IF2224 – Teori Bahasa Formal & Otomata
Laporan Tugas Besar Milestone 3 - PASCAL-S Compiler

Dipersiapkan oleh:

K3 Kelompok2

Rhio Bimo Prakoso Sugiyanto	13523123
Muhammad Aulia Azka	13523137
Arlow Emmanuel Hergara	13523161
Fachriza Ahmad Setiyono	13523162
Filbert Engyo	13523163

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
JL. GANESA 10, BANDUNG 40132

2025

Daftar Isi

Daftar Isi **2**
Daftar Gambar **4**
1. Landasan Teori **5**
1.1. Semantic Analysis 5
1.2. Attributed Grammar 5
1.3. Symbol Table 5
1.4. Visitor / Visit Function 6
2. Perancangan & Implementasi **7**
2.1. Perancangan Program 7
2.2. Implementasi Program 7
2.2.1. access.go 10
2.2.2. additiveoperator.go 11
2.2.3. analyzer.go 12
2.2.4. arrayaccess.go 16
2.2.5. arraytype.go 21
2.2.6. assignmentstatement.go 23
2.2.7. compoundstatement.go 24
2.2.8. constdeclaration.go 25
2.2.9. constdeclarationpart.go 26
2.2.10. declarationpart.go 27
2.2.11. errors.go 28
2.2.12. expression.go 34
2.2.13. factor.go 35
2.2.14. formalparameterlist.go 36
2.2.15. forstatement.go 39
2.2.16. functiondeclaration.go 41
2.2.17. identifierlist.go 45
2.2.18. ifstatement.go 46
2.2.19. multiplicativeoperator.go 47
2.2.20. parameterlist.go 48
2.2.21. proceduredeclaration.go 49
2.2.22. program.go 52
2.2.23. programheader.go 53
2.2.24. range.go 54
2.2.25. recordtype.go 56
2.2.26. relationaloperator.go 58
2.2.27. simpleexpression.go 59
2.2.28. statement.go 62
2.2.29. statementlist.go 62
2.2.30. staticaccess.go 63

2.2.31. staticevaluator.go 67
2.2.32. subprogramcall.go 72
2.2.33. subprogramdeclaration.go 74
2.2.34. term.go 75
2.2.35. token.go 79
2.2.36. type.go 84
2.2.37. typecompat.go 85
2.2.38. typedeclaration.go 88
2.2.39. typedeclarationpart.go 90
2.2.40. vardeclaration.go 91
2.2.41. vardeclarationpart.go 93
2.2.42. whilestatement.go 94
2.2.43. atab.go 95
2.2.44. btab.go 97
2.2.45. dst.go 98
2.2.46. dstdisplay.go 102
2.2.47. strtab.go 105
2.2.48. tab.go 107
2.2.49. main.go 110
2.3. Alur Kerja Program 113
3. Pengujian 115
4. Kesimpulan dan Saran 124
4.1. Kesimpulan 124
4.2. Saran 124
Lampiran 125
Daftar Pustaka 125

Daftar Gambar

1. Landasan Teori

1.1. Semantic Analysis

Semantic analysis merupakan fase ketiga dalam proses kompilasi *compiler* setelah *lexical analysis* dan *syntax analysis*, yang bertugas memverifikasi makna semantik program menggunakan *parse tree* atau *Abstract Syntax Tree* (AST) beserta *symbol table*. Fase ini memastikan bahwa deklarasi dan pernyataan program secara semantik benar, seperti melakukan *type checking* untuk kompatibilitas tipe data pada operasi, *scope resolution* untuk memvalidasi penggunaan *identifier* dalam konteks scope yang tepat, serta deteksi kesalahan seperti variabel tidak terdefinisi atau duplikasi deklarasi.

Semantic analysis berfokus pada *type checking* yang memeriksa apakah operand operasi sesuai definisi tipe, *label checking* untuk memastikan label referensi ada, serta *flow control check* yang mengonfirmasi penggunaan struktur kontrol seperti *break* atau *continue* berada di tempat yang benar. *Semantic analyzer* mengumpulkan informasi tipe dan menyimpannya di *syntax tree* atau *symbol table*, yang kemudian digunakan untuk generasi kode antara seperti *three-address code*. Proses ini membedakan antara *static semantics* yang dicek saat kompilasi dan *dynamic semantics* yang dievaluasi saat runtime.

Input *semantic analysis* berasal dari *parser* berupa *syntax tree* lengkap dan *symbol table* awal, sementara outputnya adalah *annotated tree* dengan atribut semantik tambahan yang siap untuk fase optimasi dan *code generation*. Fase ini befungsi sebagai jembatan antara *front-end* (analisis) dan *back-end* (sintesis) *compiler*, mendeteksi error semantik seperti *type mismatch* atau *reserved identifier misuse* tanpa mengeksekusi program. Hubungannya dengan *attributed grammar* menyediakan aturan semantik, *symbol table* menyimpan data *identifier*, dan *visitor pattern* memfasilitasi *traversal* AST secara efisien.

1.2. Attributed Grammar

Attributed grammar memperluas *context-free grammar* (CFG) atribut tersebut. Atribut dibagi menjadi *synthesized* yang dihitung dari anak node ke parent dan *inherited* dari parent ke anak, memungkinkan evaluasi semantik selama parsing.

Struktur *attributed grammar* mencakup CFG dasar, atribut untuk menyimpan informasi seperti tipe atau nilai, serta semantic rules yang dievaluasi saat membangun parse tree. *Grammar L-attributed* mendukung evaluasi satu kali turun pohon, efisien untuk *compiler top-down*.

1.3. Symbol Table

Symbol table adalah struktur data yang menyimpan informasi semantik tentang *identifier* seperti variabel, fungsi, dan tipe, termasuk nama, tipe data, *scope*, dan lokasi memori. Digunakan di berbagai fase *compiler*: *lexer* memasukkan *identifier*, *parser* membangun entri dasar, *semantic analyzer* memvalidasi dan menambahkan atribut seperti tipe.

Implementasi umum meliputi *hash table* untuk pencarian cepat atau *stack table* bersarang untuk *scope block-structured language*. Setiap entri berisi *pointer* ke informasi terkait, mendukung operasi *insert*, *lookup*, dan *delete* selama analisis semantik.

1.4. Visitor / Visit Function

Visitor pattern memisahkan algoritma *traversal* AST dari struktur *node*, memungkinkan penambahan operasi semantik baru tanpa modifikasi kelas AST. *Interface Visitor* mendeklarasikan metode *visit* untuk setiap tipe *node*, sementara *concrete visitor* mengimplementasikan logika seperti pembuatan *symbol table* atau *type checking*.

Pada AST, setiap *node* memanggil accept dari visitor yang men-*dispatch* ke *visit* yaitu *node* pada *visitor*, memfasilitasi *traversal* rekursif dan injeksi *semantic actions*. Dalam compiler, ini mendukung *multiple passes*: satu untuk *build symbol table*, lain untuk *semantic checks*.

2. Perancangan & Implementasi

2.1. Perancangan Program

Semantic Analyzer terdiri dari beberapa tipe data: Tab, Atab, Btab, StrTab, dan DST. Tab adalah tabel simbol yang memuat semua simbol yang didefinisikan baik oleh user maupun sistem. Tabel ini menyimpan informasi mengenai peran dari simbol, tipenya, dan informasi lain untuk menjelaskan simbol tersebut. Atab merupakan tabel berisi definisi tipe array termasuk jenis index, elemen, serta range dari suatu array. Tabel ini juga menyimpan informasi mengenai ukuran dari array. Btab adalah tabel yang berisi informasi mengenai blok-blok subprogram serta record yang digunakan dalam program. Informasi ini digunakan untuk mengidentifikasi parameter-parameter dalam fungsi atau field-field dalam record. StrTab adalah tabel bantuan yang menyimpan string literal. DST (Decorated Syntax Tree) adalah struktur data pohon yang menggambarkan struktur program setelah sudah diberikan makna semantic.

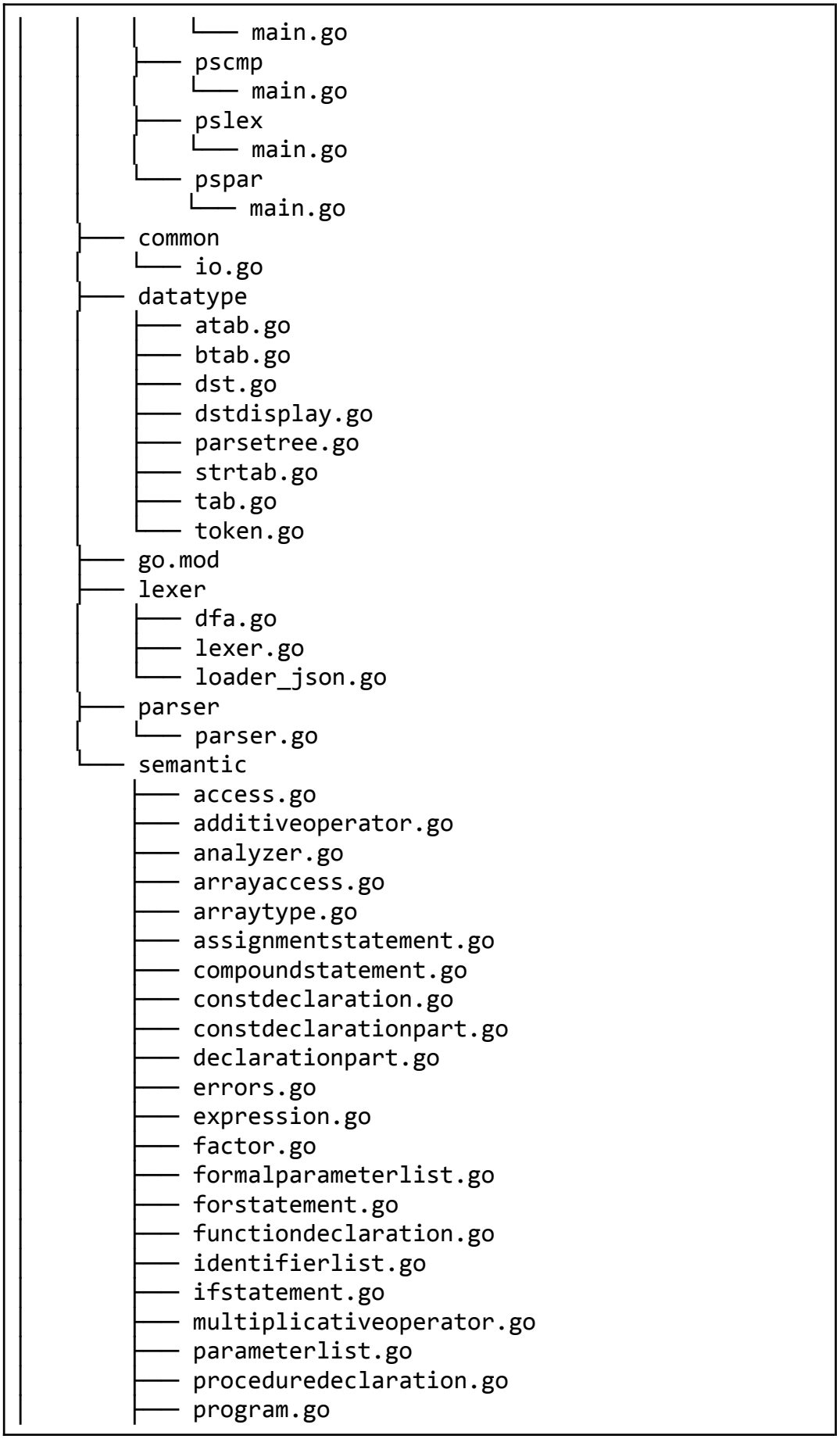
Untuk melakukan analisis semantik, compiler menggunakan kelas SemanticAnalyzer. Kelas ini memiliki fungsi visit untuk setiap node pada Parse Tree sehingga dapat mentransformasikan node pada Parse Tree menjadi node dalam DST. SemanticAnalyzer juga menyimpan Tab, Atab, Btab, dan StrTab yang berhubungan dengan DST yang dibangun. Kelas ini juga memiliki fungsi-fungsi bantuan untuk melakukan type-checking yang lebih handal, error message yang lebih bermakna, dan lain-lain.

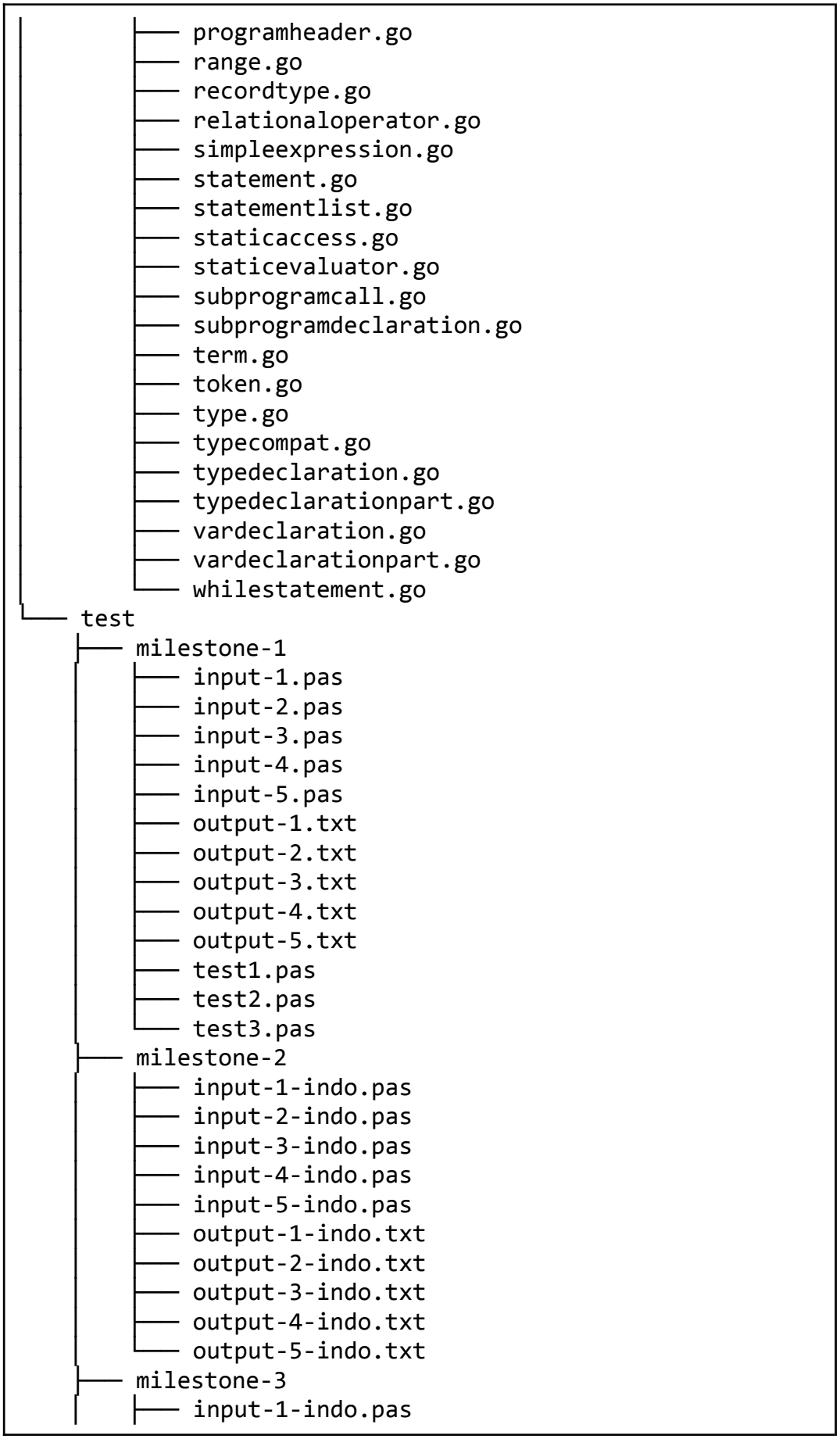
2.2. Implementasi Program

Untuk pengimplementasian *semantic analysis*, kami memutuskan untuk menggunakan **Golang** versi 1.24.2 Golang dipilih karena sintaksnya yang mudah dimengerti (*high-level language*), tetapi tetap memiliki kelebihan seperti efisiensi memori, performa yang cepat, dan *build ecosystem* yang stabil.

Project Structure

```
NIG-TUBES-IF2224
├── README.md
├── bin
│   ├── pschk
│   └── pschk.exe
├── config
│   ├── tokenizer.json
│   ├── tokenizer_indo.json
│   └── tokenizer_m3.json
├── doc
│   ├── Laporan-1-NIG.pdf
│   ├── Laporan-2-NIG.pdf
│   └── lexer-dfa-full.pdf
├── psc
│   ├── cmd
│   │   ├── pschk
```


│   │   │   └── main.go
│   │   ├── pscmp
│   │   │   └── main.go
│   │   ├── pslex
│   │   │   └── main.go
│   │   └── pspar
│   │   └── main.go
│   ├── common
│   │   └── io.go
│   ├── datatype
│   │   ├── atab.go
│   │   ├── btab.go
│   │   ├── dst.go
│   │   ├── dstdisplay.go
│   │   ├── parsetree.go
│   │   ├── strtab.go
│   │   ├── tab.go
│   │   └── token.go
│   ├── go.mod
│   ├── lexer
│   │   ├── dfa.go
│   │   ├── lexer.go
│   │   └── loader_json.go
│   ├── parser
│   │   └── parser.go
│   └── semantic
│   ├── access.go
│   ├── additiveoperator.go
│   ├── analyzer.go
│   ├── arrayaccess.go
│   ├── arraytype.go
│   ├── assignmentstatement.go
│   ├── compoundstatement.go
│   ├── constdeclaration.go
│   ├── constdeclarationpart.go
│   ├── declarationpart.go
│   ├── errors.go
│   ├── expression.go
│   ├── factor.go
│   ├── formalparameterlist.go
│   ├── forstatement.go
│   ├── functiondeclaration.go
│   ├── identifierlist.go
│   ├── ifstatement.go
│   ├── multiplicativeoperator.go
│   ├── parameterlist.go
│   ├── proceduredeclaration.go
│   ├── program.go


├── programheader.go
├── range.go
├── recordtype.go
├── relationaloperator.go
├── simpleexpression.go
├── statement.go
├── statementlist.go
├── staticaccess.go
├── staticevaluator.go
├── subprogramcall.go
├── subprogramdeclaration.go
├── term.go
├── token.go
├── type.go
├── typecompat.go
├── typedeclaration.go
├── typedeclarationpart.go
├── vardeclaration.go
├── vardeclarationpart.go
└── whilestatement.go
└── test
├── milestone-1
│ ├── input-1.pas
│ ├── input-2.pas
│ ├── input-3.pas
│ ├── input-4.pas
│ ├── input-5.pas
│ ├── output-1.txt
│ ├── output-2.txt
│ ├── output-3.txt
│ ├── output-4.txt
│ ├── output-5.txt
│ ├── test1.pas
│ ├── test2.pas
│ └── test3.pas
├── milestone-2
│ ├── input-1-indo.pas
│ ├── input-2-indo.pas
│ ├── input-3-indo.pas
│ ├── input-4-indo.pas
│ ├── input-5-indo.pas
│ ├── output-1-indo.txt
│ ├── output-2-indo.txt
│ ├── output-3-indo.txt
│ ├── output-4-indo.txt
│ └── output-5-indo.txt
├── milestone-3
│ ├── input-1-indo.pas

```
│   ├── input-10-indo.pas
│   ├── input-11-indo.pas
│   ├── input-2-indo.pas
│   ├── input-3-indo.pas
│   ├── input-4-indo.pas
│   ├── input-5-indo.pas
│   ├── input-6-indo.pas
│   ├── input-7-indo.pas
│   ├── input-8-indo.pas
│   └── input-9-indo.pas
└── test_features.pas
```

2.2.1. access.go

```go
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeAccess(parsetree *dt.ParseTree)
(*dt.DecoratedSyntaxTree, semanticType, error) {
    if parsetree.RootType != dt.ACCESS_NODE {
        return nil, semanticType{}, errors.New("expected access node")
    }

    callNode := parsetree.Children[0]
    var staticAccessNode *dt.ParseTree

    var prev *dt.DecoratedSyntaxTree
    var typ semanticType
    var err error

    if callNode.RootType == dt.SUBPROGRAM_CALL_NODE {
        if len(parsetree.Children) == 3 {
            staticAccessNode = &parsetree.Children[2]
        }

        prev, typ, err = a.analyzeSubprogramCall(&callNode)
```

```
        if err != nil {
            return nil, semanticType{}, err
        }
    } else {
        staticAccessNode = &parsetree.Children[0]
    }

    if staticAccessNode != nil {
        return a.analyzeStaticAccess(staticAccessNode, prev)
    }

    return prev, typ, nil
}
```

Program access.go bertugas sebagai dispatcher atau utility untuk menganalisis akses ke identifier (bisa simple identifier atau complex access dengan field/index).

