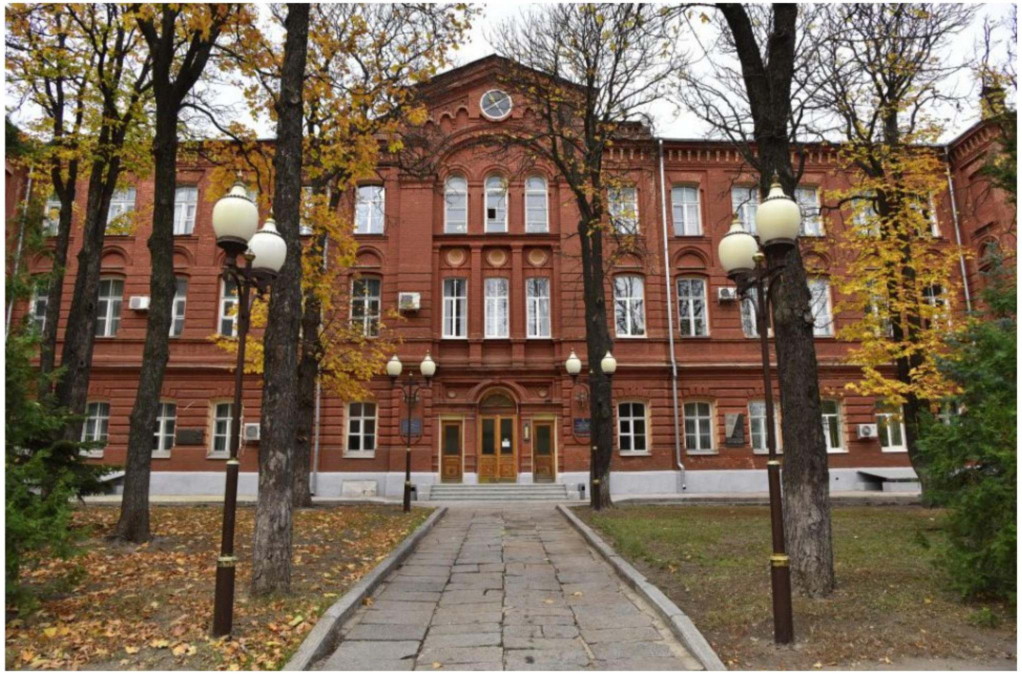
NATIONAL TECHNICAL UNIVERSITY
«KHARKIV POLYTECHNIC INSTITUTE»
Department of Computer Engineering and Programming

# Compiler Design Theory

*Practical lesson 7*

## Push-down automata



Prof. Gavrylenko Svitlana Yuryivna
+380664088551 (Viber)
+380632864663 (Telegram)

Svitlana.Gavrylenko@khpi.edu.ua
Evening building, study # 306

# Problem statement

**There are example of string:**

- *Struct a { int b; }*
- *Struct a {float a; int b; }*
- *Struct a {float a; int b; int c; }.*

1. Identify value four objects of formal grammar G ={$V_T$, $V_A$, $I \in V_A$, R}. Create production rules. Make a derivation of a string.
2. Check this grammar for having non-generating and non-reachable symbols in the production rules.
3. Create Functions FIRST(μ), Functions FOLLOW(A) and SELECTION (μ) sets.
4. Construct transition functions. Make stack implementation of predictive parsing for any string.

# Building grammar productions

1) Write some examples of the input strings.

2) Analyze the structure of the strings, picked out beginning, end, repeated symbols or group of symbols.

3) Introduce notations for complex structures consisting of groups of symbols; such notations are non-terminal symbols of the desired grammar.

4) Build production rule for each of the selected structures, using recursive production to specify repeated structures.

5) Combine all the production rule.

6) Check the possibility of obtaining strings with different structures by derivation.

# Example 1. Part 1

Objects of formal grammar:

$V_T$ = {struct, {, }, *a*, *b*, *c*, “;’” int, float},

$V_A$ = {*I*, *T*, *A*, *S* ,*E* ,*R* }

Production rules:

1) I → struct A{S}

2) A → a | b | c

3) S → ER

4) E → TA;

5) T → int | float

6) R → ER | $ (=ε)

Struct a { int b; }

Struct b { float a; int b; }

Struct c { float a; int b; float c; }.

