

# Laporan Milestone I TBFO

## Lexical Analysis

### ParsingIsAllYouNeed



Muhammad Raihan Nazhim Oktana

13523021

[13523021@std.stei.itb.ac.id](mailto:13523021@std.stei.itb.ac.id)

Shanice Feodora Tjahjono

13523097

[13523097@std.stei.itb.ac.id](mailto:13523097@std.stei.itb.ac.id)

Faqih Muhammad Syuhada

13523057

[13523057@std.stei.itb.ac.id](mailto:13523057@std.stei.itb.ac.id)

Muhammad Fathur Rizky

13523105

[13523105@std.stei.itb.ac.id](mailto:13523105@std.stei.itb.ac.id)

**Program Studi Teknik Informatika  
Sekolah Teknik Elektro dan Informatika  
Institut Teknologi Bandung  
2025**

## Daftar Isi

<b>BAB I: Deskripsi Tugas</b>	<b>2</b>
<b>BAB II: Landasan Teori</b>	<b>3</b>
2.1 Deterministic Finite Automata	3
2.2 Lexer	3
2.3 Token	3
<b>BAB III: Perancangan &amp; Implementasi</b>	<b>4</b>
3.1 Perancangan	4
3.2 Implementasi	4
<b>BAB IV: Pengujian</b>	<b>10</b>
4.1 Pengujian Test Cases	10
<b>BAB V: Kesimpulan dan Saran</b>	<b>19</b>
5.1 Kesimpulan	19
5.2 Saran	19
<b>Lampiran</b>	<b>19</b>
<b>Daftar Pustaka</b>	<b>19</b>

## **BAB I: Deskripsi Tugas**

Pada era modern, bahasa pemrograman menjadi pondasi utama dalam pengembangan perangkat lunak dan sistem komputasi. Proses penerjemahan kode sumber dari bahasa pemrograman ke bentuk yang dapat dimengerti oleh mesin dilakukan oleh compiler. Dalam Tugas Besar Mata Kuliah (MK) Teori Bahasa Formal dan Otomata (IF2224-24) bagian milestone 1 ini, mahasiswa ditugaskan untuk mengimplementasikan bagian paling awal dari proses kompilasi tersebut, yaitu melakukan lexical analysis atau analisis leksikal, pada proyek pengembangan Pascal-S Compiler.

Bahasa Pascal-S merupakan subset dari bahasa pemrograman Pascal yang disederhanakan dan dirancang untuk tujuan pembelajaran konsep compiler construction. Dalam tugas besar ini, mahasiswa berperan sebagai tim pengembang yang akan membangun compiler Pascal-S. Pada tahap awal, milestone 1 akan berfokus pada pembangunan lexer (lexical analyzer) yang bertugas mengidentifikasi dan mengelompokkan setiap rangkaian karakter dari kode sumber Pascal-S menjadi unit-unit bermakna yang disebut token.

Analisis leksikal merupakan tahap pertama dalam proses kompilasi. Pada tahap ini, lexer membaca kode sumber dari kiri ke kanan dan menggunakan pola yang didefinisikan oleh Deterministic Finite Automata (DFA) untuk mengenali berbagai jenis token seperti keyword, identifier, operator, literal, dan simbol lain yang valid dalam bahasa Pascal-S. Proses ini menghasilkan daftar token yang kemudian akan menjadi masukan bagi tahap parsing pada Milestone berikutnya.

## BAB II: Landasan Teori

### 2.1 Deterministic Finite Automata

Deterministic Finite Automata (DFA) adalah model komputasi formal yang digunakan untuk mengenali bahasa reguler. Sebuah DFA didefinisikan sebagai 5-tuple  $M = (Q, \Sigma, \delta, q_0, F)$  dengan  $Q$  himpunan keadaan hingga,  $\Sigma$  alfabet input,  $\delta: Q \times \Sigma \rightarrow Q$  fungsi transisi deterministik,  $q_0 \in Q$  keadaan awal, dan  $F \subseteq Q$  himpunan keadaan akseptor. Dalam konstruksi pemindai leksikal, pola leksikal yang mula-mula dirumuskan sebagai ekspresi reguler dikonversi menjadi NFA, lalu diubah menjadi DFA melalui subset construction dan dapat diminimalkan untuk efisiensi. Pada konteks Pascal-S, DFA dipakai untuk mengenali kelas token seperti identifier, literal bilangan bulat/riil, operator relasional ( $=/ </ <=, >, >=$ ), operator penugasan ( $:=$ ), serta delimiter (titik, koma, titik koma, tanda kurung). Ambiguitas lokal—misalnya pembedaan ":" vs ":" dan ":" vs ":"—ditangani tetap secara deterministik melalui desain keadaan dan kebijakan pencocokan terpanjang (longest match) sehingga tidak mengorbankan determinisme pemindaian [1][2].

### 2.2 Lexer

Lexer (analisis leksikal). Lexer merupakan tahap awal front-end kompilasi yang mengubah aliran karakter sumber menjadi aliran token. Tanggung jawab utamanya meliputi: mengabaikan whitespace dan komentar, menerapkan strategi maximal-munch dengan prioritas kelas token yang konsisten, melakukan lookahead terbatas untuk operator majemuk (misalnya " $<=$ ", " $<$ ", " $:=$ ", "..."), melaporkan galat leksikal dengan koordinat baris-kolom yang presisi, serta mengisi atribut token (misalnya nilai literal numerik atau tautan ke entri tabel simbol untuk identifier). Kompleksitas waktu pemindaian bersifat linear terhadap panjang input karena setiap karakter dikonsumsi paling banyak sekali. Untuk Pascal-S, praktik lazim adalah case-insensitive pada identifier dan kata kunci; komentar umumnya didukung dalam bentuk {...} dan/atau (\* ... \*); disambiguasi "." (akhir unit program) dan ".." (operator rentang) mengikuti longest match; dan literal riil didefinisikan dengan titik desimal serta, bila diaktifkan, eksponen. Selain itu, kata kunci biasanya diidentifikasi melalui tabel reserved words setelah lexeme terlebih dahulu dikenali oleh pola identifier, sehingga pemeliharaan daftar kata kunci tetap terpisah dari mesin pengenal dasar [2][3][4][5].

