豊田工業高等専門学校　情報工学科
Department of Information and Computer Engineering,
National Institute of Technology, Toyota College

卒業論文
Graduation Thesis
February 22, 2024

コンパイラ学習者のための Yacc 用言語サーバの開発

稲垣研究室　　石部 鳳空

あらまし　Yacc は大学のコンパイラの講義に採用されていたりコンパイラ開発者に使用されたりしている LALR(1) パーサジェネレータである．現在 Yacc コーディング時に利用できる静的解析に基づく支援ツールとして VSCode 拡張のものが存在するが，他の主要なエディタでは同等のコーディング支援機能は提供されていない．そこで本研究では，LSP に則った Yacc の言語サーバを開発した．各エディタで Yacc ファイルを開いた時に，作成した言語サーバを実行するようにしたことで，異なるエディタで静的解析に基づくコーディング支援機能が容易に得られることが確認できた．

キーワード　コンパイラ学習支援，コーディング支援，Yacc，Bison，LSP，言語サーバ，静的解析

1. 背景

1.1. Yacc とは

Yacc は高専や大学のコンパイラの講義やコンパイラ開発者，研究者などに広く使用されている一般的な LALR(1) パーサジェネレータである．Yacc ファイルには終端記号と非終端記号を定義し，それらを参照しながら BNF で構文規則と対応する意味動作を記述する．作成したファイルを Yacc で処理すると，内部でオートマトン (実体としては構文解析表である) とスタックを生成し，構文解析関数がこれらを操作して構文解析を実行するため，ユーザはこの関数を呼び出すだけで構文解析が実行できる．

コンパイラの授業の演習などで Yacc を採用している事例として，豊田高専専攻科情報科学専攻の「コンパイラ」[1] や岡山大学工学部情報系学科「コンパイラ」[2]，電気通信大学情報理工学域「言語処理系論」[3] などがある．また，プリンストン大学のコンパイラの講義の教科書である『最新コンパイラ構成技法』[4] では構文解析器を開発するために ML-Yacc(Yacc の standard ML 実装) を使用している．このような使用例から分かるように，Yacc は学生やコンパイラ学習者に記述されていると言える．

パーサジェネレータを使用する場面での Lex/Yacc アプローチ以外の主要な選択肢として ANTLR を用いることがある．先行研究では，多くの大学の講義で Lex/Yacc アプローチを採用しているという事実があるが，実験の結果，ANTLR を用いた方が学生の試験合格率や最終的な成績が高いことが示されている [5]. また，学生のアンケートでの意見によると，ANTLR は Lex/Yacc よりもシンプルで直感的で保守性が高いと評価されている．つまり，この実験では Yacc が他のツールと比較してユーザにとって実装の難易度が高いことが示唆されている．

1.2. Yacc の周辺ツールの現状

Yacc の周辺ツールについて，理解の補助のためのツールとして Leon Aaron Kaplan による“yaccviso − a tool for visualizing yacc grammars”[6] や楠目勝利らによる「コンパイラにおける構文解析過程の視覚化」[7] が提案されている．Leon が開発した yaccviso は，Yacc ファイルを入力し，終端記号と非終端記号の依存関係をグラフで表示するものであり，楠目らが開発したツールは，Bison が生成した構文解析表のファイルと LALR(1) 構文解析器のログから，構文解析過程の動作アニメーションを生成するものである．ここでは先読み記号やスタックの状態，解析木，還元時に適用した生成規則を表示している．これと同様のアプローチで開発されたツールとして“Parser Visualizations for Developing Grammars with Yacc”[8] がある．

また，コーディング時の補助ツールとして，シンタックスハイライトを施すもの (vscode-yacc[9]) や静的解析を行うもの (Yash[10]) が存在する．しかし，これらは Visual Studio Code (VSCode) 拡張として提供されているソフトウェアであり，Vim や Emacs などの他の主要なエディタでは同等の支援機能が得られない．特に静的解析によるコーディング支援を提供するものは Yash のみであるが，これは現時点 (2024 年 2 月 8 日時点) で 73,000 回以上インストールされており，Yacc コーディング時の静的解析による支援機能の提供への需要が存在することが分かる．また，セクション 1.1 で示したように，Yacc に初めて触れる学生にとって，Yacc を用いて構文解析器を開発することは比較的難易度が高い作業である．