Derivation the string: Struct a{ int b;}.

I ⇒ struct A{S} ⇒ struct a{S} ⇒ struct a{ER} ⇒ struct a{TA; R} ⇒ struct a{int A; R} ⇒ struct a{int b; R} ⇒ struct a{int b;}

# Non-generating symbols. Part 1

A symbol $X \in V_A$ is defined as non-generating if no finite terminal symbol can be derived from it.

The procedure for detecting non-generating symbols:

1. Make a list of non-terminal symbols, for which there's at least one production rule that doesn't have a non-terminal symbols in the right part.
2. If a production rule is found, in which all non-terminals in the right part are already in the list, add the non-terminal from the left part to the list.
3. Once no more non-terminals can be added to the list from step 2, the list contains all generating symbols of the grammar, and the non-terminals that are not in the list are non-generating symbols.

*Production rules containing non-generating symbols should be eliminated from the grammar.*

# Non-generating symbols. Part 2

The procedure for detecting non-generating symbols:

1. Make a list of non-terminal symbols, for which there's at least one production rule that doesn't have a non-terminal symbols in the right part.
2. If a production rule is found, in which all non-terminals in the right part are already in the list, add the non-terminal from the left part to the list.
3. Once no more non-terminals can be added to the list from step 2, the list contains all generating symbols of the grammar, and the non-terminals that are not in the list are non-generating symbols.

1) $I \rightarrow$ struct $A\{S\}$
2) $A \rightarrow a \mid b \mid c$
3) $S \rightarrow ER$
4) $E \rightarrow TA;$
5) $T \rightarrow$ int | float
6) $R \rightarrow ER \mid \$$

Non-generating symbols

1. A, T, R
2. A, T, R, E
3. A, T, R, E,S
4. A, T, R, E,S, I

There are no Non-generating symbol.

# Unreachable symbols. Part 1

Symbol $\boldsymbol{X} \in \boldsymbol{V_A}$ is called unreachable in context-free grammar G, if $X$ does not appear in any generated string.

The procedure for detecting unreachable symbols:

1. Create a single-element list that contains the initial symbol I of the grammar.
2. If there is a production rule for which the left non-terminal symbol is already in the list, add to the list all symbols of the right part of this production rule.
3. Once no more non-terminals can be added to the list from step 2, the list contains all reachable symbols. Non-terminals symbols that aren't in the list are unreachable.

*Production rules containing unreachable symbols should be eliminated from the grammar.*

# Unreachable symbols. Part 2

The procedure for detecting unreachable symbols:

1. Create a single-element list that contains the initial symbol I of the grammar.
2. If there is a production rule for which the left non-terminal symbol is already in the list, add to the list all symbols of the right part of this production rule.
3. Once no more non-terminals can be added to the list from step 2, the list contains all reachable symbols. Non-terminals symbols that aren't in the list are unreachable.

1) $\mathrm{I} \rightarrow \mathrm{struct}\ \mathrm{A}\{\mathrm{S}\}$
2) $\mathrm{A} \rightarrow a \mid b \mid c$
3) $\mathrm{S} \rightarrow \mathrm{ER}$
4) $\mathrm{E} \rightarrow \mathrm{TA};$
5) $\mathrm{T} \rightarrow \mathrm{int} \mid \mathrm{float}$
6) $\mathrm{R} \rightarrow \mathrm{ER} \mid \$$

Unreachable symbols

1. I
2. I, A, S
3. I, A, S, E, R
4. I, A, S, E, R, T

There are no Unreachable symbols.

# Function FIRST(μ)

FIRST (μ) is a set of terminal symbols that begins a string μ under a derivation.

