WebAssembly GCをターゲットとしたCommon Lispコンパイラの実装戦略：データ構造、メモリ管理、および動的評価モデルの包括的解析

1. 序論：WebAssembly GCによる言語実装のパラダイムシフト

WebAssembly（Wasm）の登場以来、多くの言語処理系がブラウザ上での動作を目指して移植されてきました。Common Lispもその例外ではありませんが、従来のWebAssembly（Wasm MVP）環境における実装は、C言語やC++で書かれたランタイムをEmscripten等でコンパイルし、Wasmの「線形メモリ（Linear Memory）」を巨大なバイト配列として管理する手法が主流でした。この「線形メモリモデル」は、既存のC実装（ECLやSBCLのサブセットなど）を比較的容易に移植できる反面、ブラウザ環境との親和性において重大な欠陥を抱えていました [1]。

最大の問題は、Wasmモジュール内に閉じたガベージコレクション（GC）と、ブラウザ（ホスト環境）のGCとの間に断絶があることです。LispオブジェクトがDOMノードを参照し、そのDOMノードがJavaScriptクロージャを通じて再びLisp関数を参照するような「言語間の循環参照」が発生した場合、線形メモリ内のGCは外部の参照関係を追跡できず、メモリリークを引き起こします [3]。また、独自のGCをWasmバイナリに同梱する必要があるため、配信サイズが増大し、起動時間が遅延するという実用上の課題もありました [4]。

これに対し、WebAssembly Garbage Collection（Wasm GC）プロポーザルは、Wasmの型システムを拡張し、ホストVM（V8, SpiderMonkey, JavaScriptCoreなど）が管理するヒープ領域に直接オブジェクトを割り当てることを可能にします。これにより、Lispコンパイラは独自のGCを実装する必要がなくなり、ホストのGCにメモリ管理を委譲できるだけでなく、言語間の循環参照も自動的に解決されるようになります [3]。

本報告書では、Wasm GCをターゲットとしたCommon Lispコンパイラの実装戦略について、データ構造のマッピング、制御フローの管理、そしてCommon Lispの根幹である動的性（Eval/JIT）の実現方法に焦点を当て、網羅的に分析・提案を行います。特に、Guile Hoot [5] やSchism [7] といった先行するScheme実装の研究成果を参照しつつ、Common Lisp特有の仕様（多値返却、スペシャル変数、CLOSなど）をどのようにWasm GCの制約下で表現するかについて、詳細な技術的検討を加えます。

2. 型システムの設計とデータ構造のマッピング

Wasm GC環境におけるCommon Lisp実装の核となるのは、動的なLispの型システムを、静的なWasmの型システム（struct, array, i31ref 等）にいかに効率的にマッピングするかという点です。

2.1 ユニバーサル・ルート型とタグ付け戦略

Common Lispは動的型付け言語であり、任意の変数は任意の型の値を保持し得ます。Wasm GCの型階層において、すべての参照型の頂点に位置するのは anyref（より厳密には (ref null any)）です。したがって、すべてのLispオブジェクトは、Wasmレベルでは anyref として扱われ、実行時にダウンキャスト（ref.cast）や型検査（ref.test）を行うことで具体的な操作が可能になります [8]。
ネイティブコードへのコンパイル（SBCL等）では、ポインタの下位ビットを利用した「タグ付け（Tagging）」によって即値（Fixnum）とポインタを区別しますが、Wasm GCではポインタのビット操作は禁止されています。その代わり、Wasmエンジン自体が提供する i31ref 型を利用します。これは31ビットの符号付き整数を、ヒープ割り当てなしに anyref 互換の参照として扱うための型であり、LispのFixnumを表現するのに最適です [9]。

Common Lisp型	Wasm GC型	表現戦略と備考
Fixnum	(ref i31)	31ビット整数。割り当てコストゼロ。ref.i31で生成、i31.get_sで値取得 [11]。
Cons	struct	car と cdr フィールドを持つ構造体。
Symbol	struct	名前、値、関数、PLIST、パッケージへの参照を保持。
Function	struct	func_ref と環境（クロージャ）を保持する構造体 [12]。
Instance	struct	クラスメタオブジェクトへの参照と、スロット値を格納する配列。
Character	i31ref	Unicodeコードポイントを i31ref にエンコード（タグビット等でFixnumと区別）。

