МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

О.І. Марченко, О.О. Марченко

ОСНОВИ ПРОЕКТУВАННЯ ТРАНСЛЯТОРІВ

ІНСТРУКЦІЇ ТА ЗАВДАННЯ ДО ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ ТА РГР

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського як навчальний посібник для здобувачів ступеня бакалавра за освітньо-професійною програмою підготовки бакалаврів спеціальності 123 – «Комп'ютерна інженерія»

Київ

КПІ ім. Ігоря Сікорського

2021

Рецензент: *Заболотня Т.М.,* канд. техн. наук, доцент, доцент, КПІ ім. Ігоря Сікорського

Відповідальний редактор: *Орлова М.М.,* канд. техн. наук, доцент, доцент

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол № 2 від 09.12.2021р.) за поданням Вченої ради факультету прикладної математики (протокол №2 від 27.09.2021р.)

Електронне мережне навчальне видання

Марченко Олександр Іванович, канд. техн. наук, доцент
Марченко Олексій Олександрович, асистент

ОСНОВИ ПРОЕКТУВАННЯ ТРАНСЛЯТОРІВ
ІНСТРУКЦІЇ ТА ЗАВДАННЯ ДО ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ ТА РГР

Основи проектування трансляторів: Інструкції та завдання до виконання лабораторних робіт та РГР з дисципліни «Основи проектування трансляторів» : [Електронний ресурс] : навч. посіб. для студ. спеціальності 123 – «Комп'ютерна інженерія» / О. І. Марченко, О. О. Марченко ; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 1,08 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2021. – 118 с.

Навчальний посібник розроблено для ознайомлення студентів із завданнями дисципліни «Основи проектування трансляторів». Навчальний посібник містить інформацію до виконання проєктів програм лабораторних робіт та розрахунково-графічної роботи, яка включає постановку завдання для кожної роботи, необхідні теоретичні відомості, вимоги до проектів про-грам, що повинні розробити студенти, вказівки до оформлення звіту та те-стування розроблених алгоритмів і відповідних їм програм, варіанти інди-відуальних завдань, а також наведені контрольні питання для підготовки до захисту лабораторних робіт.

Навчальний посібник призначений для студентів очної форми навчання за спеціальністю 123 – «Комп'ютерна інженерія» факультету прикладної математики КПІ ім. Ігоря Сікорського.

ЗМІСТ

ВСТУП

Дисципліна «Основи проектування трансляторів» є спеціальною нормативною дисципліною у загальній схемі алгоритмічної і програмістської підготовки бакалаврів за спеціальністю 123 – «Комп'ютерна інженерія». Вивченню цієї дисципліни повинні пере-дувати дисципліни "Структури даних та алгоритми", "Програму-вання" та "Системне програмування". Ця дисципліна забезпечує ви-вчення дисциплін „Архітектура комп'ютерів" та „Автоматизація проектування комп'ютерних систем" і призначена для вивчення ос-нов формальних граматик, а також принципів та концепцій, які пок-ладені в основу проектування трансляторів, зокрема їх складових ча-стин: лексичних аналізато-рів, синтаксичних аналізаторів та генера-торів коду.

Цикл семестрових завдань складається з двох комплексних ла-бораторних робіт розробницького характеру та розрахунково-графі-чної роботи і призначений для покриття практичної частини дисци-пліни «Основи проектування трансляторів».

Роботи дозволяють отримати практичний досвід та навички з розробки лексичних аналізаторів, синтаксичних аналізаторів та генераторів коду.

Інформація до кожної роботи включає:

постановку завдання та вимоги до алгоритмів та програм, що повинні розробити студенти;

0 вказівки до оформлення звіту та тестування розроблених програм;

1 методичні вказівки та/або базовий теоретичний матеріал, необхідні для виконання робіт;

5888 контрольні питання для самоконтролю та підготовки до захисту роботи;

0 варіанти індивідуальних завдань.

ЗАГАЛЬНІ ВИМОГИ
ДО ПРОГРАМ ЛАБОРАТОРНИХ РОБІТ ТА РГР

Розробка програм лабораторних робіт та розрахунково-гра-фічної роботи повинна бути виконана мовою C|C++, оскільки транслятори належать до категорії «системне програмне забезпечення», а основною мовою системного програмування де-факто є мова C/C++.

Професійний транслятор повинен працювати максимально швидко наскільки це можливо досягнути при розробці. Тому:

заборонено використовувати будь-які бібліотечні засоби роботи з регулярними виразами;

кожен символ вхідної програми на мові високого рівня **по-винен читатися із файлу і оброблюватися у трансляторі рівно один раз і не більше**: тобто, читання із вхідного файлу лексичним аналізатором (ЛА) повинно виконуватися винятково посимвольно, відразу повинно прийматися рішення автоматом ЛА, що з цим символом робити, і більше повер-нення до читання/обробки цього символа не повинно бути;

заборонено використовувати бібліотеки функцій для обробки рядків (пошук у рядку, виділення підрядків тощо).

0 **Backend лабораторних робіт та РГР** (тобто алгоритмічна частина лексичного аналізатора, синтаксичного аналізатора та генератора коду) **повинен бути строго відокремлений від frontend-у** (інтерфейсної частини роботи з програмою) **і знаходитись у окремих модулях/класах тощо**.

Функції виведення рядка лексем, дерева розбору та інформацій-них таблиць на друк є частиною frontend-у, а не backend-у. Вони повинні викликатись один раз після того як структури даних вже остаточно сформовані, а не бути переплетеними з основною логікою в алгоритмічній частині.

1 Кожна лабораторна робота та РГР повинні бути виконані у вигляді незалежної програми. Кожна наступна робота може містити в собі код попередньої, але не навпаки.

2 Повідомлення про помилки повинні бути якнайбільш змістовними, містити в собі всі дані про помилку, які є на момент її виникнення: стадія транслятора (Lexer, Parser, Code Generator), тип помилки (Error, Warning), позиція помилки (Line, Column), текст повідомлення про помилку.

Наприклад:

Lexer: Error (Line 2, Column 1): Illegal character '^' detected.
Parser: Error (Line 10, Column 15): <identifier> expected but ';' found.
Code Generator: Error (Line 2, Column 1): Incompatible types of variables 'i:integer' and 's:string' in the assignment.

0 На етапі лексичного аналізу (лабораторна робота №1) для кож-ної лексеми (токена) у рядку лексем (токенів) повинна зберіга-тися позиція їх знаходження в оригінальному коді програми, що транслюється, тобто номер рядка та колонки першого символу лексеми (токена), так, як показано на лекційній схемі, що пока-зує загальних принцип роботи лексичного аналізатора.

Позиції лексем нумеруються **від одиниці, а не від нуля**. Син-таксичний аналізатор та генератор коду повинні використову-вати ці позиції для виведення повідоьлень про помилки. Звер-ніть увагу на те, що колонка першого символу лексеми (токена) — це **не** те ж саме, що порядковий номер лексеми (токена) в межах одного рядка.

ЗАГАЛЬНІ ВИМОГИ ДО ТЕСТІВ ТА ТЕСТУВАННЯ

0 Кожен тест містить: 1) код мовою SIGNAL; 2) результат роботи програми, що тестується (лексичний аналізатор, синтаксичний аналізатор, генератор коду), для цього тесту.

1 Тести повинні бути двох категорій:

  0 тести, що демонструють правильну роботу програми при правильному вхідному коді тесту (True-тести).

  1 тести, що демонструють коректну роботу програми при неправильній вхідній програмі (False-тести), тобто виведення правильного повідомлення про помилку з вказанням правильного місця (рядок, колонка) цієї помилки.

2 True-тести повинні покривати різні варіанти правильного вико-ристання конструкцій граматики свого варіанту.

3 False-тести повинні покривати всі можливі типи помилок у кон-струкціях лексики, синтаксису та семантики граматики свого варіанту.

4 На захист лабораторних робіт та розрахунково-графічної роботи потрібно приходити з вже підготовленими тестами обох категорій, що повністю покривають лексику (ЛР №1), синтаксис (РГР) та семантику (ЛР №2) свого варіанту.

5 Тести на захист повинні надаватись у різних файлах вже у гото-вому вигляді, а не коригуватись під час захисту в одному й тому ж файлі.

6 Кількість підготовлених тестів залежить від варіанту граматики та її семантики.

0 У звітах достатньо надрукувати по 3-5 тестів кожної категорії (True-тести, False-тести) для кожної лабораторної роботи та ро-зрахунково-графічної роботи.

1 У звітах не повинно бути скріншотів вікна консолі з чорним фо-ном. Треба або копіювати текст (бажано), або інвертувати колір скріншоту (https://goo.gl/ki6Amb). В разі використання скрін-шоту вирізайте область з результатом так, щоб весь результат було видно і його можна було прочитати.

Приклади оформлення тестів лабораторної роботи №1 1. True-тест

Вхідна програма	**Рядок лексем**
	Row Col Code Lexem
PROGRAM SIG01;	
BEGIN	1 1 1001 PROGRAM
END.	1 9 2001 SIG01
	1 14 501 ;
	2 1 1002 BEGIN
	3 1 1003 END
	3 4 502 .

Плюс роздруківка сформованих інформаційних таблиць.

2. False-тест

Вхідна програма	**Рядок лексем**
	Row Col Code Lexem
PROGRAM SIG01;	
BEGIN ?	1 1 1001 PROGRAM
END.	1 9 2001 SIG01
	1 14 501 ;
	2 1 1002 BEGIN
	3 1 1003 END
	3 4 502 .

Lexer: Error (line 2, column 7): Illegal symbol ‘?’

Плюс роздруківка сформованих інформаційних таблиць.

Зауваження. Лексичний аналізатор, на відміну від синтаксичного, як правило працює до кінця, а не до першої помилки, і виводить повний рядок лексем, ігноруючи у рядку лексем виведення лексем недопустимих символів. Але повідомлення про недопустимі символи виводиться обов'язково. Хоча варіант реалізації роботи ле-ксичного аналізатора до першої помилки також допускається.

Приклади оформлення тестів для РГР

1. True-тест

Вхідна програма	**Дерево розбору**
PROGRAM SIG01; BEGIN END.	<signal-program> ..<program> ....1001 PROGRAM ....<procedure-identifier> ......<identifier> ..... 2001 SIG01 ....501 ; ....<block> ......<declarations> ..... <variable-declarations> .....<empty> ......1002 BEGIN ......<statements-list> ..... <empty> ......1003 END ....502 .

2. False-тест

Вхідна програма	**Дерево розбору**
PROGRAM SIG01; BEGUN END.	<signal-program> ..<program> ....1001 PROGRAM ....<procedure-identifier> ......<identifier> .....2001 SIG01 ....501 ;

Parser: Error (line 2, column 1): Keyword 'BEGIN' expected

ЛАБОРАТОРНА РОБОТА №1
«РОЗРОБКА ЛЕКСИЧНОГО АНАЛІЗАТОРА»

Мета лабораторної роботи

Метою лабораторної роботи «Розробка лексичного аналізатора» є засвоєння теоретичного матеріалу та набуття практичного досвіду і практичних навичок розробки лексичних аналізаторів (ска-нерів).

Постановка задачі

23 Розробити програму лексичного аналізатора (ЛА) для підмножини мови програмування SIGNAL.

23 Лексичний аналізатор має забезпечувати наступні дії:

5888 видалення (пропускання) пробільних символів: пробіл (код ASCII 32), повернення каретки (код ASCII 13); перехід на новий ря-док (код ASCII 10), горизонтальна та вертикальна табуляція (коди ASCII 9 та 11), перехід на нову сторінку (код ASCII 12);

23 згортання ключових слів;

24 згортання багато-символьних роздільників (якщо передбачаються граматикою варіанту);

5888 згортання констант із занесенням до таблиці значення та типу константи (якщо передбачаються граматикою варіанту);

5889 згортання ідентифікаторів;

• видалення коментарів, заданих у вигляді **(*<текст коментаря>*)**;

0 формування рядка лексем зінформацією про позиції лексем;

1 заповнення таблиць ідентифікаторів та констант інформацією, отриманою підчас згортки лексем;

0 виведення повідомлень про помилки.

Вимоги до програми лексичного аналізатора

0 Входом ЛА має бути наступне:

• вхідна програма, написана підмножиною мови SIGNAL відповідно до варіанту;

0 масив Attributes (див. лекцію з поясненням принципу роботи ЛА), який містить символи таблиці кодів ASCII з відповідними їм атрибутами для визначення лексем (токенів);

← таблиця ключових слів;

← таблиця багато-символьних роздільників (якщо потрібно);

← таблиця констант, в яку, при необхідності, попередньо можуть бути занесені стандартні константи;

← таблиця ідентифікаторів, в яку, при необхідності, попередньо занесені наперед визначені ідентифікатори.

← Виходом ЛА має бути наступне:

← закодований рядок лексем з інформацією про їх розташування у вихідній програмі (номер рядка, номер колонки);

← таблиця констант, що сформована для конкретної програми

← яка містить значення та тип констант;

← таблиця ідентифікаторів, що сформована для конкретної програми;

← інформаційні повідомлення про виявлені помилки.

← Для кодування лексем при їх згортанні необхідно використовувати числові діапазони, вказані в Таблиці 1.

Таблиця 1. Діапазони кодування лексем

Вид лексеми	**Числовий діапазон**
Односимвольні роздільники та знаки операцій (: / ; + тощо)	0–255 (тобто коди ASCII)
Багатосимвольні роздільники (:= <= <= тощо)	301 – 400
Ключові слова (BEGIN, END, FOR тощо)	401 – 500
Константи	501 – 1000
Ідентифікатори	1001– . . .

← ***Рекурсивні*** **алгоритми реалізації лексичного аналізатора** ***не допускаються*** по причині їх катастрофічної глибини рекурсії, яка має лінійну залежність від кількості лексем програми. Для вели-ких програм це неприпустимо.

← **Читання із вихідного файлу, як зазначалося у загальних вимогах, відбувається винятково посимвольно**.

← Позиції повинні коректно визначатися для всіх лексем (токенів); перевіряти можна в текстовому редакторі, наприклад, Notepad. Позиція кожного токена, збережена в рядку токенів, повинна співпадати з фактичною позицією цього токена в текстовому редакторі.

← При визначенні позиції лексем символ табуляції повинен враховуватися як відповідна кількість пробілів у текстовому редакторі. Для універсального підлаштування ЛА до конкретних текстових ре-дакторів в коді ЛА можна, наприклад, ввести опцію відповідності кількості пробілів символу табуляції.

← Позиції лексем (токенів) після багаторядкового коментаря також повинні бути коректними. Наприклад, у наступних двох рядках колонка токена “token” є різною і відрізняється на одну колонку.

```
The last line of comment *)token The last
line of comment *) token
```

← Різні варіанти коментарів повинні оброблюватись коректно, наприклад

- порожні коментарі (**) (* *)
- ← (* багаторядковий
  коментар *)
- (*****) (*(()*())*) (*;.:*)
- ← незакриті коментарі
- ← коментарі із забороненими символами: всередині коментаря заборонені символи ***допускаються***

← Коректна обробка заборонених символів: повинна друкуватись помилка щодо знаходження забороненого символа.

← Коректна обробка суцільних послідовностей роздільників (хоча це й не відповідає більшості граматик), таких як

...;;:(()()());;:=:=:::=;;:.

Наприклад, суцільна послідовність роздільників

:::==

повинна бути розібрана як дві окремих двокрапки, потім одне присвоєння і один знак рівності.

Вимоги до тестів

- Див. розділ «Загальні вимоги до тестів та тестування».
- Тести до лексичного аналізатора повинні покривати всі ситуації пунктів 6-11 розділу «Вимоги до програми лексичного аналіза-тора».

Зміст звіту

Звіт оформлюється згідно вимог до лабораторних робіт і має містити наступне:

- титульний аркуш;
- індивідуальне завдання згідно до варіанту:
  - постановка задачі;
  - номер варіанту;
  - граматика за варіантом в гарно читабельному вигляді
- граф автомату, що визначає алгоритм ЛА:
  - граф в точності повинен відповідати граматиці за варіантом, тобто повинен містити стани для розбору всіх мо-жливих за вашим варіантом токенів і **не** містити станів, які не потрібні в даному варіанті.

- граф повинен містити гілку станів розбору коментаря. Необхідно добре розуміти як ця гілка станів працює, оскільки одним із завдань може бути рисування гілки станів розбору коментаря і пояснення як вона працює;
- граф ЛА конкретного варіанту – це не скріншот графа із конспекту лекцій чи методички, оскільки для конкретного варіанту граф ЛА мйже завжди буде іншим;
- Рекомендація: для рисування графа можна використати, наприклад, http://graphviz.org/ (використовуйте dot) або https://www.draw.io/ ;

← лістинг коду програми ЛА:

- звіт повинен містити **весь** код програми;
- для коду використовувати **моноширинний** шрифт;
- якщо код не широкий, використовуйте форматування у дві колонки (https://goo.gl/wy9ub6);

← контрольні приклади, необхідні для демонстрації всіх конструкцій заданої граматики, а також всіх можливих помилкових ситуацій (опис кожного контрольного прикладу має містити згенерований рядок лексем та заповнені таблиці).