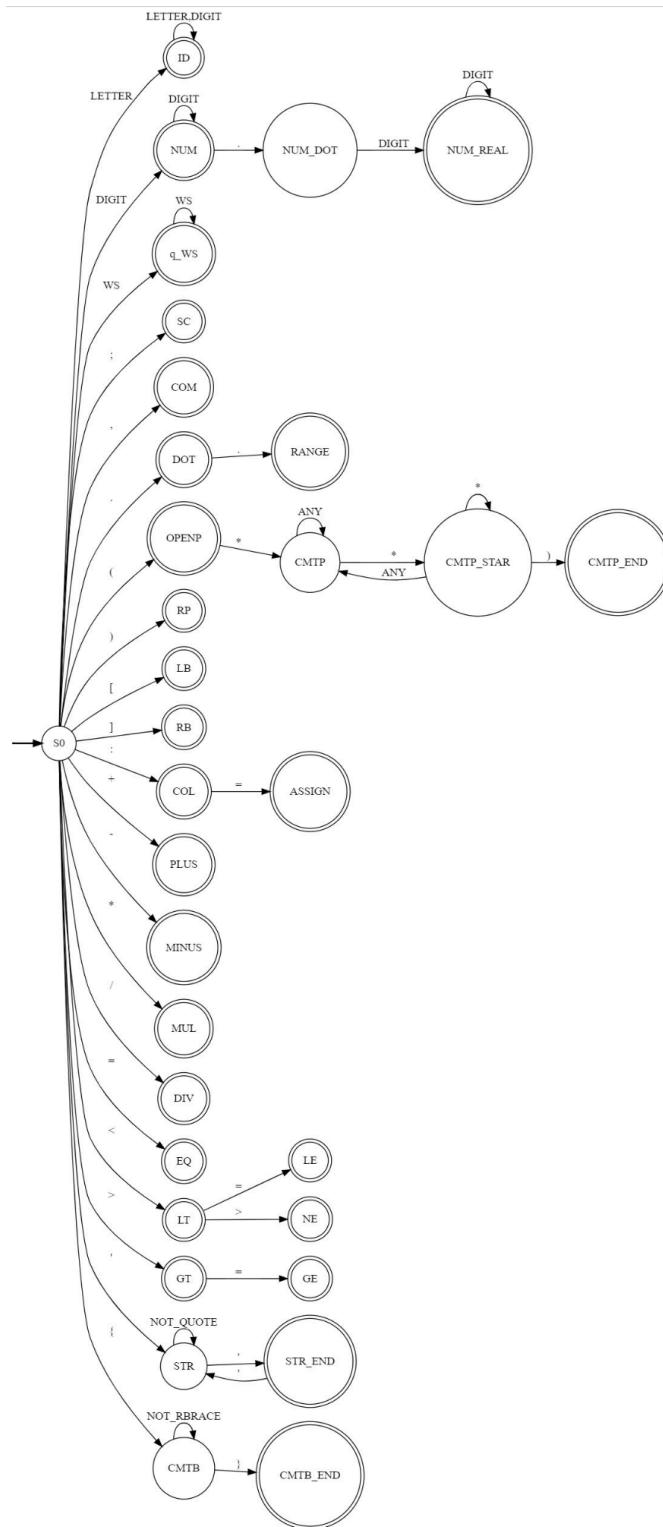
### 2.3 Token

Token adalah unit leksikal yang menjadi antarmuka antara lexer dan parser, umumnya direpresentasikan sebagai pasangan  $\langle$ jenis, atribut $\rangle$ . Jenis (type) token menyatakan kategori leksikal—contoh umum: ID, INT\_LIT, REAL\_LIT, STRING\_LIT, KEYWORD, OP, DELIM—sedangkan atribut menyimpan informasi yang relevan untuk tahap sintaksis atau semantik, seperti lexeme, nilai konversi numerik, atau indeks ke tabel simbol. Dalam ekosistem Pascal dan variannya, himpunan kata kunci meliputi, antara lain, program, var, const, begin, end, if, then, else, while, do, for, to, downto, repeat, until, case, of, serta tipe dasar seperti integer, real, boolean, char; operator aritmetika +, -, \*, /; operator relasional =, <, <=, >, >=; operator logika and, or, not; operator penugasan :=; serta delimiter ;, :, ( ) [ ] ... . Implementasi Pascal-S yang bersifat subset mempertahankan kompatibilitas semantik dengan standar Pascal sembari menyederhanakan repertoar token untuk keperluan pendidikan dan referensi.

Penanganan galat leksikal—misalnya karakter tak dikenal, string literal tidak tertutup (tanda petik tunggal), atau komentar tidak tertutup—wajib menghasilkan diagnostik yang jelas dan dapat direproduksi agar memudahkan penelusuran kesalahan pada tahap berikutnya [2][4][5].

## BAB III: Perancangan & Implementasi

### 3.1 Perancangan



Gambar 1. Diagram DFA untuk Pascal-S Compiler

Diagram deterministik hingga (DFA) yang ditunjukkan merepresentasikan tahap perancangan leksikal dalam proses kompilasi bahasa Pascal, khususnya pada subsistem lexical analyzer (scanner). Tujuan utama perancangan ini adalah mengidentifikasi setiap token valid dalam bahasa Pascal berdasarkan urutan karakter masukan. Setiap simpul (state) dalam diagram merepresentasikan kondisi internal dari automaton, sementara setiap sisi berlabel merepresentasikan simbol input yang memicu transisi antar state.

Automaton dimulai dari state awal S0, yang berfungsi sebagai titik masuk universal bagi semua kategori token. Transisi dari S0 bersifat deterministik terhadap simbol pertama yang dibaca. Apabila karakter awal merupakan huruf alfabet, automaton berpindah ke state ID, yang menandai proses pembentukan identifier atau keyword. Di dalam state ini, automaton dapat menerima karakter huruf atau digit secara berulang hingga menemui karakter bukan alfanumerik. Pada tahap pasca-akseptasi, hasil tokenisasi dari ID akan diverifikasi terhadap tabel kata kunci (keyword table) untuk menentukan apakah token tersebut merupakan identifier biasa atau kata kunci terdefinisi seperti program, var, atau begin.

