修 士 論 文

最適化器の生産性向上を目的とした
コンパイラフレームワークの設計と実現

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

斉木 晃治

2003 年 3 月

修 士 論 文

最適化器の生産性向上を目的とした コンパイラフレームワークの設計と実現

指導教官　権藤克彦 助教授

審査委員主査　権藤克彦 助教授
審査委員　片山卓也 教授
審査委員　落水 浩一郎 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

110046 斉木 晃治

提出年月: 2003 年 2 月

概 要

コンパイラフレームワークは，最適化器の開発におけるコスト削減と再利用性の実現を目的に開発されたものであるが，研究開発の過程における生産性については，軽視されている．我々は，最適化器開発の生産性向上のために，プラットフォーム非依存な構造化文書のための仕様である XML に着目した．XML の処理は，多くのプログラミング言語，プラットフォームでサポートされており，XML プログラムの開発者は用途に応じて自由にそれらを選択可能である．我々は，XML のデータスキーマに基づくコンパイラフレームワークを実装し，XML ベースの最適化器の実装と予備評価を行なった．C 言語サブセットに対するものではあるが，XML とその関連規格が，最適化器の生産性向上に有効であることを確認した．

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　背景

現在のマイクロプロセッサはソフトウエアレベルでの高度な最適化がなされてはじめて，その性能を十分に発揮できるように設計されている．並列計算機では，これに加えて計算機の構成とアプリケーションの処理に応じた最適化が必要とされる．また，組み込み機器では性能以外にもコードサイズや消費電力の観点からも最適化が必要とされる．

マイクロプロセッサ及び計算機アーキテクチャの複雑化，高性能化に対応すべく，さまざまな最適化手法が研究されている．特に並列計算機の分野では，ハードウエアの最高性能と実効性能の差が大きいのが実状で，並列化コンパイラによる最適化が課題となっている．しかし，最適化コンパイラは非常に大規模で複雑なシステムであり，開発には多大な時間とコストがかかる．

この問題を解決するためには，パーサやコードジェネレータなどのすべてのコンパイラで共通な部分を再利用し，最適化器のみの開発で新たな最適化手法を組み込んだコンパイラを開発できる拡張可能な環境が必要である．コンパイラフレームワークはコンパイラをオブジェクト指向に基づいてモジュール化し，ユーザーからの変更を可能とすることによって，これを実現したものである．

SUIF[Gro] に代表されるコンパイラフレームワークの多くが，ソースコードを受け付けるフロントエンド部分と機械語コードを出力するバックエンド部分，プログラムの中間表現とそれにアクセスするためのオブジェクト指向の API から構成される．コンパイラフレームワークでは最適化器はローダブルモジュールとして実現されている．SUIF ではこれに加えて中間表現をファイルに保存するためのデータフォーマットも定義している．

しかし，既存のコンパイラフレームワークの中間表現は特定のプログラミング言語に限定してインターフェースを提供しており，最適化器の開発者が実装言語を自由に選択することができない．このため，最適化手法の実証実験やプロトタイプ作成のような本来，高級言語を用いて行なう開発工程を実施することができないため生産性を低下させる要因となっている．また，CASE ツールなどでの中間表現の 2 次的利用においても，コンパイラフレームワークを介す必要があるため開発効率が悪い．

以上のことから，オブジェクト指向のコンパイラフレームワークは最終成果物の実装には適しているものの，最適化手法の研究開発のすべての段階で用いるためには多くの問題を抱えていると考える．したがって，より効率的に最適化器の実装を行うことのできる生産性の高いコンパイラフレームワークの実現が必要である．

1.2 目的

本研究の目的は，最適化器の研究開発における生産性向上を目的としたコンパイラフレームワークを XML(Extensible Markup Language) を用いて構築し，その有効性を評価・検討することにある．生産性を向上させる方法は複数考えられるが，我々は特に最適化器実装における再利用性の向上に重点を置いて，フレームワークの設計を行なった．

XML を用いたコンパイラフレームワークとは，オブジェクト指向の API によって構築されるフレームワークの対極に位置するもので，データスキーマを中心としたフレームワークである．API レベルではなく，データスキーマレベルでコンパイラへのアクセスを可能にすることで以下のことが期待できる．

1. 実装言語の選択肢の多様化とそれに伴なう新たな開発手法の導入
   例えば，スクリプト言語などの高級言語により，最適化器のプロトタイプ実装が可能になる．
2. 中間表現の 2 次的利用の促進
   中間表現を XML 文書として保存することで，CASE ツールなどから利用可能．
3. コンパイラフレームワーク自体の小型化
   コンパイラに最適化器を組み込む必要がなくなる．コンパイラは，XML と中間表現のマッピングを定義すればよい．

我々は再利用性を向上させるために，XML 関連技術の中でも XML 文書に対する検索処理を記述する XPath 言語に注目した．XPath は独自の文法を持つ検索言語で，プログラム中や XSLT スタイルシート言語で使用される．本研究で提案するフレームワークは，複数の異なるプログラミング言語による実装を可能とすることを特徴の 1 つとしているため，モジュールやライブラリとは異なった再利用の方法が必要となる．そこで XPath によって記述された検索式をリポジトリ化し，必要に応じて最適化器に埋め込むことで再利用性を実現する．

1.3 アプローチ

本研究で提案・構築するコンパイラフレームワークはオブジェクト指向フレームワークの形をとらないため，他のコンパイラフレームワークとは異なったアプローチで構築する必要がある．我々は次の 2 つのアプローチによるコンパイラフレームワークの構築を試みた．

- 既存のコンパイラをベースとする手法

  オープンソース開発を中心に，幅広く利用されているコンパイラである GCC をベースとする．GCC の中間言語 RTL を XML を用いてマークアップすることによって構築を行なう．実際に使用されているコンパイラをベースとするため，構築された

フレームワークを現実のソフトウエアのソースに対して適用できる．GCC のソースコードの解析と修正を行なう必要がある．

- C 言語サブセットを対象に新たに構築する手法

  C 言語のサブセットである XC 言語を対象にフレームワークを構築する．GCC をベースとしたものとは異なり，抽象構文木 (AST) をマークアップする．フレームワークの実用性は低いが，開発は容易である．

これらのコンパイラフレームワークは RTL もしくは，通常のコンパイラと同様にソースコードを読み込み，コンパイラによって生成された RTL または AST を XML 文書に変換する．次に最適化を実装した外部プログラムを起動し，変換した XML 文書を渡す．外部プログラムは，受け取った XML 文書上で最適化を行ない，結果を XML 文書として出力する．コンパイラフレームワークは外部プログラムが出力した XML 文書を受け取り，再び RTL もしくは AST に変換する．起動される外部プログラムを外部オプティマイザと呼ぶ．外部オプティマイザを変更，組み合わせることにより，プログラムごとに異なる最適化を施すことができる．

1.4 結論

2 つの手法による実装を試みた結果，RTL をベースにした実装では，RTL がそれ自体で独立しておらず，GCC 内部のデータ構造に依存していることが原因で十分な評価を行なうまでの実装には致らなかった．これはオープンソースソフトウエアの研究目的での利用の難しさを示している．

RTL を基盤としたコンパイラフレームワークの構築は RTL の XML 形式での出力に留まっているため，RTL の 2 次的利用について，視覚化ツールの実装を行なって評価を行なった．その結果では，CASE ツールへの応用が容易であることが明らかになった．また，コンパイラの行なう最適化の過程を確認できるので，教育目的の利用も可能である．

次に XC をベースとしたフレームワーク上で実装言語に Python 言語を用いて，最適化器実装の予備実験を行い，以下のことを確認した．

- 一旦，XML 文書として出力された抽象構文木は，他の最適化において参照することで再利用が可能である．
- 最適化器のテストにおいては，既に出力された XML 文書を入力とすることで，最適化器のみでデバッグ，テストを行なうことが可能である．
- XPath による記述は最適化器の間で再利用が可能である．
- XML と高級言語によって実装，デバッグ，テストを短かいサイクルで行なうことができた．

この予備実験の結果に限定すれば，XML とその関連技術，そして高級言語が最適化器の生産性向上に有効であるとの結論を得た．しかし，XML に基づくコンパイラフレームワークの有効性を確かめるには，より実用的なコンパイラフレームワークの構築と最適化器の実装実験が必要である．

1.5 本論文の構成

第 1 章 はじめに コンパイラフレームワークの必要性について述べた上で，既存のコンパイラフレームワークの問題点と XML による解決について述べ，研究のアプローチとそこから得られた結論について概説する．

第 2 章 XML，及び関連技術の概要 XML は構造化文書のための汎用性のあるフォーマットである．本研究では，これをコンパイラフレームワークにおける中間表現として用いている．第 2 章では XML とその関連技術である XPath について述べる．

第 3 章 コンパイラフレームワーク技術の概要と現状 コンパイラフレームワークの持つ機能について述べ，既存のコンパイラフレームワークとして SUIF，CFL，OpenJIT の概要を示す．これを踏まえて，既存のコンパイラフレームワークの問題点を列挙し，コンパイラフレームワークにおける XML の有効性を示す．

第 4 章 既存のコンパイラをベースとするコンパイラフレームワークの構築 ここでは GCC とその中間言語 RTL について述べた上で，これをベースとしたコンパイラフレームワークの構築について述べる．

第 5 章 C 言語サブセットに対するコンパイラフレームワークの構築 C 言語のサブセットである XC 言語について概説し，これを対象としたコンパイラフレームワークと高級言語による最適化器の実装実験について述べる．

第 6 章 議論 最適化器の実装実験の結果に基づいて，中間表現の 2 次的利用法，XML および XPath の再利用性，最適化器実装における高級言語の利便性，そして本研究で試みた GCC の拡張における問題点と XML プログラミングインターフェースの内部実装について議論する．

第 7 章 おわりに 本論文についてまとめる．予備実験の結果から得られた結論として，XML に基づくコンパイラフレームワークが，予備実験に限れば有効であることと，今後さらにその実用性を高めた上での実験が必要があることを述べる．

第2章　XML，及び関連技術の概要

2.1　XML

2.1.1　XML 勧告 1.0

拡張可能なマーク付け言語 XML(Extensible Markup Language) は SGML(Standard Generalized Markup Language) のサブセットとして規定されたメタタグ言語である．XML の最初の勧告は 1998 年に W3C(World Wide Web Consortium) によって制定された．以下に XML の設計目標を示す．

1. XML は，インターネット上でそのまま使用できる．
2. XML は，広範囲のアプリケーションを支援する．
3. XML は，SGML と互換性を持つ．
4. XML 文書を処理するプログラムは容易に書ける．
5. XML では，オプションの機能はできるだけ少なくし，理想的には 1 つも存在しない．
6. XML 文書は，人間にとって読みやすく，十分に理解しやすい．
7. XML の設計は，すみやかに行う．
8. XML の設計は，厳密で，しかも簡潔なものとする．
9. XML 文書は容易に作成できる．
10. XML では，マーク付けの数を減らすことは重要ではない．

XML により記述された文書を XML 文書と呼ぶ．XML 文書は論理構造を持ったデータオブジェクトで，その実体は単純なテキストデータである．XML は XML 文書の論理構造を記述するための符号を規定している．また，XML は論理構造に関する制約を記述するための機構を提供する．

2.1.2　構造化文書によるデータ表現

XML 文書は主として次の 6 つの基本的な単位から構成される．

- **XML 宣言**

  その XML 文書が従う XML のバージョンと XML 文書の文字符号化について記述する．

- **文書型定義 (DTD)**

  XML 文書の論理構造に関する制約を記述する．DTD を独立したファイルとして記述し，XML 文書から参照することもできる．

- **開始タグ**

  XML 文書上の論理的な単位の始まりを表わす．タグは名前 (要素名) と属性を持つ．開始タグは符号'<' と'>' で表わす．

- **終了タグ**

  XML 文書上の論理的な単位の終わりを表わす．終了タグは対応する開始タグと同じ要素名を持つ．終了タグは要素名の前に'/' をつける．

- **属性**

  開始タグの持つ付加的な情報．名前と値の組．開始タグには空白で区切って複数の属性を記述できる．

- **文字列**

  上の 5 つ以外の部分全て，特に開始タグと終了タグに囲まれた中にある文字列はデータとして処理される．空白は処理される場合とされない場合がある．

開始タグと終了タグにかこまれた範囲を 1 つの要素と呼ぶ．要素はその内部にさらに別の要素を含むことができる．このときの外側の要素を親要素，内側の要素を子要素と呼ぶ．親要素と子要素の関係によって XML 文書は全体で 1 つの木構造になる．また，要素内に出現する文字列で，DTD にデータとして処理することが明示されているものは，PCDATA と呼ばれる．

DTD は XML 文書内に出現する要素とその属性について記述する．要素の記述では，XML 文書の論理構造を規定する要素内部の記述に関する制約 (内容モデル) を記述する．

XML 文書は，XML 宣言，文書型定義，ルート要素の順で記述する．ルート要素は DTD で定められた要素で，XML 文書で最も外側の要素である．以下に XML 文書の例を示す．

