課題研究報告書

保守性の高いファームウェア開発を支援するプログラミング環境の構築

北陸先端科学技術大学院大学
情報科学研究科

栗林 大樹

2016 年 3 月

課題研究報告書

保守性の高いファームウェア開発を支援するプログラミング環境の構築

1210904 栗林 大樹

主指導教員　田中 清史

審査委員主査　田中 清史
審査委員　井口 寧
審査委員　金子 峰雄

北陸先端科学技術大学院大学
情報科学研究科

提出年月: 2016 年 2 月

概 要

近年，組込みシステムのファームウェアのレビューやテストの一部をソフトウェア解析ツールで代替することが増えてきている．プログラマがプリプロセス前のソースコードを調査し，複数の製品を意識しながら開発を行っているのに対し，既存の C 言語用ソフトウェア解析ツールはプリプロセスを実行した後のソースコードを解析しているため，参照しているソースコードに乖離が生じている．本研究では C 言語の静的コード解析において，プリプロセス前のソースコードに対する解析を行う手法を提案し，保守性や可読性の高いソースコードを記述するための検査を行う静的コード解析ツールを作成した．組込みシステムで使用されるデバイスドライバ対して静的コード解析を行い，保守性や可読性を低下させるコード片が検出できることを確認した．

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　研究の背景と目的

競争力のある製品を生み出すために，組込みシステムは多機能でありながら短期間での開発が求められる．組込みシステム上で動作するソフトウェア（以降はファームウェアと記述する）も，ハードウェアの高性能化に従って，より多くの複雑な機能が実装できるようになり，その規模は増大し続けている．

大規模なファームウェアを短期間で生産するために，既存のファームウェアを再利用し，機能の追加や改良といった部分的な修正をすることで新製品の開発を行う派生開発（図1.1）が主流になっている．派生開発では大部分の機能は一度作成されると長期間使用されることになる．製品間で共通の機能は共通部品として整備，保守されるが，そのためには製品毎の微妙な差異や依存するモジュールのバージョンの違いなどを吸収する必要が生じることがあり，ファームウェアをビルドする際に条件分岐[1]で解決される．また，多人数のプログラマが短期間の開発を繰り返すことになるため，流動的な人員調整が頻繁に行われることになり，プログラマはモジュールの全体像と修正の影響範囲を短期間で把握することが求められる．

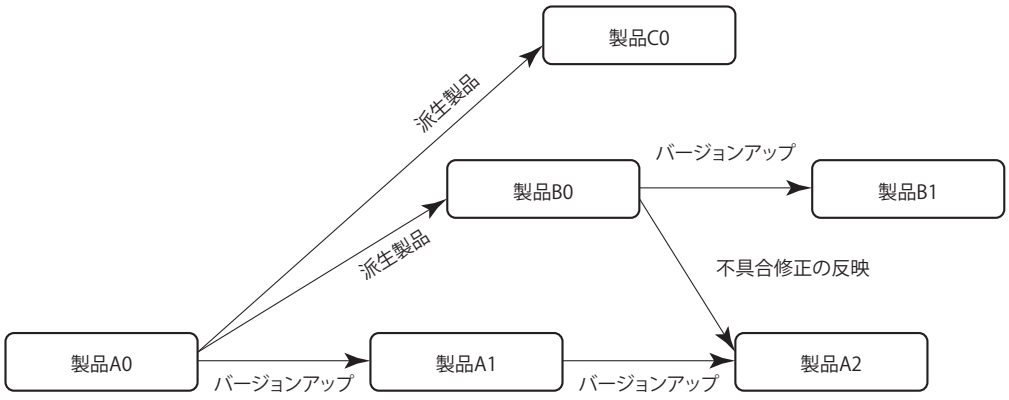


図 1.1: 派生開発の例

その一方で組込みシステムは一度出荷されてしまうと不具合を修正することが困難である．近年，パーソナルコンピュータ上で実行される OS やアプリケーションは，インター

[1] C 言語ではプリプロセッサ指令やリンク対象ファイルを切り替えることによって実現する．コンパイルスイッチとも呼ばれる．

ネット経由でのアップデート機能を備えていることが多く，機能の追加や不具合の修正が比較的容易に行えるのに対し，組込みシステムのファームウェアはその運用の都合上，必ずしもインターネットに接続できるとは限らない．重大な欠陥が発生した際は交換や回収などの対応をする必要があり，金銭的な損害の問題だけでなく社会的な信頼を失うことにもつながるため，高度な信頼性が要求される．

不具合の発生を防止するためにレビュー（e.g. インスペクション，ウォークスルー）やテスト（e.g. 単体テスト，結合テスト，回帰テスト）を実施するのが一般的だが，大規模開発ではソースコードが複雑化するため，人手によるコードレビューやテストが開発期間の増加に直結する．近年ではソフトウェア解析ツールを導入し，人手によるコードレビューやテストを実施する前の段階で，ソフトウェア解析ツールによる自動検査を実施し，レビューやテストの一部を自動化する手法が一般的になってきている．

派生開発において機能の追加や不具合の修正を行うには，ソースコード修正箇所を特定し，修正が他の機能に悪影響を与えないかを調査する．円滑に作業を進めるためには，調査対象のソースコードが理解しやすい（可読性が高い）ことが望ましい．そこでプロジェクト毎にコーディング時の取り決め（コーディング規約）を作成するのが一般的である．

ファームウェア（OS, デバイスドライバ，専用アプリケーション）の開発には，汎用プログラミング言語である C 言語が使用されることが多い．C 言語は高級言語に分類されるが，高級言語設計の黎明期に設計された言語であり後発の高級言語（e.g. Perl, Ruby, Python, Java, Haskell）と比較すると表現能力が低い．しかし，大抵の高級言語はガベージコレクションや仮想マシン，インタプリタ，正規表現[2]，クロージャ（ラムダ式）[3] などを備えており，メモリの管理や I/O の制御を厳密に記述することが求められる組込みシステムのファームウェア開発には適していない．以上のことからファームウェアの開発には今後も C 言語が選択されることが多いと予想される．

一般に C 言語のソフトウェア解析ツールは，C 言語のソースコードについてプリプロセスを行ってから構造を解析するため，プリプロセッサディクティブによる条件分岐を含んだソースコードの解析は行われない．プログラマがプリプロセス前のソースコードに対し調査，修正，レビュー等を行うのに対し，ソフトウェア解析ツールはある１つのケース（製品）に対して解析を行っており乖離が生じていることになる．よって，本研究ではプリプロセス前の段階のソースコードを解析する手法を研究する．

なお，本論文はソースコード解析ツールによって検出された問題のすべてを修正すべきということを主張するものではない．ソースコード解析ツールによって規則に沿わない箇所を明確化し，意図的な逸脱であればその理由を文書化してプロジェクト内で共有することが重要である．

[2] 正規表現の機能が言語の実行環境に組み込まれていることも多い．

[3] クロージャが参照している変数は解放できないため，変数をスタックに積むことができない．大抵はガベージコレクションと共に用いられる．

1.2 研究方法

プリプロセス前の段階でソフトウェアの解析を行うツールの設計し，保守性や可読性を低下させるコード片を検出するための検査項目を実装する．作成したツールを組込みシステムのデバイスドライバに対して実行し，その結果を考察する．

1.3 本論文の貢献

プリプロセッサを含めた状態でのソフトウェア解析は一般に不可能だが，本論文では現実的な範囲の制限を設けることで，プリプロセッサを含めたソースコードに対して，有効な静的コード解析が行えることを示す．これによりソースコードの読み手（プログラマ，レビューア）が参照しているコードをそのまま解析することができ，組込みシステムのファームウェア開発においてコンパイルスイッチによって一部の製品にのみ含まれるコード片も同時に検査することが可能となる．

1.4 本論文の構成

本論文の構成は次の通りである．第２章では C 言語の規格や規約，既存のソフトウェア解析ツール等を調査した結果を紹介し，本研究との関連を述べる．第３章では本研究で作成した静的コード解析ツールの設計について説明し，第４章では実装した解析ルールについて説明する．第５章では組込み Linux のデバイスドライバの一部を対象として，静的コード解析ツールを実際に使用し，どのような問題が検出できるかを例示する．最後に第６章にてまとめと今後の課題について述べる．

第2章　関連研究

本章ではC言語の規格や既存のソフトウェア解析の研究等を調査し，本研究の位置付けや関係を述べる．

2.1　C言語の規格

C言語には幾つかのバージョンが存在するが，現在は主にISO[1]/IEC[2]が標準化および改定を行っており，日本ではJIS[3]が翻訳版[4][6]を作成している．

1978年に出版された書籍，プログラミング言語C[1]が事実上最初の標準であり，1989年にISO/IECとANSI[4]によってC89/C90[2][3]が正式に標準化された．その後，ISO/IECによってC99[5]とC11[7]の2度の改定が行われている．

ファームウェア開発で使用される代表的なコンパイラと，そのコンパイラが対応しているC言語の規格を表2.1に示す．現段階では主要な組込み向けのコンパイラの中にもC11に対応していないものがあるため，移植性の問題から当面はC89/C90またはC99の範囲内のコーディングを行うことが望ましい．よって本研究ではC89/C90とC99を主眼において静的コード解析のルールを検討する．

表2.1: ファームウェア開発で使用されるコンパイラとC言語の規格

コンパイラ	対象CPU	対応している規格
ARM Compiler(ADS, RVDS)	ARM	C90, C99
HEW [5]	SuperH	C89, C99
Wind River Diab Compiler	ARM, MIPS, PowerPC, IA etc.	ANSI
Intel C++ Compiler	ATOM	C90, C99, C11(一部)
GCC [6]	多数	C89, C99, C11

[1]ISO: 国際標準化機構, International Organization for Standardization
[2]IEC: 国際電気標準会議, International Electrotechnical Commission
[3]JIS: 日本工業規格, Japanese Industrial Standards
[4]ANSI: 米国国家規格協会, American National Standards Institute
[6]HEW: High-performance Embedded Workshop
[6]GCC: GNU Compiler Collection

2.2 コーディング規約

可読性や安全性，移植性の高いソースコードを作成するためにいくつかのコーディング規約が存在する．ファームウェア開発ではC言語のコーディング規約には以下のものがある．

- ISO/IEC 25010, JIS X 012901
  ISO/IEC及びJISでのソフトウェア品質モデルの定義が含まれる．
- MISRA-C:1998, MISRA-C:2004, MISRA-C:2012
  自動車業界における組込みシステムの安全性と移植性を確保することを目的とした規格．
- SEI CERT C Coding Standard
  カーネギーメロン大学 ソフトウェア工学研究所 (SEI) が作成したC言語のコーディングスタンダードであり，セキュアなコーディングを行うための作法を定めている．
- Recommended C Style and Coding Standards
  AT&T Indian Hill研究所が作成した一貫性，移植性を確保することを目的としたコーディングスタンダード．
- GNU coding standards
  GNUプロジェクトで使用されている一貫性，移植性，堅牢性を確保することを目的としたコーディングスタンダード．

組込み業界では特にMISRA[7]が作成したC言語のためのソフトウェア設計標準規格であるMISRA Cが広く使用されている．MISRA-CにはMISRA-C:1998,MISRA-C:2004,MISRA-C:2012の3つのバージョンが存在する．MISRA-C:1998,MISRA-C:2004はC89/C90を対象として作成された規格であり，MISRA-C:2012はC99に対応した規格である．よって本研究でも静的コード解析ツールの検出ルールに関して，MISRA-Cの規約の中から特に有効であるものをピックアップして実装する．

2.3 ソフトウェア解析

ソフトウェア解析には大別して静的コード解析と動的プログラム解析がある．静的コード解析はプログラムを実行せずに解析する手法であり，動的プログラム解析はプログラムを実際に実行して解析する手法である．本研究の対象は静的コード解析であるため，既存の静的コード解析ツールの調査を行った．静的コード解析ツールとその特徴を表2.2に示す．

[7] MISRA: Motor Industry Software Reliability Association

表 2.2: 静的コード解析ツール

名称	製作者	特徴
QA・C	Programming Research	静的コード解析の一種であるデータフロー解析を行う．MISRA C に適合しているかを評価するためのアドオンも存在する．
Coverity	synopsys	プロシージャ間データフロー解析や統計解析，ブール値充足可能性ソルバ等を使用して解析を行う．
CODESONAR	GRAMMATECH	コントロールグラフ，コールグラフを作成して静的に解析を行う．関数，ファイル間の解析が可能．
C++test	PARASOFT	静的コード解析，動的プログラム解析，単体テストを行うツール．静的コード解析では MISRA-C などのコーディングルールチェックや，フロー解析によるメモリリークなどの検証を行う
Understand	scitools	コントロールグラフの視覚的な表示やコード行数，サイクロマティック複雑度等の診断を行う．プログラムの内容の解析は行わない．
BLAST	Dirk Beyer ほか	C プログラムが指定した性質を満たすかどうかを検査する．指定したラベルに到達するかや，アサーション (assert()) の式を満たさないケースが存在するかを静的に検査する．
C99parser	Mario Konrad	プリプロセス済みの C プログラムを入力してコールグラフを作成する．
CCured	GEORGE C. NECULA ほか (University of California, Berkeley)	プリプロセス済みの C プログラムを静的に解析し，型安全でメモリエラーが発生しないようにランタイムチェックを自動で挿入する．

次ページに続く

表 2.2 – 静的コード解析ツール（続き）

名称	製作者	特徴
cppcheck	Daniel Marjamm̈ki	プリプロセス済みのCプログラムのコントロールフローグラフを作成し，メモリリークや配列の範囲外のアクセスを検出する．ただし，確実に問題がある場合以外は問題を報告しない.
Cqual	Jeff Foster ほか	Cプログラムの型を独自のアノテーションで修飾することで，アノテーションで修飾された型の伝搬をトレースしアノテーションが正しいかどうかを検査する.
CScout	Diomidis D. Spinellis	リファクタリングを支援するために，コールグラフや変数の一覧などを作成する.
Flawfinder	David A. Wheeler	Cプログラム内（文字列リテラルとコメントを除く）を検索し，経験的に脆弱性問題を起こしやすい関数が使用されていないか調べる（字面上の検索であり構文解析は行わない）.
Fortify SCA	Hwelet Packard Enterprise	静的アプリケーションセキュリティテストを行うソフトウェア.
MOPS	Hao Chen	Unixシステムコール等の誤った使用による脆弱性を検出する．検出するルールは有限状態オートマトンで表現し，終了状態に到達するかを調べる.
PGRelief C/C++	富士通ソフトウェア	MISRA-C等，各種コーディングガイドラインに沿っているかどうかを評価する.
PolySpace Code Prover	MathWorks	静的コード解析と形式手法を用いて，オーバーフローなどのランタイムエラーが無いことを証明する.
Sparse	Linus Torvalds	Linux Kernelに特化されたコーディングミスを検出するツール．例えばユーザ空間とカーネル空間のポインタの混在などを検出する.

次ページに続く

表 2.2 – 静的コード解析ツール（続き）

名称	製作者	特徴
Splint	Inexpensive Program Analysis at the University of Virginia	変数や関数に特殊なコメント（アノテーション）を付加することで静的コード解析の精度を向上することができる.
Visual Studio	Microsoft	統合開発環境であるがコード分析機能が搭載されている．バッファオーバーランやメモリリークなどを検出する.
Clang Static Analyzer	LLVM org	LLVM コンパイラのフロントエンドである Clang の一部であり，未初期化変数の検出などの基本的な検査やセキュリティ上の検査を行う．検査項目の開発マニュアルが公開されており，ユーザが検査項目を追加することができる.
AdLint	オージス総研	ruby 言語で開発された静的コード解析ツールであり 720 種類の検査項目が実装されている.

2.4 字句解析・構文解析ツール

静的コード解析を行うためにはまず，C 言語で記述されたプログラムに対して字句解析及び構文解析を行い構文木を作成する必要がある．字句解析と構文解析を行うツールを調査した結果を以下に示す.

1. lex/yacc, flex/bison
   字句解析と構文解析を行うもっとも標準的なツール．lex は正規表現で字句規則を記述することで C 言語で記述された字句解析器を生成し，yacc は拡張 BNF で構文規則を記述することで C 言語で記述された構文解析器を生成する．yacc は LALR 法[8] を使用したパーサジェネレータである．flex/bison は lex/yacc の再実装版であり基本的な機能は変わらない.

2. ML-yacc/ML-lex
   ML 言語用の yacc/lex であり，ML のライブラリとして実現されているため，構文規則も ML 言語で記述可能である.

[8] LALR : Lookahead LR の略であり先読みを行う LR 法である．LR 法は入力を左 (Left) から解釈し，右端を導出 (Rightmost derivation) するボトムアップ構文解析手法の一種である.

3. Happy
   関数型言語 Haskell 用のパーサジェネレータであり，yacc と同等の機能を持つ．yacc と同様に BNF で構文規則を記述する．Happy を使用した C 言語用のパーサとして language-c[13] というライブラリが公開されている．[9]

4. JavaCC(Java Compiler Compiler)
   Java 言語用のパーサジェネレータであり，拡張 BNF で構文規則を記述することで LL 法の構文解析器を出力する．

5. Parsec
   関数型言語 Haskell 用のパーサコンビネータライブラリであり，構文規則を Haskell 言語で記述する．トップダウン型の構文解析を行うが，バックトラックが使用できるため，無限先読みの文脈依存文法も解析可能である．

C 言語の構文は文脈自由文法に属するため LR 法によるボトムアップ構文解析を行うのが一般的であるが，静的コード解析の用途ではトップダウンの構文解析も可能（詳細は次章で説明する）と判断した．トップダウン構文解析はボトムアップ構文解析と比べて効率は劣るが，直感的に実装できるため，本研究ではトップダウン型の構文解析を行う Parsec を採用する．

[9]ただしプリプロセスは gcc(-E オプション) などの外部ツールを必要とする

第3章　静的コード解析ツールの設計

本研究ではC言語で記述されたソースコードの検査を行うツールを作成した．本章ではその設計について述べる．

3.1　設計方針

最初にC言語を解釈するソフトウェアであるプリプロセッサ，コンパイラ，リンカの処理の流れ（図3.2）について説明する．C言語で記述されたソースコードはまずプリプロセッサによって処理される．プリプロセッサはソースコード中のプリプロセス指令（e.g. #define, #include, #ifdef）を解釈し，ヘッダファイルの検索および展開や，#defineで定義された識別子を文字列に置き換えるなどの処理を行う．プリプロセッサによって処理されたソースコード（プリプロセス済みソースコード）はコンパイラによって処理される．コンパイラは文字の集合であるプリプロセス済みソースコードを字句解析器に与えトークンの列を取り出す．構文解析器はトークン列から中間コードを生成し，意味解析器によって型や宣言・定義の正しさが検査される．中間コードに対して最適化やコード生成が施されてオブジェクトファイルが生成される．これらの工程で作成した複数のオブジェクトファイルやライブラリをリンカによって結合することで実行可能ファイルが完成する．以上から，静的コード解析はプリプロセッサとコンパイラの一部（字句解析，構文解析，意味解析）とほぼ同様の処理を行うと言える．静的コード解析では意味解析器に相当する処理として検査項目を実装する．

第2章で調査した静的コード解析ツールは主にプリプロセス済みソースコードを検査し，プリプロセス前のソースコードやヘッダファイルは対象とされない．プログラマやレビューアが参照しているのはソースコードであり，静的コード解析ツールが参照しているプリプロセス済みソースコードとは異なっている．これはプリプロセス前の段階のソースコードの構文解析が不可能であることが原因である．図3.1のように，波括弧の対応関係が条件によって変化するような構文を受理するには，膨大な数の状態が必要になり実用的な構文解析器を作成することができない．

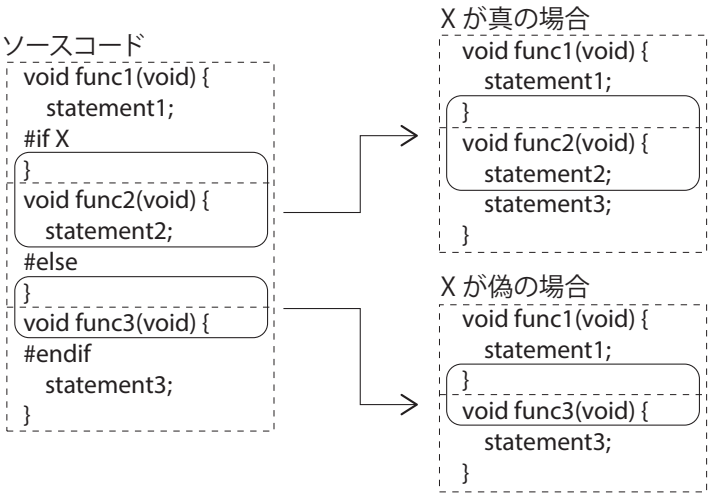


図 3.1: 構文解析が困難であるソースコードの例

実際のファームウェア開発においては，図 3.1 のソースコードのような記述は，可読性および保守性の観点から可能な限り避けるべきである．

MISRA-C でもマクロに関する制限がいくつか存在する．以下に MISRA-C の Rule 90(required) を引用する．

> C macros shall only be used for symbolic constants, function-like macros, type qualifiers, and storage class specifiers.