Apabila simbol pertama berupa digit, automaton berpindah menuju state NUM untuk memproses konstanta numerik. Transisi dari NUM ke NUM\_DOT diaktifkan oleh simbol titik ("."), yang menandakan potensi terbentuknya bilangan real. Setelah titik, automaton memasuki state NUM\_REAL jika menerima digit berikutnya, dan melanjutkan pembacaan digit secara berulang hingga akhir token. Dengan demikian, NUM dan NUM\_REAL menjadi dua state akhir berbeda untuk membedakan bilangan bulat dan bilangan pecahan.

Selain itu, diagram mencakup beberapa cabang khusus yang berfungsi mendeteksi operator, tanda baca, dan simbol pembatas (delimiters). Misalnya, state COL menangani tanda titik dua ("."), yang dapat diikuti oleh tanda sama dengan ("=") untuk membentuk operator penugasan (":=") pada state ASSIGN. Pola serupa berlaku untuk operator relasional: tanda "<" diikuti "=" menghasilkan state LE (less than or equal), sedangkan diikuti ">" menghasilkan state NE (not equal).

Token string literal dimodelkan melalui state STR, yang diaktifkan oleh tanda kutip tunggal (""). Di dalam state ini, automaton membaca setiap karakter non-kutip (NOT\_QUOTE) hingga menemui tanda kutip penutup yang mengalihkan automaton ke state STR\_END sebagai kondisi akseptasi.

Dua jenis komentar Pascal dimodelkan secara terpisah. Komentar bertanda kurung kurawal ("{...}") direpresentasikan melalui state CMTB, yang terus menerima semua karakter selain tanda penutup "}". Sedangkan komentar bertanda kurung dan asterisk ("\* ... ") dimodelkan melalui rangkaian state OPENP → CMTP → CMTP\_STAR → CMTP\_END, yang memastikan komentar hanya berakhir dengan urutan "}). Struktur ini menjamin determinisme dengan menghindari ambiguitas pada simbol "\*" yang dapat berfungsi ganda sebagai operator atau bagian dari komentar.

Seluruh state dengan lingkaran ganda pada diagram merupakan state akseptor, yang menandai akhir pembacaan token valid. DFA ini dirancang agar setiap urutan karakter dari bahasa Pascal menghasilkan lintasan deterministik tunggal menuju salah satu state akseptor tanpa

backtracking, sehingga efisien untuk diimplementasikan dalam scanner berbasis tabel transisi. Dengan demikian, diagram ini menjadi representasi formal yang menggabungkan prinsip regular language recognition dan deterministic finite-state computation untuk tahap analisis leksikal dalam perancangan kompiler Pascal.

### 3.2 Implementasi

```
# abstract.py
from abc import ABC, abstractmethod
from typing import Generic, TypeVar, Hashable, Optional, Set, AbstractSet,
Iterable

StateT = TypeVar("StateT", bound=Hashable)
SymbolT = TypeVar("SymbolT", bound=Hashable)

class AutomatonABC(ABC, Generic[StateT, SymbolT]):
    def __init__(self) -> None:
        self._start: Optional[StateT] = None
        self._finals: Set[StateT] = set()

    # konfigurasi
    def set_start(self, state: StateT) -> None: self._start = state
    def add_final(self, state: StateT) -> None: self._finals.add(state)
    def is_final(self, state: StateT) -> bool: return state in self._finals

    @property
    def start_state(self) -> Optional[StateT]: return self._start
    @property
    def final_states(self) -> AbstractSet[StateT]: return
frozenset(self._finals)

    # runtime
    @abstractmethod
    def reset(self) -> None: ...
    @abstractmethod
    def step_all(self, states: AbstractSet[StateT], symbol:
Optional[SymbolT]) -> Set[StateT]: ...
    @abstractmethod
    def accepts(self, symbols: Iterable[SymbolT]) -> bool: ...
```

Abstraksi AutomatonABC memformalkan mesin hingga sebagai entitas generik (parametris pada tipe keadaan dan simbol), dengan antarmuka konfigurasi (start, final) dan antarmuka eksekusi (reset, step\_all, accepts). Secara teoretik, struktur ini memetakan 5-tuple  $M = (Q, \Sigma, \delta, q_0, F)$ : set\_start dan add\_final merealisasikan  $q_0$  dan  $F$ ; is\_final menguji keanggotaan  $F$ ; sedangkan metode abstrak menerima implementasi  $\delta$  di kelas turunan. Pemisahan antarmuka/implementasi ini mendukung substitutability antara variasi automata (misal NFA/DFA) dan memfasilitasi pembuktian sifat-sifat korektitas pada level kontrak kelas.

```
@dataclass
class DFAConfig:
    start_state: str
    final_states: Dict[str, str]
    char_classes: Dict[str, str]
```

```

transitions: List[Tuple[str, str, str]]
keywords: List[str]
reserved_map: Dict[str, str]

class DFA(AutomatonABC[str, str]):
    """DFA lexical analysis."""

    def __init__(self, config: DFAConfig):
        super().__init__()

        self.config = config
        self.set_start(config.start_state)
        for final_state in config.final_states:
            self.add_final(final_state)

        self.current_state: Optional[str] = self.start_state

        self.char_class_patterns: Dict[str, re.Pattern] = {
            name: re.compile(pattern)
            for name, pattern in config.char_classes.items()
        }

        self.transitions: Dict[Tuple[str, str], str] = {
            (from_state, input_sym): to_state
            for from_state, input_sym, to_state in config.transitions
        }
    ...

```

Pendekatan konfigurasi pada pemindai leksikal Pascal-S dirancang agar spesifikasi bahasa dinyatakan sebagai data, sementara mesin pengenal (DFA) direalisasikan sebagai kode generik. Struktur DFAConfig berperan sebagai kontrak formal antara spesifikasi dan mesin, sehingga perubahan kebijakan leksikal—misalnya penambahan operator majemuk atau variasi komentar—dapat dilakukan tanpa memodifikasi algoritme inti. Secara teoretis, pemetaan ini mengikuti model DFA  $M = (Q, \Sigma, \delta, q_0, F)$  serta praktik maximal-munch dalam pemindaian deterministik dengan kompleksitas waktu linear terhadap panjang masukan.