2.2.2. additiveoperator.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeAdditiveOperator(parsetree *dt.ParseTree)
(dt.DSTNodeType, error) {
    if parsetree.RootType != dt.ADDITIVE_OPERATOR_NODE {
        return dt.DST_ADD_OPERATOR, errors.New("operator is not additive")
    }

    switch parsetree.Children[0].TokenValue.Lexeme {
    case "+":
        return dt.DST_ADD_OPERATOR, nil
    case "-":
        return dt.DST_SUB_OPERATOR, nil
    case "atau":
        return dt.DST_OR_OPERATOR, nil
```

```
    default:
        return dt.DST_ADD_OPERATOR, errors.New("unknown additive operator")
    }
}
```

Program additiveoperator.go bertugas untuk menangani operator aditif (+, -). Melakukan type promotion dan memastikan kedua operand compatible.

2.2.3. analyzer.go

```
package semantic

import (
    "strconv"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

type SemanticAnalyzer struct {
    parseTree *dt.ParseTree
    tab       dt.Tab
    atab      dt.Atab
    btab      dt.Btab
    strtab    dt.StrTab
    root      int
    depth     int
    stackSize int
}

type semanticType struct {
    StaticType dt.TabEntryType
    Reference  int
}

func New(parseTree *dt.ParseTree) *SemanticAnalyzer {
    return &SemanticAnalyzer{
        parseTree: parseTree,
        tab: dt.Tab{
            dt.TabEntry{
                Identifier: "string",
                Link:       -1,
```

```
            Object:     dt.TAB_ENTRY_TYPE,
            Type:       dt.TAB_ENTRY_ARRAY,
            Reference:  0,
            Normal:     false,
            Level:      0,
            Data:       0,
        },
        dt.TabEntry{
            Identifier: "write",
            Link:       0,
            Object:     dt.TAB_ENTRY_PROC,
            Type:       dt.TAB_ENTRY_NONE,
            Reference:  0,
            Normal:     false,
            Level:      0,
            Data:       0,
        },
        dt.TabEntry{
            Identifier: "writeparam1",
            Link:       1,
            Object:     dt.TAB_ENTRY_PARAM,
            Type:       dt.TAB_ENTRY_ALIAS,
            Reference:  0,
            Normal:     false,
            Level:      1,
            Data:       0,
        },
        dt.TabEntry{
            Identifier: "writeparam2",
            Link:       2,
            Object:     dt.TAB_ENTRY_PARAM,
            Type:       dt.TAB_ENTRY_ALIAS,
            Reference:  0,
            Normal:     false,
            Level:      1,
            Data:       0,
        },
    },
    atab: dt.Atab{
        dt.AtabEntry{
```

```
                IndexType:        dt.TAB_ENTRY_INTEGER,
                ElementType:      dt.TAB_ENTRY_CHAR,
                ElementReference: 0,
                LowBound:         0,
                HighBound:        255,
                ElementSize:      1,
                TotalSize:        256,
            },
        },
        btab: dt.Btab{
            dt.BtabEntry{
                Start:        2,
                ParamEnd:     3,
                ReturnEnd:    3,
                End:          3,
                ParamSize:    512,
                ReturnSize:   0,
                VariableSize: 0,
            },
        },
        strtab:    make(dt.StrTab, 0),
        root:      3,
        depth:     0,
        stackSize: 0,
    }
}

func (a *SemanticAnalyzer) GetSymbols() (dt.Tab, dt.Atab, dt.Btab,
dt.StrTab) {
    return a.tab, a.atab, a.btab, a.strtab
}

func (a *SemanticAnalyzer) Analyze() (dt.Tab, dt.Atab, dt.Btab, dt.StrTab,
*dt.DecoratedSyntaxTree, error) {
    dst, err := a.analyzeProgram(a.parseTree)
    tab, atab, btab, strtab := a.GetSymbols()
    return tab, atab, btab, strtab, dst, err
}
```

```
func (a *SemanticAnalyzer) resolveAliasType(t semanticType) semanticType {

    for t.StaticType == dt.TAB_ENTRY_ALIAS {

        t = semanticType{
            StaticType: a.tab[t.Reference].Type,
            Reference:  a.tab[t.Reference].Reference,
        }
    }
    return t
}

func (a *SemanticAnalyzer) checkTypeEquality(t1 semanticType, t2
semanticType) bool {

    resolved1 := a.resolveAliasType(t1)
    resolved2 := a.resolveAliasType(t2)

    if resolved1.StaticType == resolved2.StaticType {
        switch resolved1.StaticType {
        case dt.TAB_ENTRY_NONE:
            fallthrough
        case dt.TAB_ENTRY_INTEGER:
            fallthrough
        case dt.TAB_ENTRY_REAL:
            fallthrough
        case dt.TAB_ENTRY_BOOLEAN:
            fallthrough
        case dt.TAB_ENTRY_CHAR:
            return true
        case dt.TAB_ENTRY_RECORD:
            fallthrough
        case dt.TAB_ENTRY_ARRAY:
            return resolved1.Reference == resolved2.Reference
        }
    }

    return false
}
```

```
func (a *SemanticAnalyzer) getTypeSize(t semanticType) int {
    switch t.StaticType {
    case dt.TAB_ENTRY_INTEGER:
        return strconv.IntSize / 8
    case dt.TAB_ENTRY_REAL:
        return strconv.IntSize / 8
    case dt.TAB_ENTRY_BOOLEAN:
        return 1
    case dt.TAB_ENTRY_CHAR:
        return 1
    case dt.TAB_ENTRY_RECORD:
        return a.btab[t.Reference].VariableSize
    case dt.TAB_ENTRY_ARRAY:
        return a.atab[t.Reference].TotalSize
    case dt.TAB_ENTRY_ALIAS:
        return a.getTypeSize(semanticType{
            StaticType: a.tab[t.Reference].Type,
            Reference:  a.tab[t.Reference].Reference,
        })
    default:
        return 0
    }
}
```

Program analyzer.go menjadi program utama yang mengkoordinasikan seluruh proses analisis semantik. Berisi SemanticAnalyzer struct yang mengelola tabel simbol (tab), tabel blok (btab), tabel array (atab), dan tabel string (strtab). File ini memiliki method Analyze() yang dispatch ke berbagai fungsi analisis berdasarkan tipe node parse tree.

2.2.4. arrayaccess.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeArrayAccess(parsetree *dt.ParseTree, prev
```

```
*dt.DecoratedSyntaxTree) (*dt.DecoratedSyntaxTree, semanticType, error) {
    if parsetree.RootType != dt.ARRAY_ACCESS_NODE {
        return nil, semanticType{}, errors.New("parse tree node is not array
access node")
    }

    root := a.root

    if prev != nil {
        switch prev.SelfType {
        case dt.DST_CONST:
            fallthrough
        case dt.DST_VARIABLE:
            fallthrough
        case dt.DST_RECORD_FIELD:
            fallthrough
        case dt.DST_FUNCTION_CALL:
            symbolIndex := prev.Data

            for a.tab[symbolIndex].Type == dt.TAB_ENTRY_ALIAS {
                symbolIndex = a.tab[symbolIndex].Reference
            }

            if a.tab[symbolIndex].Type != dt.TAB_ENTRY_RECORD {
                return nil, semanticType{}, errors.New("non record type has
no fields")
            }

            root = a.btab[a.tab[symbolIndex].Reference].End
        }
    }

    token := parsetree.Children[0].TokenValue
    index, tabEntry := a.tab.FindIdentifier(token.Lexeme, root)

    if tabEntry == nil {
        return nil, semanticType{}, errors.New("undeclared identifier")
    }

    var dstType dt.DSTNodeType
```

```
    switch tabEntry.Object {
    case dt.TAB_ENTRY_CONST:
        dstType = dt.DST_CONST
    case dt.TAB_ENTRY_VAR:
        dstType = dt.DST_VARIABLE
    case dt.TAB_ENTRY_FIELD:
        dstType = dt.DST_RECORD_FIELD
    default:
        return nil, semanticType{}, errors.New("identifier does not
reference a constant, variable, or field")
    }

    if tabEntry.Type != dt.TAB_ENTRY_ARRAY {
        return nil, semanticType{}, errors.New("identifier does not hold an
array value")
    }

    var children []dt.DecoratedSyntaxTree
    if prev == nil {
        children = make([]dt.DecoratedSyntaxTree, 0)
    } else {
        children = []dt.DecoratedSyntaxTree{*prev}
    }

    prev = &dt.DecoratedSyntaxTree{
        Property: dt.DST_FROM,
        SelfType: dstType,
        Data:     index,
        Children: children,
    }


    recursiveNodes := make([]dt.ParseTree, 0)
    for i := 2; i < len(parsetree.Children); i += 3 {
        recursiveNodes = append(recursiveNodes, parsetree.Children[i])
    }

    return a.analyzeRecursiveArrayAccess(recursiveNodes, prev)
}
```

```
func (a *SemanticAnalyzer) analyzeRecursiveArrayAccess(nodes []dt.ParseTree,
prev *dt.DecoratedSyntaxTree) (*dt.DecoratedSyntaxTree, semanticType, error)
{
    if nodes[0].RootType != dt.EXPRESSION_NODE {
        return nil, semanticType{}, errors.New("expected valid array index")
    }

    atabIndex := 0

    switch prev.SelfType {
    case dt.DST_VARIABLE:
        tabIndex := prev.Data

        for a.tab[tabIndex].Type == dt.TAB_ENTRY_ALIAS {
            tabIndex = a.tab[tabIndex].Reference
        }

        if a.tab[tabIndex].Type != dt.TAB_ENTRY_ARRAY {
            return nil, semanticType{}, errors.New("expected variable to
point to an array")
        }

        atabIndex = a.tab[tabIndex].Reference
    case dt.DST_ARRAY_ELEMENT:
        atabIndex = prev.Data
    default:
        return nil, semanticType{}, errors.New("object cannot be indexed")
    }

    expectedType := semanticType{
        StaticType: a.atab[atabIndex].ElementType,
        Reference:  a.atab[atabIndex].ElementReference,
    }

    index, indexType, err := a.analyzeExpression(&nodes[0])

    if err != nil {
        return nil, semanticType{}, err
    }
```

```
    if indexType.StaticType != a.atab[atabIndex].IndexType {
        return nil, semanticType{}, errors.New("expression type does not
match index type of array")
    }

    index.Property = dt.DST_INDEX

    var self *dt.DecoratedSyntaxTree

    if prev != nil {
        prev.Property = dt.DST_FROM
        self = &dt.DecoratedSyntaxTree{
            SelfType: dt.DST_ARRAY_ELEMENT,
            Data:     atabIndex,
            Children: []dt.DecoratedSyntaxTree{
                *prev,
                *index,
            },
        }
    } else {
        self = &dt.DecoratedSyntaxTree{
            SelfType: dt.DST_ARRAY_ELEMENT,
            Data:     atabIndex,
            Children: []dt.DecoratedSyntaxTree{*index},
        }
    }

    if len(nodes) == 1 {
        return self, expectedType, nil
    }

    if expectedType.StaticType != dt.TAB_ENTRY_ARRAY && len(nodes) > 1 {
        return nil, semanticType{}, errors.New("cannot access non-array type
as if it was an array")
    }

    return a.analyzeRecursiveArrayAccess(nodes[1:], self)
}
```

Program arrayaccess.go bertugas untuk menganalisis array access (subscript notation seperti array[index]). Memvalidasi index type dan menghasilkan tipe elemen array.

2.2.5. arraytype.go

```
package semantic

import (
    "errors"
    "strconv"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeArrayType(parsetree *dt.ParseTree) (int,
dt.TabEntry, error) {
    if parsetree.RootType != dt.ARRAY_TYPE_NODE {
        return -1, dt.TabEntry{}, errors.New("expected array type")
    }

    begin, end, indexType, err := a.analyzeRange(&parsetree.Children[2])

    if err != nil {
        return -1, dt.TabEntry{}, err
    }

    if indexType.StaticType == dt.TAB_ENTRY_REAL {
        return -1, dt.TabEntry{}, errors.New("cannot use real value as array
index")
    }

    _, tabEntry, err := a.analyzeType(&parsetree.Children[5])

    if err != nil {
        return -1, dt.TabEntry{}, err
    }

    atabEntry := dt.AtabEntry{
        IndexType:        indexType.StaticType,
        ElementType:      tabEntry.Type,
        ElementReference: tabEntry.Reference,
```

```
        LowBound:         begin,
        HighBound:        end,
    }

    atabIdx, _ := a.atab.FindArray(atabEntry)

    if atabIdx == -1 {
        atabIdx = len(a.atab)

        switch tabEntry.Type {
        case dt.TAB_ENTRY_ARRAY:
            atabEntry.ElementSize = a.atab[tabEntry.Reference].TotalSize
        case dt.TAB_ENTRY_RECORD:
            atabEntry.ElementSize = a.btab[tabEntry.Reference].VariableSize
        case dt.TAB_ENTRY_BOOLEAN:
            atabEntry.ElementSize = 1
        case dt.TAB_ENTRY_CHAR:
            atabEntry.ElementSize = 1
        case dt.TAB_ENTRY_INTEGER:
            atabEntry.ElementSize = strconv.IntSize
        case dt.TAB_ENTRY_REAL:
            atabEntry.ElementSize = strconv.IntSize
        default:
            return -1, dt.TabEntry{}, errors.New("unknown type")
        }

        atabEntry.TotalSize = atabEntry.ElementSize * (end - begin + 1)

        a.atab = append(a.atab, atabEntry)
    }

    return -1, dt.TabEntry{
        Type:      dt.TAB_ENTRY_ARRAY,
        Reference: atabIdx,
    }, nil
}
```

Program arraytype.go bertugas untuk menganalisis deklarasi tipe array. Menyimpan informasi tentang index type, element type, dan dimensi.

2.2.6. *assignmentstatement.go*

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeAssignmentStatement(parsetree
*dt.ParseTree) (*dt.DecoratedSyntaxTree, error) {
    if parsetree.RootType != dt.ASSIGNMENT_STATEMENT_NODE {
        return nil, errors.New("expected an assignment statement node")
    }

    target, targetType, err := a.analyzeStaticAccess(&parsetree.Children[0],
nil)

    if err != nil {
        return nil, err
    }

    value, valueType, err := a.analyzeExpression(&parsetree.Children[2])

    if err != nil {
        return nil, err
    }



    if !a.checkTypeEquality(targetType, valueType) {
        if a.canCastImplicitly(valueType, targetType) {
            value, valueType = a.insertImplicitCast(value, valueType,
targetType)
        } else {

            assignToken := parsetree.Children[1].TokenValue
            return nil, a.newAssignmentError(
                targetType.StaticType.String(),
                valueType.StaticType.String(),
```

```
                assignToken,
            )
        }
    }

    target.Property = dt.DST_TARGET
    value.Property = dt.DST_VALUE

    return &dt.DecoratedSyntaxTree{
        SelfType: dt.DST_ASSIGNMENT_OPERATOR,
        Data:     int(targetType.StaticType),
        Children: []dt.DecoratedSyntaxTree{
            *target,
            *value,
        },
    }, nil
}
```

Program assignmentstatement.go bertugas untuk menganalisis statement penugasan. Melakukan type checking antara target dan value, melakukan implicit casting jika perlu, dan memastikan kedua tipe kompatibel.

2.2.7. compoundstatement.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeCompoundStatement(parsetree *dt.ParseTree)
(*dt.DecoratedSyntaxTree, error) {
    if parsetree.RootType != dt.COMPOUND_STATEMENT_NODE {
        return nil, errors.New("expected compound statement section")
    }

    statements, err := a.analyzeStatementList(&parsetree.Children[1])

    if err != nil {
```

```
        return nil, err
    }

    return &dt.DecoratedSyntaxTree{
        SelfType: dt.DST_BLOCK,
        Children: statements,
    }, nil
}
```

Program compoundstatement.go bertugas untuk menganalisis compound statement (BEGIN...END). Menganalisis statement list di dalamnya.

2.2.8. constdeclaration.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeConstDeclaration(parsetree *dt.ParseTree)
(*dt.DecoratedSyntaxTree, error) {
    if parsetree.RootType != dt.CONST_DECLARATION_NODE {
        return nil, errors.New("expected const declaration")
    }

    identifier := parsetree.Children[0].TokenValue.Lexeme
    _, prev := a.tab.FindIdentifier(identifier, a.root)

    if prev != nil {
        if prev.Level == a.depth {
            return nil, errors.New("identifier redefined in the same scope")
        }
    }

    val, valtype, err := a.analyzeToken(&parsetree.Children[2])

    if err != nil {
        return nil, err
```

```
    }

    tabEntry := dt.TabEntry{
        Identifier: identifier,
        Link:       a.root,
        Object:     dt.TAB_ENTRY_CONST,
        Type:       valtype.StaticType,
        Reference:  valtype.Reference,
        Level:      a.depth,
        Data:       val.Data,
    }

    a.root = len(a.tab)
    a.tab = append(a.tab, tabEntry)

    return &dt.DecoratedSyntaxTree{
        SelfType: dt.DST_CONST,
        Data:     a.root,
    }, nil
}
```

Program constdeclaration.go bertugas untuk mengmanalisis deklarasi konstanta. Melakukan static evaluation pada nilai konstanta dan menyimpannya di strtab dengan link ke scope sebelumnya.

2.2.9. constdeclarationpart.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeConstDeclarationPart(parsetree
*dt.ParseTree) (*dt.DecoratedSyntaxTree, error) {
    if parsetree.RootType != dt.CONST_DECLARATION_PART_NODE {
        return nil, errors.New("expected const declaration part")
    }
```

```
    declarations := make([]dt.DecoratedSyntaxTree,
len(parsetree.Children)-1)

    for i, constDeclaration := range parsetree.Children[1:] {
        declaration, err := a.analyzeConstDeclaration(&constDeclaration)

        if err != nil {
            return nil, err
        }

        declarations[i] = *declaration
    }

    return &dt.DecoratedSyntaxTree{
        SelfType: dt.DST_CONSTANT_DECLARATIONS,
        Children: declarations,
    }, nil
}
```

Program constdeclarationpart.go yang menganalisis deklarasi konstanta. Melakukan static evaluation pada nilai konstanta dan menyimpannya di strtab dengan link ke scope sebelumnya.

2.2.10. declarationpart.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeDeclarationPart(parsetree *dt.ParseTree)
([]dt.DecoratedSyntaxTree, error) {
    if parsetree.RootType != dt.DECLARATION_PART_NODE {
        return nil, errors.New("expected declaration part")
    }

    declarations := make([]dt.DecoratedSyntaxTree, 0)
```

```
    for _, child := range parsetree.Children {
        var declaration *dt.DecoratedSyntaxTree
        var err error

        switch child.RootType {
        case dt.CONST_DECLARATION_PART_NODE:
            declaration, err = a.analyzeConstDeclarationPart(&child)
        case dt.TYPE_DECLARATION_PART_NODE:
            declaration, err = a.analyzeTypeDeclarationPart(&child)
        case dt.VAR_DECLARATION_PART_NODE:
            declaration, err = a.analyzeVarDeclarationPart(&child)
        case dt.SUBPROGRAM_DECLARATION_NODE:
            declaration, err = a.analyzeSubprogramDeclaration(&child)
        default:
            return nil, errors.New("unknown declaration section")
        }

        if err != nil {
            return nil, err
        }

        declarations = append(declarations, *declaration)
    }

    return declarations, nil
}
```

Program declarationpart.go merupakan program yang mengelola dan mendispatch berbagai tipe deklarasi (variabel, tipe, konstanta, prosedur, fungsi) yang ada dalam sebuah blok program.

2.2.11. errors.go

```
package semantic

import (
    "fmt"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)
```

```go
type SemanticError struct {
    Message string
    Line    int
    Column  int
    Context string
    Token   *dt.Token
}

func (e *SemanticError) Error() string {
    if e.Token != nil {
        return fmt.Sprintf("Semantic error at line %d, column %d:
%s\nContext: %s\nNear: '%s'",
            e.Line, e.Column, e.Message, e.Context, e.Token.Lexeme)
    }
    return fmt.Sprintf("Semantic error at line %d, column %d: %s\nContext:
%s",
        e.Line, e.Column, e.Message, e.Context)
}

func NewSemanticError(message string, token *dt.Token, context string)
*SemanticError {
    line := 0
    column := 0
    if token != nil {
        line = token.Line
        column = token.Col
    }
    return &SemanticError{
        Message: message,
        Line:    line,
        Column:  column,
        Context: context,
        Token:   token,
    }
}

const (
    ErrRedeclaration      = "identifier already declared in this scope"
    ErrUndeclaredIdent    = "undeclared identifier"
    ErrTypeMismatch       = "type mismatch"
```

```
    ErrIncompatibleTypes  = "incompatible types"
    ErrInvalidOperation   = "invalid operation"
    ErrConstantExpected   = "constant expression expected"
    ErrArrayBoundsInvalid = "invalid array bounds"
    ErrDivisionByZero     = "division by zero"
    ErrNotAFunction       = "identifier is not a function"
    ErrNotAProcedure      = "identifier is not a procedure"
    ErrWrongArgCount      = "wrong number of arguments"
    ErrCannotAssign       = "cannot assign to this expression"
)



func (a *SemanticAnalyzer) newRedeclarationError(identifier string, token
*dt.Token) error {
    return NewSemanticError(
        fmt.Sprintf("%s: '%s'", ErrRedeclaration, identifier),
        token,
        "identifier declaration",
    )
}

func (a *SemanticAnalyzer) newUndeclaredIdentError(identifier string, token
*dt.Token) error {
    return NewSemanticError(
        fmt.Sprintf("%s: '%s'", ErrUndeclaredIdent, identifier),
        token,
        "identifier reference",
    )
}

func (a *SemanticAnalyzer) newTypeMismatchError(expected string, got string,
token *dt.Token) error {
    return NewSemanticError(
        fmt.Sprintf("%s: expected %s, got %s", ErrTypeMismatch, expected,
got),
        token,
        "type checking",
    )
}
```

```
func (a *SemanticAnalyzer) newIncompatibleTypesError(type1 string, type2
string, token *dt.Token) error {
    return NewSemanticError(
        fmt.Sprintf("%s: %s and %s", ErrIncompatibleTypes, type1, type2),
        token,
        "type compatibility check",
    )
}

func (a *SemanticAnalyzer) newConstantExpectedError(token *dt.Token) error {
    return NewSemanticError(
        ErrConstantExpected,
        token,
        "static evaluation",
    )
}

func (a *SemanticAnalyzer) newArrayBoundsError(token *dt.Token) error {
    return NewSemanticError(
        ErrArrayBoundsInvalid,
        token,
        "array type declaration",
    )
}

func (a *SemanticAnalyzer) newParameterCountError(expected int, got int,
funcName string, token *dt.Token) error {
    return NewSemanticError(
        fmt.Sprintf("parameter count mismatch for '%s': expected %d, got
%d", funcName, expected, got),
        token,
        "subprogram call",
    )
}

func (a *SemanticAnalyzer) newParameterTypeError(paramIndex int, expected
string, got string, funcName string, token *dt.Token) error {
    return NewSemanticError(
        fmt.Sprintf("parameter %d type mismatch for '%s': expected %s, got
```

```
%s", paramIndex+1, funcName, expected, got),
        token,
        "subprogram call",
    )
}

func (a *SemanticAnalyzer) newNotCallableError(identifier string, actualType
string, token *dt.Token) error {
    return NewSemanticError(
        fmt.Sprintf("'%s' is not callable (it is a %s)", identifier,
actualType),
        token,
        "subprogram call",
    )
}

func (a *SemanticAnalyzer) newInvalidArrayAccessError(identifier string,
actualType string, token *dt.Token) error {
    return NewSemanticError(
        fmt.Sprintf("cannot index '%s': not an array (type is %s)",
identifier, actualType),
        token,
        "array access",
    )
}

func (a *SemanticAnalyzer) newInvalidRecordAccessError(identifier string,
actualType string, token *dt.Token) error {
    return NewSemanticError(
        fmt.Sprintf("cannot access field of '%s': not a record (type is
%s)", identifier, actualType),
        token,
        "record field access",
    )
}

func (a *SemanticAnalyzer) newUndeclaredFieldError(fieldName string,
recordName string, token *dt.Token) error {
    return NewSemanticError(
        fmt.Sprintf("record '%s' has no field named '%s'", recordName,
```

```
fieldName),
        token,
        "record field access",
    )
}

func (a *SemanticAnalyzer) newOperatorTypeError(operator string, leftType
string, rightType string, token *dt.Token) error {
    return NewSemanticError(
        fmt.Sprintf("operator '%s' cannot be applied to types %s and %s",
operator, leftType, rightType),
        token,
        "operator type checking",
    )
}

func (a *SemanticAnalyzer) newInvalidTypeError(identifier string,
expectedKind string, actualKind string, token *dt.Token) error {
    return NewSemanticError(
        fmt.Sprintf("'%s' is not a %s (it is a %s)", identifier,
expectedKind, actualKind),
        token,
        "type checking",
    )
}

func (a *SemanticAnalyzer) newConditionTypeError(actualType string, token
*dt.Token) error {
    return NewSemanticError(
        fmt.Sprintf("condition must be boolean, got %s", actualType),
        token,
        "control flow statement",
    )
}

func (a *SemanticAnalyzer) newAssignmentError(targetType string, valueType
string, token *dt.Token) error {
    return NewSemanticError(
        fmt.Sprintf("cannot assign %s to %s", valueType, targetType),
        token,
```

```
        "assignment statement",
    )
}
```

Program errors.go berisi definisi error messages dan error handling utilities untuk semantic analyzer.

2.2.12. expression.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeExpression(parseTree *dt.ParseTree)
(*dt.DecoratedSyntaxTree, semanticType, error) {
    if len(parseTree.Children) == 1 {
        return a.analyzeSimpleExpression(&parseTree.Children[0])
    } else {
        optype, err := a.analyzeRelationalOperator(&parseTree.Children[1])

        if err != nil {
            return nil, semanticType{}, err
        }

        lhs, ltype, err := a.analyzeSimpleExpression(&parseTree.Children[0])

        if err != nil {
            return nil, ltype, err
        }

        rhs, rtype, err := a.analyzeSimpleExpression(&parseTree.Children[2])

        if err != nil {
            return nil, rtype, err
        }

        promotedLhs, promotedRhs, _, compatible := a.promoteTypes(lhs,
```

```
ltype, rhs, rtype)
        if !compatible {
            return nil, ltype, errors.New("operand types are incompatible")
        }

        return &dt.DecoratedSyntaxTree{
            SelfType: optype,
            Children: []dt.DecoratedSyntaxTree{
                *promotedLhs,
                *promotedRhs,
            },
        }, semanticType{StaticType: dt.TAB_ENTRY_BOOLEAN}, nil
    }
}
```

Program expression.go yang berperan sebagai analyzer utama untuk ekspresi. Melakukan dispatch ke simpleexpression dan menangani operator relasional (comparison).

