特別研究報告

題目

Dockerfile の保守性改善を目的とした言語 Myriad の提案とその処理系の試作

指導教員
肥後　芳樹 教授

報告者
山本　貴之

令和 5 年 2 月 7 日

大阪大学 基礎工学部 情報科学科

令和 4 年度 特別研究報告

Dockerfile の保守性改善を目的とした言語 Myriad の提案とその処理系の試作

山本　貴之

内容梗概

Docker とは，コンテナ仮想化技術の一つであり，多くの開発者が使用している技術である．また，Docker は Dockerfile と呼ばれるテキストファイルに，インフラ環境の構築手順を記述することができる．

コードクローンとは，ソースコード上に存在する，同一または類似のコード片を指す．

Dockerfile には，コードクローンの存在による保守性低下や，Dockerfile 特有の処理を記述することによる学習コストの高さなど，複数の問題点が存在する．

本研究では，Dockerfile の問題を解決し，開発者の支援を行うため，Dockerfile における保守性改善を目的とした言語 Myriad の提案とその処理系の試作を行った．Myriad は手続き型言語であり，Dockerfile にはない，関数，変数による置換，外部からの値挿入，条件分岐などの機能をサポートしている．さらに，Dockerfile の学習コスト削減のためにライブラリの試作も行った．

最後に，作成した Myriad の性能を評価すべく，可読性に関する比較実験，ライブラリを用いた比較実験，保守性に関する比較実験の 3 つの実験を行った．実験結果から，Myriad の可読性が既存ツールよりも優れていることが示された．さらに，アンケート結果から，ライブラリを用いることによる学習コストの低下について述べる意見を得た．また，Myriad で作成したプロジェクトの保守コストが，ツールを使わない場合より優れており，既存ツールと同等であることも示された．一方で，ライブラリを用いることで余計に記述時間が長くなることが判明した．

主な用語

Dockerfile
保守性
可読性
言語処理系

目 次

1 まえがき **4**

2 背景 **6**

2.1 Docker 6

2.2 Dockerfile 6

2.3 コードクローン 7

2.4 Dockerfile 間のクローン 8

2.4.1 Dockerfile におけるクローンによる保守性低下 8

2.4.2 Dockerfile 間のクローンに対応する既存ツール 9

2.5 Dockerfile の学習コスト 10

3 Myriad **12**

3.1 要件・設計 12

3.1.1 言語要件 12

3.1.2 トークンの種類 14

3.1.3 構文定義 14

3.1.4 コンパイル方法 15

3.2 言語処理系の実装 15

3.3 ライブラリ 15

4 評価 **17**

4.1 可読性に関する比較実験 17

4.1.1 実験手法 17

4.1.2 実験結果 19

4.2 ライブラリに関する比較実験 20

4.2.1 実験手法 20

4.2.2 実験結果 21

4.3 保守性に関する比較実験 21

4.3.1 実験手法 21

4.3.2 実験結果 23

5 関連研究 **24**

5.1 Modus 24

6 あとがき **25**

謝辞 **27**

参考文献 **28**

付録 A Myriad のトークン **30**

付録 B Myriad の構文定義 **31**

付録 C Myriad のコンパイル方法 **33**

付録 D 言語処理系の実装方法 **34**

付録 D.1 字句解析 34
付録 D.2 構文解析 34
付録 D.3 意味解析・中間言語生成 35
付録 D.4 Dockerfile 生成 36

付録 E 作成したライブラリの一覧 **37**

1　まえがき

Docker とはコンテナ仮想化技術を用いてアプリケーションを開発，運用，実行するためのオープンプラットフォームであり [1]，Web サーバなどのインフラ環境を高速に構築できることから，多くの開発者が使用している技術である [7]．また，Docker は IaC（Infrastructure as Code）とも呼ばれ，Dockerfile と呼ばれるテキストファイルにインフラ環境の構築手順を記述することができる [3].

コードクローンとはソースコード上に存在する同一または類似のコード片を指し，ソフトウェアの保守性を低下させる原因の一つであると報告されている [5].

Dockerfile にもコードクローンの存在が指摘されており，それにより複数の Dockerfile を扱うプロジェクトでは修正箇所の波及など，保守性の低下が懸念されている [6]．また，Dockerfile のコードクローンに対応するために保守を目的としたツールもいくつか提案されているが，いずれのツールも一長一短の性能を持ち，統一された方法が存在しない．さらに，Dockerfile は特有の処理を記述することが多いことから，Dockerfile の初学者にとっては学習コストが高く，ミスの誘発にも繋がると考える．

以上に挙げた背景を基に，Dockerfile 開発者の問題を解決し，支援するため，本研究では Dockerfile における保守性改善を目的とした言語 Myriad の提案とその処理系の試作を行った．Myriad は手続き型言語であり，Dockerfile にはない関数，変数による置換，外部からの値挿入，条件分岐などの機能をサポートしている．さらに，Dockerfile の学習コスト削減のためにライブラリの試作も行なった．また，既存の Dockerfile プロジェクトからのコードクローン検出や経験則によって頻出パターンを候補に挙げ，そこから 5 つのライブラリを試作した．

作成した Myriad の性能を評価すべく，3 つの比較実験を行なった．1 つ目は Myriad の可読性を既存ツールの内の 1 つと比較する実験であり，同じ Dockerfile 群を生成するプロジェクトをそれぞれ作成し，被験者に評価をしてもらった．実験で行ったアンケート結果から，Myriad の可読性が既存ツールよりも優れていることが示された．関数によって可読性が向上しているという意見や馴染みのある手続き型言語を採用していることで可読性が高いという意見を得た．2 つ目は Myriad のライブラリの有用性を示す実験であり，ライブラリを持たない Dockerfile そのものとの比較を行った．あるテーマに従い，Myriad のライブラリを使用して記述するグループと，Dockerfile をそのまま記述するグループに分け，それぞれが完成するまでの時間を計測した．その結果，完成するまでにかかった時間は，Myriad のライブラリを使用したグループの方が長いということが判明した．これは，ライブラリの中身を読み，適切なライブラリを選択するのに時間を要したためであると考えられる．しかし，集計したアンケートによると，ライブラリによって，Dockerfile やシェルコマンドの使い方

の詳細を知らなくても記述できる点を評価する意見もあり，学習コストの削減を実現することができたと考える．

2 背景

本章では Dockerfile における既存の問題点を指摘し，本研究の背景について述べる．

2.1 Docker

Docker は，コンテナ仮想化技術を用いてアプリケーションを開発，運用，実行するためのオープンプラットフォームである [1]．Docker を使用してアプリケーションを運用，テスト，デプロイすることで，コードを書いてから本番環境で実行するまでの時間を大幅に短縮することが可能である．こうした背景から，Docker は近年注目されている技術であり，基本的な開発ツールとして約 64%もの開発者が使用している [7].

Docker イメージには，コンテナのファイルシステムが含まれ，アプリケーションの実行に必要な，依存関係，設定，スクリプト，バイナリなどを含まれる [2]．その他にも環境変数やデフォルトのコマンド，その他のメタデータなど，コンテナの設定を含む．また，イメージに含まれるファイルシステムはレイヤ構造を持つ．レイヤはキャッシュ機能を持つため，イメージを再度ビルドする際にビルド時間の短縮を実施することができる [4].

