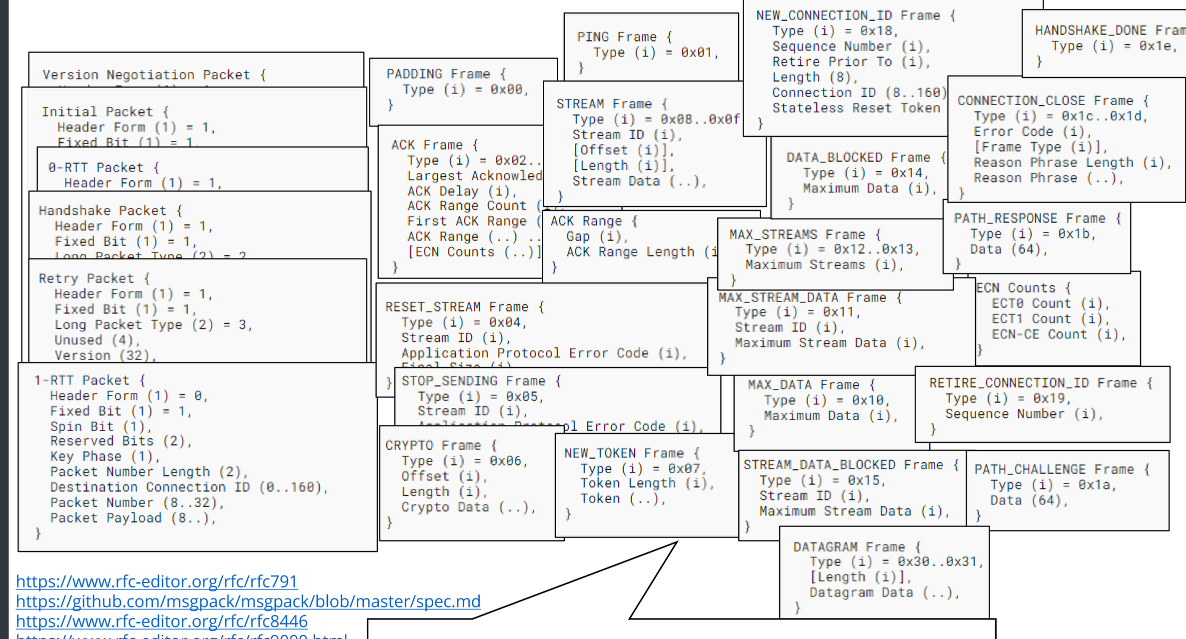
SecHack365

バイナリエンコーダー/デコーダージェネレーター

開発駆動コース 仲山ゼミ 藤岡航介

brgen



量が多くてつらい!

いろんな形式があって混乱する!

手書きの解析/生成コード分かりづらい

仕様だけでわからないから OSSのソースコードとか見ても結構エンコード/デコード部分が分散していたりしてわかりにくい...

DRY原則にも違反だし...

DRY原則 Don't Repeat Yourselfの略。同じ定義のことを繰り返し書くなというプログラミングをする上での格言/設計思想

手書きで解析/生成処理書くのめんどくさい

構造体定義/エンコード/デコード etc...で同じ定義についてを何回も何回も...

自作TCP/IPプロトコルとかはよくあるが、プロトコルで使われるフォーマットが増えていくとどんどん敷居が高くなっていく...

コードジェネレーター

少量のソースコードから多量の定型的なソースコードを生成するツール

昔の人(プログラマー)

DSL(ドメイン固有言語)

ある特定の分野に特化したプログラミング言語

例: Web - HTML、CSS
gRPC – Protocol Buffers
RDB - SQL
GitHub Actions Workflow File

任意のバイナリフォーマットのエンコード/デコードルールを表せるDSL

からコードを生成する

コードジェネレーター

を作ればいいんだ!

SecHack365での紆余曲折

Before SecHack365: QUICクライアントを実装する
〜6月: QUICサーバーを作ろうとするがプレッシャーを感じて病みだす
7月〜8月: バイナリエンコーダー/デコーダージェネレーターにテーマを変更。プロトタイプを作る。競合事例を調べ始めてまた病みだす。
8月〜: いろいろなバイナリフォーマットのパーサを書いてみたりする。サンプルを書き始める。
9月〜: 現在の定義言語パーサーを書き始める。ジェネレータードライバーを作り始める。LSPサーバーを作り始める。json2cppを作り始める。json2goを作り始める。
10月〜: ASTコードジェネレーターを作り始める。WebPlaygroundを作り始める。
11月〜: json2cpp2を作り始める
12月: QUICについての同人誌を書く/C103で頒布する。
1月〜: brgenのドキュメントを書き始める。