2.2.13. factor.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeFactor(parseTree *dt.ParseTree)
(*dt.DecoratedSyntaxTree, semanticType, error) {
    if parseTree.Children[0].RootType == dt.TOKEN_NODE {
        if parseTree.Children[0].TokenValue.Type == dt.LPARENTHESIS {
            return a.analyzeExpression(&parseTree.Children[1])
        } else if parseTree.Children[0].TokenValue.Lexeme == "tidak" {
            dst, typ, err := a.analyzeFactor(&parseTree.Children[1])

            if err != nil {
                return nil, typ, err
            }

            if typ.StaticType != dt.TAB_ENTRY_BOOLEAN {
```

```
                return nil, typ, errors.New("not operator only works on
boolean expressions")
            }

            dst.Property = dt.DST_OPERAND

            return &dt.DecoratedSyntaxTree{
                SelfType: dt.DST_NOT_OPERATOR,
                Children: []dt.DecoratedSyntaxTree{*dst},
            }, typ, nil
        }

        return a.analyzeToken(&parseTree.Children[0])
    }

    return a.analyzeAccess(&parseTree.Children[0])
}
```

Program factor.go menganalisis factor (unit terkecil dalam ekspresi): identifier, konstanta, function call, atau sub-expression dalam parentheses.

2.2.14. formalparameterlist.go

```
package semantic

import (
    "errors"
    "fmt"
    "strconv"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeFormalParameterList(parsetree
*dt.ParseTree) (*dt.DecoratedSyntaxTree, error) {
    if parsetree.RootType != dt.FORMAL_PARAMETER_LIST_NODE {
        return nil, errors.New("expected formal parameter list")
    }

    parameters := make([]dt.DecoratedSyntaxTree, 0)
    isRef := false
```

```
    i := 0
    for i < len(parsetree.Children) {
        child := &parsetree.Children[i]
        if child.RootType == dt.TOKEN_NODE && child.TokenValue.Type ==
dt.KEYWORD && child.TokenValue.Lexeme == "variabel" {
            isRef = true
            i++
            continue
        }

        if child.RootType == dt.TOKEN_NODE {
            if child.TokenValue.Type == dt.SEMICOLON {
                isRef = false
            }
            i++
            continue
        }

        if child.RootType == dt.IDENTIFIER_LIST_NODE {
            if i+2 >= len(parsetree.Children) {
                return nil, errors.New("malformed parameter list: missing
type for identifier list")
            }

            identifierListNode := child
            typeNode := &parsetree.Children[i+2]

            identifierList, err :=
a.analyzeIdentifierList(identifierListNode)
            if err != nil {
                return nil, err
            }

            tabIndex, tabEntry, err := a.analyzeType(typeNode)
            if err != nil {
                return nil, err
            }

            for _, identifier := range identifierList {
```

```
                _, check := a.tab.FindIdentifier(identifier, a.root)
                if check != nil && check.Level == a.depth {
                    return nil, fmt.Errorf("cannot redeclare identifier '%s'
in the same scope", identifier)
                }


                var entry dt.TabEntry
                if tabIndex != -1 {
                    entry.Type = dt.TAB_ENTRY_ALIAS
                    entry.Reference = tabIndex
                } else {
                    entry.Type = tabEntry.Type
                    entry.Reference = tabEntry.Reference
                }

                entry.Identifier = identifier


                entry.Link = a.root
                entry.Object = dt.TAB_ENTRY_PARAM
                entry.Level = a.depth
                entry.Normal = !isRef
                entry.Data = a.stackSize

                paramSize := 0
                if isRef {
                    paramSize = strconv.IntSize
                } else {
                    paramSize = a.getTypeSize(semanticType{
                        StaticType: entry.Type,
                        Reference:  entry.Reference,
                    })
                }
                a.stackSize += paramSize

                a.root = len(a.tab)
                a.tab = append(a.tab, entry)

                parameters = append(parameters, dt.DecoratedSyntaxTree{
```

```
                    Property: dt.DST_PARAMETER,
                    SelfType: dt.DST_VARIABLE,
                    Data:     a.root,
                })
            }

            isRef = false
            i += 3
        } else {
            return nil, fmt.Errorf("unexpected node type '%s' in formal
parameter list at index %d", child.RootType, i)
        }
    }

    return &dt.DecoratedSyntaxTree{
        SelfType: dt.DST_VARIABLE_DECLARATIONS,
        Children: parameters,
    }, nil
}
```

Program formalparameterlist.go menganalisis daftar parameter formal dari prosedur/fungsi. Mendeteksi keyword var untuk parameter pass-by-reference dan menambahkan setiap parameter ke tabel simbol.

2.2.15. forstatement.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeForStatement(parsetree *dt.ParseTree)
(*dt.DecoratedSyntaxTree, error) {
    if parsetree.RootType != dt.FOR_STATEMENT_NODE {
        return nil, errors.New("expected for block")
    }

    target, targetType, err := a.analyzeToken(&parsetree.Children[1])
```

```
    if err != nil {
        return nil, err
    }

    if target.SelfType != dt.DST_VARIABLE {
        return nil, errors.New("expected variable in for loop assignment")
    }

    initial, initialType, err := a.analyzeExpression(&parsetree.Children[3])

    if err != nil {
        return nil, err
    }

    if !a.checkTypeEquality(targetType, initialType) {
        return nil, errors.New("assigned expression type does not match
variable type")
    }

    final, finalType, err := a.analyzeExpression(&parsetree.Children[5])

    if err != nil {
        return nil, err
    }

    if !a.checkTypeEquality(targetType, finalType) {
        return nil, errors.New("final expression type does not match
variable type")
    }

    block, err := a.analyzeStatement(&parsetree.Children[7])

    if err != nil {
        return nil, err
    }

    target.Property = dt.DST_TARGET
    initial.Property = dt.DST_VALUE
    block.Property = dt.DST_EXECUTE
```

```
    switch parsetree.Children[4].TokenValue.Lexeme {
    case "ke":
        final.Property = dt.DST_UPTO
    case "turun_ke":
        final.Property = dt.DST_DOWNTO
    default:
        return nil, errors.New("expected 'ke' or 'turun_ke'")
    }

    return &dt.DecoratedSyntaxTree{
        SelfType: dt.DST_FOR_BLOCK,
        Children: []dt.DecoratedSyntaxTree{
            *target,
            *initial,
            *final,
            *block,
        },
    }, nil
}
```

Program forstatement.go menganalisis for loop. Memproses variable loop counter, range (lower dan upper bound), dan body statement.

2.2.16. functiondeclaration.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeFunctionDeclaration(parsetree
*dt.ParseTree) (*dt.DecoratedSyntaxTree, error) {
    if parsetree.RootType != dt.FUNCTION_DECLARATION_NODE {
        return nil, errors.New("expected procedure declaration")
    }

    identifier := parsetree.Children[1].TokenValue.Lexeme
```

```
    _, check := a.tab.FindIdentifier(identifier, a.root)
    if check != nil {
        if check.Level == a.depth {
            return nil, errors.New("cannot redeclare identifier in the same
scope")
        }
    }

    tabIndex := len(a.tab)
    a.tab = append(a.tab, dt.TabEntry{
        Identifier: identifier,
        Link:       a.root,
        Object:     dt.TAB_ENTRY_FUNC,
        Level:      a.depth,
    })

    a.root = tabIndex

    root := a.root
    stackSize := a.stackSize
    a.stackSize = 0
    a.depth++

    paramSize := 0
    returnSize := 0

    var returnIndex int
    var returnEntry dt.TabEntry
    var parameters, block *dt.DecoratedSyntaxTree
    var declarations []dt.DecoratedSyntaxTree
    var err error

    for _, child := range parsetree.Children[2:] {
        switch child.RootType {
        case dt.FORMAL_PARAMETER_LIST_NODE:
            parameters, err = a.analyzeFormalParameterList(&child)
            paramSize = a.stackSize

        case dt.TYPE_NODE:
```

```
            _, returnEntry, err = a.analyzeType(&child)

            returnSize = a.getTypeSize(semanticType{
                StaticType: returnEntry.Type,
                Reference:  returnEntry.Reference,
            })

            _, check := a.tab.FindIdentifier(identifier, a.root)
            if check != nil {
                if check.Level == a.depth {
                    return nil, errors.New("cannot redeclare identifier in
the same scope")
                }
            }

            returnIndex = len(a.tab)
            a.tab = append(a.tab, dt.TabEntry{
                Identifier: identifier,
                Link:       a.root,
                Object:     dt.TAB_ENTRY_RETURN,
                Type:       returnEntry.Type,
                Reference:  returnEntry.Reference,
                Level:      a.depth,
            })

            a.tab[tabIndex].Type = returnEntry.Type
            a.tab[tabIndex].Reference = returnEntry.Reference

            a.root++

        case dt.DECLARATION_PART_NODE:
            declarations, err = a.analyzeDeclarationPart(&child)

        case dt.COMPOUND_STATEMENT_NODE:
            block, err = a.analyzeCompoundStatement(&child)
        }

        if err != nil {
            return nil, err
        }
```

```
    }

    varSize := a.stackSize - paramSize

    start := 0
    paramEnd := 0
    varEnd := 0

    for _, part := range declarations {
        if part.SelfType == dt.DST_VARIABLE_DECLARATIONS &&
len(part.Children) != 0 {
            start = part.Children[0].Data
            varEnd = part.Children[len(part.Children)-1].Data
        }
    }

    if parameters != nil {
        start = parameters.Children[0].Data
        paramEnd = parameters.Children[len(parameters.Children)-1].Data
    }

    btabIndex := len(a.btab)
    a.btab = append(a.btab, dt.BtabEntry{
        Start:        start,
        ParamEnd:     paramEnd,
        ReturnEnd:    returnIndex,
        End:          varEnd,
        ParamSize:    paramSize,
        ReturnSize:   returnSize,
        VariableSize: varSize,
    })

    a.tab[tabIndex].Data = btabIndex

    a.depth--
    a.stackSize = stackSize
    a.root = root

    var children []dt.DecoratedSyntaxTree
```

```
    if parameters != nil {
        children = []dt.DecoratedSyntaxTree{*parameters}
    } else {
        children = []dt.DecoratedSyntaxTree{}
    }

    children = append(children, declarations...)
    children = append(children, *block)

    return &dt.DecoratedSyntaxTree{
        SelfType: dt.DST_FUNCTION,
        Data:     tabIndex,
        Children: children,
    }, nil
}
```

Program functiondeclaration.go mirip dengan proceduredeclaration tetapi untuk fungsi. Tambahan menangani tipe return value dan menambahkan fungsi itu sendiri sebagai variabel lokal untuk assignment return value.

2.2.17. identifierlist.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeIdentifierList(parsetree *dt.ParseTree)
([]string, error) {
    if parsetree.RootType != dt.IDENTIFIER_LIST_NODE {
        return nil, errors.New("expected identifier list")
    }


    identifiers := make([]string, 0)

    for i := 0; i < len(parsetree.Children); i += 2 {
        if parsetree.Children[i].TokenValue != nil {
```

```
            identifiers = append(identifiers,
parsetree.Children[i].TokenValue.Lexeme)
        }
    }

    return identifiers, nil
}
```

Program identifierlist.go melakukan analisis daftar identifier (nama-nama yang dipisahkan comma). Digunakan dalam deklarasi variabel dan parameter.

2.2.18. ifstatement.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeIfStatement(parsetree *dt.ParseTree)
(*dt.DecoratedSyntaxTree, error) {
    if parsetree.RootType != dt.IF_STATEMENT_NODE {
        return nil, errors.New("expected if block")
    }

    condition, typ, err := a.analyzeExpression(&parsetree.Children[1])

    if err != nil {
        return nil, err
    }

    if typ.StaticType != dt.TAB_ENTRY_BOOLEAN {
        token := parsetree.Children[0].TokenValue
        return nil, a.newConditionTypeError(typ.StaticType.String(), token)
    }

    thenBlock, err := a.analyzeStatement(&parsetree.Children[3])

    if err != nil {
```

```
        return nil, err
    }

    elseBlock, err := a.analyzeStatement(&parsetree.Children[5])

    if err != nil {
        return nil, err
    }

    condition.Property = dt.DST_CONDITION
    thenBlock.Property = dt.DST_THEN
    elseBlock.Property = dt.DST_ELSE

    return &dt.DecoratedSyntaxTree{
        SelfType: dt.DST_IF_BLOCK,
        Children: []dt.DecoratedSyntaxTree{
            *condition,
            *thenBlock,
            *elseBlock,
        },
    }, nil
}
```

Program ifstatement.go yang menganalisis if statement. Menganalisis kondisi (harus bertipe BOOLEAN) dan kedua branch (then dan else jika ada).

2.2.19. multiplicativeoperator.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeMultiplicativeOperator(parsetree
*dt.ParseTree) (dt.DSTNodeType, error) {
    if parsetree.RootType != dt.MULTIPLICATIVE_OPERATOR_NODE {
        return dt.DST_ADD_OPERATOR, errors.New("operator is not
multiplicative")
```

```
    }

    switch parsetree.Children[0].TokenValue.Lexeme {
    case "*":
        return dt.DST_MUL_OPERATOR, nil
    case "/":
        return dt.DST_DIV_OPERATOR, nil
    case "bagi":
        return dt.DST_DIV_OPERATOR, nil
    case "mod":
        return dt.DST_MOD_OPERATOR, nil
    case "dan":
        return dt.DST_AND_OPERATOR, nil
    default:
        return dt.DST_ADD_OPERATOR, errors.New("unknown multiplicative
operator")
    }
}
```

Program multiplicativeoperator.go menangani operator multiplikatif (*, /). Melakukan type checking dan promotion.

2.2.20. parameterlist.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeParameterList(parseTree *dt.ParseTree)
([]dt.DecoratedSyntaxTree, []semanticType, error) {
    if parseTree.RootType != dt.PARAMETER_LIST_NODE {
        return nil, nil, errors.New("parse tree node is not parameter list")
    }

    parameters := parseTree.Children
    decoratedParams := make([]dt.DecoratedSyntaxTree, len(parameters)/2+1)
    paramTypes := make([]semanticType, len(parameters)/2+1)
```

```
    for i, p := range parameters {
        if i%2 == 1 {
            continue
        }

        param, typ, err := a.analyzeExpression(&p)

        if err != nil {
            return nil, nil, err
        }

        decoratedParams[i/2] = *param
        paramTypes[i/2] = typ
    }

    return decoratedParams, paramTypes, nil
}
```

Program parameterlist.go menangani daftar parameter yang diberikan saat pemanggilan subprogram (argument list), terpisah dari formal parameter list.

2.2.21. proceduredeclaration.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeProcedureDeclaration(parsetree
*dt.ParseTree) (*dt.DecoratedSyntaxTree, error) {
    if parsetree.RootType != dt.PROCEDURE_DECLARATION_NODE {
        return nil, errors.New("expected procedure declaration")
    }

    identifier := parsetree.Children[1].TokenValue.Lexeme

    _, check := a.tab.FindIdentifier(identifier, a.root)
```

```
    if check != nil {
        if check.Level == a.depth {
            return nil, errors.New("cannot redeclare identifier in the same
scope")
        }
    }

    tabIndex := len(a.tab)
    a.tab = append(a.tab, dt.TabEntry{
        Identifier: identifier,
        Link:       a.root,
        Object:     dt.TAB_ENTRY_PROC,
        Level:      a.depth,
    })

    a.root = tabIndex

    root := a.root
    stackSize := a.stackSize
    a.stackSize = 0
    a.depth++

    paramSize := 0

    var parameters, block *dt.DecoratedSyntaxTree
    var declarations []dt.DecoratedSyntaxTree
    var err error

    for _, child := range parsetree.Children[2:] {
        switch child.RootType {
        case dt.FORMAL_PARAMETER_LIST_NODE:
            parameters, err = a.analyzeFormalParameterList(&child)
            paramSize = a.stackSize
        case dt.DECLARATION_PART_NODE:
            declarations, err = a.analyzeDeclarationPart(&child)
        case dt.COMPOUND_STATEMENT_NODE:
            block, err = a.analyzeCompoundStatement(&child)
        }

        if err != nil {
```

```
            return nil, err
        }
    }

    varSize := a.stackSize - paramSize

    start := 0
    paramEnd := 0
    varEnd := 0

    for _, part := range declarations {
        if part.SelfType == dt.DST_VARIABLE_DECLARATIONS &&
len(part.Children) != 0 {
            start = part.Children[0].Data
            varEnd = part.Children[len(part.Children)-1].Data
        }
    }

    if parameters != nil {
        start = parameters.Children[0].Data
        paramEnd = parameters.Children[len(parameters.Children)-1].Data
    }

    btabIndex := len(a.btab)
    a.btab = append(a.btab, dt.BtabEntry{
        Start:        start,
        End:          varEnd,
        ParamEnd:     paramEnd,
        ParamSize:    paramSize,
        VariableSize: varSize,
    })

    a.tab[tabIndex].Data = btabIndex

    a.depth--
    a.stackSize = stackSize
    a.root = root

    var children []dt.DecoratedSyntaxTree
```

```
    if parameters != nil {
        children = []dt.DecoratedSyntaxTree{*parameters}
    } else {
        children = []dt.DecoratedSyntaxTree{}
    }

    children = append(children, declarations...)
    children = append(children, *block)

    return &dt.DecoratedSyntaxTree{
        SelfType: dt.DST_PROCEDURE,
        Data:     tabIndex,
        Children: children,
    }, nil
}
```

Program proceduredeclaration.go menganalisis deklarasi prosedur Pascal. Membuat scope baru, menambahkan parameter ke tabel simbol, menganalisis blok prosedur, kemudian keluar dari scope dengan menghapus entri dari tabel simbol.

2.2.22. program.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeProgram(parsetree *dt.ParseTree)
(*dt.DecoratedSyntaxTree, error) {
    if parsetree.RootType != dt.PROGRAM_NODE {
        return nil, errors.New("expected program")
    }

    headerIndex, _, err := a.analyzeProgramHeader(&parsetree.Children[0])

    if err != nil {
        return nil, err
    }
```

```
    declarations, err := a.analyzeDeclarationPart(&parsetree.Children[1])

    if err != nil {
        return nil, err
    }

    block, err := a.analyzeCompoundStatement(&parsetree.Children[2])

    if err != nil {
        return nil, err
    }

    return &dt.DecoratedSyntaxTree{
        Property: dt.DST_ROOT,
        SelfType: dt.DST_PROGRAM,
        Data:     headerIndex,
        Children: append(declarations, *block),
    }, nil
}
```

Program program.go menangani analisis program Pascal secara keseluruhan. Menganalisis header program, deklarasi blok, dan statement list. File ini mengatur struktur global program dan menginisialisasi scope untuk seluruh program.

2.2.23. programheader.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeProgramHeader(parsetree *dt.ParseTree)
(int, dt.TabEntry, error) {
    if parsetree.RootType != dt.PROGRAM_HEADER_NODE {
        return -1, dt.TabEntry{}, errors.New("expected program header")
    }
```

```
    identifier := parsetree.Children[1].TokenValue.Lexeme

    _, check := a.tab.FindIdentifier(identifier, a.root)
    if check != nil {
        if check.Level == a.depth {
            return -1, dt.TabEntry{}, errors.New("program name is reserved")
        }
    }

    tabIndex := len(a.tab)

    tabEntry := dt.TabEntry{
        Identifier: identifier,
        Link:       a.root,
        Object:     dt.TAB_ENTRY_PROGRAM,
        Type:       dt.TAB_ENTRY_NONE,
        Level:      a.depth,
    }

    a.tab = append(a.tab, tabEntry)

    a.root++

    return tabIndex, tabEntry, nil
}
```

Program programheader.go yang menganalisis bagian header dari program Pascal (kata kunci PROGRAM dan nama program). Menambahkan nama program ke tabel simbol dan memastikan tidak ada duplikasi nama program.