2.2 コンスセル（Cons Cell）とリスト構造

Lispの基本構成要素であるコンスセルは、2つのフィールドを持つミュータブルな構造体として定義されます。Common Lispの仕様上、rplaca や rplacd による書き換えが必要なため、フィールドは mut（可変）である必要があります [8]。

コード スニペット

```
(type $cons (struct
  (field $car (mut anyref))
  (field $cdr (mut anyref))
))
```

ここで重要な設計判断となるのが NIL の表現です。Wasm参照型には null 値が存在しますが、

Common Lispの NIL はシンボルであり、かつリストの終端でもあり、さらに偽（False）としても扱われる特殊なオブジェクトです。
実装戦略としては以下の2つが考えられます。

1. **NILをWasmのnull**として扱う**:** メモリ効率は高いですが、car/cdr 操作のたびにnullチェックが必要です。また、NIL がシンボルとしての機能（名前やプロパティリスト）を持つため、null に対してそれらのアクセスが発生した場合の特例処理が複雑になります。
2. **NIL**を特異なシングルトン構造体とする**:** null を使わず、常に実体のあるオブジェクトとして NIL を表現します。型チェックは ref.eq によるポインタ等価性判定で行います。Common Lispの仕様への準拠性を高めるためには、このアプローチが推奨されます [9]。

2.3 シンボルとインターン（Interning）

シンボルは単なる識別子ではなく、リッチなデータ構造です。Common Lispのシンボルは、印字名（Print Name）、値（Value）、関数定義（Function）、プロパティリスト（Property List）、そして所属パッケージ（Package）への参照を保持する必要があります [13]。
シンボル構造体の定義案**:**

コード スニペット

```
(type $symbol (struct
  (field $name (ref $string))      ;; 不変の文字列（印字名）
  (field $value (mut anyref))      ;; 動的な値（Unbound状態の管理が必要）
  (field $function (mut anyref))   ;; グローバル関数定義
  (field $plist (mut anyref))      ;; プロパティリスト
  (field $package (mut anyref))    ;; ホームパッケージ
))
```

特筆すべきは $value スロットの「Unbound（未束縛）」状態の扱いです。Common Lispでは、値が代入されていないシンボルを評価するとエラーになります。Wasm GCでは、この未束縛状態を表すために、Lispレベルからはアクセスできない専用のセンチネルオブジェクト（UNBOUND）を定義し、初期値として設定する戦略が有効です [15]。
シンボルの同一性を保証するための「インターン（Interning）」機構は、パッケージシステムと連動します。パッケージは「文字列からシンボルへのマッピング」を管理するハッシュテーブル（またはトライ木）として実装され、Wasm GC の array をバッキングストアとして利用します。読み込み時（Reader）に文字列が与えられた際、このテーブルを検索し、既存のシンボルがあればそれを返し、なければ新規作成して登録します [14]。

2.4 数値塔（Numeric Tower）の実装

Common Lispは豊富な数値型（整数、比数、浮動小数点数、複素数）を持ちます。Wasm GC上での効率的な実装には、階層的な表現が必要です。

1. **Fixnum:** 前述の通り i31ref を使用します。これにより、頻出する小整数の計算においてヒープ割り当てが発生せず、GCの圧力を回避できます [11]。

2. **Bignum:** 31ビットを超える整数は、64ビット整数配列 (array (mut i64)) をラップした構造体として表現します。演算時には i31ref と Bignum の混在を処理するディスパッチロジックが必要です。
3. **Float:** Wasmの f64 はボックス化（Boxing）して構造体に格納します。ただし、局所的な数値計算においては、コンパイラの最適化によりアンボックス化（Unboxing）し、Wasmのスタック上で f64 として計算を行うことがパフォーマンス向上の鍵となります [8]。

2.5 配列とベクトル

Common Lispの配列は、多次元配列、フィルポインタ（Fill Pointer）付き配列、他の配列への変位（Displaced Array）など、複雑な機能を持ちます。これらをWasmの単純な array 型だけで表現することは不可能です。
階層的配列モデル:

- **Simple-Vector:** Wasmの (array (mut anyref)) に直接マッピングします。最も高速です。
- **Specialized Array:** 例えば (simple-array single-float (*)) は (array (mut f32)) に、文字列 string は (array (mut i8)) （UTF-8エンコーディング）にマッピングします。これによりメモリ効率が劇的に向上します [8]。
- **Complex Array:** 以下のフィールドを持つ構造体として実装します。
    - data: 実際のデータストレージ（Simple-Vector等への参照）。
    - dimensions: 次元情報を保持するリストまたはベクタ。
    - fill-pointer: 現在の有効長を示す i31ref。
    - displacement: 元の配列に対するオフセット。

アクセス時には、まずComplex Array構造体かSimple-Vectorかを ref.test で判定し、Complexであればオフセット計算を行った上で、下層の data 配列にアクセスする2段階の手順を踏みます。

3. 関数表現とクロージャ変換

Wasmにおける関数（func）は、コードそのものであり、データとしてヒープに保存したり、自由変数をキャプチャしたりする機能を持っていません。しかし、Common Lispの関数はファーストクラスオブジェクトであり、レキシカルな環境を閉包（クロージャ）する必要があります。このギャップを埋めるために「クロージャ変換（Closure Conversion）」が必須となります [12]。

3.1 クロージャ構造体の設計

すべてのLisp関数は、Wasmレベルでは以下の構造体として表現されます。

コード スニペット

```
(type $closure (struct
  (field $code (ref $func_type)) ;; コンパイルされたWasm関数への参照
  (field $env (mut anyref))      ;; キャプチャされた環境（自由変数）
))
```

関数呼び出し (funcall f arg1...) は、Wasmでは以下のような手順に変換されます。

1. f がクロージャ構造体であることを確認（ref.cast）。
2. 構造体から $code（関数ポインタ）と $env（環境）を取り出す。
3. call_ref 命令を用いて $code を呼び出す。この際、$env を第**1**引数として明示的に渡す。

つまり、Lisp上の関数 (lambda (x) (+ x y)) は、Wasmレベルでは (func (param $env anyref) (param $x anyref)...) というシグネチャに変換され、y は $env から取り出されます [16]。

3.2 関数シグネチャと動的ディスパッチの問題

Wasmの call_ref は厳密に型付けされており、呼び出し元と呼び出し先のシグネチャ（引数の数と型）が完全に一致している必要があります。しかし、Common Lispの関数は可変長引数（&rest）やキーワード引数（&key）を持ち、実行時まで引数の数が確定しない場合があります。
Guile Hoot等の先行実装では、この問題を解決するために、**統一シグネチャ（Uniform Representation）**を採用するアプローチが見られます [6]。
アプローチA：完全統一シグネチャ
すべての関数を (param $env anyref) (param $args anyref)（引数はリストとして渡す）という形式でコンパイルします。

- メリット：実装が単純で、call_ref の型不一致が発生しない。
- デメリット：すべての関数呼び出しで引数リストのコンス（メモリ割り当て）が発生し、パフォーマンスが著しく低下する。

アプローチB：多形ディスパッチ構造体
クロージャ構造体に、異なるアリティ（引数数）に対応する複数の関数ポインタスロット（エントリーポイント）を持たせます。

コード スニペット

```
(type $closure (struct
  (field $code_0 (ref null $func_0)) ;; 0引数用
  (field $code_1 (ref null $func_1)) ;; 1引数用
  (field $code_2 (ref null $func_2)) ;; 2引数用
  (field $code_N (ref null $func_N)) ;; 汎用（リスト渡し）
  (field $env (mut anyref))
))
```

呼び出し側は、引数の数に応じて適切なフィールドを選択して call_ref します。対応するスロットが null の場合（例：2引数の関数を3引数で呼んだ場合）は、汎用の $code_N にフォールバックするか、エラーを送出します。この手法により、頻出する少数引数の呼び出しを高速化できます [7]。

3.3 多値返却（**Multiple Values**）の実装

Common Lispは (values 1 2) のように多値を返すことができます。Wasm自体も多値返却（Multi-value return）をサポートしていますが、その数は静的に固定されている必要があります。一方、Lispの values は動的に数が変化し得ます。
これに対処するため、Lispの多値は以下の戦略で実装します [19]。