До звіту в електронному вигляді мають бути додані:

← файл з текстом звіту;

← проект працездатної програми на мові програмування;

← завантажувальний модуль програми з необхідними файлами даних.

Методичні вказівки

Для реалізації лексичного аналізатора спочатку виконується категоризація символів таблиці ASCII, тобто кожен символ цієї таб-лиці призначається до певної категорії. Категорії символів будуть використовуватись в автоматі лексичного аналізатора в якості вхід-них символів для переходу з одного стану в наступний. У випадках обробки багатосимвольних роздільників в якості вхідних символів автомата можуть бути використані також власне самі символи таб-лиці ASCII замість їх категорій.

Категоризація символів таблиці ASCII в лексичному аналіза-торі, як правило, виконується у вигляді одномірного масива (век-тора) чисел SymbolCategories, де коди символів таблиці ASCII вико-ристовуються в якості індексів елементів цього масива, а значен-нями елементів масива є категорії відповідних символів, що призна-чаються цим символам.

Приклад категоризації символів.

SymbolCategories – масив категорій символів таблиці ASCII

NUL		BEL	BS		CR		Space		(	)		0		9
0		7	8		13		32		40	41		48		57
6	…	6	0	…	0	…	0	…	5	3	…	1	…	1

:	;	<	=	>		A		Z		a		z		DEL
58	59	60	61	62		65		90		97		122		127
41	3	42	3	43	…	2	…	2	…	2	…	2	…	6

A 65 ← символ таблиці ASCII (в масиві SymbolCategories не використовується)

2 ← код символа таблиці ASCII (індекс масива SymbolCategories)

← категорія символа таблиці ASCII, призначена символу для реалізації лексичного аналізатора

В даному прикладі використані наступні категорії символів ASCII:

← **– категорія пробільних символів (whitespace)**: пробіл (space) –
код 32 та прирівняні до пробіла керуючі символи (як правило, сим-воли з кодами від 8 до 13);

← **– категорія символів**, з яких можуть починатись **числа**;

← **– категорія символів**, з яких можуть починатись **ідентифіка-тори та ключові слова**;

← **– категорія односимвольних роздільників**;

4x (41, 42, …) – група категорій символів, з яких починаються **ба-гатосимвольні роздільники** (на кожен такий роздільник – окрема категорія);

5x (51, 52, …) – група категорій символів, з яких починаються **ба-гатосимвольні роздільники коментарів** (в даному прикладі така категорія одна);

← **– категорія недопустимих символів**.

Розглянемо приклад побудови лексичного аналізатора (ЛА) для граматики умовного оператора, показаної на рис.1.

```
←  <statement> → if <expr> then <expr> |
                 if <expr> then <expr> else <expr> |
                 <empty>
←  <expr> → <term> <operation> <term>
←  <operation> → = | < | <=
←  <term> → <number> | <idn>
←  <number> → <digit> | <digits>
←  <digits> → <digit><digits>
←  <idn> → <let><lets_or_digits>
←  <lets_or_digits> → <let><lets_or_digits> |
                 <digit><lets_or_digits> |
                 <empty>
←  <let> → a | b | ...  | z
←  <digit> →  0 | 1 | ... | 9
```

Рис.1 Граматика умовного оператора

← цій граматиці правила 1–4 складають синтаксичну частину граматики і будуть реалізовані у синтаксичному аналізаторі. Правила 5–10 складають лексичну частину граматики, тобто описують **лексеми (токени)** граматики і повинні бути реалізовані у лексичному аналізаторі. Лексична частина граматики, що визначається правилами 5–10, може бути для зручності записана також у вигляді **регулярних визначень:**

number → `digit*`

identifier → `let (let | digit)*`

let → `[a-z]`

digit → `[0-9]`

Визначимо лексеми (токени), які будуть виділятись лексичним аналізатором. Згідно правил 5–10, для даної граматики такими лексемами будуть «число» (**number**) і «ідентифікатор» (**identifier**) Крім того, до лексем граматики належать також всі її ключові слова (**if**, **then**, **else**) та роздільники (=, <, <=), які присутні безпосередньо в граматиці.

Для розпізнавання цих лексем, виконаємо категоризацію сим-волів ASCII згідно їх використанню в даній граматиці. Основні еле-менти масива SymbolCategories категорій символів ASCII для даної граматики будуть такими:

NUL		BEL	BS		CR		Space		(	)		0		9
0		7	8		13		32		40	41		48		57
6	…	6	0	…	0	…	0	…	5	6	…	1	…	1

:	;	<	=	>		A		Z		A		z		DEL
58	59		60	61	62	65		90		97		122		127
6	6	4	3	6	…	2	…	2	…	2	…	2	…	6

Призначення кодів категоріям символів виконано згідно вищенаведеного прикладу. Символи, з яких може починатися лексема «число» (**number**), належать до категорії 1 (позначимо її символами **dig**). Символи, з яких можуть починатися лексеми «ідентифікатор» та «ключове слово» (**identifier**), належать до категорії 2 (позначимо

← символами **let**).

← даній маленькій граматиці є тільки одна «чиста» лексема типу «односимвольний роздільник» (**delimiter1**) – символ **'='**, тому цей символ належить до категорії 3 (позначимо її символами **dm1**), і тільки одна лексема типу «багатосимвольний роздільник» **'<='** (**delimiter2**), тому символ **'<'**, з якого ця лексема починаєтися, нале-жить до категорії 4 (позначимо її символами **dm2**). Символ **'<'** в яко-сті односимвольного роздільника буде розпізнаватись в рамках об-робки категорії 4.

Крім того, коментарі в даному прикладі будуть обмежуватись символами **(*** коментар ***)**. Тому, символу **'('** призначається катего-рія 5 (позначимо її символами **com**). Інших роздільників в даній граматиці нема, тому вони є недопустимими символами. Недопустимі символи належать до категорії 6 (позначимо її символами **err**). Нагадаємо також, що лексичний аналізатор, крім лексем, що визначаються граматикою мови, повинен ще виділяти і оброблювати пробільні символи (категорія 0, позначимо її символами **ws**).

Закінчення роботи автомата виконується при надходженні си-мвола кіця файла **eof**.

Побудуємо **укрупнений граф лексичного автомата** для даного прикладу (рис.2).

Стани в графі позначаються еліпсами. Початковий стан позна-чається еліпсом з товстою лінією, а кінцевий «подвійним» еліпсом. Стани з'єднуються між собою ребрами з мітками, що означають умову переходу (вхідний символ автомата). Ребра, що не позначені мітками, означають безумовний перехід до стану.

В укрупненому графі автомата ЛА стартовий стан позначений **S**, кінцевий стан – **EXIT**, стан введення наступного символа – **INP** . Інші стани відповідають розпізнаванню визначених вище лексем (токенів) з іменами, що відповідають іменам лексем (**number, iden-tifier, delimiter1, delimiter2**). Крім того, є також додаткові стани (**comment, whitespace**), які відповідають обробці інших категорій символів (**com, ws**), з яких вихідні лексеми не формуються, але які також потрібно оброблювати. Стан **ERR** введений для обробки знайденої помилки при надходженні символа категорії недопусти-мих символів **err**. Вважається, що на виході кожного із цих станів, окрім **ERR**, завжди буде вже новий, ще необроблений, символ.

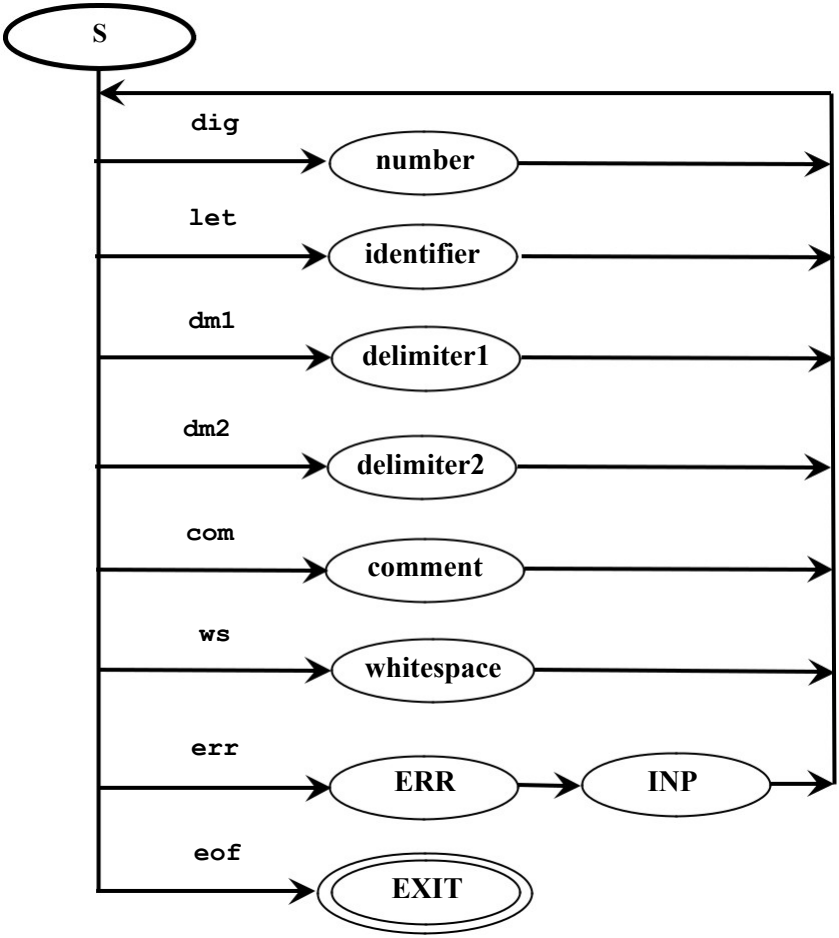

Рис.2 Укрупнений граф автомата ЛА

Значення міток на укрупненому графі автомата лексичного аналізатора (категорій вхідних символів):

← **let** – категорія літер (**a .. z**);

← **dig** – категорія цифр (**0 .. 9**);

← **dm1** – категорія односимвольних роздільників (=);

← **dm2** – категорія багатосимвольних роздільників (<=);

← **com** – категорія початкових символів коментаря (**(**)

← **ws** – категорія пробільних символів;

← **err** – категорія недопустимих символів;

← **eof** – символ кінця файлу.

Значення станів на укрупненому графі автомата лексичного аналізатора:

← **S** – стартовий стан (стан графа автомата в момент початку роботи ЛА);

← **EXIT** – кінцевий стан ЛА;

← **INP** – введення чергового символа;

← **number** – виділення і обробка лексеми (токена) **number**;

← **identifier** – виділення і обробка лексеми (токена) **identifier**;

← **delimiter1** – виділення і обробка лексеми (токена) **delimiter1**;

← **delimiter2** – виділення і обробка лексеми (токена) **delimiter2**;

← **comment** – виділення і обробка коментарів;

← **whitespace** – виділення і обробка пробільних символів;

← **ERR** – видача повідомлення про помилку.

Подальшу розробку ЛА будемо вести методом покрокової деталізації. Деталізуємо роботу ЛА для станів **identifier**, **number**, **delimiter1**, **delimiter2**, **whitespace** та **comment**. Для цього побудуємо часткові підграфи роботи автомата ЛА в цих станах. Необхідно

зауважити, що для невеликих граматик можна одразу побудувати всю діаграму переходів ЛА повністю, без покрокової деталізації.

Значення станів на часткових підграфах виділення та обробки вищезазначених лексем:

← INP – введення чергового символа;

← OUT – обробка та виведення лексеми (токена);

← NUM – обробка чергового символа лексеми (токена) **number** і введення наступного символа;

← IDN – обробка чергового символа лексеми (токена) **identifier** і введення наступного символа;

← DM2 – введення наступного символа після символа '<', який є початковим символом можливого двосимвольного роздільника '<=',

← прийняття рішення про наступний перехід;

← WS – пропуск чергового пробільного символа і введення наступного символа;

← BCOM – введення наступного символа після першого символа початку коментаря (в даному випадку символа '(') і прийняття рішення про наступний перехід;

← COM – введення наступного символа коментаря і прийняття рішення про наступний перехід;

← ECOM – введення наступного символа після символа '*' і прийняття рішення про закінчення чи продовження коментаря;

← ERR – видача повідомлення про помилку.

У всіх станах, окрім OUT та ERR, виконується введення чергового символу. У стані OUT, окрім виведення сформованої лексеми

(код, рядок, колонка) у файл (послідовність лексем), виконуються також всі необхідні дії з відповідними таблицями ідентифікаторів, констант, тощо, а вже введений, але ще не проаналізований, наступ-ний символ передається далі на обробку.

На ребрах часткових підграфів автомата ЛА використовується також мітка ‘**other**’, яка означає, що перехід виконується при надходженні будь-якого іншого символа, крім тих, категорії яких є на інших ребрах, що виходять з цього ж стану.

На рис.3 показаний підграф виділення лексеми (токена) **number**.

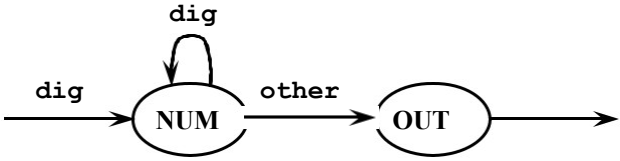

Рис.3 Підграф виділення лексеми (токена) **number**

На рис.4 показаний підграф виділення лексеми (токену) **identifier.**

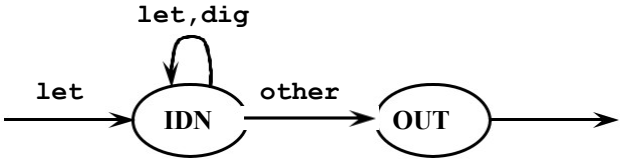

Рис.4 Підграф виділення лексеми (токену) **identifier**

Одним із способів відділення ключових слів від ідентифікаторів є розміщення ключових слів у окремій таблиці при ініціалізації ЛА. Для нашого прикладу граматики ця таблиця має бути

проініціалізована ключовими словами **if**, **then**, **else**. Але оскільки ключові слова відповідають правилам запису ідентифікаторів, то при початковому виділенні ЛА не може відрізнити їх від ідентифі-каторів. Саме тому обробка лексем (токенів) **if, then, else** буде відбу-ватись у стані **identifier**. Після виділення лексеми, що може бути або ключовим словом або ідентифікатором, спочатку виконується по-шук цієї лексеми у таблиці ключових слів. Якщо ця лексема буде знайдена у цій таблиці, значить було виділене ключлве слово, інакше
– звичайний ідентифікатор.

На рис.5 показаний підграф виділення лексеми (токена) **delimiter1**:

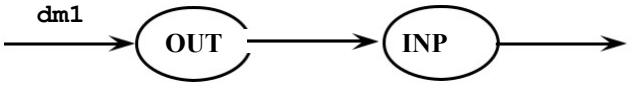

Рис.5 Підграф виділення лексеми (токена) **delimiter1**

На рис.6 показаний підграф виділення лексеми (токена) **delimiter2**:

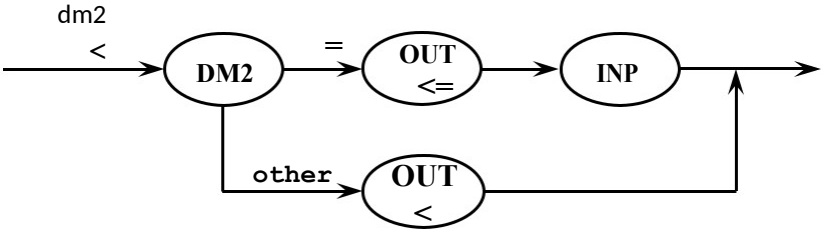

Рис.6 Підграф виділення лексеми (токена) **delimiter2**

На рис.7 показаний підграф виділення та обробки пробільних символів **whitespace**.

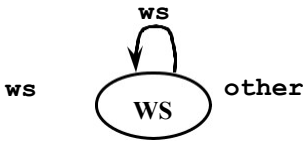

Рис.7 Підграф виділення та обробки **whitespace**

Однією із задач ЛА є видалення коментарів. Хоча коментар не можна назвати лексемою (токеном), його обробку все ж необхідно описати у вигляді окремого підграфа. На рис. 8 показаний підграф обробки коментарів, заданих у вигляді **(***<текст коментаря>***)**.