2.2.24. range.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeRange(parsetree *dt.ParseTree) (int, int,
```

```
semanticType, error) {
    if parsetree.RootType != dt.RANGE_NODE {
        return 0, 0, semanticType{}, errors.New("expected range expression")
    }

    beginExpression, beginType, err :=
a.analyzeExpression(&parsetree.Children[0])

    if err != nil {
        return 0, 0, semanticType{}, err
    }

    begin, err := a.staticEvaluate(beginExpression, beginType)

    if err != nil {
        return 0, 0, semanticType{}, err
    }

    endExpression, endType, err :=
a.analyzeExpression(&parsetree.Children[2])

    if err != nil {
        return 0, 0, semanticType{}, err
    }

    if !a.checkTypeEquality(beginType, endType) {
        return 0, 0, semanticType{}, errors.New("range expression types do
not match")
    }

    end, err := a.staticEvaluate(endExpression, endType)

    if err != nil {
        return 0, 0, semanticType{}, err
    }

    return begin, end, beginType, nil
}
```

Program range.go yang menganalisis range specification dalam for loop. Melakukan static evaluation pada lower dan upper bounds.

2.2.25. recordtype.go

```go
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeRecordType(parsetree *dt.ParseTree,
identifier string) (int, dt.TabEntry, error) {
    if parsetree.RootType != dt.RECORD_TYPE_NODE {
        return -1, dt.TabEntry{}, errors.New("expected record type")
    }

    btabIndex := len(a.btab)

    entry := dt.TabEntry{
        Identifier: identifier,
        Link:       a.root,
        Object:     dt.TAB_ENTRY_TYPE,
        Type:       dt.TAB_ENTRY_RECORD,
        Reference:  btabIndex,
        Normal:     false,
        Level:      a.depth,
    }
    a.tab = append(a.tab, entry)
    a.root = len(a.tab) - 1

    btabEntry := dt.BtabEntry{
        Start:        len(a.tab),
        ParamEnd:     0,
        ReturnEnd:    0,
        End:          0,
        ParamSize:    0,
        ReturnSize:   0,
        VariableSize: 0,
    }

    oldRoot := a.root
```

```
    oldDepth := a.depth

    a.depth++

    for _, child := range parsetree.Children {
        if child.RootType != dt.VAR_DECLARATION_NODE {
            if child.RootType == dt.TOKEN_NODE {
                continue
            }
            return -1, dt.TabEntry{}, errors.New("expected a var declaration
node for record field")
        }

        oldStackSize := a.stackSize
        a.stackSize = btabEntry.VariableSize

        _, err := a.analyzeVarDeclaration(&child)
        if err != nil {
            return -1, dt.TabEntry{}, err
        }

        for j := btabEntry.Start; j < len(a.tab); j++ {
            if a.tab[j].Object == dt.TAB_ENTRY_VAR {
                a.tab[j].Object = dt.TAB_ENTRY_FIELD
            }
        }
        btabEntry.VariableSize = a.stackSize
        a.stackSize = oldStackSize
    }

    btabEntry.End = len(a.tab) - 1

    totalSize := 0
    for i := btabEntry.Start; i < btabEntry.End; i++ {
        fieldType := semanticType{
            StaticType: a.tab[i].Type,
            Reference:  a.tab[i].Reference,
        }
        totalSize += a.getTypeSize(fieldType)
    }
```

```
    btabEntry.VariableSize = totalSize

    a.root = oldRoot
    a.depth = oldDepth

    a.btab = append(a.btab, btabEntry)

    return a.root, entry, nil
}
```

Program recordtype.go menganalisis deklarasi tipe record. Memproses field declarations dan membuat struktur tipe yang kompleks dengan sub-fields.

2.2.26. relationaloperator.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeRelationalOperator(parsetree
*dt.ParseTree) (dt.DSTNodeType, error) {
    if parsetree.RootType != dt.RELATIONAL_OPERATOR_NODE {
        return dt.DST_ADD_OPERATOR, errors.New("operator is not relational")
    }

    switch parsetree.Children[0].TokenValue.Lexeme {
    case ">":
        return dt.DST_GT_OPERATOR, nil
    case "<":
        return dt.DST_LT_OPERATOR, nil
    case ">=":
        return dt.DST_GE_OPERATOR, nil
    case "<=":
        return dt.DST_LE_OPERATOR, nil
    case "=":
        return dt.DST_EQ_OPERATOR, nil
    case "!=":
```

```
        return dt.DST_NE_OPERATOR, nil
    default:
        return dt.DST_ADD_OPERATOR, errors.New("unknown relational
operator")
    }
}
```

Program relationaloperator.go yang menangani operator relasional (<, >, =, <=, >=, <>). Menghasilkan tipe BOOLEAN.

2.2.27. simpleexpression.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeSimpleExpression(parseTree *dt.ParseTree)
(*dt.DecoratedSyntaxTree, semanticType, error) {
    beginOffset := 0
    negated := false

    if parseTree.Children[0].RootType == dt.TOKEN_NODE {
        switch parseTree.Children[0].TokenValue.Lexeme {
        case "-":
            negated = true
            fallthrough
        case "+":
            beginOffset = 1
        default:
            return nil, semanticType{}, errors.New("unknown modifier at
beginning of expression")
        }
    }

    var dst *dt.DecoratedSyntaxTree
    var typ semanticType
    var err error
```

```
    if len(parseTree.Children) == 1 {
        dst, typ, err = a.analyzeTerm(&parseTree.Children[beginOffset])
    } else {
        dst, typ, err =
a.recurseSimpleExpression(parseTree.Children[beginOffset:])
    }

    if err != nil {
        return nil, typ, err
    }

    if negated {
        switch typ.StaticType {
        case dt.TAB_ENTRY_INTEGER:
        case dt.TAB_ENTRY_REAL:
        default:
            return nil, typ, errors.New("cannot negate non numeric
expression")
        }

        dst.Property = dt.DST_OPERAND
        return &dt.DecoratedSyntaxTree{
            SelfType: dt.DST_NEG_OPERATOR,
            Children: []dt.DecoratedSyntaxTree{*dst},
        }, typ, nil
    }

    return dst, typ, nil
}

func (a *SemanticAnalyzer) recurseSimpleExpression(nodes []dt.ParseTree)
(*dt.DecoratedSyntaxTree, semanticType, error) {
    if len(nodes) == 1 {
        return a.analyzeTerm(&nodes[0])
    }

    optype, err := a.analyzeAdditiveOperator(&nodes[len(nodes)-2])

    if err != nil {
```

```
        return nil, semanticType{}, err
    }

    dst := &dt.DecoratedSyntaxTree{
        SelfType: optype,
        Children: make([]dt.DecoratedSyntaxTree, 2),
    }

    lval, ltype, err := a.recurseSimpleExpression(nodes[:len(nodes)-2])

    if err != nil {
        return nil, ltype, err
    }

    rval, rtype, err := a.analyzeTerm(&nodes[len(nodes)-1])

    if err != nil {
        return nil, rtype, err
    }

    promotedLval, promotedRval, resultType, compatible :=
a.promoteTypes(lval, ltype, rval, rtype)

    if !compatible {
        token := nodes[len(nodes)-2].Children[0].TokenValue
        return nil, ltype, a.newOperatorTypeError(
            token.Lexeme,
            ltype.StaticType.String(),
            rtype.StaticType.String(),
            token,
        )
    }

    dst.Children[0] = *promotedLval
    dst.Children[0].Property = dt.DST_OPERAND

    dst.Children[1] = *promotedRval
    dst.Children[1].Property = dt.DST_OPERAND

    return dst, resultType, nil
```

```
}
```

Program simpleexpression.go yang menganalisis simple expression yang terdiri dari terms dengan operator aditif (+, -, OR).

2.2.28. statement.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeStatement(parsetree *dt.ParseTree)
(*dt.DecoratedSyntaxTree, error) {
    switch parsetree.RootType {
    case dt.COMPOUND_STATEMENT_NODE:
        return a.analyzeCompoundStatement(parsetree)
    case dt.IF_STATEMENT_NODE:
        return a.analyzeIfStatement(parsetree)
    case dt.WHILE_STATEMENT_NODE:
        return a.analyzeWhileStatement(parsetree)
    case dt.FOR_STATEMENT_NODE:
        return a.analyzeForStatement(parsetree)
    case dt.ASSIGNMENT_STATEMENT_NODE:
        return a.analyzeAssignmentStatement(parsetree)
    case dt.SUBPROGRAM_CALL_NODE:
        dst, _, err := a.analyzeSubprogramCall(parsetree)
        return dst, err
    default:
        return nil, errors.New("unrecognized statement")
    }
}
```

Program statement.go sebagai dispatcher untuk berbagai tipe statement (assignment, if, while, for, compound, procedure call).

2.2.29. statementlist.go

```
package semantic
```

```
import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeStatementList(parsetree *dt.ParseTree)
([]dt.DecoratedSyntaxTree, error) {
    if parsetree.RootType != dt.STATEMENT_LIST_NODE {
        return nil, errors.New("expected statement list section")
    }

    decoratedStatements := make([]dt.DecoratedSyntaxTree, 0)

    for _, statement := range parsetree.Children {
        if statement.RootType == dt.TOKEN_NODE {
            continue
        }

        dst, err := a.analyzeStatement(&statement)

        if err != nil {
            return nil, err
        }

        if dst != nil {
            decoratedStatements = append(decoratedStatements, *dst)
        }
    }

    return decoratedStatements, nil
}
```

Program statementlist.go yang menganalisis daftar statement yang dipisahkan dengan semicolon. Mengiterasi children dan hanya memproses node yang merupakan statement, mengabaikan token separator.

2.2.30. staticaccess.go

```
package semantic
```

```
import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeStaticAccess(parsetree *dt.ParseTree, prev
*dt.DecoratedSyntaxTree) (*dt.DecoratedSyntaxTree, semanticType, error) {
    if parsetree.RootType != dt.STATIC_ACCESS_NODE {
        return nil, semanticType{}, errors.New("expected static access")
    }

    nodes := make([]dt.ParseTree, 0)
    for i := 0; i < len(parsetree.Children); i += 2 {
        nodes = append(nodes, parsetree.Children[i])
    }

    root := a.root

    var typ semanticType
    var err error

    if prev != nil {
        typeIndex := prev.Data

        for a.tab[typeIndex].Type == dt.TAB_ENTRY_ALIAS {
            typeIndex = a.tab[typeIndex].Reference
        }

        if a.tab[typeIndex].Type != dt.TAB_ENTRY_RECORD {
            return nil, semanticType{}, errors.New("cannot access field of
non struct object")
        }

        root = a.btab[a.tab[typeIndex].Reference].End
    }

    switch nodes[0].RootType {
    case dt.TOKEN_NODE:
```

```
        tabIndex, tabEntry :=
a.tab.FindIdentifier(nodes[0].TokenValue.Lexeme, root)
        if tabIndex == -1 {
            return nil, semanticType{}, errors.New("undeclared identifier")
        }

        typ = semanticType{
            StaticType: tabEntry.Type,
            Reference:  tabEntry.Reference,
        }

        if prev != nil {
            prev.Property = dt.DST_FROM
            prev = &dt.DecoratedSyntaxTree{
                SelfType: dt.DST_RECORD_FIELD,
                Data:     tabIndex,
                Children: []dt.DecoratedSyntaxTree{*prev},
            }
        } else {
            var dstType dt.DSTNodeType

            switch a.tab[tabIndex].Object {
            case dt.TAB_ENTRY_CONST:
                dstType = dt.DST_CONST
            case dt.TAB_ENTRY_PARAM:
                fallthrough
            case dt.TAB_ENTRY_RETURN:
                fallthrough
            case dt.TAB_ENTRY_VAR:
                dstType = dt.DST_VARIABLE
            default:
                return nil, semanticType{}, errors.New("unexpected object")
            }

            prev = &dt.DecoratedSyntaxTree{
                SelfType: dstType,
                Data:     tabIndex,
            }
        }
    case dt.ARRAY_ACCESS_NODE:
```

```
        prev, typ, err = a.analyzeArrayAccess(&nodes[0], prev)
        if err != nil {
            return nil, semanticType{}, err
        }
    default:
        return nil, semanticType{}, errors.New("found unexpected node")
    }

    for _, node := range nodes[1:] {
        typeIndex := prev.Data

        for a.tab[typeIndex].Type == dt.TAB_ENTRY_ALIAS {
            typeIndex = a.tab[typeIndex].Reference
        }

        if a.tab[typeIndex].Type != dt.TAB_ENTRY_RECORD {
            return nil, semanticType{}, errors.New("cannot access field of
non struct object")
        }

        root = a.btab[a.tab[typeIndex].Reference].End

        switch node.RootType {
        case dt.TOKEN_NODE:
            tabIndex, tabEntry :=
a.tab.FindIdentifier(node.TokenValue.Lexeme, root)
            if tabIndex == -1 {
                return nil, semanticType{}, errors.New("undeclared
identifier")
            }

            typ = semanticType{
                StaticType: tabEntry.Type,
                Reference:  tabEntry.Reference,
            }

            prev.Property = dt.DST_FROM
            prev = &dt.DecoratedSyntaxTree{
                SelfType: dt.DST_RECORD_FIELD,
                Data:     tabIndex,
```

```
                Children: []dt.DecoratedSyntaxTree{*prev},
            }
        case dt.ARRAY_ACCESS_NODE:
            prev, typ, err = a.analyzeArrayAccess(&node, prev)
            if err != nil {
                return nil, semanticType{}, err
            }
        default:
            return nil, semanticType{}, errors.New("found unexpected node")
        }
    }

    return prev, typ, nil
}
```

Program staticaccess.go menganalisis static access (identifier atau record field access seperti record.field). Melakukan traversal melalui record structure untuk tipe yang kompleks.

2.2.31. staticevaluator.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)



func (a *SemanticAnalyzer) staticEvaluate(dst *dt.DecoratedSyntaxTree, typ
semanticType) (int, error) {
    switch typ.StaticType {
    case dt.TAB_ENTRY_INTEGER:
        return a.staticEvaluateInt(dst)
    case dt.TAB_ENTRY_REAL:
        return a.staticEvaluateReal(dst)
    case dt.TAB_ENTRY_CHAR:
        return a.staticEvaluateChar(dst)
    case dt.TAB_ENTRY_BOOLEAN:
        return a.staticEvaluateBool(dst)
```

```
    default:
        return -1, errors.New("cannot evaluate expression statically")
    }
}

func (a *SemanticAnalyzer) staticEvaluateInt(dst *dt.DecoratedSyntaxTree)
(int, error) {
    switch dst.SelfType {
    case dt.DST_INT_LITERAL:
        return dst.Data, nil
    case dt.DST_CONST:
        constEntry := a.tab[dst.Data]
        if constEntry.Type != dt.TAB_ENTRY_INTEGER {
            return 0, errors.New("constant is not an integer")
        }
        return constEntry.Data, nil
    case dt.DST_ADD_OPERATOR:
        left, err := a.staticEvaluateInt(&dst.Children[0])
        if err != nil {
            return 0, err
        }
        right, err := a.staticEvaluateInt(&dst.Children[1])
        if err != nil {
            return 0, err
        }
        return left + right, nil
    case dt.DST_SUB_OPERATOR:
        left, err := a.staticEvaluateInt(&dst.Children[0])
        if err != nil {
            return 0, err
        }
        right, err := a.staticEvaluateInt(&dst.Children[1])
        if err != nil {
            return 0, err
        }
        return left - right, nil
    case dt.DST_MUL_OPERATOR:
        left, err := a.staticEvaluateInt(&dst.Children[0])
        if err != nil {
            return 0, err
```

```
        }
        right, err := a.staticEvaluateInt(&dst.Children[1])
        if err != nil {
            return 0, err
        }
        return left * right, nil
    case dt.DST_DIV_OPERATOR:
        left, err := a.staticEvaluateInt(&dst.Children[0])
        if err != nil {
            return 0, err
        }
        right, err := a.staticEvaluateInt(&dst.Children[1])
        if err != nil {
            return 0, err
        }
        if right == 0 {
            return 0, errors.New("division by zero in static evaluation")
        }
        return left / right, nil
    case dt.DST_MOD_OPERATOR:
        left, err := a.staticEvaluateInt(&dst.Children[0])
        if err != nil {
            return 0, err
        }
        right, err := a.staticEvaluateInt(&dst.Children[1])
        if err != nil {
            return 0, err
        }
        if right == 0 {
            return 0, errors.New("modulo by zero in static evaluation")
        }
        return left % right, nil
    case dt.DST_NEG_OPERATOR:
        val, err := a.staticEvaluateInt(&dst.Children[0])
        if err != nil {
            return 0, err
        }
        return -val, nil
    default:
        return 0, errors.New("cannot statically evaluate integer
```

```
expression")
    }
}

func (a *SemanticAnalyzer) staticEvaluateReal(dst *dt.DecoratedSyntaxTree)
(int, error) {
    switch dst.SelfType {
    case dt.DST_REAL_LITERAL:
        return dst.Data, nil
    case dt.DST_CONST:
        constEntry := a.tab[dst.Data]
        if constEntry.Type != dt.TAB_ENTRY_REAL {
            return 0, errors.New("constant is not a real")
        }
        return constEntry.Data, nil
    default:
        return 0, errors.New("cannot statically evaluate real expression")
    }
}

func (a *SemanticAnalyzer) staticEvaluateChar(dst *dt.DecoratedSyntaxTree)
(int, error) {
    switch dst.SelfType {
    case dt.DST_CHAR_LITERAL:
        return dst.Data, nil
    case dt.DST_CONST:
        constEntry := a.tab[dst.Data]
        if constEntry.Type != dt.TAB_ENTRY_CHAR {
            return 0, errors.New("constant is not a char")
        }
        return constEntry.Data, nil
    default:
        return 0, errors.New("cannot statically evaluate char expression")
    }
}

func (a *SemanticAnalyzer) staticEvaluateBool(dst *dt.DecoratedSyntaxTree)
(int, error) {
    switch dst.SelfType {
    case dt.DST_BOOL_LITERAL:
```

```
        return dst.Data, nil
    case dt.DST_CONST:
        constEntry := a.tab[dst.Data]
        if constEntry.Type != dt.TAB_ENTRY_BOOLEAN {
            return 0, errors.New("constant is not a boolean")
        }
        return constEntry.Data, nil
    case dt.DST_AND_OPERATOR:
        left, err := a.staticEvaluateBool(&dst.Children[0])
        if err != nil {
            return 0, err
        }
        right, err := a.staticEvaluateBool(&dst.Children[1])
        if err != nil {
            return 0, err
        }
        if left != 0 && right != 0 {
            return 1, nil
        }
        return 0, nil
    case dt.DST_OR_OPERATOR:
        left, err := a.staticEvaluateBool(&dst.Children[0])
        if err != nil {
            return 0, err
        }
        right, err := a.staticEvaluateBool(&dst.Children[1])
        if err != nil {
            return 0, err
        }
        if left != 0 || right != 0 {
            return 1, nil
        }
        return 0, nil
    case dt.DST_NOT_OPERATOR:
        val, err := a.staticEvaluateBool(&dst.Children[0])
        if err != nil {
            return 0, err
        }
        if val == 0 {
            return 1, nil
```

```
        }
        return 0, nil
    default:
        return 0, errors.New("cannot statically evaluate boolean
expression")
    }
}
```

Program staticevaluator.go melakukan static evaluation pada ekspresi konstanta untuk mendapatkan nilai compile-time. Digunakan untuk range bounds dan konstanta.

2.2.32. subprogramcall.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeSubprogramCall(parseTree *dt.ParseTree)
(*dt.DecoratedSyntaxTree, semanticType, error) {
    if parseTree.RootType != dt.SUBPROGRAM_CALL_NODE {
        return nil, semanticType{}, errors.New("parse tree node is not
subprogram call node")
    }

    subprogramIdentifier := parseTree.Children[0].TokenValue.Lexeme
    index, tabEntry := a.tab.FindIdentifier(subprogramIdentifier, a.root)

    if tabEntry == nil {
        token := parseTree.Children[0].TokenValue
        return nil, semanticType{},
a.newUndeclaredIdentError(subprogramIdentifier, token)
    }

    var callType dt.DSTNodeType

    switch tabEntry.Object {
    case dt.TAB_ENTRY_FUNC:
```

```
        callType = dt.DST_FUNCTION_CALL
    case dt.TAB_ENTRY_PROC:
        callType = dt.DST_PROCEDURE_CALL
    default:
        token := parseTree.Children[0].TokenValue
        return nil, semanticType{}, a.newNotCallableError(
            subprogramIdentifier,
            tabEntry.Object.String(),
            token,
        )
    }

    btabEntry := a.btab[tabEntry.Data]
    paramStart := btabEntry.Start
    paramEnd := btabEntry.ParamEnd

    callParams, callTypes, err :=
a.analyzeParameterList(&parseTree.Children[2])

    if err != nil {
        return nil, semanticType{}, err
    }

    if len(callParams) != (paramEnd - paramStart + 1) {
        token := parseTree.Children[0].TokenValue
        return nil, semanticType{}, a.newParameterCountError(
            paramEnd-paramStart+1,
            len(callParams),
            subprogramIdentifier,
            token,
        )
    }

    dst := &dt.DecoratedSyntaxTree{
        SelfType: callType,
        Data:     index,
        Children: make([]dt.DecoratedSyntaxTree, len(callParams)),
    }

    for i := paramStart; i < paramEnd+1; i++ {
```

```
        declaredType := semanticType{
            StaticType: a.tab[i].Type,
            Reference:  a.tab[i].Reference,
        }

        if !a.checkTypeEquality(declaredType, callTypes[i-paramStart]) {
            token := parseTree.Children[0].TokenValue
            return nil, semanticType{}, a.newParameterTypeError(
                i-paramStart,
                declaredType.StaticType.String(),
                callTypes[i-paramStart].StaticType.String(),
                subprogramIdentifier,
                token,
            )
        }

        dst.Children[i-paramStart] = callParams[i-paramStart]
    }

    return dst, semanticType{
        StaticType: tabEntry.Type,
        Reference:  tabEntry.Reference,
    }, nil
}
```

Program subprogramcall.go menganalisis pemanggilan prosedur atau fungsi. Memvalidasi argument count dan type terhadap formal parameters.