1. プライマリ値の返却: 関数の戻り値として、常に第1値（Primary Value）を返します。
2. セカンダリ値の格納: 第2値以降は、スレッドローカルな（Wasmグローバル変数として定義された）「多値バッファ（Multiple Values Buffer）」に格納します。
3. 呼び出し側の処理: multiple-value-bind 等のフォームは、関数から戻った直後にこのバッファを確認し、値を取り出します。通常の呼び出しではバッファは無視され、第1値のみが使われます。これにより、Wasmの静的なシグネチャ制約を守りつつ、Common Lispの多値セマンティクスを再現できます。

4. 制御フローと例外処理

Lispの強力な制御フロー（非局所脱出、Unwind-Protect、末尾再帰）をWasm上で実現するには、Wasmの最新の制御命令を駆使する必要があります。

4.1 末尾呼び出し最適化（Tail Calls）

Common Lisp（およびScheme）において、末尾再帰がスタックを消費せずに実行できることは極めて重要です。Wasmの tail_call プロポーザル（return_call 命令）は、現在のスタックフレームを破棄して次の関数にジャンプする機能を提供します [22]。
Safari 18.2を含む主要ブラウザでサポートが進んでいるため、コンパイラは末尾位置（Tail Position）にある関数呼び出しを積極的に return_call に変換すべきです。これにより、無限ループに近い再帰呼び出しも安全に実行可能となります。tail_call がサポートされていない環境へのフォールバックとして「トランポリン（Trampoline）」方式（関数が継続を返し、ループでそれを実行する）も考慮すべきですが、パフォーマンスコストが高いため、基本的には return_call の利用を前提とすべきでしょう [24]。

4.2 非局所脱出と例外処理

block/return-from や tagbody/go、そして catch/throw は、レキシカルなスコープを超えて制御を移動させます。Wasmの Exception Handling（EH）プロポーザルは、これらを実装するための基盤を提供します [26]。

- **Throw:** Lispの (return-from block-name value) は、Wasmの throw 命令にマッピングされます。この際、脱出先のブロックを一意に識別する「タグ」と、戻り値を例外ペイロードとして投げます。
- **Catch:** block 構文の入り口には Wasmの try_table を配置し、特定のタグを持つ例外を捕捉（catch）します。捕捉後、ペイロード（戻り値）をスタックに積み、通常の実行フローに復帰します。
- **Unwind-Protect:** (unwind-protect form cleanup) は、try_table の catch_all 節（または cleanup 機能）を用いて実装します。form の実行中に例外（throw や return-from）が発生してスタックが巻き戻される際、Wasmランタイムは自動的に catch ブロックを実行するため、そこで cleanup 処理を行い、その後例外を再送出（rethrow）することで、Lispのセマンティクスを忠実に再現できます [28]。

5. 動的スコープ（スペシャル変数）の実装戦略

Common Lispの最大の特徴の一つである「スペシャル変数（動的スコープ変数）」は、レキシカルな環境ではなく、実行時のコールスタックに依存して値が決定されます。Wasmはスタックの検査（

Stack Walking）を直接許可していないため、この機構を自前で実装する必要があります [15]。

5.1 シャローバインディング（Shallow Binding）と証跡（Trail）

最も効率的な実装は「シャローバインディング」と呼ばれる手法です。

1. グローバル値スロット: 各シンボル構造体は、現在の値を保持する $value フィールドを持ちます。
2. バインディング時の退避: (let ((spec-var new-val))...) のようなスペシャル変数の束縛に入るとき、コンパイラは以下のコードを生成します。
    - 現在の $value を取得し、スレッドローカルな「バインディングスタック（またはTrail）」にプッシュします。
    - シンボルの $value を new-val で上書きします。
3. 変数の参照: 単にシンボルの $value フィールドを読み取るだけです。これにより、スペシャル変数のアクセスは、レキシカル変数とほぼ同等の O(1) の速度で実行できます [29]。
4. バインディングの復元（**Unwinding**）: let のスコープを抜ける際（正常終了、非局所脱出を問わず）、unwind-protect（Wasmの try_table）を利用して、バインディングスタックから古い値をポップし、シンボルの $value に書き戻します。

この手法は、Wasmが現状シングルスレッド（Web Worker間でのメモリ共有はあっても、単一の実行コンテキスト内では直列実行）であることを前提としており、マルチスレッド化（SharedArrayBuffer利用時）の際には、シンボル値へのアクセスに排他制御やスレッドローカルストレージ（TLS）の考慮が必要となりますが、現在のブラウザWasm環境では最もパフォーマンスに優れた戦略です。