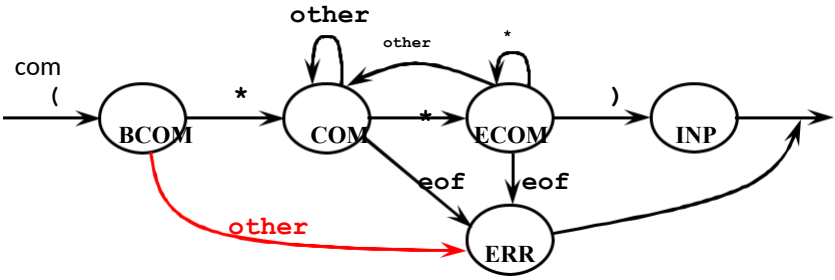

Рис.8 Підграф виділення та обробки коментаря

Важливе зауваження: ***в даному підграфі символ "other" у стані BCOM призводить до помилки (червоне ребро). Однак це вірно лише для граматики даного прикладу, у якій символ "(" використовується тільки як символ початку коментаря і не є односимвольним роздільником. Як виконується обробка цього ребра підграфа у загальному випадку описано в матеріалах лекцій.***

Повний граф автомата ЛА для граматики умовного оператора (рис.1), показаний на рис.9.

Після побудови повного графа автомата можна приступити до реалізації програми ЛА будь-якою мовою програмування так, як це описано в матеріалах лекцій на псевдокоді. Варто зауважити, що при такому підході розмір програми буде пропорційним кількості виділених станів. Кожен стан дає частину коду, що описує поведінку ЛА.

Контрольні питання

← Визначення транслятора, компілятора, інтерпретатора, асемблера.

← Визачення формальної граматики та її алфавіту. Приклад.

← Визначення сентенції граматики, сентерціальної форми граматики та мови граматики.

← Класифікація мов за Хомським.

← Поняття еквівалентних граматик, однозначних граматик та неоднозначних граматик.

← Нормальна форма Грейбах і нормально форма Хомського.

← Структурна схема та основні функції лексичного аналізатора (сканера)..

← Поняття лексеми (токена). Види лексем.

← Типові стани графу автомата лексичного аналізатора (сканера).

← Спосіб написання програми за графом автомату лексичного аналізатора (сканера).

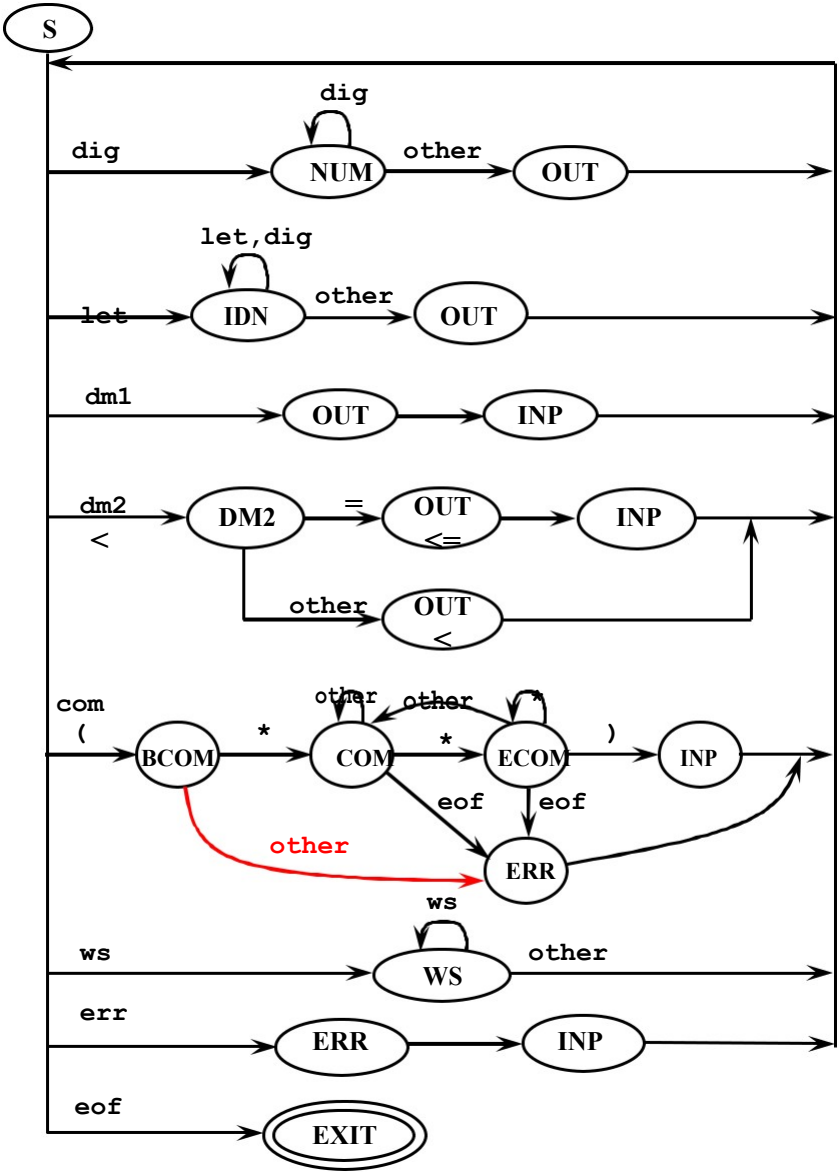

Рис.9. Граф автомата ЛА для граматики умовного оператора

Варіанти індивідуальних завдань

Варіанти граматик складені на основі граматики мови SIGNAL [7] і наведені в додатку 1. В додатку 2 наведена повна граматика мови SIGNAL для загального уявлення про мову та взаємозв'язки її конструкцій.

Варіант визначається відповідно порядковому номеру сту-дента в списку журнала групи.

РОЗРАХУНКОВО-ГРАФІЧНА РОБОТА «РОЗРОБКА СИНТАКСИЧНОГО АНАЛІЗАТОРА»

Мета розрахунково-графічної роботи

Метою розрахунково-графічної роботи «Розробка синтаксисного аналізатора» є засвоєння теоретичного матеріалу та набуття практичного досвіду і практичних навичок розробки синтаксичних аналізаторів (парсерів).

Постановка задачі

← Розробити програму синтаксичного аналізатора (СА) для підмножини мови програмування SIGNAL згідно граматики за варіантом.

← Програма має забезпечувати наступне:

← читання рядка лексем та таблиць, згенерованих лексичним аналізатором, який було розроблено в лабораторній роботі «Розробка лексичного аналізатора»;

← синтаксичний аналіз (розбір) програми, поданої рядком лексем (алгоритм синтаксичного аналізатора вибирається за варіантом);

← побудову дерева розбору;

← формування таблиць ідентифікаторів та різних констант з повною інформацією, необхідною для генерування коду;

← формування лістингу вхідної програми з повідомленнями про лексичні та синтаксичні помилки.

Вимоги до програми синтаксичного аналізатора

← **Заборонено використовувати будь-які бібліотечні засоби роботи з регулярними виразами**;

← Входом синтаксичного аналізатора має бути наступне:

← закодований рядок лексем;

← таблиці ідентифікаторів, числових, символьних та рядкових констант (якщо це передбачено граматикою варіанту), згенеровані лексичним аналізатором;

← вхідна програма на підмножині мови програмування SIGNAL згідно з варіантом (необхідна для формування лістнигу програми).

← Виходом синтаксичного аналізатора має бути наступне:

← дерево розбору вхідної програми;

← таблиці ідентифікаторів та різних констант з повною інформацією, необхідною для генерування коду;

← лістинг вхідної програми з повідомленнями про лексичні та синтаксичні помилки.

← Дерево розбору повинно містити в собі

- ← всі без винятку нетермінали, по яких розібрався вхідний код;
- ← тобто, якщо у вас відсутній який-небуть список (тобто, за граматикою може бути <empty>), то необхідно показати в дереві весь шлях до <empty> включно.

← Правила граматики, які містять в собі альтернативу <empty> , по факту, дозволяють використовувати порожні конструкції.

Вимоги до тестів

- Див. розділ «Загальні вимоги до тестів та тестування».
- Тести до синтаксичного аналізатора повинні покривати всі окремі ситуації згідно граматики варіанту.
- Повинен бути комплексний тест, що містить в собі всі можливі конструкції граматики одночасно (максимальний тест).

Зміст звіту

Звіт оформлюється згідно вимог до розрахунково-графічних робіт і має містити наступне:

- титульний аркуш розрахунково-графічної роботи;
- індивідуальне завдання згідно до варіанту:
  - постановка задачі;
  - номер варіанту;
  - граматика за варіантом в гарно читабельному вигляді
- перетворену граматику (якщо це необхідно для реалізації заданого алгоритму синтаксичного аналізатора);
- таблиця переходів машини Кнута відповідно до граматики, якщо за варіантом заданий метод синтаксичного аналізу з використанням аналізуючої машини Кнута;
- лістинг коду програми СА:
  - звіт повинен містити **весь** код програми;
  - для коду використовувати **моноширинний** шрифт;
  - якщо код не широкий, використовуйте форматування у дві колонки (https://goo.gl/wy9ub6);

← контрольні приклади, необхідні для демонстрації всіх конструкцій заданої граматики, а також всіх можливих помилкових ситуацій;

← опис кожного контрольного прикладу має містити вхідний рядок лексем (результат роботи ЛА, розробленого в лабораторній роботі «Розробка лексичного аналізатора»), згенеровані таблиці, а також зображення побудованого дерева розбору;

← ***в якості графічної частини розрахунково-графічної роботи*** включити до звіту рисуки дерев розбору, побудованих для всіх кон-трольних прикладів (мінімум 5 прикладів і дерев).

До звіту в електронному вигляді мають бути додані:

← файл з текстом звіту;

← проект працюючої програми на мові програмування;

← завантажувальний модуль програми з необхідними файлами даних.

Методичні вказівки

Загальна схема синтаксичного аналізу на основі недетермінованого МП-автомата працює з поверненнями. Однак, якщо на граматику, яка породжує мову, накласти певні обмеження, а також ефе-ктивно використовувати стек автомата, то можна використати певні відомі прийоми, які дозволяють будувати ефективні синтаксичні аналізатори, що працюють без повернень.

На практиці, в СА існуючих мов програмування використовується саме такий підхід. Найефективніші методи синтаксичного

аналізу працюють тільки з підкласами граматик. Найбільш швидкими є алгоритми синтаксичного аналізу для LL(1)-граматик, які працюють за низхідною стратегією розбору.

Зауваження: низхідні методи розбору не можуть працювати з граматиками, що мають ліворекурсивні правила. Тому при використанні таких методів необхідно попередньо усунути ліву рекурсію з граматики.

У завданнях розрахунково-графічної роботи використовуються два методи синтаксичного аналізу: метод рекурсивного спуску та метод на основі аналітичної машини Кнута.

Ці методи детально пояснюються в конспекті лекцій та у відео-лекції https://youtu.be/oLDmeX-X5rI

Контрольні питання

← Принцип виконання низсхідного аналізу (розбору).

← Принцип виконання висхідного аналізу (розбору).

← Визначення автомату з магазинною (стековою) пам'яттю (МП-автомату).

← Конфігурація і такт роботи МП-автомата.

← Детерміновані та не детерміновані МП-автомати.

← Відповідність між КВ-граматикою та МП-автоматом.

← Визначення LL(k), LR(k), RL(k) і RR(k) граматик.

← Основні алгоритми низсхідної стратегії синтаксичного розбору та переваги використання LL(1)-граматик.

← Принцип роботи синтаксичного аналізу за алгоритмом аналізуючої машини Кнута (АМК).

10. Формування таблиці АМК (програмування АМК).

11. Принцип роботи синтаксичного аналізу за алгоритмом рекур-сивного спуску.

← Принцип роботи синтаксичного аналізу за алгоритмом граматик передування.

13. Принцип роботи синтаксичного аналізу за алгоритмом табли-чно-керованого передбачаючого (прогнозуючого) розбіру.

Варіанти індивідуальних завдань

Варіанти граматик складені на основі граматики мови SIGNAL [7] і наведені в додатку 1. В додатку 2 наведена повна граматика мови SIGNAL для загального уявлення про мову та взаємозв'язки її конструкцій.

Алгоритм синтаксичного аналізу вибирається за таблицею 2.

Числа в таблиці означають номер алгоритму синтаксичного аналізу для даного варіанту згідно з наступним списком:

1 – низхідний розбір за алгоритмом аналізуючої машини Кнута;

2 – низхідний розбір за алгоритмом рекурсивного спуску.

Таблиця 2. Варіанти алгоритму синтаксичного аналізу

Номер варіанту	**Номер алгоритму синтаксичного аналізу**	**Номер варіанту**	**Номер алгоритму синтаксичного аналізу**
1	1	19	1
2	2	20	2
3	1	21	1
4	2	22	2
5	1	23	1
6	2	24	2
7	1	25	1
8	2	26	2
9	1	27	1
10	2	28	2
11	1	29	1
12	2	30	2
13	1	31	1
14	2	32	2
15	1	33	1
16	2	34	2
17	1	35	1
18	2		

Варіант визначається відповідно порядковому номеру сту-дента в списку журнала групи.

ЛАБОРАТОРНА РОБОТА №2
«РОЗРОБКА ГЕНЕРАТОРА КОДУ»

Мета лабораторної роботи

Метою лабораторної роботи «Розробка генератора коду» є за-своєння теоретичного матеріалу та набуття практичного досвіду і практичних навичок розробки генераторів коду.

Постановка задачі

← Разробити програму генератора коду (ГК) для подмножини мови програмування SIGNAL, заданої за варіантом.

← Програма генератора коду має забезпечувати:

← читання дерева розбору та таблиць, створених синтаксичним аналізатором, що було розроблено в розрахунково-графічній роботі;

← виявлення семантичних помилок;

← генерацію коду та/або побудову внутрішніх таблиць для генерації коду.

← Зкомпонувати повний компілятор, що складається з розроблених раніше лексичного та синтаксичного аналізаторів і генератора коду, який забезпечує наступне:

← генерацію коду та/або побудову внутрішніх таблиць для генерації коду;

← формування лістингу вхідної програми з повідомленнями про лексичні, синтаксичні та семантичні помилки.

Вимоги до програми генератора коду

← Входом генератора коду (ГК) мають бути:

← дерево розбору;

← таблиці ідентифікаторів та констант з повною інформацією, необхідною для генерації коду;

← вхідна програма на підмножині мови програмування SIGNAL згідно з варіантом (необхідна для формування лістингу програми).

← Виходом ГК мають бути:

← асемблерний код згенерований для вхідної програми та/або внутрішні таблиці для генерації коду;

← внутрішні таблиці генератора коду (якщо потрібні).

← лістинг вхідної програми з повідомленнями про лексичні, синтаксичні та семантичні помилки.

← Генератор коду виявляє всі семантичні помилки помилки відразу, а не зупиняється після першої виявленої семантичної помилки, оскільки семантичні помилки, на відміну від синтаксичних, є неза-лежними.

← З іншої сторони, влане генерація коду втрачає сенс вже навіть при наявності тільки однієї семантичної помилки і витрачати час на формування вихідного асемблерного коду після виявленої першої семантичної помилки не потрібно, оскільки швидкість роботи транслятора є однією з найважливіших його характеристик.

← **Заборонено використовувати будь-які бібліотечні засоби роботи з регулярними виразами**.

Вимоги до тестів

- Див. розділ «Загальні вимоги до тестів та тестування».
- Тести до генератора коду повинні покривати всі окремі ситуації семантичних помилок згідно семантики конструкцій граматики варіанту.
- Повинен бути комплексний тест, що містить в собі всі можливі конструкції граматики одночасно (максимальний тест).

Зміст звіту

Звіт оформлюється згідно до вимог, що висуваються до лабораторних робіт і має містити наступне:

- титульний аркуш розрахунково-графічної роботи;
- індивідуальне завдання згідно до варіанту:
  - постановка задачі;
  - номер варіанту;
  - граматика за варіантом в гарно читабельному вигляді
- лістинг коду програми ГК:
  - звіт повинен містити **весь** код програми;
  - для коду використовувати **моноширинний** шрифт;
  - якщо код не широкий, використовуйте форматування у дві колонки (https://goo.gl/wy9ub6);
- контрольні приклади, що демонструють коректність генерування коду для всіх конструкцій заданої граматики, а також всіх мо-жливих помилкових ситуацій;

← контрольні приклади, що демонструють коректність виведення повідомлень про помилки для всіх можливих семантичних по-милкових ситуацій;

← опис кожного контрольного прикладу має містити вхідну програму мовою SIGNAL та відповідний їй код асемблерної програми (для коректних SIGNAL-програм) або вхідну програму мовою SIGNAL та відповідні їй повідомлення про наявні семантичні помилки;

← контрольні приклади, необхідні для демонстрації всіх конструкцій заданої граматики, а також всіх можливих помилкових ситуацій. Опис кожного контрольного прикладу має містити вхідне де-рево розбору і таблиці сформовані синтаксичним аналізатором, а та-кож згенерований код та додаткові внутрішні таблиці ГК (якщо не-обхідно).