2.2.33. subprogramdeclaration.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeSubprogramDeclaration(parsetree
*dt.ParseTree) (*dt.DecoratedSyntaxTree, error) {
    if parsetree.RootType != dt.SUBPROGRAM_DECLARATION_NODE {
```

```
        return nil, errors.New("subprogram declaration expected")
    }

    child := parsetree.Children[0]

    switch child.RootType {
    case dt.PROCEDURE_DECLARATION_NODE:
        return a.analyzeProcedureDeclaration(&child)
    case dt.FUNCTION_DECLARATION_NODE:
        return a.analyzeFunctionDeclaration(&child)
    default:
        return nil, errors.New("expected procedure or function declaration")
    }
}
```

Program subprogramdeclaration.go yang mengelola deklarasi subprogram (prosedur dan fungsi) secara keseluruhan, melakukan dispatch ke proceduredeclaration atau functiondeclaration.

2.2.34. term.go

```
package semantic

import (
    "errors"
    "math"
    "strconv"
    "strings"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeToken(parsetree *dt.ParseTree)
(*dt.DecoratedSyntaxTree, semanticType, error) {
    if parsetree.RootType != dt.TOKEN_NODE {
        return nil, semanticType{}, errors.New("parsetree root is not token
node")
    }

    switch parsetree.TokenValue.Type {
    case dt.CHAR_LITERAL:
```

```
        return &dt.DecoratedSyntaxTree{
            SelfType: dt.DST_CHAR_LITERAL,
            Data:     int(parsetree.TokenValue.Lexeme[0]),
        }, semanticType{StaticType: dt.TAB_ENTRY_CHAR}, nil

    case dt.STRING_LITERAL:
        value := parsetree.TokenValue.Lexeme[1 :
len(parsetree.TokenValue.Lexeme)-1]

        stridx, _ := a.strtab.FindString(value)

        if stridx == -1 {
            stridx = len(a.strtab)
            a.strtab = append(a.strtab, dt.StrTabEntry{
                Length: len(parsetree.TokenValue.Lexeme),
                String: parsetree.TokenValue.Lexeme,
            })
        }

        return &dt.DecoratedSyntaxTree{
                SelfType: dt.DST_STR_LITERAL,
                Data:     stridx,
            }, semanticType{
                StaticType: dt.TAB_ENTRY_ALIAS,
                Reference:  0,
            }, nil

    case dt.NUMBER:
        if strings.ContainsAny(parsetree.TokenValue.Lexeme, "e") {
            parts := strings.Split(parsetree.TokenValue.Lexeme, "e")

            if len(parts) != 2 {
                return nil, semanticType{}, errors.New("unexpected error
reading float")
            }

            significant, err := strconv.ParseFloat(parts[0], 64)

            if err != nil {
                return nil, semanticType{}, errors.New("unexpected error
```

```
reading float")
            }

            exponent, err := strconv.ParseFloat(parts[1], 64)

            if err != nil {
                return nil, semanticType{}, errors.New("unexpected error
reading float")
            }

            rawValue := significant * math.Pow(10, exponent)
            var data int

            if strconv.IntSize == 32 {
                data = int(math.Float32bits(float32(rawValue)))
            } else {
                data = int(math.Float64bits(float64(rawValue)))
            }

            return &dt.DecoratedSyntaxTree{
                SelfType: dt.DST_REAL_LITERAL,
                Data:     data,
            }, semanticType{StaticType: dt.TAB_ENTRY_REAL}, nil
        } else if strings.ContainsAny(parsetree.TokenValue.Lexeme, ".") {
            val, err := strconv.ParseFloat(parsetree.TokenValue.Lexeme,
strconv.IntSize)

            if err != nil {
                return nil, semanticType{}, errors.New("unexpected error
reading float")
            }

            var data int

            if strconv.IntSize == 32 {
                data = int(math.Float32bits(float32(val)))
            } else {
                data = int(math.Float64bits(val))
            }
```

```
            return &dt.DecoratedSyntaxTree{
                SelfType: dt.DST_REAL_LITERAL,
                Data:     data,
            }, semanticType{StaticType: dt.TAB_ENTRY_REAL}, nil
        } else {
            val, err := strconv.ParseInt(parsetree.TokenValue.Lexeme, 10,
64)

            if err != nil {
                return nil, semanticType{}, errors.New("unexpected error
reading integer")
            }

            return &dt.DecoratedSyntaxTree{
                SelfType: dt.DST_INT_LITERAL,
                Data:     int(val),
            }, semanticType{StaticType: dt.TAB_ENTRY_INTEGER}, nil
        }

    case dt.KEYWORD:
        switch parsetree.TokenValue.Lexeme {
        case "true":
            return &dt.DecoratedSyntaxTree{
                SelfType: dt.DST_BOOL_LITERAL,
                Data:     1,
            }, semanticType{StaticType: dt.TAB_ENTRY_BOOLEAN}, nil
        case "false":
            return &dt.DecoratedSyntaxTree{
                SelfType: dt.DST_BOOL_LITERAL,
                Data:     0,
            }, semanticType{StaticType: dt.TAB_ENTRY_BOOLEAN}, nil
        default:
            return nil, semanticType{}, errors.New("unexpected keyword")
        }

    case dt.IDENTIFIER:
        index, tabEntry := a.tab.FindIdentifier(parsetree.TokenValue.Lexeme,
a.root)

        if tabEntry == nil {
```

```
            return nil, semanticType{}, a.newUndeclaredIdentError(
                parsetree.TokenValue.Lexeme,
                parsetree.TokenValue,
            )
        }

        if tabEntry.Object != dt.TAB_ENTRY_VAR {
            return nil, semanticType{}, a.newInvalidTypeError(
                parsetree.TokenValue.Lexeme,
                "variable",
                tabEntry.Object.String(),
                parsetree.TokenValue,
            )
        }

        return &dt.DecoratedSyntaxTree{
                SelfType: dt.DST_VARIABLE,
                Data:     index,
            }, semanticType{
                StaticType: tabEntry.Type,
                Reference:  tabEntry.Reference,
            }, nil
    }

    return nil, semanticType{}, errors.New("unexpected token type")
}
```

Program term.go yang menganalisis term yang terdiri dari factors dengan operator multiplikatif (*, /, DIV, MOD, AND).

2.2.35. token.go

```
package semantic

import (
    "errors"
    "math"
    "strconv"
    "strings"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
```

```
)

func (a *SemanticAnalyzer) analyzeToken(parsetree *dt.ParseTree)
(*dt.DecoratedSyntaxTree, semanticType, error) {
    if parsetree.RootType != dt.TOKEN_NODE {
        return nil, semanticType{}, errors.New("parsetree root is not token
node")
    }

    switch parsetree.TokenValue.Type {
    case dt.CHAR_LITERAL:
        return &dt.DecoratedSyntaxTree{
            SelfType: dt.DST_CHAR_LITERAL,
            Data:     int(parsetree.TokenValue.Lexeme[0]),
        }, semanticType{StaticType: dt.TAB_ENTRY_CHAR}, nil

    case dt.STRING_LITERAL:
        value := parsetree.TokenValue.Lexeme[1 :
len(parsetree.TokenValue.Lexeme)-1]

        stridx, _ := a.strtab.FindString(value)

        if stridx == -1 {
            stridx = len(a.strtab)
            a.strtab = append(a.strtab, dt.StrTabEntry{
                Length: len(parsetree.TokenValue.Lexeme),
                String: parsetree.TokenValue.Lexeme,
            })
        }

        return &dt.DecoratedSyntaxTree{
                SelfType: dt.DST_STR_LITERAL,
                Data:     stridx,
            }, semanticType{
                StaticType: dt.TAB_ENTRY_ALIAS,
                Reference:  0,
            }, nil

    case dt.NUMBER:
        if strings.ContainsAny(parsetree.TokenValue.Lexeme, "e") {
```

```
            parts := strings.Split(parsetree.TokenValue.Lexeme, "e")

            if len(parts) != 2 {
                return nil, semanticType{}, errors.New("unexpected error
reading float")
            }

            significant, err := strconv.ParseFloat(parts[0], 64)

            if err != nil {
                return nil, semanticType{}, errors.New("unexpected error
reading float")
            }

            exponent, err := strconv.ParseFloat(parts[1], 64)

            if err != nil {
                return nil, semanticType{}, errors.New("unexpected error
reading float")
            }

            rawValue := significant * math.Pow(10, exponent)
            var data int

            if strconv.IntSize == 32 {
                data = int(math.Float32bits(float32(rawValue)))
            } else {
                data = int(math.Float64bits(float64(rawValue)))
            }

            return &dt.DecoratedSyntaxTree{
                SelfType: dt.DST_REAL_LITERAL,
                Data:     data,
            }, semanticType{StaticType: dt.TAB_ENTRY_REAL}, nil
        } else if strings.ContainsAny(parsetree.TokenValue.Lexeme, ".") {
            val, err := strconv.ParseFloat(parsetree.TokenValue.Lexeme,
strconv.IntSize)

            if err != nil {
                return nil, semanticType{}, errors.New("unexpected error
```

```
reading float")
            }

            var data int

            if strconv.IntSize == 32 {
                data = int(math.Float32bits(float32(val)))
            } else {
                data = int(math.Float64bits(val))
            }

            return &dt.DecoratedSyntaxTree{
                SelfType: dt.DST_REAL_LITERAL,
                Data:     data,
            }, semanticType{StaticType: dt.TAB_ENTRY_REAL}, nil
        } else {
            val, err := strconv.ParseInt(parsetree.TokenValue.Lexeme, 10,
64)

            if err != nil {
                return nil, semanticType{}, errors.New("unexpected error
reading integer")
            }

            return &dt.DecoratedSyntaxTree{
                SelfType: dt.DST_INT_LITERAL,
                Data:     int(val),
            }, semanticType{StaticType: dt.TAB_ENTRY_INTEGER}, nil
        }

    case dt.KEYWORD:
        switch parsetree.TokenValue.Lexeme {
        case "true":
            return &dt.DecoratedSyntaxTree{
                SelfType: dt.DST_BOOL_LITERAL,
                Data:     1,
            }, semanticType{StaticType: dt.TAB_ENTRY_BOOLEAN}, nil
        case "false":
            return &dt.DecoratedSyntaxTree{
                SelfType: dt.DST_BOOL_LITERAL,
```

```
                Data:     0,
            }, semanticType{StaticType: dt.TAB_ENTRY_BOOLEAN}, nil
        default:
            return nil, semanticType{}, errors.New("unexpected keyword")
        }

    case dt.IDENTIFIER:
        index, tabEntry := a.tab.FindIdentifier(parsetree.TokenValue.Lexeme,
a.root)

        if tabEntry == nil {
            return nil, semanticType{}, a.newUndeclaredIdentError(
                parsetree.TokenValue.Lexeme,
                parsetree.TokenValue,
            )
        }

        if tabEntry.Object != dt.TAB_ENTRY_VAR {
            return nil, semanticType{}, a.newInvalidTypeError(
                parsetree.TokenValue.Lexeme,
                "variable",
                tabEntry.Object.String(),
                parsetree.TokenValue,
            )
        }

        return &dt.DecoratedSyntaxTree{
                SelfType: dt.DST_VARIABLE,
                Data:     index,
            }, semanticType{
                StaticType: tabEntry.Type,
                Reference:  tabEntry.Reference,
            }, nil
    }

    return nil, semanticType{}, errors.New("unexpected token type")
}
```

Program token.go adalah utility functions untuk handling token-level operations dalam konteks semantic analysis.

2.2.36. type.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeType(parsetree *dt.ParseTree) (int,
dt.TabEntry, error) {
    if parsetree.RootType != dt.TYPE_NODE {
        return -1, dt.TabEntry{}, errors.New("expected type")
    }

    child := parsetree.Children[0]

    switch child.RootType {
    case dt.TOKEN_NODE:
        if child.TokenValue.Type == dt.IDENTIFIER {
            index, tabEntry := a.tab.FindIdentifier(child.TokenValue.Lexeme,
a.root)

            if tabEntry == nil {
                token := child.TokenValue
                return -1, dt.TabEntry{},
a.newUndeclaredIdentError(child.TokenValue.Lexeme, token)
            }

            if tabEntry.Object != dt.TAB_ENTRY_TYPE {
                token := child.TokenValue
                return -1, dt.TabEntry{}, a.newInvalidTypeError(
                    child.TokenValue.Lexeme,
                    "type",
                    tabEntry.Object.String(),
                    token,
                )
            }

            return index, *tabEntry, nil
```

```
        } else {
            switch child.TokenValue.Lexeme {
            case "integer":
                return -1, dt.TabEntry{Type: dt.TAB_ENTRY_INTEGER}, nil
            case "real":
                return -1, dt.TabEntry{Type: dt.TAB_ENTRY_REAL}, nil
            case "boolean":
                return -1, dt.TabEntry{Type: dt.TAB_ENTRY_BOOLEAN}, nil
            case "char":
                return -1, dt.TabEntry{Type: dt.TAB_ENTRY_CHAR}, nil
            default:
                return -1, dt.TabEntry{}, errors.New("unrecognized type
declaration")
            }
        }
    case dt.ARRAY_TYPE_NODE:
        return a.analyzeArrayType(&child)
    default:
        return -1, dt.TabEntry{}, errors.New("unrecognized type
declaration")
    }
}
```

Program type.go menganalisis tipe specification dan melakukan lookup tipe di atab.

2.2.37. typecompat.go

```
package semantic

import (
    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) checkTypeCompatibility(t1 semanticType, t2
semanticType) bool {
    if a.checkTypeEquality(t1, t2) {
        return true
    }

    resolved1 := a.resolveAliasType(t1)
    resolved2 := a.resolveAliasType(t2)
```

```
    if resolved1.StaticType == dt.TAB_ENTRY_REAL && resolved2.StaticType ==
dt.TAB_ENTRY_INTEGER {
        return true
    }
    if resolved1.StaticType == dt.TAB_ENTRY_INTEGER && resolved2.StaticType
== dt.TAB_ENTRY_REAL {
        return true
    }

    if resolved1.StaticType == dt.TAB_ENTRY_CHAR && resolved2.StaticType ==
dt.TAB_ENTRY_CHAR {
        return true
    }

    if resolved1.StaticType == dt.TAB_ENTRY_BOOLEAN && resolved2.StaticType
== dt.TAB_ENTRY_BOOLEAN {
        return true
    }

    return false
}


func (a *SemanticAnalyzer) insertImplicitCast(dst *dt.DecoratedSyntaxTree,
fromType semanticType, toType semanticType) (*dt.DecoratedSyntaxTree,
semanticType) {
    if a.checkTypeEquality(fromType, toType) {
        return dst, toType
    }

    resolvedFrom := a.resolveAliasType(fromType)
    resolvedTo := a.resolveAliasType(toType)

    if resolvedFrom.StaticType == dt.TAB_ENTRY_INTEGER &&
resolvedTo.StaticType == dt.TAB_ENTRY_REAL {
        return &dt.DecoratedSyntaxTree{
            SelfType: dt.DST_CAST_OPERATOR,
            Data:     int(resolvedTo.StaticType),
            Children: []dt.DecoratedSyntaxTree{*dst},
```

```
        }, toType
    }

    return dst, toType
}

func (a *SemanticAnalyzer) canCastImplicitly(fromType semanticType, toType
semanticType) bool {
    if a.checkTypeEquality(fromType, toType) {
        return true
    }

    resolvedFrom := a.resolveAliasType(fromType)
    resolvedTo := a.resolveAliasType(toType)

    if resolvedFrom.StaticType == dt.TAB_ENTRY_INTEGER &&
resolvedTo.StaticType == dt.TAB_ENTRY_REAL {
        return true
    }

    return false
}

func (a *SemanticAnalyzer) promoteTypes(dst1 *dt.DecoratedSyntaxTree, type1
semanticType, dst2 *dt.DecoratedSyntaxTree, type2 semanticType)
(*dt.DecoratedSyntaxTree, *dt.DecoratedSyntaxTree, semanticType, bool) {
    if a.checkTypeEquality(type1, type2) {
        return dst1, dst2, type1, true
    }


    resolved1 := a.resolveAliasType(type1)
    resolved2 := a.resolveAliasType(type2)

    if resolved1.StaticType == dt.TAB_ENTRY_INTEGER && resolved2.StaticType
== dt.TAB_ENTRY_REAL {
        promoted2, _ := a.insertImplicitCast(dst2, type2, type1)
        return dst1, promoted2, type1, true
    }
```

```
    if resolved1.StaticType == dt.TAB_ENTRY_REAL && resolved2.StaticType ==
dt.TAB_ENTRY_INTEGER {
        promoted2, _ := a.insertImplicitCast(dst2, type2, type1)
        return dst1, promoted2, type1, true
    }

    if resolved1.StaticType == dt.TAB_ENTRY_CHAR && resolved2.StaticType ==
dt.TAB_ENTRY_CHAR {
        return dst1, dst2, type1, true
    }

    if resolved1.StaticType == dt.TAB_ENTRY_BOOLEAN && resolved2.StaticType
== dt.TAB_ENTRY_BOOLEAN {
        return dst1, dst2, type1, true
    }


    return dst1, dst2, type1, false
}
```

Program typecompat.go mengimplementasikan pemeriksaan type compatibility dan implicit casting rules. Menentukan apakah dua tipe dapat dicast secara implicit (misal: INTEGER ke REAL).

2.2.38. typedeclaration.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeTypeDeclaration(parsetree *dt.ParseTree)
(*dt.DecoratedSyntaxTree, error) {
    if parsetree.RootType != dt.TYPE_DECLARATION_NODE {
        return nil, errors.New("expected type declaration")
    }

    identifier := parsetree.Children[0].TokenValue.Lexeme
```

```
    _, prev := a.tab.FindIdentifier(identifier, a.root)

    if prev != nil {
        if prev.Level == a.depth {
            token := parsetree.Children[0].TokenValue
            return nil, a.newRedeclarationError(identifier, token)
        }
    }

    var tabIndex int
    var tabEntry dt.TabEntry
    var err error
    if parsetree.Children[2].RootType == dt.RECORD_TYPE_NODE {
        _, _, err = a.analyzeRecordType(&parsetree.Children[2], identifier)
        if err != nil {
            return nil, err
        }

        return &dt.DecoratedSyntaxTree{
            SelfType: dt.DST_TYPE,
            Data:     a.root,
        }, nil
    } else {
        tabIndex, tabEntry, err = a.analyzeType(&parsetree.Children[2])
        if tabIndex != -1 {
            tabEntry = dt.TabEntry{
                Type:      dt.TAB_ENTRY_ALIAS,
                Reference: tabIndex,
            }
        }

        tabEntry.Identifier = identifier
        tabEntry.Object = dt.TAB_ENTRY_TYPE
        tabEntry.Level = a.depth
        tabEntry.Link = a.root
        if err != nil {
            return nil, err
        }

        a.root = len(a.tab)
```

```
        a.tab = append(a.tab, tabEntry)

        return &dt.DecoratedSyntaxTree{
            SelfType: dt.DST_TYPE,
            Data:     a.root,
        }, nil
    }

}
```

Program typedeclaration.go menganalisis deklarasi tipe user-defined. Menambahkan tipe baru ke atab (array type table) dan memastikan tidak ada redeclaration dalam scope yang sama.

2.2.39. typedeclarationpart.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeTypeDeclarationPart(parsetree
*dt.ParseTree) (*dt.DecoratedSyntaxTree, error) {
    if parsetree.RootType != dt.TYPE_DECLARATION_PART_NODE {
        return nil, errors.New("expected type declaration part")
    }

    declarations := make([]dt.DecoratedSyntaxTree,
len(parsetree.Children)-1)

    for i, typeDeclaration := range parsetree.Children[1:] {
        declaration, err := a.analyzeTypeDeclaration(&typeDeclaration)

        if err != nil {
            return nil, err
        }

        declarations[i] = *declaration
    }
```

```
    return &dt.DecoratedSyntaxTree{
        SelfType: dt.DST_TYPE_DECLARATIONS,
        Children: declarations,
    }, nil
}
```

Program typedeclarationpart.go bertugas untuk mengelola bagian deklarasi tipe secara keseluruhan, similar dengan vardeclarationpart.

2.2.40. vardeclaration.go

```
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeVarDeclaration(parsetree *dt.ParseTree)
([]dt.DecoratedSyntaxTree, error) {
    if parsetree.RootType != dt.VAR_DECLARATION_NODE {
        return nil, errors.New("expected var declaration")
    }

    identifiers, err := a.analyzeIdentifierList(&parsetree.Children[0])

    if err != nil {
        return nil, err
    }

    tabIndex, tabEntry, err := a.analyzeType(&parsetree.Children[2])

    if err != nil {
        return nil, err
    }

    declarations := make([]dt.DecoratedSyntaxTree, len(identifiers))

    for i, identifier := range identifiers {
```

```
        _, check := a.tab.FindIdentifier(identifier, a.root)
        if check != nil {
            if check.Level == a.depth {
                return nil, errors.New("cannot redeclare identifier in the
same scope")
            }
        }

        if tabIndex != -1 {
            tabEntry = dt.TabEntry{
                Type:      dt.TAB_ENTRY_ALIAS,
                Reference: tabIndex,
            }
        }

        tabEntry.Identifier = identifier
        tabEntry.Link = a.root
        tabEntry.Object = dt.TAB_ENTRY_VAR
        tabEntry.Level = a.depth
        tabEntry.Data = a.stackSize

        a.stackSize += a.getTypeSize(semanticType{
            StaticType: tabEntry.Type,
            Reference:  tabEntry.Reference,
        })

        a.root = len(a.tab)
        a.tab = append(a.tab, tabEntry)

        declarations[i] = dt.DecoratedSyntaxTree{
            SelfType: dt.DST_VARIABLE,
            Data:     a.root,
        }
    }

    return declarations, nil
}
```

Program vardeclaration.go melakukan analisis deklarasi variabel (VAR). Memproses identifier list dan tipe variabel, kemudian menambahkan setiap variabel ke tabel simbol dengan link ke scope sebelumnya.