If **μ** is a string of grammar symbols, than FIRST($A \rightarrow$ **μ**) is the set of terminal symbols that begin the strings derived from **μ**:

- if the string μ starts with the terminal symbol ($A \rightarrow b\mu'$):

$$FIRST(A \rightarrow b\mu') = \{b\};$$

- if the string μ is an empty ( $A \rightarrow \$$) :

$$FIRST(A \rightarrow \$) = \{\$\};$$

- if the string μ starts with a non-terminal symbol B ($A \rightarrow B\mu'$), and there are than some productions for B :

$$B \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n ,$$

and there is no derivation $B \Rightarrow^* \$$, then:

$$FIRST(A \rightarrow B\mu') = FIRST(\alpha_1) \cup FIRST(\alpha_2) \cup \dots \cup FIRST(\alpha_n);$$

- if the string μ starts with a non-terminal symbol B ($A \rightarrow B\mu'$), and there are next productions rules:

$$B \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n ,$$

and there is derivation $B \Rightarrow^* \$$, then:

$$FIRST(A \rightarrow B\mu') = FIRST(\mu') \cup FIRST(\alpha_1) \cup FIRST(\alpha_2) \cup \dots \cup FIRST(\alpha_n).$$

# Example 1. Part 3

1. I $\rightarrow$ struct A{S}
2. A $\rightarrow$ *a* | *b*| *c*
3. S $\rightarrow$ ER
4. E $\rightarrow$ TA;
5. T $\rightarrow$ int | float
6. R $\rightarrow$ ER | $

FIRST (I $\rightarrow$ struct A{S})= FIRST (1) = {struct}

FIRST (2.1) = {*a*}; FIRST (2.2) = {*b*}; FIRST (2.3) = {*c*}

FIRST (5.1) = {int}; FIRST (5.2) = {float}

FIRST (6.2) = {$}

FIRST (3) = FIRST (E)= FIRST (T)= {int, float}

FIRST (4) = FIRST (T) = {int, float}

FIRST (6.1) = FIRST (E)= FIRST (T)= {int, float}

# **Function** FOLLOW$(B)$

FOLLOW(B), for nonterminal B, to be the set of terminals $a$ that can appear immediately to the right of B in the derivation started from initial symbol $I$:
if there are production rules:

$$X_1 \rightarrow \mu_1 B\, \alpha_1,\;\; X_2 \rightarrow \mu_2 B\, \alpha_2,\; \dots ,\; X_n \rightarrow \mu_n B\, \alpha_n,$$

- there is no derivation $\alpha_i \Rightarrow^* \$$, then

$$\text{FOLLOW}(B) = \text{FIRST}(\alpha_1) \cup \text{FIRST}(\alpha_2) \cup \dots \cup \text{FIRST}(\alpha_n).$$

- there is derivation $\alpha_i \Rightarrow^* \$$, for example $\alpha_1 \rightarrow \$$, :

$$\text{FOLLOW}(B) = \text{FOLLOW}(X_1) \cup \text{FIRST}(\alpha_2) \cup \dots \cup \text{FIRST}(\alpha_n).$$

# Example 1. Part 4

1) I → struct A{S}
2) A → *a* | *b*| *c*
3) S → ER
4) E → TA;
5) T → int | float
6) R → ER | $

FIRST (1) = {struct}

FIRST (2.1) = {*a*}; FIRST (2.2) = {*b*}; FIRST (2.3) = {*c*}

FIRST (5.1) = {int}; FIRST (5.2) = {float}

FIRST (6.2) = {$}

FIRST (3) = FIRST (E)= FIRST (T)= {int, float}

FIRST (4) = FIRST (T) = {int, float}

FIRST (6.1) = FIRST (E)= FIRST (T)= {int, float}

FOLLOW (A)= {{, ;}

FOLLOW (S)= {}}

FOLLOW (E)= FIRST (R) ∪ FOLLOW (S) = FIRST (E) ∪

∪ FOLLOW (S) = FIRST (T) ∪ FOLLOW (S) = {int, float, }}

Because there is production R→$

FOLLOW (R )= FOLLOW (S)= {}} because R is the last symbol

FOLLOW (T)= FIRST (A) = {a, b, c};