Docker を用いてコンテナ環境を構築するには，Dockerfile と呼ばれるファイルをビルドして生成した Docker イメージ，もしくは Docker レジストリに公開されている Docker イメージを用意し，これらを起動する [1].

例えば，アプリケーション開発の際に，片方の開発者が Docker コンテナを使用して修正後のアプリケーションを共有することで，もう片方の開発者は簡単にテスト環境を構築できる．さらに，修正後の Docker イメージを本番環境に上げるだけでデプロイすることができる．

2.2 Dockerfile

Docker は IaC（Infrastructure as Code）の一つであり，Dockerfile と呼ばれるファイルに，Docker イメージの構築手順をソースコードとして管理することができる [3]．これにより，再現性を持った Docker イメージのビルドを行うことができる [12]．また，Dockerfile では一部の例外的な命令を除き，1 行ごとに命令とその引数という組み合わせで記述する．Dockerfile に記述できる命令は 18 種類存在し，RUN 命令のみ引数を複数行に分割して記述することができる．以下の Dockerfile の例 1[1]では FROM，COPY，RUN，CMD が命令に当たり，それぞれの命令の右側に存在する文字列が引数である．また，この Dockerfile は次の一連の処理を行い，イメージを作成する．

[1] https://docs.docker.jp/develop/develop-images/dockerfile_best-practices.html

ソースコード 1: Dockerfile の例

```
1 FROM ubuntu:18.04
2 COPY . /app
3 RUN make /app
4 CMD python /app/app.py
```

1. FROM 命令を用いてベースイメージを ubuntu:18.04（Ubuntu Linux 18.04）に設定する．
2. COPY 命令を用いて Docker クライアントで操作しているディレクトリ内のファイルを全てコンテナ内のディレクトリ/app にコピーする．
3. RUN 命令を用いて/app 内のアプリケーションを make コマンドでビルドする．
4. CMD 命令を用いてコンテナ起動時に Python アプリケーションである/app/app.py を実行する．

2.3 コードクローン

ソースコード上に存在する同一または類似のコード片のペアをコードクローンまたは単にクローンと呼ぶ．クローンはソースコードのコピーアンドペーストや頻繁に使用されるパターンや表現によって生じる．Inoue はクローンを以下のように分類している [5].

Type-1

空白やタブ，コメントなどの実行できない要素の違いを除き，構文的に等しいクローン．

Type-2

識別子名やリテラル，型名が異なるが，構造的に等しいクローン．

Type-3

予め定義された閾値に収まる範囲で複数のコードが付加もしくは削除，またはコードに変更が施されたクローン．

また，Inoue らによると，ソースコード中にクローンが存在することによりプログラムの保守性が低下することが指摘されており，バグの複数箇所への伝播やソースコードの冗長性などが指摘されている．

2.4 Dockerfile 間のクローン

Dockerfile を扱うプロジェクトではクローンが存在することが報告されている [11] [6]．本節では Dockerfile を扱うプロジェクトにクローンが存在することによる影響と，それに対応するための既存ツールについて述べる．

2.4.1 Dockerfile におけるクローンによる保守性低下

GitHub 上の約 5000 の Dockerfile に対し，Tsuru らが Type-2 クローンの検出を行った結果，Docker 構文だけで約 50,000，RUN 命令内の Shell 構文で約 85,000，計約 200,000 ものクローンセットを検出した（表 1）．

一般的なプログラミング言語を用いたプロジェクトにおいて，クローンの存在は，ソフトウェアの保守性を悪化させる一つの要因であると考えられている [5]．Dockerfile におけるクローンの存在も，一般的なプログラミング言語と同様に，Dockerfile を含むプロジェクトの保守性を低下させる脅威になりうる．例えば，公式の Docker Library では，複数のバージョンや複数のベースイメージに応じてそれぞれ異なる Dockerfile を開発・管理している．しかし，これらの Dockerfile 間にクローンが存在し，そのクローンに修正を施す場合は，クローンを含む全てのファイルが修正の対象になる．これにより，保守性が低下していると考えられる．

Dockerfile 中のクローンによって保守性が低下している事例が，Oumaziz らの調査 [6] により示されている（図 1）．図 1 は Docker Library における Python プロジェクトの Commit 履歴の例 [2]である．この履歴は，Python のライブラリを管理するツール pip のバージョンを変更したことを示している．Python プロジェクトではベースイメージ毎に 8 種類の Dockerfile を管理しており，内 6 種類の Dockerfile にクローンが存在しているため，全てのファイルに変更を適用している．

表 1: Tsuru らが検出した Dockerfile 間の Type-2 クローン数

	Docker 構文	Shell 構文	両方
クローンセグメント数	240,639	458,737	987,597
クローンセット数	50,756	84,011	203,103
1 セット毎のセグメント数	4.74	5.46	4.86
1 セット毎のセグメント長	18.64	12.63	21.30

[2] https://github.com/docker-library/python/commit/af2cf72d9c6c304d041c88db3

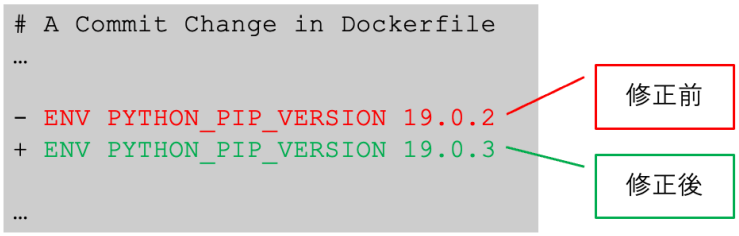

図 1: Dockerfile 修正の例

2.4.2 Dockerfile 間のクローンに対応する既存ツール

Oumaziz らの報告によると，公式の Docker プロジェクトでは，Dockerfile におけるクローンの存在による保守性低下に対応するために，保守ツールとして，検索・置換，テンプレートプロセッサー，生成器の 3 種類が使用されている [6]．ただし，Oumaziz らが調査した全てのプロジェクトは Dockerfile を更新するスクリプトファイルを持つ．また，スクリプトファイルはバージョンやベースイメージを入力パラメータとして持ち，Dockerfile を生成する．

検索・置換 検索・置換は入力データを元に，現存する Dockerfile を更新する．正規表現や Unix の sed コマンドなどを用いて文字列を検索し，置換する．正規表現や sed コマンドなどを使用するだけで良いため，ツールの準備が簡単である．しかし，検索・置換に使用した正規表現や sed コマンドは一つの変更にしか使用できず，多くの変更に対応することができない．また，開発者は変更箇所の重複部分を探し出し，探し出した全ての Dockerfile に手作業で変更を施す必要がある．

テンプレートプロセッサー テンプレートプロセッサーはテンプレートを持ち，テンプレートに対する入力データとテンプレートエンジンを含む．より高度なテンプレートはサブテンプレートやループ，条件分岐を含む．2 つ以上の変更に対応でき，管理している Dockerfile の重複を削減することができる．しかし，Dockerfile を管理しているプロジェクトでは複数のテンプレートを扱うことが多く，テンプレート間で重複が存在することがある．