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <!DOCTYPE xmlsample SYSTEM "xmlsample.dtd">
 3 <xmlsample>
 4   <title>This is sample XML document</title>
 5   <body lang="japanese">
 6     <p>これは XML 文書のサンプルです．</p>
 7     <b>要素は子要素を持ちます．</b>
 8     <note>ルート要素は一番外側の要素です．</note>
 9   </body>
10 </xmlsample>
```

1 行目が XML 宣言で，この XML 文書が XML のバージョン 1.0 に準拠し，文字符号化は UTF-8 で行なわれていることを示す．2 行目が文書型定義で，ルート要素が xmlsample であることを示す．ここでは文書型定義本体は外部から参照している．3 行目以降がこの XML 文書のデータの本体である．ルート要素 xmlsample は子要素として title，body 要素を持つ．title は子要素を持たず，内容として文字列を持つ．body 要素は p，b，note の 3 つの要素を子要素に持つ．また，body 要素は属性 lang を持ち，その値は japanese である．要素を木のノードとし，親要素を木の上位のノードに，子要素を下位のノードとして，この XML 文書を木の形で表現すると図 2.1 のようになる．

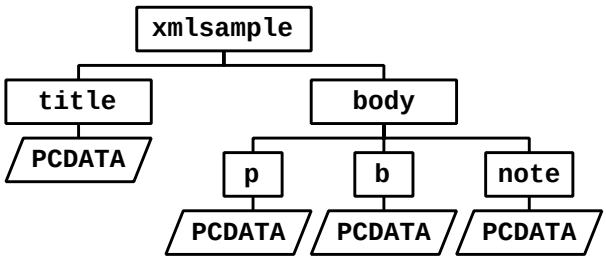

図 2.1: サンプル XML 文書の木構造

2.1.3 文書型定義と妥当性の検証

ある XML 文書内で使用可能なタグの集合は，文書型定義 DTD(Document Type Definition) で定義する．DTD には要素の名前，要素が取り得る属性の種類と値の候補，そして XML 文書の論理構造に関する制約を与える要素と子要素の関係を記述する．これは内容モデルと呼ばれ，要素が内部に持つ子要素の種類とその順序を記述する． DTD を設計するということは，すなわち，XML に基づくマークアップ言語を設計するということである．

データの論理構造を規定するものを一般にデータスキーマ (Data Schema) と呼び，データスキーマの記述に使用する言語をスキーマ言語と呼ぶ．XML 勧告 1.0 が提供している DTD もスキーマ言語である．DTD は拡張正規表現に近い記述で要素と子要素の関係を記述する．

XML 文書がデータスキーマに沿ったものになっていることを検査する作業を妥当性の検証 (Validation) と呼ぶ．妥当性の検証は処理する XML 文書の数とサイズによっては大きなオーバヘッドとなることもあるので，省略することもできる．この場合，XML 文書は，XML の基本的な制約にのみ適合した整形式 XML 文書として処理される．整形式 XML 文書は，機械的に生成された XML 文書をシステム間で交換する場合のような，常に DTD に沿った妥当な XML 文書が与えられることが保証された処理において，妥当性の検証を省略することによって処理効率を向上させるために用いることができる．

XML 勧告 1.0 の DTD は簡潔で，妥当性の検証を行なうソフトウエアを容易に実装可能である．しかし，DTD では 1 つの要素に対して 1 つの内容モデルしか定義できないなど，柔軟性に欠ける部分が多く，これを改善すべく XML Schema[WWWa, WWWb, WWWc], RELAX NG[Org] などのスキーマ言語の開発が行なわれている．

本研究でスキーマ言語として DTD を用いてマークアップ言語の設計を行なうが，これは，本研究で開発するマークアップ言語が DTD で十分記述可能であるということと，DTD に基づく妥当性検証のためのソフトウエアが最も一般的で安定した実装が提供されていることによる．

2.2 XPath

XPath(XML パス言語)[CD] は，XML 文書の 1 部分をアドレッシングするための言語である．XPath の最初の勧告は 1999 年 11 月 16 日に W3C によって規定された．XPath は XML 文書の変換を行なう XSLT[WWWd]，XML 文書上の特定の位置を指定するための XPointer[DJG+] で使用されるように設計されている．

2.2.1 データモデル

XPath では XML 文書を要素と子要素の関係によってつくられる木としてモデル化している．XPath のデータモデルでは木のノードは XML の要素に対応する．XPath のデータモデルにおけるルートノードは，XML 文書のルート要素と一致しない．ルート要素に対応するノードはルートノードの直下に存在する．文書のルート要素は木の根にあたるルートノードに対応する．属性は属性ノードとして各要素に結び付けられているが，親子関係ではない．

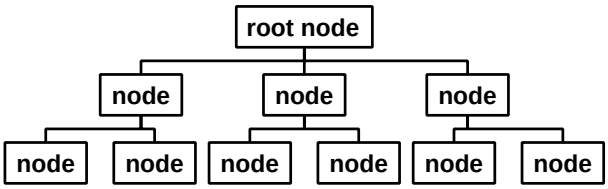

図 2.2: XPath のデータモデル

2.2.2 基本的な構文

XPath の基本的な構造は XPath 式と呼ばれる．XPath 式は評価されると，オブジェクトを生成する．このオブジェクトは次のいずれか 1 つである．

- ノードセット (ノードの集合)
- ブール値
- 数値
- 文字列

本研究では，主として XML 文書の中から特定の条件にマッチするノードの集合を得るために XPath を使用する．そのため，ここではノードセットを返す XPath 式であるロケーションパスに絞って話をすすめる．

ロケーションパスは，'/' で区切られた複数のロケーションステップの連なりで構成される．ロケーションパスの結果得られるノードセットはロケーションパスの各ロケーションステップを左から右に評価した結果得られたものである．

ロケーションステップは，評価される文脈に応じてノードセットを返す．ロケーションステップを評価する文脈は 1 つのノードであり，これを文脈ノードと呼ぶ．ロケーションステップは 1 つ前のステップの結果得られたノードセットの各ノードを文脈ノードとして評価し，それぞれの結果の和をそのロケーションステップの結果とする．

ロケーションパスには，相対パスと絶対パスがある．相対パスは，特定のノードを文脈ノードとして評価と行なう．絶対パスは，'/' に相対パスをつなげたものである．この'/' は，文脈ノードを包含しており，XML 文書のルートノードを文脈ノードとして選択する．

ロケーションステップは，ノードを選択するノードテスト，選択されたノードセットを絞り込むための条件式を記述した任意の数の述部，選択するノードと文脈ノードの木の上での関係を指定する軸から成る．ロケーションステップは軸，軸とノードテストを区切る'::'，角括弧でかこまれた 0 個以上の述部の順で記述する．

2.2.3 軸と文脈ノード

軸はロケーションステップにおいて，選択されるノードセットと文脈ノードの木構造の上での関係を指定する．軸を指定することで，ノードを検索する方向，結果として得られるノードセット内のノードの順序などを変更できる．XPath1.0 で規定されている軸を表 2.3 に示す.

2.3 XML プログラミングインターフェース

XML を処理するプログラムを書く場合，XML 文書を読み込むための XML パーサ，パーサによって読み込まれた XML 文書にアクセスするためのプログラミングインターフェースとその実装が必要である．加えて，XPath を使用する場合には XPath の処理系である XPath プロセッサも必要である．

軸	包含するノード
child	文脈ノードの子ノード
descendant	文脈ノードの子孫ノード
parent	文脈ノードの親ノード
ancestor	文脈ノードの祖先ノード
following-sibling	文脈ノードの継続する兄弟
preceding-sibling	文脈ノードの先行の兄弟
following	子孫ノードを除く，文脈ノードの後にあるすべてのノード
preceding	祖先を除く，文脈ノードの前にあるすべてのノード
attribute	文脈ノードの属性ノード
namespace	文脈ノードの名前空間ノード
self	文脈ノードのみ
descendant-or-self	文脈ノードと文脈ノードの子孫
ancestor-or-self	文脈ノードと文脈ノードの祖先

表 2.3: XPath 1.0 で規定されている軸

2.3.1 XML パーサ

XML パーサはテキストデータとして与えられる XML 文書を解析し，メモリに格納する．各種プログラミング言語，オペレーティングシステムにおいて，XML パーサを実現することによって，XML を処理するアプリケーションを実装することができる．

XML パーサによっては，DTD に基づく XML 文書の妥当性の検証を行なうことができる．妥当性を検証する XML パーサは，XML 文書が整形式であることに加えて，DTD を読み込み，XML 文書が DTD に沿ったものになっていることを検査し，必要に応じてエラーをアプリケーションに対して報告することができる．

XML パーサが読み込んだ XML 文書はメモリに格納される．メモリ上に格納された XML 文書にアクセスするためのインターフェースは複数存在し，XML パーサはそのうちの 1 つ，または複数を実装し，提供することも可能である．また，パーサによっては独自のインターフェースを備えたものもある．

代表的な XML パーサを以下に示す．

- expat[exp]
- XP[Cla]
- Xerces Java[Xer]
- libxml[lib]

expat は C 言語で実装された XML パーサである．非常にコンパクトで処理も高速だが，妥当性の検証を行なわないなど機能は限定されている．また，プログラミングインターフェースは SAX に近いが，独自のものを使用している．Xerces は Apache.org による XML パーサの実装で，広範囲にわたる XML の規格をサポートしている．Xerces は妥当性の検証を行なうことができる．また，SAX と DOM の 2 種類のプログラミングインターフェースに対応している．

2.3.2 DOM

DOM(Document Object Model)[ABC+] は，W3C によって規定された XML 文書操作のためのプログラミングインターフェースである．W3C 自身は DOM の実装を提供しておらず，IDL によるインターフェースの定義を提供している．DOM の仕様は，その機能により 1 から 3 までのレベルに分けられており，各レベルごとにさらに細かく分類されている．DOM の使用には XML パーサが必要である．

DOM は，XML に対して独自のデータモデルを定義している．DOM のデータモデルも XPath と同様に木であり，木の各ノードがプログラミング言語における 1 つのオブジェクトに対応している．オブジェクトの種類とその振舞いを定義するインターフェースは，継承関係を用いて階層的に定義されている．DOM は木へのノードの追加，削除，変更といったメソッドを提供する．図 2.4 に DOM Level 1 で規定されているインターフェース階層を示す．Node インターフェースを最上位のインターフェースとして，要素を表わす Element インターフェース，DOM のデータモデルにおけるルートノードとなる Document インターフェース，属性を表わす Attr インターフェースなどがある．

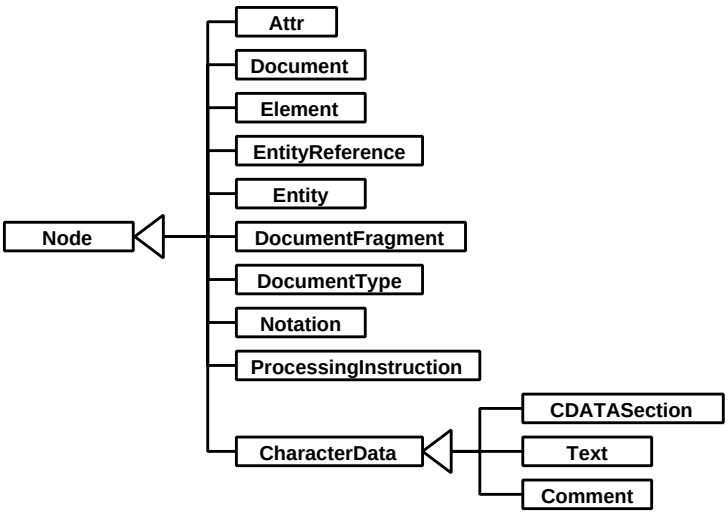

図 2.4: DOM Level 1 のインターフェース階層

DOM で XML を表現する木を特に DOM ツリーと呼ぶ．DOM ツリーは，メモリに置

かれるため，一旦 XML 文書から DOM ツリーを構築すれば，同じ XML 文書に繰り返しアクセスする場合に，毎回パースする必要がなく，高速である．しかし，一般にオブジェクト指向言語ではメモリ消費におけるオブジェクトのオーバーヘッドが大きいため，巨大な XML 文書を読み込むとメモリを圧迫する可能性がある．

2.3.3 SAX

SAX(Simple API for XML)[SAX] は，イベント駆動型のインターフェースである．SAX は XML 文書をメモリ上に保持することはしない．SAX をインターフェースとして用いるパーサは，XML 文書を読み込むと出現した開始タグ，終了タグ，文字列に対してイベントを発生させる．アプリケーションはこのイベントを受け取ることで XML 文書を処理する．

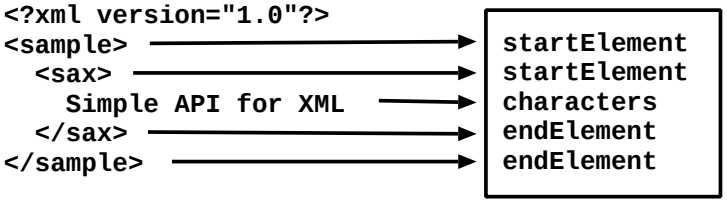

図 2.5: SAX による XML 文書の処理

SAX は，XML 文書をメモリ上に保持しないため，少ないメモリで動作する．しかし，同一の XML 文書に繰り返しアクセスするような処理では，逆にパースのためのコストが高くなる．

2.3.4 XPath プロセッサ

XPath 式を解釈実行するソフトウエアを XPath プロセッサと呼ぶ．XPath の主な用途が XSLT 内での使用であることもあり，XPath プロセッサ単体で提供されることはなく，XSLT プロセッサの一部として提供されている．

本研究では，XML パーサを用いて構築した DOM ツリー上で，XPath 式を評価し，条件を満すノードの集合を DOM のオブジェクトの集合として受け取り，処理する．XPath プロセッサ単体で使用可能で，かつ DOM ツリー上で動作する XPath プロセッサとして，本研究では Python 処理系上で利用可能な 4Suite[Fou] を用いている．

第3章　コンパイラフレームワーク技術の概要と現状

3.1　コンパイラフレームワークの必要性

現在のマイクロプロセッサは，高い最高性能を持っているが，その性能を最大限に発揮するためには，ソフトウエアがそのマイクロプロセッサに対して十分に最適化されている必要がある．

デスクトップPCのアーキテクチャでは，プロセッサの動作速度とデータバスの動作速度の差によって生じるレイテンシを短縮することが課題となっており，キャッシュの有効利用のための最適化が重要となる．

並列計算機では，ハードウエアの最高性能と実効性能の差が大きく，投資に見合うだけの成果が得られていない．現在の最適化技術では，すべてのプログラムをコンパイラが透過的に，並列実行可能なプログラムに分割，計算機資源への分配を行なうことができないため，アプリケーションとそれを動作させるハードウエアの特性を考慮したアプリケーションの実装が必要となる．

また，組み込みシステムでは，デスクトップ機のような処理速度優先の最適化は必ずしも必要ではなく，むしろコードサイズの削減や消費電力の低減を行なうことによって，システムの製造コストを抑えるための最適化が必要となる．

これに加えて，組み込み分野で利用されるプロセッサは種類が豊富で，市場のニーズに応じた様々なプロセッサが，常に開発されている．これらの個々のプロセッサについて，最適化コンパイラを開発することは，莫大なコストが必要であり，コンパイラの開発コストの削減が必要である．

このように最適化の要求はさまざまなものがあるが，現在の最適化技術ではこれらの要求の全てを満すことができないのが現状である．このため，現在さまざまな最適化手法の研究が行なわれている．

最適化を行なうソフトウエアのことを，最適化器 (Optimizer) と呼ぶ．最適化器の開発には，実験や評価，テストのために，開発した最適化器を組み込むコンパイラが不可欠である．しかし，実用に耐える機能を持ったコンパイラの実装は，莫大なコストと時間を要する．また，コンパイラは1つのプログラムに対して複数の最適化手法の適用を行なうことが一般的であるため，自分の実装した複数の最適化器や，他の研究者の開発した最適化器を組み合せて利用できることが望ましい．

このような最適化手法の実現における要求に応えるために，開発されたのがコンパイラ

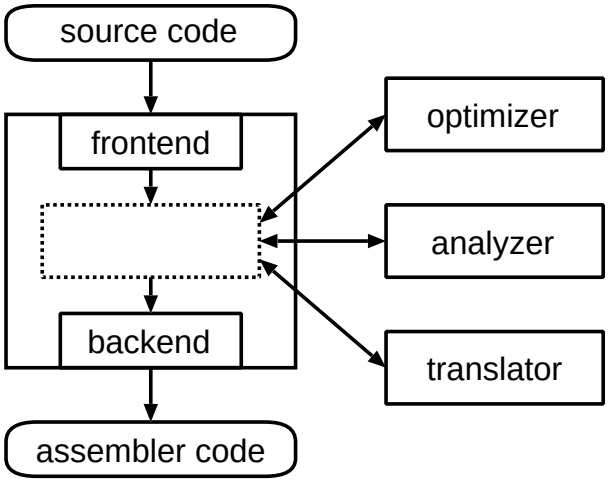

図 3.1: コンパイラフレームワークの基本的なアイデア

フレームワークと呼ばれるソフトウエアである．これは，コンパイラを再利用可能な部分と必要に応じて組み替えることのできる部分とに分け，再利用可能な部分をあらかじめ実装して提供することによって，最適化器の開発コスト削減を目指したものである．

3.2 コンパイラフレームワークの機能

現在，利用されているコンパイラフレームワークの多くが，コンパイラとしての基本的な機能 (パーサ，コードジェネレータ) の他に，プログラムの中間表現，中間表現にアクセスするためのプログラミングインターフェース，最適化器のためのランタイムシステムを提供している点で共通する．各フレームワークが異なっているのは，その実現方法や対応するプログラミング言語，プロセッサなどである．

3.2.1 中間表現

中間表現 (IR: Intermediate Representation) は，最適化を行なうために抽象構文木から生成されるデータオブジェクトである．中間表現は抽象構文木とアセンブラコードの中間に位置する．中間表現は，中間フォーマット，または中間言語とも呼ばれ，プログラムを最適化しやすい形で表現するだけでなく，プログラムの解析結果などの最適化の過程で得られた情報を格納する役割も持っている．

コンパイラフレームワーク上で実装される最適化器は，コンパイラフレームワークの提供する中間表現の上で動作する．最適化器は，入力として中間表現を受け取り最適化を施した後，結果として中間表現を出力する．コンパイラフレームワークは，一旦生成した中間表現を複数の最適化器で処理し，最終的に得られた中間表現から，アセンブラコードを

生成する．

通常，中間表現は 1 つのコンパイラフレームワークに 1 つしか存在しないが，最適化の用途によって複数の中間表現を持つものもある．この場合，抽象構文木に近い中間表現から，よりハードウエア寄りの表現に順次変換していくことで最終的な結果を得る．

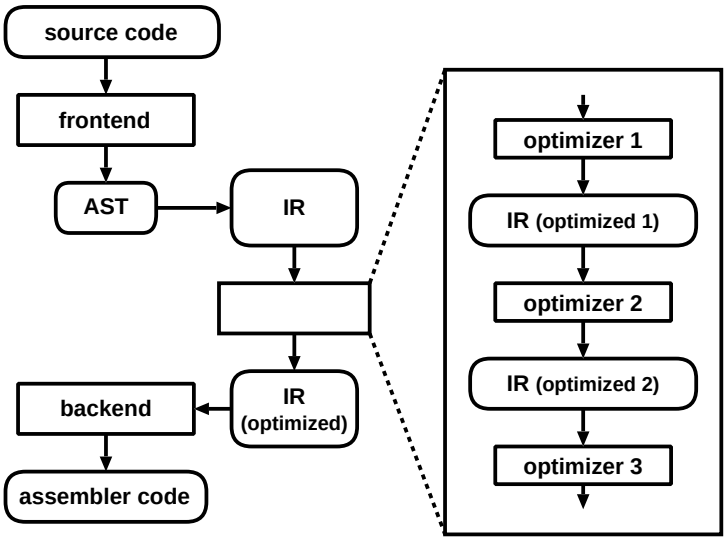

図 3.2: 中間表現 (IR) とコンパイラフレームワーク

コンパイラフレームワークの提供する中間表現は，必要に応じてユーザーにより拡張可能なものになっている場合が多い．これによって最適化器の開発者は，独自のデータ構造を中間表現に付加し，最適化に利用することができる．これは特にオブジェクト指向技術を用いて実現されたコンパイラフレームワークで用いられ，クラスの継承を利用した差分プログラミングによって既存の中間表現に独自の表現を加えることができる．

3.2.2 プログラミングインターフェース

コンパイラフレームワークの提供するプログラミングインターフェースは，最適化器が中間表現を含むコンパイラ内部のデータ構造にアクセスするための方法を規定する．中間表現の実装の詳細を最適化器に対して隠蔽し，より高い抽象度での操作を可能にする．

オブジェクト指向技術に基づいて実現されたコンパイラフレームワークでは，プログラミングインターフェースはクラスライブラリとして提供される．この場合，中間表現は，クラスライブラリからインスタンス化されたオブジェクトの集合である．これに対して，手続き指向言語によって実装されたコンパイラフレームワークのプログラミングインターフェースは，中間表現を表わす一連の構造体と関数の集合として定義される．

コンパイラフレームワークは，単一のプログラミング言語で実装されることがほとんどで，提供されるプログラミングインターフェースもコンパイラフレームワークの実装言

語と同じ言語に対してのみ，提供されていることが多い．複数の言語に対してプログラミングインターフェースを提供する場合，既に提供されているインターフェースを隠蔽するラッパークラスや，異なる言語間でのやりとりを可能にする変換系が必要になる．

さまざまな言語で共通な API を提供するため，IDL(Interface Definition Language) のような言語に中立な表現を用いて API を定義する方法もある．この場合，IDL の定義では言語の特徴を最大限に活かせない．DOM の Java における実装と Java に特化した JDOM の例に見られるように，一般的に容易ではない．

3.2.3 最適化器のためのランタイムシステム

コンパイラフレームワークの多くが，中間表現とプログラミングインターフェースを提供する単なるライブラリではなく，ソースコードの読み込み，中間表現の生成，任意に選択された最適化器の起動，コード生成までのコンパイルの一連の工程を制御する機能を持っている．これによって，コンパイラフレームワークは，最適化器にとっての実行環境と考えることができる．

コンパイラフレームワークから見た場合，個々の最適化器は，子プロセスとして起動可能な小さなプログラム，もしくは動的にロード可能なモジュールとして実現されている．コンパイラフレームワークは，最適化器を選択し，起動するためのインターフェースを提供する．また，複数の最適化器を使用する場合，ある最適化器の出力を一時的に保存し他の最適化器の入力として与える作業は，コンパイラフレームワークの仕事である．

3.3 既存のコンパイラフレームワーク

既存のコンパイラフレームワークについて調査を行ない，評価・検討を行なった．ここではその中でも SUIF，CFL，OpenJIT の 3 つのコンパイラフレームワークについて述べる．SUIF は代表的なコンパイラフレームワークであり，CFL は現在開発中のものである．OpenJIT は開放型コンパイラ技術の影響を強く受けている．

3.3.1 SUIF

SUIF(Stanford University Intermediate Format)[Gro] は，主として並列化における最適化手法の研究を目的として開発されたコンパイラフレームワークである．C++で実装されたオブジェクト指向のクラスライブラリを提供する．SUIF はすでに多くの研究の基盤として利用されている．SUIF の以下に挙げる特徴を持っている．

- **ストレージに保存可能な中間表現**

  SUIF の中間表現は，メモリ中だけでなくファイルなどのストレージに保存することが可能．これによって独立したプログラムとして実装された最適化器を，ファイル

を介して連携させて動作させることができる．SUIF の中間表現はバイナリファイルであり，その詳細は SUIF の提供する API によって隠蔽されている．

- **プログラムの変換をサポート**

  並列計算機のための最適化では，特定のプロセッサに対して効率的に実行可能なコードを生成するだけでなく，プログラムを複数の計算機資源上で並列して実行可能なように，分割することが重要である．SUIF はこれを可能にするために，中間表現からソースコードを再構成することが可能になっている．

- **拡張可能な中間表現**

  SUIF の提供するオブジェクト指向のフレームワークはクラスの継承を利用した差分プログラミングによって拡張が可能である．これは，Hoof と呼ばれるファイル内容に基づいて生成されたスケルトンコードを実装することによって行なわれる．Hoof は SUIF の中間表現を拡張するクラスについて記述したファイルで，SUIF 組み込みのオブジェクトも Hoof の記述を持っている．

SUIF には 1994 年に公開された SUIF1 と 1999 年に公開された SUIF2 とがあり，SUIF1 は最終バージョン 1.3.0.5 をもって開発が終了し，現在は SUIF2 の開発が行なわれている．SUIF に基づいて実装された最適化器，解析器等が数多く存在する．

3.3.2 CFL

CFL(Compiler Framework Library for Embedded Systems)[中西 01] はメモリ制約の厳しい組み込みシステムを対象に，特定のアプリケーションについて，そのコードサイズが最小となるような命令セットを持ったプロセッサとそのためのコードジェネレータの自動生成を目指す試みである．CFL は C，C++，Java などの高級言語をあつかうフレームワークを目指しているが，現在の実装はコード圧縮アセンブラのみである．

CFL はアプリケーションのコードに対してパターンマッチングを行ない，頻出する命令列を抽出する．組み込み機器向けのコンパイラは，頻出命令列を関数化してコードサイズの短縮を図るが，CFL では頻出命令列と等価な機能を持った新たな命令を定義し，その実装をプロセッサの拡張機能として VHDL のコードを生成する．同時に，拡張された命令に対応したコードジェネレータも生成し，これを用いてコード生成を行なう．

CFL も SUIF 同様，C++によって実装されたオブジェクト指向のフレームワークを提供している．現在の実装ではアセンブラ実装のためのクラスが整備されているのみである．CFL を用いてコード圧縮アセンブラを実装する場合，CFL の提供するクラスライブラリの中の命令を表わすクラスを拡張することによって，ベースとなるプロセッサ向けのアセンブラを実装する．この際，命令間の等価性を判断するためのメソッドや命令サイズを測るためのメソッドをオーバーライドする．これによって CFL に実装されたアルゴリ

ズムで頻出する命令を抽出し，ベースとなるプロセッサに対する拡張命令の VHDL コードを生成することができるようになる．

3.3.3 OpenJIT

OpenJIT[小林 00] は，Java 仮想機械のための JIT(Just-In-Time) コンパイラであり，Java プログラムの実行時に起動される点が，他のコンパイラフレームワークと大きく異なっている．しかし，大域的な解析は行なえないものの，中間表現と拡張可能なフレームワク，コードジェネレータを提供し，ユーザーによる拡張が可能という点ではコンパイラフレームワークと呼んでも問題はない．

OpenJIT は，Java による拡張が可能な Java 仮想機械のための JIT である．Java のクラスファイルに最適化に関するアノテーション情報を付加することで，OpenJIT はクラス毎に計算環境に特化した最適化器を選択し，実行することができる．一般的な JIT に実装し，全てのプログラムに対して行なうにはコストの高い最適化も必要なクラスに限定して行なうことができるため，効率的である．OpenJIT を用いることで，これまで可搬性のない非標準的な手法，もしくは特定の計算環境に強く依存したコーディングによって実現されていた処理を，オリジナルの単純さを損なわずに JIT によって透過的に行なうことが可能になる．

OpenJIT はフロントエンド部分，最適化器，バックエンド部分の 3 つの部分から構成される．フロントエンドはクラスからアノテーションを抽出し，使用する最適化器を決定する．OpenJIT における最適化器は Java のクラスであり，アノテーションにはメソッド毎に最適化を実装したクラスが記述されている．OpenJIT はアノテーションに記されたクラスをロードし，起動する．このとき最適化器には，Java のバイトコードまたはバイトコードをもとに生成された中間表現を与える．最適化の結果も同様にバイトコードまたは中間表現の形で与えられ，バックエンドはこれを機械語コードに変換する．変換されたコードは Java 仮想機械によって呼び出される．

3.4 XML に基づくコンパイラフレームワークの実現とその利点

3.4.1 既存のコンパイラフレームワークの問題点

SUIF をはじめとする既存のコンパイラフレームワークの多くが C++もしくは Java 言語によるオブジェクト指向フレームワークの形で提供されている．これは，既存のコンパイラフレームワークが研究開発を目的としてはいるものの，最終的な成果物である最適化器の実装，コンパイラ本体及び実装済みの最適化器の再利用を主としているためである．しかし，新たな最適化手法の実現は，プロトタイプおよび予備的な実装による実験，最適

化されたコードの評価の繰り返しである．このため，最終成果物である最適化器の実装にターゲットを絞った既存のコンパイラフレームワークには，生産性の観点から以下の問題がある．

- **実装効率の低い言語を強制**

  実験のための予備的な実装，実証実験のためのプロトタイプの実装においては，性能よりも正しく動作することが重要である．このため，プロトタイプの開発はデバッグやテストの容易な言語で実装することが効果的である．また，プロトタイプは，最終成果物ではないため，低コスト，短期間での開発が求められる．

  C++や Java は，オブジェクト指向フレームワークのもとで最適化器を実装するのに十分な記述能力を持っている．これらの言語は高速で高機能だが，上述の要求を満すものではない．既存のコンパイラフレームワークは，最適化器の実装言語をこのような実装効率の低い言語に限定している．これが，最適化器の生産性を低下させる要因の 1 つになっている．

- **ストレージ上での中間表現に可搬性が欠如**

  多くのコンパイラフレームワークは，中間表現をデータストレージに記録する手段を提供していない．このため，プログラムの解析や，最適化結果の評価のために，毎回，ソースコードの読み込みからコード生成までの，一連の工程を行なう必要があり，効率が悪い．

  また，SUIF は中間表現をファイルに保存し，再利用することが可能だが，そのためには SUIF の提供するフレームワーク上にアプリケーションを構築する必要がある．SUIF のフレームワークが実装言語を限定していることもあり，中間表現の 2 次的利用は，制限されたものにならざるをえない．

  ファイルに保存された SUIF の中間表現を独自に解析する方法も考えられるが，SUIF の中間表現は独自のバイナリファイルであり，ユーザーによる拡張を含んでいる場合もあるため，解析は困難である．

3.4.2 XML によるコンパイラフレームワーク実現の利点

本研究の目的は，前述した既存のコンパイラフレームワークの抱える問題点を解決し，最適化器の研究開発における生産性を向上させることにある．このために我々は，XML を用いたコンパイラフレームワークを提案する．これは，データスキーマを中心としたフレームワークであり，プログラミングインターフェース中心のコンパイラフレームワークの対極に位置するものである．

XML に基づくコンパイラフレームワークは，中間表現とコンパイラ内部のデータ構造を XML を用いてマークアップすることによって実現する．これには以下の利点がある．

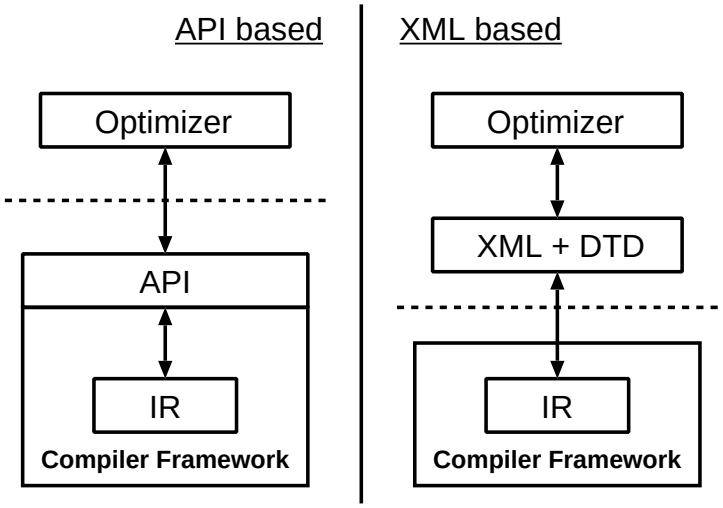

図 3.3: インターフェース中心とデータスキーマ中心の違い

1. **最適化器の実装言語を任意に選択可能**
   プログラミングインターフェースに沿って実装していた最適化器は，データスキーマに沿って XML を操作するプログラムに変化する．XML パーサは，さまざまな言語で実装されているため，最適化器は XML 文書を読み込むことのできる全ての言語で実装が可能である．
2. **中間表現の 2 次的利用の促進**
   中間表現の読み込みに，コンパイラフレームワーク自体を経由する必要がない．また，XML 文書自体は単純なテキストファイルなので，一旦 XML 文書になった中間表現は，再利用が容易である．これによって中間表現の 2 次的利用の幅が広がる．
3. **テスト，デバッグの効率化**
   最適化器の入力となる中間表現を，ファイルに保存することのできないコンパイラフレームワークでは，テストやデバッグのために，最適化器をソースコードの読み込み，最適化，コード生成の一連の工程の中に組み込む必要があるため，効率が悪かった．しかし，XML 文書として保存された中間表現を入力として再利用すれば，コンパイルの一連の工程を行なわずに，最適化器単体でテストやデバッグを行なうことができる．
4. **コンパイラフレームワーク自体の小型化**
   コンパイラフレームワークは，最適化器に対して大規模なオブジェクト指向のクラスライブラリを実装する必要がなく，最適化器とコンパイラフレームワークのやりとり中間表現と XML 文書間のマッピングだけで済む．これによってコンパイラフレームワークそのものを小型で可搬性の高いものにできる．
5. **XML 関連技術を導入可能**
   XML は，XML 勧告 1.0 以外にもさまざまな関連規格が存在する．XPath，XSLT な

どはその代表例である．XPath によって，XML 文書化された中間表現内を検索することが可能になる．XSLT を使えば，中間表現を HTML などの別の形式に変換することが可能である．また，単純な変換であれば XSLT のみで記述することも可能である．これ以外にも XML ネームスペースを使えば，XML で表現された中間表現の中に任意 DTD に基づく情報を，中間表現に影響を及ぼすことなく混在させることが可能である．

我々は，XML 関連規格の中でも特に XPath に注目し，最適化器の実装における XPath の利用について，生産性と再利用性の観点から評価を行なった．

3.4.3 XML によるコンパイラフレームワーク実現の欠点

XML をベースにコンパイラフレームワークを構築する際の欠点として，以下のことを挙げる．

1. **コンパイラのパフォーマンス低下**
   XML を中間表現として用いるコンパイラフレームワークを構築する場合，コンパイラと最適化器は独立したプログラムとして実現される．コンパイラと最適化器は，中間表現を XML 文書として交換する．最適化時には，これによるオーバーヘッドを生じる．1 つは，コンパイラと最適化器の双方が，XML 文書として表現された中間表現を入出力することによる．もう 1 つは，コンパイラとは独立したプログラムである最適化器の起動の起動によるもので，起動されるプログラムによって異なる．これは，インタープリタの起動時間などである．
2. **データ型の欠如**
   中間表現を XML で表現する際には，文字列だけでなく，数値も文字列として表現する必要がある．したがって，XML 化された中間表現を取り扱うプログラムは，要素に応じてその内容を適切に解釈しなければならない．例えば，整数型の場合，データ型に応じて表現可能な桁数や符号の有無が異なり，これを適切に処理しなければ，データの一部を失なうことになる．しかし，現行の DTD では，要素内容の文字列の解釈までは，制約を記述できない．このため，要素内容が適切なデータ型の値を表わすものかを検査するためには，妥当性の検証だけでは不十分で，コンパイラ側でのチェックが必要である．
3. オブジェクト指向の利点を失なう
   オブジェクト指向でコンパイラフレームワークを実装する際のメリットとして，中間表現の実現において隠蔽や継承といったオブジェクト指向の基本的な利点を活かせることが挙げられる．XML をベースにコンパイラフレームワークを実現した場合，中間表現における共通部分をスーパークラスに集約するなどのオブジェクト指向の利点を失しなうことになる．
4. 意味的なチェックが弱い
   XML をベースにしたコンパイラフレームワークでは，DTD を使って中間表現の構文

的なチェックが可能である．しかし，数値の有効範囲，シンボル名の妥当性などの意味的なチェックはDTDではできない．オブジェクト指向のフレームワークでは，これら意味的なチェックをクラス内に実装できる．

3.4.4 XMLに基づく最適化器の実装

XMLに基づくコンパイラフレームワークでは，最適化器はXML文書を入出力するプログラムとして実現される．XMLを読み込むことができれば，実装言語は任意のプログラミング言語を選択可能である．

中間表現に対して，検索と操作を繰り返すような実装は，DOMを用いることによって実現可能である．

最適化器の主な処理は，1．依存関係の取得(制御依存，データ依存)，2．依存関係を解析，3．中間表現の変更と出力，の3つである．XMLベースのコンパイラフレームワークでは，このうち1と3をXMLを用いて行なう．1は中間表現への変更を伴なわない検索処理が中心で，XPathが有効に活用できると考える．3で行なわれる中間表現への変更は，DOMツリーに上で行なうのが適当である．2に関しては，DOMツリーのみでは不十分であり，DOMツリーを含むデータ構造を最適化器が定義する必要がある．

また，DOM以外の独自のデータ表現に変換した上で最適化を行なうことも技術的には可能である．そのような実装を行なった場合でも，出力がXML文書になっていれば最適化器としては問題ない．単純な解析器であれば，XPath式とXPathプロセッサのみで実現することも可能である．SAXを用いた実装も有効である．

データスキーマを中心としたフレームワークでは，最適化器の実行時に中間表現の構文的な検査を行なうことができない．そこで，DTDに基づく妥当性の検証を，最適化器の出力したXML文書に対して行なうことで，中間表現の構文的な誤りを発見することができる．但し，これを実現するためには妥当性の検証で，構文的な誤りを発見することができるようなDTDをあらかじめ設計しておくことが必要である．

3.4.5 XMLに基づくコンパイラフレームワークの実現方法

コンパイラフレームワークの実現方法には，さまざまな方法が考えられるが，我々はその中でも次の2つの方法について，評価検討を行ない，実装を試みた．

1. **既存のコンパイラをベースにする方法**
   既存のコンパイラのフロントエンド部分とバックエンド部分を再利用する方法．最適化に関する部分に変更を加え，中間表現のXMLへの変換，最適化器の起動，最適化の結果の中間表現への再変換を実現する．
   既存のコンパイラを利用しているため，実用性の高いコンパイラフレームワークを構築できる反面，コンパイラの解析と修正，ベースとなるコンパイラの中間表現のマー

クアップが必要になる．ベースのコンパイラの構造によっては，実装上の問題が生ずる可能性がある．

我々では，この方法を具体的に検討し，GCC(GNU Compiler Collection) とその中間言語 RTL に対して，XML ベースの最適化器を組み込むための拡張を施すことを試みた．これについては，第 4 章で詳述する．

2. **新たにコンパイラを構築する方法**

既存のコンパイラの修正によって発生する技術的な問題を避けるため，新規にコンパイラとコンパイラフレームワークの構築を試みる．実用的なレベルのコンパイラを実装するにはコストと時間がかかるが，既存のコンパイラの内部構造による制約を受けず，理想的なかたちで，中間表現とコンパイラを設計することができる．

我々は，北陸先端科学技術大学院大学のソフトウエア環境特論 (i425) で出題された ANSI C 言語のサブセットである XC 言語を対象に，コンパイラおよびコンパイラフレームワークの実現を試みた．これについては，第 5 章で述べる．

第4章　既存のコンパイラをベースとするコンパイラフレームワークの構築

4.1　GCC と RTL

4.1.1　GCC の概要

GCC[Fre] は FSF の Richard Stallman らによって開発されたコンパイラである．GCC のバージョン 1.0 は 1987 年 5 月にリリースされた．GCC はその後，多くの改良が加えられ，現在は C 言語だけでなく，Objective-C，C++，Java，Fortran，Ada もサポートするコンパイラ群となった．また，GCC はバージョンアップを重ねるごとにサポートするアーキテクチャを増やし，Linux をはじめとする UNIX 上での標準的なコンパイラとなっている．GCC は，当初 GNU C Compiler と呼ばれていたが，複数の言語に対応したコンパイラ群であることから，現在では，GNU Compiler Collection と呼ばれている．

GCC の特徴は，複数のプログラミング言語と多数のアーキテクチャのサポートにある．(図 4.1) これを可能にしたのが，複数の言語の構文上の特性を包含する抽象構文木と RTL(Register Transfer Language) と呼ばれる中間表現になる．GCC は対応する各言語ごとにフロントエンドを持ち，それぞれのフロントエンドは，パースしたソースコードを抽象構文木に変換する．GCC の抽象構文木は，複数の言語の特徴を表現可能であり，フロントエンドによってパースされたプログラムは同じ 1 つの抽象構文木で表現される．抽象構文木になったプログラムは，RTL へと変換される．GCC は，RTL に対して最適化を施し，各アーキテクチャごとに用意されたバックエンドによってコード生成が行なわれる．

4.1.2　GCC における最適化器の実装

GCC における最適化器の実装は，RTL を操作するプログラムとして実現されている．GCC の最適化器は複数存在し，それらの組み合せによって全体の最適化が成される．最適化の施される一連の工程を最適化パスと呼び，1 つの工程をステップと呼ぶ．1 つの最適化器が使用されるのは，通常 1 回のみであるが，経験的に有効であることがわかっているものについては，複数回行なわれる．最適化器のエントリポイントは 1 つの関数である．最適化の中心は `toplev.c` 内の関数 `rest_of_compilation` で，`rest_of_compilation` は抽象構文木から生成された RTL を受け取り，GCC の起動時に与えられたオプションから，適用する最適化の種類を決定し，RTL に対して順次，適用していく．

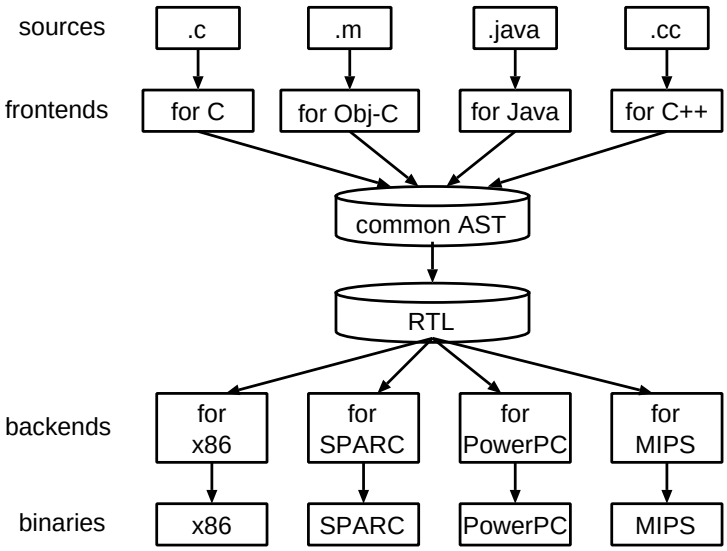

図 4.1: GCC のアーキテクチャ

最適化の施された RTL は，バックエンドに渡される．バックエンド部分はピープホールオプティマイザを兼ねており，RTL の特定のパターンに対してより効率の良いコードを生成する．RTL のパターンとそれに対応するアセンブラコードのテンプレートは，各アーキテクチャごとに用意されており，機械記述ファイル (`.md`) に記述されている．GCC の構築時にターゲットとなるホストに応じて `.md` ファイルが選択され，バックエンドの C 言語ソースコードが生成され，GCC に組み込まれる．

4.2 RTL

RTL(Register Transfer Language) は，GCC の最適化器で使用される中間言語である．RTL は，Language(言語) と名付けられているものの，その実体は GCC 内部のデータオブジェクトである．第 4 章では，GCC 内部のデータ構造について述べる前に，RTL のデバッグ用テキスト表現を用いて RTL の基本的な構成要素について解説し，次に GCC 内部で RTL を表現するデータ構造について述べる．

4.2.1 RTL テキストダンプ

GCC には RTL をテキストファイルとして出力するためのオプションが存在する．主にデバッグに用いられるこのオプションを使うことで GCC の最適化のステップごとに RTL を出力することができる．GCC のバージョン 2.95.3 で使用可能なデバッグ出力用オプションの中で代表的なものを図 4.2 に示す．

オプション	デバッグ出力をする場所
-dr	RTL 生成の直後
-dj	ジャンプ最適化のあと
-ds	共通部分式削除のあと
-dG	広域共通部分式削除のあと
-dL	ループ最適化のあと
-df	データフロー解析のあと
-dc	命令組み合わせのあと
-dN	レジスタ移動のあと
-dS	命令スケユーリングのあと
-dl	局所的レジスタ割り当てのあと
-dg	再ロードのあと
-dR	2 度目の命令スケジューリングのあと
-dJ	2 度目のジャンプ最適化のあと
-dd	遅延分岐スロットの最適化のあと

表 4.2: GCC の RTL 出力オプション

4.2.2 RTL のデータ型

RTL には整数，文字列，ベクトル，RTL 式の 4 つのデータオブジェクトが存在する．RTL 式がプログラムを構成し，整数，文字列，ベクトルは RTL 式のオペランドとして出現する．また，RTL 式自体も別の RTL 式のオペランドになる場合がある．

整数型

整数型と幅広整数型の 2 種類の整数型が存在する．十進表記のあとに十六進表記を'[' と']' で括って書く．浮動小数点数の表現はターゲットとなるアーキテクチャによって異なり，複数個の幅広整数型を使って表現する．

例)