# Set SELECTION($\mu$)

1) If there is production rule $B \rightarrow \alpha$ and there is no derivation $\alpha \rightarrow \$$, then

$$\text{SELECTION}(B \rightarrow \alpha) = \text{FIRST}(\alpha).$$

2) If there is production rule $B \rightarrow \$$, then

$$\text{SELECTION}(B \rightarrow \$) = \text{FOLLOW}(B).$$

3) If there is production rule $B \rightarrow \alpha$ and there is derivation $\alpha \rightarrow \$$, then

$$\text{SELECTION}(B \rightarrow \alpha) = \text{FIRST}(\mu) \cup \text{FOLLOW}(B).$$

# Example 1. Part 5

1) $I \rightarrow$ struct A{S}

2) $A \rightarrow$ *a* | *b*| *c*

3) $S \rightarrow ER$

4) $E \rightarrow TA$;

5) $T \rightarrow$ int | float

6) $R \rightarrow ER$ | $

FIRST (5.1) = {int}; FIRST (5.2) = {float}

FIRST (2.1) = {*a*}; FIRST (2.2) = {*b*}; FIRST (2.3) = {*c*}

FIRST (1) = {struct}

FIRST (6.1) = FIRST (E)= FIRST (T)= {int, float}

FIRST (6.2) = {$}

FIRST (3) = FIRST (E)= FIRST (T)= {int, float}

FIRST (4) = FIRST (T) = {int, float}

FOLLOW (R)= {}}

SELECTION (1) = FIRST (1) = {struct}

SELECTION (2.1) = FIRST (2.1) = {*a*}

SELECTION (2.2) = FIRST (2.2) = {*b*};

SELECTION (2.3) = FIRST (2.3) = {c}

SELECTION (3) = FIRST (E)= FIRST (T)= {int, float}

SELECTION (4) = FIRST (4)= FIRST (T) = {int, float}

SELECTION (5.1) = FIRST (5.1) = {int}

SELECTION (5.2) = FIRST (5.2) = {float}

SELECTION (6.1) = FIRST (6.1) = {int, float};

SELECTION (6.2) = FOLLOW (R) = {}}

This grammar is an LL(1) grammar, since the SELECTION set for rules starting with the same terminals does not contain the same characters.

# Construction transition functions

1) For each production rule $A \rightarrow a\alpha$, starting with a terminal symbol $a$, construct a transition function:

$$f\,(\,s\,,\,a\,,\,A\,) = (\,s\,,\,\alpha'\,)\,,$$ where $\alpha'$ is the mirror image of the string $\alpha$.

2) For each production rule $A \rightarrow B\alpha$, starting with a non-terminal symbol $B$, construct a transition function:

$$f^*\,(\,s\,,\,x\,,\,A\,) = (\,s\,,\,\alpha'\,B)$$

where $f^*$ is a transition function without shifting an input head, a $\alpha'$ is a mirror image of the string $\alpha$, $x \in$ **SELECTION** $(A \rightarrow B\alpha)$ set. The number of transition functions determines by the number of the $x \in$ **SELECTION** $(A \rightarrow \$)$.

3) For each empty production rule $A \rightarrow \$$ construct a transition function:

$$f^*\,(\,s\,,\,x\,,\,A\,) = (\,s\,,\,\$\,)$$

where $f^*$ is a transition function without shifting an input head, a $\alpha'$ is a mirror image of the string $\alpha$, $x \in$ **SELECTION** $(A \rightarrow \$)$ set. The number of transition functions determines by the number of the $x \in$ **SELECTION** $(A \rightarrow \$)$.