Komponen start\_state: str merepresentasikan keadaan awal  $q_0$  secara eksplisit. Satu nilai string memudahkan serialisasi (berkas JSON) dan menjamin setiap proses pembentukan lexeme baru selalu berawal dari titik yang sama dan terdefinisi. Sejalan dengan itu, final\_states: Dict[str, str] memuat himpunan keadaan akseptor  $F$  sekaligus anotasi yang memetakan tiap keadaan final ke jenis token. Bentuk kamus dipilih agar keputusan jenis token berlangsung satu langkah; misalnya dua keadaan final berbeda (NUM, NUM\_REAL) dapat dipromosikan ke kategori yang sama (NUMBER), atau sebaliknya dipetakan ke kategori yang berbeda (ASSIGN\_OPERATOR untuk “:=” dibanding COLON untuk “.”).

Agar spesifikasi ringkas tanpa kehilangan determinisme, char\_classes: Dict[str, str] mendefinisikan kelas karakter satu-langkah (misalnya LETTER, DIGIT, WS) melalui pola setara kelas karakter regex. Dengan cara ini, transisi yang secara semantik ekuivalen—seperti seluruh huruf atau seluruh digit—tidak perlu ditulis satu per satu. Batasan “satu karakter per langkah” menjaga kesesuaian dengan fungsi transisi delta:  $Q \times \Sigma \rightarrow Q$ , sementara disjungsi antarkelas yang rapi mencegah tumpang tindih yang dapat merusak determinisme.

Seluruh tepi delta diserialisasi pada transitions: List[Tuple[str, str, str]] sebagai daftar (src, sym, dst). Representasi daftar dipilih karena kompatibel dengan JSON (yang tidak memiliki kunci bertipe tuple) dan mudah ditinjau pada version control. Saat pemutuan, daftar ini dikompilasi menjadi struktur pencarian cepat dan divalidasi (misalnya memastikan tidak ada dua tepi aktif untuk pasangan keadaan-simbol yang sama). Di tingkat eksekusi, mesin memberi prioritas pada transisi simbol literal (misalnya “.”, “..”, “<”) sebelum transisi berbasis kelas karakter (LETTER, DIGIT). Kebijakan prioritas ini krusial untuk menangani operator majemuk dan ambiguitas lokal di Pascal-S, seperti perbedaan “.” vs “:=”, “.” vs “..”, atau “<” vs “<=>”, sekaligus menegakkan maximal-munch.

Pascal dan variannya lazim menerapkan perbedaan pasca-pencocokan (post-match) antara identifier dan kata kunci. Oleh karena itu, keywords: List[str] memuat daftar kata kunci yang diperiksa setelah suatu lexeme terlebih dahulu dikenali sebagai IDENTIFIER. Strategi ini menjaga DFA tetap kecil (tanpa keadaan khusus per kata kunci) dan memudahkan pemeliharaan daftar secara terpisah. Praktik ini juga selaras dengan kebijakan case-insensitive yang umum pada Pascal: pemeriksaan dilakukan pada bentuk ternormalisasi (misalnya huruf kecil).

Terakhir, reserved\_map: Dict[str, str] memetakan lexeme berbasis kata langsung ke kategori token yang lebih spesifik daripada sekadar KEYWORD. Contoh khas Pascal-S ialah operator berbasis kata: “div” dan “mod” (operator aritmetika), serta “and”, “or”, “not” (operator logika). Dengan peta ini, parser menerima token yang sudah tepat kategorinya (setara dengan “+” atau “\*\*”) tanpa aturan promosi tambahan. Dalam proses pasca-pencocokan, reserved\_map diberi prioritas lebih tinggi daripada keywords agar lexeme seperti “div” tidak dilabeli KEYWORD generik, melainkan langsung sebagai ARITHMETIC\_OPERATOR. Kombinasi keywords dan reserved\_map ini memisahkan kebijakan bahasa dari mesin, menyederhanakan pipeline front-end, dan meningkatkan keterjagaan desain.

```
class DFA(AutomatonABC[str, str]):
    """DFA lexical analysis."""

    def __init__(self, config: DFAConfig):
        ...

    def reset(self) -> None:
        """Reset automaton to start state."""
        self.current_state = self.start_state

    def step(self, char: str) -> Optional[str]:
        key = (self.current_state, char)
        if key in self.transitions:
            self.current_state = self.transitions[key]
            return self.current_state

        for (state, input_sym), next_state in self.transitions.items():
            if state == self.current_state and input_sym in
                self.char_class_patterns:
                if self.char_class_patterns[input_sym].match(char):
                    self.current_state = next_state
```

```

        return self.current_state

    return None

def step_all(self, states: Set[str], symbol: Optional[str]) -> Set[str]:
    if not symbol:
        return set()

    current_state_before = self.current_state
    self.current_state = next(iter(states))

    next_s = self.step(symbol)

    self.current_state = current_state_before

    return {next_s} if next_s else set()

def accepts(self, symbols: List[str]) -> bool:
    self.reset()
    for symbol in symbols:
        if self.step(symbol) is None:
            return False
    return self.is_final(self.current_state) if self.current_state else
False

def get_token_type(self) -> Optional[str]:
    return self.config.final_states.get(self.current_state)

def can_transition(self, char: str) -> bool:
    key = (self.current_state, char)
    if key in self.transitions:
        return True

    return any(
        state == self.current_state
        and input_sym in self.char_class_patterns
        and self.char_class_patterns[input_sym].match(char)
        for state, input_sym in self.transitions.keys()
    )