この先行研究では，Yacc よりも ANTLR を用いた方が良い結果を得られたことの理由として，ANTLR が提供するプライグインサポートと TestRig テストツールの存在について言及しており，Yacc の記述を困難にしている原因の一つにコーディング時に利用できるツールなどのサポート不足があると推測できる．

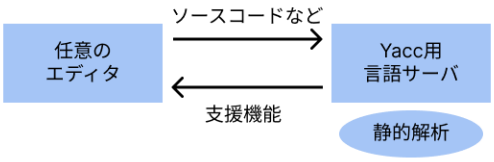

図 1: 言語サーバを利用したコーディング支援

ソースコード 1: Bison の記述例

```
%{
    #include <stdio.h>
    #include <stdlib.h>

    extern int yylex();
    extern char *yytext;
    extern FILE *yyin;

    void yyerror(const char *s) {
        fprintf(stderr, "error: %s\n", s);
    }
%}

%token NUMBER
%token PLUS MINUS MULTIPLY DIVIDE
%token LPAREN RPAREN

%%
/* definition of grammar rules */
expr:
    expr PLUS term   { $$ = $1 + $3; }
    | expr MINUS term  { $$ = $1 - $3; }
    | term           { $$ = $1; }
    ;

term:
    term MULTIPLY factor  { $$ = $1 * $3; }
    | term DIVIDE factor    { $$ = $1 / $3; }
    | factor               { $$ = $1; }
    ;

factor:
    NUMBER            { $$ = atoi(yytext); }
    | LPAREN expr RPAREN  { $$ = $2; }
    ;

%%
/* main function */
int main() {
    printf("input expression: ");
    yyparse();
    return 0;
}
```

2. 問題と解決法

セクション 1.2 で述べたように，現時点では Yacc コーディング時の静的解析によるコーディング支援機能の提供を行うソフトウェアは VSCode 上でのみ動作し，他の主要なエディタでは同等の支援機能が得られない．

そこで本研究では，現在多くの主要なプログラミング言語の実装が存在する Language Server Protocol (LSP)[11] の仕様に則った言語サーバを Yacc 用に開発する (図 1)．この言語サーバを利用することで，ユーザは使い慣れたエディタでコーディング支援機能を使うことができ，Yacc コーディング時の負担が軽減する．また，新しいエディタが現われた場合でも，エディタ開発者は LSP に則ってクライアントを開発するだけで Yacc コーディング時の支援機能をユーザに提供することができる．

なお，現在 Yacc よりも一般に使用されている Bison 3.8.1[12] の仕様を元に静的解析器を実装する．Bison は GNU による Yacc 実装で，軽微な記述の差異や拡張された機能はあるが，基本的なファイル記述は Yacc と同様であり，Yacc との上位互換性のあるソフトウェアのため，便宜上 Yacc と表現している．Bison ファイルの記述例をソースコード 1 に示す．Bison ファイルは基本的にプロローグ，Bison 宣言，文法規則，エピローグの 4 つのセクションからなり，プロローグでは出力ファイルの最初に記述される C コード，Bison 宣言ではトークンの定義と優先順位や結合規則などの設定，文法規則では構文解析器が認識する文法規則，エピローグでは出力ファイルの最後に記述される C コードが書かれる．

3. LSP とは

LSP は 2016 年に Microsoft が発表したプロトコルで，エディタと言語サーバ間の通信方法を規定している (図 2)．現在は多くの主要なプログラミング言語が公式，非公式を問わず言語サーバを持っており [13]，プログラミング言語のコーディング環境を整備する際に複数の拡張やプラグインをインストールする必要がないだけでなく，言語クライアントが存在するエディタならばどのエディタでも利用することができることが強みである．

近年，Go や TypeScript, Rust といったプログラミング言語が注目を集め利用者が増えているが，エディタや IDE などの開発者がそれぞれのプログラミング言語用に補完や定義ジャンプ，ホバー時のヒント表示などの多くの支援機能を追加するには多大な労力を要する．

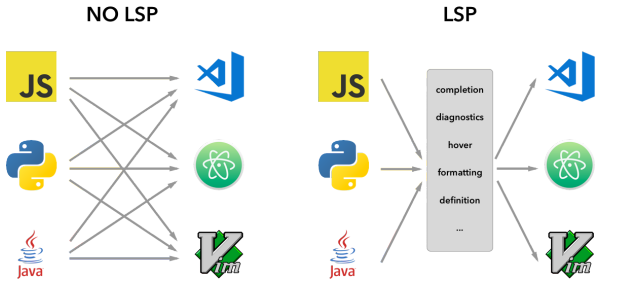

図 2: LSP 登場による言語とエディタの関係の変化 ([14] より引用)