上記ルールに従い，マクロが記号定数，関数型マクロ，型修飾子，記憶クラス指定子のいずれかとして使用されるのであれば，前述のような問題は発生しない．本研究では幾つかの制限を設けることで，読み手が実際に参照しているプリプロセス前のソースコードをそのまま解析する方法を検討する．

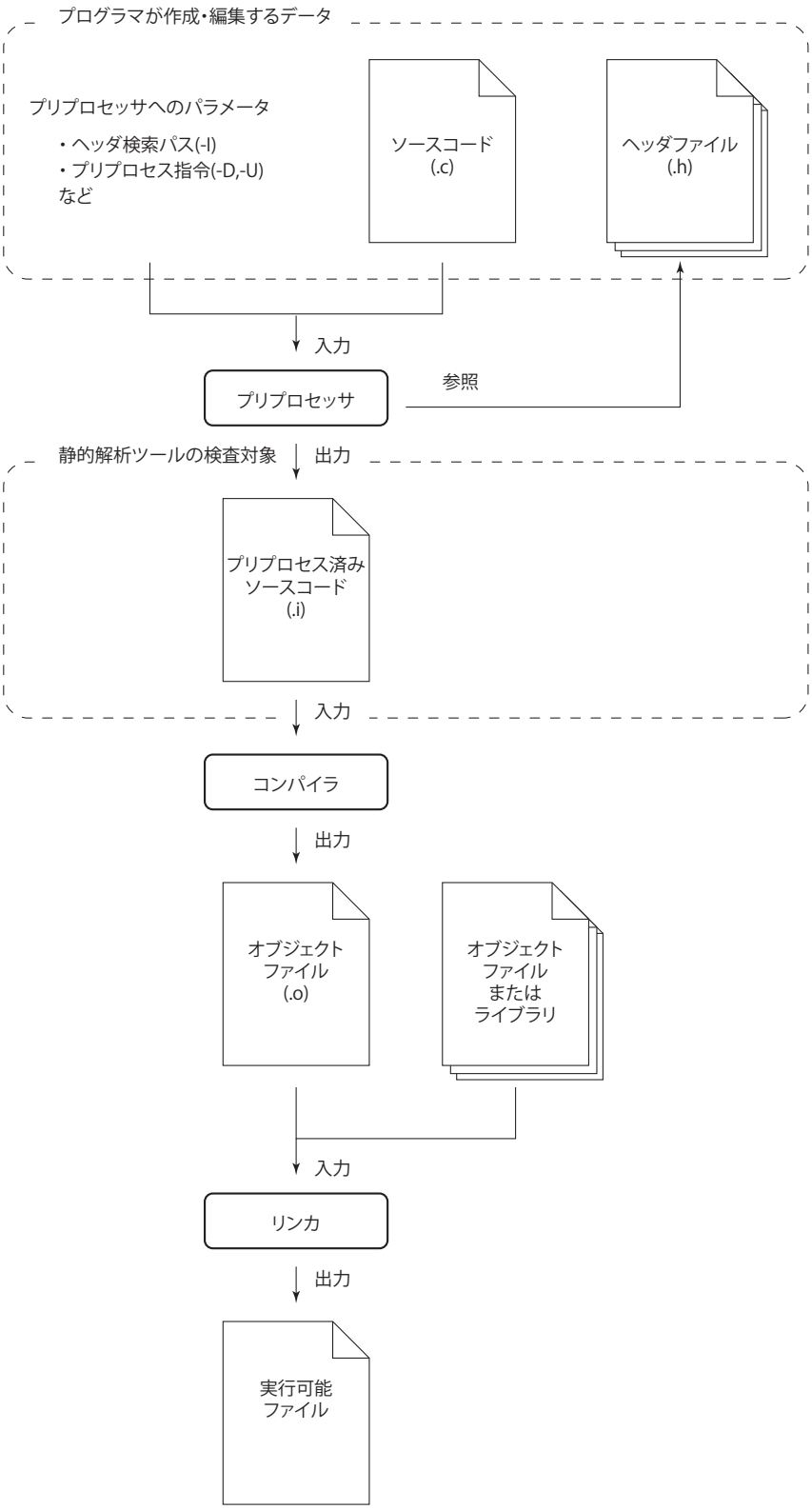


図 3.2: プリプロセッサ，コンパイラ，リンカの処理の流れ

3.1.1 コンパイラ拡張機能

プログラマが最適化の指標を与えたりハードウェアの特殊な機能を扱うために，各コンパイラが独自の拡張機能を備えていることが多い．例えば gcc(GNU Compiler Collection) の属性付加 (__attribute__) やインラインアセンブラ (asm)，セミコロンによる複合式 (例 3.1.1) などが独自の拡張機能である．しかし，独自の拡張機能が出現すると C 言語の規格を逸脱してしまうため，静的コード解析の構文解析器がソースコードを受理することができない．本研究で作成する静的コード解析ツールでは以下の方針をとる．

1. __attribute__((属性)) のように字句並びに対するパターンマッチで検出できるものは，拡張機能が使用されていることを警告して構文解析対象から取り除く．
2. 複合式のように構文解析が必要なものは構文を拡張してパーサで受理できるようにし，機能拡張が行われていることを警告する．

ただし，独自の拡張機能はコンパイラ毎に多数存在するため，本研究では必要に迫られた場合のみ実装することとする．

例 3.1.1 (複合式)
GNUC では以下の様に括弧で括られた複合文は式として扱うことができる．以下の場合，括弧で括られた式の値 result は変数 c と一致し 30 となる．

```
int result = ({
    int a = 10;
    int b = 20;
    int c = a + b;
    c;
});
```

3.2 設計の概要

本節では本研究で作成した静的解析ツールの処理の概要を述べ，次節から各工程の詳細を説明する．まず，入力されたソースコードに対して字句解析を行いトークン列を取り出す．このとき，プリプロセッサ指令 (e.g. #if, #include, #define) の存在する行を 1 つのトークンとして扱う．次にトークン列からプリプロセッサ指令を取り出し，その有効範囲に含まれるトークン列にプリプロセッサ指令の情報を付加する．プリプロセッサ指令の情報はそのトークンが所属している条件（コンパイルスイッチ）とトークンから参照可能なデファインのリストの 2 種類がある．字句解析後のトークン列からプリプロセッサ情報付きトークン列を作成する例を図 3.3 に示す．

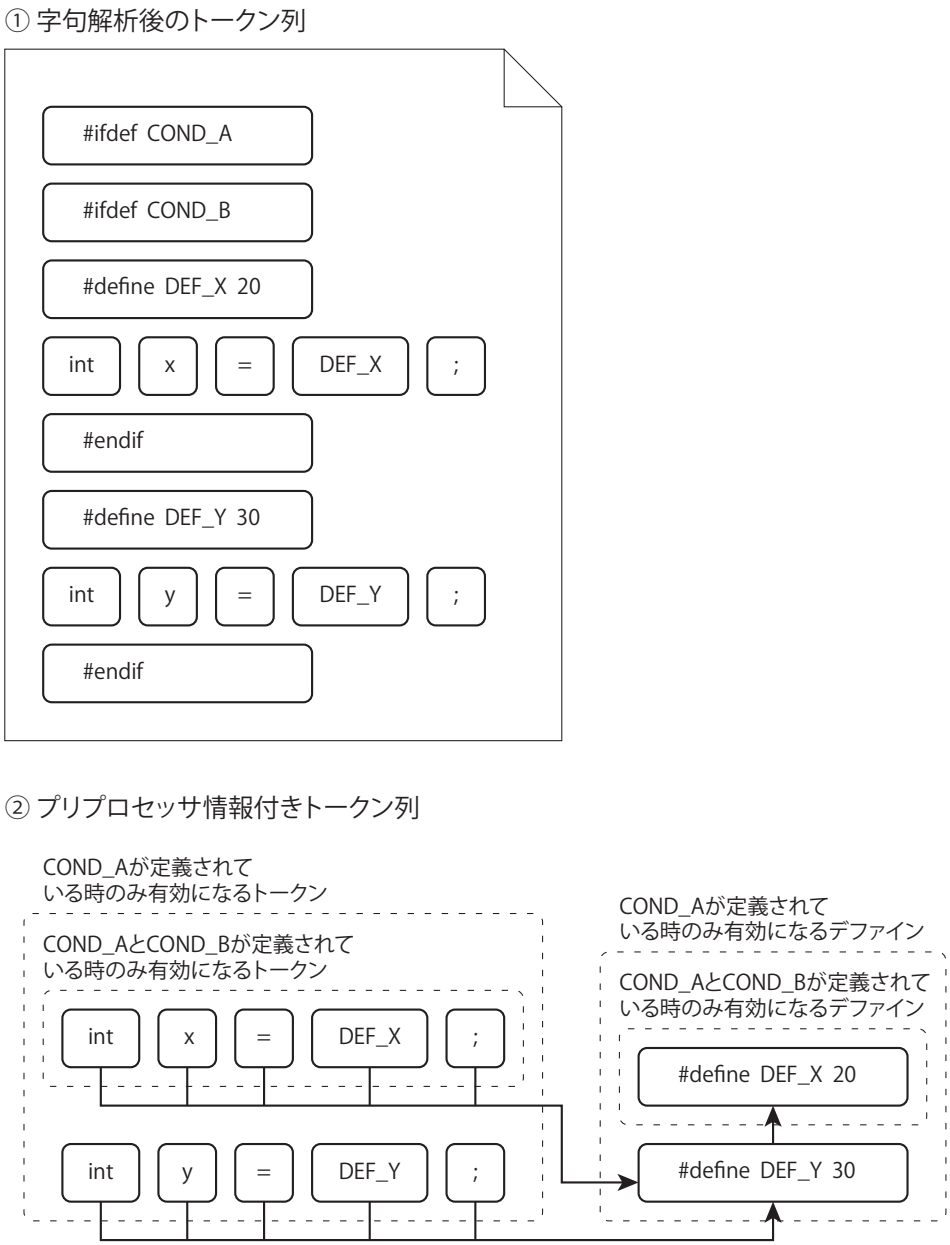


図 3.3: プリプロセッサ情報付きトークン列を作成する例

また，インクルードされているヘッダファイルを読み込み，字句解析とプリプロセッサ指令の処理を行い，インクルードディレクティブが記述されていた箇所にトークン列を挿入する．

出力されたトークン列に対して構文解析を行い，構文木を作成する．構文木を辿りながら各検査を実施して警告情報を出力する．本研究で作成した静的解析ツールの処理の流れを図 3.4 に示す．

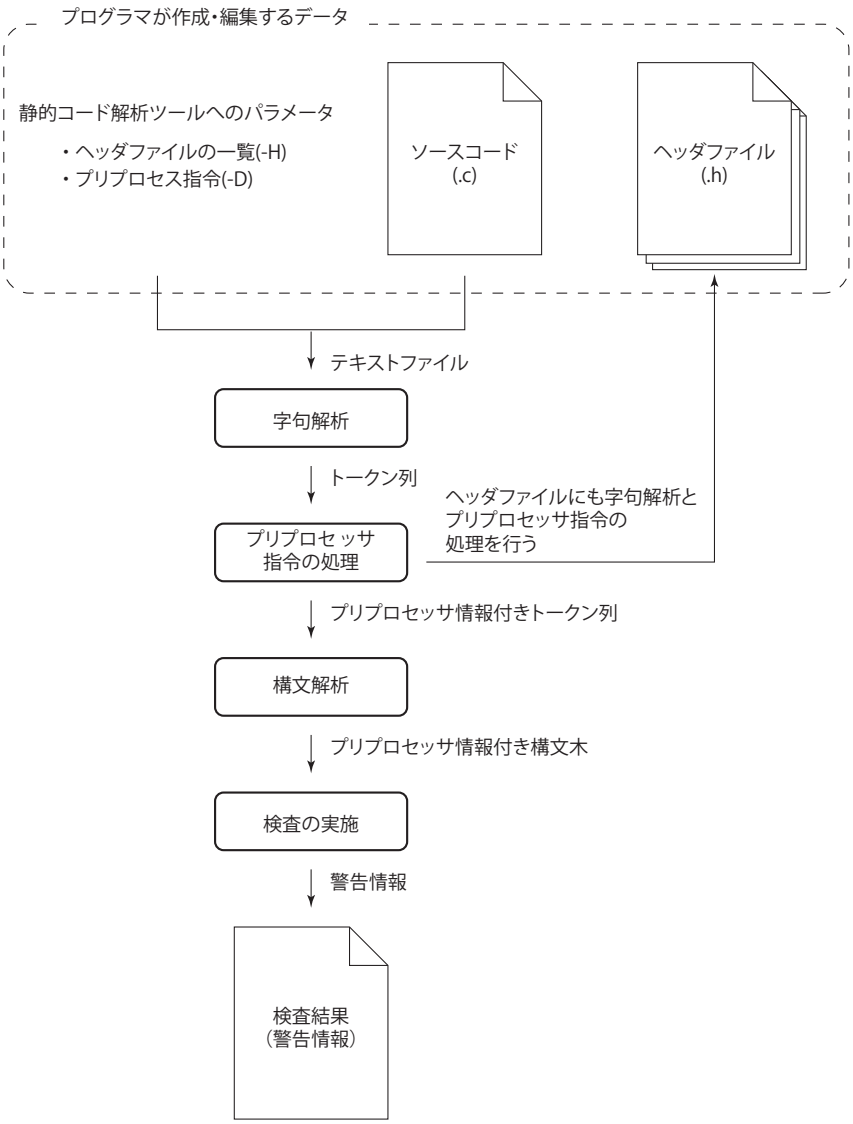


図 3.4: 本研究で作成した静的解析ツールの処理の流れ

3.3 字句解析器

本研究で作成する静的コード解析ツールではC言語の予約語，識別子，区切り文字，リテラル（定数，文字列），プリプロセッサ指令をトークンとして扱う．トークンは静的コード解析ツールの警告表示に使用するために位置情報を持つ．なお，コメントの除去は字句解析の空白文字を読み飛ばす際に行う．静的コード解析ツールはトークンを検出した際，ファイル名，トークン位置（行，列）をトークンの情報として記録する．

Token BNF

⟨*token*⟩ ::= ⟨*token-directive*⟩
| ⟨*token-keyword*⟩
| ⟨*token-identifier*⟩
| ⟨*token-constant*⟩
| ⟨*token-string-literal*⟩
| ⟨*token-punctuator*⟩

表 3.1: token の構成要素

要素	説明	例
token-directive	プリプロセッサ指令．詳細は 3.3.1 節を参照せよ．	#define A 10
token-keyword	C 言語の予約語．詳細は表 3.2 を参照せよ．	int, struct, return
token-identifier	C 言語の識別子．英数字とアンダースコア (_) の組み合わせ（ただし先頭文字は数字以外）．universal character names [1] を使用することも可能．	_id1
token-constant	整数や小数といった定数リテラル．	0L, 3.14f, 0xFF, 1.23e-2
token-string-literal	ダブルクォーテーションで囲まれた文字列リテラル．	”hello world.”
token-punctuator	C 言語の記号．詳細は表 3.3 を参照せよ．	+, -, *, /

表 3.2: 予約語一覧

alignof	auto	break	case	char
const	continue	default	do	double
else	enum	extern	float	for
goto	if	inline	int	long
register	restrict	return	short	signed
sizeof	static	struct	switch	typedef
union	unsigned	void	volatile	while
_Align	_Bool	_Complex	_Imaginary	_Static_assert
_Thread_local	typeof [2]			

[1] universal character names: UTF-8 などの文字を埋め込むことができる．e.g. \u3042 はひらがなの”あ” を表す

[2] typeof 演算子は gcc 拡張である．

表 3.3: 記号一覧

%:%:	<<=	>>=	...	->	++	–	<<
>>	<=	>=	==	!=	&&	||	*=
/=	%=	+=	-=	&=	^=	|=	##
<:	:>	<%	%>	%:	[	]	(
)	{	}	.	&	*	+	-
~	!	/	%	<	>	^	|
?	:	;	=	,	#		

3.3.1 プリプロセッサ指令

本研究で作成する静的コード解析ツールは，プリプロセッサ指令の情報も構文木に取り込むため，字句解析ではトークンの一種として扱う．

⟨*token-directive*⟩ ::= ‘`#`’ ⟨*token-directive-sub*⟩

⟨*token-directive-sub*⟩ ::= ‘`if`’ ⟨*token*⟩+ ⟨*newline*⟩
| ‘`ifdef`’ ⟨*token*⟩ ⟨*newline*⟩
| ‘`ifndef`’ ⟨*token*⟩ ⟨*newline*⟩
| ‘`elif`’ ⟨*token*⟩+ ⟨*newline*⟩
| ‘`else`’ ⟨*newline*⟩
| ‘`endif`’ ⟨*newline*⟩
| ‘`include`’ ‘`<`’ ⟨*hChar*⟩+ ‘`>`’
| ‘`include`’ ‘`"`’ ⟨*qChar*⟩+ ‘`"`’
| ‘`define`’ ⟨*token*⟩ ‘`(`’ ⟨*token*⟩+ ‘`)`’ ⟨*token*⟩*
| ‘`undef`’ ⟨*token*⟩ ⟨*newline*⟩
| ‘`line`’ ⟨*token*⟩+ ⟨*newline*⟩
| ‘`warning`’ ⟨*token*⟩* ⟨*newline*⟩
| ‘`error`’ ⟨*token*⟩* ⟨*newline*⟩
| ‘`pragma`’ ⟨*token*⟩* ⟨*newline*⟩

ただし，hChar は改行 (newline) と ‘>’ を除く任意の文字，qChar は改行 (newline) と ‘”’ を除く任意の文字とする．

注意事項 1 C 言語のプリプロセッサ指令以外の字句解析では改行を空白文字の一種として読み飛ばすが，プリプロセッサ指令は行頭の#から改行 (newline) までが対象であり，改行は読み飛ばさない．

注意事項 2　BNF では表現されていないが，#define の識別子と括弧の間にスペースが入った場合は関数形式マクロとして解釈してはならない．以下に例を示す．

```
#define A(x) x  // 関数形式マクロの定義として解釈する
#define A (x) x // 関数形式マクロの定義として解釈してはならない
                // マクロ A の本体が (x) x である
```

3.4　プリプロセッサ指令の処理

前段の字句解析器から出力されたトークン列を受け取り，プリプロセッサ指令トークンを検索する．発見したプリプロセッサ指令トークンに表 3.4 に示す処理を行う．また，プリプロセッサ指令の処理を実行した場合に，コンパイルスイッチ情報 (#if, #ifdef, #ifndef, #elif) のリストとデファイン情報（デファイン名と本体のペア）のリストがトークンに付加される例を図 3.5 に示す．なお，トークンだけでなくデファイン情報もコンパイルスイッチ情報のリストを持つ．

表 3.4: プリプロセッサ指令トークンに対する処理

プリプロセッサ指令	処理内容
#if	#elif, #else, #endif が見つかるまで，後続のトークンにコンパイルスイッチ情報を付加する．
#ifdef	#elif, #else, #endif が見つかるまで，後続のトークンにコンパイルスイッチ情報を付加する．
#ifndef	#elif, #else, #endif が見つかるまで，後続のトークンにコンパイルスイッチ情報を付加する．
#elif	#else, #endif が見つかるまで，後続のトークンにコンパイルスイッチ情報を付加する．
#else	#endif が見つかるまで，後続のトークンに#if の否定のコンパイルスイッチ情報を付加する．
#endif	対応する#if, #elif, #else のコンパイルスイッチ情報付加を停止する．
#include	指定されたヘッダファイルを開き，字句解析とプリプロセッサ指令の処理を行い，出力されたトークン列を#include 指令トークン位置に挿入する．
#define	後続のトークンのデファイン情報リストにデファイン名とデファイン本体を追加する．
#undef	後続のトークンのデファイン情報リストからデファイン名を取り除く．
#line	警告を出力して#line トークンを取り除く．
#warning	何もせず#warning トークンを取り除く．
#error	何もせず#error トークンを取り除く．
#pragma	何もせず#pragma トークンを取り除く．
#(空)	何もせず#トークンを取り除く．

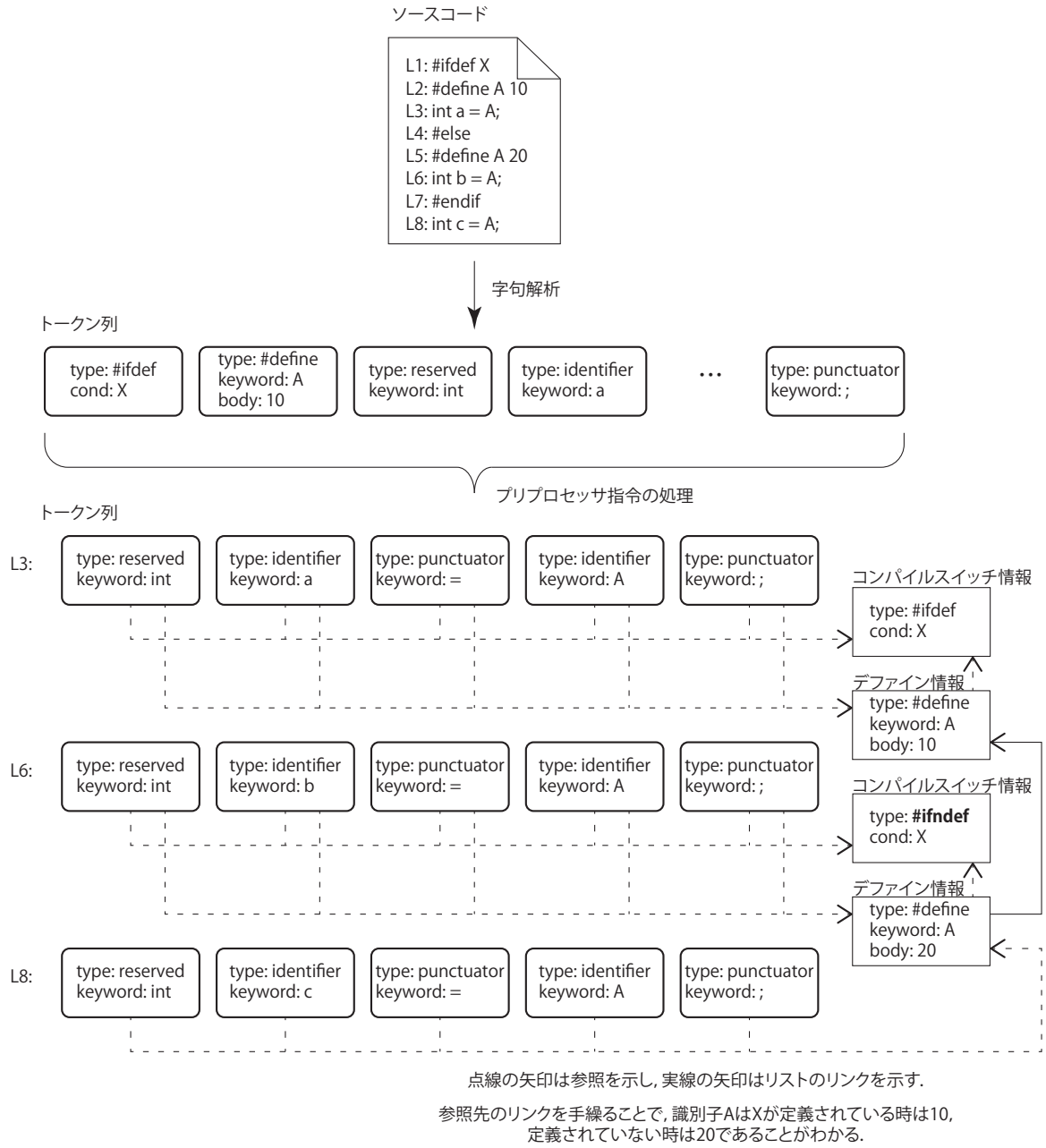


図 3.5: プリプロセッサ指令の処理の例

3.5 構文解析器

前段のプリプロセッサ指令の処理によって出力された，コンパイルスイッチ情報とデファイン情報が付加されたトークン列に対して構文解析を行い構文木を生成する.

3.5.1 構文木の生成

構文解析器（パーサ）は先頭のトークンからトークン列をスキャンしていき，そのパーサがもつ文法構造と合致した場合にはトークン列を消費して構文木（C 言語の文法構造の情報を持った木）を生成する．パーサは部分的なパーサの組み合わせで構成されており，部分的なパーサは部分的な構文木を生成する．

本研究で作成した静的解析ツールでは構文の検査に使用するため，構文解析器がどの構文を受理したかの情報 (トレース情報) をすべて構文木に保存している．トレース情報について代入式と加算減算のみが受理可能なシンプルなパーサを例に説明する．

assignment-expression の BNF

⟨*assignment-expression*⟩ ::= ⟨*id*⟩ '=' ⟨*additive-expression*⟩

additive-expression の BNF

⟨*additive-expression*⟩ ::= ⟨*id*⟩ '+' ⟨*additive-expression*⟩
| ⟨*id*⟩ '-' ⟨*additive-expression*⟩
| ⟨*id*⟩

id の BNF

⟨*id*⟩ ::= 'a'
| 'b'
| 'c'

この単純なパーサが代入式 (assignment-expression) $a = b - c$ を受理する例を図 3.6 に示す．図中の太線で記された部分がトレース情報である．

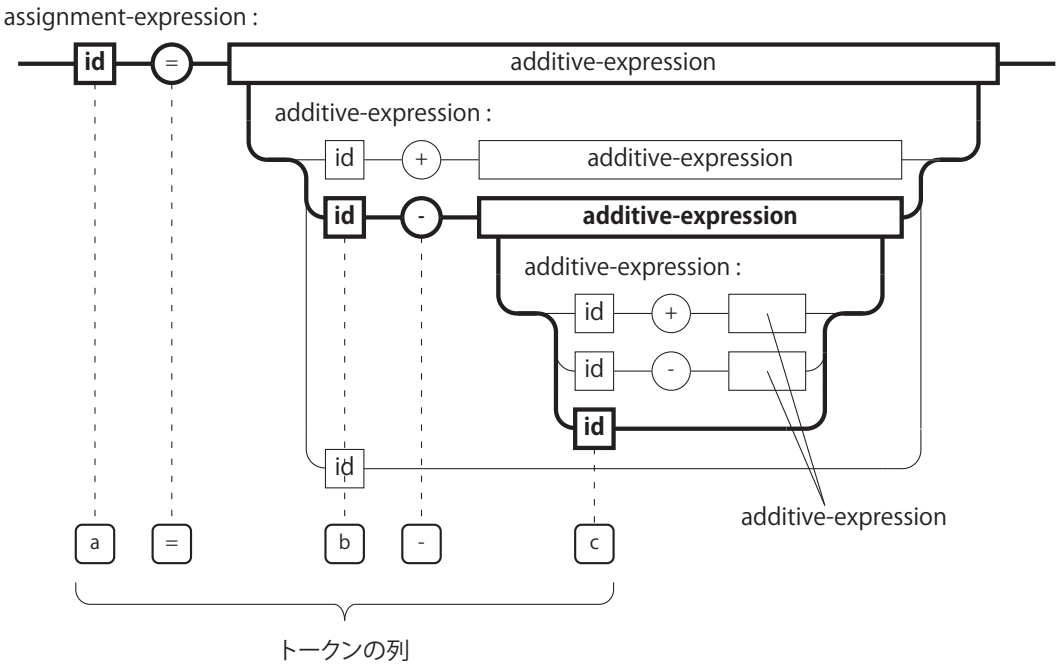


図 3.6: 単純なパーサの例

3.5.2 左再帰の除去

C 言語の構文は文脈自由文法であり，ボトムアップ構文解析の一種である LR 法にて解析可能であることが知られているが，本研究ではより単純なトップダウン構文解析の一種である LL 法を採用する．

LL 法による構文解析では C 言語の構文に登場する左再帰をそのまま解析することはできない．まず左再帰について説明する．以下は C 言語の構文の一部である multiplicative-expression を定義する BNF である．この BNF における右辺の左端に multiplicative-expression 自身が出現している．これを左再帰と言う．

multiplicative-expression の BNF

⟨*multiplicative-expression*⟩ ::= ⟨*cast-expression*⟩
| ⟨*multiplicative-expression*⟩ ‘*’ ⟨*cast-expression*⟩
| ⟨*multiplicative-expression*⟩ ‘/’ ⟨*cast-expression*⟩
| ⟨*multiplicative-expression*⟩ ‘%’ ⟨*cast-expression*⟩