```

Realisasi DFA mengompilasi dua bentuk transisi: (i) transisi spesifik karakter (key persis (state, char)) untuk operator/delimiter satu karakter; dan (ii) transisi berbasis kelas karakter melalui regex prakomplikasi (char\_classes) untuk kelompok seperti LETTER, DIGIT, WS. Fungsi step terlebih dahulu menguji transisi persis, lalu mengevaluasi kecocokan kelas karakter—strategi ini konsisten dengan priority rule agar operator majemuk tidak “terserap” oleh kelas karakter umum. Metode can\_transition digunakan sebagai lookahead predicate untuk memutuskan apakah lexeme masih dapat diperluas; get\_token\_type memetakan keadaan final ke jenis token yang telah diputuskan oleh konfigurasi. Secara formal, determinisme dijaga karena dari setiap keadaan dan simbol input, paling banyak ada satu transisi yang diambil—entah melalui tepi eksplisit atau melalui satu kelas karakter yang saling lepas dalam definisi konfigurasi.

```
class Lexer:
```

```

def __init__(self, dfa_config_path: str = "config/dfa_rules.json"):
    """Init lexer"""
    self.config = self._load_config(dfa_config_path)
    self.dfa = DFA(self.config)
    self.keywords = set(self.config.keywords)
    self.reserved_map = self.config.reserved_map

def _load_config(self, path: str) -> DFAConfig:
    """Load DFAConfig"""
    try:
        with open(path, "r", encoding="utf-8") as f:
            data = json.load(f)
    except FileNotFoundError:
        raise FileNotFoundError(
            f"Config file not found at '{path}'"
        )

    return DFAConfig(
        start_state=data["start_state"],
        final_states=data["final_states"],
        char_classes=data["char_classes"],
        transitions=[tuple(t) for t in data["transitions"]],
        keywords=data["keywords"],
        reserved_map=data["reserved_map"],
    )

def tokenize(self, source_code: str) -> list[Token]:
    reader = Reader(source_code)
    tokens = []
    while not reader.eof():
        start_pos = reader.pos.index
        lexeme = ""
        self.dfa.reset()
        last_accepted_state = None
        accepted_lexeme = ""
        accepted_pos = None # Track position after accepted lexeme

        while not reader.eof() and
self.dfa.can_transition(reader.current_char):
            lexeme += reader.current_char
            self.dfa.step(reader.current_char)
            reader.advance()
            if self.dfa.get_token_type():
                last_accepted_state = self.dfa.current_state
                accepted_lexeme = lexeme
                accepted_pos = reader.pos.index # Save position after
this character

            if accepted_lexeme:
                token_type_str = self.config.final_states.get(
                    last_accepted_state)
                token_type = TokenType[token_type_str]

                if token_type == TokenType.IDENTIFIER and
accepted_lexeme.lower() in self.reserved_map:
                    token_type =

```

```

TokenType[self.reserved_map[accepted_lexeme.lower()]]
    elif token_type == TokenType.IDENTIFIER and
accepted_lexeme.lower() in self.keywords:
        token_type = TokenType.KEYWORD

    tokens.append(
        Token(
            type=token_type,
            value=accepted_lexeme,
            start=start_pos,
            end=start_pos + len(accepted_lexeme),
        )
    )
    # Position reader right after the accepted lexeme (only if
we_overshot)
    if reader.pos.index != accepted_pos:
        reader.set_position(accepted_pos)

elif not reader.eof():
    # Handle unknown characters
    tokens.append(
        Token(
            type=TokenType.UNKNOWN,
            value=reader.current_char,
            start=reader.pos.index,
            end=reader.pos.index + 1,
        )
    )
    reader.advance()

# Post-process to merge unary minus with numbers
tokens = merge_negative_numbers(tokens)
# Post-process to distinguish char literals from string literals
tokens = fix_char_literals(tokens)
return tokens

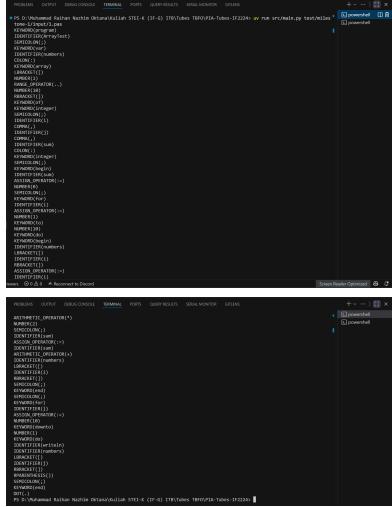
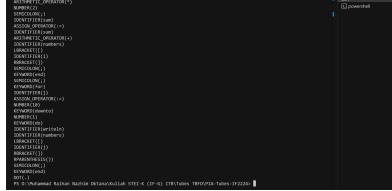
```

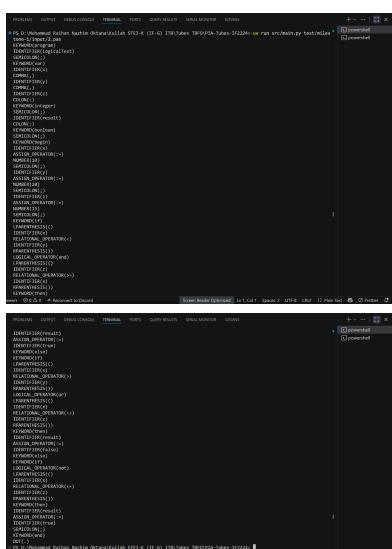
Algoritma pemindaian menerapkan prinsip maximal-munch dengan penanda “penerimaan terakhir” (accepted\_lexeme, last\_accepted\_state). Pada setiap iterasi, DFA di-reset, lexeme diperluas selama can\_transition bernilai benar, dan setiap kali mencapai keadaan final, checkpoint penerimaan diperbarui. Setelah perluasan berhenti, jika ada lexeme yang diterima, jenis token diperoleh dari peta final\_states; kemudian dilakukan promosi pasca-pencocokan: reserved\_map memiliki prioritas lebih tinggi daripada keywords untuk menangani operator berbasis kata (div, mod, and, or, not) serta kategori khusus lain yang tidak ingin disatukan sebagai KEYWORD. Mekanisme fallback UNKNOWN memastikan karakter ilegal tetap dikonsumsi satu per satu agar pemindaian maju (progress guarantee). Kompleksitas waktu bersifat O(n) terhadap panjang masukan dengan implementasi dua-penunjuk; pada cuplikan ini, reset Reader setelah penerimaan bersifat langsung namun dapat dioptimalkan dengan menyimpan indeks “right boundary” tanpa membuat kembali Reader. Secara teoretik, pendekatan ini ekivalen dengan komposisi DFA token-spesifik yang dijalankan serentak (melalui product construction) di mana longest-match dan prioritas kelas token dipaksakan pada tie-break—praktik baku dalam desain pemindai leksikal.