До звіту в електронному вигляді має бути додано наступне:

← файл з текстом звіту;

← проект працюючої програми (компілятора, що включає ЛА, СА та ГК) з вхідними кодами, завантажувальний модуль та необхід-ними файлами даних;

← набір тестів, що показують коректність роботи компілятора, та відповідні лістинги з результатами компіляції.

Методичні вказівки

Неформальна семантика конструкцій мови програмування SIGNAL.

← **Визначення головної програми (PROGRAM) та процедур (PROCEDURE) без параметрів.**

Дії:

← генерація заголовку для виклику програми чи процедури;

← генерація інструкцій повернення з підпрограми.

Обмеження:

← не дозволяється використовувати однакові імена для двох і більше процедур;

← не дозволяється використовувати однакові імена для процедури та довільної змінної або константи.

← **Визначення процедур (PROCEDURE) з параметрами.**

Дії:

← генерація заголовку для виклику програми чи процедури;

← генерація інструкцій повернення з підпрограми;

← виділення кадру стеку для зберігання параметрів.

Обмеження:

← не дозволяється використовувати однакові імена для двох і більше процедур;

← не дозволяється використовувати однакові імена для процедури та довільної змінної або константи;

← має бути відповідність за типом і порядком розташування між формальними параметрами, заданими в списку параметрів, та фактичними параметрами, заданими у операторі виклику;

← повторне використання того самого атрибута – це не помилка,

← попередження, і код повинен генеруватися.

← атрибут EXT у параметрів бути не може, треба видавати помилку;

← порядок будь-яких атрибутів може бути довільним, важлива тільки їх смислова сумісність (див вказавки до типів даних).

Пояснення:

Обробка виклику процедури/функції складається з двох час-тин. Перша частина цього процесу відбувається при обробці опера-тора виклику, при якому фактичні параметри (аргументи) заштовху-ються у стек. А друга частина – це доступ до цих параметрів, що лежать у стеку із коду процедури/функції. Як відомо, передавання параметрів може бути за значенням (значення копіюються, а оригі-нал не змінюється) і за адресою (значення за переданою адресою може бути змінене у процедурі). Взагалі, може бути два варіанти об-робки параметрів в згенерованому коді процедури/функції:

← при доступі до параметра генерується звернення прямо до потрібної комірки стеку за зміщенням в усіх точках використання параметрів у процедурі;

← спочатку параметри зі стеку дістаються (копіюються) у локальну пам'ять процедури/функції, а потім в коді процедури/функції вже використовуються саме ці комірки.

Є певні переваги/недоліки у кожного з підходів. Але в спрощених граматиках варіантів завдань при першому підході незрозуміло що генерувати, оскільки конкретна граматика може не вклю-чати оператора виклику процедури/функції, або вона може бути без параметрів. Тому, щоб показати обробку параметрів при обробці тексту процедури/функції, потрібно на самому початку її коду згенерувати обернені команди POP до тих PUSH, які могли б бути згенеровані при виклику, незалежно, чи містить задана граматика опера-тор виклику, чи ні. Тобто, вважаємо, що при обробці оператора ви-клику всі фактичні параметри вже були заштовхнуті у стек і вони зараз лежать на вершині стека. Відповідно при обробці конструкції оголошення процедури/функції потрібно згенерувати потрібну кількість POP для параметрів у оберненому порядку і зберегти їх у якихось тимчасових локальних комірках пам'яті.

Наприклад, для

PROCEDURE (A:FLOAT; B:INTEGER);

повинні бути виділені комірки для A і B відповідно до типів FLOAT і INTEGER, а також повинно бути згенеровані команди забирання з вершини стеку значень у комірки A і B.

Якщо у граматиці варіанту параметри не мають типу, то будемо вважати, що для всіх їх треба виділяти по 4 байти.

← **Визначення математичних функцій (DEFFUNC)**

Семантика:

← визначення математичної функції задає ініціалізований масив, заданий за допомогою формули в заданих межах.

Дії:

← виділення пам'яті для масиву заданого розміру;

← ініціалізація масиву значеннями згідно до заданої формули.

Обмеження:

← не дозволяється використовувати однакові імена для двох і більше функцій;

← імена функцій, описаних в розділі DEFFUNC фактично є іменами масивів, а тому не мають збігатися з іменами інших змінних та констант.

Пояснення:

← повній граматиці мови SIGNAL та варіантах з ускладненими граматиками основне правило опису однієї функції має вигляд

<function> --> <function-identifier> = <expression><function-characteristic> ;

Де передбачається, що у виразі <expression> може використовуватися неоголошений ідентифікатор x, як і у математичних функціях з аргументом x, типу f(x) = 3x2+x-5, де 3x2+x-5 і є <expression>, тобто вираз, що визначає функцію.

Математичні функції, що оголошуються у розділі DEFFUNC, представляються у пам'яті як масиви, які ініціалізуються значеннями виразу <expression> на етапі компіляції (генерації коду),

причому кількість та діапазон значень цієї функції визначаються її характеристикою

<function-characteristic> --> \ <unsigned-integer> , <unsigned-integer>

У варіантах зі спрощеними граматиками на місці <expression> стоїть <constant>, що означає, що всі значення цієї функції будуть однаковими і дорівнювати значенню <constant>.

Перше значення <unsigned-integer> із характеристики функції <function-characteristic> є верхньою границею діапазону зміни значень аргумента x. Нижнею границею завжди є 0. Але цей момент є важливим тільки при повному визначенні DEFFUNC коли використовується <expression>. У варіантах з <constant> замість <expression> цей перший <unsigned-integer> на генерацію коду не впливає, оскільки всі значення функції будуть однаковими незалежно від аргумента x.

Друге значення <unsigned-integer> із характеристики функції є кількістю точок, на які треба розбити інтервал від 0 до першого <unsigned-integer>. Тобто, друге значення <unsigned-integer> є кількістю елементів (розміром) ініціалізованого масива, що визначається функцією розділу DEFFUNC.

Тому, все, що треба зробити у спрощених варіантах при генерації коду для цієї конструкції, це виділити пам’ять для масива з іме-нем <function-identifier> з кількістю елементів визначених другим значенням <unsigned-integer> із характеристики функції і

проініціалізувати всі елементи цього масива значеннями що дорівнюють <constant>.

← **Типи даних**

Базові типи:

← INTEGER – ціле число;

← FLOAT – число з плаваючою комою. В пам'яті таке число міститься у вигляді <мантиса><порядок>. Достатньо взяти дві окремі комірки для мантиси та порядку, наприклад по 4 байти, і покласти туди задані значення, якщо це константа. Пам'ять для масиву такого типу виділяється наступним чином:

<порядок1>

<мантиса1>

<порядок2>

<мантиса2>

…

<порядокN>

← ***мантисаN>***

← BLOCKFLOAT – використовується тільки для масивів чисел з плаваючою комою, що мають однаковий порядок. Пам'ять для масиву такого типу виділяється наступним чином:

<порядок, спільний для всіх мантис>

<мантиса1>

…

← ***мантисаN>***

← COMPLEX – визначає комплексний тип даних для одного з базових типів. Якщо для простої змінної чи константи цілого типу INTEGER виділяється, наприклад, 4 байти, то для змінної чи константи комплексного цілого типу COMPLEX INTEGER виділяється 8 байтів, з яких перших 4 трактується як дійсна частина комплексного числа, а других 4 байти як уявна частина комплексного цисла. Ана-логічно, для інших комплексних типів.

Масиви:

• для оголошення масиву необхідно описати атрибут [<range><ranges-list>], який задає розмірності масиву (кількість роз-мірностей та їхні індексні діапазони); якщо допускаються тільки од-номірні масиви, то в граматиці вказаний тільки один діапазон [<range>]. На відміну від мови C/C++, де всі масиви індексуються від нуля, мова SIGNAL допускає оголошення масивів з довільними індексними діапазонами значень. Головне, щоб значення нижньої границі діапазону індексів не перевищувало верхню границю, ін-акше помилка;

← якщо граматика допускає використання значення одного масива в якості індекса іншого масива, наприклад,

A[B[C[K]]]

то ідентифікатори A, B, C повинні бути оголошені масивами, а гене-рація коду виконується наступним чином: спочатку береться зна-чення комірки K в якості індекса масива C, потім треба дістати у як-ийсь регістр значення K-го ел-та масива C і цей регістр використати в якості індекса масива B, і т.д.

Обмеження на використання атрибутів типів:

← синтаксис (граматика) оголошення змінних не обмежує кількості атрибутів в одному оголошенні, але перевірка коректності записаного при оголошенні списку атрибутів та сумісності цих атрибутів між собою якраз і є однією із головних задач семантичного аналізу;

← довільна змінна чи константа може мати тільки один з базових специфікаторів INTEGER, FLOAT або BLOCKFLOAT, інакше помилка;

← атрибут комплексного типу даних COMPLEX має сенс тільки ← парі з одним із базових типів INTEGER, FLOAT або

← доповнюючий атрибут SIGNAL можна використовувати разом з одним з числових типів;

← атрибут SIGNAL вказує на те, що змінна є вхідною чи вихідною. Така змінна обов'язково має бути пов'язана з регістром (портом) вводу-виводу за допомогою оператору LINK, після чого дії вводу-виводу виконуються операторами IN або OUT;

← всі змінні з однаковим ідентифікатором, оголошені в різних процедурах з доповнюючим атрибутом EXT, зв'язуються з однією й тією ж коміркою пам'яті. Специфікатор EXT можна використову-

вати з будь-яким з базових або комплексним типом, а також з масивом;

← Якщо в одному оголошенні якийсь атрибут вказаний більше одного разу, то видається попередження.

← Оголошення змінних

Дії:

генерація коду виділення пам'яті для змінних.

Обмеження:

не дозволяється використовувати однакові імена для двох і більше змінних в одній підпрограмі чи на глобальному рівні.

Оголошення констант

Дії:

генерація коду виділення пам'яті для констант згідно їх типу;

ініціалізація виділеної пам'яті заданими значеннями констант.

Обмеження:

не дозволяється використовувати однакові імена для двох і більше констант в одній підпрограмі чи на глобальному рівні;

не дозволяється використовувати однакові імена для константи та довільної змінної або процедури чи програми;

0 не дозволяється змінювати значення константи в програмі.

Пояснення:

дійсні числа типу FLOAT у показниковій формі, наприклад, -123e-5 в мові SIGNAL записуються так само, тільки замість бу-кви 'e' використовується символ '#', тобто буде -123#-5;

якщо комплексна константа записана у коді в експоненційній формі **<вираз>$EXP(<вираз>)**, де вирази обов'язково повинні бути константними (тобто їх можна обчислити к трансляторі), то спочатку потрібно перевести прямо у трансляторі це число у

звичайну арифметичну форму і записати у виділену пам'ять отримані значення дійсної та уявної частин комплексного числа. Тобто, в результаті, надалі на асемблері робота над компле-ксними числами буде відбуватися тільки у арифметичній формі. Якщо хтось не пам’ятає як перевести комплексне число із екс-поненційної (показникової) форми у арифметичну, зверніться до довідників та інтернету.

Оголошення міток

Дії:

генерація таблиці міток;

перетворення міток до вигляду ідентифікаторів для можливості вставки в асемблерний код.

Обмеження:

не дозволяється використовувати однакові імена для двох і більше міток;

імена згенерованих міток не мають збігатися з іменами змінних та констант в одній підпрограмі та з зовнішніми іменами на гло-бальному рівні.

Використання змінних

Дії:

конструкція виду <змінна>[<вираз>,<список-виразів>] задає звернення до елемента масиву;

числова константа в одинарних лапках задає комплексну константу, яка повинна обчислюватись на етапі згортки у лексичному аналізаторі;

конструкція виду “<вираз>,<вираз>” задає комплексну змінну (тобто в виразі можуть бути присутні змінні);

конструкція виду “<вираз>$EXP(<вираз>)” задає комплексну змінну в експоненційній формі.

Обмеження:

не дозволяється використання комплексних змінних в лівій частині виразу (ліворуч від знаку присвоєння), заданих у вигляді “<вираз>,<вираз>” або “<вираз>$EXP(<вираз>)”, тобто у лівій частині оператора присвоєння дозволяється використовувати тільки одиничні ідентифікатори комплексних змінних.

Керуючі конструкції (Оператори)

Дії:

правило <statements-list> --> <empty> відповідає інструкції nop;

конструкція <variable> := <expression> задає оператор присвоєння;

конструкція <procedure-identifier><actual-arguments> задає оператор виклику процедури;

конструкція IF-THEN-ELSE-ENDIF задає умовний оператор (існує варіант як з однією, так і з двома гілками умовного оператора);

конструкція WHILE-DO задає цикл з передумовою;

конструкція LOOP-ENDLOOP задає нескінчений цикл;

конструкція FOR-ENDFOR задає цикл з лічильником;

конструкція CASE-OF-ENDCASE задає оператор вибору;

конструкція GOTO задає оператор безумовного переходу до заданої мітки;

конструкція RETURN задає оператор повернення з процедури;

пустому оператору „ ; ” відповідає асемблерна інструкція **nop**.

Обмеження:

мітка, на яку передає керування оператор безумовного переходу GOTO повинна бути обов’язково: 1) оголошена у розділі LABEL; 2) використана перед одним з операторів програми, інакше помилка;

Якщо у варіанті граматики розділ LABEL відсутній, то всі мітки вважаються оголошеними і залишається тільки ситуація 2) попереднього пункту;

використання однакової мітки перед операторами у коді програми більше обного разу є помилкою;

якщо при оголошенні міток якась мітка у списку міток зустрічається більше одного разу, видається попередження.

10. **Конструкції вводу-виводу**

До цієї групи конструкцій належать оператори LINK, IN, OUT. Для цих операторів в рамках лабораторної роботи код генерувати не потрібно. Треба буде виконати тільки семантичний аналіз.

Дії:

конструкція LINK <variable-identifier> , <unsigned-integer> зв'язує задану змінну з номером (<unsigned-integer>) одного з регіс-трів (портів) вводу-виводу;

конструкції IN <unsigned-integer> и OUT <unsigned-integer> задають властивості *«тільки для вводу»* чи *«тільки для виводу»* регі-стру (порту) вводу-виводу з номером <unsigned-integer>, який ра-ніше повинен бути зв'язаний оператором LINK з деякою змінною, що оголошена в програмі і має атрибут SIGNAL (цей атрибут озна-чає, що ця змінна використовується в програмі для вводу чи виводу інформації). Ці властивості задаються віртуально і є необхідними на етапі компіляції тільки для перевірки семантичної коректності. Опе-ратор IN задає регістру (порту) вводу-виводу властивість *«тільки для вводу»*, а оператор OUT – властивість *«тільки для виводу»*.

Обмеження:

в операторах LINK можуть використовуватись тільки змінні, оголошені з атрибутом SIGNAL, тобто якщо серед атрибутів в описі ідентифікатора змінної, що вказаний у операторі LINK, немає атри-бута SIGNAL, то це помилка;

0 якщо в операторі IN або OUT вказаний номер порту, який попередньо не був зв'язаний раніше з ід-ром оператором LINK, то це помилка;

1 один і той же номер регістра (порту) вводу-виводу не може використовутися в одній програмі одночасно як в операторі IN, так в операторі OUT, оскільки вони визначають протилежні властивості вводу-виводу, тобто:

не дозволяється запис до регістру, якщо для нього була вста-новлена властивість „тільки читання” за допомогою оператору IN;

не дозволяється читання з регістру, якщо для нього була встановлена властивість „тільки запис” за допомогою оператору OUT.

Наприклад, приймаєте для апаратної платформи, для якої Ви генеруєте код, що на цій платформі є, наприклад, три порти (регіс-три) вводу-виводу тільки для вводу (нехай з номерами 1, 2, 3) і три порти тільки для виводу (нехай з номерами 4, 5, 6). LINK зв'язує порт конкретним номером з раніше оголошеною змінною A, яка обов’язково має серед атрибітів атрибут SIGNAL. Якщо певний порт встановлений тільки для вводу, то цей номер може бути в коді програми тільки в операторі IN, якщо це порт для виводу, то тільки в операторі OUT. Інакше буде семантична помилка.

11. **Операції в виразах**

Дії:

результатом логічних операцій (OR, AND, NOT) та операцій порівняння є 0 чи 1;

операція ! відповідає побітовому OR над операндами типу INTEGER;

операція & відповідає побітовому AND над операндами типу INTEGER;

операція ^ відповідає побітовому NOT над операндом типу INTEGER;

результатом операції MOD є залишок від ділення.