LL 法による左端導出では導出位置のトークン並びが cast-expression でない場合，multiplicative-expression が合致するかを検査するが，やはり cast-expression ではないため，multiplicative-expression との合致を検査し続けてしまい停止しない．そこで左再帰を除去することで対応する．multiplicative-expression の例を以下に示す．

multiplicative-expression の BNF（左再帰除去版）

⟨*multiplicative-expression*⟩ ::= ⟨*cast-expression*⟩
| ⟨*cast-expression*⟩ ‘*’ ⟨*multiplicative-expression*⟩
| ⟨*cast-expression*⟩ ‘/’ ⟨*multiplicative-expression*⟩
| ⟨*cast-expression*⟩ ‘%’ ⟨*multiplicative-expression*⟩

左再帰除去後の BNF は，受理可能な構文は等価であるが生成される構文木が異なることに注意が必要である．例えば，$a/b*c$ という C 言語の式に関して，乗除算の演算子 $(*, /)$ は左結合 (left to right) であるから，$(a/b)*c$ と解釈されるのが正しいが (図 3.7)，左再帰の除去を行ったパーサでは $a/(b*c)$ と解釈されてしまい，式の値 (計算結果) が異なる (図 3.8)．しかし，静的コード解析ツールでは式が実際にどのように解釈されるか（どのような順序で計算されるか）ではなく，複数の二項演算の優先度が括弧によって明示されているかどうかを検査することが重要である．括弧も含めた字面上の情報が構文木に保存されているため，字面上異なるコード片は構文木も異なる．計算結果が必要になった際には構文木から結合優先度を判断することも可能である．左再帰を除去したパーサの一覧を付録 A に掲載する．

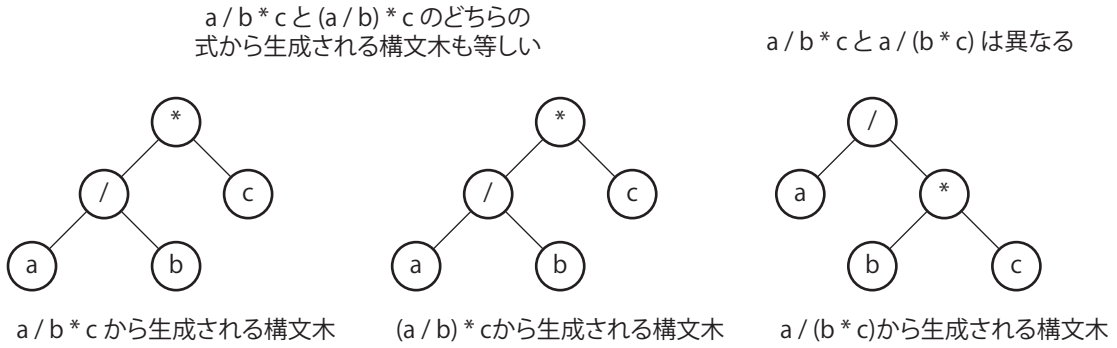


図 3.7: コンパイラの LR 構文解析の例

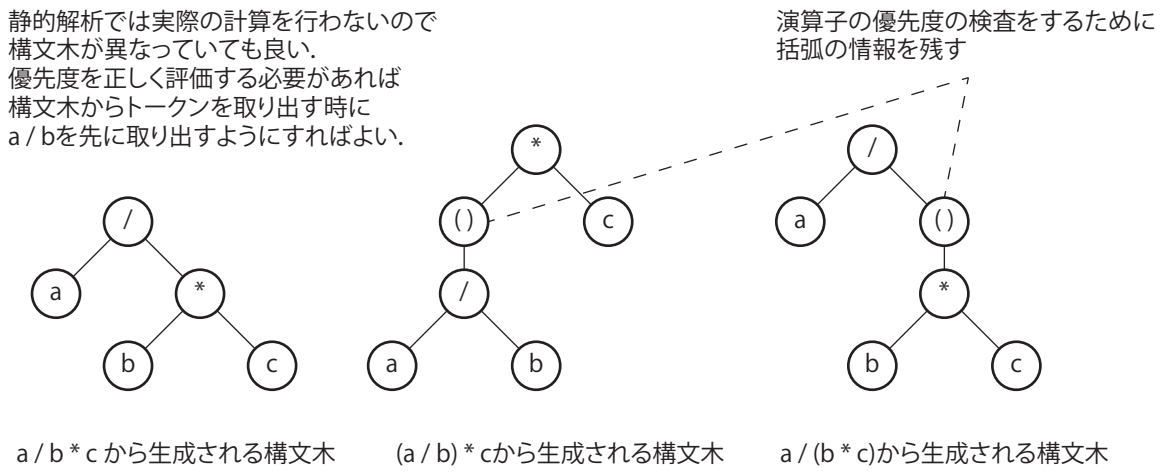


図 3.8: 静的解析ツールの LL 構文解析の例

3.5.3 構文解析の制限

本研究で作成した静的解析ツールの構文解析器には以下の制限が存在する.

- 関数形式でないマクロは識別子 (identifier) が記述できる場所以外では使用できない
- 関数形式マクロは関数コールが記述できる場所以外では使用できない

3.5.3.1 関数形式でないマクロの制限

関数形式でないマクロは識別子 (identifier) が記述できる場所以外では使用できない. identifier は primary-exprssion の一部であるので，式が記述できる場所ならば，関数形式でないマクロを使用することができる．関数形式でないマクロが使用できる例を例 3.5.1 に，使用できない例を例 3.5.2 に示す.

例 3.5.1 (関数形式でないマクロが使用できる例)

```
#define DEF_VAR int x = 10

DEF_VAR;

void func(void) {
  DEF_VAR;
  int y = 10;
}
```

例 3.5.2 (関数形式でないマクロが使用できない例)

```
#define DEF_VAR int x;
#define DEF_IF if (exp) { proc(); }

DEF_VAR // セミコロンがないため，区切りが判断できない.

void func(void) {
  DEF_IF // セミコロンがないため，区切りが判断できない.
  proc();
}
```

3.5.3.2 関数形式マクロの制限

関数形式マクロは関数コールが記述できる場所以外では使用できない．関数コールは primary-exprssion に postfix-expression の引数リストが続いた形であり，式が記述できる場所ならば関数形式マクロを使用することができる．関数形式マクロが使用できる例を例 3.5.3 に，使用できない例を例 3.5.4 に示す.

例 3.5.3 (関数形式マクロが使用できる例)

```
#define DEF_COND(proc1, proc2, proc3) { proc1(); proc2(); proc3(); }

void func(void) {
  DEF_COND(f1, f2, f3);
}
```

例 3.5.4 (関数形式マクロが使用できない例)

```
#define times(x) for (int i = 0; i < (x); i++)

void func(void) {
    // 関数コールの後にブロックが続くことはない.
    times(10) {
      loop_body();
    }
}
```

3.6 検査の実施

今回の研究では大きく分けて以下の 3 種類の検査を行う．本節の各小節にてそれぞれの検査方法を説明する.

- 字句の検査
- 構文の検査
- 型の検査

3.6.1 字句の検査

字句の検査はトークン内の情報を調べることで実現する．例えばソースコード内で８進定数が使用されているかを検査する場合，トークン種別が８進定数であるものを探索する (図 3.9).

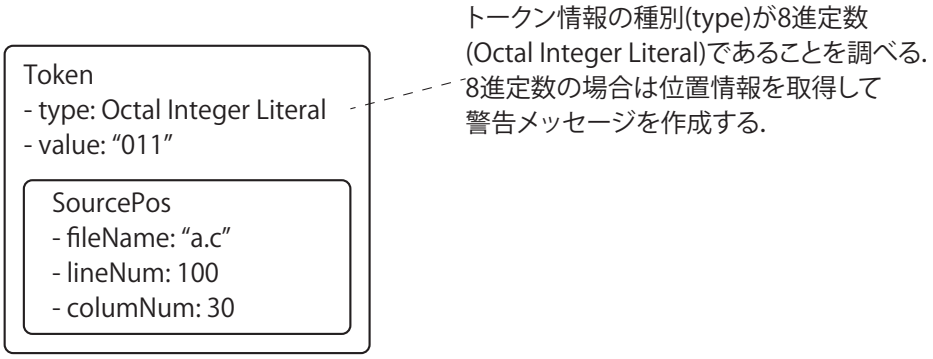


図 3.9: 8 進定数を検出する例

3.6.2 構文の検査

構文の検査は構文木を調べることで実現する．例えば，if 文の本体が波括弧で囲まれているかどうかを調べるには，if 文の本体 (statement) が複合文 (compound-statement) であるかを確認することで調べることができる．selection-statement, statement, compound-statement の BNF を以下に示す.

selection-statement の BNF

$$
\begin{aligned}
\langle \textit{selection-statement} \rangle \quad ::= \; & \text{'if' '('} \langle \textit{expression} \rangle \text{ ')'} \langle \textit{statement} \rangle \\
| \; & \text{'if' '('} \langle \textit{expression} \rangle \text{ ')'} \langle \textit{statement} \rangle \text{ 'else'} \langle \textit{statement} \rangle \\
| \; & \text{'switch' '('} \langle \textit{expression} \rangle \text{ ')'} \langle \textit{statement} \rangle
\end{aligned}
$$

statement の BNF

⟨*statement*⟩ ::= ⟨*labeled-statement*⟩
| ⟨*compound-statement*⟩
| ⟨*expression-statement*⟩
| ⟨*selection-statement*⟩
| ⟨*iteration-statement*⟩
| ⟨*jump-statement*⟩

compound-statement の BNF

⟨*compound-statement*⟩ ::= '{' '}'
| '{' ⟨*block-item-list*⟩ '}'

3.6.3 型の検査

式 (expression) と宣言 (statement) の中で演算（二項演算など）を行っている箇所の型を検査する．

3.6.3.1 識別子情報のスタック

型名や変数名から型の情報を参照するために，名前空間ごとに識別子の情報を保存する必要がある．C 言語の名前空間は以下の 4 種類の名前空間が存在し，異なる名前空間に同名の識別子が存在してもエラーになることはない．

1. ラベル
2. タグ（構造体，共用体，列挙体）
3. メンバ（構造体，共用体）
4. その他の識別子（変数宣言，typedef，列挙定数）

「1. ラベル」は型情報とは関わらないため，型検査用の処理でラベル情報を保持する必要はない．「2. タグ」と「4. その他の識別子」については，検索の効率化のため [3] さらに細かい分類を行う．まず，「2. タグ」については構造体，共用体，列挙体で別々にタグ（識別子）を保存する．また，「4. その他の識別子」については typedef で宣言された型名と，通常の変数宣言の情報を分けて保存する．なお，「3. メンバ」は構造体，共用体のタグ情報と共に保存する．型検査のための識別子の分類を表 3.5 にまとめた．

C 言語の識別子には有効範囲（スコープ）が定められている．関数の本体やブロック（複合文: compound-statement）の内部で定義された識別子はその終了時に削除（解放）

[3] 例えば struct tag_a という構造体タグを検索するときにタグ全体から検索するよりも，構造体用のタグの中から検索したほうが都合が良い．

表 3.5: 型検査のための識別子の分類

識別子種別	意味	保持する情報
typedef 型名	typedef で宣言された識別子	型情報
構造体名	struct で宣言されたタグ	メンバとその型の情報
共用体名	union で宣言されたタグ	メンバとその型の情報
列挙体名	enum で宣言されたタグ	列挙定数とその値
変数名	変数宣言で宣言された識別子	型情報
関数名	変数宣言で宣言された識別子	引数とその型の情報，戻り値の型情報

され，以降のコードからは参照することができない．静的コード解析の際も同様に識別子のスコープを意識する必要がある．C 言語の変数はスタックとして表現されており [4]，関数の本体やブロック内で変数が定義されたときにはスタックにプッシュされ，そのスコープの終了時にはスタックからポップされる．静的コード解析でも識別子のスコープをスタックを使用して表現することができる．そこで，表 3.5 で示した識別子の分類ごとにスタックを用意する．

3.6.3.2 型情報の比較

構文木をトップレベルから探索していき，識別子が現れたらデファイン情報のリスト，識別子のスタックの順に検索し，型情報を取得する．識別子の型情報から，その識別子を使用している式や宣言の型が一致するかを調べる．図 3.10 に型の不一致を検出する例を示す．

[4] 大域変数はスタックに格納されないが，ここではプログラムの終了までポップされないと捉えることとする．

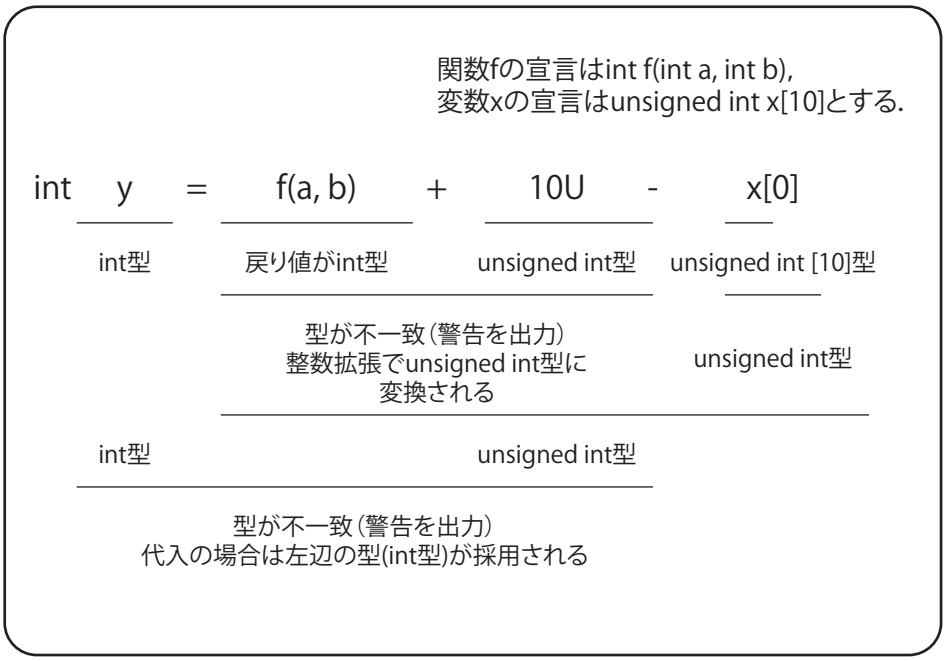


図 3.10: 型の不一致を検出する例（マクロなし）

なお，コンパイルスイッチによって同名の識別子が複数種類存在する場合には，それらの型が一致するかどうかを確認する．図 3.11 にデファイン情報とコンパイルスイッチを含んだ場合の例を示す.

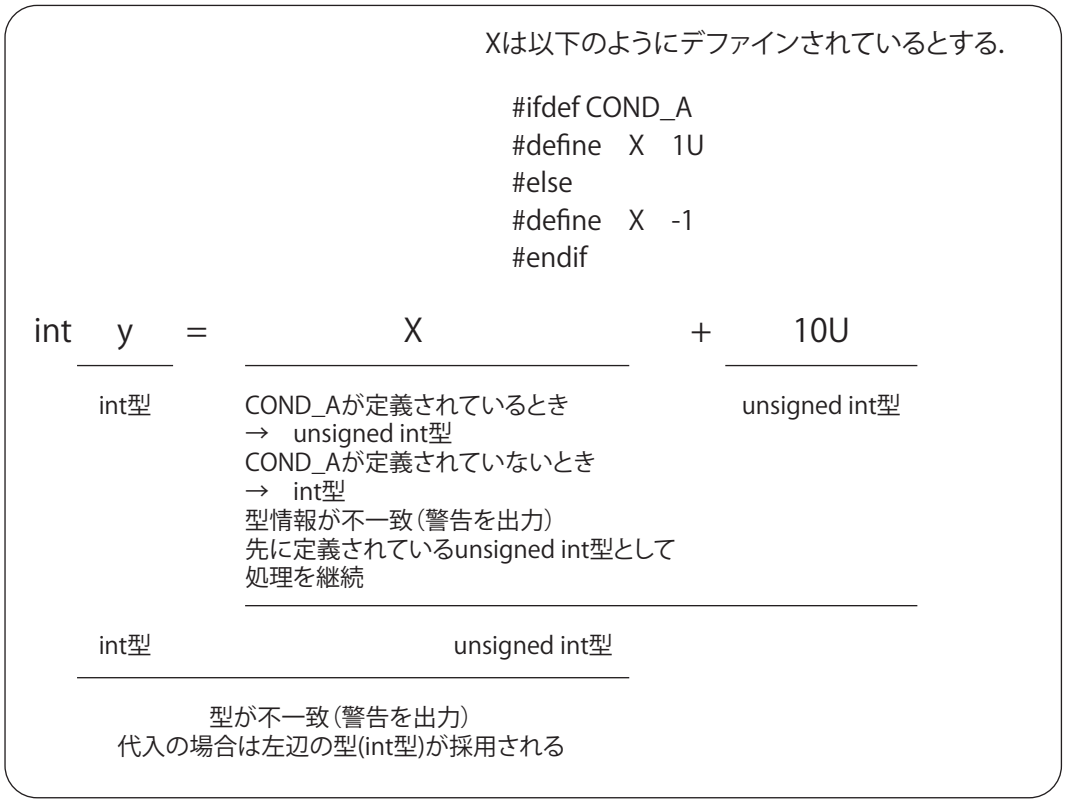


図 3.11: 型の不一致を検出する例（マクロあり）

第4章　検査項目

本章では静的コード解析ツールで検査する項目について述べる．静的コード解析ツールで検査する項目は文献 [10] を基に作成した．

4.1　字句に関する警告

本節では字句解析時に検出する項目について述べる．実装した項目の一覧を表 4.1 に示す．これらの項目は字句解析時にトークンの文字列に対してパターンマッチを行うことで検出する．

表 4.1: 字句解析時に検出する項目

番号	内容
1	言語規格外のエスケープシーケンス
2	文字列リテラル内での改行
3	goto 文, continue 文
4	８進定数
5	#line マクロ
6	#undef マクロ
7	#include マクロへの間接的なファイル指定
8	#include マクロへの絶対パス指定

4.1.1　言語規格外のエスケープシーケンス

エスケープシーケンスとは C 言語の規格 [8] で定義されている escape-sequence を指し，文字定数または文字列リテラル中に出現するバックスラッシュとそれに続く文字または文字列である．規格で定義されているもの以外は処理系依存であり，誤記の可能性や移植時に問題となる可能性があるため警告する．規格で定義されているエスケープシーケンスは以下の通り．

\', \", \?, \\, \<8 進定数列>, \x<16 進定数列>,
\a, \b, \f, \n, \r, \t, \v

4.1.2 文字列リテラル内での改行

文字列リテラルの中で改行[1] の前にバックスラッシュを置くことで複数行に分割して記述する (line splicing) ことができるが，可読性が低下するため警告する．

```
    char s[] = "First line.\
Second line.\
Third line.\
";
```

これは以下の文と等価である．

```
    char s[] = "First line.Second line.Third line";
```

なお，文字列が長くなり複数行に記述したい場合は，文字列リテラルを続けて書くことでコンパイル時に連結される．以下も上記二つの文と等価である．

```
    char s[] = "First line."
               "Second line."
               "Third line.";
```

4.1.3 goto 文, continue 文

不用意に goto 文や continue 文を使用するとプログラムの構造が複雑になることがある．また，goto 文や continue 文を使用したプログラムは，これらを使用しない形に書き直すことができる．多重ループからの脱出など，goto 文や continue 文を使用することで構造が単純にできる場合もあるが，コード片がより単純な構造に変換できるかどうかは一概に評価することができないため，使用箇所をすべて警告する．

4.1.4 8 進定数

C 言語では 0 に続く数値は 8 進数定数として解釈されるが，10 進定数表記と紛らわしいため，混在すると可読性が低下する．よって 0 を除く 8 進数定数を検出して警告する．8 進数定数と 10 進数定数が混在している例を例 4.1.1 に示す．

例 4.1.1 (8 進数定数と 10 進数定数が混在している例)

[1] 改行コード'\n' ではない．

```
a =   0; // 0 は OK
b = 000; // 8 進数表記の 0
c = 011; // 11 ではなく 9
d = 111; // 10 進数の 111
e = 012; // 12 ではなく 10
```

4.1.5 #line マクロ

#line マクロはコンパイラの警告やエラーの行番号を変更するために使用する．コンパイラのメッセージ（警告やエラー）を確認する際に，実際の行番号とメッセージの行番号がずれてしまうため警告する．

4.1.6 #undef マクロ

#undef マクロを使用することで定義されているマクロを無効化することができるが，#undef マクロを見落とすと以下の例のように容易にコードの意味が変わってしまうため警告する．#undef マクロの例を例 4.1.2 に示す．

例 4.1.2 (#undef マクロの例)

```
#define D
// <中略>
#undef D
// <中略>
#ifdef D
  // #undef を見落とすとこちらがコンパイル対象であると判断してしまう
#else
  // 実際にコンパイル対象になるのはこちらのコード
#endif
```

4.1.7 #include マクロへの間接的なファイル指定

#include マクロには，⟨filename⟩ または”filename” のようにファイル名を直接指定するのが一般的であるが，以下の例のように#include 指令の後にはマクロで読み込むヘッダファイルを置き換えることができる．ヘッダファイルが切り替わるような変更は，Makefile によるコンパイル時のパス指定などで一元管理することを検討した方が良い．#incude マクロにファイル名以外を指定している例を例 4.1.3 に示す．

例 4.1.3 (#incude マクロにファイル名以外を指定している例)

```
#if A
#define HEADER "file_a.h"
#else
#define HEADER "file_b.h"
#endif
// <中略>
// file_a.h と file_b.h のどちらがインクルードされるかは，A の値に依存する．
#include HEADER
```

4.1.8 #include マクロへの絶対パス指定

インクルードファイルを絶対パスで指定すると，ソースコードを配置するディレクトリを変更した時に修正しなくてはならない．

4.2 構文に関する警告

本節では構文解析時に検出する項目について述べる．実装した項目の一覧を表 4.2 に示す．これらの項目は構文解析時にパーサが受理した構文木を検査することで検出する．

表 4.2: 構文解析時に検出する項目

番号	内容
1	下線で始まる名前の宣言または定義
2	条件演算子の条件式の括弧の省略
3	二項演算子の混在
4	カンマ演算子が使用されている
5	ヘッダファイル内の変数，関数の定義
6	外部変数の配列における要素数の省略
7	宣言文に複数の変数が含まれている
8	初期化されていない const 変数
9	3 段階以上のポインタ変数の定義
10	共用体の定義
11	関数の引数が空
12	可変個引数を持つ関数の定義
13	複合型変数の値渡し
14	プロトタイプ宣言における引数名の省略

次ページに続く

表 4.2 – 構文解析時に検出する項目（続き）

番号	内容
15	ビットフィールドの使用
16	ビットフィールドの型が規格外
17	switch 文が default 節を持たない
18	if, for, while の条件式に代入が含まれている
19	if, for, while の本体が波括弧で囲まれていない
20	if 文に else 節が存在しない
21	switch 文内のフォールスルー
22	K&R 形式の関数定義の使用

4.2.1 下線で始まる名前の宣言または定義

下線で始まる識別子は C 言語規格により予約済みであり，処理系によって使用されている可能性があるため，処理系のユーザ（プログラマ）が宣言または定義すべきでない．この項目を検出するために，型 (typedef, struct, union, enum) や変数，関数の宣言または定義を検索して識別子を検査する．

4.2.2 条件演算子の条件式の括弧の省略

if 文や while 分の条件式とは異なり，条件演算子の条件式部の括弧は必須ではないが，可読性の面から括弧をつけるべきである．条件演算子の例を例 4.2.1 に示す．

例 4.2.1 (条件演算子の例)

```
// 括弧が省略されている状態
v = (e1 < e2) && (e2 < e3) ? s1 : s2;
// 括弧をつけた状態
v = ((e1 < e2) && (e2 < e3)) ? s1 : s2;
```

4.2.3 二項演算子の混在

括弧をつけずに優先度の異なる二項演算子を混在させると可読性が低下するため，括弧をつけて演算の順序を明示すべきである．二項演算子の混在の例を例 4.2.2 に示す．

例 4.2.2 (二項演算子の混在の例)

```
int a = b >> c + 1 & 0x3;
// 以下のように括弧をつけると計算の順序が理解しやすい.
int a = (b >> (c + 1)) & 0x3;
```

4.2.4 カンマ演算子が使用されている

カンマ演算子を使用した式は，使用しない形に書き直すことができる．意図的にカンマ演算子を使用している場合，プログラマはその理由を把握しているべきである．よって使用箇所を警告する．カンマ演算子の例を 4.2.3 に示す.