従来は開発ツール毎に支援機能を開発する必要があり，M 個の言語に対して N 個のエディタが存在し，その実装の数は M×N であった．しかし LSP の登場により，言語ツールとエディタの API が統一されたことで，それぞれは API に沿った実装をするだけで同様の機能が実現できるようになり，実装の数は M+N となった．このようなことから，LSP は言語のユーザとエディタ開発者双方の負担を軽減することができると言える．また，LSP を用いることで言語の静的解析器を一度実装すれば良いため，新しい言語を異なるエディタに素早く取り込めたり，特定のエディタの拡張やプラグインを大量に記述する必要がないため，言語の普及を早めたりする効果が期待できる．さらに，特定のエディタでのみ記述ができる言語は潜在的なユーザグループを排除しているという指摘 [15] もあるため，LSP を用いて多様なエディタに対応することで，Yacc ユーザやエディタ開発者の負担軽減だけでなく，言語の普及に貢献することもできると考える．なお，本研究では LSP のバージョン 3.17 に準拠する．

4. Yacc 用言語サーバの開発

4.1. 開発概要

本研究では Yacc 用言語サーバの開発と複数の言語クライアントの設定および開発を行う．また言語サーバについて，LSP に則ってリクエストやレスポンスを処理したり，ドキュメントを読み書きしたりするサーバとしての機能だけでなく，静的解析に基づく支援機能を提供するために静的解析器も実装する．つまり，開発は主に 3 つのプログラムからなり，それは言語サーバと言語クライアント，静的解析器である．

表 1 に開発環境を示す．言語サーバは OCaml で実装し，VSCode の言語クライアントは TypeScript で実装する．主要なエディタではすでに言語クライアントが実装済みであり，開発者はパッケージの利用と設定ファイルへの追記のみで対応できることが多いが，VSCode は言語クライアントのパッケージを用いて軽微な実装を行う必要があった．表 1 中の言語クライアントは VSCode の言語クライアントを指している．なお，今回開発した Yacc 言語サーバのソースコードは https://github.com/is-hoku/yacc-language-server から参照できる．

4.2. 言語クライアントの実装・設定

VSCode で動作する言語クライアントを VSCode 拡張として実装した．そこでは Microsoft が開発している vscode-languageclient[16] という npm パッケージを使用し，VSCode の拡張機能としての設定や言語サーバとの接続方法などの設定を記述した．また，Vim と Emacs でも動作させるために，Yacc ファイルを開いた状態で言語サーバと通信を行うように設定ファイルに記述を追加した．

4.3. 言語サーバの通信

LSP では JSON-RPC を用いて通信をするが，通信方式は指定されていない．しかし LSP においては，同一のコンピュータでクライアントとサーバが動作することが通常であるため，今回は同一コンピュータ上で標準入出力による通信を確立することとした．実際に VSCode の言語クライアントのパッケージでは標準入出力だけでなく，パイプ通信やソケット通信，IPC 通信が選択できる．一方で，Vim や Emacs の言語クライアントのプラグインの多くが標準入出力での通信のみをサポートしているという事実があり，これは標準入出力による通信が一般的に用いられていることの証左であると言える．なお，言語サーバが適用できる状況を増やすために，ソケット通信でも通信を開始できるように実装した．実行時のコマンドライン引数で，標準入出力かソケット通信のどちらかの通信方式を指定するようにし，拡張性を高めた．

LSP の仕様に則って，ドキュメント (ファイル) はクライアントではなくサーバ側のハッシュ表で管理することとした．ドキュメントが開かれると，クライアントはサーバにドキュメントの内容を含むドキュメント情報を送信する．このリクエストを処理する際に，サーバがドキュメント情報をハッシュ表に格納する．これ以降のドキュメントの編集では，差分のみがサーバに送信されるため，差分をハッシュ表に格納されている情報に適用することで，サーバはクライアントが開いているドキュメントと同期するように実装した．

表 1: 開発環境

	言語サーバ	言語クライアント
プログラミング言語	OCaml 4.14.0	TypeScript 5.3.3
ビルドシステム	dune 3.7.2	tsc 5.3.3
ランタイム	OCaml 4.14.0	Node.js v21.6.1
OS	Arch Linux x86_64	
CPU	Intel i7-10710U	