2.2.41. vardeclarationpart.go

```go
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeVarDeclarationPart(parsetree
*dt.ParseTree) (*dt.DecoratedSyntaxTree, error) {
    if parsetree.RootType != dt.VAR_DECLARATION_PART_NODE {
        return nil, errors.New("expected var declaration part")
    }

    declarationNodes := parsetree.Children[1:]
    declarations := make([]dt.DecoratedSyntaxTree, 0)

    for _, node := range declarationNodes {
        partialDeclarations, err := a.analyzeVarDeclaration(&node)

        if err != nil {
            return nil, err
        }

        declarations = append(declarations, partialDeclarations...)
    }

    for i := range declarations {
        declarations[i].Property = dt.DST_DECLARE
    }

    return &dt.DecoratedSyntaxTree{
        SelfType: dt.DST_VARIABLE_DECLARATIONS,
        Children: declarations,
    }, nil
}
```

Program vardeclarationpart.go menangani bagian deklarasi variabel secara keseluruhan, mengiterasi melalui multiple variable declaration groups yang dipisahkan dengan semicolon.

2.2.42. *whilestatement.go*

```go
package semantic

import (
    "errors"

    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
)

func (a *SemanticAnalyzer) analyzeWhileStatement(parsetree *dt.ParseTree)
(*dt.DecoratedSyntaxTree, error) {
    if parsetree.RootType != dt.WHILE_STATEMENT_NODE {
        return nil, errors.New("expected while block")
    }

    condition, typ, err := a.analyzeExpression(&parsetree.Children[1])

    if err != nil {
        return nil, err
    }

    if typ.StaticType != dt.TAB_ENTRY_BOOLEAN {
        token := parsetree.Children[0].TokenValue
        return nil, a.newConditionTypeError(typ.StaticType.String(), token)
    }

    block, err := a.analyzeStatement(&parsetree.Children[3])

    if err != nil {
        return nil, err
    }

    condition.Property = dt.DST_CONDITION
    block.Property = dt.DST_EXECUTE

    return &dt.DecoratedSyntaxTree{
        SelfType: dt.DST_WHILE_BLOCK,
        Children: []dt.DecoratedSyntaxTree{
            *condition,
            *block,
```

```
        },
    }, nil
}
```

Program whilestatement.go menganalisis while loop. Memastikan kondisi bertipe BOOLEAN dan menganalisis body statement.

2.2.43. atab.go

```
package datatype

import (
    "fmt"
)

type AtabEntry struct {
    IndexType        TabEntryType
    ElementType      TabEntryType
    ElementReference int
    LowBound         int
    HighBound        int
    ElementSize      int
    TotalSize        int
}

type Atab []AtabEntry

func (t Atab) FindArray(entry AtabEntry) (int, *AtabEntry) {
    for i, v := range t {
        if v.IndexType != entry.IndexType {
            continue
        }

        if v.ElementType != entry.ElementType {
            continue
        }

        if v.ElementReference != entry.ElementReference {
            switch v.ElementType {
            case TAB_ENTRY_ARRAY:
                fallthrough
            case TAB_ENTRY_RECORD:
```

```
                continue
            }
        }

        if v.LowBound != entry.LowBound {
            continue
        }

        if v.HighBound != entry.HighBound {
            continue
        }

        return i, &v
    }

    return -1, nil
}

func (t Atab) String() string {
    if len(t) == 0 {
        return "<empty array table>"
    }

    out := "Idx  IdxType      ElemType     ElemRef  Low   High  ElemSize
TotalSize\n"
    out += "---- ------------ ------------ -------- ----- ----- ---------
----------\n"

    for i, e := range t {
        out += fmt.Sprintf(
            "%-4d %-12s %-12s %-8d %-5d %-5d %-9d %-10d\n",
            i,
            e.IndexType.String(),
            e.ElementType.String(),
            e.ElementReference,
            e.LowBound,
            e.HighBound,
            e.ElementSize,
            e.TotalSize,
        )
```

```
    }

    return out
}
```

Program atab.go menangani tipe data array melalui struktur AtabEntry. Tabel ini menyimpan detail spesifik array seperti tipe indeks, tipe elemen, batas bawah dan atas (bounds), serta ukuran memori yang dibutuhkan. Informasi ini dipisahkan dari tabel simbol utama untuk efisiensi penanganan tipe terstruktur.

2.2.44. btab.go

```
package datatype

import (
    "fmt"
)

type BtabEntry struct {
    Start        int
    ParamEnd     int
    ReturnEnd    int
    End          int
    ParamSize    int
    ReturnSize   int
    VariableSize int
}

type Btab []BtabEntry

func (t Btab) String() string {
    if len(t) == 0 {
        return "<empty block table>"
    }

    out := "Idx  Start  End    ParamEnd  ReturnEnd  ParamSize  ReturnSize
VarSize\n"
    out += "---- ------ ------ --------- ---------- ---------- ----------
--------\n"

    for i, e := range t {
```

```
        out += fmt.Sprintf(
            "%-4d %-6d %-6d %-9d %-10d %-10d %-11d %-8d\n",
            i,
            e.Start,
            e.End,
            e.ParamEnd,
            e.ReturnEnd,
            e.ParamSize,
            e.ReturnSize,
            e.VariableSize,
        )
    }

    return out
}
```

Program btab.go mendefinisikan BtabEntry yang menyimpan informasi mengenai blok kode, seperti program utama, prosedur, atau fungsi. Tabel ini mencatat rentang indeks parameter dan variabel lokal dalam tabel simbol, serta ukuran memori yang dibutuhkan untuk stack frame (parameter, return value, dan variabel lokal) saat eksekusi.

2.2.45. dst.go

```
package datatype

type DSTProperty int

const (
    DST_ROOT DSTProperty = iota
    DST_INDEX
    DST_OPERAND
    DST_TARGET
    DST_VALUE
    DST_DOWNTO
    DST_UPTO
    DST_CONDITION
    DST_PARAMETER
    DST_DECLARE
    DST_EXECUTE
    DST_THEN
    DST_ELSE
```

```
    DST_FROM
)

func (p DSTProperty) String() string {
    names := [...]string{
        "root",
        "index",
        "operand",
        "target",
        "value",
        "downto",
        "upto",
        "condition",
        "parameter",
        "declare",
        "execute",
        "then",
        "else",
        "from",
    }
    if int(p) < 0 || int(p) >= len(names) {
        return "unknown"
    }
    return names[p]
}

type DSTNodeType int

const (
    DST_CHAR_LITERAL DSTNodeType = iota
    DST_STR_LITERAL
    DST_INT_LITERAL
    DST_REAL_LITERAL
    DST_BOOL_LITERAL
    DST_ADD_OPERATOR
    DST_SUB_OPERATOR
    DST_MUL_OPERATOR
    DST_MOD_OPERATOR
    DST_DIV_OPERATOR
    DST_AND_OPERATOR
```

```
    DST_OR_OPERATOR
    DST_EQ_OPERATOR
    DST_NE_OPERATOR
    DST_GT_OPERATOR
    DST_LT_OPERATOR
    DST_GE_OPERATOR
    DST_LE_OPERATOR
    DST_NOT_OPERATOR
    DST_NEG_OPERATOR
    DST_CAST_OPERATOR
    DST_ASSIGNMENT_OPERATOR
    DST_ARRAY_ELEMENT
    DST_RECORD_FIELD
    DST_PROCEDURE_CALL
    DST_FUNCTION_CALL
    DST_BLOCK
    DST_IF_BLOCK
    DST_FOR_BLOCK
    DST_WHILE_BLOCK
    DST_CONST
    DST_TYPE
    DST_VARIABLE
    DST_FUNCTION
    DST_PROCEDURE
    DST_CONSTANT_DECLARATIONS
    DST_TYPE_DECLARATIONS
    DST_VARIABLE_DECLARATIONS
    DST_SUBPROGRAM_DECLARATIONS
    DST_PROGRAM
)

func (t DSTNodeType) String() string {
    names := [...]string{
        "char-literal",
        "str-literal",
        "int-literal",
        "real-literal",
        "bool-literal",
        "add-op",
        "sub-op",
```

```
        "mul-op",
        "mod-op",
        "div-op",
        "and-op",
        "or-op",
        "eq-op",
        "ne-op",
        "gt-op",
        "lt-op",
        "ge-op",
        "le-op",
        "not-op",
        "neg-op",
        "cast-op",
        "assign-op",
        "array-element",
        "record-field",
        "procedure-call",
        "function-call",
        "block",
        "if-block",
        "for-block",
        "while-block",
        "const",
        "type",
        "variable",
        "function",
        "procedure",
        "const-decls",
        "type-decls",
        "var-decls",
        "subprogram-decls",
        "program",
    }
    if int(t) < 0 || int(t) >= len(names) {
        return "unknown"
    }
    return names[t]
}
```

```
type DecoratedSyntaxTree struct {
    Property DSTProperty
    SelfType DSTNodeType
    Data     int
    Children []DecoratedSyntaxTree
}
```

Program dst.go mendefinisikan struktur pohon sintaksis yang telah didekorasi (DecoratedSyntaxTree). Ia berisi definisi tipe node (DSTNodeType) dan properti node (DSTProperty) yang merepresentasikan elemen program secara semantik. Struktur ini menjadi jembatan antara hasil parsing dan tahap kode generasi selanjutnya.

2.2.46. dstdisplay.go

```
package datatype

import (
    "fmt"
    "strings"
)

func (t DecoratedSyntaxTree) String() string {
    var sb strings.Builder
    t.writeString(&sb, "", true, true, nil, nil, nil, nil)
    return sb.String()
}


func (t DecoratedSyntaxTree) StringWithSymbols(tab Tab, atab Atab, btab
Btab, strtab StrTab) string {
    var sb strings.Builder
    t.writeString(&sb, "", true, true, &tab, &atab, &btab, &strtab)
    return sb.String()
}


func formatDataWithSymbols(nodeType DSTNodeType, data int, tab *Tab, atab
*Atab, btab *Btab, strtab *StrTab) string {
    if tab == nil {
        return fmt.Sprintf(" (%d)", data)
    }
```

```
    switch nodeType {
    case DST_CHAR_LITERAL:
        return fmt.Sprintf(": '%c'", rune(data))

    case DST_STR_LITERAL:
        if strtab != nil && data >= 0 && data < len(*strtab) {
            return fmt.Sprintf(": \"%s\"", (*strtab)[data].String)
        }
        return fmt.Sprintf(" (strtab[%d])", data)

    case DST_INT_LITERAL:
        return fmt.Sprintf(": %d", data)

    case DST_BOOL_LITERAL:
        if data == 1 {
            return ": true"
        }
        return ": false"

    case DST_CAST_OPERATOR:
        typeName := TabEntryType(data).String()
        return fmt.Sprintf(": to %s", typeName)

    case DST_VARIABLE, DST_CONST, DST_TYPE:

        if data >= 0 && data < len(*tab) {
            return fmt.Sprintf(": %s (tab[%d])", (*tab)[data].Identifier,
data)
        }
        return fmt.Sprintf(" (tab[%d])", data)

    case DST_FUNCTION_CALL, DST_PROCEDURE_CALL:

        if data >= 0 && data < len(*tab) {
            return fmt.Sprintf(": %s (tab[%d])", (*tab)[data].Identifier,
data)
        }
        return fmt.Sprintf(" (tab[%d])", data)
```

```
    case DST_ARRAY_ELEMENT, DST_RECORD_FIELD:

        if data >= 0 && data < len(*tab) {
            return fmt.Sprintf(": %s (tab[%d])", (*tab)[data].Identifier,
data)
        }
        return fmt.Sprintf(" (tab[%d])", data)

    case DST_FUNCTION, DST_PROCEDURE, DST_PROGRAM:

        if data >= 0 && data < len(*tab) {
            return fmt.Sprintf(": %s (tab[%d])", (*tab)[data].Identifier,
data)
        }
        return fmt.Sprintf(" (tab[%d])", data)

    default:

        if data != 0 {
            return fmt.Sprintf(" (%d)", data)
        }
        return ""
    }
}


func (t DecoratedSyntaxTree) writeString(sb *strings.Builder, prefix string,
isLast bool, isRoot bool, tab *Tab, atab *Atab, btab *Btab, strtab *StrTab)
{
    if prefix != "" {
        if isLast {
            sb.WriteString(prefix + "└─")
        } else {
            sb.WriteString(prefix + "├─")
        }
    }


    if t.Property != DST_ROOT {
        sb.WriteString(t.Property.String())
```

```
        sb.WriteString(": ")
    }
    sb.WriteString(t.SelfType.String())


    dataStr := formatDataWithSymbols(t.SelfType, t.Data, tab, atab, btab,
strtab)
    sb.WriteString(dataStr)

    sb.WriteString("\n")


    childPrefix := prefix
    if isRoot || prefix != "" {
        if isLast {
            childPrefix += "  "
        } else {
            childPrefix += "| "
        }
    }

    for i, child := range t.Children {
        child.writeString(sb, childPrefix, i == len(t.Children)-1, false,
tab, atab, btab, strtab)
    }
}
```

Program dstdisplay.go berisi logika untuk menampilkan DST dalam format teks yang mudah dibaca. Ia menyediakan metode StringWithSymbols yang menghubungkan node DST kembali ke tabel-tabel simbol (Tab, Atab, Btab, StrTab), sehingga output pohon dapat menampilkan nama identifier asli, nilai literal, dan informasi tipe secara detail, bukan hanya sekadar angka indeks.

2.2.47. strtab.go

```
package datatype

import "fmt"

type StrTabEntry struct {
    Length    int
    String    string
```

```
    Reference int
}

type StrTab []StrTabEntry

func (t StrTab) FindString(value string) (int, *StrTabEntry) {
    for i, v := range t {
        if v.String == value {
            return i, &v
        }
    }

    return -1, nil
}

func (t StrTab) String() string {
    if t == nil || len(t) == 0 {
        return "<empty string table>"
    }

    const maxStringWidth = 30

    trunc := func(s string, max int) string {
        if len(s) <= max {
            return s
        }
        return s[:max-1] + "..."
    }

    out := "Idx  Len   String\n"
    out += "---- ----- ------------------------------\n"

    for i, e := range t {
        s := trunc(e.String, maxStringWidth)

        out += fmt.Sprintf(
            "%-4d %-5d %-30s\n",
            i,
            e.Length,
            s,
```

```
        )
    }

    return out
}
```

Program strtab.go mengelola literal string yang ditemukan dalam kode sumber. StrTab menyimpan isi string dan panjangnya, memungkinkan penggunaan kembali string yang sama (string interning) dan memisahkan penyimpanan data teks yang panjang dari logika tabel simbol utama.

2.2.48. tab.go

```
package datatype

import (
    "fmt"
)

type TabEntryObject int

const (
    TAB_ENTRY_PROGRAM TabEntryObject = iota
    TAB_ENTRY_CONST
    TAB_ENTRY_VAR
    TAB_ENTRY_TYPE
    TAB_ENTRY_PROC
    TAB_ENTRY_FUNC
    TAB_ENTRY_PARAM
    TAB_ENTRY_FIELD
    TAB_ENTRY_RETURN
)

func (o TabEntryObject) String() string {
    names := []string{
        "program",
        "constant",
        "variable",
        "type",
        "procedure",
        "function",
```

```
        "parameter",
        "field",
        "return",
    }

    if int(o) < 0 || int(o) > len(names) {
        return "unknown"
    }

    return names[o]
}

type TabEntryType int

const (
    TAB_ENTRY_NONE TabEntryType = iota
    TAB_ENTRY_INTEGER
    TAB_ENTRY_REAL
    TAB_ENTRY_BOOLEAN
    TAB_ENTRY_CHAR
    TAB_ENTRY_ARRAY
    TAB_ENTRY_RECORD
    TAB_ENTRY_ALIAS
)

func (o TabEntryType) String() string {
    names := []string{
        "none",
        "integer",
        "real",
        "boolean",
        "char",
        "array",
        "record",
        "alias",
    }

    if int(o) < 0 || int(o) > len(names) {
        return "unknown"
    }
```

```
    return names[o]
}

type TabEntry struct {
    Identifier string
    Link       int
    Object     TabEntryObject
    Type       TabEntryType
    Reference  int
    Normal     bool
    Level      int
    Data       int
}

type Tab []TabEntry

func (t *Tab) FindIdentifier(id string, start int) (int, *TabEntry) {
    current := start

    if current == -1 || len(*t) == 0 {
        return -1, nil
    }

    for i := 0; current != -1 && i < 100; i++ {
        if current >= len(*t) {
            return -1, nil
        }

        if (*t)[current].Identifier == id {
            return current, &(*t)[current]
        }
        current = (*t)[current].Link
    }

    return -1, nil
}

func (t Tab) String() string {
    if len(t) == 0 {
```

```
        return "<empty symbol table>"
    }

    out := "Idx  Identifier       Link  Object        Type          Ref
Norm  Level  Data\n"
    out += "---- ---------------- ----- ------------ ------------ -----
----- ------ -----\n"

    for i, e := range t {
        identifier := e.Identifier

        if len(identifier) > 16 {
            identifier = identifier[:13] + "..."
        }

        out += fmt.Sprintf(
            "%-4d %-16s %-5d %-13s %-13s %-5d %-5t %-6d %-5d\n",
            i,
            identifier,
            e.Link,
            e.Object.String(),
            e.Type.String(),
            e.Reference,
            e.Normal,
            e.Level,
            e.Data,
        )
    }

    return out
}
```

Program tab.go mendefinisikan struktur utama tabel simbol (Tab dan TabEntry) yang menyimpan seluruh identifier dalam program. Setiap entri mencatat nama identifier, jenis objeknya (seperti variabel, konstanta, atau prosedur), tipe datanya, level scope, serta referensi ke tabel lain jika diperlukan. Struktur ini menggunakan sistem linked list melalui field Link untuk menangani scoping dan pencarian identifier.

2.2.49. main.go

```
package main
```

```
import (
    "flag"
    "fmt"
    "log"
    "os"
    "slices"

    iox "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/common"
    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
    "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/lexer"
    "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/parser"
    "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/semantic"
)

func main() {
    rules := flag.String("rules", "config/tokenizer_m3.json", "path ke DFA
JSON")
    in := flag.String("input", "", "path file sumber")
    flag.Parse()

    if *in == "" {
        fmt.Fprintln(os.Stderr, "missing --input <file>")
        os.Exit(2)
    }


    d, err := lexer.LoadJSON(*rules)
    if err != nil {
        log.Fatal(err)
    }


    rr, err := iox.NewRuneReaderFromFile(*in)
    if err != nil {
        log.Fatal(err)
    }


    tokens, errs := lexer.New(d, rr).ScanAll()
```

```
    for _, e := range errs {
        fmt.Fprintln(os.Stderr, e)
    }

    if len(errs) > 0 {
        os.Exit(1)
    }


    tokens = slices.Collect(func(yield func(dt.Token) bool) {
        for _, token := range tokens {
            if token.Type != dt.COMMENT {
                if !yield(token) {
                    return
                }
            }
        }
    })


    parseTree, err := parser.New(tokens).Parse()

    if err != nil {
        fmt.Fprintf(os.Stderr, "Parse error: %v\n", err)
        os.Exit(1)
    }

    if parseTree == nil {
        fmt.Fprintln(os.Stderr, "Parse tree is nil")
        os.Exit(1)
    }


    analyzer := semantic.New(parseTree)
    tab, atab, btab, strtab, dst, err := analyzer.Analyze()

    if err != nil {
        fmt.Fprintf(os.Stderr, "Semantic error: %v\n", err)
        os.Exit(1)
    }
```

```
    if dst != nil {
        fmt.Println(dst.StringWithSymbols(tab, atab, btab, strtab))
    }
    fmt.Println()


    fmt.Println("=== Symbol Table (TAB) ===")
    fmt.Println(tab.String())
    fmt.Println()

    fmt.Println("=== Array Table (ATAB) ===")
    fmt.Println(atab.String())
    fmt.Println()

    fmt.Println("=== Block Table (BTAB) ===")
    fmt.Println(btab.String())
    fmt.Println()

    fmt.Println("=== String Table (STRTAB) ===")
    fmt.Println(strtab.String())
}
```

File main.go bertindak sebagai titik pemicu dan penampil hasil. Setelah proses parsing berhasil menyediakan sebuah Parse Tree yang valid secara sintaksis, tugas main.go adalah menginisialisasi SemanticAnalyzer dengan Parse Tree tersebut. Kemudian, ia memanggil method analyze(), yang merupakan satu-satunya perintah untuk menjalankan keseluruhan proses analisis semantik termasuk validasi tipe, manajemen scope, dan pemeriksaan deklarasi. Setelah eksekusi selesai, main.go akan menangani hasilnya: jika terjadi kesalahan semantik, ia akan menangkap, melaporkan error tersebut, dan menghentikan program. Jika berhasil, ia akan mengambil Decorated Syntax Tree (DST) dan keempat tabel simbol (tab, atab, btab, strtab) yang dihasilkan, lalu mencetak semuanya ke konsol dalam format yang terstruktur dan mudah dibaca.