生成器 bash や perl などのシェル言語を用いて全ての Dockerfile を生成するスクリプト．記述する言語が変数やループ，条件分岐，関数などの多くの機能を持っているため，多くの機能を持つ．生成器は 2 つ以上の変更に対応できることに加え，テキスト置換やループ，条件分岐を使用することで全ての重複箇所を除去することが可能である．一方で，ShellScript などを用いてツールを実装することは非常に面倒であり，Dockerfile とは全く異なる言語を用いてツールを記述していることから，生成する Dockerfile の概要を掴むことが難しい．

以下の表 2 は上述の 3 つのツールに関して利点と欠点を示している．変更にかかる工数は開発者が Dockerfile に変更を施す際の工数に関する評価を表す．また，重複コードの含有度は Dockerfile を管理するプロジェクト内にどの程度重複箇所が存在するかを表す．さらに，導入コストはツールを準備するのに要する時間や難易度を表す．例えば検索・置換の場合，Dockerfile に存在する重複箇所を探し出し，全てに変更を施す必要があるため，Dockerfile の変更にかかる工数は大きい．また，Dockerfile のファイル群をそのまま管理するため重複コードが多く含まれる可能性がある．一方で簡単な命令によって処理ができるためツールの導入コストは低い．以上より，紹介した既存ツールは一長一短の性能を持ち，Dockerfile を管理する統一した手段は存在しない．

2.5 Dockerfile の学習コスト

Dockerfile は特有の処理を記述することが多く，学習コストが高いと考える．例えば Ubuntu をベースイメージに持つ Docker イメージに必要なパッケージをインストールする場合，例 2 のように一連の決まった処理を Dockerfile に記述する必要がある．行ごとの処理についての説明を以下に示す．

1. エラーが発生した場合に処理の中断やエラーの標準出力をする．
2. apt-get コマンドを更新する．
3. apt-get コマンドを用いてパッケージをインストールする．問い合わせに対して全て y と回答し，他の推薦されたパッケージをインストールしない．
4. インストールするパッケージ名
5. インストールするパッケージ名
6. パッケージインストール時に使用したアーカイブファイルを削除する．
7. キャッシュされているパッケージリストを削除する．

表 2: 既存ツールの比較

	変更にかかる工数	重複コードの含有度	導入コスト
検索・置換	多	多	低
テンプレートプロセッサー	中	中	中
生成器	少	少	高

しかし，Stack Overflow [9] でパッケージのインストール方法が質問されているように，Dockerfile の経験が浅い開発者にとって，例 2 のような処理を記述することは，開発の障壁になると言える．また，例 2 は生成するイメージのサイズを削減する処理も含んでいる．イメージを作成する上で必ずしも必要な処理ではないが，Docker におけるベストプラクティスと言える [3]．これらの処理を記述することも経験の浅い開発者にとっては困難であり，ミスを招く原因となる．

ソースコード 2: パッケージインストール

```
1 RUN set -eux \
2     && apt-get update \
3     && apt-get install -y --no-install-recommends \
4     wget \
5     curl \
6     && apt-get -y clean \
7     && rm -rf /var/lib/apt/lists/*
```

3 Myriad

前章で述べた背景を基に，現状の Dockerfile 開発者の問題を解決し，支援するため，本研究では Dockerfile における保守性改善を目的とした言語 Myriad の提案を行う．プログラミング言語の JavaScript に対して TypeScript を導入することで型推論やエディタ上でのエラー発見など，開発者のサポートを図ったように，Myriad は Dockerfile の開発をサポートする．TypeScript は，プログラミング言語やスクリプト言語，マークアップ言語の中で 5 番目に使用されているため [8]，本研究でも解決策として有用であると考えた．本章では Myriad についての要件や設計，その言語処理系の実装，用意したライブラリについて述べる．

3.1 要件・設計

3.1.1 言語要件

Myriad は，Dockerfile が持たない言語機能を複数サポートする．それだけでなく，例 3 のように，エントリーポイントとなる main 関数の中に，Dockerfile をそのまま記述することも可能である．例 3 に記載されたソースコードをコンパイルすると例 1 に記載された Dockerfile が出力される．

ソースコード 3: Myriad のエントリーポイント

```
1 main() {
2     FROM ubuntu:18.04
3     COPY . /app
4     RUN make /app
5     CMD python /app/app.py
6 }
```

さらに，その上で Dockerfile にはない以下の 4 つの機能をサポートしている．

関数　Myriad は関数によるモジュール化機能を持つ．2.4 で紹介したように，複数の Dockerfile を扱うプロジェクトにおいてクローンの存在が報告されている [11]．これは，Dockerfile がモジュール化機能を持たないことに起因すると考える．クローンに対応するための方法として，クローンを関数やメソッドなどのモジュールに集約することが通例である [5]．例 4 に関数定義と関数呼び出しの例を示す．1 行目から 7 行目まで関数定義を行い，12 行目で関数を呼び出している．関数は引数として文字列を受け取ることが可能であり，1 行目の package が受け取る引数の名前を表している．また，12 行目の”curl”で文字列を関数に渡している．例 5 に出力される Dockerfile を示す．

ソースコード 4: Myriad: 関数使用例

```
aptGetInstall(packageName) {
    RUN set -eux \
        && apt-get update \
        && apt-get install -y --no-install-recommends ${packageName} \
        && apt-get -y clean \
        && rm -rf /var/lib/apt/lists/*
}

...

main() {
    aptGetInstall("curl")
}
```

ソースコード 5: Dockerfile: 関数使用結果

```
...
RUN set -eux \
    && apt-get update \
    && apt-get install -y --no-install-recommends curl \
    && apt-get -y clean \
    && rm -rf /var/lib/apt/lists/*
...
```

変数による置換　Myriad は，変数による置換機能を持つ．例 5 の 4 行目のように Myriad で${packageName} と記述した場合，変数 packageName が持つ文字列を参照し，置換する．例 5 の場合，引数 packageName に“ curl”という値が渡されているため，この値で置換される．また，Myriad はプロジェクトのコンパイル時に外部から値を挿入し，その値に応じた処理を実行することが可能である．2.4.1 で紹介したように，複数のバージョンやベースイメージごとに異なる Dockerfile を管理することがある．一致しているコード片を Myriad によるプロジェクトで管理し，異なる箇所だけを外部から挿入できる方が望ましい．例 6 に記載した base や version のように，main 関数内に引数を定義することで，外部から受け取った値をプログラム内で使用することができる．また，外部から受け取った値は大域変数として扱うことができ，プロジェクト内の全ての関数から参照可能である．コンパイル時に第一引数に“ ubuntu”，第二引数に“ 18.0.4”，と値を設定すると，例 7 のような Dockerfile が生成される．

ソースコード 6: Myriad: 外部からの値挿入例

```
main(base, version) {
    FROM ${base}:${version}$
    ...
```

```
4 }
```

ソースコード 7: Dockerfile: 外部からの値挿入結果

```
1 main(base, version) {
2     FROM ubuntu:18.0.4
3     ...
4 }
```

条件分岐　Myriad は if 文による条件分岐機能を持つ．2.4.1 で指摘したように，公式の Docker Library では複数のバージョンやベースイメージごとに異なる Dockerfile を管理する場合がある．この場合，与えられた値によって条件分岐を行い，それぞれに応じたコンテンツを生成する必要がある．例 8 に条件分岐の例を示す．条件節は 2 行目や 4 行目のように丸括弧により囲う．また，条件節では変数と文字列の間で等価性と非等価性を判定することが可能である．さらに，if 文では if 節と else if 節で条件判定を行い，else 節によってそれまで条件節で真にならなかった場合の処理を行う．例 8 では，変数 version が “ 18” という文字列を持つ場合，最初の if 節内の ENV 命令が Dockerfile のコンテンツとして生成される（例 9）.