6. 動的コンパイル（Eval/JIT）のアーキテクチャ

Wasmはハーバードアーキテクチャを採用しており、実行コードとデータメモリが厳密に分離されています。したがって、Lispの eval や compile のように、実行時に生成したデータをコードとして実行することは、セキュリティ上の制約から直接的には不可能です [31]。

6.1 ティアード実行モデル

この制約を回避しつつ動的な評価を実現するために、2段階（Tiered）の実行モデルを提案します。

Tier 1: Wasm内インタプリタ

コンパイラ自体の一部として、Wasmで記述されたLispインタプリタ（S式評価器）を実装します。

- REPLでの対話的実行や、一度しか実行されないコードに対して使用されます。
- S式（リスト構造）を再帰的にトラバースして評価します。
- 速度は遅いですが、コンパイルのオーバーヘッドがなく、即応性に優れます [1]。

Tier 2: 動的Wasmモジュール生成（JIT）

頻繁に実行される関数や compile 関数が呼び出された場合、以下の手順でネイティブWasmコードへのコンパイルを行います 6。

1. インメモリ・コンパイル: Lispで書かれたコンパイラが、対象のS式を解析し、Wasmバイナリ形式（Uint8Array）をメモリ上に生成します。Guile Hootプロジェクトは、このツールチェーン全体をSchemeで実装し、Wasm内で完結させています。
2. モジュール実体化（**Instantiation**）: 生成したバイナリを、ホスト（JavaScript）の WebAssembly.instantiate() 関数に渡します。Wasm単体では自身のコード領域を拡張できないため、JS側のAPIを呼び出す必要があります。

3. 動的リンク: 新しく生成されたモジュールは、メインのLispランタイムのメモリ（GCヒープ）や関数テーブルを「インポート」する必要があります。これにより、新旧のコードが同じヒープを共有し、既存のシンボルや関数にアクセスできるようになります。
4. ホットパッチ: 新しいモジュールからエクスポートされた関数（func_ref）を受け取り、それをシンボルの関数スロットやCLOSのディスパッチテーブルに登録します。これにより、次回の呼び出しからコンパイル済みの高速なコードが実行されます。

このアプローチは、JSの eval などを経由せずにWasmのセマンティクス内で完結する「正当な」JIT手法であり、セキュリティサンドボックスを維持したまま高いパフォーマンスを実現します。

7. CLOS（Common Lisp Object System）の実装

CLOSはCommon Lispの強力なオブジェクト指向システムであり、多重継承、総称関数（Generic Function）、メソッド結合などをサポートします。

7.1 クラスとインスタンスのレイアウト

クラス（standard-class）自体もメタオブジェクトとしてヒープに存在します。インスタンスは、クラスへの参照と、スロット値を保持するベクタを持つ構造体として表現されます。

コード スニペット

```
(type $instance (struct
  (field $class (ref $standard-class)) ;; クラスメタオブジェクト
  (field $slots (ref $vector))        ;; スロット値（ベクタ）
))
```

7.2 総称関数のディスパッチ

Wasmは動的なジャンプ（Computed Goto）が制限されているため、総称関数のメソッドディスパッチロジックは工夫が必要です。

1. キャッシュベースのディスパッチ: 各総称関数は「識別ネットワーク（Discrimination Network）」や「キャッシュ」を持ちます。これは、引数のクラス（または型）をキーとし、対応するメソッド関数（func_ref）を値とするハッシュテーブルです [33]。
2. ディスパッチ関数: 総称関数が呼ばれると、まずこのキャッシュを検索します。ヒットすればメソッドを呼び出し、ミスすればLispで書かれた低速なメソッド解決ロジック（compute-applicable-methods）を実行し、結果をキャッシュに登録します。

静的なWasm struct の型システムを利用してメソッド呼び出しを最適化することは困難（Lispのクラス階層とWasmの型階層は必ずしも一致しないため）ですが、Wasm GCの高速なフィールドアクセスを活用することで、キャッシュ検索のオーバーヘッドを最小限に抑えることが可能です。