例 4.2.3 (カンマ演算子の例)

```
// カンマ演算子を使用する例.
// 変数 v には関数 h() の戻り値が格納される.
v = (f(), g(), h());

// カンマ演算子を使用しない記述
f();
g();
v = h();
```

4.2.5 ヘッダファイル内の変数，関数の定義

ヘッダファイルは複数のソースコードから参照される可能性があるため，ヘッダファイル内に変数や関数を定義するとそれぞれのオブジェクトに変数や関数の実態が含まれることになる．ヘッダファイルには変数と関数の宣言を記述すべきである.

4.2.6 外部変数の配列における要素数の省略

配列の extern 宣言はその要素数を省略することができるが，要素数を意識してアクセスするために明示すべきである．外部変数の配列における要素数の省略の例を 4.2.4 に示す.

例 4.2.4 (外部変数の配列における要素数の省略の例)

```
// ヘッダファイルの外部変数宣言
// 配列サイズが省略されてもエラーにならない.
extern int x[];

// ソースコードの変数の定義
// 実際の配列の要素数は 20 である.
int x[20] = { /* 初期化 */ };
```

4.2.7　宣言文に複数の変数が含まれている

１つの文で複数の変数宣言を行うと可読性が低下する．特にポインタや型修飾子 (const, volatile, restrict) が混在すると混乱を招きやすい．宣言文に複数の変数が含まれている例を 4.2.5 に示す．

例 4.2.5 (宣言文に複数の変数が含まれている例)

```
// 複数の変数宣言を同時に行っている例
int a = 10, b[2] = {1, 2}, * const c = &a, d;

// １つの文に１つの変数を宣言するように書き直した例.
int a;
int b[2] = {1, 2};
int * const c = &a;
int d;
```

4.2.8　初期化されていない const 変数

const 型変数は宣言時以外には初期化できないため，宣言時に初期化すべきである．コンパイルエラーにはならないためツールにて検出する．

4.2.9　３段階以上のポインタ変数の定義

ソースコードを記述するにあたって必要となるポインタは以下の２種類である．

1 段階のポインタ ：組込み型[2] や複合型[3] のポインタ

2 段階のポインタ ：ポインタ自身を書き換えるために関数に渡す「ポインタのポインタ」

３段階以上のポインタは不要であるため警告する．

4.2.10　共用体の定義

共用体のメンバの間にはパディングが挿入されることがあるが，そのルールは処理系によって異なる．共用体を使用する場合には，代入時に使用したメンバのみにアクセスするなどの注意が必要となる．

[2]char, short, int, long などの処理系によって定義されている型
[3]構造体，共用体，列挙型

4.2.11 関数の引数が空

プロトタイプ宣言の引数を空 () とした場合，引数が不明と解釈されコンパイラによる引数のチェックが行われない．引数のチェックが行われないと，関数を呼び出す側と呼び出される側で認識が異なってもコンパイルエラーとならず，重大な問題を引き起こす可能性がある．もし引数が不要な場合は空 () ではなく (void) と書くのが正しい．関数の引数が空の例を例 4.2.6 に示す.

例 4.2.6 (関数の引数が空の例)

```
// 関数 f のプロトタイプ宣言（引数が空）
void f();

void g(void) {
  // 関数 f の呼び出し（引数の数が異なるがコンパイルエラーとならない）
  f(10);
}

// 関数 f の定義（プロトタイプ宣言と引数が異なっているがエラーとならない）
void f(int a, int b) {
  // 引数 b の値は不定となる.
}
```

4.2.12 可変個引数を持つ関数の定義

まず可変個引数を持った関数のプロトタイプ宣言の例を示す.

```
void func(char * format, ...);
```

可変個引数を持った関数はプロトタイプ宣言だけでは引数の数や型がわからないため，コンパイル段階ではチェックできない．関数を呼び出す側と呼び出される側で認識が異なれば，実行時に重大な問題[4] を引き起こす可能性がある．この項目は関数の引数 (parameter-type-list) の末尾に，”...” トークンが出現するかを検査することで検出可能である.

なお，GCC 拡張による可変長引数も検出する.

```
void func(char * format...); // GCC 拡張では’’...’’ の前のカンマが無い
```

[4] スタックの破壊やメモリのアクセス例外等.

4.2.13 複合型変数の値渡し

複合型（構造体, 共用体）の変数を関数に渡す場合，メンバを全てコピーする値渡し (call by value) は行わず，ポインタを渡す参照渡し (call by reference) を使用すべきである．理由は複合型のサイズが大きい場合はコピーに時間がかかることや，一時領域（スタック）を大量に消費するためである．特に組込みのプログラムはシステムの動作速度やスタックの使用量の制限が厳しい．構造体のサイズが 1 ワード以下の場合は値渡しでも問題はないが，本研究で作成するツールでは区別せず検出することとする．複合型変数の値渡しの例を例 4.2.7 に示す.

例 4.2.7 (複合型変数の値渡しの例)

```
struct ST {
  int m1;
  int m2;
  int m3;
};

// 値渡しの場合
void callee(struct ST st);

void caller(void) {
  struct ST st = { /* メンバの初期化 */ };
  callee(st); // メンバが全てコピーされる.
}

// 参照渡しの場合
void callee(struct ST * st);

void caller(void) {
  struct ST st = { /* メンバの初期化 */ };
  callee(&st); // 構造体変数のポインタのみがコピーされる.
}
```

4.2.14 プロトタイプ宣言における引数名の省略

関数のプロトタイプ宣言では引数の名前を省略し，型のみを記述することができる．しかし，引数名はインタフェースの意味を説明する重要な情報となるため省略すべきでない．プロトタイプ宣言における引数名の省略の例を例 4.2.8 に示す.

例 4.2.8 (プロトタイプ宣言における引数名の省略の例)

```
// 引数名を省略した例
void send(char * , int);

// 引数名を記述した例
void send(char * buffer, int buffer_size);
```

4.2.15 ビットフィールドの使用

ビットフィールドはビットの割り付けが処理系によって異なるため，特定の割り付けに依存するコーディングを行うと移植時に問題となる場合がある．

4.2.16 ビットフィールドの型が規格外

C 言語の規格ではビットフィールドは以下の 3 種類以外は処理系依存とされている．よって，以下の 3 種類以外の型を使用すべきではない．

1. signed int
2. unsigned int
3. _Bool(C99 以降で使用可能)

4.2.17 switch 文が default 節を持たない

switch 文の最後に default 節がない場合，十分に考えたうえで省略されたのか，書き忘れたのかが判別できない．必ず default 節を作成し，処理が不要の場合にはコメントで説明を記載すべきである．

4.2.18 if, for, while の条件式に代入が含まれている

if, for, while の条件式に代入 (=) が含まれている場合，等価演算子 (==) の記述ミスである可能性が高く，意図がわかりにくいため使用すべきでない．if, for, while の条件式に代入が含まれている例を例 4.2.9 に示す．

例 4.2.9 (if, for, while の条件式に代入が含まれている例)

```
if (a = b) {
  // b が真のときの処理
} else {
  // b が偽のときの処理
}
```

4.2.19 if, for, while の本体が波括弧で囲まれていない

if, for, while の本体が空文（セミコロンのみ）または 1 つの文のみであっても波括弧で囲むことで一貫性が保たれ可読性が向上する．また，本体に文を追加する際に，波括弧をつけ忘れるミスを防ぐことができる．本体が波括弧で囲まれていない例を例 4.2.10 に示す．

例 4.2.10 (本体が波括弧で囲まれていない例)

```
if (p != NULL)
  ;
else
  // ここに文を追加すると，その下の処理が else 節の範囲から外れてしまう．
  error();
```

4.2.20 if 文に else 節が存在しない

if 文に else 節を記述しなかった場合，記述漏れかどうかが判断できない．何もしない場合であっても else 節を記述し，コメントなどで何もしないことを示すべきである．

4.2.21 switch 文内のフォールスルー

switch 文の本体において，case 節に break 文を記載しなければ，次の case の処理が続けて実行される（フォールスルー）動作となるが，break 文を書き忘れたのかどうかがわからないため使用すべきでない．case ラベルのフォールスルーの例を例に示す．

例 4.2.11 (switch 文内のフォールスルーの例)

```
switch (n) {
case 0:
  // ここに break がないため，処理 A に進む．
case 1:
  // 処理 A
```

```
  break;
default:
  // 処理B
  break;
}
```

4.2.22 K&R 形式の関数定義の使用

K&R 形式の関数定義は引数の型を記述しない．コンパイラによる型のチェックができないため使用すべきでない．

K&R 形式の関数定義を例 4.2.12 に示す．

例 4.2.12 (K&R 形式の関数定義)

```
void f(a, b)
int a;
char b;
{
  // 何らかの処理
}

void g(void) {
  int x;
  int y;
  f(x, y); // 型チェックが行われず，コンパイルエラーとならない．
}
```

4.3 型に関する警告

本節では構文解析時に検出する項目について述べる．実装した項目の一覧を表 4.3 に示す．これらの項目は構文木の式の型を求め検査することで検出する．

表 4.3: 型に関する警告

番号	内容
1	条件演算の結果の型が不一致
2	ループカウンタと比較する変数の型が不一致

次ページに続く

表 4.3 – 型に関する警告（続き）

番号	内容
3	符号なし整数への単項マイナス演算子の適用
4	二項演算子の左右の式の型の不一致
5	浮動小数点式に対する等価または非等価の比較
6	ループカウンタに浮動小数点を使用
7	基本型を使用している
8	条件式が真偽値でない
9	論理否定演算子を適用する式が真偽値でない

4.3.1 条件演算の結果の型が不一致

条件演算子（?:演算子）を使用した式において，結果が左辺に代入される場合，条件式の評価結果によらず同じ型になるべきである．条件演算の結果の型が異なる例を例 4.3.1 に示す.

例 4.3.1 (条件演算の結果の型が異なる例)

```
int  a = 10;
char b = 'b';
// b は char 型なので結果 result の型である int と合致しない.
int result = exp ? a : b;
```

4.3.2 ループカウンタと比較する変数の型が不一致

for と while のループ継続条件において，ループカウンタ変数とループ継続条件の比較に使用する変数の型は一致させるべきである．特にループカウンタよりループ継続条件に使用する変数の型が大きい場合に注意が必要である．ループカウンタと比較する変数の型が一致しない例を例 4.3.2 に示す.

例 4.3.2 (ループカウンタと比較する変数の型が一致しない例)

```
int max = 1000;
for (unsigned char i = 0; i < max; i++) {
  // unsigned char 型のループカウンタ i は 255 までしか
  // 表現することができないため，オーバーフローが発生し 0 に戻るため
  // max(=1000) に達することがない．よってこのループは停止しない.
}
```

4.3.3 符号なし整数への単項マイナス演算子の適用

C 言語の仕様により int で表現できる型同士の演算を行う場合，整数拡張により int に変換され，unsigned int(int で表現できない) 型を含む演算を行う場合は整数拡張により unsigned int 型に変換してから演算される．long や long long 型についても同様に整数拡張が行われる．符号なし整数に単項マイナス演算子を適用すると，そのサイズによって動作が異なるためこのような記述は使用すべきでない．符号なしの式への単項マイナス演算子の適用の例を例 4.3.3 に示す.

例 4.3.3 (符号なし整数への単項マイナス演算子の適用の例)

```
// 以下の例では short は 16bit，int は 32bit の幅を持つこととする．
// int と short のビット幅が等しい処理系では以下の例は成り立たない．

// 対象が int 型以上のサイズである場合
unsigned int u = 1;
if (-u < 0) {
  // u は unsigned int であり，int の表現範囲に含まれないため，
  // -u は unsigned int として演算され，-1 とはならない．
  // (int が 32bit の場合，4294967295(=0xffffffff) となる)
  // よって，こちらが実行されることはない．
} else {
  // こちらが実行される．
}

// 対象が int 型未満のサイズである場合
unsigned short u = 1;
if (-u < 0) {
  // u は unsigned short であり int で表現可能であるため，
  // -u は int に拡張されて計算されるため-1 となる．
  // よって，こちらが実行される．
} else {
  // こちらが実行されることはない．
}
```

4.3.4 二項演算子の左右の式の型の不一致

二項演算によって算術演算を行う場合には算術変換によって暗黙で型の変換が行われる．型の異なる式同士の演算を行う場合には，型変換が行われても問題がないことを保証

して，明示的にキャストすべきである．二項演算子の左右の式の型の不一致の例を例 4.3.4 に示す．

例 4.3.4 (二項演算子の左右の式の型の不一致の例)

```
unsigned int a = 10;
int b = -100;

// 整数拡張により unsigned int に変換される．
// int が 32bit の場合，4294967206(=0xfffffffa6) となり，
// -90 にはならない．
// a の取りうる値が int に収まるのであれば，(int)a + b
// とすることで-90 が得られる．
if (a + b < 0) {
  // こちらは実行されない．
} else {
  // こちらが実行される．
}
```

4.3.5 浮動小数点式に対する等価または非等価の比較

浮動小数点型の変数は計算の過程で誤差を含むことがあるため，等価演算子 (==) および非等価演算子 (!=) を使用した厳密な合致を期待してはならない．誤差の範囲を考慮して不等号による比較を行う必要が有る．浮動小数点式に対する等価または非等価の比較の例を例 4.3.5 に示す．

例 4.3.5 (浮動小数点式に対する等価比較の例)

```
// 等価演算子 (==) による比較
bool comp1(double a, double b) {
  // a と b が理論上は一致する場合であっても
  // 求める過程で誤差が発生していると a == b は真にならない．
  if (a == b) {
    return TRUE;
  } else {
    return FALSE;
  }
}

// 誤差を考慮した比較
```

```
// fabs 関数と DBL_EPSILON（計算機イプシロン）を使用するために
// 標準ライブラリの float.h と math.h をインクルードする.
bool comp2(double a, double b) {
  if (fabs(a - b) < DBL_EPSILON) {
    return TRUE;
  } else {
    return FALSE;
  }
}
```

4.3.6 ループカウンタに浮動小数点を使用

浮動小数点数の計算を繰り返すと誤差が累積するため，期待したループ回数にならないことがある．ループカウンタは整数型を使用すべきである．ループカウンタに浮動小数点を使用している例を例 4.3.6 に示す.

例 4.3.6 (ループカウンタに浮動小数点を使用している例)

```
// 0.1 は 2 進数では循環小数であるため，浮動小数点数では正確に
// 表現することができない.
// 例えば，0.0 に 0.1 を 10 回加えても 1.0 とは一致しない.
// IEEE754 で定義されている単精度浮動小数点数 (4 バイト) では 10 回ループし，
// 倍精度浮動小数点数（8 バイト）では 11 回ループする.

// 単精度浮動小数点数
for (float i = 0.0f; i < 1.0f; i += 0.1f) {
    // IEEE754 に準拠する場合，10 回実行される.
}

// 倍精度浮動小数点数
for (double i = 0.0; i < 1.0; i += 0.1) {
    // IEEE754 に準拠する場合，11 回実行される.
}
```

4.3.7 基本型を使用している

プロジェクトのコーディング規約で基本型 (char, short, int, long, float, double, _Complex) をプログラマが直接使用することを禁じている場合がある．これには以下のような利点がある.

- 変数のサイズを明示した型を用意し，オーバーフロー等の不具合に気付きやすくなる
- 基本型の組み合わせによる多彩な表現[5]を統一できる

例えばC99のstdint.hで定義されている型 (e.g. int8_t, int16_t, int32_t) を使用する．本項目はそのようなプロジェクトで意図せずに基本型を使用してしまっている部分を検出するために有用である．

4.3.8 条件式が真偽値でない

if, while, for 文の条件式は真偽値であるべきである．C 言語では真は 0 以外，偽は 0 と定められており，条件式に整数型を使用することができるが可読性が低下する．条件式が真偽値でない例を例 4.3.7 に，条件式を真偽値に直した例を 4.3.8 に示す．どちらも動作は同じだが意図が明確になる．

例 4.3.7 (条件式が真偽値でない例)

```
int func(void);
int x = func();
if (x) {
  // x が 0 以外の場合の処理
} else {
  // x が 0 の場合の処理
}
```

例 4.3.8 (条件式を真偽値に直した例)

```
int func(void);
int x = func();
if (x != 0) {
  // x が 0 以外の場合の処理
} else {
  // x が 0 の場合の処理
}
```

switch 文の条件式は真偽値の場合，偽 (0) の処理は case で扱い，真 (0 以外) の処理は default で扱う必要があり複雑である．真偽値で分岐したい場合は switch ではなく，if 文を使用したほうが単純になる．条件式が真偽値の場合の switch 文と if 文の例を例 4.3.9 に示す．

[5] int, signed, signed int, int signed はすべて同じ意味を持つ

例 4.3.9 (条件式が真偽値の場合の switch 文と if 文の比較)

```
switch (a == b) {
  case 0: // FALSE
    // 偽のときの処理
    break;
  deafult:
    // 真のときの処理
    break;
}

if (a == b) {
  // 真のときの処理
} else {
  // 偽のときの処理
}
```

4.3.9 論理否定演算子を適用する式が真偽値でない

論理否定演算子 (!) は真偽値 (bool) に適用されるべきである．論理否定演算子を適用する式が真偽値でない例を例 4.3.10 に示す.

例 4.3.10 (論理否定演算子を適用する式が真偽値でない例)

```
int * a = 100;
if (!a) { // if (a == 0) と書いた方がわかりやすい
  // a が 0 のときの処理
} else {
  // a が 0 でないときの処理
}
```

第5章　評価

本章では，本研究で作成した静的コード解析ツールを使用してソースコードの解析を行い，その結果について述べる．

5.1　解析対象

解析対象のソースコードは組込み Linux でも使用されている Linux カーネルの I^2C ドライバとした．Linux カーネルは BeagleBoard-xM 用のパッチを当てたものを使用した．BeagleBoard-xM は Texas Instruments Incorporated(TI) により開発された教育用のシングルボードコンピュータであり，組込みシステムにおけるデバイスドライバのプログラミング教材としても使用されている．ソースコード及びパッチは GIT リポジトリ `https://github.com/RobertCNelson/armv7-multiplatform` から取得できる．使用した GIT リポジトリのリビジョンは b9f136626202405ba63cac267ac93b759c22f2ba である．

なお，解析の主な対象は BeagleBoard-xM であるが，他の製品用のソースコードも解析が可能であることを示すため，I^2C ドライバのすべてのファイル，つまり KERNEL/driver/i2c 以下の 128 ファイル，合計 67960 ステップ [1] に対して解析を行った．対象ファイルを表 5.1 に示す．

表 5.1: 対象ファイル一覧

ファイル	ステップ数
KERNEL/drivers/i2c/algos/i2c-algo-bit.c	665
KERNEL/drivers/i2c/algos/i2c-algo-pca.c	561
KERNEL/drivers/i2c/algos/i2c-algo-pca.mod.c	37
KERNEL/drivers/i2c/algos/i2c-algo-pcf.c	436
KERNEL/drivers/i2c/busses/i2c-acorn.c	96
KERNEL/drivers/i2c/busses/i2c-ali1535.c	536
KERNEL/drivers/i2c/busses/i2c-ali1563.c	443

次ページに続く

[1] このステップ数はコメントを含む．なおインクルードされるヘッダファイルも解析対象となるが，ヘッダファイルのステップ数は含めていない．

表 5.1 – 対象ファイル一覧（続き）

ファイル	ステップ数
KERNEL/drivers/i2c/busses/i2c-ali15x3.c	517
KERNEL/drivers/i2c/busses/i2c-amd756-s4882.c	254
KERNEL/drivers/i2c/busses/i2c-amd756.c	413
KERNEL/drivers/i2c/busses/i2c-amd8111.c	492
KERNEL/drivers/i2c/busses/i2c-at91.c	1129
KERNEL/drivers/i2c/busses/i2c-au1550.c	427
KERNEL/drivers/i2c/busses/i2c-axxia.c	601
KERNEL/drivers/i2c/busses/i2c-bcm-iproc.c	517
KERNEL/drivers/i2c/busses/i2c-bcm-kona.c	907
KERNEL/drivers/i2c/busses/i2c-bcm2835.c	335
KERNEL/drivers/i2c/busses/i2c-bfin-twi.c	740
KERNEL/drivers/i2c/busses/i2c-brcmstb.c	694
KERNEL/drivers/i2c/busses/i2c-cadence.c	1005
KERNEL/drivers/i2c/busses/i2c-cbus-gpio.c	303
KERNEL/drivers/i2c/busses/i2c-cpm.c	725
KERNEL/drivers/i2c/busses/i2c-cros-ec-tunnel.c	323
KERNEL/drivers/i2c/busses/i2c-davinci.c	926
KERNEL/drivers/i2c/busses/i2c-designware-baytrail.c	162
KERNEL/drivers/i2c/busses/i2c-designware-core.c	861
KERNEL/drivers/i2c/busses/i2c-designware-pcidrv.c	345
KERNEL/drivers/i2c/busses/i2c-designware-platdrv.c	404
KERNEL/drivers/i2c/busses/i2c-digicolor.c	384
KERNEL/drivers/i2c/busses/i2c-diolan-u2c.c	528
KERNEL/drivers/i2c/busses/i2c-diolan-u2c.mod.c	49
KERNEL/drivers/i2c/busses/i2c-dln2.c	263
KERNEL/drivers/i2c/busses/i2c-dln2.mod.c	40
KERNEL/drivers/i2c/busses/i2c-efm32.c	484
KERNEL/drivers/i2c/busses/i2c-eg20t.c	935
KERNEL/drivers/i2c/busses/i2c-elektor.c	343
KERNEL/drivers/i2c/busses/i2c-emev2.c	332
KERNEL/drivers/i2c/busses/i2c-exynos5.c	875
KERNEL/drivers/i2c/busses/i2c-gpio.c	290
KERNEL/drivers/i2c/busses/i2c-highlander.c	481
KERNEL/drivers/i2c/busses/i2c-hix5hd2.c	557

次ページに続く

表 5.1 – 対象ファイル一覧（続き）

ファイル	ステップ数
KERNEL/drivers/i2c/busses/i2c-hydra.c	158
KERNEL/drivers/i2c/busses/i2c-i801.c	1471
KERNEL/drivers/i2c/busses/i2c-ibm_iic.c	811
KERNEL/drivers/i2c/busses/i2c-img-scb.c	1414
KERNEL/drivers/i2c/busses/i2c-imx.c	1124
KERNEL/drivers/i2c/busses/i2c-iop3xx.c	528
KERNEL/drivers/i2c/busses/i2c-isch.c	322
KERNEL/drivers/i2c/busses/i2c-ismt.c	1002
KERNEL/drivers/i2c/busses/i2c-jz4780.c	834
KERNEL/drivers/i2c/busses/i2c-kempld.c	409
KERNEL/drivers/i2c/busses/i2c-lpc2k.c	513
KERNEL/drivers/i2c/busses/i2c-meson.c	492
KERNEL/drivers/i2c/busses/i2c-mpc.c	856
KERNEL/drivers/i2c/busses/i2c-mt65xx.c	744
KERNEL/drivers/i2c/busses/i2c-mv64xxx.c	1001
KERNEL/drivers/i2c/busses/i2c-mxs.c	919
KERNEL/drivers/i2c/busses/i2c-nforce2-s4985.c	249
KERNEL/drivers/i2c/busses/i2c-nforce2.c	451
KERNEL/drivers/i2c/busses/i2c-nomadik.c	1135
KERNEL/drivers/i2c/busses/i2c-ocores.c	561
KERNEL/drivers/i2c/busses/i2c-ocores.mod.c	63
KERNEL/drivers/i2c/busses/i2c-octeon.c	633
KERNEL/drivers/i2c/busses/i2c-omap.c	1553
KERNEL/drivers/i2c/busses/i2c-opal.c	294
KERNEL/drivers/i2c/busses/i2c-parport-light.c	276
KERNEL/drivers/i2c/busses/i2c-parport.c	331
KERNEL/drivers/i2c/busses/i2c-pasemi.c	418
KERNEL/drivers/i2c/busses/i2c-pca-isa.c	225
KERNEL/drivers/i2c/busses/i2c-pca-platform.c	290
KERNEL/drivers/i2c/busses/i2c-pca-platform.mod.c	63
KERNEL/drivers/i2c/busses/i2c-piix4.c	700
KERNEL/drivers/i2c/busses/i2c-pmcmsp.c	617
KERNEL/drivers/i2c/busses/i2c-pnx.c	778
KERNEL/drivers/i2c/busses/i2c-powermac.c	468