2.3. Alur Kerja Program

Proses analisis semantik ini dimulai setelah parser berhasil menghasilkan sebuah Parse Tree yang valid secara sintaksis. Pohon ini, yang merepresentasikan struktur gramatikal dari kode sumber, kemudian menjadi input bagi komponen SemanticAnalyzer. Penganalisis ini bekerja dengan cara berjalan (traversal) secara rekursif di sepanjang Parse Tree, dari node akar (program) hingga ke daun-daunnya, untuk memverifikasi makna dan konsistensi logis dari program. Untuk melacak konteks, SemanticAnalyzer mengelola serangkaian tabel data inti:

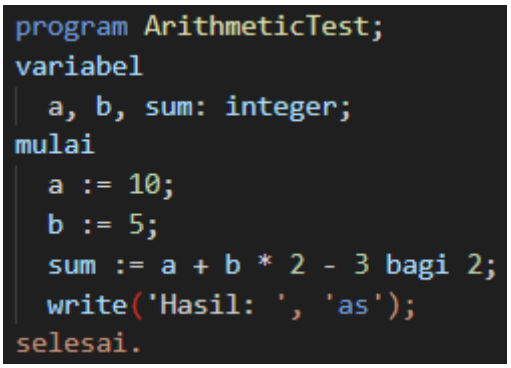
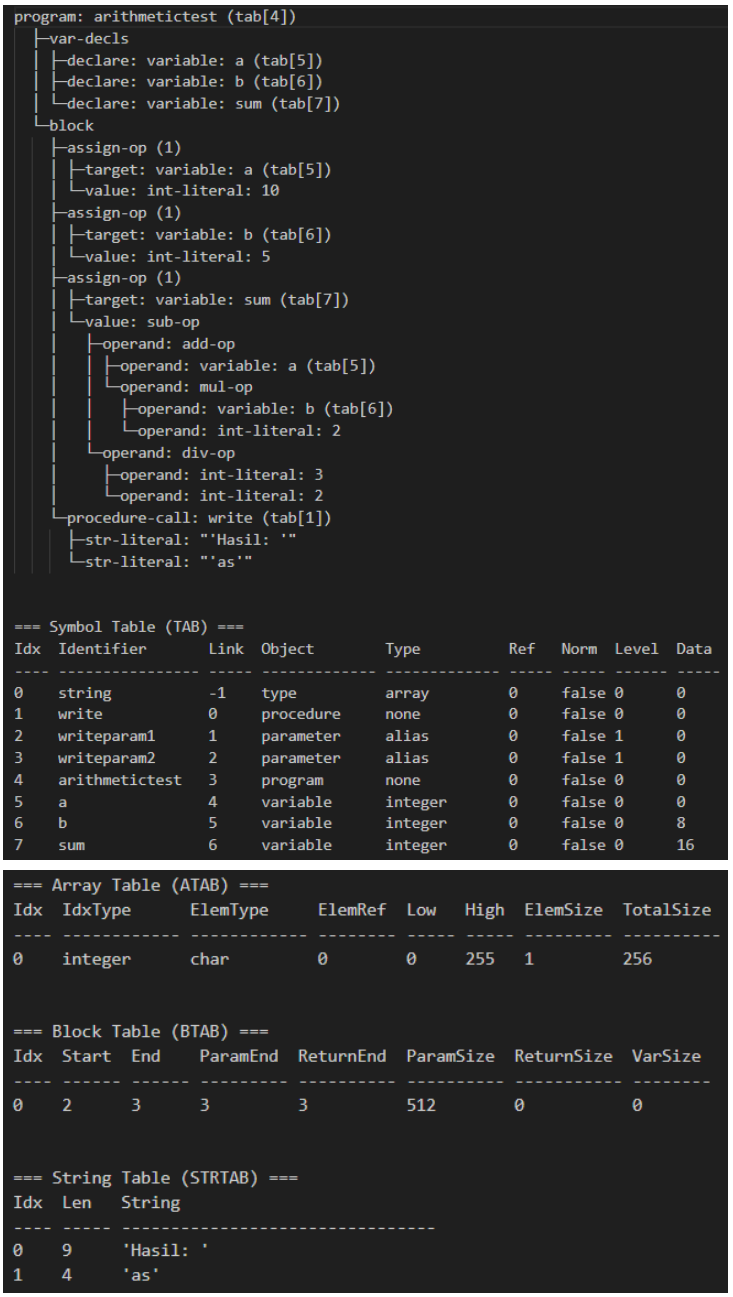
tabel simbol (tab) untuk identifier, tabel blok (btab) untuk scope, tabel tipe (atab) untuk definisi array dan record, serta tabel string (strtab) untuk nilai literal.

Saat penganalisis menelusuri cabang-cabang deklarasi (VAR, TYPE, PROCEDURE), ia akan mengisi tabel simbol. Setiap identifier yang dideklarasikan (seperti nama variabel atau fungsi) akan ditambahkan sebagai entri baru ke dalam tab. Bagian paling krusial dari proses ini adalah pengaturan Link pada setiap entri. Link ini tidak menunjuk ke entri sebelumnya secara acak, melainkan secara spesifik ke deklarasi identifier dengan nama yang sama yang berada di scope yang lebih tinggi (luar). Mekanisme ini menciptakan sebuah rantai scope (scope chain) untuk setiap nama, yang memungkinkan penganalisis untuk menemukan variabel yang benar saat ada nama yang sama di scope yang berbeda (misalnya, variabel global x dan variabel lokal x). Penganalisis juga akan melaporkan error jika ada upaya untuk mendeklarasikan ulang identifier dalam scope yang sama.

Setelah memproses deklarasi, penganalisis melanjutkan ke bagian statement dan ekspresi. Di sini, ia menggunakan informasi yang tersimpan di tabel simbol untuk melakukan validasi. Untuk setiap statement penugasan (:=), ia akan memeriksa tipe data dari variabel target di sebelah kiri dan ekspresi di sebelah kanan untuk memastikan keduanya kompatibel. Untuk ekspresi aritmatika atau logika, ia memeriksa tipe operan dan menentukan tipe hasil operasi. Pada statement kontrol alur seperti IF dan WHILE, ia memastikan bahwa ekspresi kondisi harus menghasilkan tipe BOOLEAN. Demikian pula, saat memproses pemanggilan prosedur atau fungsi, ia memvalidasi jumlah dan tipe argumen yang diberikan terhadap daftar parameter formal yang telah dideklarasikan sebelumnya.

Seluruh proses ini pada dasarnya adalah validasi berkelanjutan terhadap aturan-aturan bahasa. Jika penganalisis berhasil menelusuri seluruh Parse Tree tanpa menemukan pelanggaran aturan semantik seperti penggunaan variabel yang belum dideklarasikan, ketidakcocokan tipe, atau jumlah parameter yang salah—maka kode sumber dianggap valid secara semantik dan logis. Namun, jika satu saja kesalahan ditemukan, proses akan dihentikan dan sebuah semantic error yang spesifik akan dilaporkan kepada pengguna.

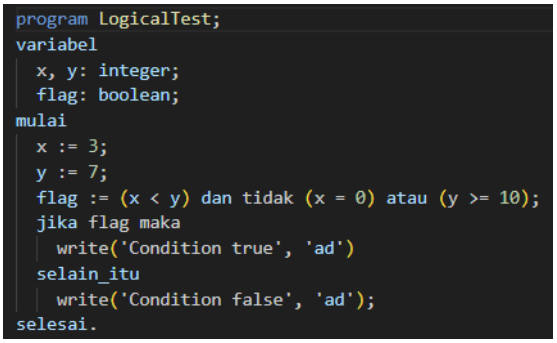
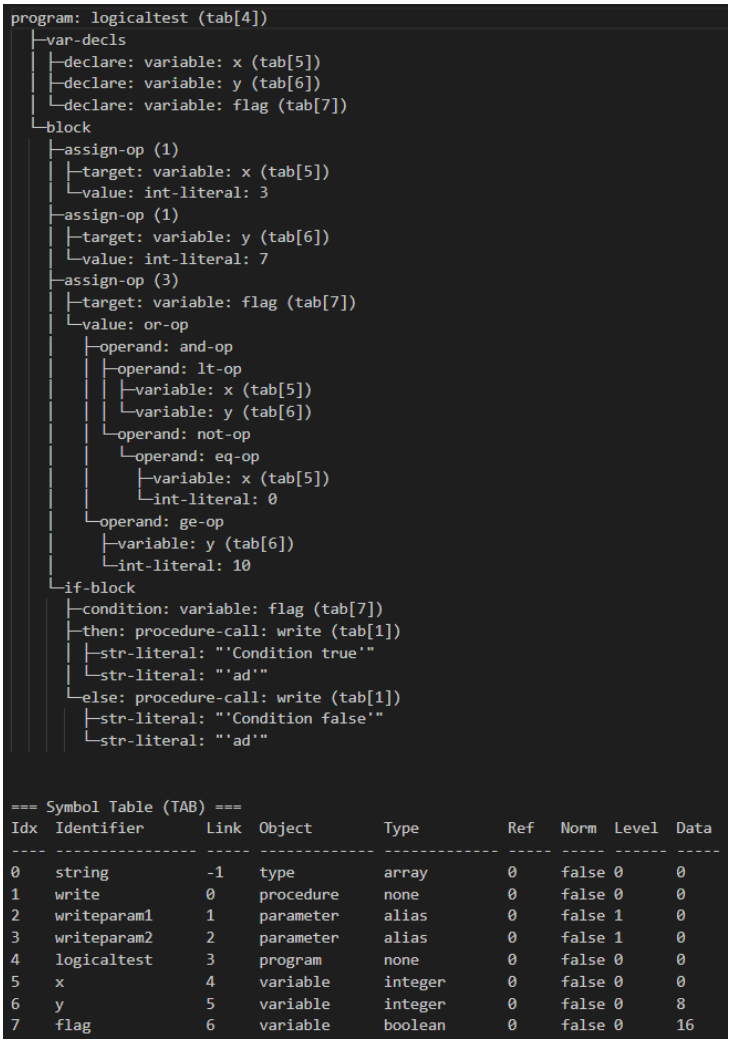
3. Pengujian

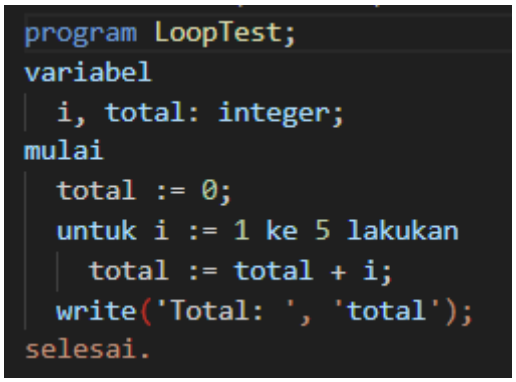
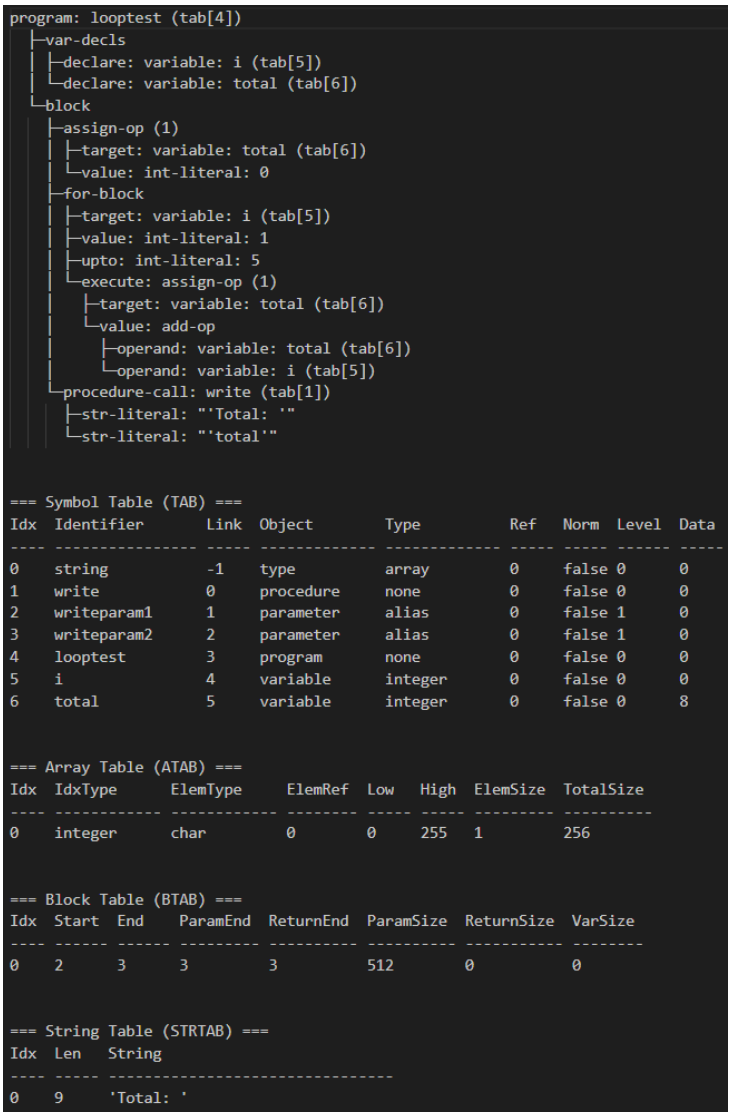
No	Input	Output
1.		

```
program ArithmeticTest;
variabel
  a, b, sum: integer;
mulai
  a := 10;
  b := 5;
  sum := a + b * 2 - 3 bagi 2;
  write('Hasil: ', 'as');
selesai.
```

```
program: arithmetictest (tab[4])
  ├─var-decls
  │ ├─declare: variable: a (tab[5])
  │ ├─declare: variable: b (tab[6])
  │ └─declare: variable: sum (tab[7])
  └─block
    ├─assign-op (1)
    │ ├─target: variable: a (tab[5])
    │ └─value: int-literal: 10
    ├─assign-op (1)
    │ ├─target: variable: b (tab[6])
    │ └─value: int-literal: 5
    ├─assign-op (1)
    │ ├─target: variable: sum (tab[7])
    │ └─value: sub-op
    │   ├─operand: add-op
    │   │ ├─operand: variable: a (tab[5])
    │   │ └─operand: mul-op
    │   │   ├─operand: variable: b (tab[6])
    │   │   └─operand: int-literal: 2
    │   └─operand: div-op
    │     ├─operand: int-literal: 3
    │     └─operand: int-literal: 2
    └─procedure-call: write (tab[1])
      ├─str-literal: "'Hasil: '"
      └─str-literal: "'as'"


=== Symbol Table (TAB) ===
Idx  Identifier       Link  Object        Type          Ref   Norm  Level  Data
---- ---------------- ----- ------------- ------------- ----- ----- ------ -----
0    string           -1    type          array         0     false 0      0
1    write            0     procedure     none          0     false 0      0
2    writeparam1      1     parameter     alias         0     false 1      0
3    writeparam2      2     parameter     alias         0     false 1      0
4    arithmetictest   3     program       none          0     false 0      0
5    a                4     variable      integer       0     false 0      0
6    b                5     variable      integer       0     false 0      8
7    sum              6     variable      integer       0     false 0      16
```

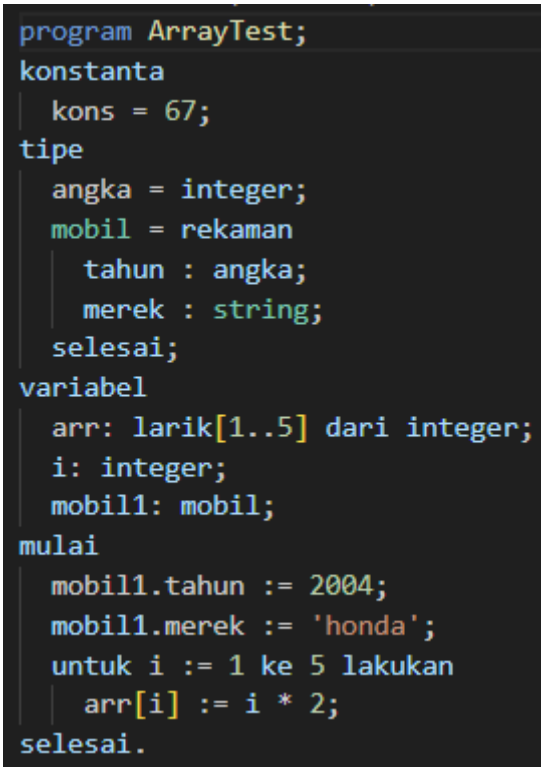
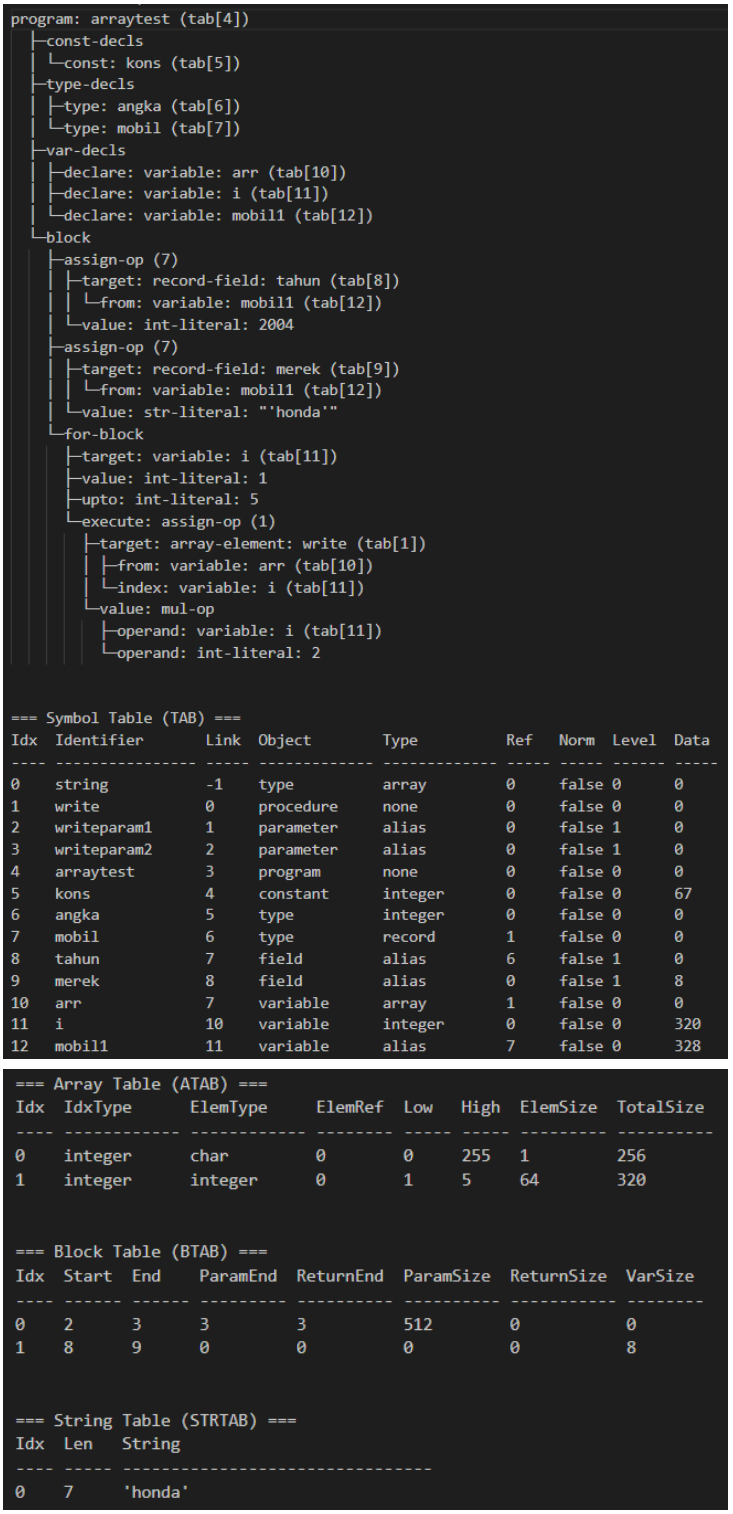
```
=== Array Table (ATAB) ===
Idx  IdxType      ElemType     ElemRef  Low   High  ElemSize  TotalSize
---- ------------ ------------ -------- ----- ----- --------- ----------
0    integer      char         0        0     255   1         256


=== Block Table (BTAB) ===
Idx  Start  End    ParamEnd  ReturnEnd  ParamSize  ReturnSize  VarSize
---- ------ ------ --------- ---------- ---------- ----------- --------
0    2      3      3         3          512        0           0


=== String Table (STRTAB) ===
Idx  Len   String
---- ----- --------------------------------
0    9     'Hasil: '
1    4     'as'
```

No	Input	Output
2.		