4.4. 言語サーバのライフサイクル

言語サーバのライフサイクルを図 3 に示す．初期化と起動という 2 つの状態を保持し，これらの状態に応じてリクエストを受理してレスポンスを返却するか，状態を遷移させてエラーレスポンスを返却するかなどを判断する．

LSP の通信でやりとりされるメッセージには，request, response, notification の 3 種類がある．request は response を必要とし，notification は response を必要としないものである．クライアントとサーバのどちらからも，どのメッセージでも送受信できる．クライアントは支援機能に関する通信を行う前に，initialize request を送信する．これに対してサーバは response を返し，クライアントは initialized notification を送信する．この一連の通信でクライアントとサーバはそれぞれの capabilities を送り合う．これによって，双方はそれぞれが対応している支援機能などについて知ることができる．このハンドシェイクが完了すると，サーバはクライアントから送信された補完や定義ジャンプなどの支援機能に関わる request を受理するようになる．

今回は textDocument/didSave notification をクライアントから受信した時，サーバで静的解析を行い，その結果として得られた構文エラーなどの情報を textDocument/publishDiagnostics notification を用いてクライアントに送信するように実装した．現在，実装が完了し対応している LSP メソッドとその役割や実装上での動作について表 2 に示す．

4.5. Yacc の静的解析器

Bison 実装の字句解析器と構文解析器の部分を ocamllex, Menhir を用いて OCaml への移植を行った．これによって Bison の仕様に則った字句解析器と構文解析器を得られた．また，構文解析の結果の出力として抽象構文木を得るように実装をすることで，静的解析が可能となった．静的解析の流れを図 4 に示す．図中では今回実装した部分を青色の文字で表している．

完全な入力を想定している通常のコンパイラと異なり，言語サーバは多くの場面で不完全なソースコードが入力として渡される．抽象構文木を走査して型検査や変数の定義と使用の検査などを行うため，どのような入力であっても構文木を得ることができなければ十分なコーディング支援を提供することが困難である．そのため，不完全な入力に対してエラー回復を行い抽象構文木を得る必要がある．エラー回復戦略としては，特殊な error 記号を使用した局所的なエラー回復や，Burke-Fisher エラー修復などが考えられるが，前者はエラー回復後の解析の精度に問題があり，後者は実装が複雑になることや，計算量が大きいこと，副作用のある意味動作においてエラー修復動作が予期しない影響を及ぼすという問題がある．

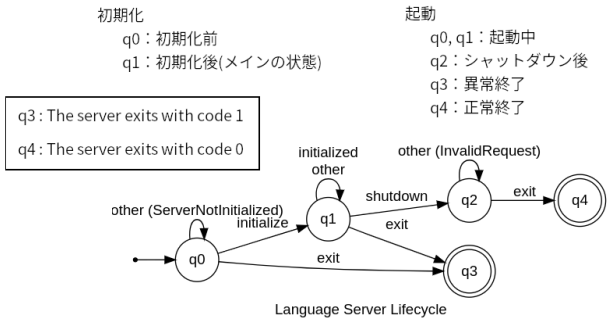

図 3: 言語サーバのライフサイクル

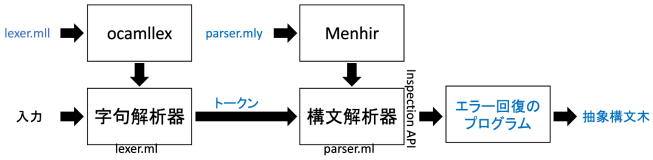

図 4: 静的解析の流れ

まず error トークンを使用した局所的なエラー回復について考える．この方法では，終端記号とみなされる error トークンを含んだ生成規則を定義するだけで実現でき，実装にかかる労力は比較的小さい．しかしこの方法での LR 構文解析器の動作が，error トークンをシフトできる状態に達するまでスタックをポップし，その後 error トークンをシフトしてから，エラーでない動作を持つ先読みトークンまで必要ならば入力トークンを破棄して構文解析を再開するというものであり，エラー回復後の解析の精度に問題がある．例えば，適切に error トークンを含んだ生成規則を定義できなければ，エラー回復が別のエラーを引き起こしたり，スタックのポップによって意味動作が実行不能となったりすることが考えられる．