次ページに続く

表 5.1 – 対象ファイル一覧（続き）

ファイル	ステップ数
KERNEL/drivers/i2c/busses/i2c-puv3.c	281
KERNEL/drivers/i2c/busses/i2c-pxa-pci.c	168
KERNEL/drivers/i2c/busses/i2c-pxa.c	1344
KERNEL/drivers/i2c/busses/i2c-qup.c	774
KERNEL/drivers/i2c/busses/i2c-rcar.c	741
KERNEL/drivers/i2c/busses/i2c-riic.c	426
KERNEL/drivers/i2c/busses/i2c-rk3x.c	1043
KERNEL/drivers/i2c/busses/i2c-robotfuzz-osif.c	202
KERNEL/drivers/i2c/busses/i2c-robotfuzz-osif.mod.c	44
KERNEL/drivers/i2c/busses/i2c-s3c2410.c	1367
KERNEL/drivers/i2c/busses/i2c-scmi.c	432
KERNEL/drivers/i2c/busses/i2c-sh7760.c	564
KERNEL/drivers/i2c/busses/i2c-sh_mobile.c	1045
KERNEL/drivers/i2c/busses/i2c-sibyte.c	194
KERNEL/drivers/i2c/busses/i2c-simtec.c	167
KERNEL/drivers/i2c/busses/i2c-simtec.mod.c	47
KERNEL/drivers/i2c/busses/i2c-sirf.c	467
KERNEL/drivers/i2c/busses/i2c-sis5595.c	430
KERNEL/drivers/i2c/busses/i2c-sis630.c	557
KERNEL/drivers/i2c/busses/i2c-sis96x.c	326
KERNEL/drivers/i2c/busses/i2c-st.c	878
KERNEL/drivers/i2c/busses/i2c-stu300.c	1013
KERNEL/drivers/i2c/busses/i2c-sun6i-p2wi.c	343
KERNEL/drivers/i2c/busses/i2c-taos-evm.c	314
KERNEL/drivers/i2c/busses/i2c-taos-evm.mod.c	59
KERNEL/drivers/i2c/busses/i2c-tegra.c	975
KERNEL/drivers/i2c/busses/i2c-tiny-usb.c	290
KERNEL/drivers/i2c/busses/i2c-tiny-usb.mod.c	48
KERNEL/drivers/i2c/busses/i2c-versatile.c	160
KERNEL/drivers/i2c/busses/i2c-via.c	163
KERNEL/drivers/i2c/busses/i2c-viapro.c	507
KERNEL/drivers/i2c/busses/i2c-viperboard.c	473
KERNEL/drivers/i2c/busses/i2c-viperboard.mod.c	46
KERNEL/drivers/i2c/busses/i2c-wmt.c	476

次ページに続く

表 5.1 – 対象ファイル一覧（続き）

ファイル	ステップ数
KERNEL/drivers/i2c/busses/i2c-xgene-slimpro.c	470
KERNEL/drivers/i2c/busses/i2c-xiic.c	823
KERNEL/drivers/i2c/busses/i2c-xlp9xx.c	445
KERNEL/drivers/i2c/busses/i2c-xlr.c	274
KERNEL/drivers/i2c/busses/scx200_acb.c	608
KERNEL/drivers/i2c/i2c-boardinfo.c	89
KERNEL/drivers/i2c/i2c-core.c	3168
KERNEL/drivers/i2c/i2c-dev.c	675
KERNEL/drivers/i2c/i2c-mux.c	219
KERNEL/drivers/i2c/i2c-slave-eeprom.c	169
KERNEL/drivers/i2c/i2c-smbus.c	246
KERNEL/drivers/i2c/i2c-stub.c	421
KERNEL/drivers/i2c/muxes/i2c-arb-gpio-challenge.c	249
KERNEL/drivers/i2c/muxes/i2c-arb-gpio-challenge.mod.c	52
KERNEL/drivers/i2c/muxes/i2c-mux-gpio.c	290
KERNEL/drivers/i2c/muxes/i2c-mux-pca9541.c	399
KERNEL/drivers/i2c/muxes/i2c-mux-pca954x.c	313
KERNEL/drivers/i2c/muxes/i2c-mux-pinctrl.c	278
KERNEL/drivers/i2c/muxes/i2c-mux-reg.c	290

5.2 解析対象外のマクロ

本研究で作成した静的コード解析ツールでは，マクロの情報を残したまま構文解析を行うが，第 3 章で述べた通り，構文木の作成を阻害するようなマクロの解析は行えない．このようなマクロは出現時に警告を表示して解析対象から除外する．解析対象から除外されたマクロの一覧を表 5.2 に示す．また，除外されたマクロのうちの幾つかを小節 5.2.1 以降で例示する．

表 5.2: 解析対象外のマクロ一覧

除外したトークン列	出現数
ACPI_MODULE_NAME	1

次ページに続く

表 5.2 – 解析対象から除外されたマクロ一覧（続き）

除外したトークン列	出現数
DEB2	33
DEB3	5
DEBPROTO	4
DEVICE_ATTR	5
DEVICE_ATTR_IGNORE_LOCKDEP	1
MODULE_ALIAS	47
MODULE_AUTHOR	114
MODULE_DESCRIPTION	112
MODULE_DEVICE_TABLE	82
MODULE_INFO	22
MODULE_LICENSE	115
MODULE_PARM_DESC	40
SET_RUNTIME_PM_OPS	5
SET_SYSTEM_SLEEP_PM_OPS	2
SIMPLE_DEV_PM_OPS	12
UNIVERSAL_DEV_PM_OPS	1
__attribute__	33
__exit	32
__force	1
__init	44
__initdata	1
__iomem	134
__maybe_unused	2
__packed	5
__used	22
__user	8
__visible	11
clamp_t	1
container_of	13
dev_get_drvdata	29
for_each_available_child_of_node	1
for_each_child_of_node	5
for_each_set_bit	2
list_for_each_entry	3

次ページに続く

表 5.2 – 解析対象から除外されたマクロ一覧（続き）

除外したトークン列	出現数
list_for_each_entry_safe	4
min_t	13
module_param	49
module_param_array	7
noinline	2

5.2.1 SET_RUNTIME_PM_OPS マクロ

SET_RUNTIME_PM_OPS マクロは構造体のメンバの初期化の一部分であり，完全な式 (expression) や文 (statement) とはなっていないため解析対象から除外される．

```
KERNEL/include/linux/pm.h:
 358: #define SET_RUNTIME_PM_OPS(suspend_fn, resume_fn, idle_fn) \
 359:     .runtime_suspend = suspend_fn, \
 360:     .runtime_resume = resume_fn, \
 361:     .runtime_idle = idle_fn,
```

使用している箇所の例を以下に示す．

```
KERNEL/driver/i2c/busses/i2c-omap.c
1517: static struct dev_pm_ops omap_i2c_pm_ops = {
1518:  SET_RUNTIME_PM_OPS(omap_i2c_runtime_suspend,
1519:     omap_i2c_runtime_resume, NULL)
1520: };
```

5.2.2 list_for_each_entry マクロ

以下に list_for_each_entry マクロの定義を示す．このマクロは for キーワードと条件式の部分のみを含んでおり，for ループ本体が含まれていないため，完全な式 (expression) や文 (statement) にはなっていないため解析対象から除外される．なお，list_for_each_entry マクロは複数のヘッダファイルで定義されており，どのヘッダファイルがインクルードされるかを正確に把握していなければ，異なる定義を参照してしまう可能性もある．

```
KERNEL/include/linux/list.h:
 314: #define list_for_each_entry(pos, head, member)                  \
 315:     for (pos = list_first_entry(head, typeof(*pos), member);  \
 316:          &pos->member != (head);                                \
 317:          pos = list_next_entry(pos, member))
```

使用している箇所の例を以下に示す．関数形式マクロは一見関数のように見えるため，関数の直後にループの本体が続いており，ソースコードの読み手を混乱させる可能性が高い．このようなマクロは優先的にドキュメント化しプロジェクト内で情報を共有すべきである．実際に Linux ではこのようなマクロの使用方法は必ずドキュメントが用意されている．

```
KERNEL/drivers/i2c/i2c-dev.c:
  66:    list_for_each_entry(i2c_dev, &i2c_dev_list, list) {
  67:        if (i2c_dev->adap->nr == index)
  68:            goto found;
  69:    }
```

5.3 解析結果

I^2C ドライバの解析結果を表 5.3 にまとめた．基本型の使用や if,for,while の波括弧がないことについての警告が多く検出されているが，これらは Linux のコーディング規約では許されている部分であり問題はない．それ以外の部分で可読性に問題がありそうな警告を以降の小節で説明する．

表 5.3: token の構成要素

項目	件数
基本型を使用している	470
if, for, while の本体が波括弧で囲まれていない	225
if 文に else 節が存在しない	153
条件式が真偽値でない	136
二項演算子の左右の式の型の不一致	118
goto 文, continue 文	106
二項演算子の混在	45
条件演算子の条件式の括弧の省略	45
論理否定演算子を適用する式が真偽値でない	39
switch 文内のフォールスルー	20
宣言文に複数の変数が含まれている	10
if, for, while の条件式に代入が含まれている	7
カンマ演算子が使用されている	5
下線で始まる名前の宣言または定義	4

5.3.1 条件式が真偽値でない

以下の警告箇所 (109 行目) の if 文の条件式には amd_ec_wait_write 関数の戻り値を格納した int 型の変数 status が与えられている．amd_ec_wait_write 関数は 0 または-ETIMEDOUT を返す関数であり，真偽値ではないため警告が発生している．

```
KERNEL/driver/i2c/busses/i2c-amd8111.c:
 103: static int amd_ec_read(struct amd_smbus *smbus, unsigned char address,
 104:          unsigned char *data)
 105: {
 106:      int status;
 107:
 108:      status = amd_ec_wait_write(smbus);
 109:      if (status)
               ^^^^^^
 110:          return status;
 111:      outb(AMD_EC_CMD_RD, smbus->base + AMD_EC_CMD);
```

条件式を (status != 0) とした方が，0 ならば成功して処理を継続するという意図を伝えやすい．修正案を以下に示す．

```
KERNEL/driver/i2c/busses/i2c-amd8111.c:
 103: static int amd_ec_read(struct amd_smbus *smbus, unsigned char address,
 104:          unsigned char *data)
 105: {
 106:      int status;
 107:
 108:      status = amd_ec_wait_write(smbus);
 109:      if (status != 0)
 110:          return status;
 111:      outb(AMD_EC_CMD_RD, smbus->base + AMD_EC_CMD);
```

5.3.2 二項演算子の左右の式の型の不一致

以下の警告箇所 (138 行目) の二項演算子 (&&) の左右の型は真偽値であるべきだが，左辺 (temp & GS_HST_STS) は int 型であり一致しない.

```
KERNEL/driver/i2c/busses/i2c-amd756.c:
 134:      /* We will always wait for a fraction of a second! */
 135:      do {
 136:          msleep(1);
 137:          temp = inw_p(SMB_GLOBAL_STATUS);
 138:      } while ((temp & GS_HST_STS) && (timeout++ < MAX_TIMEOUT));
                    ^^^^^^^^^^^^^^^^^^^^    ^^^^^^^^^^^^^^^^^^^^^^^^^^
```

左辺 (temp & GS_HST_STS) はビット演算の結果が 0 以外かを確認する式なので，明示的に 0 との比較を行うべきである．修正案を以下に示す.

```
KERNEL/driver/i2c/busses/i2c-amd756.c:
 134:      /* We will always wait for a fraction of a second! */
 135:      do {
 136:          msleep(1);
 137:          temp = inw_p(SMB_GLOBAL_STATUS);
 138:      } while (((temp & GS_HST_STS) != 0) && (timeout++ < MAX_TIMEOUT));
```

5.3.3 if, for, while の条件式に代入が含まれている

以下の警告箇所 (177 行目) では if 文の条件式にて kzalloc 関数の戻り値（ポインタ）を s4882_adapter 変数に代入した後，論理否定演算子を使用して NULL かどうかを判定している．複数の処理を条件式内で同時に行っているため，わかりにくい表現となっている.

また，ポインタは真偽値 (TRUE,FALSE) ではないため,「論理否定演算子を適用する式が真偽値でない」の警告も発生する．

```
KERNEL/driver/i2c/busses/i2c-amd756-s4882.c:
 177:     if (!(s4882_adapter = kzalloc(5 * sizeof(struct i2c_adapter),
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
 178:                           GFP_KERNEL))) {
                                ^^^^^^^^^^^^
 179:         error = -ENOMEM;
 180:         goto ERROR1;
 181:     }
```

可読性を向上させるためには，代入とエラー判定を分割し，エラー判定では NULL と明示的に比較すべきである．修正案を以下に示す．

```
KERNEL/driver/i2c/busses/i2c-amd756-s4882.c:
 177:     s4882_adapter = kzalloc(5 * sizeof(struct i2c_adapter),
 178:                             GFP_KERNEL)
 179:     if (s4882_adapter == NULL) {
 180:         error = -ENOMEM;
 181:         goto ERROR1;
 182:     }
```

5.3.4 二項演算の型が異なる

変数 temp は以下のように要素数 2 の配列として宣言されている．

```
unsigned char temp[2];
```

しかし，以下の警告の箇所 (351 行目) では変数 temp をポインタとして使用している．なお，349 行目では temp[0] のように配列としてアクセスしており，記述方法にばらつきがみられる．

```
KERNEL/drivers/i2c/busses/i2c-amd8111.c:
 349:    if (~temp[0] & AMD_SMB_STS_DONE) {
 350:        udelay(500);
 351:        status = amd_ec_read(smbus, AMD_SMB_STS, temp + 0);
                                                     ^^^^^^^^
 352:        if (status)
 353:            return status;
 354:    }
```

配列へのアクセスであることを明示するために配列の添字演算子 ([]) を使用すべきである．修正案を以下に示す．

KERNEL/drivers/i2c/busses/i2c-amd8111.c:

```
 349:    if (~temp[0] & AMD_SMB_STS_DONE) {
 350:        udelay(500);
 351:        status = amd_ec_read(smbus, AMD_SMB_STS, &temp[0]);
 352:        if (status)
 353:            return status;
 354:    }
```

第6章　まとめ

本研究ではC言語の静的コード解析において，プリプロセス前のソースコードに対する解析を行う手法を提案し，保守性や可読性の高いソースコードを記述するための検査を行う静的コード解析ツールを作成した．組込みシステムで使用されるデバイスドライバに対して静的コード解析を行い，保守性や可読性を低下させるコード片が検出できることが確認できた．

本研究で作成した静的コード解析ツールをより実用的なものとするためには，さらに多くの検査項目を実装していく必要がある．また，コンパイル時に分岐が発生しないマクロは積極的に展開し，関数や変数の取りうる値に関する検査を行うなど，解析手法にも改良の余地が残されている．さらに，今回はファイル単位での解析を行ったが，ファイル単位の解析情報をもとにファイル間の情報の整合性を確認することも有用である．

なお，プリプロセス情報を持った構文木や型の検査に使用している型情報はドキュメントの自動作成にも有用であると予想される．

参考文献

[1] Brian W. Kernighan & Dennis M. Ritchie, The C Programming Language, 1978.

[2] ANSI, American National Standard for Information Systems -Programming Language-C, X3.159-1989.

[3] ISO/IEC, ISO/IEC 9899 : 1990(E) Programming Languages-C

[4] JIS, JIS X3010-1993 プログラム言語 C 1993/10

[5] ISO/IEC, ISO/IEC 9899:1999(E) Programming Language–C 1999/12

[6] JIS, JIS X3010-2003 プログラム言語 C

[7] ISO/IEC, ISO/IEC 9899:2011 Programming Language–C 2011/12

[8] ISO/IEC, ISO/IEC 9899:201x Programming languages — C, http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1494.pdf.

[9] JIS, JIS X 25010：2013(ISO/IEC 25010：2011) システム及びソフトウェア製品の品質要求及び評価（SQuaRE）－システム及びソフトウェア品質モデル.

[10] 独立行政法人 情報処理推進機構 ソフトウェア・エンジニアリング・センター, 改訂版 組込みソフトウェア開発向けコーディング作法ガイド [C 言語版], http://www.ipa.go.jp/files/000005123.pdf, 2007 IPA.

[11] Bryan O'Sullivan, John Goerzen, Don Stewart, (山下伸夫, 伊藤勝利, (株) タイムインターメディア 訳), Real World Haskell, オライリー・ジャパン, 2009.

[12] Andrew W. Appel, (神林靖, 滝本宗宏 訳), 最新コンパイラ構成技法, 翔泳社, 2009.

[13] Benedikt Huber, https://hackage.haskell.org/package/language-c, 2014.

謝辞

本研究を進め課題研究報告書をまとめるにあたり，叱咤激励とともに辛抱強くご指導いただきました田中清史准教授に心から感謝いたします．また，課題研究計画提案発表等で貴重なご助言をいただきました井口寧教授，金子峰雄教授，副テーマの指導をしていただきました石原哉教授に感謝致します。

付 録A　左再帰除去版の構文解析器

3.5.2 節で説明した左再帰の除去を C 言語の文法に適用したときのパーサの図 (railway-diagram) を付録として掲載する．なお，図中に出現する sub パーサは，そのパーサの内部でのみ参照される部分パーサであり，他のパーサに出現する sub パーサとは異なることとする．

A.1　式 (expressions)

A.1.1　primary-expression

primary-expression は postfix-expression パーサの一部であり，式内の識別子 (identifer)，定数リテラル (constant)，文字列リテラル (string-literal) および括弧で括られた式を受理するパーサである．また，GNU C(gcc) 拡張構文の複合式 (compound-statement) も primary-expression で受理する．

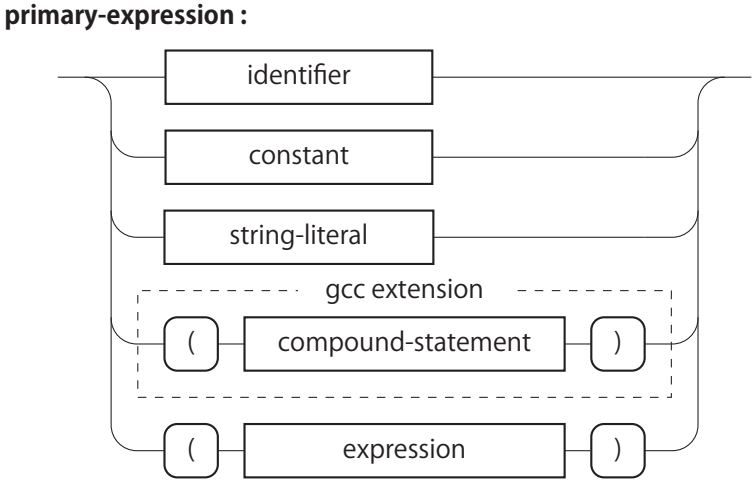


図 A.1: primary-expression

例 A.1.1 (primary-expression)

primary-expression パーサが受理する式の例を以下に示す．

```
id                                  // identifier
10                                  // constant
"hello world."                      // string-literal
({ int a = 1; int b = 2; (a + b); }) // ( compound-statement )
(id + 10)                           // ( expression )
```

A.1.2 postfix-expression

postfix-expression は unary-expression パーサの一部であり，primary-expression または丸括弧で括られた型と波括弧で括られた initializer-list の並びに 0 個以上の後置演算子が付加された式を受理するパーサである．後置演算子には配列参照，関数呼び出し，インクリメント，デクリメント，メンバ参照（ドット，アロー）が存在する．

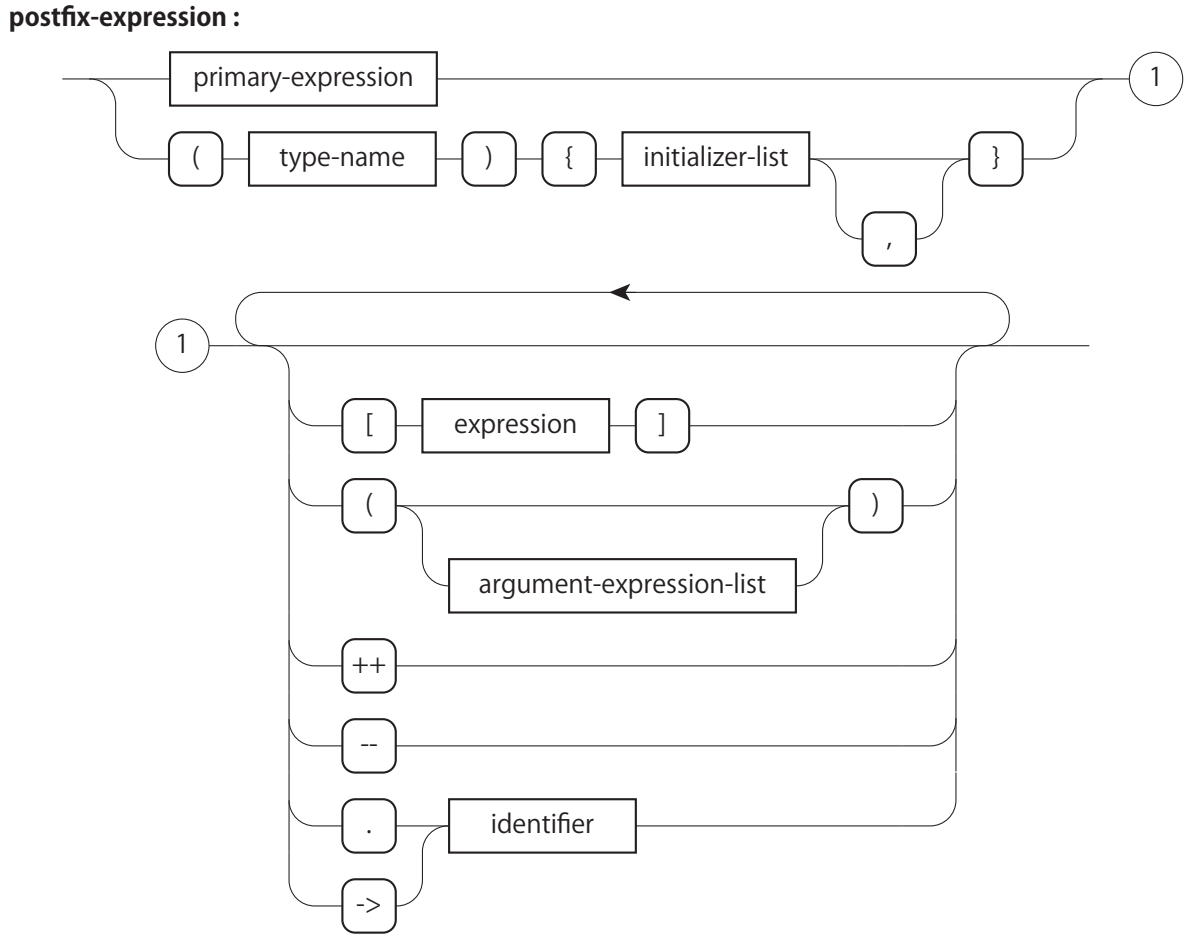


図 A.2: postfix-expression

例 A.1.2 (postfix-expression)

postfix-expression パーサが受理する式の例を以下に示す.

```
id                    // primary-expression
i ++                  // primary-expression ++
arr [ 10 ]            // primary-expression [ expression ]
arr [ 1 ] [ 2 ]       // primary-expression [ expression ] [ expression ]
f ( arg1, arg2 )      // primary-expression ( argument-expression-list )
st . member           // primary-expression . identifier
st -> member --       // primary-expression -> identifier --
st . member1 . member2 // primary-expression . identifier . identifier
(int []){ 10, 20, 30 } // ( type-name ) { initializer-list }
(struct { int x; int y; }){ 10, 20, } // ( type-name ) { initializer-list , }
```

A.1.3 argument-expression-list

argument-expression-list は postfix-expression パーサの一部であり，関数呼び出しにおける実引数の並びを受理するパーサである．各実引数は代入式 (assignment-expression) である.