ソースコード 8: Myriad: 条件分岐例

```
1 ...
2 if (version == "18") {
3     ENV JAVA_VERSION 18.0.2.1
4 } else if (version == "20") {
5     ENV JAVA_VERSION 20-ea+30
6 } else {
7     ENV JAVA_VERSION 21-ea+4
8 }
9 ...
```

ソースコード 9: Dockerfile: 条件分岐結果

```
1 ...
2 ENV JAVA_VERSION 20-ea+30
3 ...
```

3.1.2　トークンの種類

Myriad のソースコード中で使用するトークンの一覧を付録 A に示す.

3.1.3　構文定義

Myriad の構文定義を付録 B に示す.

表 3: 実装した言語処理系のパッケージ毎の詳細

	ファイル数（枚）	行数（行）
main	1	64
tokenizer	2	282
parser	2	660
compiler	3	664
generator	3	184
tyeps	2	80
others	1	51
計	14	1985

3.1.4 コンパイル方法

Myriad のコンパイル方法を付録 C に示す.

3.2 言語処理系の実装

本研究で試作した言語処理系は，Go 言語を用いて実装を行った．Myriad の言語処理系は大きく 4 つのフェーズに分かれ，字句解析，構文解析，意味解析・中間言語生成・Dockerfile 生成の順に処理を行う．それぞれの処理における実装手段を付録 D に示す.

実装したプログラムは 7 個のパッケージ（main，tokenizer，parser，compiler，generator，types，others）で構成されており，それぞれのファイル数とコードの行数を表 3 に示す.

3.3 ライブラリ

Myriad は 2.5 で述べた Dockerfile の学習コストの高さを低減するために，ライブラリによって Dockerfile に存在する頻出パターンを扱うことができる．ライブラリは Tsuru 氏と共同研究を行い，Tsuru 氏が作成したツールを用いて既存の Dockerfile プロジェクトから検出したクローンや，経験則による頻出パターンを抽出し，作成した．例 10 では，ライブラリから使用したい関数をインポートし，main 関数の中で使用している．ライブラリからインポートした関数は例 4 で紹介した関数と同じように使用する．例 11 に Dockerfile の出力結果を示す．また，用意したライブラリの一覧を付録 E に示す.

ソースコード 10: Myriad: ライブラリ使用例

```
1 IN lib/library.my
2 ...
3 aptGetInstall(packageName) {
```

```
    RUN set -eux \
        && apt-get update \
        && apt-get install -y --no-install-recommends ${packageName} \
        && apt-get -y clean \
        && rm -rf /var/lib/apt/lists/*
}
...
-------------------------------------------------------------------------

IN main.my
import aptGetInstall from "./lib/library.my"

main() {
    ...
    aptGetInstall("curl")
    ...
}
```

ソースコード 11: Dockerfile: ライブラリ使用結果

```
...
RUN set -eux \
    && apt-get update \
    && apt-get install -y --no-install-recommends curl \
    && apt-get -y clean \
    && rm -rf /var/lib/apt/lists/*
...
```

4　評価

Myriad の性能を評価すべく，２つの被験者実験（可読性に関する比較実験・ライブラリに関する比較実験）と定量的な評価（保守性に関する比較実験）を行った．可読性に関する比較実験では，Myriad の可読性について，既存ツールとの比較を行うために，被験者実験を実施した．また，ライブラリに関する比較実験では Myriad がサポートしているライブラリの有用性を評価するために，被験者実験を実施した．さらに，保守性に関する比較実験では Dockerfile に修正を施す場面を想定し，修正にかかる工数を計測した．

4.1　可読性に関する比較実験

4.1.1　実験手法

比較対象は公式の Docker Library が管理している，OpenJDK のプロジェクト [3]で用いられてる既存ツールである．OpenJDK のプロジェクトは，3 種類のバージョンと 9 種類の異なるベースイメージ（内 6 種類が Linux ベース，3 種類が Windows ベース）の計 27 個の Dockerfile を，2.4.2 で説明したテンプレートを用いて管理している．例えば OpenJDK で使用されているテンプレートは，例 12 のような構造を持つ．2,3 行目では，is_oracle という変数に，ベースイメージが OracleLinux であるかどうかの真偽値を代入している．さらに，5,6 行目では oracle_version という変数に，OpenJDK の Docker コンテナにインストールされる Java のバージョンを代入している．9-16 行目では条件分岐処理を行い，is_oracle が真であれば 12 行目が Dockerfile に出力される．また，12 行目の中括弧（{}）で囲まれた部分はバージョンを表す文字列が置換されて表示される．

ソースコード 12: OpenJDK で使用されているテンプレートの例

```
1  {{
2      def is_oracle:
3          env.variant | startswith("oraclelinux")
4          ;
5      def oracle_version:
6          env.variant | ltrimstr("oraclelinux")
7  -}}
8  ...
9  {{
10     if is_oracle then (
11 -}}
12 FROM oraclelinux:{{ oracle_version }}-slim
13 ...
14 {{
```

[3]https://github.com/docker-library/openjdk

```
15      ) end
16  -}}
```

本実験では空白や空行，コメントなどの作成されるイメージに関係のない要素の違いを除き，OpenJDK と同じ Dockerfile 群を生成するプロジェクトを Myriad で作成し，OpenJDK における既存ツールとの比較を行った．全ての Dockerfile を生成するのに必要な OpenJDK のファイル群を，表 4 に示す．拡張子が template のファイルには，生成したい Dockerfile のテンプレートが記述されている．ベースイメージによって異なる箇所は，条件分岐による生成部分の変更や，置換機能を用いて表現されている．apply-templates.sh は，ShellScript によって記述されたスクリプトファイルであり，versions.sh からベースイメージやハッシュ値などのデータを読み取り，テンプレートから全ての Dockerfile を出力する処理が記述されている．versions.json は，ベースイメージやハッシュ値などの，テンプレートに適用するデータが含まれた JSON ファイルである．.jq-templates.awk は，テンプレート内の条件分岐や置換などの処理を定義している．本来ならば apply-templates.sh が実行される際に外部 URL からダウンロードされるファイルであるが，ダウンロードの過程は Dockerfile の生成に直接関与しないため，予めプロジェクト内に存在するものとした．ただし，OpenJDK のプロジェクト内のファイルにはファイルのダウンロードやコメントなどの Dockerfile の生成に直接関与しない処理が存在する．可読性の評価に影響を与えないように予めこれらの処理は削除しておいた．