次に Burke-Fisher エラー修復について考える．この手法はエラーに遭遇した時点の K トークン前から，そ

表 2: 実装済みの LSP メソッド

種類	メソッド名	役割と実装上での動作
request	initialize	初期化．クライアントの capabilities をサーバで保持し，サーバの capabilities を生成して返す．
	shutdown	サーバの状態を遷移させ，これ以降 exit notification 以外の要求を拒否するようにする．
notification	initialized	クライアントが初期化が完了したことの通知．
	exit	サーバプロセスを終了させる．
	textDocument/didOpen	ドキュメントが開かれたことの通知．ドキュメント情報をハッシュ表で保持する．
	textDocument/didChange	ドキュメントが変更されたことの通知．ドキュメントの変更をハッシュ表に反映させる．
	textDocument/didSave	ドキュメントの変更が保存されたことの通知．これをトリガーとして静的解析を行う．
	textDocument/publishDiagnostics	静的解析によるコード診断結果の通知．サーバからクライアントに送信する．

```
symbol_declaration → percent_symbol token_decls

percent_symbol → "%nterm"
percent_symbol → "%token"
percent_symbol → "%type"

token_decls → token_decl_1
token_decls → TAG token_decl_1
token_decls → token_decls TAG token_decl_1

token_decl_1 → token_decl
token_decl_1 → token_decl_1 token_decl

token_decl → id int_opt alias

int_opt →
int_opt → INT_LITERAL

alias →
alias → string_as_id
alias → TSTRING
```

図 5: 文法 (Bison 文法の一部抜粋)

れ以降に現れる全ての時点において，可能な 1 トークンの挿入もしくは除去，置換を全て試して修復をするものであり，error トークンを用いた手法のように生成規則を変更したり，構文解析表が修正されたりすることが無い．しかし，これを実現するために K トークン前のスタックの状態を保持する旧スタックが必要となり，N 種類のトークンを持つ言語で存在する可能な除去，挿入，置換の数は K+K · N+K · N である．このようなことから，Burke-Fisher エラー修復は実装の労力と計算量が大きい点で不利であり，エラー修復中のシフトや還元動作で意味動作が何度も実行されることから，副作用を持つ意味動作を記述した場合にも問題がある．

Merlin の報告書 [18] によると，エラー回復の厳密さは解析対象の言語機能に依存するとされ，スコープ構造と前方参照を許す言語であるかどうかを考慮すべきだと述べられている．Yacc は高級なスコープ構造は持たないが，生成規則の記述中で前方参照を許しているため，厳密さを考えずにスタックをポップしたり入力トークンを破棄したりすると，意味解析段階で不都合が生じる可能性がある．そのため，前述したエラー回復手法では問題があり，より素朴なエラー回復，つまり正常に解析を続行できる状態になるまでダミートークンをシフトすることが求められた．そこで本研究では Menhir の InspectionAPI[17] を利用することとした．Inspection API は LR 構文解析器のオートマトンやスタックの状態にアクセスすることができるため，比較的汎用なエラー回復が実現できる．これまで検討したエラー回復手法の比較を表 3 に示す．エラー回復がカバーできる状況数や安定性，トークン操作の回数などは解析中のスタックの状況や文法，入力トークン列に大きく依存するため，表中の評価は比較した 3 つの手法の中での相対的かつ経験的な値であることに注意されたい．

	スタック	入力	動作	処理
1		%token NUMBER %	シフト	構文解析器
2	percent_symbol	NUMBER %	シフト	構文解析器
3	percent_symbol id	%	エラー	エラー回復機構
4	percent_symbol token_decl		還元	エラー回復機構
5	percent_symbol token_decl_1		還元	構文解析器
6	percent_symbol token_decls		還元	構文解析器
7	symbol_declaration		還元	構文解析器

図 6: エラー回復の動作例

エラー回復戦略を説明する．エラーとなるトークンに遭遇したら，その時点のスタックに還元できる生成規則があれば還元を行い，そうでなければ FIRST 集合を計算してシフトできるトークンを取得し，ダミーの値を入れてシフトする．もしシフトできるトークンが一つも無ければ，スタックからトークンをポップし還元できる生成規則を探すところから再開する．

ここで文法を図 5 として，この戦略でのエラー回復の動作例を図 6 に示す．入力として`%token NUMBER %`

表 3: エラー回復手法の比較

エラー回復手法	実装コスト	カバーできる状況数	回復後の解析の安定性	回復に必要な除去，挿入，置換の回数
error トークン	小さい	実装に依存	小さい	最良で 1(状況に依存)
Burke-Fisher エラー修復	大きい	多い	小さい	最悪で K+K · N+K · N
Inspection API	中程度	中程度	大きい	最良で 1(状況に依存)