якщо в граматиці допускаються тільки логічні (булеві) вирази, то, наприклад, результатом присвоєння

A:=0>1;

буде запис значення 0 (false) у змінну A і змінна A буде вважатися змінною логічного (булевого) типу.

Обмеження:

операції мають виконуватись тільки над даними сумісних типів.

12. **Асемблерні вставки**

Дії:

Конструкція

($** <assembly-insert-file-identifier> **$)

відповідає вставці тексту асемблерного коду, який знаходиться в заданому файлі з іменем <assembly-insert-file-identifier>, в програму без будь-яких перевірок.

<assembly-insert-file> – це ідентифікатор реального файлу, що знаходиться на диску і містить у собі код асемблерної вставки, тобто папці тесту повинен бути файл з іменем, заданим <assembly-insert-file-identifier>, який містить якийсь асемблерний код, і цей код з файлу повинен бути вставлений без змін "as is" у згенерований код на місці знаходження конструкції ($ <assembly-insert-file-identifier>
$) в коді на мові SIGNAL замість неї.

Наприклад, якщо у файлі filename.txt знаходяться команди

```
MOV AX,VAR1
MOV VAR2,AX
```

то в точці знаходження конструкції ($filename.txt$) замість неї повинен бути згенерований код

```
MOV AX,VAR1
MOV VAR2,AX
```

Контрольні питання

Поняття неформальної та формальної семантики мови програмування.

Визначення мета семантичної мови та приклади існуючих мета семантичних мов.

Види семантичної відповідності у простій мета семантичній мові.

Структура програми генератора коду (семантичного процесора) та структура рекурсивних семантичних процедур/функцій.

Генерація коду для оператора присвоєння.

Генерація коду для неповної умовної конструкції if-then.

Генерація коду для повної умовної конструкції if-then-else.

Генерація коду для конструкції циклу з передумовою while.

Генерація коду для конструкції циклу з післяумовою repeat.

10. Генерація коду для конструкції циклу з лічильником (параме-тром) for.

11. Генерація коду для керуючої конструкції вибору case.

Варіанти індивідуальних завдань

Варіанти граматик складені на основі граматики мови SIGNAL [7] і наведені в додатку 1. В додатку 2 наведена повна граматика мови SIGNAL для загального уявлення про мову та взаємозв'язки її конструкцій.

Варіант визначається відповідно порядковому номеру сту-дента в списку журнала групи.

ДОДАТОК 1. ВАРІАНТИ ГРАМАТИК ДЛЯ ІНДИВІДУАЛЬНИХ ЗАВДАНЬ.

Варіанти граматик складені на основі граматики мови SIGNAL [7]. В додатку 2 наведена повна граматика мови SIGNAL для загального уявлення про мову та взаємозв'язки її конструкцій.

Варіант визначається відповідно порядковому номеру сту-дента в списку журнала групи.

Варіант 1

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>. |
                PROCEDURE <procedure-identifier>
                <parameters-list> ; <block> ;
<block> --> <declarations> BEGIN <statements-list>
                END
<declarations> --> <label-declarations>
<label-declarations> --> LABEL <unsigned-integer>
                <labels-list>; |
                <empty>
<labels-list> --> , <unsigned-integer> <labels-list>
                |
                <empty>
<parameters-list> --> ( <declarations-list> ) |
                <empty>
<declarations-list> --> <empty>
<statements-list> --> <empty>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 2

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>. |
                PROCEDURE <procedure-
                identifier><parameters-list> ; <block> ;
<block> --> BEGIN <statements-list> END
<statements-list> --> <empty>
<parameters-list> --> ( <declarations-list> ) |
                <empty>
<declarations-list> --> <declaration> <declarations-list>
                |
                <empty>
<declaration> --><variable-identifier><identifiers-
                list>:<attribute><attributes-list> ;
<identifiers-list> --> , <variable-identifier>
                <identifiers-list> |
                <empty>
<attributes-list> --> <attribute> <attributes-list>
                |
                <empty>
<attribute> --> SIGNAL |




<procedure-identifier> --> <identifier>
<variable-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 3

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>.
<block> --> <declarations> BEGIN <statements-list>
                END
<declarations> --> <constant-declarations>
<constant-declarations> --> CONST <constant-
                declarations-list> |
                <empty>
<constant-declarations-list> --> <constant-declaration>
                <constant-declarations-list> | <empty>
<constant-declaration> --> <constant-identifier> =
                <constant>;
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> --> <variable-identifier> := <constant>
                ;
<constant> --> <unsigned-integer>
<constant> --> - <unsigned-integer>
<constant-identifier> --> <identifier>
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 4

```
<signal-program> --> <program>
<program> --> PROCEDURE <procedure-identifier><parameters-
                list> ; <block> ;
<block> --> <declarations> BEGIN <statements-list>
                END
<statements-list> --> <empty>
<parameters-list> --> ( <declarations-list> ) |
                <empty>
<declarations-list> --> <declaration> <declarations-list>
                |
                <empty>
<declaration> --> <variable-identifier> :
                <attribute> ;
<attribute> --> INTEGER | FLOAT
<declarations> --> <constant-declarations>
<constant-declarations> --> CONST <constant-
                declarations-list> |
                <empty>
<constant-declarations-list> --> <constant-declaration>
                <constant-declarations-list> | <empty>
<constant-declaration> --> <constant-identifier> =
                <constant>;
<constant> --> <unsigned-integer>
<constant> --> - <unsigned-integer>
<constant-identifier> --> <identifier>
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 5

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>.
<block> --> <declarations> BEGIN <statements-list>
                END
<statements-list> --> <empty>
<declarations> --> <constant-declarations>
<constant-declarations> --> CONST <constant-
                declarations-list> |
                <empty>
<constant-declarations-list> --> <constant-declaration>
                <constant-declarations-list> | <empty>
<constant-declaration> --> <constant-identifier> =
                <constant>;
<constant> --> <sign> <unsigned-constant>
<constant-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-constant> --> <integer-part><fractional-part>
<integer-part> --> <unsigned-integer>
<fractional-part> --> #<sign><unsigned-integer> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<sign> --> + |
                0 |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 6

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>.
<block> --> <declarations> BEGIN <statements-list>
                END
<statements-list> --> <empty>
<declarations> --> <constant-declarations>
<constant-declarations> --> CONST <constant-
                declarations-list> |
                <empty>
<constant-declarations-list> --> <constant-declaration>
                <constant-declarations-list> | <empty>
<constant-declaration> --> <constant-identifier> =
                <constant>;
<constant> --> '<complex-number>'
<complex-number> --> <left-part> <right-part>
<left-part> --> <unsigned-integer> |
                <empty>
<right-part> --> ,<unsigned-integer> | $EXP(
                <unsigned-integer> ) |
                <empty>
<constant-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 7

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>.
<block> --> <declarations> BEGIN <statements-list>
                END
<statements-list> --> <empty>
<declarations> --> <variable-declarations>
<variable-declarations> --> VAR <declarations-list>
                |
                <empty>
<declarations-list> --> <declaration> <declarations-list>
                |
                <empty>
<declaration> --><variable-identifier><identifiers-
                ist>:<attribute><attributes-list> ;
<identifiers-list> --> , <variable-identifier>
                <identifiers-list> |
                <empty>
<attributes-list> --> <attribute> <attributes-list>
                |
                <empty>
<attribute> --> SIGNAL |
                COMPLEX |
                INTEGER |
                FLOAT |
                BLOCKFLOAT |
                EXT |
                [<range><ranges-list>]
<ranges-list> --> ,<range> <ranges-list> |
                <empty>
<range> --> <unsigned-integer> .. <unsigned-integer>
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 8

```
<signal-program> --> <program>
<program> --> PROCEDURE <procedure-identifier><parameters-
                list> ; <block> ;
<block> --> <declarations> BEGIN <statements-list>
                END
<statements-list> --> <empty>
<parameters-list> --> ( <declarations-list> ) |
                <empty>
<declarations> --> <variable-declarations>
<variable-declarations> --> VAR <declarations-list>
                |
                <empty>
<declarations-list> --> <declaration> <declarations-list>
                |
                <empty>
<declaration> -->><variable-identifier><identifiers-
                list>:<attribute><attributes-list> ;
<identifiers-list> --> , <variable-identifier>
                <identifiers-list> |
                <empty>
<attributes-list> --> <attribute> <attributes-list>
                |
                <empty>
 12.   <attribute> --> SIGNAL     |
                COMPLEX     |
                INTEGER     |
                FLOAT       |
                BLOCKFLOAT |
                EXT
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 9

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>.
<block> --> <declarations> BEGIN <statements-list>
                END
<statements-list> --> <empty>
<declarations> --> <math-function-declarations>
<math-function-declarations> --> DEFFUNC <function-list>
                |
                <empty>
<function-list> --> <function> <function-list> |
                <empty>
<function> --> <function-identifier> =
                <constant><function-characteristic> ;
<function-characteristic> --> \ <unsigned-integer> ,
                <unsigned-integer>
<constant> --> <unsigned-integer>
<procedure-identifier> --> <identifier>
<function-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 10

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block> ;
<block> --> <declarations> BEGIN <statements-list>
                END
<statements-list> --> <empty>
<declarations> --> <procedure-declarations>
<procedure-declarations> --> <procedure> <procedure-
                declarations> |
                <empty>
<procedure> --> PROCEDURE <procedure-
                identifier><parameters-list> ;
<parameters-list> --> ( <declarations-list> ) |
                <empty>
<declarations-list> --> <declaration> <declarations-list>
                |
                <empty>
<declaration> --><variable-identifier><identifiers-
                list>:<attribute><attributes-list> ;
<identifiers-list> --> , <variable-identifier>
                <identifiers-list> |
                <empty>
<attributes-list> --> <attribute> <attributes-list>
                |
                <empty>
 13.  <attribute> --> SIGNAL     |
                COMPLEX     |
                INTEGER     |
                FLOAT       |
                BLOCKFLOAT |
                EXT
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 11

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>.
<block> --> <declarations> BEGIN <statements-list>
                END
<declarations> --> <label-declarations>
<label-declarations> --> LABEL <unsigned-integer>
                <labels-list>; |
                <empty>
<labels-list> --> , <unsigned-integer> <labels-list> |
                <empty>
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> --> <unsigned-integer> : <statement> | GOTO
                <unsigned-integer> ; |
                LINK <variable-identifier> , <unsigned-
                integer> ; |
                IN <unsigned-integer>; |
                OUT <unsigned-integer>;
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 12

```
<signal-program> --> <program>
<program> --> PROCEDURE <procedure-identifier>
                <parameters-list> ; <block> ;
<block> --> <declarations> BEGIN <statements-list>
                END
<declarations> --> <label-declarations>
<label-declarations> --> LABEL <unsigned-integer>
                <labels-list>; |
                <empty>
<labels-list> --> , <unsigned-integer> <labels-list>
                |
                <empty>
<parameters-list> --> ( <variable-identifier>
                <identifiers-list> ) |
                <empty>
<identifiers-list> --> , <variable-identifier>
                <identifiers-list> |
                <empty>
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> --> <unsigned-integer> : <statement> | GOTO
                <unsigned-integer> ; |
                RETURN ; |
                |
                ($ <assembly-insert-file-identifier> $)
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<assembly-insert-file-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 13

```
<signal-program> --> <program>
<program> --> PROCEDURE <procedure-
                identifier><parameters-list>; <block> ;
<block> --> <declarations> BEGIN <statements-list>
                END
<declarations> --> <procedure-declarations>
<procedure-declarations> --> <procedure> <procedure-
                declarations> |
                <empty>
<procedure> --> PROCEDURE <procedure-
                identifier><parameters-list> ;
<parameters-list> --> ( <variable-identifier>
                <identifiers-list> ) |
                <empty>
<identifiers-list> --> , <variable-identifier>
                <identifiers-list> |
                <empty>
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> --> <procedure-identifier><actual-
                arguments> ; |
                RETURN ;
<actual-arguments> --> ( <unsigned-integer> <actual-
                arguments-list> ) |
                <empty>
<actual-arguments-list> --> , <unsigned-integer>
                <actual-arguments-list> |
                <empty>
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 14

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>.
<block> --> <declarations> BEGIN <statements-list>
                END
<declarations> --> <label-declarations>
<label-declarations> --> LABEL <unsigned-integer>
                <labels-list>; |
                <empty>
<labels-list> --> , <unsigned-integer> <labels-list> |
                <empty>
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> --> <unsigned-integer> : <statement> | GOTO
                <unsigned-integer> ; | <condition-
                statement> ENDIF ;|
                ;
   <condition-statement> --> <incomplete-condition-
                        statement><alternative-part>
<incomplete-condition-statement> --> IF <conditional-
                expression> THEN <statements-list>
<conditional-expression> --> <variable-identifier> =
                <unsigned-integer>
<alternative-part> --> ELSE<statements-list> |
                <empty>
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 15

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block> ;
<block> --> BEGIN <statements-list> END
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> --> <unsigned-integer> : <statement> | <variable-
                identifier> := <unsigned-integer>
                |
                <procedure-identifier><actual-arguments> ;
                |
                GOTO <unsigned-integer> ; |
                LINK <variable-identifier> , <unsigned-
                integer> ; |
                IN <unsigned-integer>;  |
                OUT <unsigned-integer>; |
                RETURN ; |
                |
                ($ <assembly-insert-file-identifier> $)
<actual-arguments> --> ( <variable-identifier>
                <actual-arguments-list> ) | <empty>
<actual-arguments-list> --> ,<variable-identifier>
                <actual-arguments-list> |
                <empty>
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<assembly-insert-file-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 16

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>.
<block> --> BEGIN <statements-list> END
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> --> <condition-statement> ENDIF ; | WHILE
                <conditional-expression> DO
                <statements-list> ENDWHILE ;
   <condition-statement> --> <incomplete-condition-
                     statement><alternative-part>
<incomplete-condition-statement> --> IF <conditional-
                expression> THEN <statements-list>
<alternative-part> --> ELSE<statements-list> |
                <empty>
<conditional-expression> -->
                <expression><comparison-
                operator> <expression>
<comparison-operator> -->  <   |
                <=  |
                0   |
                <>  |
                >= |
                >
<expression> --> <variable-identifier> |
                <unsigned-integer>
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 17

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>.
<block> --> BEGIN <statements-list> END
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> --> LOOP <statements-list> ENDLOOP ; | FOR
                <variable-identifier> := <loop-
                declaration> ENDFOR ;
<loop-declaration> --> <expression> TO <expression>
                DO <statements-list>
 7.    <expression> --> <summand> <summands-list>   |
                - <summand> <summands-list>
<summands-list> --> <add-instruction> <summand>
                <summands-list> |
                <empty>
 9.    <add-instruction> -->     +   |
                                   -
<summand> --> <variable-identifier> |
                <unsigned-integer>
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 18

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>.
<block> --> BEGIN <statements-list> END
<statements-list> --> <statement> <statements-list>
                |
                <empty>
 <statement> --> LOOP <statements-list> ENDLOOP ; | CASE
                      <expression> OF <alternatives-list>
                ENDCASE ;
<alternatives-list> --> <alternative> <alternatives-list>
                |
                <empty>
<alternative> --> <expression> : / <statements-list>
                \
<expression> --> <multiplier><multipliers-list>
<multipliers-list> --> <multiplication-instruction>
                <multiplier><multipliers-list> |
                <empty>
 10.  <multiplication-instruction> -->      *   |
                0   |
                MOD
<multiplier> --> <variable-identifier> |
                <unsigned-integer>
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 19

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>.
   <block> --> <variable-declarations> BEGIN
                     <statements-list> END
<variable-declarations> --> VAR <declarations-list>
                |
                <empty>
<declarations-list> --> <declaration> <declarations-list>
                |
                <empty>
```

<declaration> --><variable-identifier>:<attribute> ;

```
 7.    <attribute> --> INTEGER    |
                FLOAT
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> --> <condition-statement> ENDIF ;
   <condition-statement> --> <incomplete-condition-
                      statement><alternative-part>
<incomplete-condition-statement> --> IF <conditional-
                expression> THEN <statements-list>
<alternative-part> --> ELSE <statements-list> |
                <empty>
<conditional-expression> --> <expression> =
                <expression>
<expression> --> <variable-identifier> |
                <unsigned-integer>
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 20

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>.
   <block> --> <variable-declarations> BEGIN
                      <statements-list> END
<variable-declarations> --> VAR <declarations-list>
                |
                <empty>
<declarations-list> --> <declaration> <declarations-list>
                |
                <empty>
```

<declaration> --><variable-identifier>:<attribute> ;