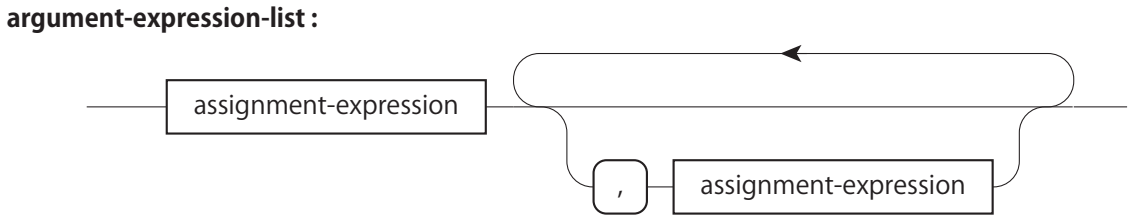


図 A.3: argument-expression-list

例 A.1.3 (argument-expression-list)

argument-expression-list パーサが受理する式の例を以下に示す.

```
arg1           // assignment-expression
arg1 , arg2    // assignment-expression , assignment-expression
x = 10         // assignment-expression
a << 8, (int)c // assignment-expression , assignment-expression
```

A.1.4 unary-expression

unary-expression は cast-expression パーサと assignment-expression パーサの一部であり，単項演算子を伴う式を受理するパーサである.

unary-expression :

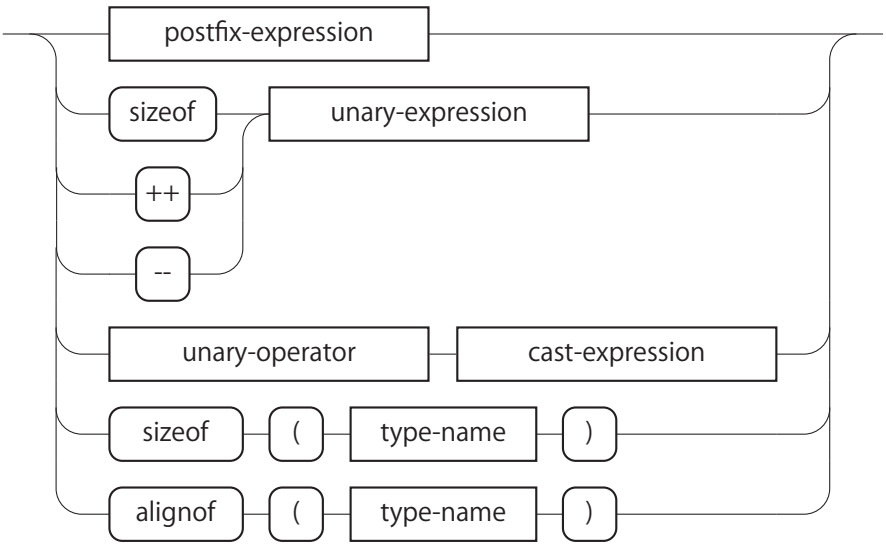


図 A.4: unary-expression

例 A.1.4 (unary-expression)

unary-expression パーサが受理する式の例を以下に示す.

```
i ++                    // postfix-expression
sizeof arr [ 10 ]       // sizeof unary-expression
++ *p                   // ++ unary-expression
++ sizeof i             // ++ unary-expression
&x                      // unary-operator
sizeof ( int )          // sizeof ( type-name )
```

A.1.5 unary-operator

unary-operator は unary-expression パーサの一部であり，単項演算子を受理するパーサである．&&演算子は GCC 拡張の一種であり，goto で使用するラベルのアドレスを求める演算子である.

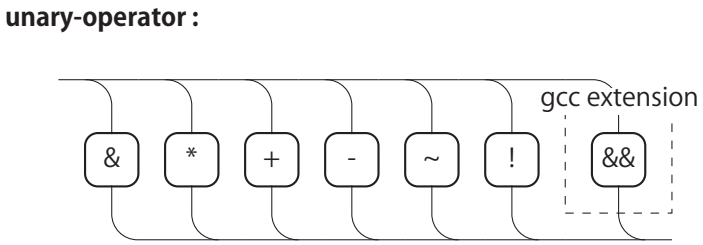


図 A.5: cast-expression

A.1.6 cast-expression

cast-expression は unary-expression パーサと binary-operation-expression パーサの一部であり，キャスト演算子を伴う式を受理するパーサである．

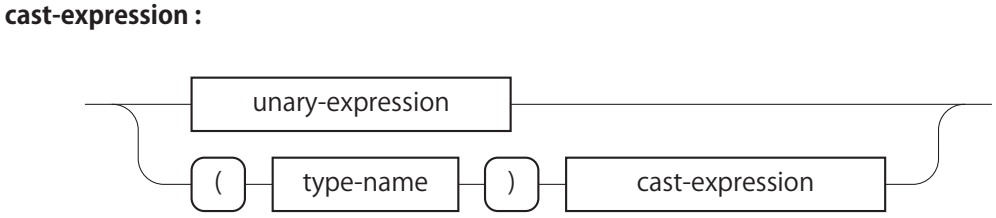


図 A.6: cast-expression

例 A.1.5 (cast-expression)

cast-expression パーサが受理する式の例を以下に示す．

```
+x                  // unary-expression
(int) x             // ( type-name ) cast-expression
(int *) (void *) p // ( type-name ) cast-expression
```

A.1.7 binary-operation-expression

binary-operation-expression は conditional-expression パーサの一部であり，二項演算を行う式を受理するパーサである．

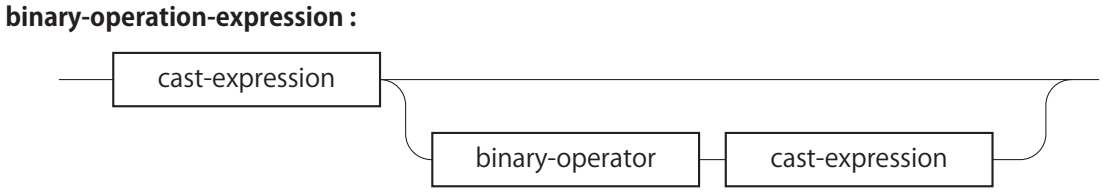


図 A.7: binary-operation-expression

例 A.1.6 (binary-operation-expression)

binary-operation-expression パーサが受理する式の例を以下に示す.

```
(short)10   // cast-expression
10 + 10     // cast-expression binary-operator cast-expression
10 + 10 * 2 // cast-expression binary-operator cast-expression
```

A.1.8 binary-operator

binary-operator は binary-operation-expression パーサの一部であり，二項演算子を受理するパーサである.

表 A.1: 二項演算子一覧

*	/	%	+	-	<<	>>	<	>
<=	>=	==	!=	&	^	|	&&	||

A.1.9 conditional-expression

conditional-expression は assignment-expression パーサと constant-expression パーサの一部であり，参考演算子を用いた式を受理するパーサである.

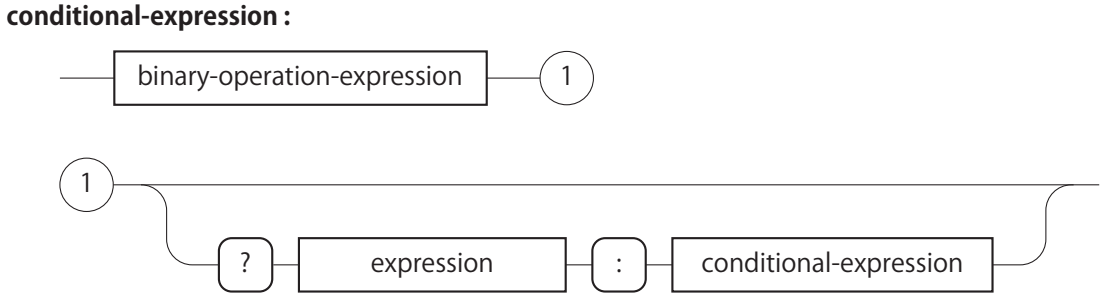


図 A.8: conditional-expression

例 A.1.7 (conditional-expression)

conditional-expression パーサが受理する式の例を以下に示す.

```
a + b          // binary-operation-expression
a == b ? 0 : 1 // binary-operation-expression
               //     ? expression : conditional-expression
```

A.1.10 assignment-expression

assignment-expression は argument-expression-list パーサ，expression パーサ，type-specifier パーサ，direct-declarator パーサ，direct-abstract-declarator パーサ，initializer パーサの一部であり，代入式を受理するパーサである．

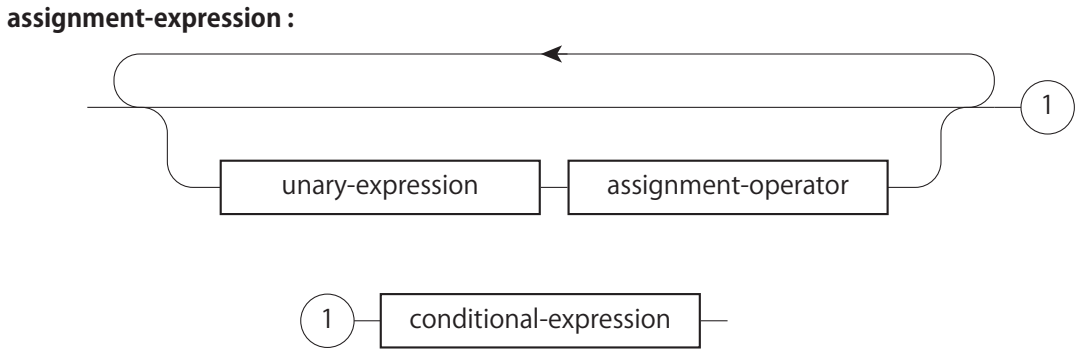


図 A.9: assignment-expression

例 A.1.8 (assignment-expression)

assignment-expression パーサが受理する式の例を以下に示す．

```
a                       // conditional-expression
a[0] = 10               // unary-expression
                        //     assignment-operator conditional-expression
*p = (b == c) ? 0 : 1 // (the same as above)
```

A.1.11 assignment-operator

assignment-operator は assignment-expression の一部であり代入演算子を受理するパーサである．

表 A.2: 代入演算子一覧

=	*=	/=	%=	+=	-=	<<=	>>=
&=	^=	|=					

A.1.12 expression

expression は primary-expression パーサ，postfix-expression パーサ，conditional-expression パーサ，expression-statement パーサ，selection-statement パーサ，iteration-statement パー

サ，jump-statement パーサの一部であり，カンマ演算子で区切られた式を受理するパーサである．

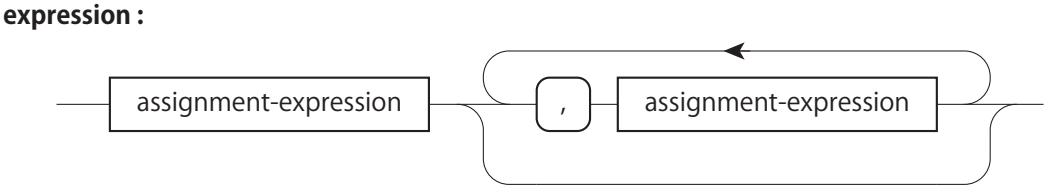


図 A.10: expression

例 A.1.9 (expression)

expression パーサが受理する式の例を以下に示す．

```
a = 10          // assignment-expression
a = 10, b = 20 // assignment-expression , assignment-expression
```

A.1.13 constant-expression

constant-expression パーサは struct-declarator パーサ，enumerator パーサ，alignment-specifier パーサ，designator パーサ，static_assert-declaration パーサ，labeled-statement パーサの一部であり，受理する式は conditional-expression パーサと等価である．

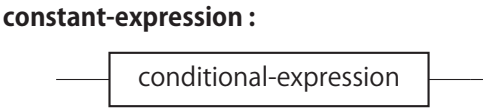


図 A.11: constant-expression

A.2 宣言 (declarations)

A.2.1 declaration

declaration は block-item パーサ，iteration-statement パーサ，external-declaration パーサ，declaration-list パーサの一部であり，宣言を受理するパーサである．

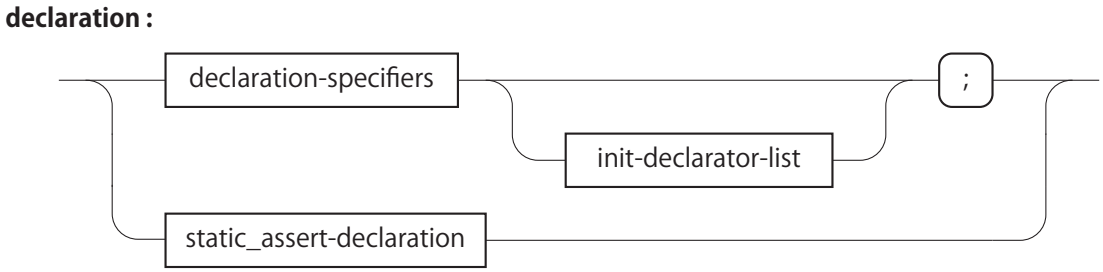


図 A.12: declaration

例 A.2.1 (declaration)

declaration パーサが受理する式の例を以下に示す.

```
struct { int a; } ;  // declaration-specifiers ;
int x = 10, y = 11 ; // declaration-specifiers iint-declarator-list ;
_Static_assert ( a == b , "assert" ) ; // static_assert-declaration
```

A.2.2 declaration-specifier

declaration-specifier は declaration パーサ，parameterDeclaration パーサ，functionDefinition パーサの一部であり，宣言の型を受理するパーサである.

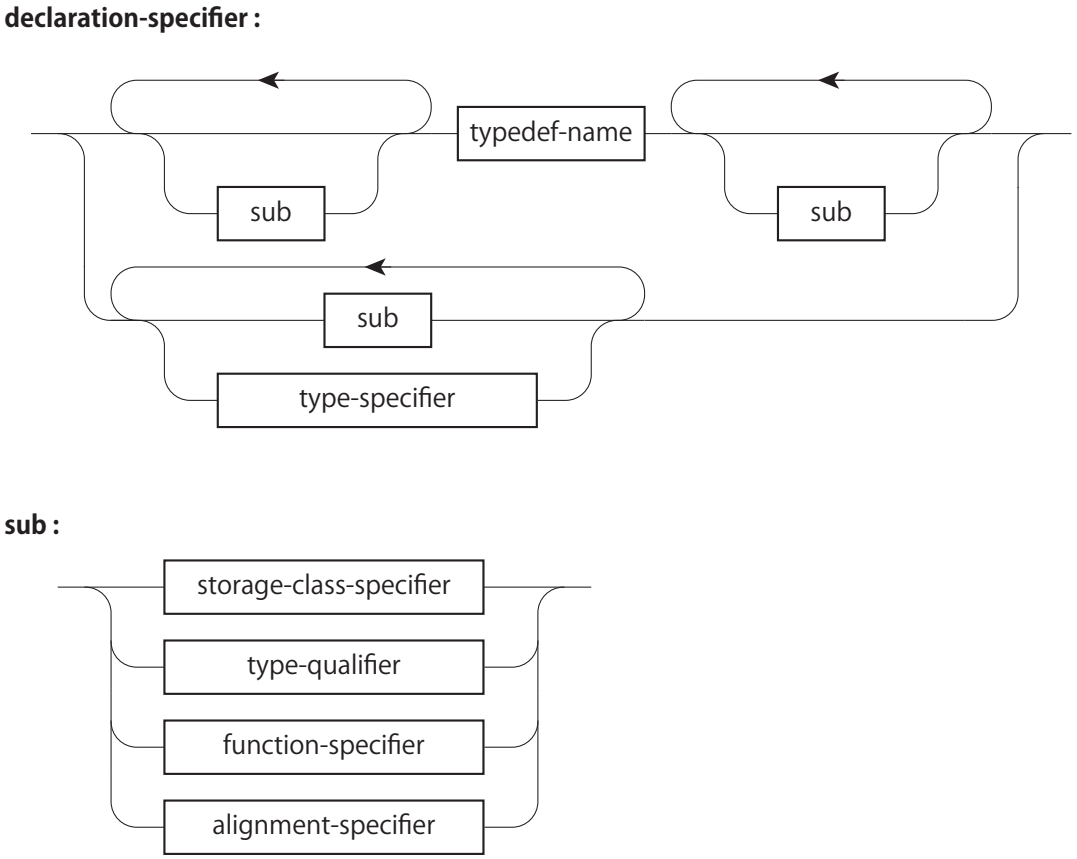


図 A.13: declaration-specifier

例 A.2.2 (declaration-specifier)

declaration-specifier パーサが受理する式の例を以下に示す.

```
T            // typedef-name
extern T     // storage-class-specifier typedef-name
const T      // type-qualifier typedef-name
T const      // typedef-name type-qualifier
int          // type-specifier
volatile int // type-qualifier type-specifier
short int    // type-specifier type-specifier
long long    // type-specifier type-specifier
```

A.2.3 init-declarator-list

init-declarator-list は declaration パーサの一部であり，カンマで区切った宣言子（変数名，配列の要素数，ポインタ，および初期値）を受理するパーサである.

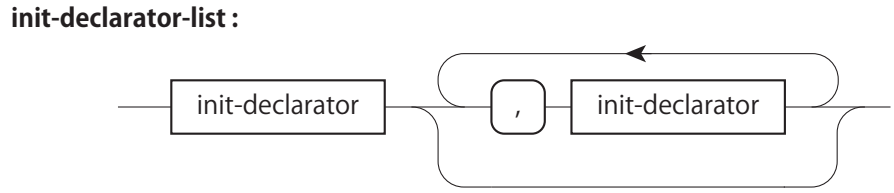


図 A.14: init-declarator-list

例 A.2.3 (init-declarator-list)

init-declarator-list パーサが受理する式の例を以下に示す.

```
x = 10               // init-declarator
x, * y               // init-declarator , init-declarator
x = 10, * const y, z // init-declarator , init-declarator , init-declarator
```

A.2.4 init-declarator

init-declarator は init-declarator-list パーサの一部であり，宣言子（変数名，配列の要素数，ポインタ，および初期値）を受理するパーサである.

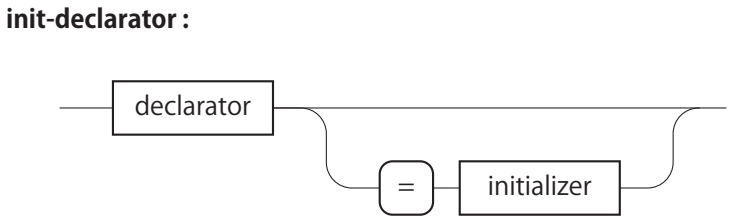


図 A.15: init-declarator-list

例 A.2.4 (init-declarator)

init-declarator パーサが受理する式の例を以下に示す.

```
x              // declarator
x = 10         // declarator = initializer
x = { 10, 20 } // declarator = initializer
```

A.2.5 storage-class-specifier

storage-class-specifier は declarationSpecifiers パーサの一部であり，typedef,extern,static,_Thread_local,auto,register キーワードのいずれかを受理するパーサである.

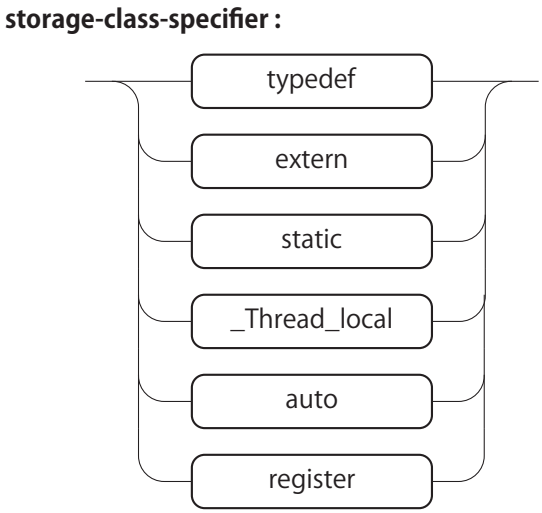


図 A.16: storage-class-specifier

A.2.6 type-specifier

type-specifier は declaration-specifiers パーサ，specifier-qualifier-list パーサの一部であり，組込み型または構造体型，共用体型，列挙体型，typedef で定義された型を受理するパーサである．

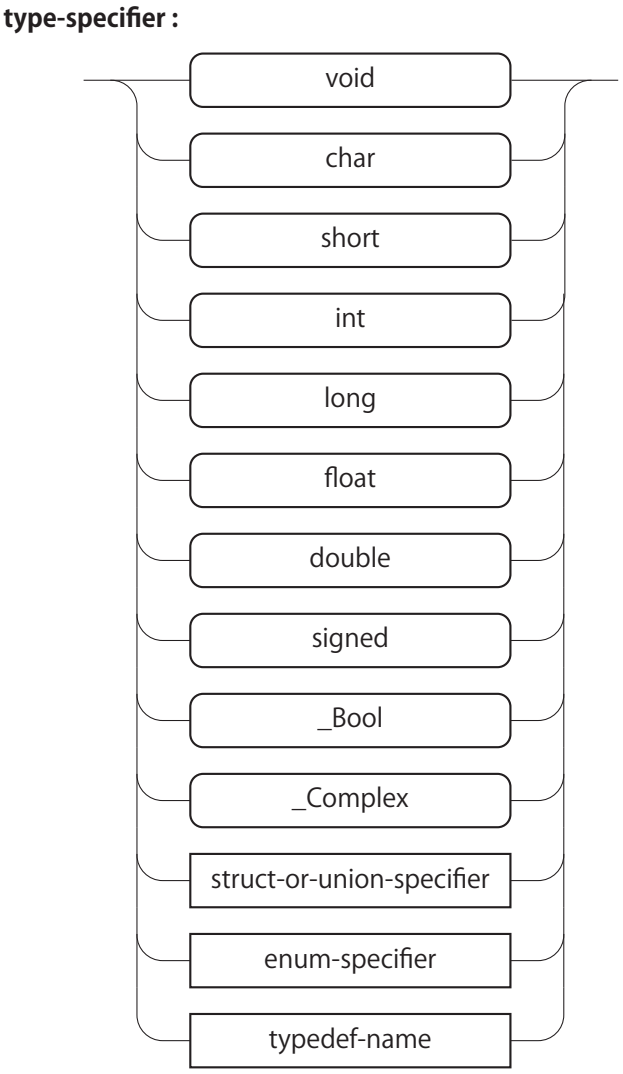


図 A.17: type-specifier

例 A.2.5 (type-specifier)

type-specifier パーサが受理する式の例を以下に示す.

```
void                    // void
int                     // int
struct T_ST             // struct-or-union-specifier
struct T_ST { int x ; } // struct-or-union-specifier
enum T_EN { A, B, C }   // enum-specifier
T                       // typedef-name
```

A.2.7 struct-or-union-specifier

struct-or-union-specifier は type-specifier パーサの一部であり，構造体型を受理するパーサである．

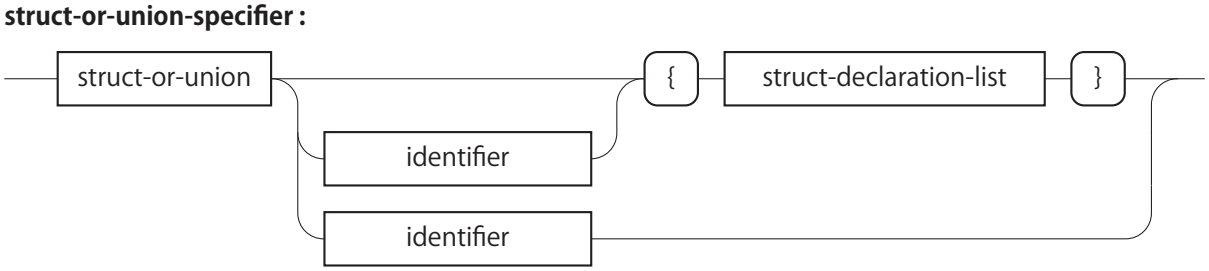


図 A.18: struct-or-union-specifier

例 A.2.6 (struct-or-union-specifier)

struct-or-union-specifier パーサが受理する式の例を以下に示す．

```
struct T_ST                     // struct-or-union identifier
union T_UN                      // (the same as above)
struct { int a ; int b ; }      // struct-or-union { struct-declaration-list }
union  { int a ; int b ; }      // (the same as above)
struct T_ST { int a ; int b ; } // struct-or-union identifier
                                //     { struct-declaration-list }
union T_UN { int a ; int b ; }  // (the same as above)
```

A.2.8 struct-or-union

struct-or-union は struct-or-union-specifier パーサの一部であり，struct, union キーワードのどちらかを受理するパーサである．

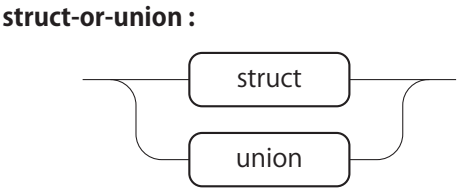


図 A.19: struct-or-union

A.2.9 struct-declaration-list

struct-declaration-list は struct-or-union-specifier パーサの一部であり，構造体および共用体のメンバ変数の宣言列を受理するパーサである．