比較対象となる Myriad のファイル群を表 5 に示す．main.my は，Myriad で記述されたエントリーポイントとなるメインファイルである．images はディレクトリであり，ベースイメージ毎の関数をそれぞれファイルとして集めている．generate.sh はバージョンやベースイメージの情報を持ち，その情報を元に Myriad で記述されたプログラムのコンパイルを行う．これにより，全ての Dockerfile を出力する．myriad は Myriad のコンパイラであり，バイナリファイルである．ただし，OpenJDK における.jq-templates.awk と，Myriad における myriad は，どちらも Dockerfile の開発者が実装するファイルではないため，可読性の評価には含めないものとする．

本実験では，9 人の被験者に対し，事前に Docker，Dockerfile，Myriad，テンプレートの基本的な知識を与えた．その上で，1 つのプロジェクトにつき 10 分間与え，上記の 2 プロジェクトを順番に読んでもらった．ただし，評価の公平性を保つために，被験者を 2 グループに分け（グループ 1 は 5 人，グループ 2 は 4 人），1 グループは Myriad プロジェクトから，2 グループは OpenJDK プロジェクトから読んでもらうことにした．両方のプロジェクトを読んでもらった後，5 段階評価のアンケートを取り，どちらが可読性に優れているかを調査した．Myriad の方が可読性に優れていると感じた場合は 1 を，OpenJDK の方が優れ

表 4: 可読性比較実験に用いた OpenJDK プロジェクト内のファイル構成

Dockerfile-Linux.template	Linux ベースのテンプレートファイル
Dockerfile-Windows.template	Windows ベースのテンプレートファイル
apply-templates.sh	全ての Dockerfile を生成するスクリプトファイル
versions.sh	テンプレートに適用するデータが入った JSON ファイル （ベースイメージやハッシュ値など）
.jq-tempaltes.awk	テンプレートファイル内の処理を記述している awk ファイル （apply-templates.sh を実行した際に外部 URL から取得する）

表 5: 可読性比較実験に用いた Myriad プロジェクト内のファイル構成

main.my	main 関数を含むファイル
images	ベースイメージ毎の関数を集めたディレクトリ （4 種類のベースイメージファイル，その他のファイルを集めたファイル）
generate.sh	全ての Dockerfile を生成するスクリプトファイル
myriad	Myriad のコンパイラ

ていると感じた場合は 5 を選択してもらう．また，5 段階評価の理由もアンケートとして収集した．

4.1.2 実験結果

9 人の被験者に 5 段階評価のアンケートを取った結果を図 2 に示す．図 2 ではグループ 1，グループ 2，合計の棒グラフをそれぞれ示す．9 人中 7 人が 1 を選択し，2 人が 2 を選択した．これらの結果から，Myriad で作成したプロジェクトは既存ツールを用いた OpenJDK のプロジェクトよりも可読性が高いということが読み取れる．

実験を通して被験者から得た感想を，OpenJDK の既存ツールと Myriad に分け，それぞれの長所と短所を以下にまとめる．

既存ツール（テンプレート） 中括弧（{}）を用いることで，Dockerfile の内容と処理の内容を区別することができ，可読性が高いという評価があった．一方で，中括弧が多く使用されていることから処理の概要を掴みにくいという評価もあった．さらに，変数の定義方法や中括弧など，普段使用しているプログラミング言語とは異なる，見慣れない記法を用いていることで，プログラムを読みづらいという評価があった．さらに，ファイルが長く，機能毎にまとまっていないことから，可読性が低いという意見もあった．総じて既存ツールについ

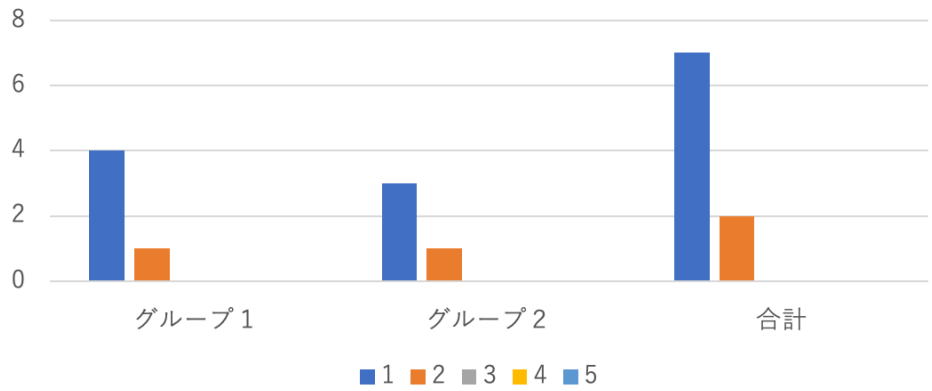

図 2: 可読性評価のアンケート結果

て評価している被験者は少なく，既存ツールを評価する趣旨の感想を書いたのは 9 人中 1 人に留まった．

Myriad　9 人中 5 人が，関数の使用によるモジュール性の高さや，関数名による可読性向上について評価した．さらに，9 人中 4 人が，ファイルをベースイメージ毎に分割して記述することで，可読性が向上していると述べた．また，9 人中 2 人が，Myriad が自身にとって馴染みのある言語仕様であり，理解しやすかったと述べた．これは，Myriad が多くのプログラミング言語で採用されている，手続型パラダイムを採用していることや，関数や if 文など，多くのプログラミング言語で使用されている機能をサポートしていることによるものだと考えられる．一方で，Myriad はファイル数が多いため可読性が低下しているという意見や，ベースイメージが増加することによる依存関係が複雑化しているという指摘を頂いた．さらに，グローバル変数を他の関数で参照することが原因で，関数間の変数の追跡が難しいという意見もあった．

4.2　ライブラリに関する比較実験

4.2.1　実験手法

4.1.1 と同じ被験者に参加してもらい，同じく 2 グループに分割し，実験を行なった．両方のグループに等しくテーマを与え，テーマに沿った Dockerfile を作成してもらう．ただし，グループ 1 は Myriad を用いて 5 つのライブラリが使用できる状態にし，グループ 2 は Myriad を用いずにライブラリが使えない状態で Dockerfile をそのまま記述してもらった．制限時間は 20 分とし，目的の Myriad ソースファイル，または Dockerfile が完成するまでの時間を計測した．ただし，作成したファイルの完成度は本論文の著者が判定した．シェルコマ

ンドの引数など，細かい違いは完成度の条件に含めず，ある程度テーマに沿った Dockerfile が完成していた場合は，そこで計測を終了する．また，Myriad のライブラリを使用したグループ 1 には，実験終了後にアンケートを実施し，ライブラリの使用感を集計した．

4.2.2 実験結果

実験で計測した時間は，以下のようになった（表 6）．被験者の計測時間は全被験者の計測時間を記載しており，20 は全て制限時間内に記述できなかったことを示している．本実験結果より，Dockerfile をそのまま記述する方が，Myriad を用いてライブラリ有りの状態で記述するよりも，早く目的の Dockerfile を作成できることが判明した．