## BAB IV: Pengujian

Dengan adanya data ini, sistem ATS dapat mengadopsi pendekatan adaptif: menggunakan Boyer-Moore untuk pencarian tunggal cepat, dan Aho-Corasick ketika jumlah kata kunci melebihi satu. Pendekatan berbasis performa ini mendukung efisiensi sistem secara ke

### 4.1 Pengujian Test Cases

Input	Output	Bukti & Keterangan
program ArrayTest; var numbers: array[1..10] of integer; i, j, sum: integer; begin sum := 0; for i := 1 to 10 do begin numbers[i] := i * 2; sum := sum + numbers[i]; end; for j := 10 downto 1 do writeln(numbers[j]); end.	KEYWORD(program) IDENTIFIER(ArrayTest) SEMICOLON(); KEYWORD(var) IDENTIFIER(numbers) COLON(:) KEYWORD(array) LBRACKET([]) NUMBER(1) RANGE_OPERATOR(..) NUMBER(10) RBRACKET() KEYWORD(of) KEYWORD(integer) SEMICOLON(); IDENTIFIER(i) COMMA(), IDENTIFIER(j) COMMA(), IDENTIFIER(sum) COLON(:) KEYWORD(integer) SEMICOLON(); KEYWORD(begin) IDENTIFIER(sum) ASSIGN_OPERATOR(:=) NUMBER(0) SEMICOLON(); KEYWORD(for) IDENTIFIER(i) ASSIGN_OPERATOR(:=) NUMBER(1) KEYWORD(to) NUMBER(10) KEYWORD(do) KEYWORD(begin) IDENTIFIER(numbers) LBRACKET([]) IDENTIFIER(i) RBRACKET() ASSIGN_OPERATOR(:=) IDENTIFIER(i) ARITHMETIC_OPERATOR(*) NUMBER(2) SEMICOLON();	Menguji kemampuan lexer dalam menangani struktur data array, range operator(..), dan nested loops dengan berbagai operator aritmatika.  Hasil uji sudah benar.   

	<pre> IDENTIFIER(sum) ASSIGN_OPERATOR(:=) IDENTIFIER(sum) ARITHMETIC_OPERATOR(+) IDENTIFIER(numbers) LBRACKET([) IDENTIFIER(i) RBRACKET()]) SEMICOLON(;) KEYWORD(end) SEMICOLON(;) KEYWORD(for) IDENTIFIER(j) ASSIGN_OPERATOR(:=) NUMBER(10) KEYWORD(downto) NUMBER(1) KEYWORD(do) IDENTIFIER(writeln) LPARENTHESIS(()) IDENTIFIER(numbers) LBRACKET([) IDENTIFIER(j) RBRACKET()]) RPARENTHESIS()) SEMICOLON(;) KEYWORD(end) DOT(.) </pre>	
<pre> program LogicalTest; var   x, y, z: integer;   result: boolean; begin   x := 10;   y := 20;   z := 15;   if (x &lt; y) and (z &gt;= x) then     result := true   else if (x &gt; y) or (x &lt;&gt; z) then     result := false   else if not (x &lt;= z) then     result := true; end. </pre>	<pre> KEYWORD(program) IDENTIFIER(LogicalTest) SEMICOLON(;) KEYWORD(var) IDENTIFIER(x) COMMA(), IDENTIFIER(y) COMMA(), IDENTIFIER(z) COLON(:) KEYWORD(integer) SEMICOLON(;) IDENTIFIER(result) COLON(:) KEYWORD(boolean) SEMICOLON(;) KEYWORD(begin) IDENTIFIER(x) ASSIGN_OPERATOR(:=) NUMBER(10) SEMICOLON(;) IDENTIFIER(y) ASSIGN_OPERATOR(:=) NUMBER(20) SEMICOLON(;) IDENTIFIER(z) ASSIGN_OPERATOR(:=) </pre> <p>Menguji semua jenis operator relasional (&lt;, &lt;=, &gt;, &gt;=, =, &lt;&gt;) dan operator logika (and, or, not) dalam kondisi kompleks.</p> 	

	NUMBER(15) SEMICOLON(;) KEYWORD(if) LPARENTHESIS() IDENTIFIER(x) RELATIONAL_OPERATOR(<) IDENTIFIER(y) RPARENTHESIS() LOGICAL_OPERATOR(and) LPARENTHESIS() IDENTIFIER(z) RELATIONAL_OPERATOR(>=) IDENTIFIER(x) RPARENTHESIS() KEYWORD(then) IDENTIFIER(result) ASSIGN_OPERATOR(:=) IDENTIFIER(true) KEYWORD(else) KEYWORD(if) LPARENTHESIS() IDENTIFIER(x) RELATIONAL_OPERATOR(>) IDENTIFIER(y) RPARENTHESIS() LOGICAL_OPERATOR(or) LPARENTHESIS() IDENTIFIER(x) RELATIONAL_OPERATOR(<>) IDENTIFIER(z) RPARENTHESIS() KEYWORD(then) IDENTIFIER(result) ASSIGN_OPERATOR(:=) IDENTIFIER(false) KEYWORD(else) KEYWORD(if) LOGICAL_OPERATOR(not) LPARENTHESIS() IDENTIFIER(x) RELATIONAL_OPERATOR(<=) IDENTIFIER(z) RPARENTHESIS() KEYWORD(then) IDENTIFIER(result) ASSIGN_OPERATOR(:=) IDENTIFIER(true) SEMICOLON(;) KEYWORD(end) DOT(.)	
program StringCharTest; var name: char; message: array[1..5] of char;	KEYWORD(program) IDENTIFIER(StringCharTes t) SEMICOLON(;) KEYWORD(var)	Menguji berbagai jenis literal (string, char), kedua format komentar ({ }) dan (* *)), dan karakter spesial dalam string.

```

{ This is a comment with
special chars: :=, <>,
>=
const
    greeting = 'Hello,
World!';
    symbol = '+';
begin
    name := 'A';
    (* Multi-line comment
       with div and mod
operators *)
    writeln('Name: ',
name);
    writeln('Special
chars: {}()[]');
end.

```

```

IDENTIFIER(name)
COLON(:)
KEYWORD(char)
SEMICOLON(;)
IDENTIFIER(message)
COLON(:)
KEYWORD(array)
LBRACKET([])
NUMBER(1)
RANGE_OPERATOR(..)
NUMBER(5)
RBRACKET())
KEYWORD(of)
KEYWORD(char)
SEMICOLON(;)
KEYWORD(const)
IDENTIFIER(greeting)
RELATIONAL_OPERATOR(=)
STRING_LITERAL('Hello,
World!')
SEMICOLON(;)
IDENTIFIER(symbol)
RELATIONAL_OPERATOR(=)
CHAR_LITERAL('+')
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(name)
ASSIGN_OPERATOR(:=)
CHAR_LITERAL('A')
SEMICOLON(;)
IDENTIFIERwriteln)
LPARENTHESIS()
STRING_LITERAL('Name: ')
COMMA(,)
IDENTIFIER(name)
RPARENTHESIS()
SEMICOLON(;)
IDENTIFIERwriteln)
LPARENTHESIS()
STRING_LITERAL('Special
chars: {}()[]')
RPARENTHESIS()
SEMICOLON(;)
KEYWORD(end)
DOT(.)