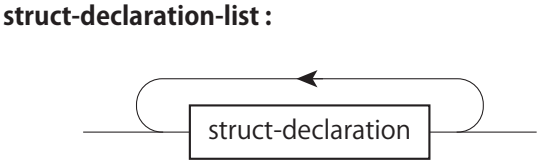


図 A.20: struct-declaration-list

例 A.2.7 (struct-declaration-list)

struct-declaration-list パーサが受理する式の例を以下に示す．

```
int a;        // struct-declaration
int a; int b; // struct-declaration struct-declaration
```

A.2.10 struct-declaration

struct-declaration は struct-or-union-specifier パーサ，struct-declaration-list パーサの一部であり，メンバ変数の宣言を受理するパーサである．

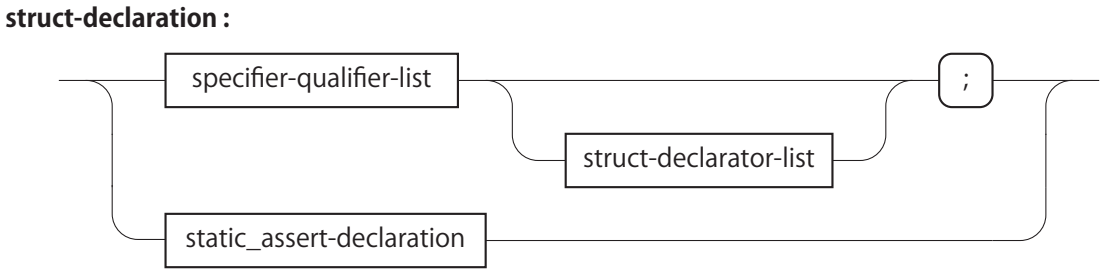


図 A.21: struct-declaration

例 A.2.8 (struct-declaration)

struct-declaration パーサが受理する式の例を以下に示す．

```
int ;         // specifier-qualifier-list ;
int a ;       // specifier-qualifier-list struct-declarator-list ;
int a, b, c ; // specifier-qualifier-list struct-declarator-list ;
int a : 8 ;   // specifier-qualifier-list struct-declarator-list ;
_Static_assert ( a == b, "assert" ) ; // static_assert-declaration
```

A.2.11 specifier-qualifier-list

specifier-qualifier-list は struct-declaration パーサ，typeName パーサの一部であり，宣言の型を受理するパーサである．このパーサは declaration-specifier パーサとほぼ同等であるが，構造体のメンバ，キャスト演算子および sizeof, alignof 演算子の型を受理するためのパーサであり，storage-class-specifier(e.g. extern) や function-specifier(e.g. inline) などが含まれていない点が異なる．

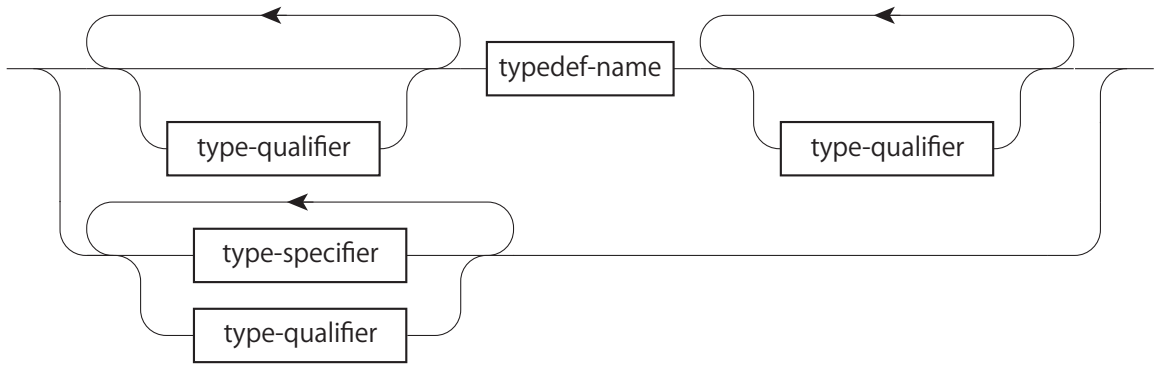


図 A.22: specifier-qualifier-list

例 A.2.9 (specifier-qualifier-list)

specifier-qualifier-list パーサが受理する式の例を以下に示す．

```
T            // typedef-name
const T      // type-qualifier typedef-name
T const      // typedef-name type-qualifier
int          // type-specifier
volatile int // type-qualifier type-specifier
short int    // type-specifier type-specifier
long long    // type-specifier type-specifier
```

A.2.12 struct-declarator-list

struct-declarator-list は struct-declaration パーサの一部であり，カンマで区切ったメンバの宣言を受理するパーサである．

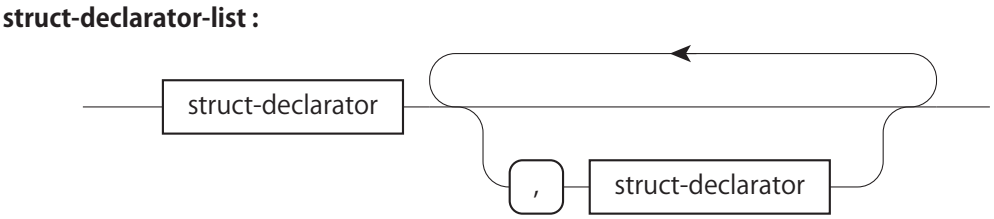


図 A.23: struct-declarator-list

例 A.2.10 (struct-declarator-list)

struct-declarator-list パーサが受理する式の例を以下に示す.

```
int x          // struct-declarator
int x, int y   // struct-declarator , struct-declarator
int * p, * q   // struct-declarator , struct-declarator
int (*f)(void) // struct-declarator
```

A.2.13 struct-declarator

struct-declarator は struct-declarator-list パーサの一部であり，構造体および共用体のメンバ変数（ビットフィールドも含む）を受理するパーサである.

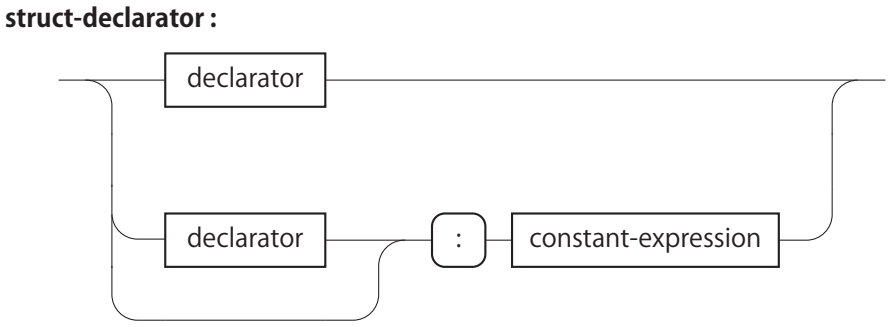


図 A.24: struct-declarator

例 A.2.11 (struct-declarator)

struct-declarator パーサが受理する式の例を以下に示す.

```
a         // declarator
a[2] , *p // declarator
a : 4     // declarator : constant-expression
a , b : 4 // declarator : constant-expression
```

A.2.14 enum-specifier

enum-specifier は type-specifier パーサの一部であり，列挙型の宣言を受理するパーサである．

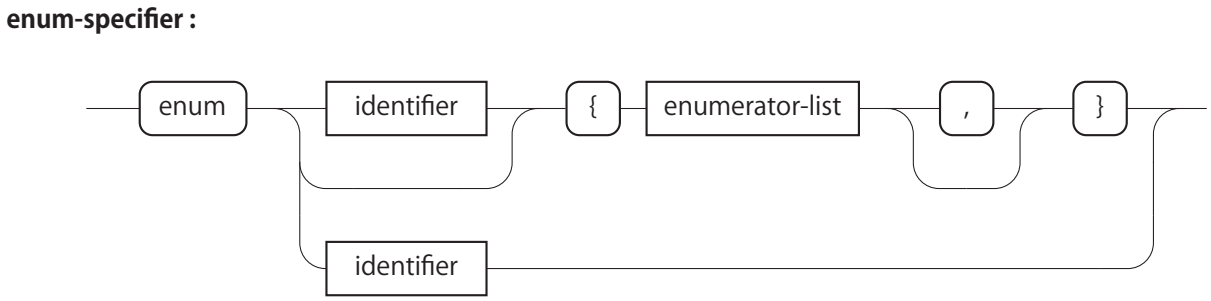


図 A.25: enum-specifier

例 A.2.12 (enum-specifier)

enum-specifier パーサが受理する式の例を以下に示す．

```
enum T_EN             // enum identifer
enum { A, B, C }      // enum { enumerator-list }
enum { A, B, C , }    // enum { enumerator-list , }
enum T_EN { A, B, C } // enum identifer { enumerator-list }
```

A.2.15 enumerator-list

enumerator-list は enumSpecifier パーサの一部であり，カンマで区切られた列挙子の列を受理するパーサである．

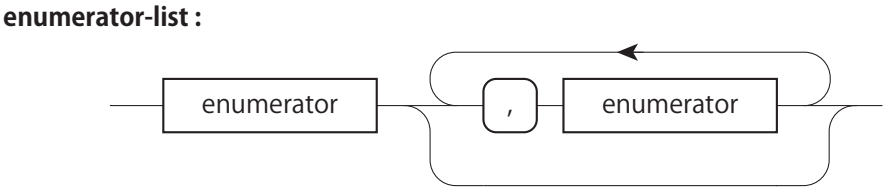


図 A.26: enumerator-list

例 A.2.13 (enumerator-list)

enumerator-list パーサが受理する式の例を以下に示す．

```
A            // enumerator
A = 0, B = 1 // enumerator , enumerator
A, B, C      // enumerator , enumerator , enumerator
```

A.2.16 enumerator

enumerator は enumerator-list パーサの一部であり列挙子とその値を受理するパーサである.

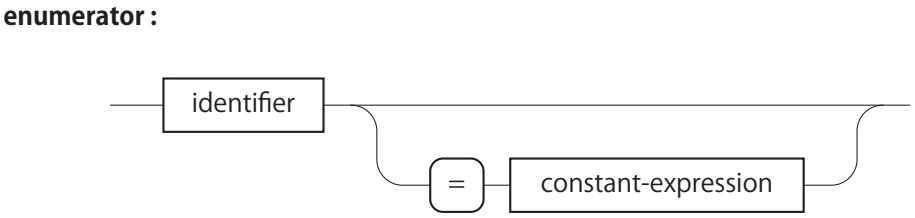


図 A.27: enumerator

例 A.2.14 (enumerator)

enumerator パーサが受理する式の例を以下に示す.

```
A          // identifier
A = 0      // identifier = constant-expression
A = 1 << 2 // identifier = constant-expression
```

A.2.17 type-qualifier

type-qualifier は type-qualifier-list パーサ, specifier-qualifier-list パーサ, declaration-specifiers パーサの一部であり, const, restrict, volatile キーワードを受理するパーサである.

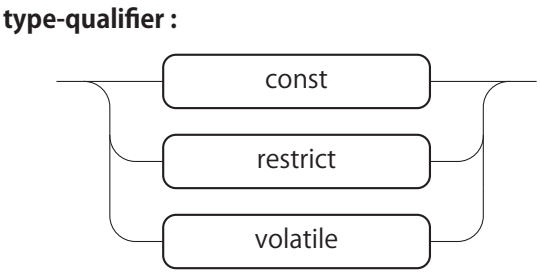


図 A.28: type-qualifier

A.2.18 function-specifier

function-specifier は declaration-specifiers パーサの一部であり，inline キーワードを受理するパーサである．

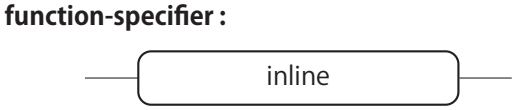


図 A.29: function-specifier

A.2.19 alignment-specifier

alignment-specifier は declaration-specifiers パーサの一部であり，境界調整指定子を受理するパーサである．

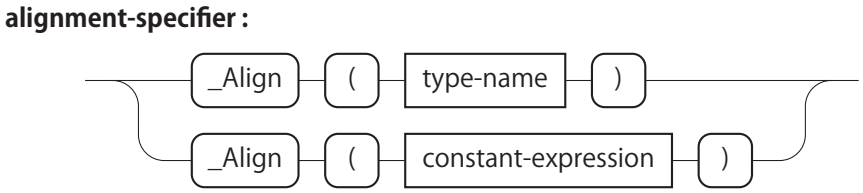


図 A.30: alignment-specifier

例 A.2.15 (alignment-specifier)

alignment-specifier パーサが受理する式の例を以下に示す．

```
_Align(int)       // _Align ( type-name )
_Align(struct ST) // _Align ( type-name )
_Align(4)         // _Align ( constant-expression )
```

A.2.20 declarator

declarator パーサは init-declarator パーサ，direct-declarator パーサ，parameter-declaration パーサ，function-definition パーサの一部であり，ポインタを含んだ宣言を受理するパーサである．

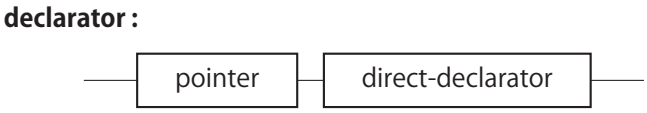


図 A.31: declarator

例 A.2.16 (declarator)

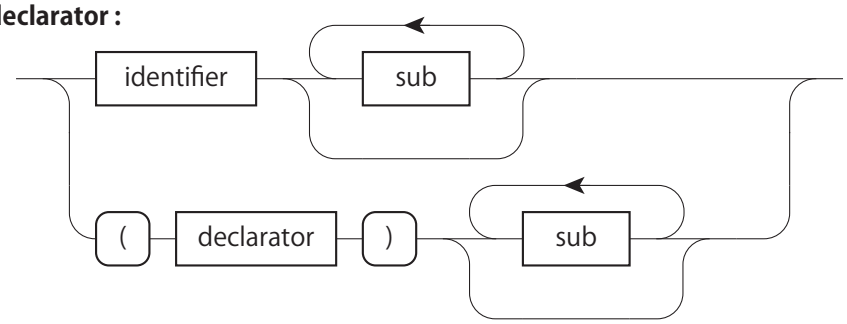
declarator パーサが受理する式の例を以下に示す.

```
a                 // pointer direct-declarator
* a               // pointer direct-declarator
** a              // pointer direct-declarator
* a [ 1 ] [ 2 ]   // pointer direct-declarator
* f(int a, int b) // pointer direct-declarator
```

A.2.21 direct-declarator

direct-declarator は declarator パーサの一部であり，具体的な宣言を受理するパーサである.

direct-declarator :



sub :

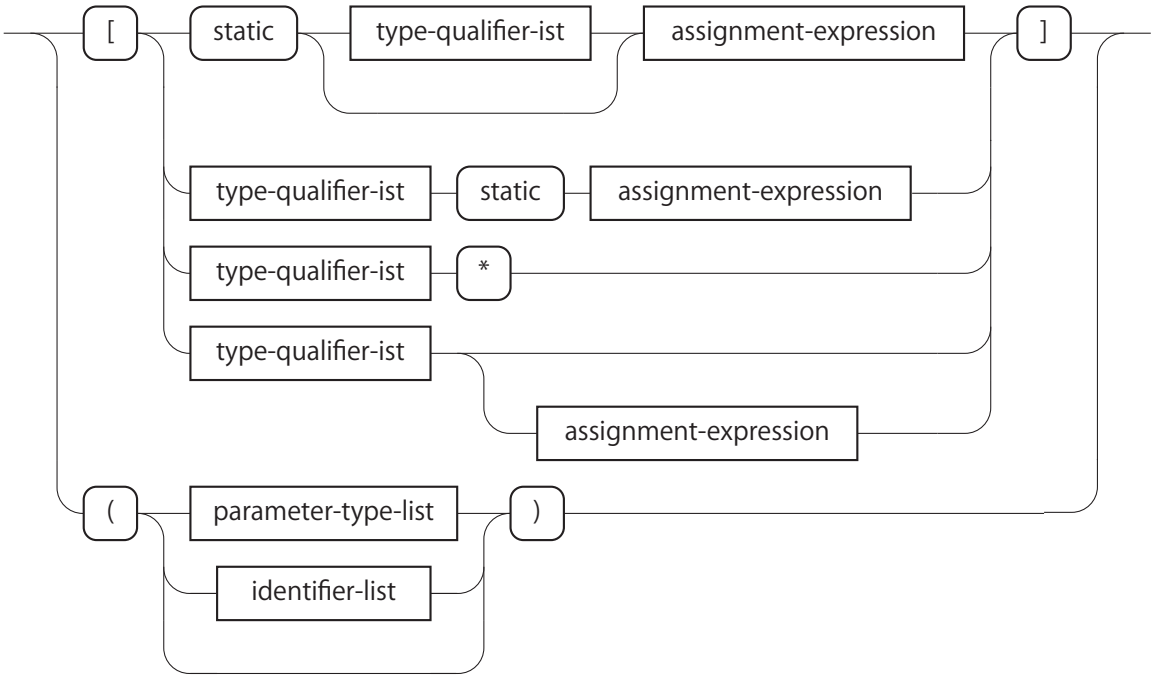


図 A.32: direct-declarator

例 A.2.17 (direct-declarator)

*direct-declarator*パーサが受理する式の例を以下に示す.

```
a               // identifier
a [ 1 ] [ 2 ]   // identifier [ assignment-expression ]
( * a )         // ( declarator )
f(int a, int b) // identifier ( parameter-type-list )
f(a, b, c)      // identifier ( identifier-list )
f()             // identifier ( )
```

A.2.22 pointer

pointer は declarator パーサ，abstract-declarator パーサの一部であり，宣言におけるポインタの部分を受理するパーサである．

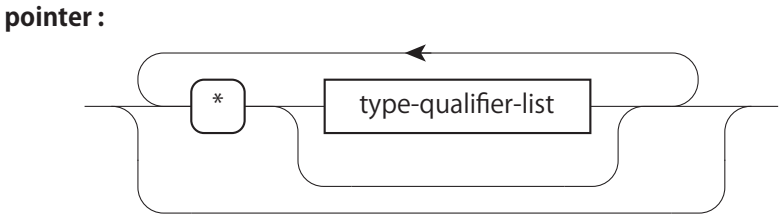


図 A.33: pointer

例 A.2.18 (pointer)

pointer パーサが受理する式の例を以下に示す．

```
     // (none)
*    // pointer
**   // pointer pointer
*** // pointer pointer
* const * const // pointer type-qualifier-list
                // pointer type-qualifier-list
* restrict      // pointer type-qualifier-list
* volatile      // pointer type-qualifier-list
```

A.2.23 type-qualifier-list

type-qualifier-list は direct-declarator パーサ，pointer パーサ，direct-abstract-declarator パーサの一部であり，型修飾子 (const, restrict, volatile) の列を受理するパーサである．

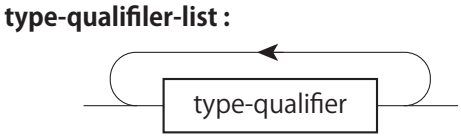


図 A.34: type-qualifier-list

A.2.24 parameter-type-list

parameter-type-list は direct-abstract-declarator パーサの一部であり，可変長引数を含んだ仮引数の宣言の列を受理する．

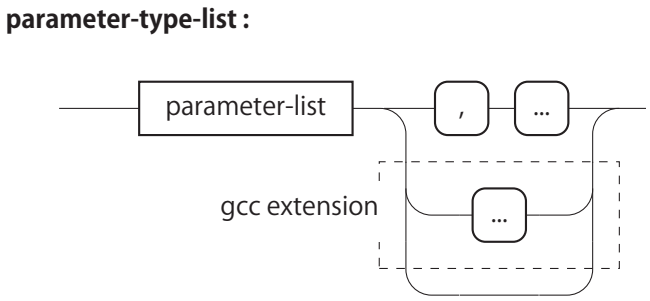


図 A.35: parameter-type-list

例 A.2.19 (parameter-type-list)

parameter-type-list パーサが受理する式の例を以下に示す.

```
int a, int b  // parameter-list
int, int, int // parameter-list
int a, ...    // parameter-list , ...
int a ...     // parameter-list ...
```

A.2.25 parameter-declaration

parameter-declaration は direct-declarator パーサ，direct-abstract-declarator パーサの一部であり，仮引数及び実引数の列を受理するパーサである.

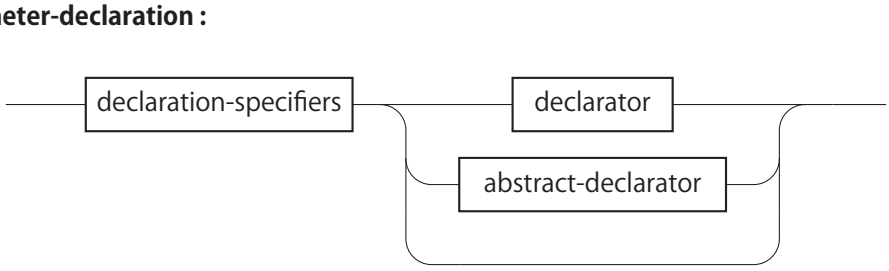


図 A.36: parameter-declaration

例 A.2.20 (parameter-declaration)

parameter-declaration パーサが受理する式の例を以下に示す.

```
int     // declaration-specifiers
int * p // declaration-specifiers declarator

// declaration-specifiers direct-abstract-declarator
    int (int, int *)
```

```
int (int a, int * p)
int [10]
void * [][]
```

A.2.26 identifier-list

identifier-list は direct-declarator パーサの一部であり，実引数の列を受理するパーサである．

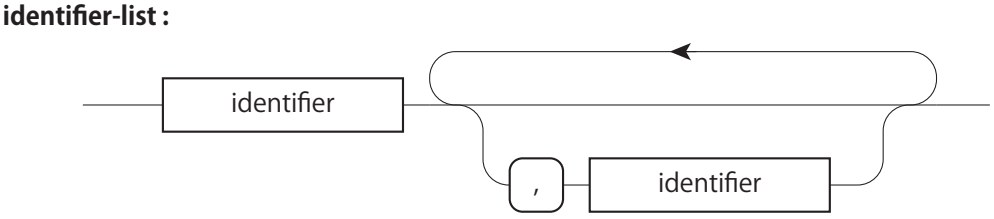


図 A.37: identifier-list

例 A.2.21 (identifier-list)

identifier-list パーサが受理する式の例を以下に示す．

```
a       // identifier
a, b, c // identifier , identifier , identifier
```

A.2.27 type-name

type-name パーサは postfix-expression パーサ，cast-expression パーサ，type-specifier パーサ，alignment-specifier パーサの一部であり，宣言における識別子 (identifier) を省略したときの型情報のみを受理するパーサである．

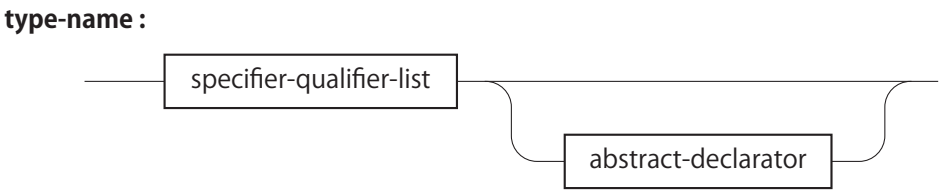


図 A.38: type-name

例 A.2.22 (type-name)

type-name パーサが受理する式の例を以下に示す．

```
const T         // specifier-qualifier-list
const long long // specifier-qualifier-list
int * [2]       // specifier-qualifier-list abstract-declarator
```

A.2.28 abstract-declarator

abstract-declarator パーサは parameter-declaration パーサ, type-name パーサ, direct-abstract-declarator パーサの一部であり宣言における識別子 (identifier) を省略した場合のポインタや配列の要素数を受理するパーサである.

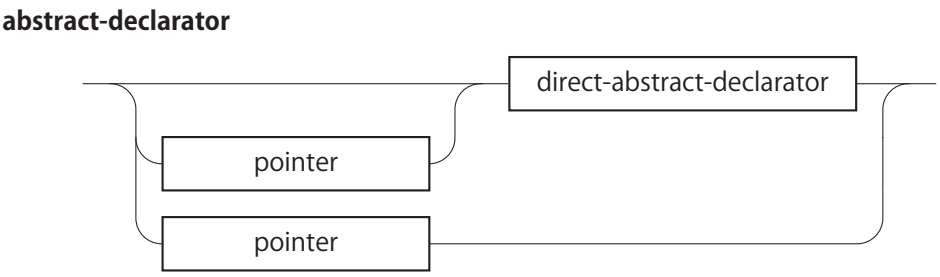


図 A.39: abstract-declarator

例 A.2.23 (abstract-declarator)

abstract-declarator パーサが受理する式の例を以下に示す.

```
[]     // direct-abstract-declarator
* [10] // pointer direct-abstract-declarator
*      // pointer
```

A.2.29 direct-abstract-declarator

direct-abstract-declarator パーサは abstract-declarator パーサの一部であり，宣言における識別子 (identifier) を省略した場合の配列の要素数を受理するパーサである.