定義言語のサンプル

バイナリフォーマット定義言語の書きやすさと十分な表現力を持っていることを示すため、既存のバイナリフォーマットを表現するサンプルたちを作成しました。ネットワーク・プロトコルのフォーマットやファイルフォーマットなどのサンプルがあります。これらのサンプルは定義言語の仕様を決めたり、定義ファイルパーサーのバグを検出したりするのに大いに役立ちました。また、WebPlaygroundではこれらのサンプルを読み込むことができます。

↑Standard MIDI File(一部)　←サンプル一覧　↑TLS(一部)　→ELF(一部)

バイナリフォーマット定義言語

書きやすい - **簡単に書けて読みやすい**

十分な表現力 - **プロトコル/ファイルフォーマットを十分表現できる**

仕様書などのバイナリフォーマット表現はそれぞれのプロトコルなどの表現に特化しているという面もありいろいろ種類がありますが、その表記はバラバラです。またフィールド間の関係についてもフォーマット図ではなく文章で仕様が書かれていたりと親切とは言い難いものがあります。さらに図のような形だとプログラムで読み込むのが難しかったり気軽に変更したりできません。そこで私は以上のような目標を設定してDSLを開発しました。

```
enum Operation:
    :u16
    Request = 1
    Reply = 2

format ArpPacket:
    hardware_type :u16
    protocol_type :u16
    hardware_len :u8
    protocol_len :u8
    operation :Operation
    operation == Operation.Request || operation == Operation.Reply
    source_hardware_address :[hardware_len]u8
    source_protocol_address :[protocol_len]u8
    target_hardware_address :[hardware_len]u8
    target_protocol_address :[protocol_len]u8
```

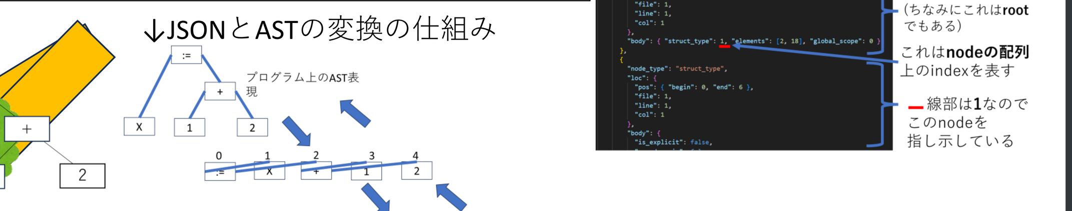
Python風インデント構文
プログラミング言語に近い形式
長さ指定により、どのフィールドがどのフィールドの長さだというのが直感的にわかる。
制約式も付けられる

定義言語パーサー/型解析

brgenの開発で**最も開発に時間をかけた部分にして最も地味な部分**です。
C++で記述されています。定義言語パーサーは定義言語の構文解析と意味解析(型解析など)を行いその結果得られたASTをJSON形式で出力します。
現状、定義言語パーサーの実装=フォーマット定義言語の仕様となっており、またこれによって解析されてASTに落とし込まれないことにはエンコーダー/デコーダージェネレーターの方にも反映できないため、かなりの時間をこれに費やしました。
私はこれのことをずっとパーサーと呼んでいましたが最近になってパーサーと言うには高級すぎると思い始めたためなんと呼べばいいのか迷い中です。

AST(Abstract Syntax Tree/抽象構文木)
プログラミング言語などのソースコードを木構造の形式で取り出したものです。本質的ではないような情報は省略されているということです。(コメントやかっこも省略したりするがbrgenのASTでは省略されていない)

↓出力されるJSONの中身　↓JSONとASTの変換の仕組み



LSPサーバー/VSCode拡張

brgenの定義言語を書くのを支援するLSPサーバーを書きました。
これは当初はバイナリエンコーダー/デコーダージェネレーターの開発に行き詰まり、その気分転換として開発を始めました。シンタックスハイライトやホバーの表示などによってより定義言語の意味がわかりやすくなりました。LSPサーバーが定義言語パーサーを頻繁に呼び出し、また中途半端な入力がたくさん入力されるため定義言語パーサーのデバッグが進むという思わぬ効果もありました。また、このLSPサーバー用につくったAST解析ツールはWebPlaygroundでも生かされています。

LSP(Language Server Protocol)
Microsoftが提唱した言語サーバー(シンタックスハイライトや定義検索などの機能を提供)とエディタ間の通信規約。これに従うことで同じ言語サーバーを異なるエディタで使用することができる。