```
10 [0x0a], 2 [0x02], -4 [0xfffffffc]
```

文字列型

関数名，グローバル変数名，ラベル名などアセンブラコード中で出現する名前を記述するために使用する．ソースコード中の文字列定数とは異なる．RTL における文字列は一部の例外を除きヌルポインタになることはない．機械記述の中では空の文字列はヌルキャ

ラクタへのポインタではなくヌルポインタとして扱われる．テキスト表現では，C 言語の文字列定数と同様に" で括って書く．

例)symbol_ref 式のオペランドとして

```
(symbol_ref "end")
```

ベクトル

複数の RTL 式を収めた順序付きの配列．ベクトルはスペースで区切られた複数の RTL 式を'[' と']' で括って表わす．一つも要素を持たないベクトルが生成されることはなく，代りにヌルポインタが使われる．

例) ベクトルをオペランドにとる parallel 式

```
(parallel [(set (reg:SI 1) (mem:SI (reg:SI 1)))
           (set (mem:SI (reg:SI 1)) (reg:SI 1))])
```

4.2.3 RTL 式

RTL 式の基本

RTL における式 (RTX と呼ぶ) は，RTL で表現されたプログラムを構成する単位として用いられるものと，他の RTX のオペランドとしてメモリ，レジスタ等のオブジェクトの参照や演算式の表現などを行なうために用いられるものの 2 種類がある．

RTX は式コード，フラグ，マシンモード，オペランドから成り，プログラム本体と最適化のための情報が全て同じ書式で記述される．(図 4.3)

```
(式コード [/フラグ 1/フラグ 2...][:マシンモード]␣
                    オペランド 1␣オペランド 2␣オペランド 3␣...)
```

図 4.3: RTX の書式

式コード

式コードは RTX の振る舞いとオペランドの数と種類，そしてオペランドの順序を決定する．オペランドは整数，文字列，ベクトル，RTX のいずれかをとる．RTL では式コードからオペランドの型を決定できる．

式クラス

式コードはその種類によっていくつかのクラスに分類される．これらのクラスは RTL クラスと呼ばれ，アルファベット 1 文字で表現される．式クラスが RTL に記述されることはない．(表 4.4)

式クラス	意味
o	レジスタやメモリなどの実際のオブジェクト
<	比較演算
1	単項演算
c	交換可能な 2 項演算
2	交換可能でない 2 項演算
b	ビットフィールド演算
3	3 項演算
i	命令全体
m	命令列にマッチするもの
x	その他全て

表 4.4: 式クラス一覧

フラグ

フラグは RTX に対して情報を補足するために用いられる．フラグはアルファベット一文字で表現され，同じフラグでも式コードによって異なる意味を持つ．

フラグの主な用途は以下のとおり．

- 参照するオブジェクトの種類を表わす．
- 使用するレジスタの用途
- RTX の属性
- ラベル参照の有無，参照の目的，参照の方法

RTX にフラグが設定されている場合には式コードの後に'/' に続けてフラグを表わすアルフェベットを書く．1 つの RTX に対して，複数のフラグがある場合にはこれを繰り返す．

マシンモード

データオブジェクトの大きさとその表現方法を表わす．同じ式コードでも異なるマシンモードを持つ場合があり，それは式の使用される文脈によって決まる．

マシンモードはフラグのあとに':' に続けて記述される．式コードによってはマシンモードは固定であり，その場合にはマシンモードの記述は不要となる．マシンモードは通常 XXmode と表記するが，RTL テキストダンプでは後の mode を省略して表記する．(表 4.5)

マシンモード	意味
QImode	1/4 精度整数 (1 バイト)
HImode	1/2 精度整数 (2 バイト)
PSImode	部分単精度整数 (4 バイト)
SImode	単精度整数 (4 バイト)
PDImode	部分倍精度整数 (8 バイト)
DImode	倍精度整数 (8 バイト)
TImode	4 倍精度整数 (16 バイト)
SFmode	単精度浮動小数点数 (4 バイト)
DFmode	倍精度浮動小数点数 (8 バイト)
XFmode	拡張精度浮動小数点数 (12 バイト)
TFmode	4 倍精度浮動小数点数 (16 バイト)
CCmode	比較演算の結果を表わす機種固有のビット群
BLKmode	ブロックモード．他のどのモードも適用できない．
VOIDmode	モードがない．もしくはモードを指定しない．

表 4.5: マシンモード

ここに挙げたマシンモードは最も一般的なものである．GCC のバージョンによっては，文書化されていないフラグが多数定義されている場合もある．

フォーマット文字列

式コードによって決まる．RTX のオペランドの数と順序，種類を表わす文字列である．オペランドの種類はフォーマット文字と呼ばれるアルファベット一文字で表記される．(表 4.6)

例)

subreg 式のフォーマット文字列は"ei"．つまりオペランドとして一つの RTL 式とひとつの整数値をとる．

この他にフォーマット文字'u' が存在するが，これは'e' に等しい．INSN へのポインタを表わすために特に用意されている．

フォーマット文字	意味
e	RTX
i	整数
w	幅広整数
s	文字列
E	RTL 式のベクトル

表 4.6: フォーマット文字

式クラスとフォーマット文字列

式クラスによってはそのクラスに属する全ての式が全て同じフォーマット文字列を持つ．(表 4.7)

式クラス	フォーマット文字列
1	"e"
<, c, 2	"ee"
b, 3	"eee"
i	"iuueiee"
o, m, x	フォーマットを仮定できない．

表 4.7: 式クラス対するフォーマット文字列

4.2.4 関数

RTL 生成

RTL の生成は構文解析のあとで関数ごとに行われる．そのため，以後の最適化パスも関数ごとに個別に行われる．同一のソースファイル内の複数の関数から参照される情報は，GCC 内部でポインタを通じて参照される．

INSN

RTL による関数の表現は INSN と呼ばれる RTX の双方向連結リストとして表現される．INSN は関数を構成するトップレベルのオブジェクトであり，構造化プログラミング言語の” 文” に相当する．他の RTX は INSN のオペランドとして出現する．

INSN は式クラス’i’ もしくは’x’ に属する式コードを持つ RTX で，式クラスが’i’ に属する式の場合，フォーマット文字列は’iuueiee’，’x’ に属する式も少なくともフォー

マット文字列の先頭の 3 文字が'iuu' になっている.

式コード	説明	式クラス
call_insn	関数呼び出し	i
jump_insn	アセンブララベルへのジャンプ	i
note	付加情報	x
insn	一般的な命令	i
code_label	アセンブララベル	x
barrier	命令ストリームが越えられない地点	x

表 4.8: INSN 一覧

INSN のフォーマット文字列の先頭の 3 つの文字の意味は以下のとおりである．テキストダンプを行なった場合の表記はポインタが参照する先の RTX ではなく参照する INSN の UID になる.

1. INSN の UID('i')
   関数の内部では他の INSN と区別するために各 INSN に与えられる一意な識別番号. UID は 0 ではなく 1 から始まる.
2. 1 つ前の INSN へのポインタ (2 番目の'u')
   1 つ前の INSN へのポインタ．先頭の INSN の場合は NULL. テキストダンプでは 0 と表記される.
3. 次の INSN へのポインタ (3 番目の'u')
   次の INSN へのポインタ．最後の INSN の場合は NULL. テキストダンプでは 0 と表記される.

第 2，第 3 オペランドによって INSN は双方向の連結リストを形成する．1 つの関数はそれを構成する INSN によって 1 つの双方向連結リストとして表わされる.

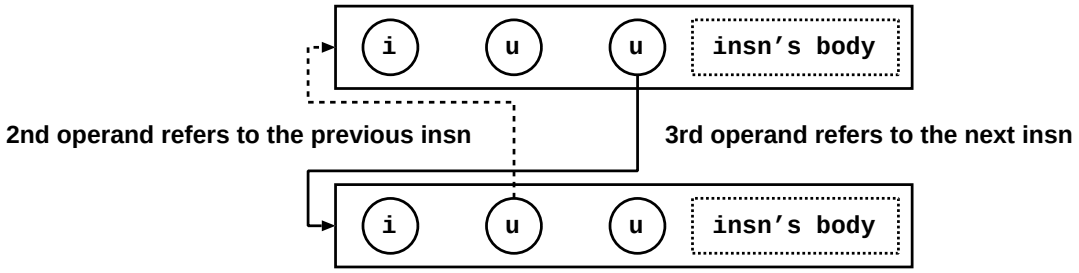

図 4.9: INSN の双方向連結リスト

4.3 GCC 内部での RTL

4.3.1 RTL を表現するデータ構造

GCC 内部における RTL の表現には rtx_def 構造体と rtunion_def 共用体が使用される．rtx_def 構造体は RTX そのものを表わし，rtunion_def 共用体は各オペランドを表わすために使われる．

rtx_def 構造体

以下に rtx_def 構造体のメンバのうち，式コード，マシンモード，オペランドに関係するものは表 4.10 に示すとおりである．フラグに関するメンバは，フラグ 1 つに対して 1 つずつ用意されているので，ここでは省略する．

メンバ	型	説明
code	rtx_code (16 ビット幅)	式コード
mode	machine_mode （8 ビット幅)	マシンモード
fld	rtunion[]	オペランドを格納する

表 4.10: rtx_def 構造体のメンバ

rtunion_def 共用体

rtunion_def 共用体は RTX のオペランドを表わす．各メンバは RTL のデータオブジェクトである整数，文字列，ベクトル，RTX に対応する．rtunion_def 共用体のどのメンバが実際に使用されているかを知るためには RTX の式コードごとに定められたフォーマット文字列を参照する必要がある．

表 4.11 に rtunion_def 共用体のメンバの一部を示す．rtunion_def 共用体には，これ以外にもメンバがあるが，GCC の最適化パスの一部で使用されるメンバであるためここでは省略する．

図 4.9 に示した関数とそれを構成する INSN による双方向連結リストは第 2，第 3 オペランドの型がフォーマット文字'u' で表わされているため，GCC の内部では rtunion_def 共用体の rtx メンバが用いられる．したがってこれらのオペランドはそれぞれ一つ前と一つ後の INSN へのポインタとなる．

メンバ	型	格納する rtl オブジェクト
rtwint	HOST_WIDE_INT	幅広整数
rtint	int	整数
rtstr	const char *	文字列
rtx	struct rtx_def *	RTX
rtvec	struct rtvec_def *	ベクトル

表 4.11: rtunion_def 共用体のメンバ

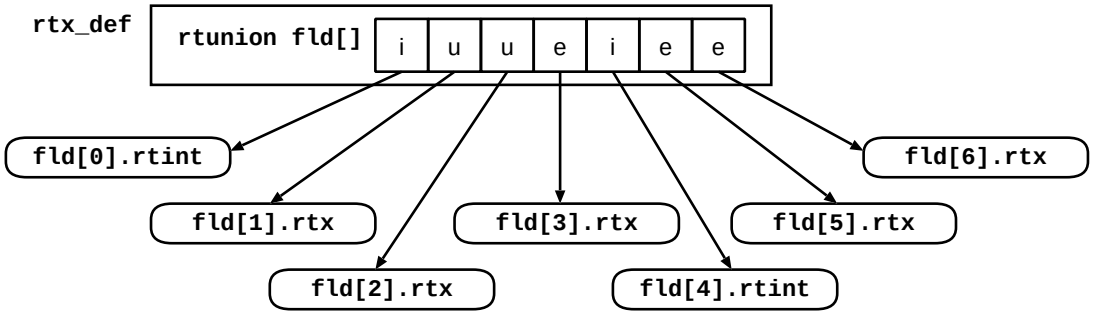

図 4.12: フォーマット文字列が iuueiee の場合に参照されるメンバ

4.3.2 RTX オペランドの読み出し

RTX に対してそのオペランドを読み出すためには，まず最初にその RTX の式コードを確認し，式コードに対応するフォーマット文字列を決定する．フォーマット文字列によってこれから参照しようとするオペランドの型を決定する．これによってはじめて参照することができる．

rtl.def

rtl.def には全ての RTX 式コード，式の属する式クラス，式のオペランドのフォーマット文字列が定義されている．また，テキストダンプ用の式コードの名前文字列も定義されている．

rtl.def で RTX を定義する書式は以下のとおり．GCC の他のソースコードでは rtl.def をインクルードし，DEF_RTL_EXPR を再定義して使用する．

```
DEF_RTL_EXPR(式コード,
             "名前",
             "フォーマット文字列",
             '式クラス')
```

マクロ

rtl.hにはRTXのオペランドにアクセスするためのさまざまなマクロが定義されている．これらのマクロの多くは基本となる4つのマクロRTL_CHECK1，RTL_CHECK2，RTL_CHECKC1，RTL_CHECKC2を用いて定義されている．この4つの基本となるマクロはフォーマット文字列に基づいてアクセス使用とするオペランドの型をチェックする．

RTL_CHECK1(RTX, N, C1)	RTXのN番目のオペランドを返す．そのときオペランドの型に対応するフォーマット文字がC1に等しく，かつ，Nがその式の式コードに対して定められているオペランドの個数に収まっていることをチェックする．
RTL_CHECK2(RTX, N, C1, C2)	RTL_CHECK1に同様だが，参照しようとするオペランドのフォーマットがC1もしくはC2であることをチェックする．
RTL_CHECKC1(RTX, N, C1)	Nに対してオペランドの個数のチェックをしない以外はRTL_CHECK1と同じ．
RTL_CHECKC2(RTX, N, C1, C2)	Nに対してオペランドの個数のチェックをしない以外はRTL_CHECK2と同じ．

表 4.13: RTXを操作する基本となる4つのマクロ

以下は，RTX_CHECKXを使った代表的なマクロ．この他にも多くのマクロが定義されている．

```
/* N番目のオペランドを幅広整数として返す．  */
#define XWINT(RTX, N) (RTL_CHECK1(RTX, N, 'w').rtwint)

/* N番目のオペランドを整数として返す．  */
#define XINT(RTX, N) (RTL_CHECK2(RTX, N, 'i', 'n').rtint)

/* N番目のオペランドを文字列として返す．  */
#define XSTR(RTX, N) (RTL_CHECK2(RTX, N, 's', 'S').rtstr)

/* N番目のオペランドをRTXとして返す．  */
#define XEXP(RTX, N) (RTL_CHECK2(RTX, N, 'e', 'u').rtx)

/* N番目のオペランドをベクトルとして返す．  */
#define XVEC(RTX, N) (RTL_CHECK2(RTX, N, 'E', 'V').rtvec)
```

4.4 RTL-XML：XMLによるRTLのマークアップ

GCCをベースにしたコンパイラフレームワーク実現の第一歩として，RTLをXMLで表現するためのマークアップ言語RTL-XMLを設計した．また，RTLの替りにRTL-XMLを出力するための拡張をGCCに施した上で，RTL-XMLを視覚化するツールを試作，評価を行なった．

4.4.1 RTL-XMLの設計

以下のことを目標にRTL-XMLの設計を行い，その実現方法を検討した上でDTDを設計した．

1. **XMLベースの最適化器で使用する中間表現として機能する．**
2. **RTXの式とオペランドの関係をXMLの要素と子要素の関係に対応付ける．**
3. **特定の最適化パスに依存した情報は持たない．**
4. **再帰的に深くなる構造を抑制する．**
5. **妥当性の検証を行なうことで，RTLの構文的な誤りを発見できる．**

特定の最適化パスに依存した情報の取り扱い

GCCの最適化器の中には，固有の情報を収めたデータオブジェクトをRTLのオペランドとして保存するものがある．これはフォーマット文字'x'のオペランドに，対象となるデータオブジェクトへのポインタを格納することで実現されている．これは`rtx_def`構造体と`rtunion_def`構造体で表現されるものとは異なるため，RTL-XMLではこれらの最適化器に固有の情報については，マークアップせずに無視する．

但し，特定の最適化パスに依存した情報を完全に取り去ると，RTL-XMLからRTLへの変換の際に，それらの情報への参照が失なわれるため，ポインタ値をダンプすることで暫定的に対処した．

再帰的に深くなる構造の抑制

生成時に要素数の決まるデータ構造は，ベクトルとして表現することができるが，最適化の過程で要素数が変化するデータの場合，再帰的なデータ構造として実現する必要がある．これは，XPathで任意の位置にある要素を参照するのに不向きである．これはXPathは,「深さ」を表現することができないためで，XPathで検索を行なうことを考えた場合は，シーケンシャルなデータ構造であることが望ましい．これを解決するために，再帰的な構造として実現された可変長配列をシーケンシャルなデータ構造に変換することで対処した．

再帰的な構造からシーケンシャルな構造への変換を図4.14に示す．この変換は，`expr_list`

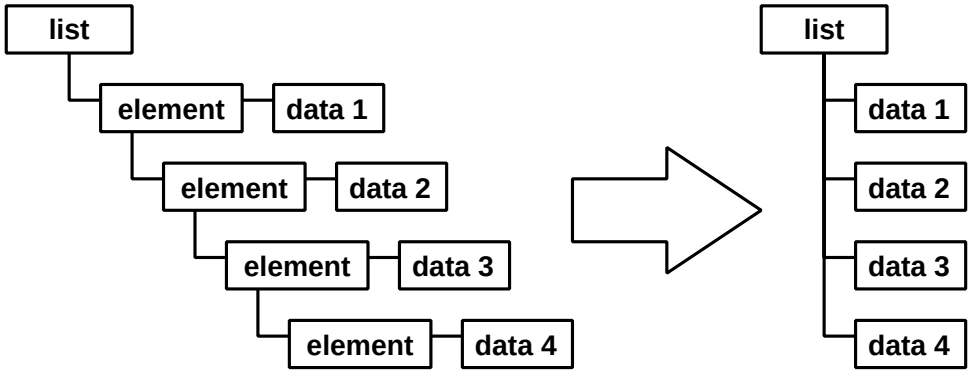

図 4.14: 再帰的な構造からシーケンシャルな構造に変換

または insn_list が INSN の第 5，第 6 オペランドとして出現した場合にのみ，行なわれる．これは，expr_list または insn_list が最も多く出現する再帰的なデータ構造だからである．

RTX のマークアップ

RTL-XML の設計に際して RTX のマークアップ方法に次の 2 つの方法について，比較検討を行なった．

案 1：式コードを属性として扱う すべての RTX を rtx 要素として単一の要素で表現し，式コードは rtx 要素の code 属性として記述する．この方法の利点は，DTD が小さく単純で，理解しやすい点にある．しかし，DTD を含む既存のスキーマ言語では，属性の値から内容モデルに制限を加えることができなため，妥当性の検証を行なっても，構文的な検査にはならない．また，XPath 式を使った検索では，属性の値を中心とした XPath 式になるため，木構造に沿った検索は記述量が増える．

案 2：式コードごとに要素を定義する 全ての式コードに対して，個別に要素を定義し，rtl.def にあるフォーマット文字列に従って，式コードごとに内容モデルを定義する．この方法は，DTD を大規模で，複雑化なものにするが，妥当性の検証を行なうだけで，オペランドの型と個数，順序などをある程度検査することができる．また，XPath 式も直感的な記述が可能になる場合が多い．

この 2 案について DTD を試作し，これに基づいて実際に，RTL を表現する XML 文書を作成し，評価を行なった．以下に例を示す．

マークアップする RTX

```
(mem/f:SI (plus:SI (reg/f:SI 54)
                   (const_int -4)))
```

第 1 案による XML

```
<rtx code="mem" frame_related="true" mode="SI">
    <rtx code="plus" mode="SI">
        <rtx code="reg" frame_related="true" mode="SI">54</rtx>
        <rtx code="const_int">-4</rtx>
    </rtx>
</rtx>
```

第 2 案による XML