No	Input	Output
3.		

```
program LoopTest;
variabel
  i, total: integer;
mulai
  total := 0;
  untuk i := 1 ke 5 lakukan
    total := total + i;
  write('Total: ', 'total');
selesai.
```

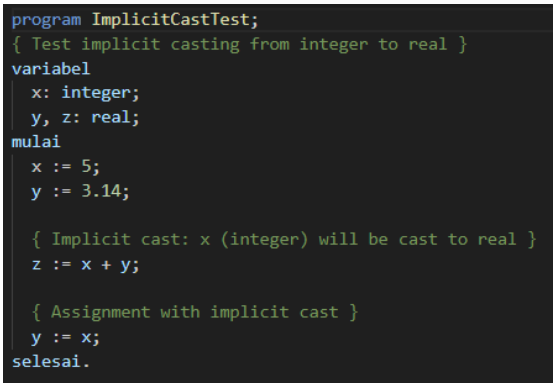
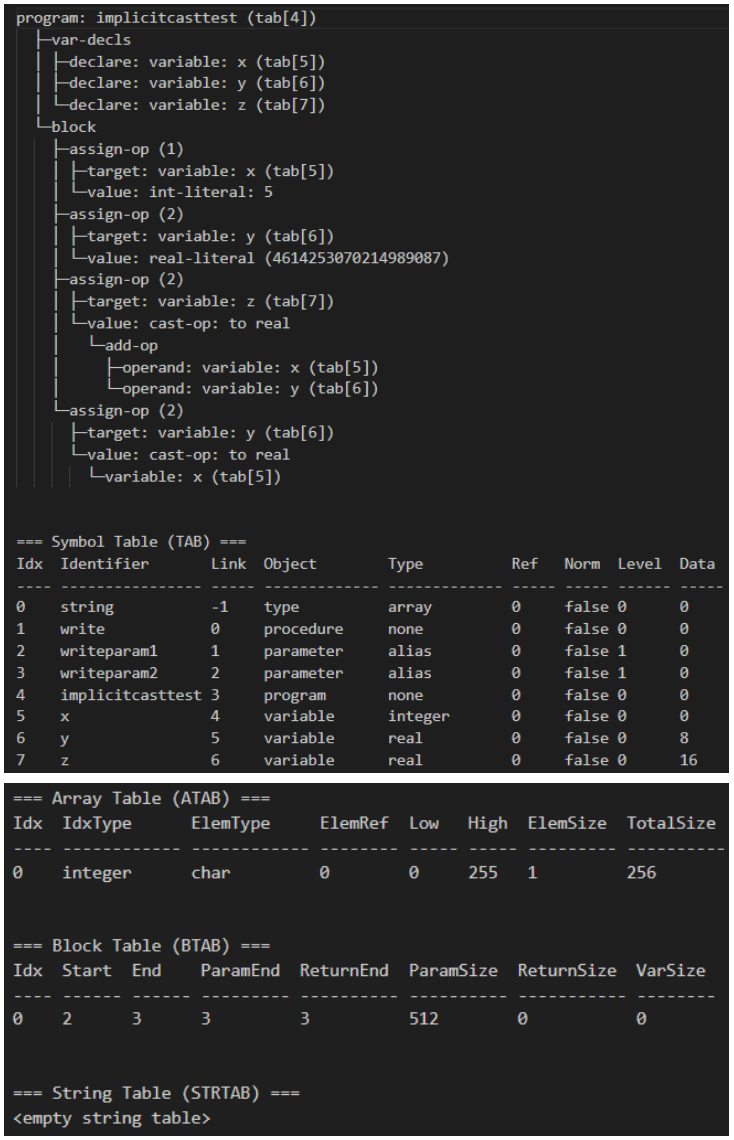
```
program: looptest (tab[4])
  ├─var-decls
  │ ├─declare: variable: i (tab[5])
  │ └─declare: variable: total (tab[6])
  └─block
    ├─assign-op (1)
    │ ├─target: variable: total (tab[6])
    │ └─value: int-literal: 0
    ├─for-block
    │ ├─target: variable: i (tab[5])
    │ ├─value: int-literal: 1
    │ ├─upto: int-literal: 5
    │ └─execute: assign-op (1)
    │   ├─target: variable: total (tab[6])
    │   └─value: add-op
    │     ├─operand: variable: total (tab[6])
    │     └─operand: variable: i (tab[5])
    └─procedure-call: write (tab[1])
      ├─str-literal: "'Total: '"
      └─str-literal: "'total'"


=== Symbol Table (TAB) ===
Idx  Identifier       Link  Object        Type          Ref   Norm  Level  Data
---- ---------------- ----- ------------- ------------- ----- ----- ------ -----
0    string           -1    type          array         0     false 0      0
1    write            0     procedure     none          0     false 0      0
2    writeparam1      1     parameter     alias         0     false 1      0
3    writeparam2      2     parameter     alias         0     false 1      0
4    looptest         3     program       none          0     false 0      0
5    i                4     variable      integer       0     false 0      0
6    total            5     variable      integer       0     false 0      8


=== Array Table (ATAB) ===
Idx  IdxType      ElemType     ElemRef  Low   High  ElemSize  TotalSize
---- ------------ ------------ -------- ----- ----- --------- ----------
0    integer      char         0        0     255   1         256


=== Block Table (BTAB) ===
Idx  Start  End    ParamEnd  ReturnEnd  ParamSize  ReturnSize  VarSize
---- ------ ------ --------- ---------- ---------- ----------- --------
0    2      3      3         3          512        0           0


=== String Table (STRTAB) ===
Idx  Len   String
---- ----- --------------------------------
0    9     'Total: '
```

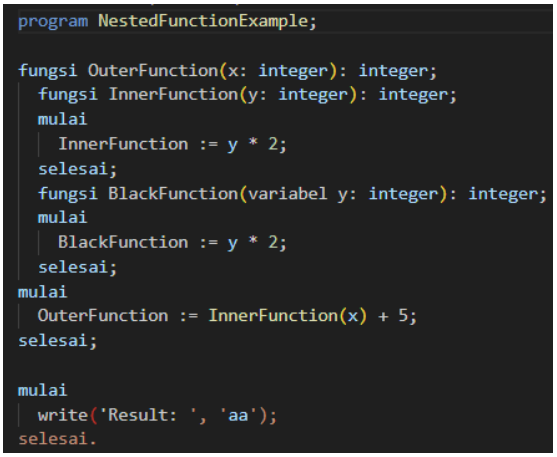
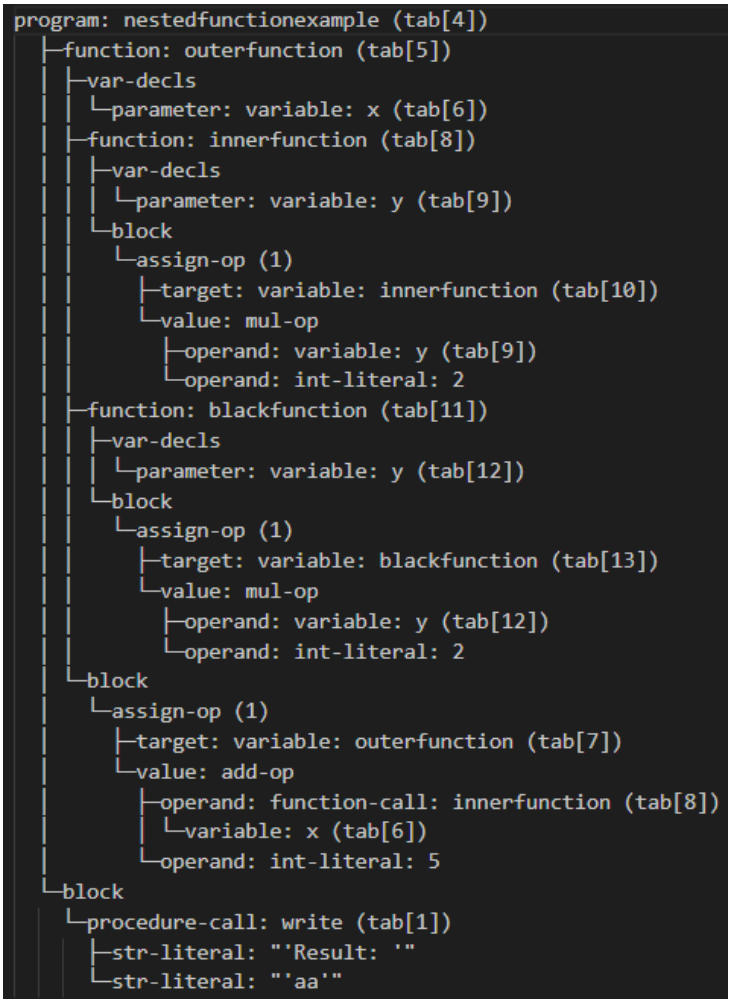
No	Input	Output
4.		

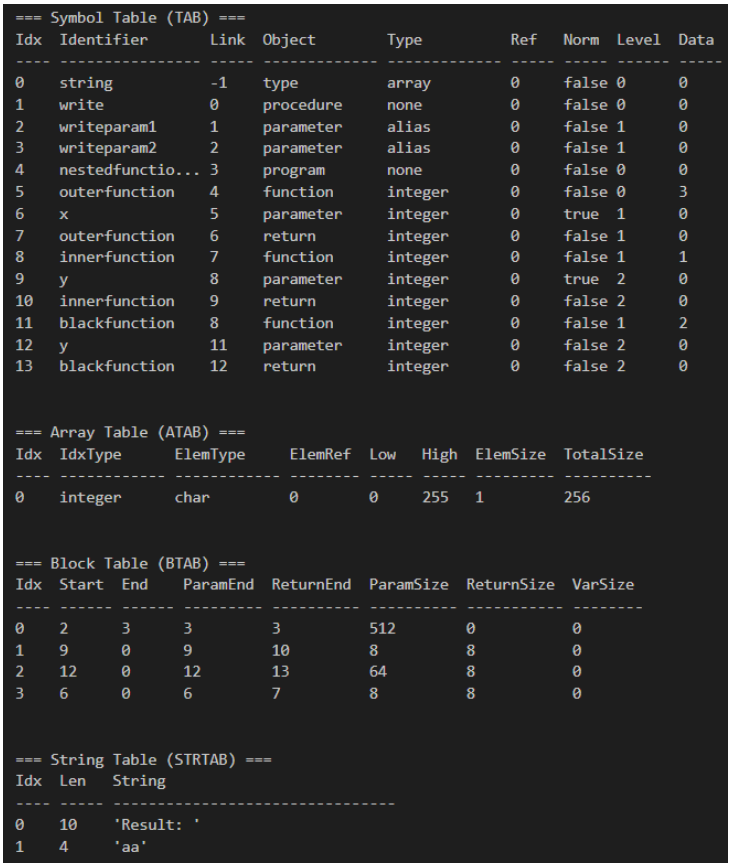
```
program ArrayTest;
konstanta
  kons = 67;
tipe
  angka = integer;
  mobil = rekaman
    tahun : angka;
    merek : string;
  selesai;
variabel
  arr: larik[1..5] dari integer;
  i: integer;
  mobil1: mobil;
mulai
  mobil1.tahun := 2004;
  mobil1.merek := 'honda';
  untuk i := 1 ke 5 lakukan
    arr[i] := i * 2;
selesai.
```

```
program: arraytest (tab[4])
  ├─const-decls
  │ └─const: kons (tab[5])
  ├─type-decls
  │ ├─type: angka (tab[6])
  │ └─type: mobil (tab[7])
  ├─var-decls
  │ ├─declare: variable: arr (tab[10])
  │ ├─declare: variable: i (tab[11])
  │ └─declare: variable: mobil1 (tab[12])
  └─block
    ├─assign-op (7)
    │ ├─target: record-field: tahun (tab[8])
    │ │ └─from: variable: mobil1 (tab[12])
    │ └─value: int-literal: 2004
    ├─assign-op (7)
    │ ├─target: record-field: merek (tab[9])
    │ │ └─from: variable: mobil1 (tab[12])
    │ └─value: str-literal: "'honda'"
    └─for-block
      ├─target: variable: i (tab[11])
      ├─value: int-literal: 1
      ├─upto: int-literal: 5
      └─execute: assign-op (1)
        ├─target: array-element: write (tab[1])
        │ ├─from: variable: arr (tab[10])
        │ └─index: variable: i (tab[11])
        └─value: mul-op
          ├─operand: variable: i (tab[11])
          └─operand: int-literal: 2


=== Symbol Table (TAB) ===
Idx  Identifier       Link  Object        Type          Ref   Norm  Level  Data
---- ---------------- ----- ------------ ------------ ----- ----- ------ -----
0    string           -1    type          array         0     false 0      0
1    write            0     procedure     none          0     false 0      0
2    writeparam1      1     parameter     alias         0     false 1      0
3    writeparam2      2     parameter     alias         0     false 1      0
4    arraytest        3     program       none          0     false 0      0
5    kons             4     constant      integer       0     false 0      67
6    angka            5     type          integer       0     false 0      0
7    mobil            6     type          record        1     false 0      0
8    tahun            7     field         alias         6     false 1      0
9    merek            8     field         alias         0     false 1      8
10   arr              7     variable      array         1     false 0      0
11   i                10    variable      integer       0     false 0      320
12   mobil1           11    variable      alias         7     false 0      328
```

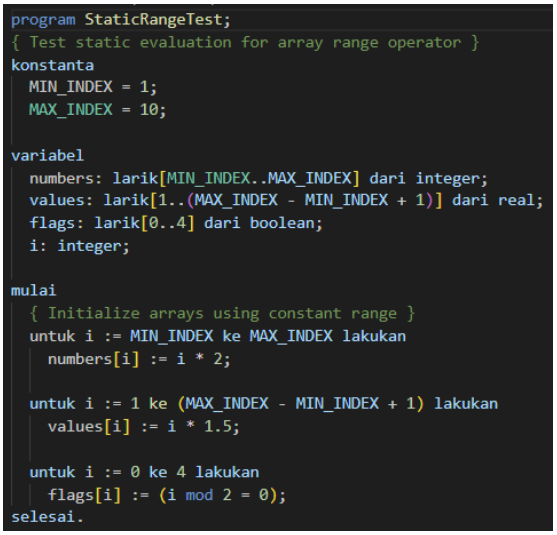
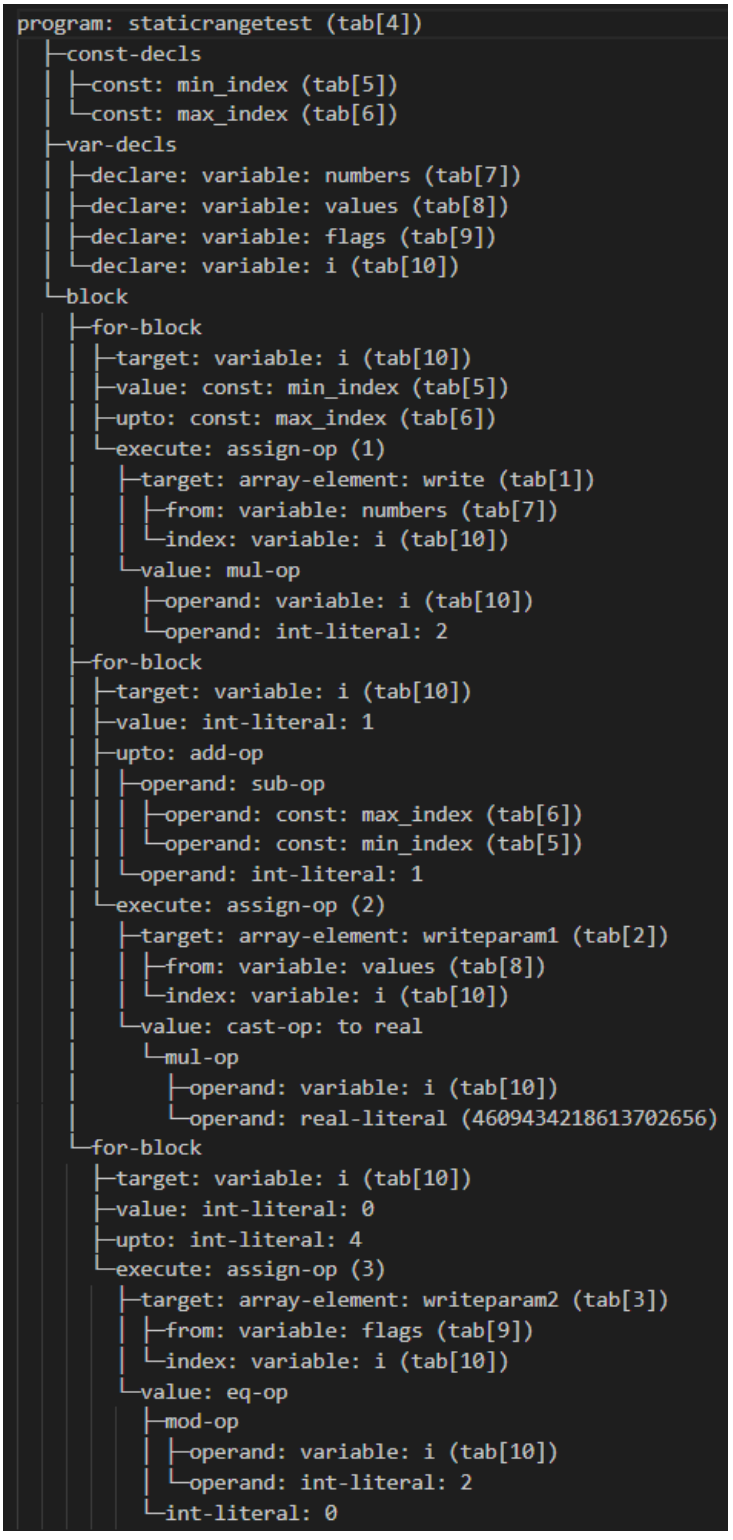
```
=== Array Table (ATAB) ===
Idx  IdxType      ElemType     ElemRef  Low   High  ElemSize  TotalSize
---- ------------ ------------ -------- ----- ----- --------- ----------
0    integer      char         0        0     255   1         256
1    integer      integer      0        1     5     64        320


=== Block Table (BTAB) ===
Idx  Start  End    ParamEnd  ReturnEnd  ParamSize  ReturnSize  VarSize
---- ------ ------ --------- ---------- ---------- ----------- --------
0    2      3      3         3          512        0           0
1    8      9      0         0          0          0           8


=== String Table (STRTAB) ===
Idx  Len   String
---- ----- --------------------------------
0    7     'honda'
```

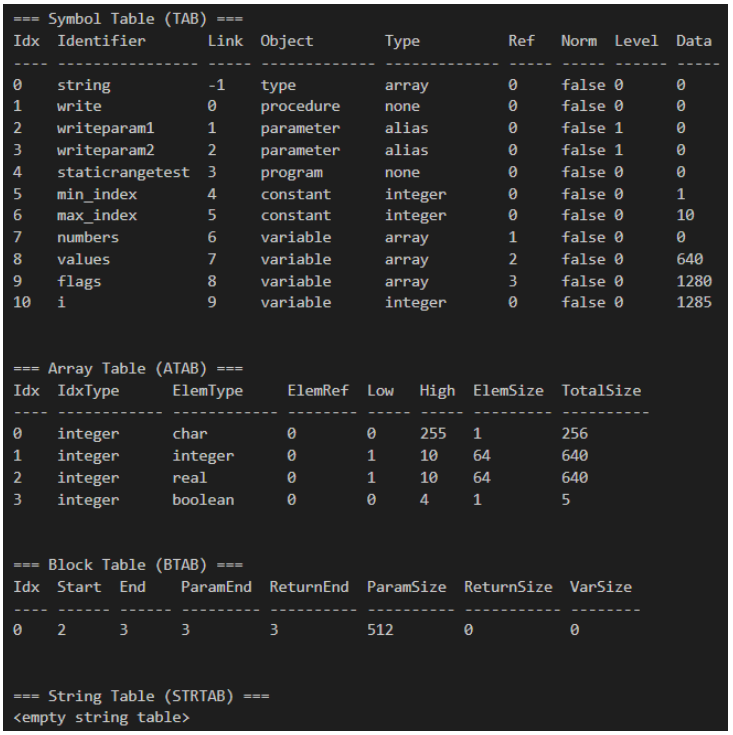
No	Input	Output
5.		 **Keterangan:** Ini test-case yang memang harus salah

No	Input	Output
6.		

No	Input	Output
		

```
=== Symbol Table (TAB) ===
Idx  Identifier       Link  Object        Type          Ref   Norm  Level  Data
---- ---------------- ----- ------------- ------------- ----- ----- ------ -----
0    string           -1    type          array         0     false 0      0
1    write            0     procedure     none          0     false 0      0
2    writeparam1      1     parameter     alias         0     false 1      0
3    writeparam2      2     parameter     alias         0     false 1      0
4    nestedfunctio... 3     program       none          0     false 0      0
5    outerfunction    4     function      integer       0     false 0      3
6    x                5     parameter     integer       0     true  1      0
7    outerfunction    6     return        integer       0     false 1      0
8    innerfunction    7     function      integer       0     false 1      1
9    y                8     parameter     integer       0     true  2      0
10   innerfunction    9     return        integer       0     false 2      0
11   blackfunction    8     function      integer       0     false 1      2
12   y                11    parameter     integer       0     false 2      0
13   blackfunction    12    return        integer       0     false 2      0


=== Array Table (ATAB) ===
Idx  IdxType      ElemType     ElemRef  Low   High  ElemSize  TotalSize
---- ------------ ------------ -------- ----- ----- --------- ----------
0    integer      char         0        0     255   1         256


=== Block Table (BTAB) ===
Idx  Start  End    ParamEnd  ReturnEnd  ParamSize  ReturnSize  VarSize
---- ------ ------ --------- ---------- ---------- ----------- --------
0    2      3      3         3          512        0           0
1    9      0      9         10         8          8           0
2    12     0      12        13         64         8           0
3    6      0      6         7          8          8           0


=== String Table (STRTAB) ===
Idx  Len   String
---- ----- --------------------------------
0    10    'Result: '
1    4     'aa'
```

No	Input	Output
7.		

No	Input	Output
		 **Keterangan:** Ini test-case yang memang harus salah

4. Kesimpulan dan Saran

4.1. Kesimpulan

Berdasarkan eksperimen dan analisis yang dilakukan sebelumnya, dapat disimpulkan bahwa:

- *Semantic Analyzer* berhasil menghasilkan *Decorated Syntax Tree* (DST) yang konsisten dari Parse Tree.
- Pengelolaan tabel simbol (TAB, ATAB, BTAB, STRTAB) berjalan terintegrasi dengan proses analisis.
- Aturan semantik seperti *type checking*, *scope checking*, dan validasi struktur kontrol telah diterapkan sesuai spesifikasi.
- *Static evaluation* untuk ekspresi tertentu bekerja dengan baik dan akurat.
- Mekanisme analisis subprogram (fungsi dan prosedur) berhasil mengelola dan menutup *scope* dengan benar.
- Pengujian menunjukkan bahwa sistem dapat membedakan input valid dan invalid melalui deteksi error semantik yang tepat.

4.2. Saran

- Meningkatkan kejelasan dan detail pada pesan error semantik.
- Menambah cakupan pengujian untuk kasus kompleks dan kombinasi fitur.
- Melakukan *refactoring* pada fungsi-fungsi *analyzer* yang terlalu panjang.
- Menambahkan anotasi lanjutan pada DST untuk mempermudah tahap *compiler* berikutnya.
- Membuat mekanisme pengecekan konsistensi otomatis untuk tabel simbol.

Lampiran

Pembagian Tugas

Anggota	Tugas	Persentase
13523123 Rhio Bimo Prakoso S	Laporan, Kode, Fix	20
13523137 M Aulia Azka	Kode, Testing, Fix	20
13523161 Arlow Emmanuel Hergara	Kode, Debugging, Testing	20
13523162 Fachriza Ahmad Setiyono	Kode, Debugging, Testing	20
13523163 Filbert Engyo	Laporan, Kode, Debugging	20

Pranala Repository

https://github.com/Azzkaaaa/NIG-Tubes-IF2224

Daftar Pustaka

https://www.geeksforgeeks.org/compiler-design/semantic-analysis-in-compiler-design/
https://learn.microsoft.com/id-id/dotnet/csharp/roslyn-sdk/get-started/semantic-analysis
https://www.tutorialspoint.com/compiler_design/compiler_design_attributed_grammars.htm
https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.htm
https://www.slideshare.net/slideshow/symbol-table-in-compiler-design/244014427
https://www.geeksforgeeks.org/compiler-design/symbol-table-compiler/
https://home.adelphi.edu/~siegfried/cs372/372l3.pdf
https://www.geeksforgeeks.org/system-design/visitor-design-pattern/

- POV, anda melihat (dan/atau membuat) laporan dengan 115 halaman