サイズ(可変長時はdynamic表記)
ビットアラインメント
依存構造体/enum

WebPlayground

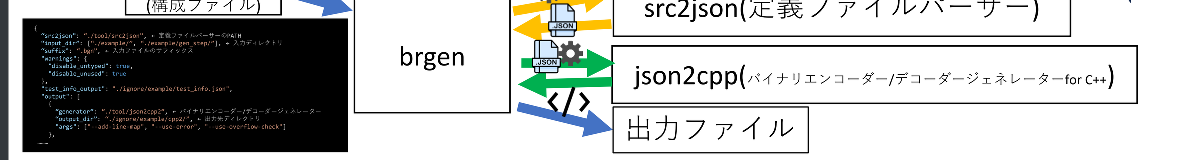
Webブラウザ上でbrgenを試せるWebPlaygroundを開発しました。
定義言語パーサーとバイナリエンコーダー/デコーダージェネレーターをWebAssemblyに変換して動かすことで、GETする以外すべてローカルで完結して動きます。またGitHub Pagesを使ってホスティングしており、サーバー管理の必要などはなく(下手にサーバー建てるよりは)安全です。エディタにはMonaco Editorを使用しています。
また、LSPサーバーと共通のロジックを使っており、LSPサーバーで表示されるのと同じようなシンタックスハイライトやエラー表示、ホバーなどが使えます。
GitHub Actionsを使ってmainブランチの最新版をデプロイしているので新機能が実装されたらすぐ使えます。
https://on-keyday.github.io/brgen

サンプルをロードする↑

DSL
バイナリフォーマット
コードジェネレーター

ジェネレータードライバー

CLIの定義言語パーサーとバイナリエンコーダー/デコーダージェネレーターの橋渡しをするツールです。Go言語で書かれています。構成ファイルを読み込んで入力に従って定義ファイルパーサーを呼び出しその出力結果をバイナリエンコーダー/デコーダージェネレーターに渡しさらにその出力結果をファイルに保存します。またgoroutineを使い並列/並行にファイルを処理します。
ちなみにこのジェネレータドライバーの名前はbrgenとなっています。
例えるならgcc(コンパイラドライバ)とcc1(コンパイラ),ld(リンカー)の関係です。



バイナリエンコーダー/デコーダージェネレーター

brgenの２つ目のコアとなるバイナリエンコーダーデコーダージェネレーター。現時点ではC++/Go言語向けの2種類を提供しています。C++向けはC++で、Go言語向けはGo言語で書かれています。実のところ一番本質(タイトルにもなっていますね)にも関わらずかけた時間が一番少ないところだったりします。ジェネレーターは当初brgenのASTからさらに中間表現に落とし込んで生成しようとしたのですが、複数言語対応しようとしたときにいくつもの言語で中間表現を考えるのが大変だったので現在の成果物ではASTから直接落とし込んでいくような形になっています。
(~~このポスターを書いているときにその言語非依存中間表現自体も開発すればよかったなと思ったのは秘密です~~)
基本的なバイナリフォーマット(基本整数型のみ)などであればどれもきちんと生成できます。条件分岐やビルトイン関数などの対応はまちまちだったりします。バイトアラインされていないフォーマットやbit prefixで分岐するみたいなのはまだ対応できていません。正直に言って私はまだ出来に満足してはいません。まだまだ改善の余地が残っています。

生成コード→
C++(json2cpp2)
現時点で一番機能を実装しているジェネレーター。しかし、依存ライブラリが今のところ私しか使っていない独自ライブラリのため、PoC的な意味合いが強い

Go(json2go)
Go言語向けジェネレーター。こちらは標準ライブラリのみ使用なので生成されたコードは比較的使いやすいです。ただjson2cppに比べると機能が少ないです

json2c – C言語向け　　絶賛制作中!
json2rust – Rust向け
json2kaitai – Kaitai Struct(https://kaitai.io/)向け

その他雑ツール

ctobgn
C言語の構造体や列挙体の定義をbrgenの定義ファイル形式に変換するツール。
(完璧ではないです...)
json2cpp – json2cpp2のプロトタイプ版

開発リポジトリ: https://github.com/on-keyday/brgen
ドキュメント: https://on-keyday.github.io/brgen/doc