```
<mem frame_related="true" mode="SI">
    <plus mode="SI">
        <reg frame_related="true" mode="SI">54</reg>
        <const_int>-4</const_int>
    </plus>
</mem>
```

上記の 2 つの方法を検討し，妥当性の検証の有効活用と XPath 式の使用という当初の目的に最も合致した後者を選択した．

データ型の取り扱い

我々がスキーマ言語として使用する DTD では，データ型についての制約を記述することができない．つまり，内容モデルに文字列が現われる場合に，その内容を整数値を表わすものや，日付けを表わすものなど，定まった書式にしたがって制限するための機能が DTD に存在しないということである．

しかし，DTD は簡潔で，十分に安定した実装が幅広いプラットフォームに対して提供されていることもあり，データ型に関する制約を記述できるが複雑な XML Schema などへの移行は行なわなかった．

RTL-XML では，データ型に制約を課すことはしていないが，RTL の基本データ型に対して要素を定義し，文字列で表現されたその値について，検査を行なう場合のマークとして使用できるようにしている．

4.4.2 RTL から RTL-XML への変換系の実現

GCC 内部の RTL を，RTL-XML 文書として抽出するための機能拡張を GCC に対して実装した．これは，GCC がデバッグ用に RTL をテキストファイルとして出力するための機能を拡張して行なった．

要素名	対応する RTL データ型
rtl_int	整数，幅広整数
rtl_string	文字列
rtl_vector	ベクトル
rtl_intptr	GCC 内部のポインタ (RTL-XML で表現できないデータに対して使う)
rtl_bitmap	ビットマップ

表 4.15: RTL-XML 要素と RTL 基本データ型の対応

GCC のデバッグ出力機能

GCC の持つデバッグ出力機能は，最適化の過程にある RTL をステップごとにテキストファイルとして出力する．デバッグ出力機能を利用するために，GCC に与えるオプションは，表 4.2 を参照.

デバッグ出力機能の実装は，主に GCC のソースの `toplev.c` と `print-rtl.c` で実装されている．`print-rtl.c` には，RTX をファイルに出力する関数 `print_rtx` と，関数全体を出力する `print_rtl` 関数が定義されている．`print_rtx` 関数は，与えられた RTX の式コード，フラグ，マシンモードを出力したあと，各オペランドについて自身を再帰的に呼び出す．`print_rtl` 関数は，関数の先頭の INSN へのポインタを受け取ると，関数を表現する双方向連結リストを先頭から順に `print_rtx` で表示していく．

デバッグ出力の開始と終了は，`toplev.c` の `rest_of_compilation`，`open_dump_file`，`close_dump_file` 関数が行う．`rest_of_compilation` 関数は，最適化のステップごとに最適化器のエントリポイントを呼び出す手前で，`open_dump_file` 関数を呼び出す．`open_dump_file` 関数は，デバッグ用の出力を書き込むファイルを作成する．最適化のステップ終了後に `rest_of_compilation` 関数は，`close_dump_file` 関数を呼び出す．`close_dump_file` 関数は，与えられた関数を使って，関数の RTL を表示しようとする．ここで `close_dump_file` 関数が受け取る関数は，通常，`print_rtl` 関数であるが，最適化のステップによってステップ固有の情報を出力するために，別の関数が渡される場合がある．

変換系の実現

RTL から RTL-XML への変換系の実現には，`print_rtl`，`print_rtx`，`close_dump_file` の 3 つの関数に対する変更で対応した．実装は，GCC のバージョン 3.2.1 を対象に行なった．

`print_rtx` 関数は，受け取った RTX を S 式ではなく，XML でマークアップして出力する．`print_rtl` 関数は，関数を表わす要素である `rtl_func_body` の開始タグと終了タグを出力する．

最適化のステップが出力する RTL-XML でマークアップされない情報は，除去する必

要がある．このために close_dump_file 関数は，受け取った表示用の関数が，print_rtl 以外のものであった場合に，それを print_rtl で置き換える．

RTL-XML の出力をする機能は，デバッグ出力機能と置き換えられているため，GCC の持つデバッグ出力用のオプションを利用して，RTL-XML 形式での出力が得られる．

4.4.3 RTL-XML の視覚化ツールの試作と評価

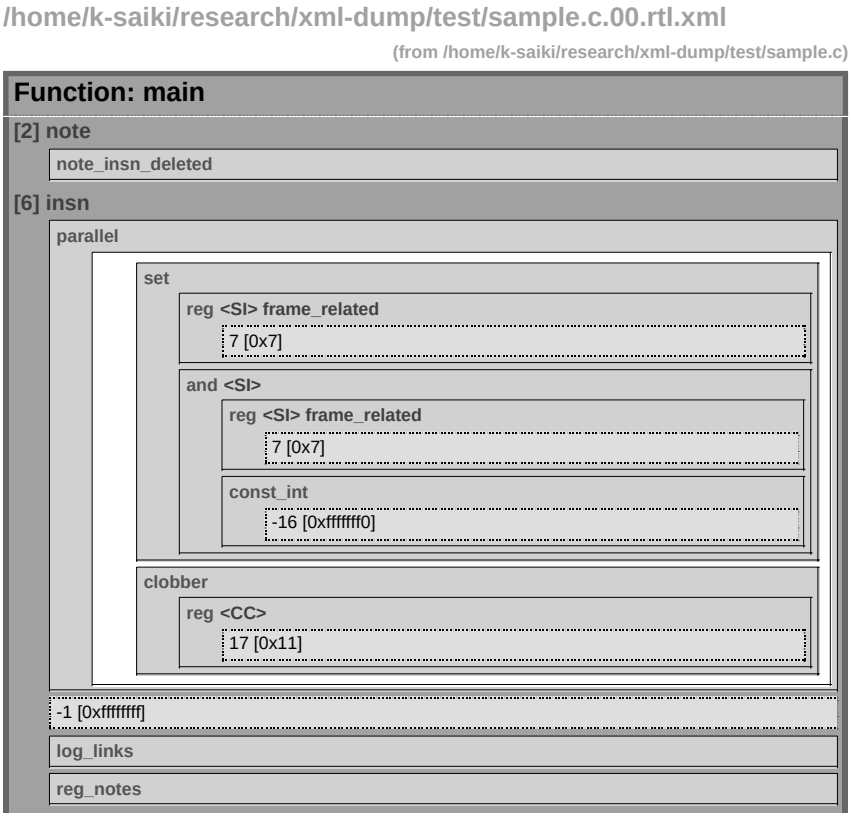

図 4.16: RTL-XML の視覚化ツールを使用した例

RTL-XML の視覚化ツールを試作した．視覚化ツールは Python 言語と CSS を用いて実装している．RTL から変換された RTL-XML 文書を PyXML[PyX] を利用して読み込み，要素を HTML に変換して出力する．結果は，ウェブブラウザを使って見ることができる．

図 4.16 は，RTL-XML の視覚化ツールを使用した例である．RTX は，実線の枠で示される．RTX のオペランドは枠の内側に別の枠で囲まれて表現される．一番左の'[' と']' で囲まれた数字は，INSN の UID である．白い背景の部分は，ベクトルを表わしている．破線で囲まれた部分は整数，文字列などの定数を表わす．

RTL-XML の視覚化ツールは，XML 化した中間表現の 2 次的利用の一例である．同様のツールは，他のプログラミング言語や XSLT などを使っても実現可能である．これは，

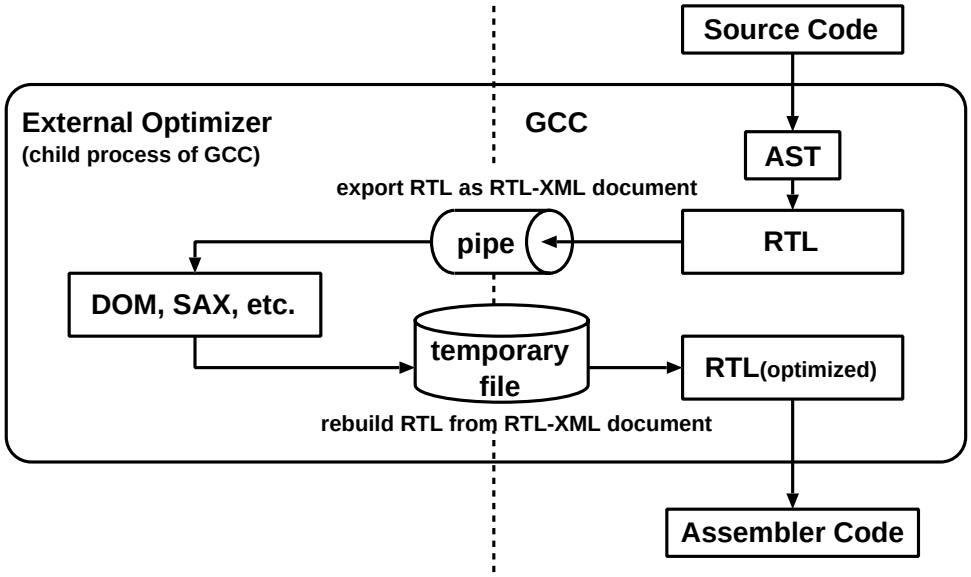

図 4.17: RTL-XML 処理系のアーキテクチャ

XML が特定の言語や環境に依存しない可搬性のあるものであることを示している．

また，最適化パスの複数のステップから RTL-XML を抽出し，特定の INSN やレジスタに注目して最適化の過程を観察するなど，一旦生成したものを繰り返し利用することも可能である．

4.5 RTL-XML 処理系の実現

4.5.1 システムのアーキテクチャ

図 4.17 に，システムのアーキテクチャを示す．修正の施された GCC は，RTL を RTL-XML に変換し，外部オプティマイザ (External Optimizer) と呼ばれるプログラムに渡す．外部オプティマイザは，ユーザーが実装した最適化器で，GCC の子プロセスとして起動される．外部オプティマイザは，RTL-XML 文書を読み込み，XML 文書上で最適化を行なう．GCC は外部オプティマイザによって最適化の施された RTL-XML 文書を受け取り，再び RTL に変換する．以降の処理は，通常の GCC と同様で，コード生成が行なわれる．

4.5.2 処理系の実装

RTL-XML 処理系は，4.4.2 で述べた変換系の実装をさらに拡張し，RTL から RTL-XML への変換以外に逆変換を実装し，実現する．実際に変更を加えられたのは，`toplev.c` 内の `open_dump_file` と `close_dump_file` 関数である．

外部オプティマイザの起動

外部オプティマイザは，GCC の子プロセスとして起動される．実際に起動を行なうのは，デバッグ出力用のファイルを作成する open_dump_file 内である．open_dump_file 関数は，デバッグ出力用のファイルを作成する代りに，子プロセスを作成し，GCC との間をパイプによって接続する．デバッグ出力用の関数は，このパイプの一端に書き込むことによって，外部オプティマイザに RTL-XML 文書を渡すことができる．

RTL-XML から RTL への変換

起動された外部オプティマイザに RTL-XML 文書を渡す処理は，close_dump_file 関数で行なわれる．通常の GCC では，close_dump_file 関数はデバッグ出力用のファイルに出力してリターンするが，修正した close_dump_file 関数は，open_dump_file 関数で作成したパイプに RTL-XML 文書を書き込み，一時ファイル経由して最適化された RTL-XML 文書を受け取る．

外部オプティマイザの実装において，XPath の使用や XML 文書の変更をする場合，外部オプティマイザは一旦 XML 文書全体を読み，DOM ツリーを構築する必要がある．このため，外部オプティマイザが処理を行なっている間，GCC は待機することになり，RTL-XML 文書の入力と出力は，それぞれが一括して行なわれる．単純なフィルタを除けば，外部オプティマイザの実装としては，これが一般的であると考える．一時ファイルではなく，select システムコールを使って入出力の多重化も可能だが，性能向上を期待できない上に，プログラムが多少複雑化することから，一時ファイルによる実装を採用した．

外部オプティマイザの出力した RTL-XML 文書の読み込みは，イベント駆動型の XML パーサである expat を用いて行なう．expat にイベントハンドラを設定し，一時ファイルからの入力を逐一与えていく．expat は，開始タグ，文字列，終了タグが出現する度に設定されているハンドラを呼び出す．expat に設定したハンドラの中では，スタックを使って，RTL-XML の要素を RTX に変換していく．トップレベルの RTX である INSN については，前後の INSN とのリスト構造も構築していく．

読み込んだ RTL-XML の要素から RTX を生成する際には，rtl.c の rtx_alloc を使用する．これは与えられた式コードに対して，同じ式コードを持つ RTX を格納するための必要なメモリ領域を確保して返す関数である．RTX を表現する要素を読み込んだ際に，rtx_alloc を用いてメモリ領域を確保し，子要素を読み込む毎にそのオペランドを再帰的に埋めていく．

外部オプティマイザの結果を全て読み込むと，最適化された関数を表わす完全な RTL が完成する．これを，外部オプティマイザ起動前の RTL と置き換えることで，最適化の結果を GCC に反映させる．

4.5.3 システムの評価

4.5.2 で行なった実装のテストを行なうために，外部オプティマイザとして，GCC から与えられた RTL-XML 文書をそのまま GCC に返すプログラム (tee コマンド) を使用して，テストを行なった．実装が期待通りの動作をすれば，通常の GCC と同じ動作をするはずである．

テストの結果，基本ブロックのみで構成されるプログラムに関しては，RTL と RTL-XML 間の変換と逆変換が問題なく動作することが確認できた．また，複数の関数を含むファイルのコンパイルも可能であった．しかし，制御構造を持つプログラムのコンパイルを行なうと，RTL の一部が欠落してしまう問題が見付かった．GCC のソースレベルでの解析とデバッグ作業を行なったが，最終的に外部オプティマイザを実装し，評価を行なうことが可能になるまでの実装には致らなかった．この原因について，我々がとったアプローチと GCC の内部構造の両面から，問題検討を行なった．

アプローチの問題点

RTL-XML 処理系の実装は，rtx_alloc 関数などの GCC の低レベルの関数を使用して実装されている．これは，RTL-XML 文書から RTL を直接合成しているためで，最適化パスのステップの間で RTL を別のオブジェクトに完全に置き換える．RTL の置き換えは，関数の先頭の INSN へのポインタを格納する first_insn と最後尾の INSN へのポインタを格納する last_insn の値を新たに合成された RTL へのポインタで置き換えることで行なわれる．

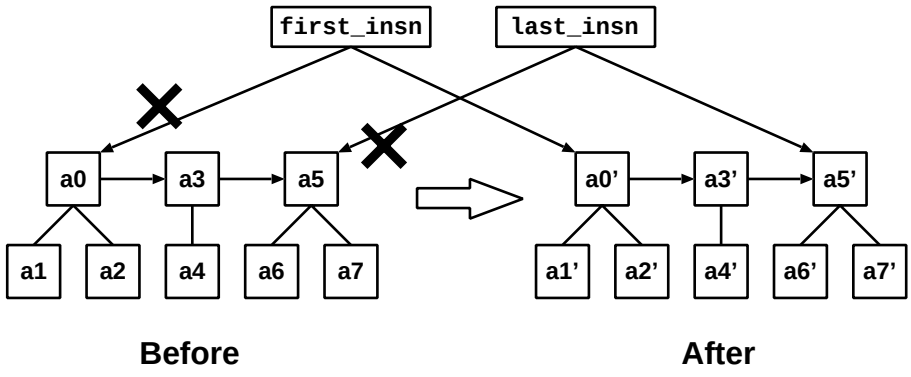

図 4.18: 最適化のステップの間で RTL が置き換わる

テストで外部オプティマイザとして使用した tee コマンドは，RTL-XML 文書に変更を加えないので，GCC 側で受け取った RTL-XML 文書から合成された RTL は，外部オプティマイザの起動前と後で全く同じ値と構造を持つ RTL になるはずである．一部の単純なプログラムについては，外部オプティマイザの起動前と後で同一の RTL が合成された

が，多くのプログラムではRTLの一部が欠落したり，GCCがエラーにより停止するなど不完全な結果に終った．

このような結果となった原因は，設計の段階でのGCCのソースコードの解析が不十分であり，かつRTLの低レベルな操作を中心に解析を行っていたために，RTLとGCCの他のデータ構造との関係の把握が不十分であったことにある．

より適切で標準的な方法でRTLの生成，操作，変更を行なうためには，GCC内の最適化器を解析する必要がある．また，RTLの操作においては，GCC内部のデータ構造とRTLの関係を最適化のステップ毎に明確にする必要がある．これをふまえて，より安全な方法でRTL-XML処理系を実装する方法とて，次の3つの方法を提案する．

1. RTL-XML文書からRTL全体を合成するのではなく，変更のあった部分に限定して，差分のRTXのみを生成するような方法であれば，GCCへの影響は少なくなると考える．
2. RTL-XMLの定義を拡張し，RTLに対する操作を記述できるようにする．これによってRTL全体を置き換えるような，大規模な変更をGCCに加える必要がなく，操作の記述をRTL操作のための関数に対応付けるだけで処理系を実現できる．これによって，中間表現をXMLで表現する利点が失なわれるが，最適化器とコンパイラフレームワークのインターフェースが言語に依存しないという利点がある．
3. GCC内部のデータ構造，制御の流れ，データの依存関係等を文書化する．我々の実装が不完全なものに終った最大の要因は，GCCの内部データ構造の把握が不十分だったことにある．これを明確に文書化することで，GCCの非標準的な拡張も容易になる．

GCCの問題点

RTL-XML処理系の実装において，GCCのソースコードと附属の文書から得られる情報は非常に有益であったが，実装において多くの場合で不十分なものであった．それは，1．RTLと他のデータ構造との依存関係が明確化されていない，2．文書化されていない情報が多過ぎる，の2点である．

1点目について，RTLから最適化パスの特定のステップに固有の情報をフォーマット文字'x'で表わされるフィールドに格納し，参照することが可能であるが，これとは逆にGCC内部からRTLで表現された関数の特定の個所を参照される場合がある．これは，RTLの生成時に行なわれるものと，最適化の過程で最適化器が個別に行なうものの2種類がある．

我々の行なったソースコードの解析において，発見することのできたRTLとGCCの内部データのうちの1つを図4.19に示す．RTLでレジスタを表現するRTXは，1つの`rtx_def`構造体のオブジェクトとして表現されるが，GCCでは全てのレジスタについて，同じレジスタを表わす`rtx_def`構造体オブジェクトを1つしか生成しない．最初に生成されたものをRTL上の他の場所からも参照し，再利用する．

我々の解析では，このような参照関係を全て把握して実装を行なうことができなかったため，完全な実装ができなかった．このような情報は本来，詳細に文書化されるべきであ

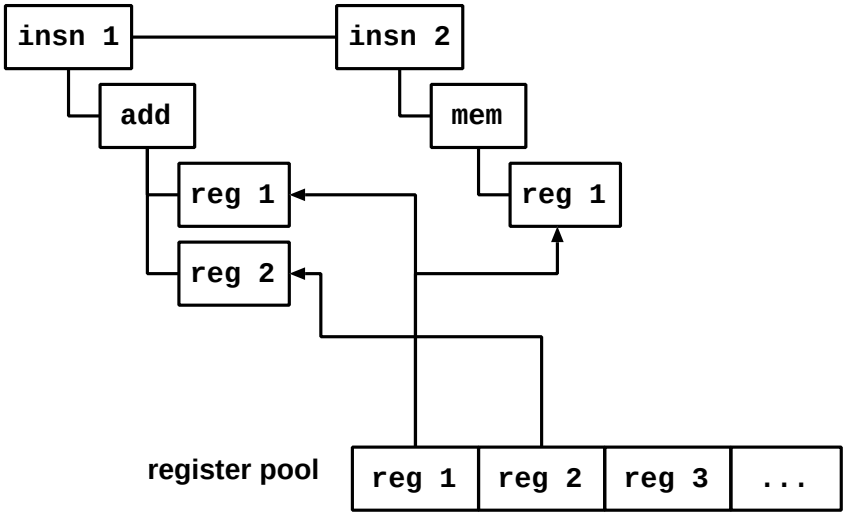

図 4.19: レジスタを表わす rtx_def 構造体オブジェクトの再利用

るが，GCC の内部実装に関して詳細に記述した文献は存在しない．

2 点目について，1 点目の問題も含めて現在の GCC には GCC の内部構造について記述した文書が不足している．例えば，表 4.5 に示したマシンモードは，文書化されている代表的なもののみで，これ以外のマシンモードも存在するが，それらに関する記述は，我々の知る限り存在しない．

GCC の開発体制の詳細を把握したわけではないが，文書化作業が十分に行なわれていないことは明らかである．特に GCC ではオープンソースソフトウエアの中でも規模の最も大きなプロジェクトの 1 つであり，多数の開発者が参加している．彼らの全員に同じ品質の文書の作成を要求することは，困難な問題である．

オープンソースソフトウエアは，誰もが自由にソフトウエアのソースコードを入手し，改変することが可能である．しかし，それが常に可能であることや，十分な品質とともに提供されていることを保証するものではない．これは附属の文書についても同じことがいえる．これはオープンソースソフトウエアの重要な側面の 1 つである．

一方で，十分な文書化がなされていれば，以後の保守開発の際の助けとなるだけでなく，新な開発者の参加や研究目的での利用の促進につながることもまた事実である．GCC の関連文書が 1 日もはやく整備されることを期待する．

4.6 RTL-XML についてのまとめ

RTL-XML の設計と処理系の実現についてまとめる．

- GCC の中間表現である RTL を表現するマークアップ言語 RTL-XML を設計した．

- GCC のデバッグ出力機能に修正を加え，RTL を RTL-XML 文書として出力できるようにした．
- 出力された RTL-XML 文書を利用する視覚化ツールの試作から，XML が中間表現の 2 次的利用に効果的であることを確認した．
- RTL-XML 文書を RTL に逆変換して GCC 内部に取り込むための拡張を，GCC に対して行なったが，完全に動作する実装は得られなかった．
- 不十分な実装に終った原因について考察した．

RTL-XML の実装における経験から，コンパイラ内部での中間表現の実現について，次の指針を得た．

- **中間表現はそれ自身で閉じているべきである．**最適化の特定のステップに依存する場合は，明確に文書化されるべきである．
- **中間表現に拡張性を持つべきである．**最適化のステップ固有の情報を中間表現に付加できるようにすべきである．
- **中間表現はストレージ上での表現をサポートすべきである．**中間表現の 2 次的利用が可能になる．

第5章　C言語サブセットに対するコンパイラフレームワークの構築

XML を用いたコンパイラフレームワークの構築と，その上での最適化器を実装が，実際に可能であることを示すことを目的として，C 言語サブセットの XC を対象にコンパイラフレームワークの構築を行なった.

5.1　XC 言語の概要

XC 言語は，北陸先端科学技術大学院大学のソフトウエア環境特論 (i425) でコンパイラ実装の課題として出題される．ANSI C 言語のサブセットである.

- **原始型 (primitive types)**

  int，char，void のみ．次のものは無し.

  - 型指定子 (type specifiers)：long，short，unsigned，signed
  - 型修飾子 (type qualifiers)：const，volatile
  - 記憶クラス (storage class)：static，extern，register，auto，typedef

- **派生型 (derived types)**

  関数とポインタのみ.

  - 配列はなし，ポインタで代用．`a[i]` ではなく `*(a + i)`.
  - 構造体，共用体，列挙型 (enum) は無し.

- **宣言**

  1 つの宣言 (declaration) で宣言できる変数は 1 つだけ.

- **制御文**

  `if`，`if-else`，`while`，`goto`，`return` のみ.

  `for`，`do-while`，`switch`，`break`，`continue` は無し.

- **制限された式**

  例：比較演算子は<のみ，<=や>は無し．

XC 言語の BNF 規則は 15 個と少ない．しかし，制御文や式に関しては，XC が持つもので C 言語の大部分を実現できる．データ型に関しては，派生型と型指定子，型修飾子，記憶クラスの一部が欠けているものの，配列はポインタにより実現可能であり，簡単なプログラム作成には十分な記述能力を持っている．また，ポインタ型をサポートすることから，ポインタに関する問題は C 言語と共通する．このことから，XC は研究に都合の良い言語ではなく，現実の問題を含んだ言語であると考える．

5.2 XC 処理系の実現

XML ベースのコンパイラフレームワークの実現を前提として，XC の処理系である XCC を実装した．XCC の実装は，北陸先端科学技術大学院大学，ソフトウエア環境特論 (i425) の課題の内容に沿ったものになっている． XCC は，SPARC v8 アーキテクチャ用のアセンブラコードを生成する．バイナリの生成自体は，GCC を用いて行なう．この XCC を，XML ベースのコンパイラフレームワークのベースとするために，中間表現の再設計とその実装を行なった．

再設計の目標は，次の 2 つである．第 4 章で示した中間表現の設計指針に基づいている．中間表現の拡張性とストレージへの保存は，XML 上で実現するため，ここでは考慮しない．

1. **抽象構文木ベースの中間表現**

   XCC は，主として教育目的での利用を前提としている．本来，コンパイラフレームワークを前提とするならば，独自の中間表現を定義すべきである．しかし，XCC の再設計においては，教育目的である点を重視し，講義中で解説された抽象構文木に沿った設計とした．
2. **中間表現がそれ自体で閉じている**

   XCC は，一切の最適化を行なわない．これは，XML ベースのコンパイラフレームワークの実現を前提としたものであり，最適化器はコンパイラフレームワーク上で実現する．最適化器を内部に持たないことにより，コンパイラ内部のデータ構造は単純化され，中間表現とコンパイラの依存関係を排除することが可能になる． XCC が中間表現に対して行なう処理は，アセンブラコードの生成を目的としたシンボルテーブルの構築と型解析のみである．

5.3 XCC-XML：XCC 中間表現のマークアップ

コンパイラフレームワーク実現の第一歩として，再設計した XCC の中間表現のためのマークアップ言語 XCC-XML[斉木] を設計した．RTL のような低レベルの中間表現をマークアップした RTL-XML の場合と異なり，XCC-XML の場合は抽象度の高い中間表現をマークアップする．XCC-XML 文書は最適化器の実装を容易にするため，抽象構文木以外の情報も含んでいる．XCC-XML のマークアップする情報を以下に示す．

- **中間表現本体**

  コンパイラフレームワークのために再設計した XCC の中間表現．RTL-XML の場合と同様，中間表現のノードの種類ごとに要素が定義され，妥当性の検証によって構文的な検査が可能である．

- **シンボル情報**

  XC はスコープとして，グローバルスコープと複文のブロック内スコープをサポートしている．XCC-XML では，最適化器の実装を容易にする目的で，構文解析時に構築したシンボル情報をマークアップして付加する．これには，後方参照を解決するための情報は含まれない．

  最適化器で追加したシンボルに関して，この情報は存在しないが，XCC-XML 文書から XCC の中間表現を再構築する際に，シンボル情報を再構築するので，最適化器がこの情報を追加する必要はない．

- **最適化器に依存，またはユーザー定義の情報**

  ユーザーや最適化器が，任意の情報を XCC-XML 文書に付加することができる．これは，XML の名前空間を利用して，任意のノードに対して情報を付加する方法と，文書の末尾の専用の領域に格納する方法の 2 種類の方法が可能である．

  例えば，XCC-XML は，1 つの XML 文書で 1 つの関数を表現するため，複数の関数にまたがる最適化や解析を行なう場合は，複数の XML 文書を組み合せる必要がある．必要な情報を 1 つの XML 文書に統合することで，最適化器の実装は容易になる．

シンボル情報と最適化器依存な情報は，XCC-XML から XCC の中間表現に変換する際に取り除かれる．

5.4 XCC-XML 処理系の実装

5.4.1 システムのアーキテクチャ

XCC-XML 処理系のアーキテクチャを図 5.1 に示す．基本的には，RTL-XML 処理系と同じである．

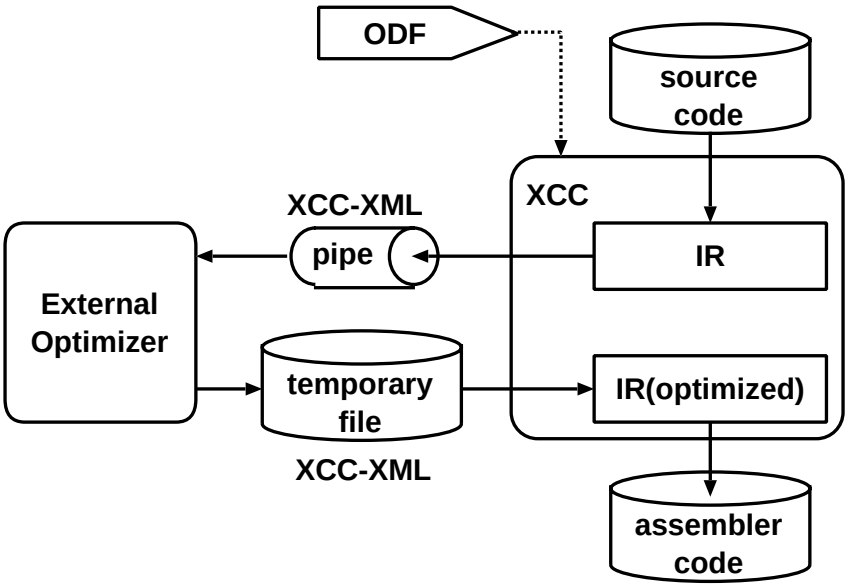

図 5.1: XCC-XML 処理系のアーキテクチャ

5.4.2 ODF(最適化器記述ファイル)

RTL-XML 処理系と XCC-XML 処理系の最も異なる点は，XCC-XML 処理系が ODF(最適化器記述ファイル) をソースコードと伴に受け取る点である．ODF はディレクトリ内にあるファイルの各関数について，起動する外部オプティマイザを記述する．ODF 内でデフォルトの外部オプティマイザを検索する．以下に ODF に可能な記述と XML 要素の関係を示す.

- `file_decl`

  ディレクトリ内のファイルについての記述．属性 `filename` にファイル名を記述する.

  - `func_decl`

    ファイル内の関数について記述する．属性 `funcname` に関数名を記述．要素の内容に，外部オプティマイザを起動するシェルのコマンドを記述する.

  - `file_default`

    ファイル内の関数に対して使用するデフォルトの外部オプティマイザを記述する.

- `dir_default`

  ディレクトリ内で使用するデフォルトの外部オプティマイザを記述する.

XCC-XML 処理系が ODF 内で外部オプティマイザ検索する順序を示す.

1. コンパイル中のファイルについて記述した file_decl 要素の子要素に，コンパイル中の関数に対応する func_decl 要素が存在する場合には，その記述を使用する．
2. 関数に対応する外部オプティマイザが見付からなければ，同じ file_decl の子要素の file_default 要素の記述を使用する．
3. file_default 要素がなければ，dir_default 要素の記述を使用する．

5.4.3 外部オプティマイザ

外部オプティマイザは，ODF の記述に従って，XCC-XML 処理系の子プロセスとして起動されるプログラムの総称である．外部オプティマイザが XCC-XML 処理系と XML 文書を受け渡しするために満すべき要件は以下のとおりである．

- シェルから起動可能である．
- 標準入力から XCC-XML 文書を読み込む．
- 最適化を施した結果を標準出力に書き出す．

以上の 3 つの要件を満しさえすれば，実装言語や実行環境の選択は自由である．これはインターフェースを中心としたコンパイラフレームワーク上での最適化器実装と比べて，制約が非常に少ないものになっている．

外部オプティマイザは，XCC-XML 文書を標準出力に書き込む前に，文書の妥当性を検証することが可能である．妥当性を検証することのできない環境を考慮し，妥当性の検証は必須要件ではない．

5.5 最適化器実装の予備実験

実装した XCC-XML 処理系の上で動作する XML ベースの最適化器を実装し，その有効性を実証するための予備的な実験を行なった．実装した最適化器は以下の 4 つである．

- 定数畳み込み．
- 関数のインライン展開
- ループアンローリング
- exit 関数のセマンティクスを考慮した解析

最適化器は Python 言語で実装した．XML パーサ，DOM 実装を提供する拡張モジュールとして PyXML を，XPath プロセッサを提供するモジュールとして 4Suite を使用した．

5.5.1 定数畳み込み

定数畳み込みは，コンパイル時に計算可能な値をあらかじめ計算しておくことで処理の演算のコストを下げる最適化手法である．1 本の XPath 式と 2 つの関数で実装されている．プログラムは 50 行の Python コードで実現されている．

アルゴリズムは次のようになる．

1. 子要素に AST_INTEGER_CONSTANT(整数定数) 要素を 2 つ持つ要素を XPath を使って検索する．
2. 検索結果から，AST_ADD(加算)，AST_SUB(減算)，AST_MUL(乗算)，AST_DIV(除算) について，計算を行なったのち，それらの要素を計算結果を表わす AST_INTEGER_CONSTANT に置き換える．
3. XPath 式の検索結果が，空になるまで 1，2 を繰り返す．

図 5.2 に，定数畳み込みの例を示す．左が簡略化した木で表現したもので，右が XCC-XML 文書の一部である．XCC-XML 文書の下線部分が変更の加わった部分である．

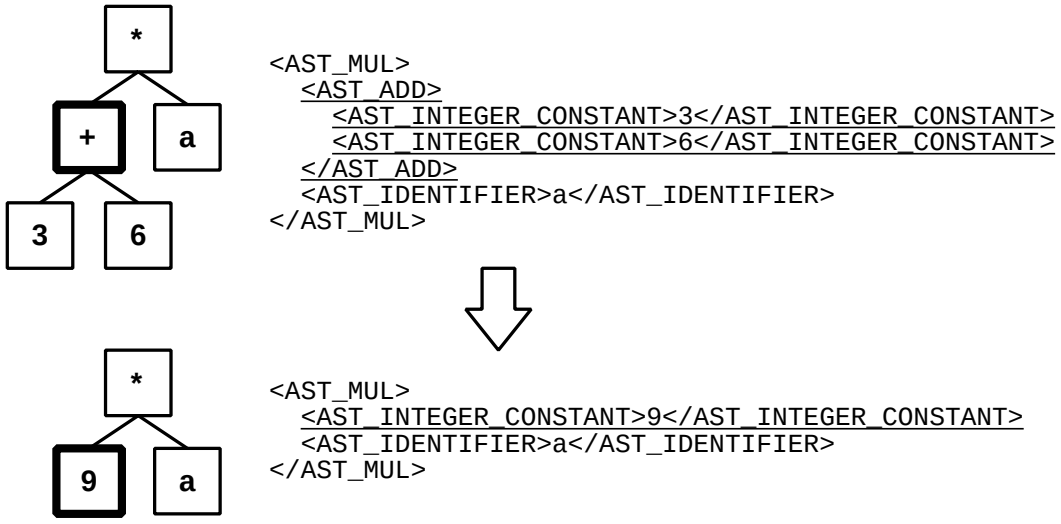

図 5.2: 定数畳み込みの例

5.5.2 関数インライン展開

関数のインライン展開は，頻繁に呼び出される小さな関数のコードを呼び出される位置に直接展開することで，関数呼び出しのオーバーヘッドを削減する最適化手法である．この最適化手法の，限定された実装を行なった．

我々の行なった実装は，次のようなコードに対して作用する．