8. ガベージコレクションとメモリ管理の統合

Wasm GCを採用する最大の利点は、メモリ管理の実装コストとランタイムの複雑さを大幅に削減できる点です。

8.1 サイクルコレクションとホスト統合

従来のWasm実装では、Lisp側で作成したオブジェクトがDOM要素への参照を持ち、そのDOM要素のイベントハンドラがLispのクロージャを保持している場合、相互に参照し合っているためにメモリが解放されない（メモリリーク）問題が発生していました。
Wasm GCでは、Lispの struct とDOMの externref は同じGC宇宙（Universe）に存在します。ブラウザのGC（V8のOrinocoなど）は、WasmヒープとJSヒープを横断してトレースを行う能力を持っているため、このような言語間のサイクルも自動的に検出し、回収することができます 3。これは、長時間動作するWebアプリケーションにとって決定的な利点です。

8.2 アロケーションのオーバーヘッド

Wasm GCの struct.new 命令は、エンジン内部で高度に最適化されています。多くのエンジンは「バンポインタ（Bump Pointer）」割り当てを採用しており、線形メモリ上で malloc を実行してフリーリストを操作するよりも高速にオブジェクトを生成できます。また、短命なオブジェクト（Consセルなど）はジェネレーショナルGCによって効率的に回収されるため、Lispのような頻繁にメモリを割り当てる言語にとって理想的な特性を持っています [4]。

9. 結論

WebAssembly GCをターゲットとしたCommon Lispコンパイラの実装は、従来の線形メモリモデルからの脱却を意味し、Webブラウザというプラットフォームの特性を最大限に活かすアプローチです。

1. **データ構造:** i31ref、struct、array を駆使してLispのデータ型を直接表現することで、ポインタ操作のオーバーヘッドを排除し、メモリ効率を向上させます。
2. **制御フロー:** tail_call と try_table を利用することで、スタックオーバーフローを恐れずに再帰や非局所脱出を記述できます。
3. **動的性:** Wasm内インタプリタと、instantiate を利用したモジュール生成JITを組み合わせることで、Lisp本来の対話的な開発体験と高い実行性能を両立させます。
4. **GC:** ホストGCへの委譲により、複雑なGC実装から解放されるだけでなく、DOMとの連携におけるメモリ安全性が保証されます。

実装の難易度は高い（特にクロージャ変換や動的スコープの実装において）ですが、Guile Hoot等の先行事例が示すように、技術的な構成要素はすでに揃っており、実用的なパフォーマンスを持つCommon Lisp処理系をブラウザ上で実現することは十分に可能です。これは、WebをLispマシンのように扱う未来への確かな第一歩となるでしょう。

引用文献

1. A tiny Lisp interpreter compiled to WebAssembly (with pluggable GC + heap visualizer), 12月 21, 2025にアクセス、https://www.reddit.com/r/WebAssembly/comments/1p9hmzj/a_tiny_lisp_interpreter_compiled_to_webassembly/
2. Using Common Lisp from inside the Browser - TurtleWare, 12月 21, 2025にアクセス、https://turtleware.eu/posts/Using-Common-Lisp-from-inside-the-Browser.html
3. A new way to bring garbage collected programming languages efficiently to