が与えられた時，`%token NUMBER` までは通常の構文解析過程でスタックにシフトされるが，次の入力である%は文法のどこにも出現しない文字であり入力が不正だと分かる．ここで構文解析器は解析を実行することができずエラー状態を報告し，これをエラー回復機構が受け取りエラー回復を試みる．まず還元できる生成規則を探すが，スタックには `percent_symbol id` が積まれているため，`id` を `token_decl` に還元できることが分かる．一度還元を実行すると，通常の構文解析過程に戻る．そこでは，もう不正な入力は無いため，生成規則に従って還元を順番に実行していき，最終的に `symbol_declaration` を得ることができる．チャプター 2 で示したように Yacc はセクション毎に記述が分かれており，セクションによって使用する文字や文字列が異なる．このエラー回復戦略はヒューリスティックな解決法ではあるが，Yacc 特有のセクション構造のために字句解析段階で一部の構文解析も行っているという事情により，良い結果が得られたと推察する．この戦略では不正な入力文字を破棄してしまう点や，エラーの発見が先延ばしになるような状況で最善の回復ができない点などで問題はあるものの，ユーザの入力途中の記述が渡されることが多いということから，他の方法に比べて有用であると考える．

5. 実行例

コード診断を VSCode, Vim, Emacs で行っている様子をそれぞれ図 7, 図 8, 図 9 に示す．ここでは%という入力が不正であることと，予期しない位置に EOF があることを構文解析の過程で検出し，LSP の textDocument/didSave メソッドによってエディタに報告している．

このコード診断の結果を得るまでにバックグラウンドで実行されている処理について説明する．まず，クライアントがサーバを起動し，標準入出力での通信を開始する．その後，initialization(初期化) のための通信が行われる．ここでクライアントとサーバそれぞれが対応している機能を通知し合う．次にエディタで Yacc ファイルが開かれると，textDocument/didOpen がクライアントから送信され，これをうけてサーバではドキュメント情報をハッシュ表で保持する．これ以降，エディタでドキュメントが編集される度に前回との入力の差分を持った textDocument/didChange が送信され，これを受信したサーバは差分を保持しているドキュメント内容に反映させることによって，クライアントとの同期を取る (毎回ドキュメントの内容を全て送信するか，差分のみ送信するかは server capabilities で制御できるが，今回はこのような実装とした)．エディタでドキュメントの変更内容が保存されると，textDocument/didSave が送信され，これをうけてサーバでは静的解析が実行される (静的解析は計算にかかるコストが比較的高いため，どのタイミングで実行するかは議論の余地がある．また，解析結果のキャッシュとして保持したり，変更箇所のみ解析を実行したりなどの改善が考えられる)．そこで得た構文エラーなどの情報を textDocument/publishDiagnostics でクライアントに通知することで，エディタの画面上にエラー内容が表示される．

6. 考察

VSCode と Vim, Emacs の言語クライアントを使用し，Yacc ファイルが開かれた時に今回作成した言語サーバを実行するように設定したことで，異なるエディタで同様の支援機能が容易に得られることが確認できた．また，Bison の仕様や実装を元に字句解析器と構文解析器を作成し，構文解析器の出力として抽象構文木を構築することで，静的解析を可能にした．今回は不完全な入力に対して可能な限りエラー回復を試みて抽象構文木を構築し，その過程で見つかった構文エラーなどを LSP の textDocument/publishDiagnostics メソッドによってクライアントに送信することで，開発した言語サーバが機能することを示した．

セクション 4.2 で述べたように，各言語クライアントはすでに開発されており，それを用いて設定ファイルに記述を追加するだけであった．VSCode の言語クライアントについては，パッケージを利用しながら TypeScript でコードを書く必要があったが，この作業のコストは非常に小さい．このことから，今後新しい