```
variable = function();
```

左辺が変数で右辺が関数呼び出しであり，関数呼び出しは引数を取らないものとする．この実装は，必ずしも有効なものであるとは言えないが，中間表現に対する複雑な検索を記述する必要があり，XPath の利便性を確認するために，十分な内容であると考える．ここでは，関数 A を関数 B の中に展開する過程を例に説明する．

関数インライン展開を実現するためには，いくつかの前処理が必要である．まず．最初に関数 A と関数 B の XCC-XML 文書を 1 つの XML 文書にまとめる必要がある．これは，PyXML の DOM 実装が，2 つの異なる文書間でノードの付け替えをサポートしていないためで，他の DOM 実装では必ずしも必要ではない．関数 B の XCC-XML 文書の XCC_XML 要素の直下に INFO 要素を追加し，関数 A の FUNC_BODY 要素を付け加える．

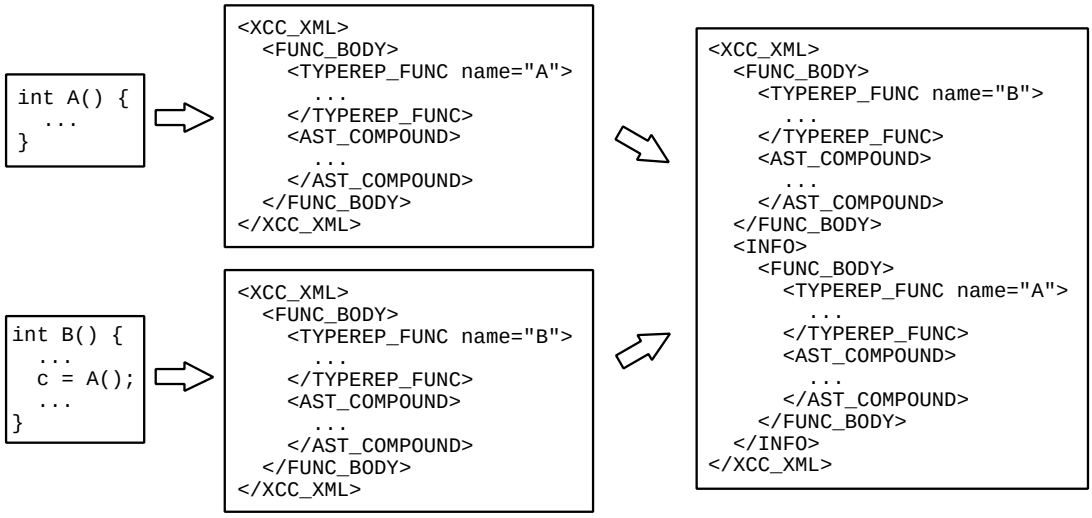

図 5.3: 2 つの XCC-XML 文書を統合

次に，関数 A を関数 B の中に展開した際に，双方のローカル変数名が衝突することを防ぐために，2 つの関数のローカル変数に関数名をプリフィックスをつける．これで前処理は，完了である．

前処理の完了した XCC-XML 文書に対して，インライン展開可能な個所を検索する XPath 式を実行する．XPath 式を評価した結果として得られた呼び出し個所に，関数 A のコードを展開する．関数 A のコードは，統合された XCC-XML 文書の INFO 要素から取得する (この処理も XPath による検索で実現される)．

関数 A のコードを関数 B のコードに展開する際に，関数 A のコードの return 文を，結果を受け取る変数への代入文に変更しておく．関数 A のコードを AST_COMPOUND 要素として，関数呼び出しを表現するノードと置き換える．

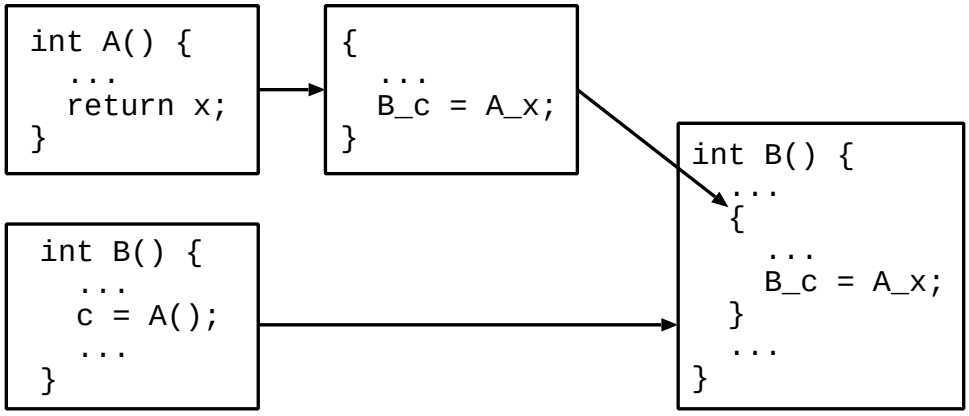

図 5.4: インライン展開の例：関数 A のコードを関数 B の中に展開する

5.5.3 ループアンローリング

ループアンローリングは，コンパイル時に実行される回数が決定できるループのループ回数を減らし，ループカウンタの計算を削除，分岐回数の削減で CPU のパイプラインを有効活用し，高速化を図る最適化手法である．

我々の実装したループアンローリングは，次のようなコードに対して作用する．多くのコンパイラで一般的に行なわれているループアンローリングは，ループ回数を減らすために，ループ数回分のコードを展開する．これに対して我々の実装したものは，実装を簡素化するために，ループを全て展開する．

```
int i;
...
i = 0;
while (i < 10) {
  ...
  i = i + 1;
}
```

XC には `for` 文がないため，`while` 文を対象とする．`while` 文は，ループカウンタとして整数型の変数を使用する．ループの繰り返し条件は，<演算子によるもので，左辺がループカウンタ，右辺繰り返し回数の上限である．ループカウンタは，ループの直前で `0` に初期化され，ループ内の最後の文で，インクリメントされる．

我々の実装したループアンローリングのアルゴリズムを以下に示す．

1. `while` 文の中から，条件が `variable <` 整数定数になっているものを見付ける．(検索に 1 つ，条件部分のチェックに 1 つの XPath 式で実現)
2. 見付かった `while` 文の 1 つ前の行にループカウンタの初期化があるかどうかをチェックする．(1 つの XPath 式で実現)

3. ループの最後にループカウンタのインクリメントがあるかどうかを確認する (1 つの XPath 式で実現)．エイリアスはないと仮定する．
4. ループ本体内に，ループカウンタへの代入がないことをチェック．(1 つの XPath 式で実現)
5. ループ本体から，ループカウンタのインクリメントを削除し，ループ本体をループ回数分だけ複製．このとき，ループカウンタを参照する部分は，その時点でのループカウンタの値に置き換えられる．

本来は，ループカウンタへのポインタ参照の有無などもチェックする必要があるが，ここでは簡便のために行わない．また，コードサイズとの関係を考えて，展開する回数の上限を設けるなど，実用的な実装にはさらなる改善が必要であるが，ここでは簡便のために省略する．

5.5.4 関数のセマンティクスを考慮した解析

GCC では，exit() システムコールの呼び出しに続いてコードが記述されていても，そのコードが到達不能であることを検出しない．これはコンパイラから見れば，システムコールも一般の関数と同じであり，プログラムの実行について特別な意味を持たないものとして取り扱うからである．これ以外にも abort システムコールや setjmp，longjmp なども言語のセマンティクスを越えてプログラムの実行フローを変化させる．

このような関数の暗黙のセマンティクスを考慮した解析の例として，exit システムコールによる到達不能コードの発生を検査する解析器を実装した．実装自体は非常に単純で，exit システムコールの呼び出しを検査し，後続するコードの記述がある場合に警告を出す．

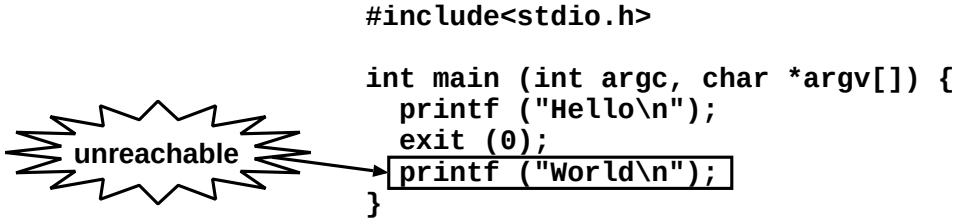

図 5.5: exit システムコールに起因する到達不能コードの検出

このようなシステムコールに関する単純な，アノテーションの集合を構築し，コンパイラフレームワークと伴に提供することによって，より詳細な解析と最適化を行なうことが可能になる．また，exit システムコールの呼び出しにつながる関数についてのアノテーションも，制御フロー解析に基づいて半自動で生成することが可能である．これをライブラリの構築時に生成しておいた，XCC-XML 文書に対して一括して行なうことで，効率良くアノテーションを収集することが可能である．

5.6 XCC-XML 処理系の評価

5.6.1 XCC-XML 処理系の性能測定

外部オプティマイザによるオーバヘッドの調べる目的で，実装した外部オプティマイザと，ODF ファイルの組み合わせによるコンパイル時間の変化を測定した．図 5.6 に結果を示す．

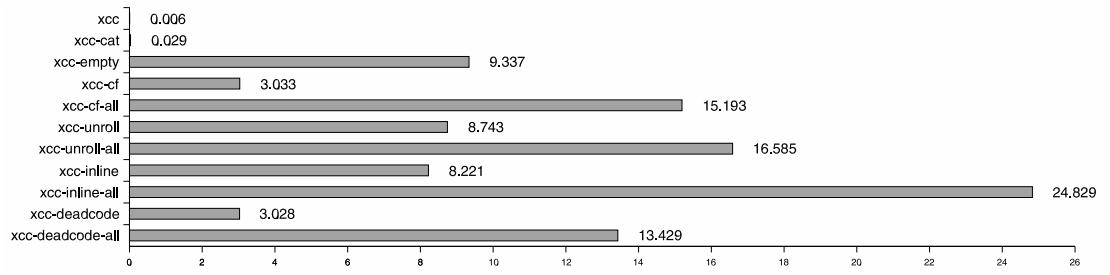

図 5.6: 外部オプティマイザを使用した場合のコンパイルに要した時間 (単位：秒)

各項目の意味は次のとおり．`xcc`，外部オプティマイザを使わずにコンパイルした場合．`xcc-cat`，外部オプティマイザとして `cat` コマンドを使用した場合，`cat` コマンドは一切の最適化を行なわないが，XCC-XML 処理系は XML 文書の入出力を行なう．`xcc-empty`，XML 文書を読み込み，DOM ツリーを構築，構築した DOM ツリーを出力する外部オプティマイザを使用した場合．`xcc-cf`，定数畳み込みを行なった場合．`xcc-unroll`，ループアンローリングを行なった場合．`xcc-inline`，関数のインライン展開を行なった場合．`xcc-deadcode`，到達不能コードの検知を行なった場合．項目名の末尾に`-all` がついているものは，定義されている関数全てに対して最適化を試みた場合であり，`-all` のついていないものは特定の関数に対して選択的に最適化を行なった場合の結果である．

測定は，同一のプログラムに対して 10 回の測定を行ない，その平均を結果とした．測定に使用した環境は，以下のとおりである．

- UltraSparc IIi 550Mhz(SPARC v9)
- 640MB メモリ
- Solaris 9

外部オプティマイザとして，Python インタープリタを起動するものは，起動しないもの (`xcc`，`xcc-cat`) に比べてコンパイルに要した時間が長くなっている．また，`xcc` と `xcc-cat` の差が，他の Python を外部オプティマイザに用いた場合とくらべて非常に小さいことから，XML 形式での入出力によるオーバヘッドは，全体の処理時間に対して問題になることはないと考える．

xcc-cat と xcc-empty の差は，Python インタープリタを起動し，DOM ツリーの構築を行なったか否かである．測定結果から，xcc-empty を使用した場合のコンパイルに要した時間の大部分が，Python インタープリタの起動と PyXML 関連モジュールの読み込み，XML 文書のパースと DOM ツリーの構築および出力に費されていることがわかる．テスト対象のプログラムが 5 つの関数を含んでおり，関数ごとに外部オプティマイザが起動されることから，関数 1 つあたりの外部オプティマイザによる XML 文書処理のオーバーヘッドは 1〜2 秒程度であると，概算できる．

各最適化器について，最適化が可能な関数に対して選択的に，最適化器を適用した場合の結果と，全ての関数に最適化器を適用した場合 (項目名の末尾に-all がついているもの) の 2 種類の測定を行なった．この結果，全ての最適化器において，選択的に起動した場合の結果が全体に一律に適用した場合に比べて短時間でコンパイルが終了した．これは，単純に起動される外部オプティマイザの数が異なることによるが，この結果が示すように，ODF ファイルを適切に記述することで，コンパイル時間の短縮が可能である．

5.6.2 実行性能の測定

次に，最適化されたコードの実行速度を測定した．XCC-XML によって，実際に有効な最適化器の実装ができることの確認が目的である．測定対象のプログラムは，整数定数同士の単純な演算を 100 万回繰り返すプログラムである．これを XCC-XML 上で実装した定数畳み込みを用いて最適化し，最適化なしの場合と実行時間を比較した．我々の用いた環境では，最適化をしないものが 0.19 秒，最適化したものが 0.06 秒であった．定数畳み込みに関しては，最適化が行なわれたことを測定結果から確認した．

この予備的な評価だけでは，XML を用いたコンパイラフレームワーク上で，最適化器の実装が可能であることを確認するには ，不十分である．これは，今後，より実用的なコンパイラフレームワークによって確認を行なう必要がある．(6.5 節で詳述)

本来，XCC-XML の実用性を示すためには，ある程度の複雑さを持った実用的なコードを対象に，最適化を行ない，その結果を示すべきである．しかし，XC は教育用の C 言語サブセットであるため，現時点では，複雑な最適化を適用できる大規模なプログラムは存在しない．加えて，XCC-XML が抽象構文木をベースにしているため，実装可能な最適化は，構文ベースの単純なものに限定されるのが現状である．

5.6.3 開発期間とコードサイズ

実装した 4 つの最適化器のコードサイズ (行数) と開発期間を表 5.7 に示す．これらの最適化器は，いずれも限定された機能しか提供しない．しかし，それをふまえても，非常に短期間で実装することができた．最適化器を短時間で実装できた要因としては，以下のことが挙げられる．

- XCC-XML 文書を保存し，再利用することで，デバッグ，テストをコンパイラを介

最適化器	行数	開発期間
定数畳み込み	49	1 日
ループアンローリング	91	1 日
関数インライン展開	135	2 日
exit システムコールに関する到達不能コードの検出	23	30 分

表 5.7: XCC-XML 評価用最適化器のコードサイズと開発期間

さずに行なうことができた.

- Python の対話型シェルの上での実験に基づて，実装を行なうことができた.
- XPath を使った記述によって，中間表現に対する検索を行なうコードを省くことができた.
- XPath の処理結果を，XPath プロセッサによって，あらかじめ確認することができた.

我々の提案するコンパイラフレームワークは，最終成果物である最適化器の実装ではなく，その前段階として行なわれる，予備的実験や，実証実験，テストやデバッグにおける生産性の向上を狙ったものである．最終成果物でなく，研究開発の過程を重視するため，処理速度よりも開発効率の良いものである必要がある．この点をふまえると，XCC-XML 処理系とその上の最適化器の速度は，高い開発効率に対する正当な対価であると考える．

XCC-XML 文書の再利用性，XPath の再利用性と利便性，高級言語による最適化器の開発については，6.3.1 節および 6.2.1 節でで議論する．

5.7 XCC-XML についてのまとめ

XC 言語を対象としたコンパイラフレームワークの実装と評価を行なった．XC 言語の処理系 `xcc` とその中間表現をベースに，中間表現を XML で表現するマークアップ言語，XCC-XML とその処理系を実装した．そして，実装した処理系の上で動作する最適化器を，Python 言語を用いて実装し，テストプログラムの最適化をとおして評価を行なった.

評価の結果，XCC-XML が，最適化器を短時間で効率良く実装するために，効果的であることを確認した．また，最適化されたコードの性能測定から，XCC-XML ベースの最適化器による，実行速度の向上を確認した.

しかし，我々の評価は，あくまでも C 言語のサブセット XC に対するもので，実際に開発に使用されている言語とは異なる．XML ベースのコンパイラフレームワークの有効性を実証するためには，より実用的なコンパイラフレームワークの構築と，より高度な最適化器の実装による評価が必要である．本研究における，我々の実装と評価は，そのための予備的なものであると考える.

第6章　議論

6.1　RTL-XMLの応用

6.1.1　CASEツールへの応用

RTL-XMLをCASEツールに応用する際の最も大きな利点として，RTLが特定のプログラミング言語に依存しないことが挙げられる．これまで，CASEツールは，各プログラミング言語ごとに実装されていた．RTL-XMLをターゲットにすることで，異なる複数のプログラミング言語で実装されたソフトウエアを，1つのツールで取り扱うことができる．また，RTLがプロセッサに依存した情報を，アセンブラより1つ上の抽象度で表現できるため，CASEツールの開発者は，アセンブラコードに近い情報をアーキテクチャに依存しない形で利用できる．

RTLでは，ソースコード中の全ての情報を保存できない．これはRTLが，ローカル変数をレジスタとメモリに置き換えること，RTLで表現されたプログラムがソースコード上の行と完全に対応しないことによる．中間表現であるRTLをXMLで表現したRTL-XMLも同様の問題を持ち，ソースコードレベルでの解析を中心としたCASEツールの開発には適さない．RTLがバックエンドと密接に関わることから，オブジェクトファイル間の関係など，プログラムバイナリに近いレベルでの処理を高い抽象度で扱うことに適していると考える．

6.1.2　教育目的での利用

通常，コンパイラの内部データ構造は，ユーザーから隠蔽されている．RTL-XMLを用いることで，実用的なコンパイラの代表格であるGCCの内部データ構造をユーザーが参照できる．ユーザーは，GCCの最適化のステップごとに，RTLをRTL-XMLとして保存することで，最適化の過程を見ることができる．これは，最適化技術を学ぶ学生のための生きた教材として，有効だと考える．

RTL-XMLは汎用的なXML文書である．XMLに対応するソフトウエアが幅広く入手可能である現在，これらのツールを使用してRTL-XML文書を閲覧できる．JavaやPython，XSLTを用いることで，RTL-XML文書を操作することも可能である．また，RTL-XML文書は，単純なテキストファイルであるため，テキストエディタのような一般的なツールでも操作できる．これにより学生は，RTL-XML文書の上で，実際に最適化の過程を体験

できる．

現在の大学で行なわれているコンパイラに関する講義では，フロントエンドとその背景にある理論を中心に行なわれている．最適化に関しては，基本的な理論のみで，実践を伴なわないのが通常である．しかし，フロントエンドの技術は，既に確立された技術であり，今後の研究の可能性は最適化にある．RTL-XML は，教官が例示に用いる表現の 1 つとして，または，学生が実践を学ぶための教材として，有用であると考える．

6.2 高級言語を用いた最適化器の開発

6.2.1 XPath の表現力

我々は，XC を対象にした予備的実験において，中間表現内の検索を XPath で実装した．汎用のプログラミング言語で実装すれば，再帰的にノードを探索し，その度に条件判定を行なう繁雑な記述になることが多い．しかし，XPath による実装では，抽象構文木をベースにした xcc 中間表現に対して，多くの場面で 1 本の XPath 式のみで記述できた．以下に XPath で中間表現を検索する例を示す．

単純な例 (1)

```
//*[count(AST_INTEGER_CONSTANT) = 2]
```

定数畳み込みの実装での例である．//は，/descendant-or-self::node()/の省略記法である．コンテキストノードとその子孫にあたる全てのノードが検索対象である．ルートノードから検索を開始し，子に 2 つの AST_INTEGER_CONSTANT 要素を持つ要素を発見する．我々の定数畳み込みの実装では，XPath 式を評価した結果，得られるノード集合の各要素について，AST_ADD(加算)，AST_SUB(減算)，AST_MUL(乗算)，AST_DIV(除算) である場合に，演算を行う．

単純な例 (2)

```
/XCC_XML/SYM_GLOBAL/*/@name
```

SYM_GLOBAL 要素は，xcc が構文解析時に認識したグローバルスコープの変数及び関数である．SYM_GLOBAL 要素の子要素には，各シンボルに対応する TYPEREP_PRIM(プリミティブ型の変数)，TYPEREP_FUNC(関数)，TYPEREP_PTR(ポインタ変数) の要素を記述する．この例では，SYM_GLOBAL の全て子要素の name 属性を列挙する．これは，全てのグローバルスコープにあるシンボル名を列挙することに等価である．

軸の特性を利用した例 (1)

```
/XCC_XML/FUNC_BODY/AST_COMPOUND/descendant::AST_STATEMENT_LIST
        [AST_STATEMENT/AST_CALL[AST_IDENTIFIER/text() = "function"]]
```

関数内での関数”function” の呼び出しを全て列挙する．この例では，descendant 軸によって，AST_COMPOUND 要素の子孫が全て検索対象になる．関数内のコードは，AST_STATEMENT_LIST 要素によって再帰的に表現されるため，木の任意の深さにある要素を取得するために descendant 軸が適している．

軸の特性を利用した例 (2)