ライブラリを用いた被験者からは，Dockerfile やシェルコマンドの使い方の詳細を知らなくても記述できることを評価する意見や，ライブラリ使用により記述時間が短縮できるという感想を得た．一方で，複数のライブラリの中から使用できるライブラリを探すのに時間を要したという意見や，適切なライブラリを選択するのが難しかったという意見も得た．

以上の実験結果より，ライブラリの導入は Dockerfile の学習コストを下げる反面，ライブラリの詳細を把握するのに時間を要するということが考えられる．今回の実験では，ライブラリはメソッドと同じく関数のソースコードとして与えたため，ライブラリ内のソースコードを読んだ上で使用できるライブラリを選択する必要があった．これはドキュメントなどを用意することでライブラリを把握するコストを下げられるのではないかと考える．

4.3 保守性に関する比較実験

4.3.1 実験手法

Myriad の保守性を評価すべく，プロジェクトで扱う Dockerfile に加える修正箇所の数を定量的に測定した．比較対象は以下の 3 つである．

- Myriad
- ツール無し

表 6: ライブラリに関する比較実験における計測時間

	被験者の計測記録（分）	平均値（分）
グループ 1 （ライブラリ有り）	20，20，20，14，12	17.2
グループ 2 （ライブラリ無し）	13，10.5，17，13	13.4

- OpenJDK プロジェクトで使用されている既存ツール（テンプレート）

Myriad のプロジェクトでは，4.1 で行ったように，OpenJDK のプロジェクトを Myriad で模倣する．ただし，本実験では修正を施す場合に保守性が高くなるように，プログラムのモジュール性を意識して模倣した．Myriad で作成したプロジェクトのファイル群を表 7 に示す．4.1 で作成したプロジェクトと異なる箇所は，generate.sh が versions.json から全てのパラメータ（ベースイメージやハッシュ値など）を読み取っていることと，Dockerfile で使用される命令のクローンを全てモジュール化していることである．

想定する修正対象は以下の 2 箇所である．

- アプリケーションのバージョン更新．具体的にはバージョンを 18.0.2.1 から 18.0.2.2 に変更する場合を想定する（例 13）．
- RUN 命令内のコマンド削除．具体的には javac コマンドと java コマンドを用いて Java のバージョンを出力する箇所を削除する場合を想定する（例 14，15，16）．

ソースコード 13: Dockerfile 中の修正対象

```
1 ...
2 ENV JAVA_VERSION 18.0.2.1
3 ...
```

ソースコード 14: Linux ベースの Dockerfile 中の削除対象

```
1 ...
2 javac --version; \
3 java --version
4 ...
```

表 7: 保守性比較実験に用いた Myriad プロジェクト内のファイル構成

main.my	main 関数を含むファイル
images	ベースイメージ毎の関数を集めたディレクトリ （4 種類のベースイメージファイル，その他のファイルを集めたファイル）
generate.sh	全ての Dockerfile を生成するスクリプトファイル
versions.json	テンプレートに適用するデータが入った JSON ファイル （ベースイメージやハッシュ値など）
myriad	Myriad のコンパイラ

ソースコード 15: Windows ナノサーバーベースの Dockerfile 中の削除対象

```
1 ...
2 && echo javac --version && javac --version \
3 && echo java --version && java --version \
4 ...
```

ソースコード 16: Windows サーバコアベースの Dockerfile 中の削除対象

```
1 ...
2 Write-Host ' javac --version'; javac --version; \
3 Write-Host ' java --version'; java --version; \
4 ...
```

4.3.2 実験結果

修正箇所の数は以下の通りである（表 8）．以上から，それぞれの想定した状況に要する

表 8: 修正箇所数（行）

	Myriad	ツール無し	既存ツール（テンプレート）
バージョン更新	7	24	7
コマンド削除	6	54	6

修正箇所は Myriad と既存ツールの間に差はなく，ツール無しの場合のみ多いということが読み取れる．このような結果となったと考えられる理由を以下にそれぞれの状況に応じて述べる．

- バージョンの更新では例 13 に意外にも Java のバージョンに依存する URL やハッシュ関数も変更する必要がある．Myriad と既存ツールはこれらの情報を Json ファイルに保管しているため，このファイルの中身を書き換えるだけで良い．
- コマンドの削除ではコンテナがベースとする OS の種類によって異なる箇所を削除する必要がある．しかし，Linux ベースの Dockerfile と Windows サーバーコアベースの Dockerfile はそれぞれ同じ処理を記述を行う．よって，Myriad では関数を，既存ツールではテンプレートを使って重複部分を取り除いているため，修正箇所を最小限に抑えることができる．

5 関連研究

5.1 Modus

Docker イメージを構築する場合，ベースイメージやアプリケーションのバージョンなど複数のパラメータをビルドシステムに与える必要がある [10]．これらのパラメータは相互に依存し，片方のパラメータを設定するともう片方のパラメータが制限されてしまうことがある．しかし，Dockerfile はこのようなパラメータの相互作用を制約することができないため，2.4.2 に記述したようなその場凌ぎのツールを使用する必要があった．これらの問題点に対応するため，Tomy らは Dockerfile の代わりに，Docker イメージを構築できる言語 Modus を開発した．Modus は論理型言語であり，Docker イメージの生成を行う．Modus の主な機能は以下の 5 点である．

- パラメータの依存関係解消
- ビルドの並列化
- イメージサイズの削減
- モジュール機能

大きく Myriad と異なる点は以下の 4 点である．

- Modus は論理型言語であるが，Myriad は手続き型言語である．
- Modus は Docker イメージをビルドするが，Myriad は Dockerfile を生成する．
- Modus はライブラリを持たないが，Myriad は持つ．
- Modus はイメージサイズの最適化を行うが，Myriad は行わない．

上述の Modus と Myriad の違いを以下の表 9 にまとめる．

表 9: Modus と Myriad の違い

	言語パラダイム	生成物	ライブラリ	イメージサイズの最適化
Modus	論理型	Docker イメージ	なし	行う
Myriad	手続き型	Dockerfile	あり	行わない

6　あとがき

本研究では Dockerfile の保守性改善を目的とした言語 Myriad の提案とその処理系の試作を行ない，評価を行なった．可読性に関する比較実験では，既存ツールと比較した場合に，Myriad の可読性が優れているという評価を得た．一方で，管理する Dockerfile のベースイメージが増加することによる依存関係の複雑化や，グローバル変数を他の関数で参照することによる保守性低下などが懸念点として指摘された．また，ライブラリに関する比較実験では，Myriad のライブラリが Dockerfile の学習コストを下げるという評価を得た．一方でライブラリを用いた場合と Dockerfile をそのまま記述する場合の記述時間を計測した結果，後者の方が早いという結果を得た．さらに，保守性に関する比較実験では，Dockerfile の修正にかかる工数を計測し，Myriad を用いた場合，ツールを使用しない場合と比べて工数が少なくて済み，既存ツールと同等の工数がかかるという結果を得た．