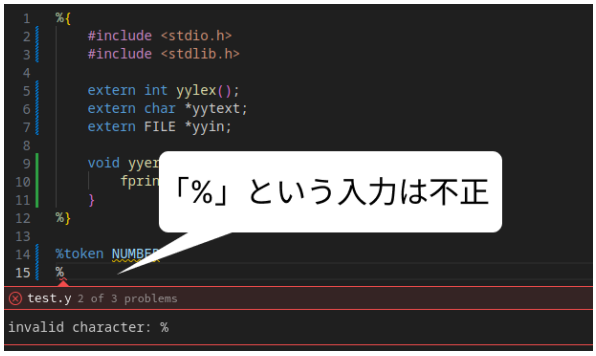

図 7: VSCode でのコード診断実行例

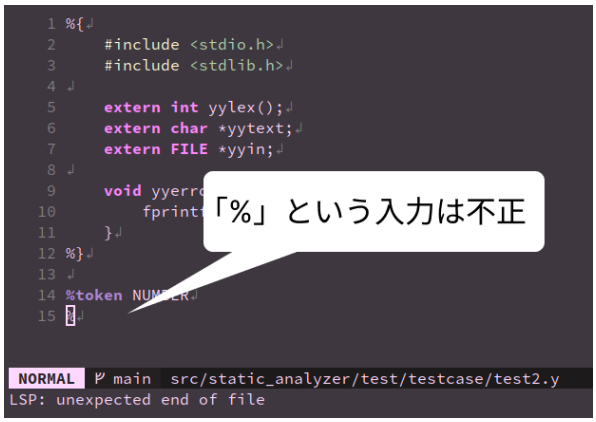

図 8: Vim でのコード診断実行例

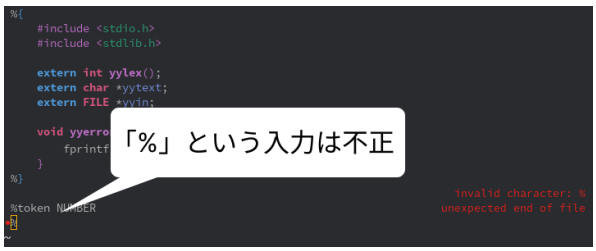

図 9: Emacs でのコード診断実行例

エディタが台頭したとしても，言語クライアントさえ実装されていれば，コストをほとんどかけることなく Yacc 記述時にコーディング支援機能が得られることが確認できた．

また，言語サーバのアーキテクチャにレイヤードアーキテクチャを採用したこと，各レイヤーとメソッド毎に単体テストを記述したことで，拡張性と保守性が高いソフトウェアを開発することができた．統合テストについては，並列プログラミングライブラリを用いてモッククライアントを動作させ，サーバの起動から initialization によるハンドシェイク，シャットダウンまでの動作を保証するものとした．

7. 今後の課題

今回記述した Menhir ファイルは Bison の記述の移植のため，コンパイラでの動作を前提としており，静的解析を行うことは想定されていない．そのため，文法に対応する意味動作に副作用がある場合が多いが，エラー回復のパターンを増やしたり，より安全にエラー回復を行ったりするためには，意味動作を副作用の無いものに変更する必要がある．また，エラー回復機構の改善について，入力文字を破棄せずにいくつかのトークンの挿入や除去を繰り返した後に再度シフトしたり，入力途中のトークンを推測して最適なトークンをシフトしたりなどの方法が考えられる．ユーザの入力のパターンとその中に現われる誤りのパターンは無数であることを考えると，すでに解析されたトークンとそこで生成された構文木の構造を壊すことなくエラー回復を行うためにヒューリスティックな解法に頼ることになり，全ての状況で最善の回復ができるアルゴリズムは存在しないと推測する．そのため，カバーできる状況数と処理の複雑さや構造を壊さないための安全性のトレードオフを考えながら，前述したような方法を追加してテストを実行する試みが要求される．さらに，意味解析器を実装し，コード補完や定義ジャンプなどサポートする機能を増やして利便性を高めていく．その後，支援機能に関する計算を並列化するなどして効率化を図りたい．

セクション 4.5 で示したように，Bison の記述では生成規則の定義中に前方参照を許している．そのため，抽象構文木を走査する際にまず生成規則の左辺を記号表に追加して，後から右辺を検査するなどの工夫が必要となり，その実装のために抽象構文木の修正が必要になる可能性がある．また，並列化について，クライアントが単一のエディタであり並列処理が行われないため，通信部は並列処理する必要はなく，サーバでの計算を並列に行うことが考えられる．計算の並列化を実現する際は，LSP の仕様に注意しなければならない．例えば，textDocument/didChange notification では変更 c_1,c_2 がある時，状態 S は c_1 によって S' に遷移し，c_2 によって S'' に遷移する．ここで c_2 は S' に対して適用されなければならない．他にもドキュメントに変更を加えるリクエストや，shutdown などのサーバの状態を遷移させるリクエストは，並列に処理されるタスクの中でも処理の順番を慎重に考慮しなければいけない．