```
ancestor::AST_COMPOUND/SYM_LOCAL/*[@name = "symbol"]
ancestor::AST_COMPOUND/SYM_LOCAL/*
```

１つ目は，コンテキストノードから参照可能なスコープにあるシンボル名” symbol” のシンボルを現わす要素を取得する．２つ目は，コンテキストノードから参照可能な全てのシンボルに関する情報を内側のスコープから順に列挙する．ancestor 軸によって列挙される要素とその順序を図 6.1 に示す．

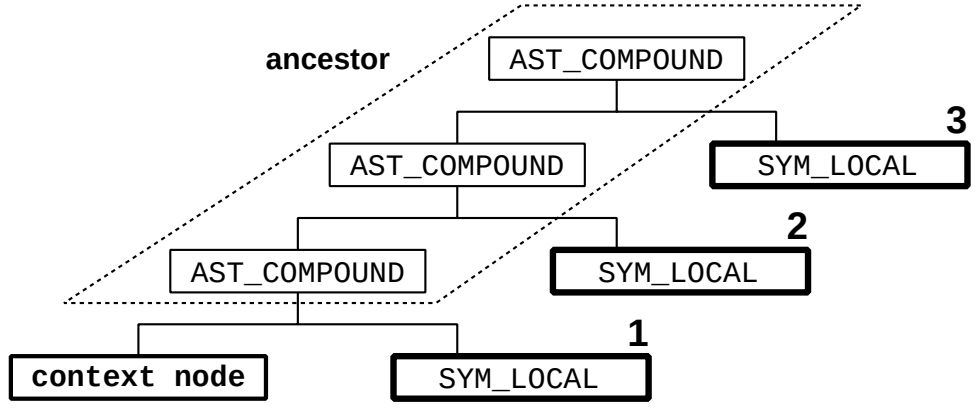

図 6.1: ancestor 軸によって列挙される XML 文書中の要素と列挙される順序

ancestor 軸で列挙される要素は，親，親の親，その親の順である．XPath で要素を列挙する際，通常は要素が XML 文書中に出現した順序で列挙される．この順序を文書順と呼ぶ．ancestor 軸を含むいくつかの軸では，これと逆で，文書順に並べた場合にコンテキストノードから近い順に列挙する．この例では，ancestor 軸を使った結果として，コンテキストノードを含むスコープから外側のスコープに向って順に SYM_LOCAL 要素を列挙する．このため，外側のスコープで宣言されたシンボルを内側のスコープで再宣言した場合でも，正確に内側で宣言されたシンボルの情報を取得できる．

軸の特性を利用した例 (3)

```
descendant-or-self::AST_STATEMENT_LIST[last()]
```

複文に含まれる最初の文を取得する．コンテキストノードは，AST_COMPOUND または AST_STATEMENT_LIST である．descendant-or-self 軸は，ancestor 軸と同様に，文書順とは逆の順序で要素を列挙する．図 6.2 に xcc の中間表現での連続した文の表現方法を示す．

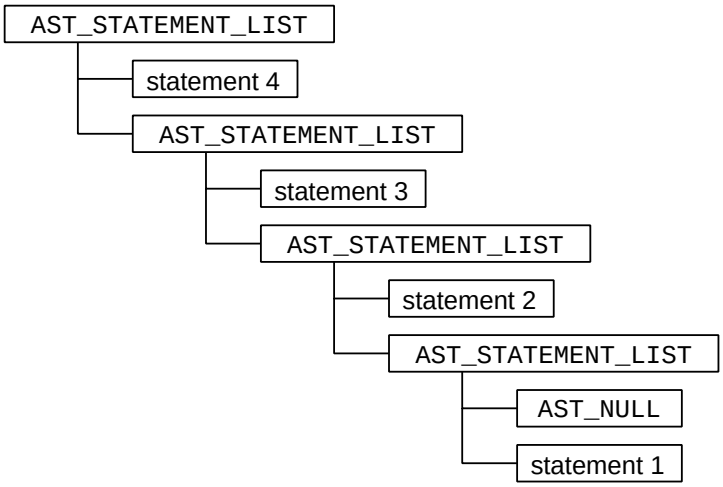

図 6.2: xcc の中間表現における連続した文の表現

最初に実行されるべき statement 1 が，木の最も深い位置にあり，後の順番の文ほど浅い位置にある．複文で最初に実行される文を取得するには，文書順で最も最後にある AST_STATEMENT_LIST 要素を取得すればよい．この例では，検索対象の部分木の中で最後に出現する AST_STATEMENT_LIST 要素を取得するために，ロケーションパスの述語部に last 組み込み関数を用いている．

XPath を有効に機能させる DTD の設計

これらの例は，XC の処理系である xcc の中間表現に対して，有効性を確認したものである．例に示した XPath 式が有効に機能するのは，xcc の中間表現が抽象構文木ベースであり，XPath が，木のノードと子ノードの関係に対するパターンの記述に適しているからだと考える．また，よりアセンブラコードに近いフラットな構造を持つ中間表現では，XPath は有効に機能しないと考える．図 6.3 に例を示す．

図 6.3 中央の図は，中間表現をフラットな構造で表現したものである．コードは全てルート要素の下に並び，ブロックスコープの開始と終了は，それぞれ block_begin 要素と block_end 要素で表現される．これに対して左の図は，ブロックスコープを要素の親子

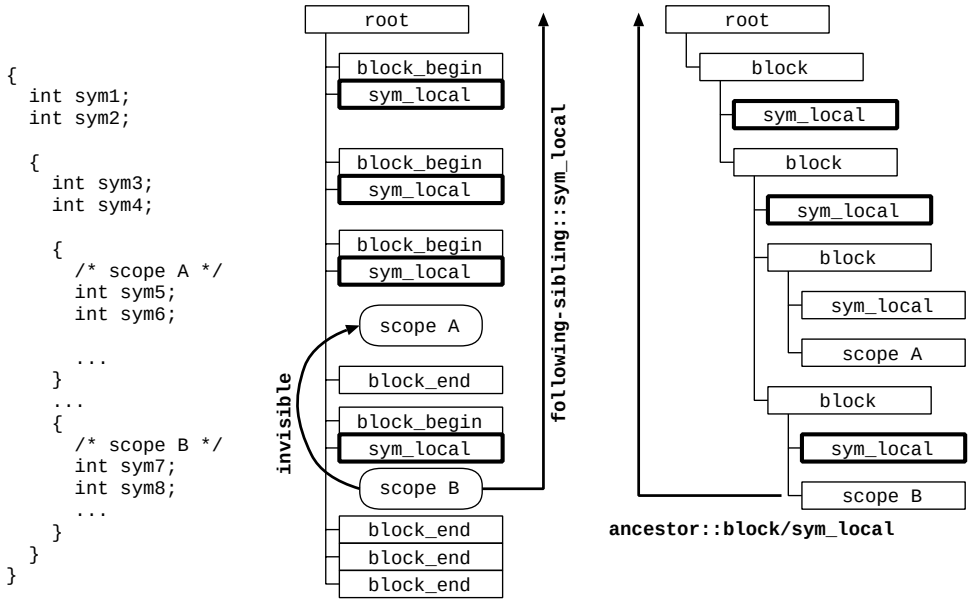

図 6.3: フラットな構造では，XPath が有効に機能しない

関係に対応付けたものである．1 つのスコープは，1 つの block 要素に対応し，スコープの入れ子は，block 要素の入れ子として表現される．

ここで，図中のスコープ B で可視であるシンボルを全て列挙することを考えた場合，図中央の記述例で，preceding-sibling 軸を使って次のようなローケーションパスを使って検索すると，

```
preceding-sibling::sym_local
```

となる．この結果，本来，不可視であるはずのスコープ A の sym_local 要素も結果のノード集合として得られ，正しい結果にならない．これを正確な結果となるように記述することは，単一の XPath 式ではできない．これに対して，図左の記述例では，ancestor 軸を使って次のようなローケションパスになる．

```
ancestor::block/sym_local
```

この場合，ancestor 軸によって，スコープ B から直接たどれる親要素のみを対象とするため，スコープ B からは，不可視のスコープ A の sym_local 要素は除外される．ancestor 軸を使った検索が，入れ子になったスコープの処理に対して有効に機能した．

図 6.3 の例では，木構造を用いてスコープを表現した方が，記述が直感的で，XPath も有効に使える．逆にフラットな構造は，XPath の適用が難しい．これは，XPath が主として，要素の親子関係に基づく検索に適しており，兄弟関係にある要素間の関係を十分に記述できないことによると考える．

このように，XML や XPath の利点を生かせない中間表現の設計も可能である．XML を有効に機能させるためには，DTD 設計のベースになる中間表現の設計が重要である．xcc の中間表現の問題点とその改善については，6.5.1 節で議論する．

6.2.2 最適化器開発におけるターンアラウンドの短縮

我々は，XCC-XML を用いた最適化器の実装予備実験において，最適化器の実装言語として，Python 言語を使用した．Python は，オブジェクト指向のスクリプト言語である．XPath は，Python 上から XPath プロセッサをとおして使用した．予備実験に際して，テスト，デバッグ用の XCC-XML 文書の生成を行なった．これには，tee コマンドを外部オプティマイザとして使用し，XCC-XML 処理系から受け取った XML 文書を，そのまま返すのと同時にファイル保存した．

実装では，まず，Python シェルおよびコマンドライン上で動作する XPath 処理系である 4xpath を用いて，XCC-XML 文書上でさまざまな実験を行なった．Python シェル上では，Python スクリプトと同様の記述を対話的に行なうことが可能である．4xpath は，1 つの XML 文書と 1 つの XPath 式を引数に取り，XPath 式の評価により得られたノードの集合を返す．Python シェル上で，一旦，XML 文書を読み込めば，インタープリタの再起動なしで，XML 文書への操作と結果の確認を繰り返すことができる．4xpath を使えば，XPath を評価するためのスクリプトを記述せずに XPath 式の評価結果を確認できる．Python シェルの実行例を以下に示す．

```
Python 2.2.2 (#1, Dec 11 2002, 04:58:42)
[GCC 2.95.3 20010315 (release)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from xml.dom.ext import PrettyPrint
>>> from xml.dom.ext.reader import Sax2
>>> reader = Sax2.Reader()
>>> xmlsource = open('main.xml', 'r')
>>> dom = reader.fromStream(xmlsource)
>>> print dom.childNodes[0].nodeName
XCC_XML
```

次に，4xpath の実行例を示す．

```
$ 4xpath main.xml '/XCC_XML/FUNC_BODY//SYM_LOCAL/*/@name'
Node Set:
<cAttr at 0x8428ef4: name u'name', value u'size'>
<cAttr at 0x84290c4: name u'name', value u'p'>
```

```
<cAttr at 0x8429404: name u'name', value u'i'>
$
```

C や Java などのコンパイラ型言語を用いた既存コンパイラフレームワークでは，実験用の最適化器のコンパイルとコンパイラフレームワークへのリンクが必要である．しかし，Python のようなスクリプト言語ではコンパイルとリンクを行なう必要がない．このため，コーディングと実行，デバッグを短かいサイクルで行なえる．また，Python の対話型シェルやコマンドラインで動作する 4xpath のようなツールにより，小規模な実験では，コーディングの一部も省略することができる．

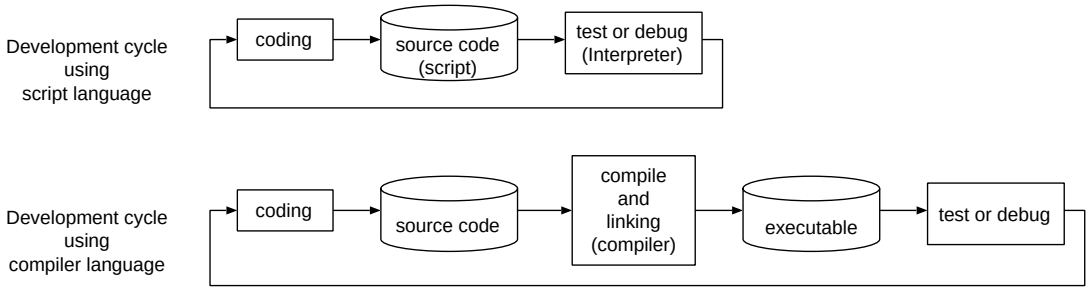

図 6.4: Python のようなスクリプト言語では，コンパイルとリンクを行なう必要がなく，開発のサイクルが短かい

予備的な実験に基づいて，最適化器を実装した．実装上の問題点をあらかじめ実験によって確認しておくことで，スムーズに実装ができる．実装上の問題が発生した場合には，デバッグだけでなく，再び Python シェルを用いた予備実験に戻ることもできる．

こうして実装した最適化器は，それ自体が単体で動作可能なスクリプトである．あらかじめ生成した XCC-XML 文書を入力として，デバッグとテストが可能である．XCC-XML 文書は，単純なテキストファイルなので，生成しておいた XCC-XML 文書をテキストエディタなどで編集し，さまざまなテストケースをつくることもできる．また，最適化器による最適化の結果もまた，XCC-XML 文書として得られるため，ウェブブラウザや XML ビューアなどを使って視覚的に結果を確認できる．

SUIF を除く既存のコンパイラフレームワークでは，中間表現をファイルとして保存できないため，最適化器のデバッグとテストのために，コンパイラフレームワークへの最適化器の組み込みが必要である．この場合，デバッグの際に，コンパイラフレームワークか最適化器のどちらがバグの発生元かを特定する必要がある．

XCC-XML では，コンパイラフレームワークでの XML の導入により，高級言語を使った最適化器の実装が可能になった．我々は，予備実験での実装から，高級言語による最適化器の実装が，開発のターンアラウンドの短縮に貢献したと考える．

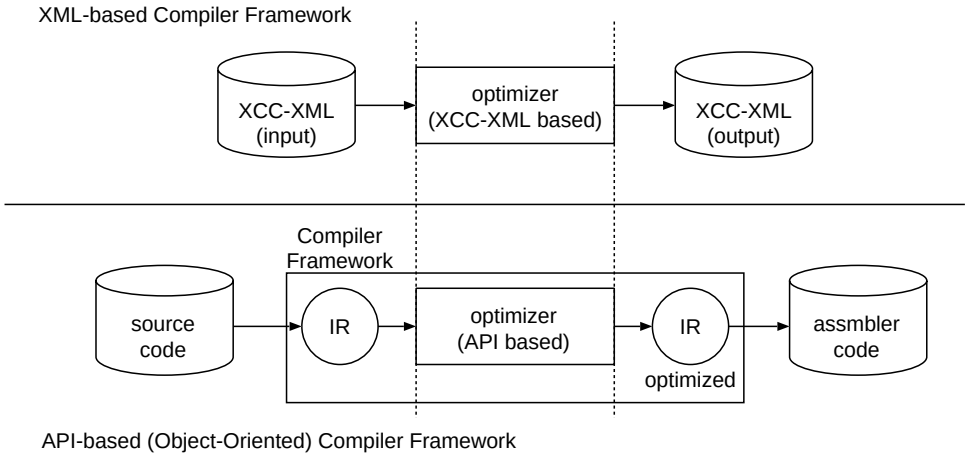

図 6.5: XCC-XML では，最適化器と XCC-XML 文書があればデバッグ，テストが可能である．これに対して，既存のコンパイラフレームワークは，フレームワークに最適化器を組み込んだ上で，実行する必要がある．

6.3 最適化器実装における再利用性

6.3.1 XPath 式の再利用性

我々は，XCC-XML を用いた最適化器の実装予備実験において，XPath が，コードの短縮と最適化器の生産性向上に貢献することを確認した．我々は，この他の XPath の利点として，XPath によって最適化器の実装における再利用性が高くできると考える．その理由は次のとおりである．

1. 言語スキーマとは独立した文法を持つ
   XPath は，特定のプログラミング言語や，データスキーマに依存しない独自の文法を持っている．
2. 入出力の形式が固定
   XPath の入力は，任意の XML 文書，またはコンテキストノードである．出力は，XPath 式を評価した結果得られるノード集合である．
3. 単体でも他の言語からでも使用可能
   XPath は，XPath プロセッサによって実行できる．単体で使用することもできる．また，他のプログラミング言語から XPath プロセッサを利用して使用できる．

これらの理由から，XPath はプラットフォームを越えて移植が可能で，最適化間で再利用可能な部品と考えることができる．しかし，我々の行なった予備実験では，XPath の再利用性を十分に確認できなかった．これは，XPath 式だけでは，再利用のための情報が不足していることによる．再利用のための情報とは，XPath 式の評価の際に期待される

コンテキストノード (検索対象)，検索可能な条件，検索の意図などである．

XPath を再利用する方法の 1 つとして，多数の XPath を，再利用のための情報とともにリポジトリに格納する方法を考える．現時点で，このリポジトリの実装と評価は行なっていない．リポジトリに関して課題となるのが，使用目的に合致した XPath 式を，リポジトリの中から如何にして探し出すか，ということである．これには，XPath に関する情報の記述と検索方法の確立が必要である．また，XPath を再利用するための情報は，XPath 式ごとにユーザーが入力する必要がある．より効率的な運用のためには，XPath 自体の自己記述性を高めることも必要である．

再利用を前提として，ソフトウエア部品の格納と検索のためのリポジトリの研究が，コンポーネントウエアの分野でなされている．しかし，XPath に対する検索のように，検索対象のセマンティクスを考慮した検索手法は，確立されていないのが現状である．また，ソフトウエアに付随してリポジトリに格納される情報に関しても，ソフトウエアのセマンティクスを記述できない．現在，仕様として存在するものの多くが，データの仕様，サービスの仕様を含むクラスやオブジェクトの実装するインターフェース，データ間の関係などの表面的な仕様の記述に留まている．

我々は予備実験において，XPath の再利用性を高める実現可能な方法として，XPath 式に再利用のための情報を埋め込んだ．XPath 式が期待するコンテキストノードを `self` 軸を使って表わすことで実現した．ロケーションパスの最初のステップを `self` 軸にすることで，コンテキストノードを明示できる．また，XPath 式に期待されるコンテキストノードとは異なるノードを基点に XPath を評価した場合には，結果は常に空である．これによって，XPath 式が予期しないコンテキストノードに対して評価された場合に，意図しないノードが列挙されることを防げる．`self` 軸を使ってコンテキストノードを明示する例を示す．

```
self::AST_COMPOUND/SYM_LOCAL/*/@name
```

この例では，`self` 軸に `AST_COMPOUND` が指定されているため，`AST_COMPOUND` 要素に対してのみ機能する．この XPath 式を評価した結果は，任意のブロック内で宣言されたシンボル名を列挙する．これに対して，全てのシンボル名を列挙することも可能である．

```
/XCC_XML/FUNC_BODY//SYM_LOCAL/*/@name
```

しかし，この場合に得られる結果は，すべてのスコープのシンボル名が 1 つの集合に含まれるため，スコープに関する情報が失しなわれている．スコープを考慮してシンボル名を列挙するためには，`AST_COMPOUND` 要素ごとにシンボル名を列挙する必要がある．

次の例も同様である．この例では，`FUNC_BODY` 要素が表わす関数の名前を取得する．

```
self::FUNC_BODY/TYPEREP_FUNC/@name
```

1 つの XCC-XML 文書に，1 つの `FUNC_BODY` 要素しか存在しな場合には，