本研究における今後の課題として，言語機能の拡張とライブラリのドキュメント作成が考えられる．それぞれの詳細を以下に述べる．

- 本研究で提案した言語 Myriad は言語機能が不十分であり，機能の拡張が必要である．例えば変数定義や，変数値の関数への受け渡し，のような機能が挙げられる．Myriad はソースコード内で変数定義を行うことができず，変数を用いる場合は外部から引数として値を与えるか，関数の引数として値を渡す必要がある．この場合，変数の値の管理が難しく，その他のスクリプト言語による外部からの値挿入などに頼らざるを得ない．これにより，複数の言語（Myriad とその他のスクリプト言語）を使用する必要があり，保守性が低下する．よって，変数定義機能をサポートすべきである．また，関数呼び出し時の引数として与えることができる値は文字列のトークンのみであり，定義された変数の値を与えることができない．例 17 では main 関数内で定義されている変数 base を関数 image に引数として渡そうとしているが，現状このような処理を行うことができない．可読性に関する実験を行なった際に指摘されたように，関数内で関数を呼び出す側の変数値を参照したい場合は main 関数の引数をグローバル変数として参照せざるを得ない．これにより変数値の追跡が難しくなり，保守性の低下につながる．よって，変数値を関数に受け渡せる機能が必要である．

- 実験では使用するライブラリを見つけ出すために時間を要している被験者が多く見られた．現状のライブラリでは，ライブラリの関数名である程度の処理を予想できるが，引数の使い方などは関数内のソースコードを読む必要があった．よって，ライブラリを把握するコストを削減するためにドキュメントを作成することが必要である．

ソースコード 17: 変数値を関数に渡せない例

```
main(base, version) {
    ...
    image(base)
    ...
}
```

謝辞

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 肥後芳樹 教授には，特別研究を進めるにあたり，研究室内の発表機会において多くの御意見，御助言をいただきました．研究に関する妥当性や発表スライドに対する御指摘は的確で特別研究の質を高める手助けとなりました．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下誠 准教授には，特別研究を進めるにあたり，直接御指導いただきました．毎週の打ち合わせなどで研究に関する相談に乗っていただき，研究の進行や論文執筆など，多くの場面で御助言を賜りました．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 神田哲也 助教には，特別研究を進めるにあたり，研究室内の発表機会において多くの御意見，御助言をいただきました．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻ソフトウェア工学講座 事務職員 軽部 瑞穂 氏には，私たち学生が快適な研究生活を送れるよう研究室内の環境を整備していただきました．研究室内外で声を掛けていただくことが多く，研究活動を進める上で心の支えになりました．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 鶴智秋 氏には，特別研究を進めるにあたり，研究のいろはを教えていただきました．鶴氏の御指摘は的確で，発表スライドや論文の書き方を御教授いただきました．また，本研究のキーアイデアを共に考えていただきました．

また，その他様々な御指導，御助言等を賜りました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻肥後研究室の皆様に，心より深く感謝申し上げます．

参考文献

[1] Docker Inc. docker docks: Docker overview. https://docs.docker.com/get-started/overview/. [Online; accessed 7. Feb. 2023].

[2] Docker Inc. docker docs: Get started overview. https://docs.docker.com/get-started/. [Online; accessed 7. Feb. 2023].

[3] Docker Inc. Dockerfile のベスト・プラクティス. https://docs.docker.jp/develop/develop-images/dockerfile_best-practices.html. [Online; accessed 7. Feb. 2023].

[4] Docker Inc. Get started: Part 9: Image-building best practices. https://docs.docker.com/get-started/09_image_best/. [Online; accessed 7. Feb. 2023].

[5] Katsuro Inoue. Introduction to code clone analysis. In *Code Clone Analysis: Research, Tools, and Practices*, pp. 1–25. Springer, 2021.

[6] Mohamed A. Oumaziz, Jean-Remy Falleri, Xavier Branc, Tegawende F. Bissyande, and Jacques Klein. Handling duplicates in dockerfiles families: Learning from experts. In *Proc. International Conference on Software Maintenance and Evolution*, pp. 524–535, 2019.

[7] Stack Overflow. 2022 developer survey. https://survey.stackoverflow.co/2022/#section-most-popular-technologies-other-tools. [Online; accessed 7. Feb. 2023].

[8] Stack Overflow. 2022 developer survey. https://survey.stackoverflow.co/2022/#section-most-popular-technologies-programming-scripting-and-markup-languages. [Online; accessed 7. Feb. 2023].

[9] Stack Overflow. Cannot install packages inside docker ubuntu image. https://stackoverflow.com/questions/27273412/cannot-install-packages-inside-docker-ubuntu-image/27273543#27273543. [Online; accessed 7. Feb. 2023].

[10] Chris Tomy, Earl T. Barr, Timgmao Wang, and Sergey Metchtaev. Modus: A datalog dialect for building container images. In *Proc. Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 595–606, 2022.

[11] Tomoaki Tsuru, Tasuku Nakagawa, Shinsuke Matsumoto, Yoshiki Higo, and Shinji Kusumoto. Type-2 code clone detection for dockerfiles. In *Proc. International Workshop on Software Clones*, pp. 1–7, 2021.

[12] Yang Zhang, Huaimin Wang, Bogdan Vasilescu, and Vladimir Filkov. One size does not fit all: An empirical study of containerized continuous deployment workflows. In *Proc. Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 295–306, 2018.

付録 A　Myriad のトークン

“**import**” “**from**” “**main**” “**if**” “**else if**” “**else**” “(” “)” “**{**” “**}**” “**==**” “**!=**”
“**Dockerfile 命令**” “**生の Dockerfile 引数**” “**置換用変数**” “**名前**” “**文字列**”

上記に含まれているトークンのうち，Dockerfile 命令，置換用変数，名前，文字列を拡張バッカス・ナウア記法（EBNF：Extended Backus-Naur Form）で以下に定義する．

Dockerfile 命令 = “RUN” | “CMD” | “LABEL” | “MAINTAINER” | “EXPOSE” | “ENV” | “ADD” | “COPY” | “ENTRYPOINT” | “VOLUME” | “USER” | “WORKDIR” | “ARG” | “ONBUILD” | “STOPSIGNAL” | “HEALTHCHECK” | “SHELL” (1)

置換用変数 = “$” “{” 名前 “}”. (2)

名前 = 英小文字 { 英字 | 数字 }. (3)

文字列 = “{ ”以外の文字 }”. (4)

英字 = 英小文字 | 英大文字. (5)

英小文字 = “a” | “b” | . . . | “z”. (6)

英大文字 = “A” | “B” | . . . | “Z”. (7)

数字 = 0 | 1 | . . . | 9. (8)

付録 B　Myriad の構文定義

以下に Myriad の構文定義を拡張バッカス・ナウア記法（EBNF：Extended Backus-Naur Form）で示す.