4) For all terminal symbol (for example, symbol b), that appears in the middle or at the end of production rules, create a transition function:

$$f\,(\,s\,,\,b\,,\,b\,) = (\,s\,,\,\$\,).$$

5) To get a final state of push-down automata create a transition function:

$$f^*\,(\,s\,,\,\$\,,\,h_0\,) = (\,s\,,\,\$\,,\,\$\,).$$

# Example 1. Part 6

1) I → struct A{S}
2) A → *a* | *b*| *c*
3) S → ER
4) E → TA;
5) T → int | float
6) R → ER | $

SELECTION (5.1) = FIRST (5.1) = {int}
SELECTION (5.2) = FIRST (5.2) = {float}
SELECTION (2.1) = FIRST (2.1) = {*a*}
SELECTION (2.2) = FIRST (2.2) = {*b*};
SELECTION (2.3) = FIRST (2.3) = {*c*}
SELECTION (3) = FIRST (E)= FIRST (T)= {int, float}
SELECTION (6.1) = FIRST (6.1) = {int, float};
SELECTION (6.2) = FOLLOW (R) = {}}
SELECTION (4) = FIRST (4) = FIRST (T) = {int, float}
SELECTION (1) = FIRST (1) = {struct}

| | |
|---|---|
| 1. *f (s, struct , I) = (s, }S{A)* | 11. *f(s, ; , ;) = (s, $)* |
| 2. *f(s, } , }) = (s, $)* | 12. *f (s, float , T) = (s, $)* |
| 3. *f(s, { , {) = (s, $)* | 13. *f (s, int , T) = (s, $)* |
| 4. *f(s, a , A) = (s, $)* | 14. *f *(s, int , R) = (s, RE)* |
| 5. *f(s, b , A) = (s, $)* | 15. *f *(s, float , R) = (s, RE)* |
| 6. *f(s, c , A) = (s, $)* | 16. *f *(s, } , R) = (s, $)* |
| 7. *f*(s, int , S) = (s, RE)* | 17. *f*(s, $ , $h_0$) = (s, $).* |
| 8. *f*(s, float, S) = (s, RE)* | |
| 9. *f*(s, int, E) = (s, ;AT)* | |
| 10. *f*(s, float, E) = (s, ;AT)* | |

# Example 4. Part 7

| Transition functions of the push-down automata | | Stack implementation of predictive parsing for string: | |
|---|---|---|---|
| 1. *f (s, struct , I) =* *(s, }S{A)* | *11. f(s, ; , ;) = (s, \$)* | *(s, struct b { float a;, int b;},* *$h_0I$ )* ├1 *start configuration* | *(s, int b;}, $h_0$}R)* ├ 14 |
| *2. f(s, } , }) = (s, \$)* | *12. f (s, float , T) = (s, \$)* | *(s, b{ float a; int b;}, $h_0$}S{A )* ├ 5 | *(s, int b;}, $h_0$}RE)* ├ 9 |
| *3. f(s, { , {) = (s, \$)* | *13. f (s, int , T) = (s, \$)* | *(s, { float a; int b;}, $h_0$}S{ )* ├ 3 | (s, int b;}, $h_0$}R;AT) ├ 13 |
| *4. f(s, a , A) = (s, \$)* | *14. f *(s, int , R) = (s, RE)* | (s, float a; int b;}, $h_0$}S ) ├ 8 | *(s, b;}, $h_0$}R;A)* ├ 5 |
| *5. f(s, b , A) = (s, \$)* | *15. f *(s, float , R) = (s, RE)* | (s, float a; int b;}, $h_0$}RE ) ├ 10 | *(s, ;}, $h_0$}R;)* ├ 11 |
| *6. f(s, c , A) = (s, \$)* | *16. f *(s, } , R) = (s, \$)* | (s, float a; int b;}, $h_0$}R;AT ) ├ 12 | *(s, }, $h_0$}R )* ├ 16 |
| *7. f*(s, int , S) = (s, RE)* | *17. f*(s, \$ , $h_0$) = (s, \$).* | *(s, a; int b;}, $h_0$}R;A )* ├ 4 | *(s, }, $h_0$} )* ├ 2 |
| *8. f*(s, float, S) = (s, RE)* | | *(s, ; int b;}, $h_0$}R;)* ├ 11 | *(s, \$, $h_0$)* ├ 17 |
| *9. f*(s, int, E) = (s, ;AT)* | | | *(s,\$, \$)* |
| *10. f*(s, float, E) =* *(s, ;AT)* | | | *finish configuration* |

**Thank you for your attention!**