```
/XCC_XML/FUNC_BODY/TYPEREP_FUNC/@name
```

としてもよい．しかし，1 つの XCC-XML 文書に，複数の FUNC_BODY が存在する場合には，前者の XPath 式を繰り返し使う方が，関数名と対応する XML の要素が明確なため有用である．

我々は，XCC-XML を使った予備実験において，XPath の再利用を試みた．その結果,, XPath が異なる言語，プラットフォーム間にまたがる最適解の開発において，技術的には再利用可能であることが確認されたと考える．しかし，我々の XPath リポジトリに関する提案も，予備実験の中で使用した手法のどちらも，完全な再利用性を実現するには不十分である．

6.3.2 XCC-XML 文書の再利用性

我々の行なった，最適化器実装の予備実験では，Python シェルを使った実証実験，デバッグ，テストで，生成済みの XCC-XML 文書を使用した．生成済みの XCC-XML 文書を最適化器の入力とすることで，コンパイラフレームワークの本体である XCC-XML 処理系を使用することなく，最適化器のデバッグとテストができた．

デバッグ，テスト時のための入力，すなわちテストケースは，まず，XC 言語を用いて記述する．次に，記述した XC ソースから XCC-XML 文書を生成する．XCC-XML 文書の生成には，tee コマンドを使用し，そのための規則を XCC-XML 文書生成用の ODF ファイルに記述した．生成された XCC-XML 文書を，テキストエディタなどで編集し，新たなテストケースの生成も行なった．

現状の ODF ファイルと XCC-XML 処理系では，複数のステップからなる最適化パスを取り扱うことができない．しかし，ある最適化器の出力を XCC-XML 文書として保存し，他の最適化器の入力として使用することで，これを実現することができる．

このように，我々の予備実験において，中間表現を XML 文書で表現することで実現される再利用性が，デバッグやテストに有効であるだけでなく，現状の処理系で実装されていない機能を，暫定的に実現する手段にもなることが明らかになった．

6.4 XML プログラミングインターフェースの現状

6.4.1 最適化器の実装言語の選択

我々の考える XCC-XML 上での最適化器の実装は，DOM と XPath を用いたものである．まず，XML をパーサで読み込み，DOM ツリーを構築する．これを XPath を用いて検索し，特定の条件にマッチする要素を列挙し，処理する．そして，その結果は，元の DOM ツリーに反映され，再び XCC-XML 文書として出力される．我々は，この実装を高級言語で行なうことで，最適化器の生産性向上を試みた．

我々は最適化実装の予備実験で，Python 言語を実装言語として選択した．C，C++，Perl，Ruby，Java など，XML を処理できる言語は数多く存在する．この中で Python を選択したのは，インタープリタ型言語の持つ利点を活かせることが主な理由である．この他に Python を選択した理由として，Python 用に提供されている XML 関連ライブラリの充実が挙げられる．

各種プログラミング言語に対して XML 文書を取り扱うためのライブラリが提供されている．その中でも，XML を処理するために用いられる言語としては，Java が一般的である．Java はコンパイラ型言語であるため，我々の考える生産性向上のための実装言語としては適当でない．しかし，XML 処理を考えた場合，最も有力な言語である．我々は，最適化器の実装言語を決定するにあたって，Java と Python における XML プログラミングインターフェースについて調査を行ない，比較検討した．

6.4.2 Xalan/Xerces における DOM 実装

Java で利用可能な XML パーサ，XSLT プロセッサの主要なものとして，Sun Microsystems による JAXP(Java API for XML Processing) と，Apache Software Foundation による Xerces/Xalan がある．JAXP は，Sun が配布する Java の実装である J2SE に標準で含まれる．Xerces/Xalan は，独立したクラスライブラリとして配布される．JAXP は，J2SE のバージョンによって仕様が異なるため，java のバージョン間の差によらず，共通のインストール手順が使える Xerces/Xalan を評価対象とした．

Xalan は XSLT プロセッサであり，Xerces は XML パーサである．Xalan は Xerces を内部的に利用している．Xalan に含まれる XPath プロセッサは，DOM ツリー上で動作する．この DOM ツリーは読み取り専用で，その実体は DTM(Document Table Model) と呼ばれるデータモデルによって実現されている．図 6.6 に DTM の概要を示す．

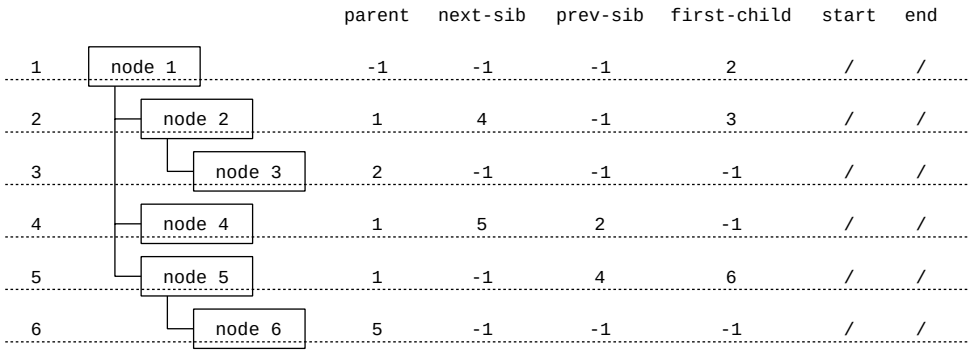

図 6.6: DTM

DTM では，XML の 1 つ 1 つの構文要素は，それぞれ固有の ID で識別される．DTM は，全ての構文要素の XML ファイル上での開始位置，終了位置，親要素の ID，前後の

兄弟要素の ID，最初の子要素の ID を収めた表をつくる．DTM の表は，DOM ツリーのノードを表の項目とすることで，DOM ツリーの各ノードごとに，オブジェクトを生成した場合より，メモリ消費を削減している．しかし，XML 文書上の位置を記録しているため，文書の更新を行なった際には，再構築が必要になる．

我々の予備実験における最適化器の実装では，DOM ツリーに対する検索と読み出しだけでなく，検索結果に基づいて DOM ツリーの更新を行なうため，DTM をベースにした Xalan の実装は，我々の要求を満たさない．

しかし，DTM は，XSLT のためのデータモデルとしては，妥当なものだと考える．それは，XSLT プロセッサの実装が，DOM ツリーに対する読み出しと，その結果に基づいて，新たに DOM ツリーを構築する方法で行なわれているからである．このような実装を行なった場合，DOM は読み取り専用にした方が効率が良い．また，XPath も DOM ツリーを更新するものではないため，DTM ベースで十分実現可能である．

6.4.3 PyXML/4Suite における DOM 実装

我々が予備実験において，実装言語として使用した Python は，標準で DOM インターフェースの実装を提供する．また，これとは別に，より高機能な XML 処理機能を提供する PyXML が存在する．PyXML は，Python が標準で提供する基本的な XML の読み込みだけでなく，SAX と SAX2 インターフェースの実装や，XML 文書の出力機能などを提供する．これに対して，4Suite は，Python で実装された XML と RDF を処理のためのプラットフォームである．我々の行なった予備実験では，4Suite の提供する XPath プロセッサを使用した．

PyXML と 4Suite を使う利点は，4Suite の XPath プロセッサが，PyXML の構築した DOM ツリーで動作することにある．4Suite も DOM の実装を提供しているが，4Suite の XPath プロセッサは，特定の DOM 実装に依存しない．このため，Xalan の場合のような，読み取り専用の特殊な実装と異なり，XPath で検索したあとで，DOM ツリーを自由に操作できる．

Python を実装言語とする利点は，DOM と XPath の関係だけではない．Python の対話型シェルを使えば，XML 文書の読み込みと DOM ツリーの構築，DOM ツリーの操作，XPath 式の評価とその結果を逐一確認しながら行える．

PyXML，4Suite 共に，十分に安定した実装を提供している．PyXML は，Python の次のリリースで，標準で提供される予定である．また，4Suite は，既に運用実績のあるソフトウエアである．

6.5 実用的なコンパイラフレームワークの構築に向けて

6.5.1 XCC-XML の改善

フルセット C への対応

XML を用いた，より実用的なコンパイラフレームワークを構築するためには，現実的なプログラムをコンパイル可能な実装を行なうことが重要である．XCC-XML を用いた実験が，あくまでも予備実験であるのも，XC が C 言語のサブセットだからである．

XML を用いたコンパイラフレームワークが，真に実用的で有効なものであることを実証するには，コンパイラとしての実用性を高める必要がある．そのためには，実際に使用されているソフトウエアのソースコードをコンパイルし，最適化を施すことが重要である．

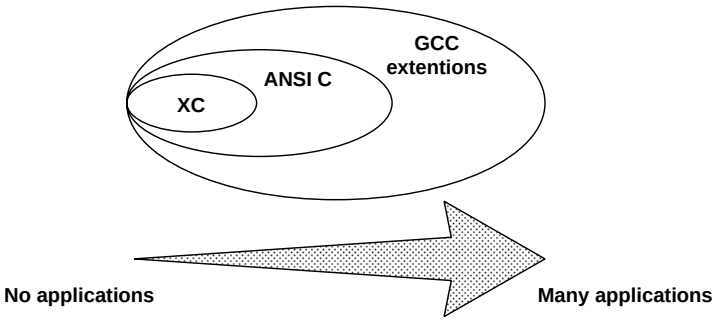

図 6.7: XC，ANSI C，GCC 拡張の関係

現在，ソフトウエア開発に一般的に用いられているコンパイラは，ANSI C に準拠したものがほとんどである．また，Linux 等のオープンソースソフトウエアでは，GCC が一般的である．GCC は，独自に ANSI C を拡張した使用を持っている．このことから，XCC-XML の実用性と有効性を検証するためには，少なくとも ANSI C をサポートしたコンパイラを構築する必要がある．その上で，現実に使用されているソフトウエアのコードを最適化し，実証実験を行なう．

木構造の改善

XML による表現が有効であるためには，表現されるデータが木構造をしていることが重要である．中間表現を XML で表現する場合，入れ子になったスコープは，木のノードと子ノードの関係に対応付けると直感的で，XML での表現も容易である．これに対して，逐次実行されるコードのように，フラットな構造が適したものもある．現状の `xcc` の中間表現では，抽象構文木の影響が強く，木構造で表現するのに適さないものも，木として表現されている．図 6.8 に例を示す．この例では，連続して実行される一連の文のならびを

表現する，AST_STATEMENT_LIST 要素の構造を示している．

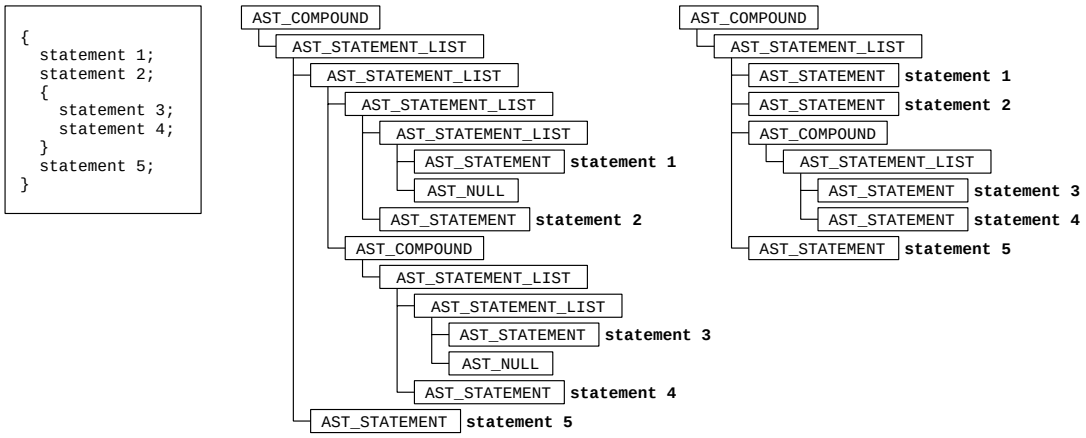

図 6.8: ソースコードと XCC-XML の対応関係．左は改善案

図 6.8 の中央が，左のプログラムを，現在の XCC-XML で表現したものである．ソースコード中に文が出現する順序は，XCC-XML 文書の木の深さに対応する．文の並びの前にある文ほど，木の深い位置に要素がある．逆に，後にある文は，木の浅い位置にある．並びの中の任意の位置にある文に対応する XML 要素を得るためには，木の最も深い位置から，必要な回数分，上にノードを探索する必要がある．

任意の位置にある文を得るためには，XPath の position 関数を使うのが，直感的である．position 関数は，XPath ロケーションステップの評価対象の，コンテキストノードにおける位置を返す関数である．しかし，現状の XCC-XML は，複文の中の各文が，同一のコンテキストノードの下に配置されないので，position 関数を有効に使うことができない．

この例に示したように，過度に木構造を用いると，DOM プログラミングは繁雑になり，XPath も効果的に使えない．逆に木構造を廃し，フラットな構造にすることもまた，XML と DOM，XPath の利点を損なう．中間表現を XML で表現するためには，XML の利点と DOM，XPath の利点が両立するバランスが重要である．

図 6.8 の例に関して，RTL を参考にした解決案を，図中右に示す．XCC-XML を用いた予備実験から，連続した文の並びを表現するには，フラットな構造が適していると考える．これは，RTL でプログラムを表現する場合の構造と同様である．逆に，入れ子になったスコープを表現するには，木構造が適している．

アーキテクチャ依存の最適化

XCC-XML では，レジスタ，メモリアドレス，命令の並行性などのアーキテクチャ依存の情報を記述できない．これは XCC-XML が抽象構文木をベースにした中間表現であることによる．このため，アーキテクチャ依存な最適化は実用的なコンパイラにとっては不

可欠な機能であるにもかかわらず，我々の実装実験では行なっていない．XML に基づくコンパイラフレームワークの有効性を示すためには，XCC-XML がアーキテクチャ依存の最適化に対しても有効であることを示す必要がある．

アーキテクチャ依存な最適化を行なうために，特に必要と考えることは以下の 2 点である．

- **シンボルとレジスタ，メモリの対応付け**

  中間表現ベースにした現在の XCC-XML は，全てのシンボル情報を含んでいる．アセンブラコードを生成するためには，関数内で宣言されたデータオブジェクトの，メモリまたはレジスタへの割り付けが必要である．RTL を例にとれば，レジスタとメモリの割り付けは，コンパイラと最適化器によって行なわれる．GCC は，全てのローカル変数を一旦，擬似的なレジスタに割り付ける．実際の MPU では，レジスタは無限ではないため，ローカル変数の一部は，スタックフレーム上に割り付ける必要がある．これを行なうための最適化器は，ターゲットとなるアーキテクチャに応じて生成される．

- **制御構造の単純化**

  C 言語の持つ制御構造をアセンブラコードで実現するためには，全ての制御構造を条件分岐とジャンプ命令に置き換えなければならない．その上でプログラムの制御フローを解析し，最適化を行なう．現在の XC には，`if`，`while`，`goto` があり，これらは中間表現と XCC-XML 文書でも保存されている．このため，制御フローの解析は全ての制御構造を考慮する必要があり，繁雑である．

現状の XCC-XML 処理系では，SPARC v8 アーキテクチャのみをサポートしている．特定のアーキテクチャを対象にした最適化を行ない，その結果をコード生成に反映するためには，中間表現におけるレジスタとメモリの表現が不可欠である．コード生成時の最適化を行なわなければ，命令の並行性について中間表現で記述しなくてもコード生成は可能である．

6.5.2 ODF の改善

XCC-XML 処理系の特徴の 1 つとして，ODF ファイルによる最適化の制御がある．ODF には，ソースファイル内の関数ごとに，起動する外部オプティマイザを記述できる．

現在の ODF ファイルは，複数の外部オプティマイザの連携をサポートしない．複数の最適化器の組み合せと適用順序が，最適化の結果に大きく影響する．この組み合せに関して，最善の解は存在しないのが現状である．GCC の最適化パスは，過去の経験に基づいて決定されている．複数の最適化器の組み合わせをテストするために．ODF ファイルは，複数のステップからなる最適化パスを記述可能である必要がある．現状の ODF では，外

部オプティマイザを用いて間接的に複数の最適化器を適用できるが，ODF に記述することで記述の再利用性と可搬性を向上できる．

ODF ファイルに記述できる外部オプティマイザの記述は，固定された記述しかできない．コンパイル中の関数に依存したパラメータを外部オプティマイザに渡す方法はない．個別の関数ごとに定義をした場合は，それほど問題ではない．デフォルトの外部オプティマイザの場合には，特に重要である．XCC-XML 文書を解釈するプログラムを，外部オプティマイザとする場合は，XCC-XML 文書に渡す情報を含めることで，間接的にこの問題を解決できる．しかし，XML 文書を解釈しないプログラムや，XCC-XML のデータスキーマを考慮しない汎用の XML ツールなどの場合，コマンドラインを経由した情報の受け渡しが不可欠である．

手続き間の関係を利用する最適化を行なう場合，それぞれの関数を表現する複数の XCC-XML 文書が必要になる．複数の XCC-XML 文書が関係する場合，XCC-XML 文書間に依存関係が生じる．ODF ファイルには，XCC-XML 文書の依存関係を記述できない．XCC-XML 処理系は，ソースコード中に関数が出現した順でコンパイルを行なうため，関数が前方参照される場合には，あらかじめ XCC-XML 文書を生成し，これを解決する必要がある．現状の処理系では，異なる ODF ファイルを使って手動でこれを行な必要がある．

6.6 既存のコンパイラフレームワークとの比較

XCC-XML は，既存のコンパイラフレームワークに対して，XML による中間表現の実現，最適化の実装言語を任意に選択可能などの点で大きく異なっている．逆に個々のコンパイラフレームワークを見ると，XCC-XML と共通点も見られ，その実現方法に一長一短がある．

SUIF

SUIF では，中間表現をストレージに格納するための専用のバイナリフォーマットを持っている．このフォーマットは，ユーザーによる拡張が可能であるが，アクセスするには SUIF のフレームワークが必要である．これに対して，XCC-XML のような XML ベースのコンパイラフレームワークでは，ストレージ上の中間表現に任意のソフトウエアでアクセスできる．

SUIF と XCC-XML では，中間表現の拡張方法も異なっている．SUIF では，中間表現の拡張に Hoof と呼ばれる一種のインターフェース記述言語によって行なわれる．これに対して，XML をベースとする XCC-XML では，XML 名前空間を利用して，中間表現に影響なく拡張が可能である．また，XCC-XML にはもう 1 つの拡張方法がある．XCC-XML の `INFO` 要素は，`XCC_XML` 要素の直下にあり，任意の要素をユーザーが格納できる．

CFL

CFL の特徴の 1 つは，コードサイズ削減のために，新たな命令を定義し，そのためのコードジェネレータを生成する点にある．拡張された命令セットをベースとなるプロセッサに追加するための VHDL コードも同時に生成することで，ASIC などへの実装が可能になる．

近年の組み込み向けプロセッサでは，SoC(System On Chip) のように，汎用プロセッサのコアと専用ハードウエアをワンチップ化した製品が多く見られる．SoC 実現方法によっては，汎用の命令セットにこれらのハードウエアの機能にアクセスするための，拡張命令セットを追加するものがある．これらの特殊なプロセッサに対しては，汎用のコンパイラが行なう最適化は効果が低い．拡張された命令セットを有効に活用する最適化器が必要である．

しかし，SoC は，特殊なニーズに応えるための製品であることが多く，そのための最適化コンパイラの開発はコストが高い．そこで，XCC-XML がアーキテクチャ依存な最適化に対応するれば，SoC などに見られる特殊な拡張命令を生かしたコンパイラを，短期間で開発するための基盤になる．

OpenJIT

OpenJIT では，Java のクラスファイルに最適化のためのアノテーション情報を付加することで，クラス毎に計算環境に特化した最適化を可能にしている．OpenJIT によって，特定のアプリケーションに特化した最適化を選択的に JIT が適用できる．

OpenJIT は，Java 仮想マシンと JIT の関係を利用して，実行時にアノテーションに基づく最適化を行なう．これに対して，XCC-XML 処理系は，アノテーションに相当する情報を ODF ファイルに記述し，コンパイル時に利用する．

第7章　おわりに

7.1　コンパイラフレームワーク実現のまとめ

本研究で我々は，コンパイラフレームワークの実現方法として，GCC を拡張する手法と新規にコンパイラを構築する手法の 2 つの手法を用いてそれぞれ実装を行なった．前者は，コンパイラの代表格である GCC をベースにすることで，実用的なコンパイラフレームワークの構築が可能であるが，GCC の約 90 万行という膨大な量のソースコードを解析する必要がある．後者は，コンパイラを新規に構築するためのコストを伴なうが，中間表現を含むコンパイラの各部を理想的な形で実現できる．

GCC を拡張する手法においては，GCC の中間表現である RTL を XML で表現するマークアップ言語 RTL-XML を設計した．我々は，GCC に RTL-XML による中間表現の出力，RTL-XML 上で最適化を行なう外部プログラムの起動，最適化された RTL-XML 文書の取り込みを行なう拡張の実装を試みた．しかし，GCC の内部データ構造が，RTL に依存していたことにより，完全な実装には致らなかった．

新規にコンパイラを構築する手法では，ANSI C のサブセットである XC 言語を対象にコンパイラフレームワークの構築を行なった．実装した処理系は，中間表現として XCC-XML 言語で表現し，XCC-XML 上で最適化を行なう外部プログラムが最適化を行なう．ソースファイルごとに適用する最適化器は，処理系に与えられる ODF ファイルの記述に基づいて選択される．

コンパイラフレームワークとして完全に機能しない GCC ベースの実装では，RTL-XML 文書として出力された RTL をウェブブラウザ上で表示するビューアを試作した．XML で表現された中間表現の 2 次的利用の一例である．また，XC を対象にした実装では，オブジェクト指向のスクリプト言語である Python を用いて最適化器を実装し，評価を行なった．

7.2　まとめ

我々は，既存のコンパイラフレームワークがプログラミングインターフェースを中心に実装されており，それが開発手法の柔軟性を損なわせ，結果的に最適化器の生産性を下げる要因になったと考えた．そこで我々は，最適化器の生産性を向上させるために，XML を用いたデータスキーマ中心のコンパイラフレームワークを提案した．

我々の提案するコンパイラフレームワークの実現方法として，GCC を拡張する手法と ANSI C のサブセットである XC をターゲットに新規にコンパイラフレームワークを構築

する手法の 2 種類の手法をとった．前者の実装は不完全なものに終わり，後者の実装を用いて最適化器の実装実験を行なった．

実装した最適化器は，多くのコンパイラで行なわれている最適化のサブセットであり，限定された条件でのみ動作する．しかし，この実験において，中間表現を読み込み，何らかの最適化を施すプログラムの開発を長いものでも 2 日，短かいものでは数十分で実装できた．

我々の実験は，ANSI C のサブセットを対象にしているため，この結果がフルセットの C 言語に適用可能であるとは言えない．また，実験で用いた中間表現は，抽象構文木をベースにしたもので，アーキテクチャに依存した最適化を行なえない．我々の行なった実装実験では，XML を用いたデータスキーマ中心のコンパイラフレームワークが，最適化器の生産性を向上させることを実証するには不十分である．そのための予備的なものとしては，十分に意義があると考える．

7.3 結論

本研究において我々の行なった実験から，XML とデータスキーマ中心のコンパイラフレームワークには，最適化器の生産性を大きく向上させる可能性があると考える．その理由として，実験において確認した事実を挙げる．

- 最適化器の実装言語を開発者が任意に選択できる．
- 高級言語を用いることで，対話型のプログラミング，プロトタイピングなどの手法が最適化器の開発に適用できる．
- XML 関連技術は，中間表現の操作に有効である．
- 中間表現を XML で表現することで，2 次的な利用と再利用が可能である．

しかし，これらの事実は，C 言語サブセットの XC を対象にした実験において確認されたもので，フルセットの C 言語で記述された，実用的なコードに対して得られたものではない．このことから本研究で行なった実験は，XML を使ったコンパイラフレームワークの有効性を確認するための，予備的な位置付けにあると考える．

また，我々は，コンパイラフレームワークの実装において GCC のソースコードの解析と拡張を試みた．結果的には十分な実装にはならなかったものの，GCC の内部データ構造に関する有益な情報を得ることができた．同時に，GCC には，文書化されていない部分が大く存在することも明かになった．これはソフトウエア，特にオープンソースソフトウエアの抱える問題の 1 つを示すものである．

我々の行なった実験を予備的なものと考え，今後，より詳細な実験によって，XML を用いたコンパイラフレームワークの有効性を実証していく必要がある．そのためには，実用的なソフトウエアのコンパイルと最適化を行なってはじめて，有効性が実証できると考

える．加えて，予備実験で行なわなかった，アーキテクチャ依存な最適化の実現が必要であり，そのための中間表現の設計が今後の課題である．

最適化器の開発手法に焦点を当て，その生産性を考慮した研究は少ない．本研究では，最適化器の生産性向上のために，XML とその関連技術，Python をはじめとする高級言語などの，広く普及した技術によってコンパイラフレームワークを構築した．これは，コンパイラと最適化器をより多くの研究者，開発者に開放する上で重要である．

7.4 今後の課題

これまで述べてきたように，本研究で行なった実装実験では，XML に基づくコンパイラフレームワークの有効性を実証するには，不十分である．これは，主に 2 つの理由がある．1 つは，XC が C 言語のサブセットであり，広く使われている ANSI C や GCC 拡張のコードを処理できないこと，もう 1 つは，XCC-XML が抽象構文木ベースであるため，アーキテクチャに依存した最適化に関して実験を行なえなかったことによる．特にアーキテクチャ依存な最適化に関しては，中間表現の再設計が必要であり，上記の 2 つの問題を解決することが当面の課題である．

当面の課題を解決し，XML に基づくコンパイラフレームワークの有効性を示した上での，今後の研究課題を示す．

- 最適化の種類と目的に応じた多階層の中間表現の設計と実現．

  コンパイラの行なう最適化は，アーキテクチャ非依存な構文的なものと，アーキテクチャに依存した最適化の 2 種類に大別される．これを効率的に行なうには，構文的な最適化に適した抽象構文木ベースの中間表現と，アーキテクチャ依存な最適化に適したアセンブラに近いレベルの，少なくとも 2 種類の中間表現が用意されることが望ましい．

- 複数アーキテクチャへの対応．

  当面の実装は，SPARC v8 アーキテクチャを対象に行なうが，最終的には複数のアーキテクチャのサポートを目指す．特殊な命令セットが追加されたプロセッサに対処するため，GCC のようにコンパイラの構築時にターゲットを静的に決定するのではなく，必要に応じて迅速にバックエンドを整備できる動的なものが必要である．

  - アーキテクチャごとの特性を記述する記述形式と処理系の実現．
  - 複数アーキテクチャをサポートする拡張可能なバックエンドフレームワークの構築．

- GCC をはじめとする既存のコンパイラとの互換性の確保．

コンパイラフレームワーク上で開発した最適化器が，より広くその効果を発揮するためには，コンパイラフレームワークが，既存のコンパイラと互換性を持つことが重要である．コンパイラフレームワークの持つべき互換性は，次の 2 つである．

  - ソースレベルで GCC 互換.
  - バイナリレベルでの GCC 互換.

- XML で表現された中間表現の 2 次的利用法の研究

  本研究で我々は，RTL-XML を視覚化するツールを RTL-XML の 2 次的利用の例として試作した．これ以外にも，スライサーやクロスリファレンサなどの各種 CASE ツールへの応用も検討すべきである．

- XPath を再利用するためのリポジトリの構築

  我々は，6.3.1 節で，最適化器の実装において，XPath 式を再利用するためのリポジトリを提案した．これに関して，リポジトリにおける XPath 式の仕様の記述と検索について，考慮すべきてんがいくつかある．コンポーネントウエアの分野で行なわれている，再利用可のためのソフトウエアリポジトリの構築とも関連して，今後研究を行なう必要がある．

謝辞

本論文を執筆するに当たり終始熱心な御指導を頂きました権藤克彦助教授，貴重な御意見と有意義な議論を共にして頂いた片山卓也教授をはじめとするソフトウエア基礎講座の皆様に感謝を申し上げます.

付 録A　RTL-XML DTD

```
<!ENTITY % common_attr "mode (VOID | BI | QI | HI | SI | DI | TI | OI
| PQI | PHI | PSI | PDI | QF | HF | TQF |   SF | DF | XF |   TF | QC |
HC | SC | DC | XC | TC | CQI | CHI | CSI | CDI | CTI | COI | V2QI |
V2HI | V2SI | V2DI | V4QI | V4HI | V4SI | V4DI | V8QI | V8HI | V8SI |
V8DI | V16QI | V2SF | V2DF | V4SF | V4DF | V8SF | V8DF | V16SF | BLK |
CC) #REQUIRED">

<!ENTITY % insn_attr "uid CDATA #REQUIRED">

<!ENTITY % fmtc_e "(unknown | nil | include | expr_list | insn_list |
match_operand | match_scratch | match_dup | match_operator |
match_parallel | match_op_dup | match_par_dup | match_insn |
define_insn | define_peephole | define_split | define_insn_and_split |
define_peephole2 | define_combine | define_expand | define_delay |
define_function_unit | define_asm_attributes | define_cond_exec |
sequence | address | define_attr | attr | set_attr |
set_attr_alternative | eq_attr | attr_flag | insn | jump_insn |
call_insn | barrier | code_label | note | cond_exec | parallel |
asm_input | asm_operands | unspec | unspec_volatile | addr_vec |
addr_diff_vec | prefetch | set | use | clobber | call | return |
trap_if | resx | const_int | const_double | const_vector |
const_string | const | pc | value | reg | scratch | subreg |
strict_low_part | concat | mem | label_ref | symbol_ref | cc0 |
addressof | queued | if_then_else | cond | compare | plus | minus |
neg | mult | div | mod | udiv | umod | and | ior | xor | not | ashift
| rotate | ashiftrt | lshiftrt | rotatert | smin | smax | umin | umax
| pre_dec | pre_inc | post_dec | post_inc | pre_modify | post_modify |
ne | eq | ge | gt | le | lt | geu | gtu | leu | ltu | unordered |
ordered | uneq | unge | ungt | unle | unlt | ltgt | sign_extend |
zero_extend | truncate | float_extend | float_truncate | float | fix |
unsigned_float | unsigned_fix | abs | sqrt | ffs | sign_extract |
zero_extract | high | lo_sum | range_info | range_reg | range_var |
range_live | constant_p_rtx | call_placeholder | vec_merge |
vec_select | vec_concat | vec_duplicate | ss_plus | us_plus | ss_minus
| us_minus | ss_truncate | us_truncate | phi)">

<!ENTITY % fmtc_i "(rtl_int)">
```

```
<!ENTITY % fmtc_w "(rtl_int)">
<!ENTITY % fmtc_str "(rtl_string)">
<!ENTITY % fmtc_vec "(rtl_vector)">
<!ENTITY % fmtc_u "(call_insn | jump_insn
                    | note | insn | code_label | barrier)">
<!ENTITY % fmtc_note "">
<!ENTITY % fmtc_n "(rtl_int)">
<!ENTITY % fmtc_str_o "(rtl_string)">
<!ENTITY % fmtc_vec_o "(rtl_vector)">
<!ENTITY % fmtc_bitmap "">
<!ENTITY % fmtc_tree "">
<!ENTITY % fmtc_tmpl "">

<!-- document root -->

<!ELEMENT rtl-xml (target-spec, rtl-body)>

<!-- machine dependet info -->

<!ELEMENT target-spec (first-pseudo-register)>

<!-- function -->

<!ELEMENT rtl-body ((%fmtc_u;)*)>

<!-- RTL basic data types -->

<!ELEMENT rtl_int (#PCDATA)>
<!ELEMENT rtl_double (#PCDATA)>
<!ELEMENT rtl_string (#PCDATA)>
<!ELEMENT rtl_vector (%fmtc_e;*)>
<!ELEMENT rtl_intprt (#PCDATA)>

<!-- rtx code -->

<!ELEMENT unknown ANY>
<!ELEMENT nil ANY>
<!ELEMENT include (%fmtc_str;)>
<!ELEMENT expr_list (%fmtc_e;,%fmtc_e;)>
<!ELEMENT insn_list (%fmtc_u;,%fmtc_e;)>
<!ELEMENT match_operand (%fmtc_i;,%fmtc_str;,%fmtc_str;)>
<!ELEMENT match_scratch (%fmtc_i;,%fmtc_str;)>
<!ELEMENT match_dup (%fmtc_i;)>
<!ELEMENT match_operator (%fmtc_i;,%fmtc_str;,%fmtc_vec;)>
<!ELEMENT match_parallel (%fmtc_i;,%fmtc_str;,%fmtc_vec;)>
<!ELEMENT match_op_dup (%fmtc_i;,%fmtc_vec;)>
```

```
<!ELEMENT match_par_dup (%fmtc_i;,%fmtc_vec;)>
<!ELEMENT match_insn (%fmtc_i;,%fmtc_str;)>
<!ELEMENT define_insn (%fmtc_str;,%fmtc_vec;,%fmtc_str;,
                       %fmtc_tmpl;,%fmtc_vec_o;)>
<!ELEMENT define_peephole (%fmtc_vec;,%fmtc_str;,%fmtc_tmpl;,%fmtc_vec_o;)>
<!ELEMENT define_split (%fmtc_vec;,%fmtc_str;,%fmtc_vec;,%fmtc_str_o;)>
<!ELEMENT define_insn_and_split (%fmtc_str;,%fmtc_vec;,%fmtc_str;,
                                 %fmtc_tmpl;, %fmtc_str;,%fmtc_vec;,
                                 %fmtc_str_o;,%fmtc_vec_o;)>
<!ELEMENT define_peephole2 (%fmtc_vec;,%fmtc_str;,%fmtc_vec;,%fmtc_str_o;)>
<!ELEMENT define_combine (%fmtc_vec;,%fmtc_str;,%fmtc_str;)>
<!ELEMENT define_expand (%fmtc_str;,%fmtc_vec;,%fmtc_str;,%fmtc_str;)>
<!ELEMENT define_delay (%fmtc_e;,%fmtc_vec;)>
<!ELEMENT define_function_unit (%fmtc_str;,%fmtc_i;,%fmtc_i;,%fmtc_e;,
                                %fmtc_i;,%fmtc_i;,%fmtc_vec_o;)>
<!ELEMENT define_asm_attributes (%fmtc_vec_o;)>
<!ELEMENT define_cond_exec (%fmtc_vec;,%fmtc_str;,%fmtc_str;)>
<!ELEMENT sequence (%fmtc_vec;)>
<!ELEMENT address (%fmtc_e;)>
<!ELEMENT define_attr (%fmtc_str;,%fmtc_str;,%fmtc_e;)>
<!ELEMENT attr (%fmtc_str;)>
<!ELEMENT set_attr (%fmtc_str;,%fmtc_str;)>
<!ELEMENT set_attr_alternative (%fmtc_str;,%fmtc_vec;)>
<!ELEMENT eq_attr (%fmtc_str;,%fmtc_str;)>
<!ELEMENT attr_flag (%fmtc_str;)>

<!-- INSN -->

<!ELEMENT insn (%fmtc_e;,%fmtc_i;,%fmtc_e;,%fmtc_e;)>
<!ELEMENT jump_insn (%fmtc_e;,%fmtc_i;,%fmtc_e;,%fmtc_e;)>
<!ELEMENT call_insn (%fmtc_e;,%fmtc_i;,%fmtc_e;,%fmtc_e;,%fmtc_e;)>
<!ELEMENT barrier ()>
<!ELEMENT code_label (%fmtc_i;,%fmtc_str;,%fmtc_str;)>
<!ELEMENT note (%fmtc_note;,%fmtc_n;,%fmtc_i;)>