プログラム = { 関数インポート文 } { 関数 } メイン部 . (9)

関数インポート文 = “import” 関数名 “from” ファイル名 . (10)

ファイル名 = 文字列 . (11)

関数 = 関数名 引数宣言部 関数記述部 . (12)

メイン部 = “main” 引数宣言部 関数記述部 . (13)

引数宣言部 = “(” [引数群] “)” . (14)

引数群 = 変数 { “,” 変数 } . (15)

変数 = 変数名 . (16)

関数記述部 = “” 記述部 “” . (17)

記述部 = 記述ブロック { 記述ブロック } . (18)

記述ブロック = Dfile 文 | 関数呼び出し文 | ifブロック . (19)

Dockerfile 文 = [Dockerfile 命令] Dockerfile 引数部 . (20)

Dockerfile 引数部 = Dockerfile 引数 { Dockerfile 引数 } . (21)

Dockerfile 引数 = 代入用変数 | 生の Dockerfile 引数 . (22)

関数呼び出し文 = 関数名 “(” [文字列の並び] “)” . (23)

文字列の並び = 文字列 { “,” 文字列 } . (24)

if ブロック = “if” “(” 条件判定式 “)” “{” 記述部 “}” { elif節 } [else 節] . (25)

条件判定式 = 式 比較演算子 式 . (26)

式 = 変数名 | 文字列 . (27)

elif 節 = “else if” “(” 条件判定式 “)” “{” 記述部 “}” . (28)

else 節 = “else” “{” 記述部 “}” . (29)

比較演算子 = “ == ” | “! = ” . (30)

関数名 = 名前 . (31)

変数名 = 名前 . (32)

付録 C　Myriad のコンパイル方法

Myriad の言語処理系は次の 3 つを引数に持ち，Myriad ファイルを Dockerfile に変換する．

- エントリーポイントを持つ Myriad ファイルのパス．
- 出力先の Dockerfile のファイルパス．
- プログラムに与える引数．

以下に Myriad ファイルを Dockerfile にコンパイルする例を示す．

```
$ ./myriad ./src/main.my Dockerfile buster 18
```

上記のコンパイル例で使用された引数をそれぞれ以下で説明する．

- ./myriad：言語処理系のバイナリファイルのパス．
- ./src/main.my：エントリーポイントを持つ Myriad ファイルのパス．
- Dockerfile：出力先の Dockerfile のファイルパス．
- buster：プログラムに与える第一引数．
- 18：プログラムに与える第二引数．

付録 D　言語処理系の実装方法

付録 D.1　字句解析

字句解析では Myriad で書かれたファイル内のテキストを 3.1.2 で示したトークンに分解し，プログラム中にトークン列として持たせることを目的とする．基本的な実装方針は次の通りである．まずはファイルを行単位で分解し，それぞれの行を前から順に読み込む． 3.1.2 で示したトークンと一致した時にトークン列にそのトークンを追加する．

付録 D.2　構文解析

構文解析では字句解析で切り出したトークン列が 3.1.3 で示した構文定義に従っているか解析する．プログラムは再帰構造を持ち，呼び出されたメソッドとその時にインデックスが指している列中のトークンが等しいかどうかを判定する．例 18 は式 10 の構文解析を行う実装例である．関数インポート文を解析するメソッドが呼び出された場合，順にトークンの検査を行う．構文にトークンが含まれる場合はそのトークンに一致するかどうかを判定し，一致してる場合は次のトークンを検査する．一致しない場合はエラーを出力する．構文に句が含まれる場合はその句の構文を解析するメソッドを呼び出す．

ソースコード 18: 構文解析における再帰構造の実装例

```
1  // 関数インポート文を解析するメソッド
2  functionImportSentence() {
3      // トークンが“import"かどうか判定する
4      if (token is “import") {
5          index++ // 次のトークンを指し示す
6      } else {
7          output error // エラーを返す
8      }
9
10     // 関数名を読み込むメソッドを呼び出す
11     index = functionName()
12
13     // トークンが“from"かどうか判定する
14     if (token is “from") {
15         index++ // 次のトークンを指し示す
16     } else {
17         output error // エラーを返す
18     }
19
20     // ファイル名を読み込むメソッドを呼び出す
21     index = fileName()
22 }
```

付録 D.3　意味解析・中間言語生成

意味解析　意味解析ではプログラムの意味的な誤りを発見し，エラーを出力する．意味解析において発見する誤りの種類とその発見方法を以下に示す．

- 未定義変数の使用：予め Myriad プログラム中ので宣言された関数毎に変数名を登録しておき，変数が使用された時に変数表に登録されているか検査する．
- 未定義関数の呼び出し：予め Myriad プログラム中で宣言された関数名を登録しておき，関数が呼び出された時に登録されているか検査する．
- 関数定義時と関数呼び出し時における引数の個数の不一致：予め Myriad プログラム中で宣言された関数毎に引数の個数を登録しておき，関数呼び出し時に引数の個数が一致しているか検査する．

中間言語生成　Dockerfile の生成をおこなう前に中間言語の生成を行う．これは Dockerfile の生成に無意味なトークンを除去し，簡潔な構造にするためである．定義された関数毎に中間言語の生成を行い，中間言語のトークン列として管理しておく．中間言語のトークン列で扱うトークンとその説明を表 10 に示す．
例えば例 19 のようなソースコードを対象に main 関数の中間言語生成を行なった場合，表 11

表 10: 中間言語で扱うトークンの種類

トークン名	トークンの説明
ROW	Dockerfile にそのまま出力するトークン
VAR	置換対象の変数
CALLFUNC	関数呼び出し （呼び出す際の引数値を含む）
IF	if 節の開始トークン （条件判定式の情報を含む）
ELIF	else if 節の開始トークン （条件判定式の情報を含む）
ELSE	else 節の開始トークン
ENDIF	if 節の終了トークン

のようなトークン列が生成される．

ソースコード 19: 中間言語の生成対象の例

```
1 import imageFromUbuntu from "./images/functions.my"
2
3 main(base) {
4     FROM ${base}:18.0.4
5     if (base == "ubuntu") {
6
7     }
8     ...
9 }
```

表 11: 生成される中間言語のトークン列

トークン名	トークンが有する情報
ROW	コンテンツ：“ FROM”
VAR	変数名：“ base”
ROW	コンテンツ：“ :18.0.4”
IF	条件判定条件： { 左辺：{ 変数名，“ base”}，右辺：{ 文字列，“ ubuntu”}，条件：“ ==”}
ENDIF	

付録 D.4　Dockerfile 生成

中間言語から Dockerfile を生成する．ここでは中間言語内のトークン VAR の変数を文字列に置換したり，if 文の判定を行って生成したいコンテンツを選択したりする．

付録 E　作成したライブラリの一覧

表 12: 作成したライブラリの一覧

ライブラリ名	ライブラリの説明
gitCloneAndMoveToDir	GitHub のリポジトリからソースファイルをクローンし， カレントディレクトリをクローンしたディレクトリに移動する
aptGetInstall	apt-get コマンドを用いてパッケージをインストールする
yumInstall	yum コマンドを用いてパッケージをインストールする
apkInstall	apk コマンドを用いてパッケージをインストールする
makeWithWget	wget コマンドを用いてソースファイルをダウンロードし， リポジトリ内のアプリケーションを make コマンドでビルドする