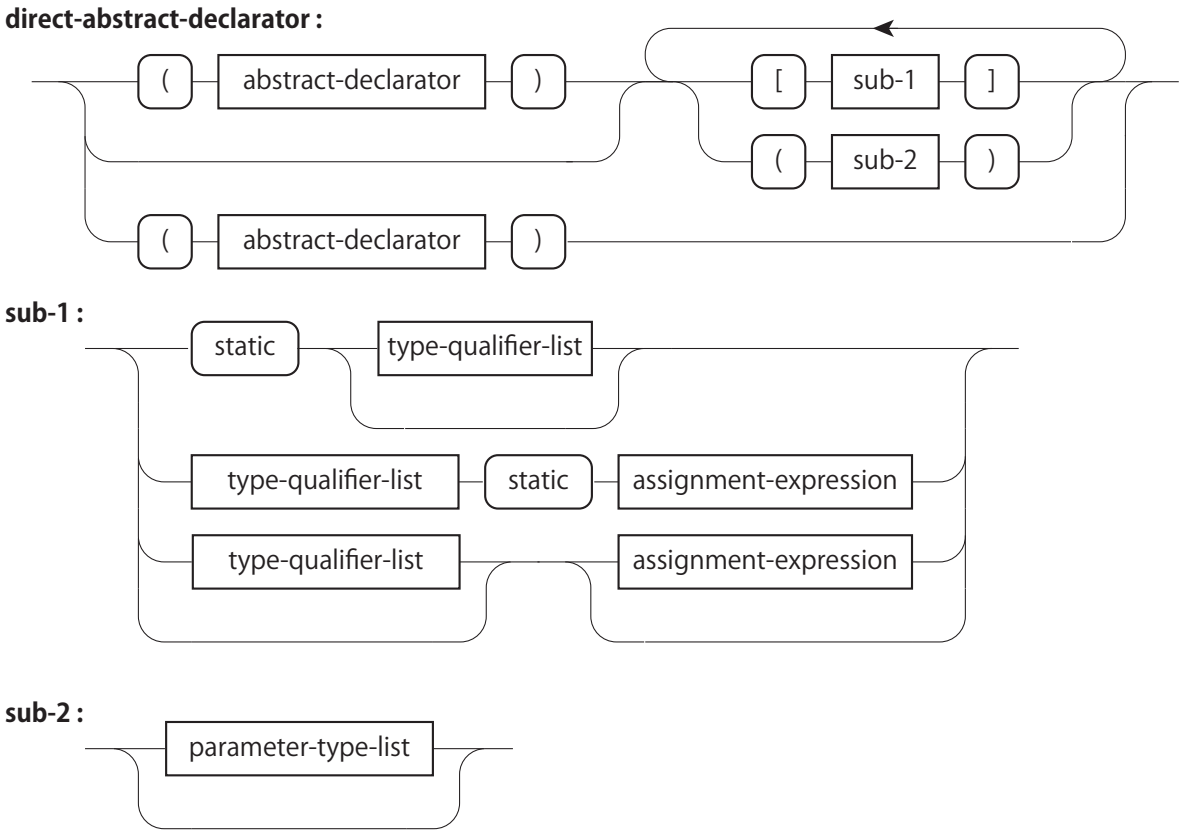


図 A.40: direct-abstract-declarator

例 A.2.24 (direct-abstract-declarator)

direct-abstract-declarator パーサが受理する式の例を以下に示す.

```
// ( abstract-declarator )
(* const [])
// ( abstract-declarator ) [ static type-qualifier-list ]
(*) [ static const ]
// ( abstract-declarator ) [ const static assignment-expression ]
(*) [ const static x ]
// [ ]
[ ]
// ( parameter-type-list )
(int a, int b)
// ( abstract-declarator ) ( parameter-type-list )
(*) ( int a, int b )
```

A.2.30 typedef-name

typedef-name は declaration-specifiers パーサ，specifier-qualifier-list パーサの一部であり，typedef により定義された型を受理するパーサである．

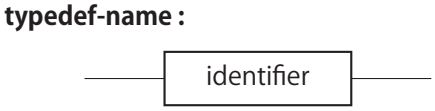


図 A.41: typedef-name

A.2.31 initializer

initializer は init-declarator パーサの一部であり，変数の初期化子を受理するパーサである．

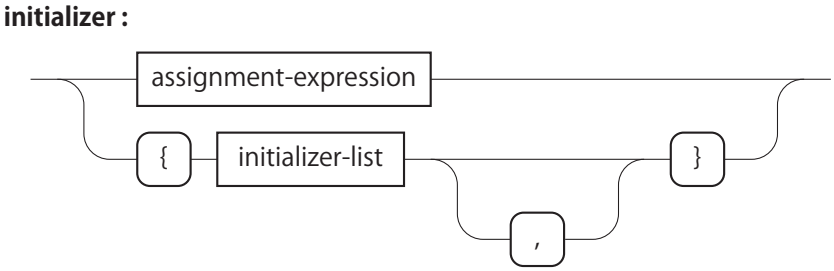


図 A.42: initializer

例 A.2.25 (initializer)

initializer パーサが受理する式の例を以下に示す．

```
10                                  // assignment-expression
10 + 20                             // assignment-expression
{ 10, 20, 30 }                      // { initializer-list }
{ 10, 20, 30, }                     // { initializer-list , }
{ [0] = 10, [1] = 20 }              // { initializer-list }
{ .member_a = 10, .member_b = 20 } // { initializer-list }
```

A.2.32 initializer-list

initializer-list は postfix-expression パーサと initializer パーサの一部であり，構造体のメンバや配列の要素を初期値するための初期化子のリストを受理するパーサである．

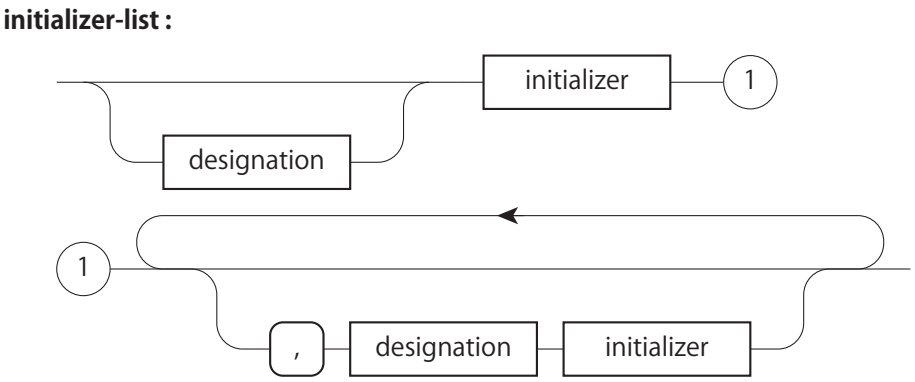


図 A.43: initializer-list

例 A.2.26 (initializer-list)

initializer-list パーサが受理する式の例を以下に示す.

```
10                  // initializer
10, 20, 30          // initializer , initializer , initializer
[0] = 10            // designation initializer
[0] = 10, [1] = 20 // designation initializer , designation initializer
.member_a = 10      // designation initializer
```

A.2.33 designation

designation は initializer-list パーサの一部であり，指示初期化子の指示部と代入演算子の組を受理するパーサである.

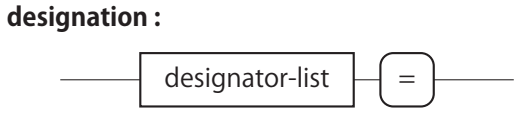


図 A.44: designation

例 A.2.27 (designation)

designation パーサが受理する式の例を以下に示す.

```
[0] =                 // designator-list =
[0][1] =              // designator-list =
.member_a =           // designator-list =
.member_a.member_b = // designator-list =
```

A.2.34 designator-list

designator-list は designation パーサの一部であり，構造体や配列の初期化のブロック内で，要素やメンバを指定する指示期化子の指示部を受理するパーサである．

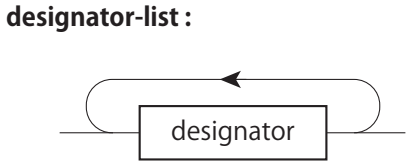


図 A.45: designator-list

例 A.2.28 (designator-list)

designator-list パーサが受理する式の例を以下に示す．

```
[10] . member_a . member_b
. member_a [0]
```

A.2.35 designator

designator パーサは designator-list パーサの一部であり，構造体や配列の指示初期化子の指示部の一部を受理するパーサである．

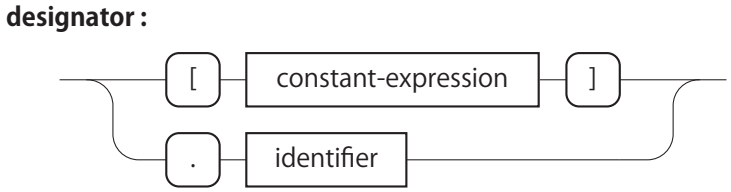


図 A.46: designator

例 A.2.29 (designator)

designator パーサが受理する式の例を以下に示す．

```
[10]
. member_a
```

A.2.36 static-assert-declaration

static-assert-declaration は declaration パーサ，structDeclaration パーサの一部であり，静的アサーションを受理するパーサである．

図 A.47: static-assert-declaration

例 A.2.30 (static-assert-declaration)

static-assert-declaration パーサが受理する式の例を以下に示す.

```
// _Static_assert ( constant-expression , string-literal )
_Static_assert ( A == B , "ERROR" )
```

A.3 文 (statements)

A.3.1 statement

statement パーサは labeled-statement パーサ，block-item パーサ，selection-statement パーサ，iteration-statement パーサの一部であり，文を受理するパーサである.

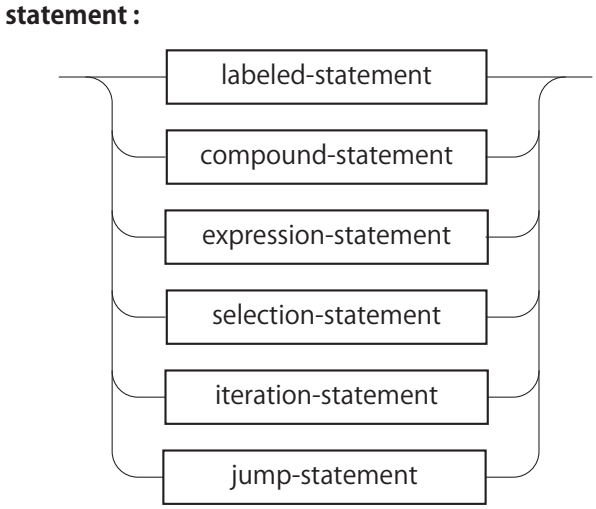


図 A.48: statement

A.3.2 labeled-statement

labeled-statement は statement パーサの一部であり goto ラベルや switch 文の case 文, default 文を受理するパーサである.

labeled-statement :

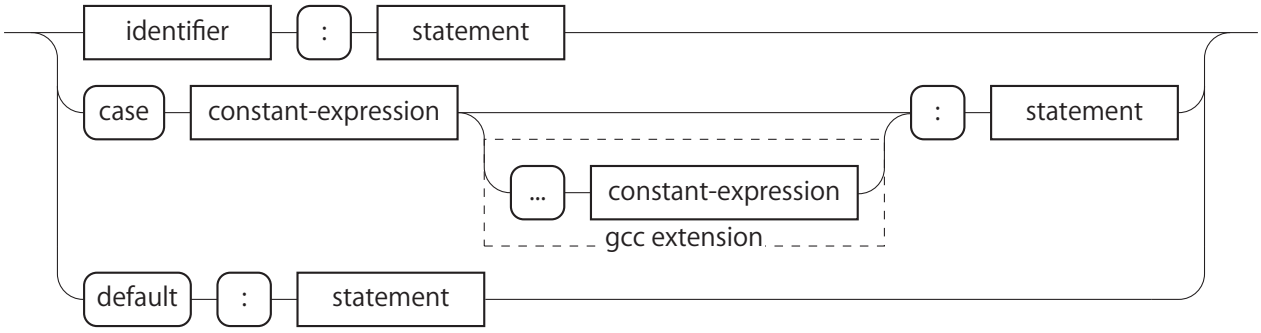


図 A.49: labeled-statement

例 A.3.1 (labeled-statement)

labeled-statement パーサが受理する式の例を以下に示す.

```
end: return;           // identifier : statement
case 1: break;         // case constant-expression : statement
default: break;        // default: statement
case 0 ... 10: break;
  // case constant-expression ... constant-expression : statement
```

A.3.3 compound-statement

compound-statement は primary-expression パーサ, statement パーサ, function-definition パーサの一部であり中括弧で囲まれた複数の文を受理するパーサである.

compound-statement :

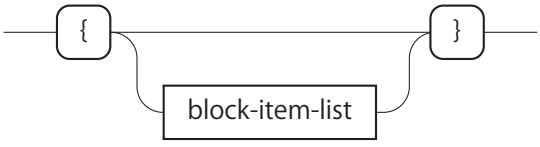


図 A.50: compound-statement

例 A.3.2 (compound-statement)

compound-statement パーサが受理する式の例を以下に示す.

```
{ }                                                 // { }
{ int a = f(); int b = g(); return a + b; } // { block-item-list }
```

A.3.4 block-item-list

block-item-list は compound-statement パーサの一部であり，１つ以上の block-item を受理するパーサである．

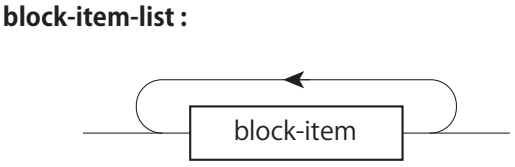


図 A.51: block-item-list

例 A.3.3 (block-item-list)

block-item-list パーサが受理する式の例を以下に示す．

```
int x;                // block-item
int a; int b; a + b; // block-item block-item block-item
```

A.3.5 block-item

block-item は block-item-list の一部であり，宣言と文を受理するパーサである．

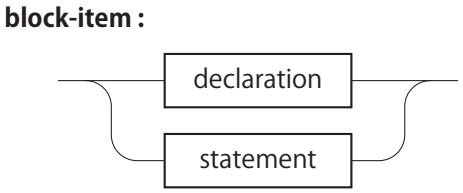


図 A.52: block-item

例 A.3.4 (block-item)

block-item パーサが受理する式の例を以下に示す．

```
int x;     // declaration
x = a + b; // statement
```

A.3.6 expression-statement

expression-statement は，statement パーサの一部であり，セミコロンで終端する式を受理するパーサである．

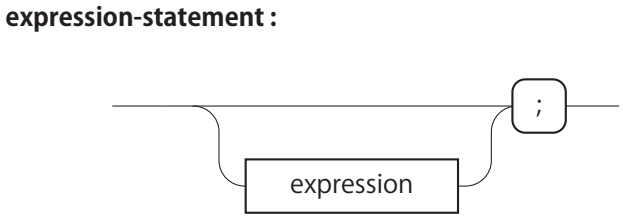


図 A.53: expression-statement

例 A.3.5 (expression-statement)

expression-statement パーサが受理する式の例を以下に示す．

```
10;                         // expresson ;
a = 10, b = 20, c = a + b; // expresson ;
```

A.3.7 selection-statement

selection-statement は statement パーサの一部であり，if 文と switch 文を受理するパーサである．

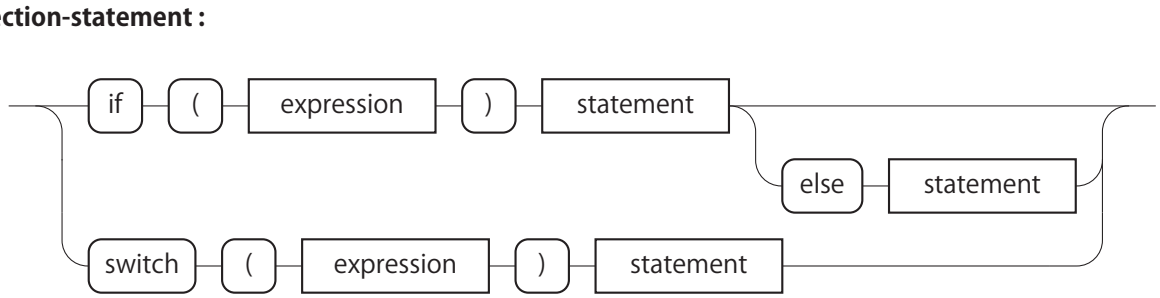


図 A.54: selection-statement

例 A.3.6 (selection-statement)

selection-statement パーサが受理する式の例を以下に示す．

```
if (a == b) c = a + b;
    // if ( expression ) statement
if (a > b) return a; else return b;
    // if ( expression ) statement else statement
switch { case 1: break; default: break; }
    // switch ( expression ) statement
```

A.3.8 iteration-statement

iteration-statement は statement パーサの一部であり，do-while 文，while 文，for 文を受理するパーサである．

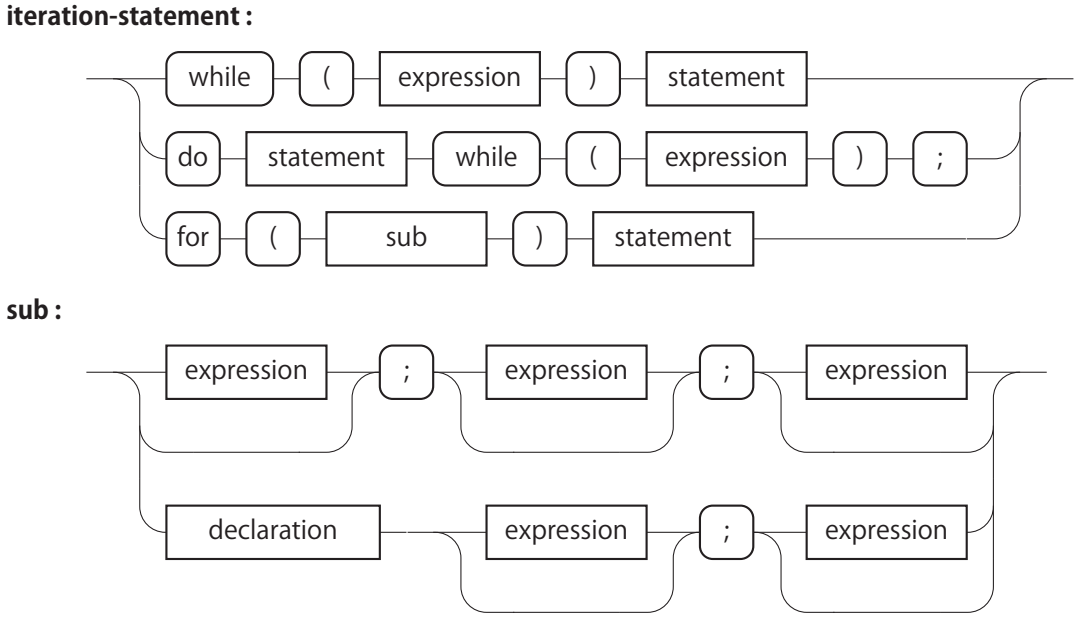


図 A.55: iteration-statement

例 A.3.7 (iteration-statement)

iteration-statement パーサが受理する式の例を以下に示す．

```
while (a < 10) a += 2;
    // while ( expression ) statement
do a += 2 while (a < 10);
    // do statement while ( expression ) ;
for (i = 0; i < 10; i++) x += a[i];
    // for ( expression ; expression ; expression ) statement
for (int i = 0; i < 10; i++) x += a[i];
    // for ( declaration ; expression ; expression ) statement
for (;;) f();
    // for ( ; ; ) statement
```

A.3.9 jump-statement

jump-statement は statement パーサの一部であり，ジャンプ文（goto, continue, break, return）を受理するパーサである．

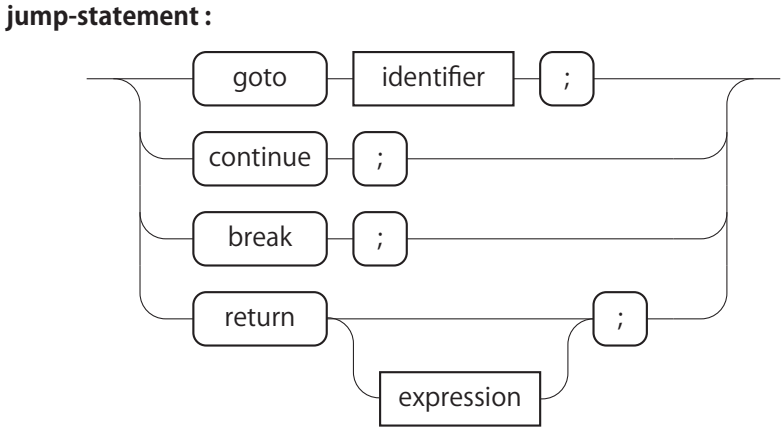


図 A.56: jump-statement

例 A.3.8 (jump-statement)

jump-statement パーサが受理する式の例を以下に示す．

```
goto LABEL; // goto identifier ;
continue;   // continue ;
break;      // break ;
return;     // return ;
return x;   // return expression ;
```

A.4 外部定義 (external definitions)

A.4.1 translation-unit

translation-unit パーサはトップレベルのパーサであり，translation-unit がすべてのトークン列を受理したとき，構文木が完成する．

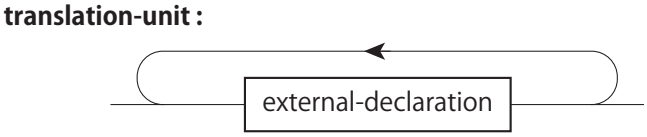


図 A.57: translation-unit

A.4.2 external-declaration

external-declaration パーサは translation-unit パーサの一部であり，変数および関数などの宣言と関数の定義を受理するパーサである．

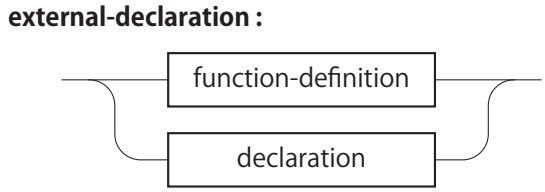


図 A.58: external-declaration

例 A.4.1 (external-declaration)

external-declaration パーサが受理する式の例を以下に示す．

```
extern int x;                // declaration
static int x;                // declaration
void func(void);             // declaration
int func(void) { reutrn 0; } // function-definition
```

A.4.3 function-definition

function-definition パーサは external-declaration パーサの一部であり，関数の定義を受理するパーサである．

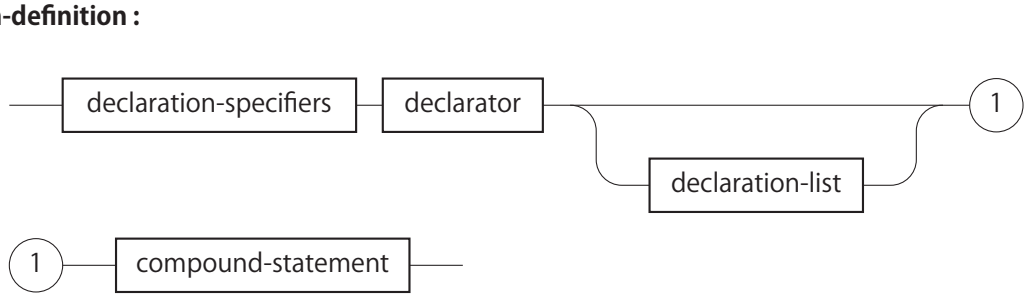


図 A.59: function-definition

例 A.4.2 (function-definition)

function-definition パーサが受理する式の例を以下に示す．

```
// declaration-specifier declarator compound-statement
void func(void) {
```

```
}
// declaration-specifier declarator compound-statement
int func(void) {
    reutrn 0;
}
// declaration-specifier declarator
//     declaration-list compound-statement
int func(a, b)
    // K&R style declaration-list
    int a;
    int b;
{
    reutrn 0;
}
```

A.4.4 declaration-list

declaration-list パーサは function-definition パーサの一部であり，K&R 形式の引数の宣言を受理するパーサである．

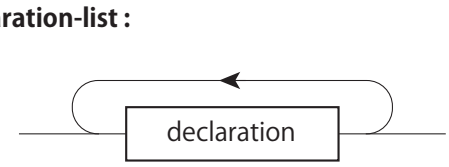


図 A.60: declaration-list

例 A.4.3 (declaration-list)

declaration-list パーサが受理する式の例を以下に示す．

```
int a; int b; // declaration declaration
```