```

```

program ProcFuncTest;
type
    MyRange = 1..100;
var
    a, b: MyRange;

procedure swap(x,     y:
integer);
var
    temp: integer;

```

```

KEYWORD(program)
IDENTIFIER(ProcFuncTest)
SEMICOLON(;)
KEYWORD(type)
IDENTIFIER(MyRange)
RELATIONAL_OPERATOR(=)
NUMBER(1)
RANGE_OPERATOR(..)
NUMBER(100)
SEMICOLON(;)

```

Menguji fitur Pascal-S yang lebih advanced seperti procedure, function, dan type definition dengan berbagai kombinasi parameter.

<pre> begin     temp := x;     x := y;     y := temp; end;  function calculate(m, n: integer): integer; begin     calculate := m div n + m mod n; end;  begin     a := 50;     b := calculate(100, 7);     swap(a, b); end. </pre>	<pre> KEYWORD(var) IDENTIFIER(a) COMMA(,) IDENTIFIER(b) COLON(:) IDENTIFIER(MyRange) SEMICOLON(;) KEYWORD(procedure) IDENTIFIER(swap) LPARENTHESIS(()) IDENTIFIER(x) COMMA(,) IDENTIFIER(y) COLON(:) KEYWORD(integer) RPARENTHESIS()) SEMICOLON(;) KEYWORD(var) IDENTIFIER(temp) COLON(:) KEYWORD(integer) SEMICOLON(;) KEYWORD(begin) IDENTIFIER(temp) ASSIGN_OPERATOR(:=) IDENTIFIER(x) SEMICOLON(;) IDENTIFIER(x) ASSIGN_OPERATOR(:=) IDENTIFIER(y) SEMICOLON(;) IDENTIFIER(y) ASSIGN_OPERATOR(:=) IDENTIFIER(temp) SEMICOLON(;) KEYWORD(end) SEMICOLON(;) KEYWORD(function) IDENTIFIER(calculate) LPARENTHESIS(()) IDENTIFIER(m) COMMA(,) IDENTIFIER(n) COLON(:) KEYWORD(integer) RPARENTHESIS()) COLON(:) KEYWORD(integer) SEMICOLON(;) KEYWORD(begin) IDENTIFIER(calculate) ASSIGN_OPERATOR(:=) IDENTIFIER(m) ARITHMETIC_OPERATOR(div) IDENTIFIER(n) ARITHMETIC_OPERATOR(+) </pre>	
--	--	--

	<pre> IDENTIFIER(m) ARITHMETIC_OPERATOR(mod) IDENTIFIER(n) SEMICOLON(;) KEYWORD(end) SEMICOLON(;) KEYWORD(begin) IDENTIFIER(a) ASSIGN_OPERATOR(:=) NUMBER(50) SEMICOLON(;) IDENTIFIER(b) ASSIGN_OPERATOR(:=) IDENTIFIER(calculate) LPARENTHESIS(()) NUMBER(100) COMMA(,) NUMBER(7) RPARENTHESIS()) SEMICOLON(;) IDENTIFIER(swap) LPARENTHESIS(()) IDENTIFIER(a) COMMA(,) IDENTIFIER(b) RPARENTHESIS()) SEMICOLON(;) KEYWORD(end) DOT(.) </pre>	
<pre> program NumberEdgeCase; var     integer1, real1: real;     variable, vari, var123: integer; beginend, ifelse: boolean; begin     integer1 := 42;     real1 := 3.14159;     variable := 0;     var123 := -999;      if integer1 &gt;= real1 then     beginend := true;      vari := integer1 + real1 * 2 - 1;     ifelse := (vari &lt;&gt; 0) and (var123 &lt;= -100); end. </pre>	<pre> KEYWORD(program) IDENTIFIER(NumberEdgeCas e) SEMICOLON(;) KEYWORD(var) IDENTIFIER(integer1) COMMA(,) IDENTIFIER(real1) COLON(:) KEYWORD(real) SEMICOLON(;) IDENTIFIER(variable) COMMA(,) IDENTIFIER(vari) COMMA(,) IDENTIFIER(var123) COLON(:) KEYWORD(integer) SEMICOLON(;) IDENTIFIER(beginend) COMMA(,) IDENTIFIER(ifelse) COLON(:) KEYWORD(boolean) SEMICOLON(;) KEYWORD(begin) </pre> <p>Menguji berbagai format angka (integer, real dengan desimal), identifier yang mirip keyword, dan kombinasi operator assignment yang kompleks.</p>	

	<pre> IDENTIFIER(integer1) ASSIGN_OPERATOR(:=) NUMBER(42) SEMICOLON(;) IDENTIFIER(real1) ASSIGN_OPERATOR(:=) NUMBER(3.14159) SEMICOLON(;) IDENTIFIER(variable) ASSIGN_OPERATOR(:=) NUMBER(0) SEMICOLON(;) IDENTIFIER(var123) ASSIGN_OPERATOR(:=) NUMBER(-999) SEMICOLON(;) KEYWORD(if) IDENTIFIER(integer1) RELATIONAL_OPERATOR(&gt;=) IDENTIFIER(real1) KEYWORD(then) IDENTIFIER(beginend) ASSIGN_OPERATOR(:=) IDENTIFIER(true) SEMICOLON(;) IDENTIFIER(vari) ASSIGN_OPERATOR(:=) IDENTIFIER(integer1) ARITHMETIC_OPERATOR(+) IDENTIFIER(real1) ARITHMETIC_OPERATOR(*) NUMBER(2) ARITHMETIC_OPERATOR(-) NUMBER(1) SEMICOLON(;) IDENTIFIER(ifelse) ASSIGN_OPERATOR(:=) LPARENTHESIS(()) IDENTIFIER(vari) RELATIONAL_OPERATOR(&lt;&gt;) NUMBER(0) RPARENTHESIS(()) LOGICAL_OPERATOR(and) LPARENTHESIS(()) IDENTIFIER(var123) RELATIONAL_OPERATOR(&lt;=) NUMBER(-100) RPARENTHESIS(()) SEMICOLON(;) KEYWORD(end) DOT(.) </pre>	
program NestedBlockTest; var	KEYWORD(program) IDENTIFIER(NestedBlockTe st) SEMICOLON(;)	Menguji nested procedure dan loop while...do yang belum diuji di kelima test

```

        outer1,      outer2:
integer;
flag: boolean;

procedure outer;
var
inner1: integer;

procedure nested;
var
deep: integer;
begin
deep := 5;
outer1 := deep;
end;

begin
inner1 := 10;
nested;
outer2 := inner1;
end;

begin
outer1 := 0;
outer2 := 0;
flag := false;

while outer1 < 5 do
begin
outer1 := outer1 +
1;
outer;
end;

if outer2 > 0 then
flag := true;
end.