```
 7.    <attribute> --> INTEGER    |
                FLOAT
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> --> WHILE <conditional-expression> DO
                <statements-list> ENDWHILE ;
<conditional-expression> --> <expression>
                <comparison-operator><expression>
<comparison-operator> -->  <   |
                <=  |
                0   |
                <>  |
                >= |
                >
<expression> --> <variable-identifier> |
                <unsigned-integer>
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 21

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>.
   <block> --> <variable-declarations> BEGIN
                     <statements-list> END
<variable-declarations> --> VAR <declarations-list>
                |
                <empty>
<declarations-list> --> <declaration> <declarations-list>
                |
                <empty>
<declaration> -->
                <variable-identifier>:<attribute><attributes-list> ;
<attributes-list> --> <attribute> <attributes-list>
                |
                <empty>
 8.    <attribute> --> INTEGER    |
                FLOAT       |
                [<range>]
<range> --> <unsigned-integer> .. <unsigned-integer>
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> --> <variable> := <expression> ;  |

<expression> --> <variable> |
                <unsigned-integer>
<variable> --> <variable-identifier><dimension>
 14.   <dimension> --> [ <expression> ]   |
                <empty>
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 22

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ; <block>.
<block> --> <variable-declarations> BEGIN <statements-
                list> END
4.  <variable-declarations> --> VAR <declarations-list>   |
                <empty>
<declarations-list> --> <declaration> <declarations-
                list> |
                <empty>
<declaration> --><variable-identifier> :
                <attribute><attributes-list> ;
<attributes-list> --> <attribute> <attributes-list> |
                <empty>
8.  <attribute> --> SIGNAL     |
                INTEGER    |
                FLOAT      |
                EXT
<statements-list> --> <statement> <statements-list> |
                <empty>
<statement> --> FOR <variable-identifier> := <loop-
                declaration> ENDFOR ;
<loop-declaration> --> <expression> TO <expression> DO
                <statements-list>
<expression> --> <multiplier><multipliers-list>
<multipliers-list> --> <multiplication-instruction>
                <multiplier><multipliers-list> |
                <empty>
14. <multiplication-instruction> -->     *   |
                0   |
                &   |
                MOD
<multiplier> --> <variable-identifier> |
                <unsigned-integer>
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 23

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>.
<block> --> <declarations> BEGIN <statements-list> END
<declarations> --> <constant-declarations>
<constant-declarations> --> CONST <constant-
                declarations-list> |
                <empty>
<constant-declarations-list> --> <constant-declaration>
                <constant-declarations-list> | <empty>
<constant-declaration> --> <constant-identifier> =
                <constant>;
<statements-list> --> <statement> <statements-list> |
                <empty>
<statement> --> CASE <expression> OF <alternatives-list>
                ENDCASE ;
<alternatives-list> --> <alternative> <alternatives-list>
                |
                <empty>
<alternative> --> <expression> : /<statements-list>\
12.  <expression> --> <summand> <summands-list>   |
                - <summand> <summands-list>
<summands-list> --> <add-instruction> <summand>
                <summands-list> |
                <empty>
14.  <add-instruction> -->     +   |
                                -
<summand> --> <variable-identifier> |
                <unsigned-integer>
<constant> --> <unsigned-integer>
<variable-identifier> --> <identifier>
<constant-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 24

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ; <block>.
<block> --> <variable-declarations> BEGIN <statements-
                list> END
4.  <variable-declarations> --> VAR <declarations-list>   |
                <empty>
<declarations-list> --> <declaration> <declarations-
                list> |
                <empty>
<declaration> --><variable-identifier> : INTEGER ;
<statements-list> --> <statement> <statements-list> |
                <empty>
<statement> --> <variable-identifier> := <conditional-
                expression> ;
<conditional-expression> --> <logical-summand>
                <logical>
<logical> --> OR <logical-summand> <logical> |
                <empty>
<logical-summand> --> <logical-multiplier> <logical-
                multipliers-list>
<logical-multipliers-list> --> AND <logical-
                multiplier><logical-multipliers-list> |
                <empty>
<logical-multiplier> --> <expression><comparison-
                operator><expression> |
                [ <conditional-expression> ] |
                NOT <logical-multiplier>
<comparison-operator> -->  <   |
                <=  |
                0   |
                <>  |
                >= |
                >
<expression> --> <variable-identifier> |
                <unsigned-integer>
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> | <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 25

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>.
   <block> --> <variable-declarations> BEGIN
                     <statements-list> END
<variable-declarations> --> VAR <declarations-list>
                |
                <empty>
<declarations-list> --> <declaration> <declarations-list>
                |
                <empty>
<declaration> --><variable-identifier>: INTEGER ;
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> --> <variable-identifier> :=
                <expression> ;
9.    <expression> --> <summand> <summands-list>   |
                - <summand> <summands-list>
<summands-list> --> <add-instruction> <summand>
                <summands-list> |
                <empty>
 11.  <add-instruction> -->     +   |
                                -
<summand> --> <multiplier><multipliers-list>
<multipliers-list> --> <multiplication-instruction>
                <multiplier><multipliers-list> | <empty>
 14.  <multiplication-instruction> -->      *   |
                                            /
   <multiplier> --> <variable-identifier> |
                     <unsigned-integer> |
                ( <expression> )
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 26

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>.
<block> --> <declarations> BEGIN <statements-list>
                END
<declarations> --> <constant-declarations><variable-
                declarations>
<constant-declarations> --> CONST <constant-
                declarations-list> |
                <empty>
<constant-declarations-list> --> <constant-declaration>
                <constant-declarations-list> | <empty>
<constant-declaration> --> <constant-identifier> =
                <constant>;
<constant> --> <complex-constant> |
                <unsigned-constant>   |
                - <unsigned-constant>
<variable-declarations> --> VAR <declarations-list>
                |
                <empty>
<declarations-list> --> <declaration> <declarations-list>
                |
                <empty>
<declaration> --><variable-identifier> <identifiers-list>:
                <attribute><attributes-list> ;
<identifiers-list> --> , <variable-identifier>
                <identifiers-list> |
                <empty>
<attributes-list> --> <attribute> <attributes-list>
                |
                <empty>
 14.   <attribute> --> SIGNAL     |
                COMPLEX     |
                INTEGER     |
                FLOAT       |
                BLOCKFLOAT  |
                EXT         |
                [<range><ranges-list>]
<ranges-list> --> ,<range> <ranges-list> |
                <empty>
<range> --> <unsigned-integer> .. <unsigned-integer>
<statements-list> --> <statement> <statements-list>
                |
                <empty>
```

```
<statement> -->
                <condition-statement> ENDIF ; |
                WHILE <conditional-expression> DO
                <statements-list> ENDWHILE ; |
                LINK <variable-identifier> , <unsigned-
                integer> ; |
                IN <unsigned-integer>;  |
                OUT <unsigned-integer>;
   <condition-statement> --> <incomplete-condition-
                     statement><alternative-part>
<incomplete-condition-statement> --> IF <conditional-
                expression> THEN <statements-list>
<alternative-part> --> ELSE<statements-list> |
                <empty>
<conditional-expression> --> <logical-summand>
                <logical>
<logical> --> OR <logical-summand> <logical> |
                <empty>
<logical-summand> --> <logical-multiplier> <logical-
                multipliers-list>
<logical-multipliers-list> --> AND <logical-
                multiplier><logical-multipliers-list> |
                <empty>
<logical-multiplier> --> <expression> <comparison-
                operator><expression> |
                [ <conditional-expression> ] |
                NOT <logical-multiplier>
<comparison-operator> -->  <   |
                <=  |
                0   |
                <>  |
                >= |
                >
<expression> --> <variable-identifier> |
                <constant-identifier>
<complex-constant> --> '<complex-number>'
<unsigned-constant> --> <unsigned-number>
<complex-number> --> <left-part> <right-part>
<left-part> --> <expression> |
                <empty>
<right-part> --> ,<expression> |
                $EXP( <expression> )   |
                <empty>
<constant-identifier> --> <identifier>
<variable-identifier> --> <identifier>
<identifier> --> <letter><string>
```

```
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-number> --> <integer-part><fractional-part>
<integer-part> --> <unsigned-integer>
<fractional-part> --> #<sign><unsigned-integer> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<sign> --> + |
                0 |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 27

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>.
<block> --> BEGIN <conditional-expression> END
<conditional-expression> --> <logical-summand>
                <logical>
<logical> --> OR <logical-summand> <logical> |
                <empty>
<logical-summand> --> <logical-multiplier> <logical-
                multipliers-list>
<logical-multipliers-list> --> AND <logical-
                multiplier><logical-multipliers-list> |
                <empty>
<logical-multiplier> --> <expression><comparison-
                operator><expression> |
                [ <conditional-expression> ] |
                NOT <logical-multiplier>
<comparison-operator> --> < |
                <=  |
                =   |
                <>  |
                >=  |
                >
 10.  <expression> --> <summand> <summands-list>   |
                - <summand> <summands-list>
```

```
<summands-list> --> <add-instruction> <summand>
                    <summands-list> |
                    <empty>
12.    <add-instruction> -->     +    |
                    -
                    |    !
                    |
<summand> --> <multiplier><multipliers-list>
<multipliers-list> --> <multiplication-instruction>
                    <multiplier><multipliers-list> | <empty>
15.    <multiplication-instruction> -->     *    |
                    /    |
                    & |
                    MOD
<multiplier> --> <unsigned-constant>  |
                    <complex-constant> | <variable> |
                    <builtin-function-identifier><actual-
                    arguments> |

                    ( <expression> )  |
                    - <multiplier>    |
                    0 <multiplier>
<variable> --> <variable-identifier><dimension> |
                    <complex-variable>
<complex-variable> --> "<complex-number>"
19.          <dimension> --> [<expression><expressions-list>] |
             <empty>
<expressions-list> --> ,<expression><expressions-list> |
                    <empty>
<complex-constant> --> '<complex-number>'
<unsigned-constant> --> <unsigned-number>
<complex-number> --> <left-part> <right-part>
<left-part> --> <expression> |
                    <empty>
<right-part> --> ,<expression> |
                    $EXP( <expression> )    |
                    <empty>
<actual-arguments> --> ( <arguments-list> )
<arguments-list> --> <unsigned-integer> <argument-list>
                    |
                    <empty>
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<builtin-function-identifier> --> <identifier>
<identifier> --> <letter><string>
```

```
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-number> --> <integer-part><fractional-part>
<integer-part> --> <unsigned-integer>
<fractional-part> --> #<sign><unsigned-integer> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<sign> --> + |
                0 |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 28

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>. |
                PROCEDURE <procedure-identifier>
                <parameters-list> ; <block> ;
<block> --> <declarations> BEGIN <statements-list>
                END
<declarations> --> <label-declarations> <variable-
                declarations> <math-function-
                declarations><procedure-declarations>
<label-declarations> --> LABEL <unsigned-integer>
                <labels-list>; |
                <empty>
<labels-list> --> , <unsigned-integer> <labels-list> |
                <empty>
<variable-declarations> --> VAR <declarations-list>
                |
                <empty>
<declarations-list> --> <declaration> <declarations-list>
                |
                <empty>
<declaration> --><variable-identifier> <identifiers-list>:
                <attribute>;
```

```
<identifiers-list> --> , <variable-identifier>
                <identifiers-list> |
                <empty>
<math-function-declarations> --> DEFFUNC <function-list>
                |
                <empty>
<function-list> --> <function><function-list> |
                <empty>
<function> --> <function-identifier> =
                <constant><function-characteristic> ;
<constant> --> <unsigned-integer>
<function-characteristic> --> \ <unsigned-integer> ,
                <unsigned-integer>
<procedure-declarations> --> <procedure> <procedure-
                declarations> |
                <empty>
<procedure> --> PROCEDURE <procedure-
                identifier><parameters-list> ;
<parameters-list> --> ( <attributes-list> ) |
                <empty>
<attributes-list> --> <attribute> , <attributes-
                list> |
                <empty>
 20.   <attribute> --> INTEGER     |
                FLOAT       |
                BLOCKFLOAT
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> --> <unsigned-integer> : <statement> | <procedure-
                identifier> <actual-arguments> ;
                |
                LOOP <statements-list> ENDLOOP ; |
                GOTO <unsigned-integer> ; |
                LINK <variable-identifier> , <unsigned-
                integer> ; |
                IN  <unsigned-integer>;  |
                OUT <unsigned-integer>; |

                |
                ($ <assembly-insert-file-identifier> $)
<actual-arguments> --> ( <variable-identifier>
                <identifiers-list> ) |
                <empty>
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<function-identifier>  --> <identifier>
```

```
<assembly-insert-file-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 29

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>. |
                PROCEDURE <procedure-identifier>
                <parameters-list> ; <block> ;
<block> --> <declarations> BEGIN <statements-list>
                END
<declarations> --> <constant-declarations> <variable-
                declarations><math-function-
                declarations><procedure-declarations>
<constant-declarations> --> CONST <constant-
                declarations-list> |
                <empty>
<constant-declarations-list> --> <constant-declaration>
                <constant-declarations-list> | <empty>
<constant-declaration> --> <constant-identifier> =
                <constant>;
<constant> --> <complex-constant> |
                <unsigned-constant>   |
                - <unsigned-constant>
<variable-declarations> --> VAR <declarations-list>
                |
                <empty>
<declarations-list> --> <declaration> <declarations-list>
                |
                <empty>
<declaration> -->><variable-identifier><identifiers-list>:
                <attribute><attributes-list> ;
<identifiers-list> --> , <variable-identifier>
                <identifiers-list> |
                <empty>
```

```
<attributes-list> --> <attribute> <attributes-list>
                |
                <empty>
 14.  <attribute> --> SIGNAL     |
                COMPLEX     |
                INTEGER     |
                FLOAT       |
                BLOCKFLOAT  |
                EXT         |
                [<range><ranges-list>]
<ranges-list> --> ,<range> <ranges-list> |
                <empty>
<range> --> <unsigned-integer> .. <unsigned-integer>
<math-function-declarations> --> DEFFUNC <function-list>
                |
                <empty>
<function-list> --> <function> function-list> |
                <empty>
<function> --> <function-identifier> =
                <constant><function-characteristic> ;
<constant> --> <unsigned-integer><function-
                characteristic> --> \ <unsigned-integer> ,
                <unsigned-integer>
<procedure-declarations> --> <procedure> <procedure-
                declarations> |
                <empty>
<procedure> --> PROCEDURE <procedure-identifier>
                <parameters-list> ;
<parameters-list> --> ( <declarations-list> ) |
                <empty>
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> --> LINK <variable-identifier> ,
                <unsigned-integer> ; |
                IN <unsigned-integer>; |
                OUT <unsigned-integer>;
<complex-constant> --> '<complex-number>'
<unsigned-constant> --> <unsigned-number>
<complex-number> --> <left-part> <right-part>
<left-part> --> <unsigned-integer> |
                <empty>
<right-part> --> ,<unsigned-integer> |
                $EXP( <unsigned-integer> ) |
                <empty>
<constant-identifier> --> <identifier>
<variable-identifier> --> <identifier>
```

```
<procedure-identifier> --> <identifier>
<function-identifier>  --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-number> --> <integer-part><fractional-part>
<integer-part> --> <unsigned-integer>
<fractional-part> --> #<sign><unsigned-integer> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<sign> --> + |
                0 |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 30

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>.
<block> --> <declarations> BEGIN <statements-list>
                END
<declarations> --> <constant-declarations>
                <variable-declarations>
<constant-declarations> --> CONST <constant-
                declarations-list> |
                <empty>
<constant-declarations-list> --> <constant-declaration>
                <constant-declarations-list> | <empty>
<constant-declaration> --> <constant-identifier> =
                <constant>;
<constant> -->
                <unsigned-constant>   |
                - <unsigned-constant>
<variable-declarations> --> VAR <declarations-list>
                |
                <empty>
```

```
<declarations-list> --> <declaration> <declarations-list>
                |
                <empty>
<declaration> --><variable-identifier> <identifiers-list>:
                <attribute><attributes-list> ;
<identifiers-list> --> , <variable-identifier>
                <identifiers-list> |
                <empty>
<attributes-list> --> <attribute> <attributes-list>
                |
                <empty>
 14.   <attribute> --> SIGNAL     |
                COMPLEX     |
                INTEGER     |
                FLOAT       |
                BLOCKFLOAT |
                EXT         |
                [<range><ranges-list>]
<ranges-list> --> ,<range> <ranges-list> |
                <empty>
<range> --> <unsigned-integer> .. <unsigned-integer>
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> -->
                LOOP <statements-list> ENDLOOP ; |
                FOR <variable-identifier> :=
                <loop-declaration> ENDFOR ; |
                CASE <expression> OF <alternatives-
                list> ENDCASE ;
<loop-declaration> --> <expression> TO <expression> DO
                <statements-list>
<alternatives-list> --> <alternative> <alternatives-list>
                |
                <empty>
<alternative> --> <expression> : / <statements-list>
                \
22.    <expression> --> <summand> <summands-list>   |
                - <summand> <summands-list>
<summands-list> --> <add-instruction> <summand>
                <summands-list> |
                <empty>
24.    <add-instruction> -->     +   |
                0   |
                !
<summand> --> <multiplier><multipliers-list>
```