WebAssembly, 12月 21, 2025にアクセス、https://v8.dev/blog/wasm-gc-porting
4. WebAssembly Garbage Collection (WasmGC) now enabled by default in Chrome | Blog, 12月 21, 2025にアクセス、https://developer.chrome.com/blog/wasmgc
5. Hoot: Scheme on WebAssembly - Spritely Institute, 12月 21, 2025にアクセス、https://spritely.institute/hoot/
6. Guile Hoot v0.1.0 released! - Spritely Institute, 12月 21, 2025にアクセス、https://spritely.institute/news/guile-hoot-v0-1-0-released.html
7. Schism - Scheme and Functional Programming Workshop, 12月 21, 2025にアクセス、https://www.schemeworkshop.org/2018/Holk.pdf
8. Data types (Guile Hoot), 12月 21, 2025にアクセス、https://files.spritely.institute/docs/guile-hoot/0.4.1/Data-types.html
9. i31 as a ref type dimension like null · Issue #130 · WebAssembly/gc - GitHub, 12月 21, 2025にアクセス、https://github.com/WebAssembly/gc/issues/130
10. Why do we need the type of i31 in WasmGC proposal? - Stack Overflow, 12月 21, 2025にアクセス、https://stackoverflow.com/questions/77468063/why-do-we-need-the-type-of-i31-in-wasmgc-proposal
11. I31 in wasmtime - Rust, 12月 21, 2025にアクセス、https://docs.wasmtime.dev/api/wasmtime/struct.I31.html
12. How can I compile closures in WebAssembly?, 12月 21, 2025にアクセス、https://langdev.stackexchange.com/questions/3621/how-can-i-compile-closures-in-webassembly
13. 2.3. Symbols, 12月 21, 2025にアクセス、https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node27.html
14. An Earnest Guide to Symbols in Common Lisp - kevingal.com, 12月 21, 2025にアクセス、https://kevingal.com/blog/cl-symbols.html
15. 5.2.1 Special variables - LispWorks, 12月 21, 2025にアクセス、https://www.lispworks.com/documentation/lcl50/aug/aug-109.html
16. Implementing Closures and First-Class Functions in WebAssembly - Reddit, 12月 21, 2025にアクセス、https://www.reddit.com/r/ProgrammingLanguages/comments/1fhdvtd/implementing_closures_and_firstclass_functions_in/
17. funcref <: anyref, or not? · Issue #293 · WebAssembly/gc - GitHub, 12月 21, 2025にアクセス、https://github.com/WebAssembly/gc/issues/293
18. hoot's wasm toolkit - wingolog, 12月 21, 2025にアクセス、https://wingolog.org/archives/2024/05/24/hoots-wasm-toolkit
19. Returning multiple items from a WASM executable - Help - Ziggit, 12月 21, 2025にアクセス、https://ziggit.dev/t/returning-multiple-items-from-a-wasm-executable/7748
20. Multi-Value All The Wasm! - Mozilla Hacks - the Web developer blog, 12月 21, 2025にアクセス、https://hacks.mozilla.org/2019/11/multi-value-all-the-wasm/
21. `values` vs `list` for returning multiple values from Lisp form, 12月 21, 2025にアクセス、https://softwareengineering.stackexchange.com/questions/268107/values-vs-list-for-returning-multiple-values-from-lisp-form

22. Wasm 3.0 Completed - WebAssembly, 12月 21, 2025にアクセス、https://webassembly.org/news/2025-09-17-wasm-3.0/
23. WasmGC and Wasm tail call optimizations are now Baseline Newly available - web.dev, 12月 21, 2025にアクセス、https://web.dev/blog/wasmgc-wasm-tail-call-optimizations-baseline
24. Show HN: Common Lisp running natively over WebAssembly for the first time | Hacker News, 12月 21, 2025にアクセス、https://news.ycombinator.com/item?id=31590819
25. Tail call optimization (WebAssembly) | Can I use... Support tables for HTML5, CSS3, etc, 12月 21, 2025にアクセス、https://caniuse.com/wf-wasm-tail-call-optimization
26. Exceptions in Cranelift and Wasmtime - Chris Fallin, 12月 21, 2025にアクセス、https://cfallin.org/blog/2025/11/06/exceptions/
27. `finally` support? · Issue #158 · WebAssembly/exception-handling - GitHub, 12月 21, 2025にアクセス、https://github.com/WebAssembly/exception-handling/issues/158
28. 7.11. Dynamic Non-Local Exits - LIX, 12月 21, 2025にアクセス、https://www.lix.polytechnique.fr/~liberti/public/computing/prog/lisp/cltl/clm/node96.html
29. Special Variable - C2 wiki, 12月 21, 2025にアクセス、https://wiki.c2.com/?SpecialVariable
30. Dynamic Scoping - C2 wiki, 12月 21, 2025にアクセス、https://wiki.c2.com/?DynamicScoping
31. just-in-time code generation within webassembly - wingolog, 12月 21, 2025にアクセス、https://wingolog.org/archives/2022/08/18/just-in-time-code-generation-within-webassembly
32. Tutorial (Guile Hoot), 12月 21, 2025にアクセス、https://files.spritely.institute/docs/guile-hoot/0.1.0/Tutorial.html
33. Dynamic Dispatch in C using virtual method table - Stack Overflow, 12月 21, 2025にアクセス、https://stackoverflow.com/questions/15733590/dynamic-dispatch-in-c-using-virtual-method-table
34. Virtual function dispatching - Nicholas Frechette's Blog, 12月 21, 2025にアクセス、https://nfrechette.github.io/2015/04/30/virtual_function_dispatching/