```

```

KEYWORD(var)
IDENTIFIER(outer1)
COMMA(,)
IDENTIFIER(outer2)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(flag)
COLON(:)
KEYWORD(boolean)
SEMICOLON(;)
KEYWORD(procedure)
IDENTIFIER(outer)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(inner1)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(procedure)
IDENTIFIER(nested)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(deep)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(deep)
ASSIGN_OPERATOR(:=)
NUMBER(5)
SEMICOLON(;)
IDENTIFIER(outer1)
ASSIGN_OPERATOR(:=)
IDENTIFIER(deep)
SEMICOLON(;)
KEYWORD(end)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(inner1)
ASSIGN_OPERATOR(:=)
NUMBER(10)
SEMICOLON(;)
IDENTIFIER(nested)
SEMICOLON(;)
IDENTIFIER(outer2)
ASSIGN_OPERATOR(:=)
IDENTIFIER(inner1)
SEMICOLON(;)
KEYWORD(end)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(outer1)
ASSIGN_OPERATOR(:=)
NUMBER(0)
SEMICOLON(;)

```

case sebelumnya. Fokusnya adalah memastikan lexer bisa mengenali struktur blok bertingkat (prosedur dalam prosedur) dan keyword while...do.

The image displays three separate terminal windows, each showing the output of a lexical analyzer. The output consists of tokens and their corresponding line numbers. The tokens include KEYWORD, IDENTIFIER, COMMA, COLON, SEMICOLON, ASSIGN\_OPERATOR, and NUMBER. The windows are arranged vertically, showing different parts of the program's lexemes.

	IDENTIFIER(outer2) ASSIGN_OPERATOR(:=) NUMBER(0) SEMICOLON(;) IDENTIFIER(flag) ASSIGN_OPERATOR(:=) IDENTIFIER(false) SEMICOLON(;) KEYWORD(while) IDENTIFIER(outer1) RELATIONAL_OPERATOR(<) NUMBER(5) KEYWORD(do) KEYWORD(begin) IDENTIFIER(outer1) ASSIGN_OPERATOR(:=) IDENTIFIER(outer1) ARITHMETIC_OPERATOR(+) NUMBER(1) SEMICOLON(;) IDENTIFIER(outer) SEMICOLON(;) KEYWORD(end) SEMICOLON(;) KEYWORD(if) IDENTIFIER(outer2) RELATIONAL_OPERATOR(>) NUMBER(0) KEYWORD(then) IDENTIFIER(flag) ASSIGN_OPERATOR(:=) IDENTIFIER(true) SEMICOLON(;) KEYWORD(end) DOT(.)	
--	---	--

## BAB V: Kesimpulan dan Saran

### 5.1 Kesimpulan

Tugas Besar Teori Bahasa Formal dan Otomata (IF2224-24) Milestone 1 ini berhasil mengimplementasikan tantangan tahap pertama dari proses kompilasi, yaitu analisis leksikal (lexical analysis) pada bahasa pemrograman Pascal-S yang merupakan subset dari bahasa Pascal. Melalui pembangunan lexical analyzer (lexer) berbasis Deterministic Finite Automata (DFA), program mampu membaca kode sumber Pascal-S dan mengubahnya menjadi deretan token yang memiliki arti semantik, seperti keyword, identifier, operator, literal, dan simbol-simbol lainnya.

### 5.2 Saran

Setelah mengerjakan Tugas Besar Teori Bahasa Formal dan Otomata (IF2224-24) Milestone 1 ini, kami mendapatkan berbagai saran dan pelajaran untuk diterapkan di kemudian hari pada milestone-milestone selanjutnya ataupun hal-hal yang masih terkait, beberapa diantaranya :

1. Definisikan dan validasikan state dengan lebih teliti;
2. Pastikan kelas karakter disusun secara disjungtif agar tidak menyebabkan NFA;
3. Lakukan lebih banyak pengujian pada kasus yang rawan untuk memastikan program berjalan dengan baik sesuai harapan.

## Lampiran

Pranala Repositori Github: <https://github.com/fathurwithyou/PIA-Tubes-IF2224>

Pranala Workspace Diagram:

[https://drive.google.com/file/d/1czvTxZ2tvb7p6O\\_OSoOQtYlc81k2ghwc/view](https://drive.google.com/file/d/1czvTxZ2tvb7p6O_OSoOQtYlc81k2ghwc/view)

Tabel Pembagian Tugas:

NIM	Nama Lengkap	Persentase Pembagian Tugas
13523021	Muhammad Raihan Nazhim Oktana	100%
13523057	Faqih Muhammad Syuhada	100%
13523097	Shanice Feodora Tjahjono	100%
13523105	Muhammad Fathur Rizky	100%

## Daftar Pustaka

[1] Hopcroft, J. E., Motwani, R., & Ullman, J. D. Introduction to Automata Theory, Languages, and Computation. Prentice Hall, edisi ke-3, 2006.

- [2] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. Compilers: Principles, Techniques, and Tools. Addison-Wesley, edisi ke-2, 2006.
- [3] Cooper, K. D., & Torczon, L. Engineering a Compiler. Morgan Kaufmann, edisi ke-3, 2022.
- [4] ISO/IEC 7185:1990 — Programming languages — Pascal. International Organization for Standardization, 1990.
- [5] Wirth, N. PASCAL-S: A Subset and its Implementation. ETH Zürich, 1975.