```
<multipliers-list> --> <multiplication-instruction>
                <multiplier><multipliers-list> | <empty>
27.   <multiplication-instruction> -->     *   |
                /   |
                & |
                MOD
<multiplier> --> <unsigned-constant> | <constant-
                identifier> | <variable> | <function-
                identifier> | <builtin-function-
                identifier><actual-arguments> |

                ( <expression> )  |
                - <multiplier>    |
                0 <multiplier>
<variable> --> <variable-identifier><dimension>
<dimension> -->[<expression><expressions-list>] |
                <empty>
<expressions-list> --> ,<expression><expressions-list> |
                <empty>
<unsigned-constant> --> <unsigned-number>
<constant-identifier> --> <identifier>
<variable-identifier> --> <identifier>
<builtin-function-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-number> --> <integer-part><fractional-part>
<integer-part> --> <unsigned-integer>
<fractional-part> --> #<sign><unsigned-integer> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<sign> --> + |
                0 |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 31

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>. |
                PROCEDURE <procedure-identifier>
                <parameters-list> ; <block> ;
<block> --> <declarations> BEGIN <statements-list>
                END
<declarations> --> <constant-declarations><variable-
                declarations><math-function-
                declarations><procedure-declarations>
<constant-declarations> --> CONST <constant-
                declarations-list> |
                <empty>
<constant-declarations-list> --> <constant-declaration>
                <constant-declarations-list> | <empty>
<constant-declaration> --> <constant-identifier> =
                <constant>;
<constant> -->
                <unsigned-constant>   |
                - <unsigned-constant>
<variable-declarations> --> VAR <declarations-list>
                |
                <empty>
<declarations-list> --> <declaration> <declarations-list>
                |
                <empty>
<declaration> --><variable-identifier> <identifiers-list>:
                <attribute><attributes-list> ;
<identifiers-list> --> , <variable-identifier>
                <identifiers-list> |
                <empty>
<attributes-list> --> <attribute> <attributes-list>
                |
                <empty>
 14.   <attribute> --> SIGNAL     |
                INTEGER     |
                FLOAT       |
                BLOCKFLOAT |
                EXT         |
                [<range><ranges-list>]
<ranges-list> --> ,<range> <ranges-list> |
                <empty>
<range> --> <unsigned-integer> .. <unsigned-integer>
<math-function-declarations> --> DEFFUNC <function-list>
                |
                <empty>
```

```
<function-list> --> <function><function-list> |
                <empty>
<function> --> <function-identifier> =
                <expression><function-characteristic> ;
<function-characteristic> --> \ <unsigned-integer> ,
                <unsigned-integer>
<procedure-declarations> --> <procedure> <procedure-
                declarations> |
                <empty>
<procedure> --> PROCEDURE <procedure-
                identifier><parameters-list> ;
<parameters-list> --> ( <declarations-list> ) |
                <empty>
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> --> <variable> := <expression> ; |
                <procedure-identifier><actual-arguments> ;
                |
                <condition-statement> ENDIF ; |
                WHILE <conditional-expression> DO
                <statements-list> ENDWHILE ; |
                LOOP <statements-list> ENDLOOP ; |
                FOR <variable-identifier> :=
                <loop-declaration> ENDFOR ; |
                CASE <expression> OF <alternatives-
                list> ENDCASE ; |
                LINK <variable-identifier> , <unsigned-
                integer> ; |
                IN  <unsigned-integer>;  |
                OUT    <unsigned-integer>;
                | RETURN ; |
                ;
   <condition-statement> --> <incomplete-condition-
                      statement><alternative-part>
<incomplete-condition-statement> --> IF
                <conditional-expression> THEN <statements-
                list>
<alternative-part> --> ELSE<statements-list> |
                <empty>
<loop-declaration> --> <expression> TO <expression> DO
                <statements-list>
<actual-arguments> --> ( <expression> <actual-
                arguments-list> ) |
                <empty>
```

```
<actual-arguments-list> --> ,<expression> <actual-
                arguments-list> |
                <empty>
<alternatives-list> --> <alternative> <alternatives-list>
                |
                <empty>
<alternative> --> <expression> : / <statements-list>
                \
<conditional-expression> --> <logical-summand>
                <logical>
<logical> --> OR <logical-summand> <logical> |
                <empty>
<logical-summand> --> <logical-multiplier> <logical-
                multipliers-list>
<logical-multipliers-list> --> AND <logical-
                multiplier><logical-multipliers-list> |
                <empty>
<logical-multiplier> --> <expression> <comparison-
                operator><expression> |
                [ <conditional-expression> ] |
                NOT <logical-multiplier>
<comparison-operator> -->  <   |
                <=  |
                |
                <>  |
                >= |
                >
40.   <expression> --> <summand> <summands-list>   |
                - <summand> <summands-list>
<summands-list> --> <add-instruction> <summand>
                <summands-list> |
                <empty>
42.   <add-instruction> -->     +   |
                -
                |   !
                |
<summand> --> <multiplier><multipliers-list>
<multipliers-list> --> <multiplication-instruction>
                <multiplier><multipliers-list> | <empty>
45.   <multiplication-instruction> -->     *   |
                /   |
                & |
                MOD
<multiplier> --> <unsigned-constant> |
                <constant-identifier> |
                <variable> | <function-
                identifier> |
```

```
                <builtin-function-identifier><actual-
                arguments> |
                ( <expression> )  |
                - <multiplier>    |
                0 <multiplier>
<variable> --> <variable-identifier><dimension>
<dimension> --> [ <expression> <expressions-list> ]
                |
                <empty>
<expressions-list> --> ,<expression><expressions-list> |
                <empty>
<unsigned-constant> --> <unsigned-number>
<constant-identifier> --> <identifier>
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<function-identifier>  --> <identifier>
<builtin-function-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-number> --> <integer-part><fractional-part>
<integer-part> --> <unsigned-integer>
<fractional-part> --> #<sign><unsigned-integer> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<sign> --> + |
                0 |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 32

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>.
<block> --> <declarations> BEGIN <statements-list>
                END
<declarations> --> <label-declarations><constant-
                declarations><variable-declarations>
```

```
<label-declarations> --> LABEL <unsigned-integer>
                <labels-list>; |
                <empty>
<labels-list> --> , <unsigned-integer> <labels-list> |
                <empty>
<constant-declarations> --> CONST <constant-
                declarations-list> |
                <empty>
<constant-declarations-list> --> <constant-declaration>
                <constant-declarations-list> | <empty>
<constant-declaration> --> <constant-identifier> =
                <constant>;
<constant> --> <complex-constant> |
                <unsigned-constant>   |
                - <unsigned-constant>
<variable-declarations> --> VAR <declarations-list>
                |
                <empty>
<declarations-list> --> <declaration> <declarations-list>
                |
                <empty>
<declaration> --><variable-identifier> <identifiers-list>:
                <attribute><attributes-list> ;
<identifiers-list> --> , <variable-identifier>
                <identifiers-list> |
                <empty>
<attributes-list> --> <attribute> <attributes-list>
                |
                <empty>
 16.  <attribute> --> SIGNAL     |
                COMPLEX     |
                INTEGER     |
                FLOAT       |
                BLOCKFLOAT |
                EXT         |
                [<range><ranges-list>]
<ranges-list> --> ,<range> <ranges-list> |
                <empty>
<range> --> <unsigned-integer> .. <unsigned-integer>
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> --> <unsigned-integer> : <statement> |
                <variable> := <expression> ; |
                <condition-statement> ENDIF ; |
```

```
                WHILE <conditional-expression> DO
                <statements-list> ENDWHILE ; |
                LOOP <statements-list> ENDLOOP ; |
                FOR <variable-identifier> :=
                <loop-declaration> ENDFOR ; |
                CASE <expression> OF <alternatives-
                list> ENDCASE ; |
                GOTO <unsigned-integer> ; |
                LINK <variable-identifier> , <unsigned-
                integer> ; |
                IN <unsigned-integer>;  |
                OUT <unsigned-integer>; |
                ;
   <condition-statement> --> <incomplete-condition-
                      statement><alternative-part>
<incomplete-condition-statement> --> IF <conditional-
                expression> THEN <statements-list>
<alternative-part> --> ELSE<statements-list> |
                <empty>
<loop-declaration> --> <expression> TO <expression> DO
                <statements-list>
<alternatives-list> --> <alternative> <alternatives-list>
                |
                <empty>
<alternative> --> <expression> : / <statements-list>
                \
<conditional-expression> --> <logical-summand>
                <logical>
<logical> --> OR <logical-summand> <logical> |
                <empty>
<logical-summand> --> <logical-multiplier> <logical-
                multipliers-list>
<logical-multipliers-list> --> AND <logical-
                multiplier><logical-multipliers-list> |
                <empty>
<logical-multiplier> --> <expression> <comparison-
                operator> <expression> |
                [ <conditional-expression> ] |
                NOT <logical-multiplier>
<comparison-operator> -->  <   |
                <=  |
                |
                <>  |
                >= |
                >
```

```
33.   <expression> --> <summand> <summands-list>   |
                - <summand> <summands-list>
<summands-list> --> <add-instruction> <summand>
                <summands-list> |
                <empty>
35.   <add-instruction> -->     +   |
                -   |
                !
<summand> --> <multiplier><multipliers-list>
<multipliers-list> --> <multiplication-instruction>
                <multiplier><multipliers-list> | <empty>
38.   <multiplication-instruction> -->     *   |
                /   |
                & |
                MOD
<multiplier> --> <unsigned-constant>  |
                <complex-constant> | <constant-
                identifier> | <variable> | <function-
                identifier> | <builtin-function-
                identifier><actual-arguments> |


                ( <expression> )  |
                - <multiplier>    |
                0 <multiplier>
<variable> --> <variable-identifier><dimension> |
                <complex-variable>
<complex-variable> --> "<complex-number>"
<dimension> --> [ <expression> <expressions-list> ]
                |
                <empty>
<expressions-list> --> ,<expression><expressions-list> |
                <empty>
<complex-constant> --> '<complex-number>'
<unsigned-constant> --> <unsigned-number>
<complex-number> --> <left-part> <right-part>
<left-part> --> <expression> |
                <empty>
<right-part> --> ,<expression> |
                $EXP( <expression> )   |
                <empty>
<constant-identifier> --> <identifier>
<variable-identifier> --> <identifier>
<builtin-function-identifier> --> <identifier>
<identifier> --> <letter><string>
```

```
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-number> --> <integer-part><fractional-part>
<integer-part> --> <unsigned-integer>
<fractional-part> --> #<sign><unsigned-integer> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<sign> --> + |
                0 |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 33

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>.
<block> --> <declarations> BEGIN <statements-list>
                END
<declarations> --> <constant-declarations><variable-
                declarations><math-function-declarations>
<constant-declarations> --> CONST <constant-
                declarations-list> |
                <empty>
<constant-declarations-list> --> <constant-declaration>
                <constant-declarations-list> | <empty>
<constant-declaration> --> <constant-identifier> =
                <constant>;
<constant> --> <complex-constant> |
                <unsigned-constant>   |
                - <unsigned-constant>
<variable-declarations> --> VAR <declarations-list>
                |
                <empty>
<declarations-list> --> <declaration> <declarations-list>
                |
                <empty>
<declaration> --><variable-identifier> <identifiers-list>:
                <attribute><attributes-list> ;
```

```
<identifiers-list> --> , <variable-identifier>
                <identifiers-list> |
                <empty>
<attributes-list> --> <attribute> <attributes-list>
                |
                <empty>
 14.   <attribute> --> COMPLEX     |
                INTEGER     |
                FLOAT       |
                BLOCKFLOAT |
                EXT         |
                [<range><ranges-list>]
<ranges-list> --> ,<range> <ranges-list> |
                <empty>
<range> --> <unsigned-integer> .. <unsigned-integer>
<math-function-declarations> --> DEFFUNC <function-list>
                |
                <empty>
<function-list> --> <function><function-list> |
                <empty>
<function> --> <function-identifier> =
                <expression><function-characteristic> ;
<function-characteristic> --> \ <unsigned-integer> ,
                <unsigned-integer>
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> --> <variable> := <expression> ; |
                <condition-statement> ENDIF ; | WHILE
                <conditional-expression> DO
                <statements-list> ENDWHILE ; |
                LOOP <statements-list> ENDLOOP ; |
                FOR <variable-identifier> :=
                <loop-declaration> ENDFOR ; |
                CASE <expression> OF <alternatives-
                list> ENDCASE ; |
                LINK <variable-identifier> , <unsigned-
                integer> ; |
                IN  <unsigned-integer>;  |
                OUT <unsigned-integer>; |
                ;
   <condition-statement> --> <incomplete-condition-
                     statement><alternative-part>
<incomplete-condition-statement> --> IF <conditional-
                expression> THEN <statements-list>
```

```
<alternative-part> --> ELSE<statements-list> |
                <empty>
<loop-declaration> --> <expression> TO <expression> DO
                <statements-list>
<alternatives-list> --> <alternative> <alternatives-list>
                |
                <empty>
<alternative> --> <expression> : / <statements-list>
                \
<conditional-expression> --> <logical-summand>
                <logical>
<logical> --> OR <logical-summand> <logical> |
                <empty>
<logical-summand> --> <logical-multiplier> <logical-
                multipliers-list>
<logical-multipliers-list> --> AND <logical-
                multiplier><logical-multipliers-list> |
                <empty>
<logical-multiplier> --> <expression> <comparison-
                operator><expression> |
                [ <conditional-expression> ] |
                NOT <logical-multiplier>
<comparison-operator> -->  <   |
                <=  |
                |
                <>  |
                >= |
                >
35.   <expression> --> <summand> <summands-list>   |
                - <summand> <summands-list>
<summands-list> --> <add-instruction> <summand>
                <summands-list> |
                <empty>
37.   <add-instruction> -->     +   |
                -   |
                !
<summand> --> <multiplier><multipliers-list>
<multipliers-list> --> <multiplication-instruction>
                <multiplier><multipliers-list> | <empty>
40.   <multiplication-instruction> -->     *   |
                /   |
                & |
                MOD
<multiplier> --> <unsigned-constant>  |
                <complex-constant> |
                <constant-identifier> |
```

```
                <variable>  |
                <function-identifier>  |
                <builtin-function-identifier><actual-
                arguments>  |
                ( <expression> )  |
                - <multiplier>    |
                0 <multiplier>
<variable> --> <variable-identifier><dimension> |
                <complex-variable>
<complex-variable> --> "<complex-number>"
<dimension> --> [ <expression> <expressions-list> ]
                |
                <empty>
<expressions-list> --> ,<expression><expressions-list> |
                <empty>
<complex-constant> --> '<complex-number>'
<unsigned-constant> --> <unsigned-number>
<complex-number> --> <left-part> <right-part>
<left-part> --> <expression> |
                <empty>
<right-part> --> ,<expression> |
                $EXP( <expression> )   |
                <empty>
<constant-identifier> --> <identifier>
<variable-identifier> --> <identifier>
<function-identifier>  --> <identifier>
<builtin-function-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-number> --> <integer-part><fractional-part>
<integer-part> --> <unsigned-integer>
<fractional-part> --> #<sign><unsigned-integer> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<sign> --> + |
                0 |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 34

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>. |
                PROCEDURE <procedure-identifier>
                <parameters-list> ; <block> ;
<block> --> <declarations> BEGIN <statements-list>
                END
<declarations> --> <label-declarations><constant-
                declarations><variable-
                declarations><procedure-declarations>
<label-declarations> --> LABEL <unsigned-integer>
                <labels-list>; |
                <empty>
<labels-list> --> , <unsigned-integer> <labels-list> |
                <empty>
<constant-declarations> --> CONST <constant-
                declarations-list> |
                <empty>
<constant-declarations-list> --> <constant-declaration>
                <constant-declarations-list> | <empty>
<constant-declaration> --> <constant-identifier> =
                <constant>;
<constant> -->
                <unsigned-constant>   |
                - <unsigned-constant>
<variable-declarations> --> VAR <declarations-list>
                |
                <empty>
<declarations-list> --> <declaration> <declarations-list>
                |
                <empty>
<declaration> --><variable-identifier><identifiers-
                list>:<attribute><attributes-list> ;
<identifiers-list> --> , <variable-identifier>
                <identifiers-list> |
                <empty>
<attributes-list> --> <attribute> <attributes-list>
                |
                <empty>
 16.  <attribute> --> SIGNAL     |
                INTEGER     |
                FLOAT       |
                BLOCKFLOAT |
```