ASTコードジェネレーター

LSPサーバーやGo言語用コードジェネレーター制作のためにASTを各言語(TypeScriptやGo言語)に人力で移植しようとしましたがノードの種類が大量で困難であり、またメンテナンス性も考えると人力移植はバグの原因になります。
同じ定義のものを移植するって面倒...**はっ、これもコードジェネレーターで作ればいいんだ!**と思い至りASTノードの定義から各言語のソースコードを生成するツールも作りました。
これのおかげでC++のノード定義をいじるだけで他言語には自動で反映することができるようになり、定義言語の仕様をASTの移植のことを気にせずかなり柔軟に変更することができるようになりました。

```
struct Ident : Expr {
    define_node_type(NodeType::ident);
    std::string ident;
    scope_ptr scope;
    std::weak_ptr<Node> base;
    IdentUsage usage = IdentUsage::unknown;

    Ident(lexer::Loc l, std::string&& i)
        : Expr(l, NodeType::ident), ident(std::move(i)) {}

    Ident(lexer::Loc l, const std::string& i)
        : Expr(l, NodeType::ident), ident(i) {}

    // for decode
    Ident()
        : Expr({}, NodeType::ident) {}

    void dump(auto&& field_) {
        Expr::dump(field_);
        sdebugf(ident);
        sdebugf(usage);
        sdebugf(base);
        sdebugf(scope);
    }
};
```

```
},
{
  "node_type": "ident",
  "base_node_type": [
    "expr",
    "node"
  ],
  "loc": "loc",
  "body": {
    "expr_type": "shared_ptr<type>",
    "constant_level": "constant_level",
    "ident": "string",
    "usage": "ident_usage",
    "base": "weak_ptr<node>",
    "scope": "shared_ptr<scope>"
  }
},
{
```

```
export interface Ident extends Expr {
    ident: string;
    usage: IdentUsage;
    base: Node|null;
    scope: Scope|null;
}

export function isIdent(obj: any): obj is Ident {
    return obj && typeof obj === 'object' &&
        typeof obj?.node_type === 'string' &&
        obj.node_type === "ident"
}
```

```
func (n *Ident) GetNodeType() NodeType {
    return NodeTypeIdent
}

type Ident struct {
    Loc           Loc
    ExprType      Type
    ConstantLevel ConstantLevel
    Ident         string
    Usage         IdentUsage
    Base          Node
    Scope         *Scope
}
```

将来的にはC++もJSONから生成にしたい....

言語によって継承があったりなかったり、ポインタ値の扱いが違ったりしたのでそれぞれどう実装するかに苦心しました。

コードジェネレーターは作りかけ...

対象ユーザー

自作プロトコルしたい人

例えば自作WebRTCとか自作QUICとか自作TLSとかしてみたいけどフォーマット多すぎてやる気にならんと言う方、そういうつまんないところはぜひbrgenを使って省略して、プロトコルの上位レイヤー(アルゴリズム等)の実装に集中しましょう!
え?ツールだよりじゃいやだ?そんなあなたはぜひジェネレーター自体も自作してみましょう!
C++,Go,Rust,Python,TypeScriptでASTを扱うためのライブラリが提供されています! (~~そしてずるずると言語処理系に沼りましょう~~)

使い方

CLI

https://github.com/on-keyday/brgen/releases/tag/v0.0.5 からビルド済みバイナリをダウンロードできます。詳しくは https://on-keyday.github.io/brgen/doc/docs/setup/

LSPサーバー

上記と同じ箇所からダウンロードできます

WebPlayground

https://on-keyday.github.io/brgen で公開しています。上のWebPlaygroundの説明の箇所にもQRコードがあります

セキュリティ

バッファオーバーフロー対策

たくさんコードを書いていると範囲チェックのコードを書いたりするのがめんどくさかったり忘れがちだったりしますがそれだとバッファオーバーフローなどの脆弱性に繋がってしまいます。brgenのコードジェネレーターは自動で範囲チェックを挿入するので安全です

算術オーバーフロー対策

C++のみですが算術オーバーフローチェックをする関数に置き換えるオプションを提供しています。
他にも自動生成なので人力で書いたときの整合性が合わないみたいなバグがなくなります。

課題

ジェネレーターの限界

現段階のジェネレーターは定義言語で表現できる範囲に比べてその能力はかなり制限されています。また、言語ごとに出来ることにも差があります。また対応言語もまだまだ少ないです。他にも生成結果もまだまだ最適とは言い難い面があります。

定義言語の限界

定義言語自体もまだまだ定義を表現するのに最適とは言い難いと思っています。より多くのサンプルを集め改善していきます。

ドキュメント

ドキュメントは書きつつありますがまだまだ十分とは言えません。内容がわかりづらい部分もあるので改善していきたいです。

展望

対応言語の拡充

現在対応している言語は2つだけですが、より多くの言語にも対応していきたいです。現段階ではTypeScriptなどへの対応構想があります。また、インタープリターのようにして定義言語と実バイナリデータの整合性チェックを行うツールも作り途中です。

使えるサンプルの拡充

現在あるサンプルの中にはジェネレーターの限界もあってジェネレータで生成できないものが多いです。それらを対応していき生成できるものを増やしたいです。またサンプル自体も増やしていきたいです。

AST操作ライブラリの整備

現在定義言語パーサーに力を入れすぎた結果複雑になって若干使いづらい部分もあります。もう少し使いやすくなるよう、また使い方がわかりやすくなるようASTを改良したり、周辺ツールをつくったり、ドキュメントを整備したりしていきたいです。

NICT National Institute of Information and Communications Technology

Jsonアイコン出典: https://icons8.com/icons/set/json
ソースコードアイコン出典: https://icons8.jp/icon/11260/%E3%82%BD%E3%83%BC%E3%82%B9%E3%82%B3%E3%83%BC%E3%83%89