<!ELEMENT cond_exec (%fmtc_e;,%fmtc_e;)>
<!ELEMENT parallel (%fmtc_vec;)>
<!ELEMENT asm_input (%fmtc_str;)>
<!ELEMENT asm_operands (%fmtc_str;,%fmtc_str;,%fmtc_i;,%fmtc_vec;,
                        %fmtc_vec;,%fmtc_str;,%fmtc_i;)>
<!ELEMENT unspec (%fmtc_vec;,%fmtc_i;)>
<!ELEMENT unspec_volatile (%fmtc_vec;,%fmtc_i;)>
<!ELEMENT addr_vec (%fmtc_vec;)>
<!ELEMENT addr_diff_vec (%fmtc_e;,%fmtc_vec;,%fmtc_e;,%fmtc_e;)>
<!ELEMENT prefetch (%fmtc_e;,%fmtc_e;,%fmtc_e;)>
<!ELEMENT set (%fmtc_e;,%fmtc_e;)>
```

```
<!ELEMENT use (%fmtc_e;)>
<!ELEMENT clobber (%fmtc_e;)>
<!ELEMENT call (%fmtc_e;,%fmtc_e;)>
<!ELEMENT return EMPTY>
<!ELEMENT trap_if (%fmtc_e;,%fmtc_e;)>
<!ELEMENT resx (%fmtc_i;)>
<!ELEMENT const_int (%rtl_int;)>
<!ELEMENT const_double (%rtl_double;)>
<!ELEMENT const_vector (%rtl_vector;)>
<!ELEMENT const_string (%rtl_string;)>
<!ELEMENT const (%fmtc_e;)>
<!ELEMENT pc EMPTY>
<!ELEMENT value EMPTY>
<!ELEMENT reg (%fmtc_i;)>
<!ELEMENT scratch EMPTY>
<!ELEMENT subreg (%fmtc_e;,%fmtc_i;)>
<!ELEMENT strict_low_part (%fmtc_e;)>
<!ELEMENT concat (%fmtc_e;,%fmtc_e;)>
<!ELEMENT mem (%fmtc_e;,%fmtc_w;)>
<!ELEMENT label_ref (%fmtc_u;)>
<!ELEMENT symbol_ref (%fmtc_str;)>
<!ELEMENT cc0 EMPTY>
<!ELEMENT addressof (%fmtc_e;,%fmtc_i;,%fmtc_tree;)>
<!ELEMENT queued (%fmtc_e;,%fmtc_e;,%fmtc_e;,%fmtc_e;,%fmtc_e;)>
<!ELEMENT if_then_else (%fmtc_e;,%fmtc_e;,%fmtc_e;)>
<!ELEMENT cond (%fmtc_vec;,%fmtc_e;)>
<!ELEMENT compare (%fmtc_e;,%fmtc_e;)>
<!ELEMENT plus (%fmtc_e;,%fmtc_e;)>
<!ELEMENT minus (%fmtc_e;,%fmtc_e;)>
<!ELEMENT neg (%fmtc_e;)>
<!ELEMENT mult (%fmtc_e;,%fmtc_e;)>
<!ELEMENT div (%fmtc_e;,%fmtc_e;)>
<!ELEMENT mod (%fmtc_e;,%fmtc_e;)>
<!ELEMENT udiv (%fmtc_e;,%fmtc_e;)>
<!ELEMENT umod (%fmtc_e;,%fmtc_e;)>
<!ELEMENT and (%fmtc_e;,%fmtc_e;)>
<!ELEMENT ior (%fmtc_e;,%fmtc_e;)>
<!ELEMENT xor (%fmtc_e;,%fmtc_e;)>
<!ELEMENT not (%fmtc_e;)>
<!ELEMENT ashift (%fmtc_e;,%fmtc_e;)>
<!ELEMENT rotate (%fmtc_e;,%fmtc_e;)>
<!ELEMENT ashiftrt (%fmtc_e;,%fmtc_e;)>
<!ELEMENT lshiftrt (%fmtc_e;,%fmtc_e;)>
<!ELEMENT rotatert (%fmtc_e;,%fmtc_e;)>
<!ELEMENT smin (%fmtc_e;,%fmtc_e;)>
<!ELEMENT smax (%fmtc_e;,%fmtc_e;)>
```

```
<!ELEMENT umin (%fmtc_e;,%fmtc_e;)>
<!ELEMENT umax (%fmtc_e;,%fmtc_e;)>
<!ELEMENT pre_dec (%fmtc_e;)>
<!ELEMENT pre_inc (%fmtc_e;)>
<!ELEMENT post_dec (%fmtc_e;)>
<!ELEMENT post_inc (%fmtc_e;)>
<!ELEMENT pre_modify (%fmtc_e;,%fmtc_e;)>
<!ELEMENT post_modify (%fmtc_e;,%fmtc_e;)>
<!ELEMENT ne (%fmtc_e;,%fmtc_e;)>
<!ELEMENT eq (%fmtc_e;,%fmtc_e;)>
<!ELEMENT ge (%fmtc_e;,%fmtc_e;)>
<!ELEMENT gt (%fmtc_e;,%fmtc_e;)>
<!ELEMENT le (%fmtc_e;,%fmtc_e;)>
<!ELEMENT lt (%fmtc_e;,%fmtc_e;)>
<!ELEMENT geu (%fmtc_e;,%fmtc_e;)>
<!ELEMENT gtu (%fmtc_e;,%fmtc_e;)>
<!ELEMENT leu (%fmtc_e;,%fmtc_e;)>
<!ELEMENT ltu (%fmtc_e;,%fmtc_e;)>
<!ELEMENT unordered (%fmtc_e;,%fmtc_e;)>
<!ELEMENT ordered (%fmtc_e;,%fmtc_e;)>
<!ELEMENT uneq (%fmtc_e;,%fmtc_e;)>
<!ELEMENT unge (%fmtc_e;,%fmtc_e;)>
<!ELEMENT ungt (%fmtc_e;,%fmtc_e;)>
<!ELEMENT unle (%fmtc_e;,%fmtc_e;)>
<!ELEMENT unlt (%fmtc_e;,%fmtc_e;)>
<!ELEMENT ltgt (%fmtc_e;,%fmtc_e;)>
<!ELEMENT sign_extend (%fmtc_e;)>
<!ELEMENT zero_extend (%fmtc_e;)>
<!ELEMENT truncate (%fmtc_e;)>
<!ELEMENT float_extend (%fmtc_e;)>
<!ELEMENT float_truncate (%fmtc_e;)>
<!ELEMENT float (%fmtc_e;)>
<!ELEMENT fix (%fmtc_e;)>
<!ELEMENT unsigned_float (%fmtc_e;)>
<!ELEMENT unsigned_fix (%fmtc_e;)>
<!ELEMENT abs (%fmtc_e;)>
<!ELEMENT sqrt (%fmtc_e;)>
<!ELEMENT ffs (%fmtc_e;)>
<!ELEMENT sign_extract (%fmtc_e;,%fmtc_e;,%fmtc_e;)>
<!ELEMENT zero_extract (%fmtc_e;,%fmtc_e;,%fmtc_e;)>
<!ELEMENT high (%fmtc_e;)>
<!ELEMENT lo_sum (%fmtc_e;,%fmtc_e;)>
<!ELEMENT range_info (%fmtc_u;,%fmtc_u;,%fmtc_vec;,%fmtc_i;,%fmtc_i;,
                      %fmtc_i;,%fmtc_i;,%fmtc_i;,%fmtc_i;,
                      %fmtc_bitmap;,%fmtc_bitmap;,%fmtc_i;,%fmtc_i;)>
<!ELEMENT range_reg (%fmtc_i;,%fmtc_i;,%fmtc_i;,%fmtc_i;,%fmtc_i;,
```

```
                    %fmtc_i;,%fmtc_i;,%fmtc_i;,%fmtc_tree;,%fmtc_tree;)>
<!ELEMENT range_var (%fmtc_e;,%fmtc_tree;,%fmtc_i;)>
<!ELEMENT range_live (%fmtc_bitmap;,%fmtc_i;)>
<!ELEMENT constant_p_rtx (%fmtc_e;)>
<!ELEMENT call_placeholder (%fmtc_u;,%fmtc_u;,%fmtc_u;,%fmtc_u;)>
<!ELEMENT vec_merge (%fmtc_e;,%fmtc_e;,%fmtc_e;)>
<!ELEMENT vec_select (%fmtc_e;,%fmtc_e;)>
<!ELEMENT vec_concat (%fmtc_e;,%fmtc_e;)>
<!ELEMENT vec_duplicate (%fmtc_e;)>
<!ELEMENT ss_plus (%fmtc_e;,%fmtc_e;)>
<!ELEMENT us_plus (%fmtc_e;,%fmtc_e;)>
<!ELEMENT ss_minus (%fmtc_e;,%fmtc_e;)>
<!ELEMENT us_minus (%fmtc_e;,%fmtc_e;)>
<!ELEMENT ss_truncate (%fmtc_e;)>
<!ELEMENT us_truncate (%fmtc_e;)>
<!ELEMENT phi (%fmtc_vec;)>

<!-- attributes -->

<!ATTLIST unknown %common_attr;>
<!ATTLIST nil %common_attr;>
<!ATTLIST include %common_attr;>
<!ATTLIST expr_list %common_attr;>
<!ATTLIST insn_list %common_attr;>
<!ATTLIST match_operand %common_attr;>
<!ATTLIST match_scratch %common_attr;>
<!ATTLIST match_dup %common_attr;>
<!ATTLIST match_operator %common_attr;>
<!ATTLIST match_parallel %common_attr;>
<!ATTLIST match_op_dup %common_attr;>
<!ATTLIST match_par_dup %common_attr;>
<!ATTLIST match_insn %common_attr;>
<!ATTLIST define_insn %common_attr;>
<!ATTLIST define_peephole %common_attr;>
<!ATTLIST define_split %common_attr;>
<!ATTLIST define_insn_and_split %common_attr;>
<!ATTLIST define_peephole2 %common_attr;>
<!ATTLIST define_combine %common_attr;>
<!ATTLIST define_expand %common_attr;>
<!ATTLIST define_delay %common_attr;>
<!ATTLIST define_function_unit %common_attr;>
<!ATTLIST define_asm_attributes %common_attr;>
<!ATTLIST define_cond_exec %common_attr;>
<!ATTLIST sequence %common_attr;>
<!ATTLIST address %common_attr;>
<!ATTLIST define_attr %common_attr;>
```

```
<!ATTLIST attr %common_attr;>
<!ATTLIST set_attr %common_attr;>
<!ATTLIST set_attr_alternative %common_attr;>
<!ATTLIST eq_attr %common_attr;>
<!ATTLIST attr_flag %common_attr;>

<!-- INSN -->

<!ATTLIST insn %insn_attr; %common_attr;>
<!ATTLIST jump_insn %insn_attr; %common_attr;>
<!ATTLIST call_insn %insn_attr; %common_attr;>
<!ATTLIST barrier %insn_attr; %common_attr;>
<!ATTLIST code_label %insn_attr; %common_attr;>
<!ATTLIST note %insn_attr; %common_attr;>

<!ATTLIST cond_exec %common_attr;>
<!ATTLIST parallel %common_attr;>
<!ATTLIST asm_input %common_attr;>
<!ATTLIST asm_operands %common_attr;>
<!ATTLIST unspec %common_attr;>
<!ATTLIST unspec_volatile %common_attr;>
<!ATTLIST addr_vec %common_attr;>
<!ATTLIST addr_diff_vec %common_attr;>
<!ATTLIST prefetch %common_attr;>
<!ATTLIST set %common_attr;>
<!ATTLIST use %common_attr;>
<!ATTLIST clobber %common_attr;>
<!ATTLIST call %common_attr;>
<!ATTLIST return %common_attr;>
<!ATTLIST trap_if %common_attr;>
<!ATTLIST resx %common_attr;>
<!ATTLIST const_int %common_attr;>
<!ATTLIST const_double %common_attr;>
<!ATTLIST const_vector %common_attr;>
<!ATTLIST const_string %common_attr;>
<!ATTLIST const %common_attr;>
<!ATTLIST pc %common_attr;>
<!ATTLIST value %common_attr;>
<!ATTLIST reg %common_attr;>
<!ATTLIST scratch %common_attr;>
<!ATTLIST subreg %common_attr;>
<!ATTLIST strict_low_part %common_attr;>
<!ATTLIST concat %common_attr;>
<!ATTLIST mem %common_attr;>
<!ATTLIST label_ref %common_attr;>
<!ATTLIST symbol_ref %common_attr;>
```

```
<!ATTLIST cc0 %common_attr;>
<!ATTLIST addressof %common_attr;>
<!ATTLIST queued %common_attr;>
<!ATTLIST if_then_else %common_attr;>
<!ATTLIST cond %common_attr;>
<!ATTLIST compare %common_attr;>
<!ATTLIST plus %common_attr;>
<!ATTLIST minus %common_attr;>
<!ATTLIST neg %common_attr;>
<!ATTLIST mult %common_attr;>
<!ATTLIST div %common_attr;>
<!ATTLIST mod %common_attr;>
<!ATTLIST udiv %common_attr;>
<!ATTLIST umod %common_attr;>
<!ATTLIST and %common_attr;>
<!ATTLIST ior %common_attr;>
<!ATTLIST xor %common_attr;>
<!ATTLIST not %common_attr;>
<!ATTLIST ashift %common_attr;>
<!ATTLIST rotate %common_attr;>
<!ATTLIST ashiftrt %common_attr;>
<!ATTLIST lshiftrt %common_attr;>
<!ATTLIST rotatert %common_attr;>
<!ATTLIST smin %common_attr;>
<!ATTLIST smax %common_attr;>
<!ATTLIST umin %common_attr;>
<!ATTLIST umax %common_attr;>
<!ATTLIST pre_dec %common_attr;>
<!ATTLIST pre_inc %common_attr;>
<!ATTLIST post_dec %common_attr;>
<!ATTLIST post_inc %common_attr;>
<!ATTLIST pre_modify %common_attr;>
<!ATTLIST post_modify %common_attr;>
<!ATTLIST ne %common_attr;>
<!ATTLIST eq %common_attr;>
<!ATTLIST ge %common_attr;>
<!ATTLIST gt %common_attr;>
<!ATTLIST le %common_attr;>
<!ATTLIST lt %common_attr;>
<!ATTLIST geu %common_attr;>
<!ATTLIST gtu %common_attr;>
<!ATTLIST leu %common_attr;>
<!ATTLIST ltu %common_attr;>
<!ATTLIST unordered %common_attr;>
<!ATTLIST ordered %common_attr;>
<!ATTLIST uneq %common_attr;>
```

```
<!ATTLIST unge %common_attr;>
<!ATTLIST ungt %common_attr;>
<!ATTLIST unle %common_attr;>
<!ATTLIST unlt %common_attr;>
<!ATTLIST ltgt %common_attr;>
<!ATTLIST sign_extend %common_attr;>
<!ATTLIST zero_extend %common_attr;>
<!ATTLIST truncate %common_attr;>
<!ATTLIST float_extend %common_attr;>
<!ATTLIST float_truncate %common_attr;>
<!ATTLIST float %common_attr;>
<!ATTLIST fix %common_attr;>
<!ATTLIST unsigned_float %common_attr;>
<!ATTLIST unsigned_fix %common_attr;>
<!ATTLIST abs %common_attr;>
<!ATTLIST sqrt %common_attr;>
<!ATTLIST ffs %common_attr;>
<!ATTLIST sign_extract %common_attr;>
<!ATTLIST zero_extract %common_attr;>
<!ATTLIST high %common_attr;>
<!ATTLIST lo_sum %common_attr;>
<!ATTLIST range_info %common_attr;>
<!ATTLIST range_reg %common_attr;>
<!ATTLIST range_var %common_attr;>
<!ATTLIST range_live %common_attr;>
<!ATTLIST constant_p_rtx %common_attr;>
<!ATTLIST call_placeholder %common_attr;>
<!ATTLIST vec_merge %common_attr;>
<!ATTLIST vec_select %common_attr;>
<!ATTLIST vec_concat %common_attr;>
<!ATTLIST vec_duplicate %common_attr;>
<!ATTLIST ss_plus %common_attr;>
<!ATTLIST us_plus %common_attr;>
<!ATTLIST ss_minus %common_attr;>
<!ATTLIST us_minus %common_attr;>
<!ATTLIST ss_truncate %common_attr;>
<!ATTLIST us_truncate %common_attr;>
<!ATTLIST phi %common_attr;>
```

付 録A　XCC-XML DTD

```
<!ELEMENT XCC_XML (SYM_GLOBAL, FUNC_BODY, MESSAGE, INFO*)>
<!ELEMENT SYM_GLOBAL (TYPEREP_PRIM | TYPEREP_FUNC | TYPEREP_PTR)*>
<!ELEMENT FUNC_BODY (TYPEREP_FUNC, AST_COMPOUND)>
<!ELEMENT MESSAGE (AST_ERRORMSG | AST_WARNINGMSG)*>
<!ELEMENT INFO any>

<!ELEMENT TYPEREP_PRIM EMPTY>
<!ELEMENT TYPEREP_FUNC (RETURN, ARGS)>
<!ELEMENT TYPEREP_PTR  (TYPEREP_PRIM | TYPEREP_FUNC | TYPEREP_PTR)>

<!ATTLIST TYPEREP_PRIM name CDATA #IMPLIED
       type (int | void | char) #REQUIRED>
<!ATTLIST TYPEREP_FUNC name CDATA #IMPLIED>
<!ATTLIST TYPEREP_PTR  name CDATA #IMPLIED>

<!ELEMENT RETURN (TYPEREP_PRIM | TYPEREP_FUNC | TYPEREP_PTR)>
<!ELEMENT ARGS (TYPEREP_PRIM | TYPEREP_FUNC | TYPEREP_PTR)*>

<!ENTITY % typespec "(AST_TYPE_VOID | AST_TYPE_INT | AST_TYPE_CHAR)">

<!ENTITY % declarator "(AST_DECLARATOR| AST_POINTER_DECL | AST_FUNC_DECL)">

<!ENTITY % statement "(AST_STATEMENT | AST_COMPOUND | AST_IF | AST_WHILE
                      | AST_GOTO | AST_STATEMENT_WITH_LABEL | AST_RETURN)">

<!ENTITY % expression "(AST_IDENTIFIER | AST_INTEGER_CONSTANT | AST_CHAR_CONST
                        | AST_STRING | AST_CALL | AST_ASSIGN | AST_OR | AST_AND
                        | AST_EQ | AST_LT | AST_ADD | AST_SUB | AST_MUL
                        | AST_DIV | AST_UNARY)">

<!ELEMENT AST_TUNIT               (AST_EXT_DECL)>
<!ELEMENT AST_EXT_DECL            (AST_FUNC_DEF | AST_DECLARATION)>
<!ELEMENT AST_FUNC_DEF            (%typespec;, AST_DECLARATOR, AST_COMPOUND)>
<!ELEMENT AST_DECL_LIST           (AST_DECL_LIST?, AST_DECLARATION)>
<!ELEMENT AST_DECLARATION         (%typespec, %declarator;)>

<!ELEMENT AST_TYPE_VOID           EMPTY>
```

```
<!ELEMENT AST_TYPE_CHAR            EMPTY>
<!ELEMENT AST_TYPE_INT             EMPTY>

<!ELEMENT AST_DECLARATOR           (AST_IDENTIFIER)>
<!ELEMENT AST_POINTER_DECL         (%declarator;)>
<!ELEMENT AST_FUNC_DECL            (%declarator;, AST_PARAM_LIST?)>

<!ELEMENT AST_PARAM_LIST           (AST_PARAM_LIST?, AST_PARAM_DECL)>
<!ELEMENT AST_PARAM_DECL           (%typespec;, %declarator;)>

<!ELEMENT AST_STATEMENT_LIST       (AST_STATEMENT_LIST?, %statement;>

<!ELEMENT AST_STATEMENT            (%expression;)>
<!ELEMENT AST_IF                   (%statement;, %statement;, (%statement;)?)>
<!ELEMENT AST_WHILE                (%statement;, %statement;)>
<!ELEMENT AST_GOTO                 (AST_IDENTIFIER)>
<!ELEMENT AST_STATEMENT_WITH_LABEL (AST_IDENTIFIER, %statement;)>
<!ELEMENT AST_RETURN               (%expression;)?>
<!ELEMENT AST_COMPOUND             (SYM_LOCAL,
                                    AST_DECL_LIST, AST_STATEMENT_LIST?)>

<!ELEMENT SYM_LOCAL (TYPEREP_PRIM | TYPEREP_FUNC | TYPEREP_PTR)*>

<!ELEMENT AST_INTEGER_CONSTANT     #PCDATA>
<!ELEMENT AST_CHAR_CONST           #PCDATA>
<!ELEMENT AST_STRING               #PCDATA>
<!ELEMENT AST_CALL                 (%expression;, AST_ARG_LIST?)>
<!ELEMENT AST_ASSIGN               (%expression;, %expression;)>
<!ELEMENT AST_OR                   (%expression;, %expression;)>
<!ELEMENT AST_AND                  (%expression;, %expression;)>
<!ELEMENT AST_EQ                   (%expression;, %expression;)>
<!ELEMEMT AST_LT                   (%expression;, %expression;)>
<!ELEMENT AST_ADD                  (%expression;, %expression;)>
<!ELEMENT AST_SUB                  (%expression;, %expression;)>
<!ELEMENT AST_MUL                  (%expression;, %expression;)>
<!ELEMENT AST_DIV                  (%expression;, %expression;)>
<!ELEMENT AST_UNARY                (%expression;)>

<!ELEMENT AST_ARG_LIST             (AST_ARG_LIST?, %expression;)>
<!ELEMENT AST_IDENTIFIER           #PCDATA>
<!ELEMENT AST_ERROR                EMPTY>
<!ATTLIST AST_UNARY                op (addr|plus|minus|deref|not) #REQUIRED>
<!ELEMENT AST_NULL                 EMPTY>
<!ELEMENT AST_ERRORMSG    #PCDATA>
<!ELEMENT AST_WARNINGMSG    #PCDATA>
```

参考文献

[ABC+] Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Arnaud Le Hors, Gavin Nicol, Jonathan Robie, Texcel Research, Peter Sharpe, Bill Smith, Jared Sorensen, NovellRobert Sutor, Ray Whitmer, and iMallChris Wilson. *Document Object Model (DOM) Level 1 Specification Version 1.0*. WWW Consortium (W3C). `http://www.w3.org/TR/1998/ REC-DOM-Level-1-19981001/`.

[ADL+99a] Gerald Aigner, Amer Diwan, David L.Heine, Monica S. Lam, David L. Moore, Brian R. Murphy, and Constantine Sapuntzakis. *The Basic SUIF Programming Guide*. The Portland Group, Inc., 1999.

[ADL+99b] Gerald Aigner, Amer Diwan, David L.Heine, Monica S. Lam, David L. Moore, Brian R. Murphy, and Constantine Sapuntzakis. *The SUIF Program Representation*. The Portland Group, Inc., 1999.

[BF02] Anasua Bhowmik and Manoj Franklin. A General Compiler Framework for Speculative Multithreading. In *SPAA'02*, pp. 10–113, August 2002.

[BPSMa] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler and. *Extensible Markup Language (XML) 1.0 (Second Edition)*. WWW Consortium (W3C). `http://www.w3.org/TR/2000/ REC-xml-20001006`.

[CD] James Clark and Steve DeRose. *XML Path Language (XPath) Version 1.0*. WWW Consortium (W3C). `http://www.w3.org/TR/xpath`.

[Cla] James Clark. XP. `http://www.jclark.com/xml/xp/`.

[DJG+] Steven DeRose, Ron Daniel Jr., Paul Grosso, Eve Maler, Jonathan Marsh, and Norman Walsh. *XML Pointer Language (XPointer)*. WWW Consortium (W3C). `http://www.w3.org/TR/xptr/`.

[exp] expat. `http://expat.sourceforge.net/`.

[Fou] Inc. Fourthought. 4suite. `http://www.4suite.org`.

[Fre] Free Software Foundation. *GNU Compiler Collection*. `http://gcc.gnu.org`.

[Gro] The Stanford SUIF Compiler Group. *SUIF*. `http://suif.stanford.edu/`.

[IMMA02] I.Kadayif, M.Kandemir, M.J.Irwin, and A.Sivasubramaniam. EAC:A Compiler Framework for High-Level Energy Estimation and Optimization. In *DATE 2002*, pp. 436–442, 2002.

[Inc92] SPARC International Inc., editor. *The SPARC Architecture Manual Version 8*. Prentice-Hall, Inc., 1992.

[lib] The xml c library for gnome(libxml). `http://xmlsoft.org/`.

[Lut98] Mark Lutz. Python 入門. オーム社, 1998. 村山 敏夫 訳, 飯坂 剛一 監訳.

[Org] The Organization for the Advancement of Structured Information Standards [OASIS]. *RELAX NG Specification*. `http://www.oasis-open.org/committees/relax-ng/spec-20011203.html`.

[Pyt] Python. `http://www.python.org`.

[PyX] XML package for Python. http://pyxml.sourceforge.net http://www.python.org/sigs/xml-sig/.

[SAX] Simple api for xml(SAX). `http://sax.sourceforge.net/`.

[Ste01] W.Richard Stevens. 詳解 UNIX プログラミング. Pearson Education Japan, 2001. 大木 敦雄 訳.

[WWWa] WWW Consortium (W3C). *XML Schema Part 0: Primer*. `http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/`.

[WWWb] WWW Consortium (W3C). *XML Schema Part 1: Structures*. `http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/`.

[WWWc] WWW Consortium (W3C). *XML Schema Part 2: Datatypes*. `http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/`.

[WWWd] WWW Consortium (W3C). *XSL Transformations (XSLT)*. `http://www.w3.org/TR/xslt`.

[Xer] Xerces. `http://xml.apache.org/xerces2-j/index.html`.

[引地 02] 引地信之, 今井正治, 武内良典. GCC を用いた ASIP 用リターゲッタブルコンパイラジェネレータの開発. 平成 13 年度未踏ソフトウエア創造事業, 情報処理振興事業協会 IPA, 2002. `http://www.ipa.go.jp/NBP/13nendo/reports/explorat/retarget/retarget.pdf`.

[小林 00] 小林宏高, 松岡聡, 丸山冬彦, 早田恭彦, 志村浩也. OpenJIT フロントエンドシステムの設計. 情報処理学会 論文誌, Vol. 41, No. SIG2 (PRO 6), pp. 1–12, 2000.

[斉木] 斉木晃治. XCC-XML. `http://www.jaist.ac.jp/ k-saiki/xcc-xml/xcc-xml.html`.

[中西 01] 中西恒夫, 福田晃, 平尾智也. 組込みシステムのためのコンパイラフレームワークライブラリ. 平成 12 年度未踏ソフトウエア創造事業, 情報処理振興事業協会 IPA, 2001. `http://www.ipa.go.jp/NBP/12nendo/ 12mito/mdata/5-2gh/5-2gh.pdf`.