このように問題を解決したり機能を追加したりすることで，言語サーバを十分実用に耐え得るものにしていき，ユーザの獲得や継続的な開発を実現したい．

謝　辞

本研究の遂行にあたり，指導教員として多大なご指導を賜りました稲垣宏教授 (豊田工業高等専門学校情報工学科)，ご多忙のなか相談に応じてくださりご助言を頂いた内山慎太郎氏 (豊橋技術科学大学大学院工学研究科情報・知能工学専攻博士後期課程応用数理ネットワーク研究室) に感謝申し上げます．

文　献

[1] 高専 Web シラバス，“コンパイラ”，https://syllabus.kosen-k.go.jp/Pages/PublicSyllabus?school_id=23&department_id=25&subject_code=95018&year=2017&lang=ja, 2023 年 10 月 18 日閲覧.

[2] Nobuya WATANABE，“コンパイラ（Compilers）”，http://www.arc.cs.okayama-u.ac.jp/~nobuya/lecture/compiler/, 2023 年 10 月 18 日閲覧.

[3] 電気通信大学 シラバス Web 公開システム，“シラバス参照”, http://kyoumu.office.uec.ac.jp/syllabus/2023/31/31_21124118.html, 2023 年 10 月 18 日閲覧.

[4] Andrew W. Appel，Modern compiler implementation in ML，Cambridge University Press，New York, Cambridge，2008.（アンドリュー・W・エイペル 神林靖・滝本宗宏 (訳), 最新コンパイラ構成技法, 翔泳社, 東京, 2020.)

[5] Francisco Ortin, Jose Quiroga, Oscar Rodriguez-Prieto, Miguel Garcia，“An empirical evaluation of Lex/Yacc and ANTLR parser generation tools”, 2022.

[6] Leon Aaron Kaplan，“yaccviso − a tool for visualizing yacc grammars”, 2006.

[7] 楠目 勝利, 佐々 政孝.“コンパイラにおける構文解析過程の視覚化”, 全国大会講演論文集, 第 55 回, ソフトウェア科学・工学, pp448-449, 1997.

[8] Mona E. Lovato and Michael F. Kleyn. 1995. Parser visualizations for developing grammars with yacc. SIGCSE Bull. 27, 1 (March 1995), 345‑ 349. https://doi.org/10.1145/199691.199855.

[9] Visual Studio Marketplace，“VSCode-YACC”，https://marketplace.visualstudio.com/items?itemName=carlubian.yacc, 2023 年 10 月 19 日閲覧.

[10] Visual Studio Marketplace，“Yash”，https://marketplace.visualstudio.com/items?itemName=daohong-emilio.yash, 2024 年 2 月 8 日閲覧.

[11] Official page for Language Server Protocol，“Language Server Protocol Specification - 3.17”，https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/, 2023 年 10 月 19 日閲覧.

[12] Bison 3.8.1，https://www.gnu.org/software/bison/manual/bison.html, 2024 年 1 月 15 日閲覧.

[13] Official page for Language Server Protocol，“Implemantations”，https://microsoft.github.io/language-server-protocol/implementors/servers/，2023 年 10 月 19 日閲覧.

[14] Language Server Extension Guide | Visual Studio Code Extension API, https://code.visualstudio.com/api/language-extensions/language-server-extension-guide, 2024 年 2 月 8 日閲覧.

[15] Bünder, H.“Decoupling Language and Editor - The Impact of the Language Server Protocol on Textual Domain-Specific Languages.”International Conference on Model-Driven Engineering and Software Development, no.10.5220/0007556301290140, pp.129-140, Prague, Czech Republic, 2019.

[16] npm,“vscode-languageclient”, https://www.npmjs.com/package/vscode-languageclient, 2023 年 10 月 19 日閲覧.

[17] Menhir Reference Manual (version 20231231)，“Inspection API”, https://gallium.inria.fr/~fpottier/menhir/manual.html#sec64, 2024 年 1 月 28 日閲覧.

[18] Frédéric Bour, Thomas Refis, and Gabriel Scherer. 2018. Merlin: a language server for OCaml (experience report). Proc. ACM Program. Lang. 2, ICFP, Article 103 (September 2018)，15 pages，https://doi.org/10.1145/3236798.