boromir:/home/lab/labadm/jcyb# ./a.out<we1
syntax error
root@boromir:/home/lab/labadm/jcyb# cat we3
ab2v+&
root@boromir:/home/lab/labadm/jcyb# ./a.out<we3
& - Unexpected character!
```

```
root@boromir:/home/lab/labadm/jcyb# cat gram1.y
%token id
%left '+' '-'
%left '*'
%left UMINUS
%%
E: E '+' E
 | E '-' E
 | E '*' E
 | '-' E %prec UMINUS
 | id
;

root@boromir:/home/lab/labadm/jcyb# cat scan1.l
%{
#include "gram2.h"
%}
%%
[a-z][a-z0-9]*  { return(id);}
"+"             { return('+');}
"-"             {return('-');}
"*"             {return('*');}
\ |\n|\t         ;
.               {printf("%c - Unexpected character!\n",yytext[0]); exit(0);}

root@boromir:/home/lab/labadm/jcyb# yacc -d -o gram1.c gram1.y
root@boromir:/home/lab/labadm/jcyb# lex -o scan1.c scan1.l
root@boromir:/home/lab/labadm/jcyb# cc gram1.c scan1.c gram1.h -ly -ll
root@boromir:/home/lab/labadm/jcyb# cat we
a1+b2+c3*d4+a
root@boromir:/home/lab/labadm/jcyb# ./a.out<we
root@boromir:/home/lab/labadm/jcyb# cat we1
aad2+d4*
root@boromir:/home/lab/labadm/jcyb# ./a.out<we1
syntax error
root@boromir:/home/lab/labadm/jcyb# cat we3
ab2v+&
root@boromir:/home/lab/labadm/jcyb# ./a.out<we3
& - Unexpected character!
```


JFLAP : (Gram6.jff)
File Input Test Convert Help
Editor Build SLR(1) Parse SLR(1) Parsing SLR(1) Parsing
Do Selected Do Step Do All Next Parse
E' → E
E → ...
E → ...
E → ...
E → -E
E → a
Parse table complete. Press "parse" to use it.
FIRST FOLLOW
E { a, - } { $, *, +, - }
q0 q1 q2 q3 q4 q5 q6 q7 q8 q9 q10
E
a
r4
s5

9. Gramatyki i języki typu LL

- ***gramatyki typu LL*** (LL(k), k w praktyce przyjmuje wartość 1) generują podklasę właściwą deterministycznych języków bezkontekstowych
- ***nazwa*** gramatyk ***LL***(1) oznacza, że podczas wywodzenia słów w tej gramatyce, ciąg wejściowy jest przeglądany od ***strony lewej do prawej***, a tworzony wywód jest ***wywodem lewostronnym***, z „podglądem” wejścia na 1 symbol do przodu
- języki generowane gramatykami typu LL(1) są akceptowane przez deterministyczne automaty ze stosem, które realizują strategię rozbioru typu „z góry na dół” („top-down”); rozważana strategia wymaga, aby gramatyka nie była lewostronnie rekurencyjna
- strategię „z góry na dół” dla języków generowanych gramatykami typu LL(1) ***reprezentuje analiza predykcyjna metodą „zejść rekurencyjnych”*** (ang. *predictive recursive-descent parsing*), w której do podjęcia właściwej akcji wystarcza dostęp (bez wczytywania) do następnej jednostki leksykalnej (tzw. „podgląd” – ang. *lookahead*)

- ***automat predykcyjny*** składa się ze zbioru procedur (funkcji), z których każda związana jest z nieterminalem gramatyki

- każda procedura realizuje dwa zadania:

1) wybiera produkcję (o określonej przez nieterminal lewej stronie), dla której symbol „podglądany” należy do zbioru symboli, które można wywieść z prawej strony rozważanej produkcji (np. dla produkcji postaci $\mathbf{A} \rightarrow \alpha$, symbol podglądany musi należeć do FIRST(α))

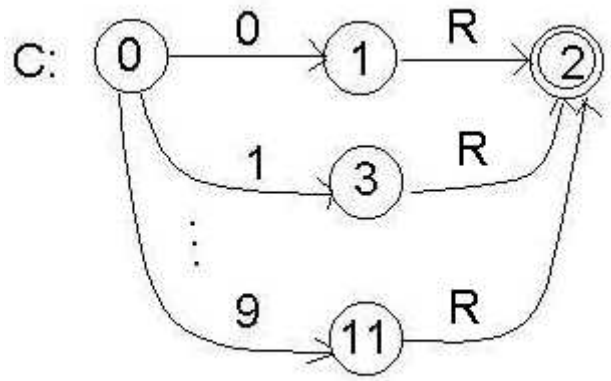
2) „symuluje” działanie prawej strony wybranej produkcji w ten sposób, ze jeśli w produkcji występuje nieterminal, to następuje wywołanie procedury (funkcji) związanej z tym nieterminalem, natomiast wystąpieniu symbolu terminalnego towarzyszy wczytanie symbolu terminalnego z wejścia i sprawdzenie, czy symbole te są identyczne – jeśli nie, to znaczy, że wystąpił błąd składniowy

- **<u>przykład</u>** (automat-akceptor języka generowanego gramatyką:

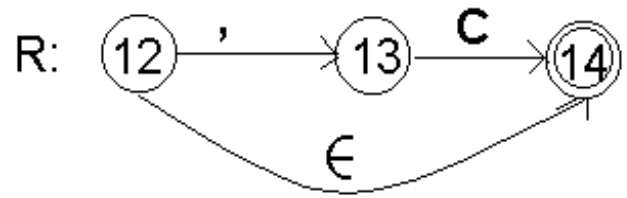
G_{P1}=<{C,R},{**,**,0,1,...,9}, {C $\rightarrow$ 0 R | 1R |...| 9R, R$\rightarrow$**,**C | ε}, C >)

pomocnicze diagramy (Conway'a):

a) dla nieterminala C



a) dla nieterminala R



funkcje pomocnicze (plik „buff-lex.c”):

```
#include <stdio.h>
#define Error 0
#define OK 1
int CurrentToken;
void InitLex()
{
  CurrentToken=yylex();
  return;
}
int LookAhead()
{
  return CurrentToken;
}
int Found(int Token)
{
  int t;
  t=CurrentToken;
  CurrentToken=yylex();
  return t==Token;
}
```

```
void GetNext()
{ CurrentToken=yylex();
 return; }
```

automat-akceptor:

```
#include "an-leks1.c"
#include "buff-lex.c"
int C()
{
  if (Found(d)) return R();
  else return Error;
}
int R()
{
 if (LookAhead()==',')
  {
   GetNext();
   return C();
  }
 else if (LookAhead()=='\n')return OK;
      else return Error;
```

```
}
void main()
{
  InitLex();
  if (C()) printf ("OK!!!\n");
  else printf("Error!!!\n");
}
```