```
                EXT        |
                [<range><ranges-list>]
<ranges-list> --> ,<range> <ranges-list> |
                <empty>
<range> --> <unsigned-integer> .. <unsigned-integer>
<procedure-declarations> --> <procedure> <procedure-
                declarations> |
                <empty>
<procedure> --> PROCEDURE <procedure-
                identifier><parameters-list> ;
<parameters-list> --> ( <declarations-list> ) |
                <empty>
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> --> <unsigned-integer> : <statement> |
                <variable> := <expression> ; | <procedure-
                identifier><actual-arguments> ;
                |
                <condition-statement> ENDIF ; |
                WHILE <conditional-expression> DO
                <statements-list> ENDWHILE ; |
                LOOP <statements-list> ENDLOOP ; |
                FOR <variable-identifier> :=
                <loop-declaration> ENDFOR ; |
                CASE <expression> OF <alternatives-
                list> ENDCASE ; |
                GOTO <unsigned-integer> ; |
                LINK <variable-identifier> , <unsigned-
                integer> ; |
                IN  <unsigned-integer>;  |
                OUT   <unsigned-integer>;
                | RETURN ; |
                ;
   <condition-statement> --> <incomplete-condition-
                     statement><alternative-part>
<incomplete-condition-statement> --> IF
                <conditional-expression> THEN <statements-
                list>
<alternative-part> --> ELSE<statements-list> |
                <empty>
<loop-declaration> --> <expression> TO <expression> DO
                <statements-list>
<actual-arguments> --> ( <expression> <actual-
                arguments-list> ) |
                <empty>
```

```
<actual-arguments-list> --> ,<expression> <actual-
                arguments-list> |
                <empty>
<alternatives-list> --> <alternative> <alternatives-list>
                |
                <empty>
<alternative> --> <expression> : / <statements-list>
                \
<conditional-expression> --> <logical-summand>
                <logical>
<logical> --> OR <logical-summand> <logical> |
                <empty>
<logical-summand> --> <logical-multiplier> <logical-
                multipliers-list>
<logical-multipliers-list> --> AND <logical-
                multiplier><logical-multipliers-list> |
                <empty>
<logical-multiplier> --> <expression> <comparison-
                operator><expression> |
                [ <conditional-expression> ] |
                NOT <logical-multiplier>
<comparison-operator> -->  <   |
                <=  |
                |
                <>  |
                >= |
                >
38.   <expression> --> <summand> <summands-list>   |
                - <summand> <summands-list>
<summands-list> --> <add-instruction> <summand>
                <summands-list> |
                <empty>
40.   <add-instruction> -->     +   |
                -   |
                !
<summand> --> <multiplier><multipliers-list>
<multipliers-list> --> <multiplication-instruction>
                <multiplier><multipliers-list> | <empty>
43.   <multiplication-instruction> -->     *   |
                /   |
                & |
                MOD
<multiplier> --> <unsigned-constant> | <constant-
                identifier> | <variable> | <builtin-
                function-identifier><actual-
```

```
                arguments>  |
                ( <expression> )  |
                - <multiplier>    |
                0 <multiplier>
<variable> --> <variable-identifier><dimension>
<dimension> --> [ <expression> <expressions-list> ]
                |
                <empty>
<expressions-list> --> ,<expression><expressions-list> |
                <empty>
<unsigned-constant> --> <unsigned-number>
<constant-identifier> --> <identifier>
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<builtin-function-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-number> --> <integer-part><fractional-part>
<integer-part> --> <unsigned-integer>
<fractional-part> --> #<sign><unsigned-integer> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<sign> --> + |
                0 |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

Варіант 35

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>. |
                PROCEDURE <procedure-identifier>
                <parameters-list> ; <block> ;
<block> --> <declarations> BEGIN <statements-list>
                END
```

```
<declarations> --> <constant-declarations><variable-
                declarations><math-function-
                declarations><procedure-declarations>
<constant-declarations> --> CONST <constant-
                declarations-list> |
                <empty>
<constant-declarations-list> --> <constant-declaration>
                <constant-declarations-list> | <empty>
<constant-declaration> --> <constant-identifier> =
                <constant>;
<constant> --> <complex-constant> |
                <unsigned-constant>   |
                0 <unsigned-constant>
<variable-declarations> --> VAR <declarations-list>
                |
                <empty>
<declarations-list> --> <declaration> <declarations-list>
                |
                <empty>
<declaration> --><variable-identifier> <identifiers-
                list> : <attribute><attributes-list> ;
<identifiers-list> --> , <variable-identifier>
                <identifiers-list> |
                <empty>
<attributes-list> --> <attribute> <attributes-list>
                |
                <empty>
<attribute> -->
```

```
<math-function-declarations> --> DEFFUNC <function-list>
                |
                <empty>
<function-list> --> <function><function-list> |
                <empty>
<function> --> <function-identifier> =
                <expression><function-characteristic> ;
<function-characteristic> --> \ <unsigned-integer> ,
                <unsigned-integer>
<procedure-declarations> --> <procedure> <procedure-
                declarations> |
                <empty>
```

```
<procedure> --> PROCEDURE <procedure-
                identifier><parameters-list> ;
<parameters-list> --> ( <declarations-list> ) |
                <empty>
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> --> <variable> := <expression> ; |
                <procedure-identifier><actual-arguments> ;
                |
                <condition-statement> ENDIF ; |
                WHILE <conditional-expression> DO
                <statements-list> ENDWHILE ; |
                LOOP <statements-list> ENDLOOP ; |
                FOR <variable-identifier> :=
                <loop-declaration> ENDFOR ; |
                CASE <expression> OF <alternatives-
                list> ENDCASE ; |

                |
                ($ <assembly-insert-file-identifier> $)
   <condition-statement> --> <incomplete-condition-
                      statement><alternative-part>
<incomplete-condition-statement> --> IF <conditional-
                expression> THEN <statements-list>
<alternative-part> --> ELSE<statements-list> |
                <empty>
<loop-declaration> --> <expression> TO <expression> DO
                <statements-list>
<actual-arguments> --> ( <expression> <actual-
                arguments-list> ) |
                <empty>
<actual-arguments-list> --> ,<expression> <actual-
                arguments-list> |
                <empty>
<alternatives-list> --> <alternative> <alternatives-list>
                |
                <empty>
<alternative> --> <expression> : / <statements-list>
                \
<conditional-expression> --> <logical-summand>
                <logical>
<logical> --> OR <logical-summand> <logical> |
                <empty>
<logical-summand> --> <logical-multiplier> <logical-
                multipliers-list>
```

```
<logical-multipliers-list> --> AND <logical-
                multiplier><logical-multipliers-list> |
                <empty>
<logical-multiplier> --> <expression> <comparison-
                operator><expression> |
                [ <conditional-expression> ] |
                NOT <logical-multiplier>
<comparison-operator> -->  <   |
                <=  |
                |
                <>  |
                >= |
                >
38.   <expression> --> <summand> <summands-list>   |
                - <summand> <summands-list>
<summands-list> --> <add-instruction> <summand>
                <summands-list> |
                <empty>
40.   <add-instruction> -->     +   |
                -   |
                !
<summand> --> <multiplier><multipliers-list>
<multipliers-list> --> <multiplication-instruction>
                <multiplier><multipliers-list> | <empty>
43.   <multiplication-instruction> -->     *   |
                /   |
                & |
                MOD
<multiplier> --> <unsigned-constant>  |
                <complex-constant> | <constant-
                identifier> | <variable> | <function-
                identifier> | <builtin-function-
                identifier><actual-arguments> |


                ( <expression> )  |
                - <multiplier>    |
                ^ <multiplier>
<variable> --> <variable-identifier><dimension> |
                <complex-variable>
<complex-variable> --> "<complex-number>"
<dimension> --> [ <expression> <expressions-list> ]
                |
                <empty>
```

```
<expressions-list> --> ,<expression><expressions-list> |
                <empty>
<complex-constant> --> '<complex-number>'
<unsigned-constant> --> <unsigned-number>
<complex-number> --> <left-part> <right-part>
<left-part> --> <expression> |
                <empty>
<right-part> --> ,<expression> |
                $EXP( <expression> )   |
                <empty>
<constant-identifier> --> <identifier>
<variable-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<function-identifier>  --> <identifier>
<builtin-function-identifier> --> <identifier>
<assembly-insert-file-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-number> --> <integer-part><fractional-part>
<integer-part> --> <unsigned-integer>
<fractional-part> --> #<sign><unsigned-integer> |
                <empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
<sign> --> + |
                   0 |
                   <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

ДОДАТОК 2. ГРАМАТИКА МОВИ SIGNAL [7]

Copyright (c) Oleksandr Marchenko, 1987-1988

```
<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
                <block>. |
                PROCEDURE <procedure-
                identifier><parameters-list> ; <block> ;
<block> --> <declarations> BEGIN <statements-list>
                END
<declarations> --> <label-declarations><constant-
                declarations><variable-declarations><math-
                function-declarations><procedure-
                declarations>
<label-declarations> --> LABEL <unsigned-integer>
                <labels-list>; |
                <empty>
<labels-list> --> , <unsigned-integer> <labels-list> |
                <empty>
<constant-declarations> --> CONST <constant-
                declarations-list> |
                <empty>
<constant-declarations-list> --> <constant-declaration>
                <constant-declarations-list> | <empty>
<constant-declaration> --> <constant-identifier> =
                <constant>;
<constant> --> <complex-constant> |
                <unsigned-constant>   |
                - <unsigned-constant>
<variable-declarations> --> VAR <declarations-list>
                |
                <empty>
<declarations-list> --> <declaration> <declarations-list>
                |
                <empty>
<declaration> -->><variable-identifier><identifiers-
                list>:<attribute><attributes-list> ;
<identifiers-list> --> , <variable-identifier>
                <identifiers-list> |
                <empty>
```

```
<attributes-list> --> <attribute> <attributes-list>
                |
                <empty>
 16.  <attribute> --> SIGNAL     |
                COMPLEX     |
                INTEGER     |
                FLOAT       |
                BLOCKFLOAT  |
                EXT         |
                [<range><ranges-list>]
<ranges-list> --> ,<range> <ranges-list> |
                <empty>
<range> --> <unsigned-integer> .. <unsigned-integer>
<math-function-declarations> --> DEFFUNC <function-list>
                |
                <empty>
<function-list> --> <function> <function-list> |
                <empty>
<function> --> <function-identifier> =
                <expression><function-characteristic> ;
<function-characteristic> --> \ <unsigned-integer> ,
                <unsigned-integer>
<procedure-declarations> --> <procedure> <procedure-
                declarations> |
                <empty>
<procedure> --> PROCEDURE <procedure-
                identifier><parameters-list> ;
<parameters-list> --> ( <declarations-list> ) |
                <empty>
<statements-list> --> <statement> <statements-list>
                |
                <empty>
<statement> --> <unsigned-integer> : <statement> |
                <variable> := <expression> ; | <procedure-
                identifier><actual-arguments> ;
                |
                <condition-statement> ENDIF ; |
                WHILE <conditional-expression> DO
                <statements-list> ENDWHILE ; |
                LOOP <statements-list> ENDLOOP ; |
                FOR <variable-identifier> :=
                <loop-declaration> ENDFOR ; |
                CASE <expression> OF <alternatives-
                list> ENDCASE ; |
                GOTO <unsigned-integer> ; |
                LINK <variable-identifier> , <unsigned-
                integer> ; |
```

```
                IN <unsigned-integer>;  |
                OUT <unsigned-integer>; |
                RETURN ; |
                |
                ($ <assembly-insert-file-identifier> $)
   <condition-statement> --> <incomplete-condition-
                     statement><alternative-part>
<incomplete-condition-statement> --> IF <conditional-
                expression> THEN <statements-list>
<alternative-part> --> ELSE <statements-list> |
                <empty>
<loop-declaration> --> <expression> TO <expression> DO
                <statements-list>
<actual-arguments> --> ( <expression> <actual-
                arguments-list> ) |
                <empty>
<actual-arguments-list> --> ,<expression> <actual-
                arguments-list> |
                <empty>
<alternatives-list> --> <alternative> <alternatives-list>
                |
                <empty>
<alternative> --> <expression> : / <statements-list>
                \
<conditional-expression> --> <logical-summand>
                <logical>
<logical> --> OR <logical-summand> <logical> |
                <empty>
<logical-summand> --> <logical-multiplier> <logical-
                multipliers-list>
<logical-multipliers-list> --> AND <logical-multiplier>
                <logical-multipliers-list> | <empty>
<logical-multiplier> --> <expression><comparison-
                operator><expression> |
                [ <conditional-expression> ] |
                NOT <logical-multiplier>
<comparison-operator> --> <  |
                <=   |
                =    |
                <>   |
                >=   |
                >
 42.   <expression> --> <summand> <summands-list>   |
                - <summand> <summands-list>
```

```
<summands-list> --> <add-instruction> <summand>
                <summands-list> |
                <empty>
44.   <add-instruction> -->     +   |
                -   |
                !
<summand> --> <multiplier><multipliers-list>
<multipliers-list> --> <multiplication-instruction>
                <multiplier><multipliers-list> | <empty>
47.   <multiplication-instruction> -->     *   |
                /   |
                & |
                MOD
<multiplier> --> <unsigned-constant>  |
                <complex-constant> | <constant-
                identifier> | <variable> | <function-
                identifier> | <builtin-function-
                identifier><actual-arguments> |


                ( <expression> )  |
                - <multiplier>    |
                ^ <multiplier>
<variable> --> <variable-identifier><dimension> |
                <complex-variable>
<complex-variable> --> "<complex-number>"
<dimension> --> [ <expression> <expressions-list> ]
                |
                <empty>
<expressions-list> --> ,<expression><expressions-list> |
                <empty>
```

Лексична частина граматики

```
<complex-constant> --> '<complex-number>'
<unsigned-constant> --> <unsigned-number> |
                <unsigned-integer>
<complex-number> --> <left-part> <right-part>
<left-part> --> <expression> |
                <empty>
<right-part> --> ,<expression> |
                $EXP( <expression> )   |
                <empty>
<constant-identifier> --> <identifier>
<variable-identifier> --> <identifier>
```

```
<procedure-identifier> --> <identifier>
<function-identifier>  --> <identifier>
<builtin-function-identifier> --> <identifier>
<assembly-insert-file-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
                <digit><string> |
                <empty>
<unsigned-number> --> <integer-part><fractional-part>
<integer-part> --> <unsigned-integer> |
                <empty>
<fractional-part> --> #<sign><unsigned-integer>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
                <empty>
71.   <sign> --> +      |
                0      |
                <empty>
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> --> A | B | C | D | ... | Z
```

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

Основна література

Ахо А., Сети Р., Ульман Д. Компиляторы: принципы, технологии и инструменты.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003, – 768 с. : ил.

Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. В 2-х томах. – М.: Мир, 1978. – 1105 с.

Грисс Д. Конструирование компиляторов для цифровых вычислительных машин. – М.: Мир, 1975. – 544 с.

Д.Э.Кнут. Нисходящий синтаксический анализ. В кн.: Кибернетический сборник. Выпуск 15. – М.: Мир, 1978. – с.101-142.

Бек Л. Введение в системное программирование. – М.: Мир, 1988. – 448 с.

Опалева Э.А., Самойленко В.П. Языки программирования и методы трансляции. – СПб: БХВ-Петербург, 2005. – 476 с.

Марченко А.И. Средства автоматизации программирования процессоров цифровой обработки сигналов Диссертация на соискание ученой степени кандидата технических наук. Кафедра вычисли-тельной техники, Киев, КПИ, 1993, 129 с.

Додаткова література

Компаниец Р.И., Маньков Е.В., Филатов Н.Е. Системное программирование. Основы построения трансляторов./Учебное

пособие для высших и средних учебных заведений – СПб.: КОРОНА принт, 2000. – 256 с.

Льюис Ф., Розенкранц Д., Стирнз Р. Теоретические основы проектирования компиляторов. – М.: Мир, 1979. – 656 с.

Касьянов В.Н., Поттосин И.В. Методы построения трансляторов. – Новосибирск: Наука, 1986. – 344 с.

11. Глушков В.М., Цейтлин Г.М., Ющенко Е.Л. Алгебра, языки, программирование. – К.: Наукова думка, 1978. – 320 с.

Янг С. Алгоритмические языки реального времени. Конструирование и разработка. – М.: Мир, 1985. – 400 с.

Ингерман П. Синтаксически ориентированный транслятор. – М.: Мир, 1969. – 176 с.

Маккиман У., Хорнинг Дж., Уортман Д. Генератор компиляторов. – М.: Статистика, 1980. – 528 с.

Языки и автоматы. Сборник переводов статей. – М.: Мир, 1975. – 360с.

Кибернетический сборник. Выпуск 15. – М.: Мир, 1978. – 224 с.