コンパイラのための意味解析器生成系

Semantic Analyzer Generator for Compilers

亀山 裕亮 † 中井 央 ††
Hiroaki KAMEYAMA Hisashi NAKAI
山下 義行 ††† 田中 二郎 †
Yoshiyuki YAMASHITA Jiro TANAKA
† 筑波大学
University of Tsukuba
†† 図書館情報大学
University of Library and Information Science
††† 佐賀大学
Saga University
kame@iplab.is.tsukuba.ac.jp

コンパイラ生成系を用いてコンパイラを作成する場合，宣言と使用の対応付けや型の検査といった記号表を用いた意味解析の処理はほとんどのプログラミング言語で同様である．しかし，記号表から識別子を検索する方法や扱う識別子の種類，あらかじめ用意されている型の種類などは対象とするプログラミング言語ごとに異なっている．これらの要素を指定することで，意味解析処理を行なうためのプログラム断片の部分を自動的に作成できる．本論文ではこれらの要素の記述から，意味解析器を自動的に生成することを目的とし，その一例として SableCC を用いてコンパイラを作成する際に意味解析器を自動的に生成する生成系について述べる．Pascal−S のコンパイラの意味解析器を作成する際に本生成系のために記述した意味解析記述の記述量を他の生成系と比較した結果，1/6 程度の記述量に抑えられていることが確認できた．

1 はじめに

コンパイラ生成系とは形式化された表現から自動的にコンパイラを生成するプログラムのことである．コンパイラ生成系としてよく知られているものには字句解析器生成系 Lex[6] や構文解析器生成系 Yacc[5], PCCTS[7]，CUP[4] ，SableCC[2] などがある．

これらのコンパイラ生成系を用いてコンパイラを作成する場合，意味解析の部分はプログラミング言語のコード断片を使って記述する．例えば Yacc を用いてコンパイラを作成する場合，各生成規則に対して C 言語のコード断片を埋め込む．SableCC を用いた場合，目的とするプログラミング言語の構文記述から SableCC が構文記述に対応する構文木のクラスと構文木の走査用のクラスを生成する．この構文木を走査することで意味解析を行なうため，ユーザは構文木上の各ノードでのアクションを Java [3] のプログラムコードとして作成する．

一般的なプログラミング言語の意味解析では記号表を用いて次のようなことを行なう．

1. 識別子の宣言があれば，記号表へその識別子の情報の登録を試みる．その際多くのプログラミング言語では二重宣言は許されないので登録時に二重宣言の検査も行なう．
2. 多くのプログラミング言語では未宣言の変数の使用はエラーとなるので，その識別子の使用があれば記号表内を検索し，識別子がすでに宣言されているかどうか検査を行なう．また，ある文脈でその識別子の使用が妥当かどうかの検査も行なう．例えばその識別子が関数や演算子の引数として正しいかどうかといった検査である．

これらはどのプログラミング言語のコンパイラにおいても一般的に行なわれる処理であるが，現在は構文解析器生成系を用いてコンパイラを作成する時でも手書きにより作成されている．

記号表処理に関する研究として，Pei らが提案する記号表ライブラリ [9] がある．[9] では，記号表処理のための基本的なライブラリを提供し，コンパイラにおける記号表処理の部分の作成の手助けを行なっている．しかし，コンパイラ作成者がそれらのライブラリを使用してプログラムを作成するということ

に変わりはない．

本論文では他のコンパイラ生成系と同じように意味解析器を自動的に生成することを目的とする．そのためにプログラミング言語ごとに異なる処理である識別子の宣言と使用の対応付けや型の検査といった記号表を用いた意味解析処理に注目する．

以降では意味解析器生成系について提案し，今回はその一例として SableCC を用いてコンパイラを作成する際の意味解析器を自動的に生成する生成系について述べる．

2 意味解析器生成系

コンパイラの各フェーズで使用される記号表とは，コンパイラが読み込んだプログラムの構成要素に関する情報を格納しておくデータ構造のことである．コンパイラの各フェーズでは記号表へ情報を追加したり登録されている情報を用いてそれぞれの処理を行なう．

プログラミング言語の意味解析は主に記号表への情報の登録と検索により行なわれるが，記号表へ登録する情報や，スコープルールに基づいて記号表内を検索する仕組みはプログラミング言語ごとに異なっている．そこで意味解析を行なうプログラムを自動的に生成するためにはこれらの異なる情報をあきらかにする必要がある．本論文で提案する意味解析器生成系ではプログラミング言語ごとに異なっている部分の中でも特に以下の二点に注目している．

1. 型とスコープに関する情報
2. 生成規則と対応した意味解析処理

(1) の情報として扱うものは，プログラミング言語の持つ基本的な型 (の名前) や配列や構造体型などの構造型の種類とスコープルール，識別子の種類などである．

(2) は解析木のどのノードでどのようなアクションを行なうかである．ここでは，登録，検索，検査のためのプリミティブを用意した．

本意味解析器生成系を用いてコンパイラを作成する際の手順を図 1 に示す．本生成系ではプログラミング言語ごとに異なる情報について記述された意味解析器記述及び構文記述から自動的に意味解析器のソースプログラムと記号表に格納する情報や扱う型に関するいくつかのクラスを生成する．本生成系が生成した意味解析器のプログラムコード SableCC が

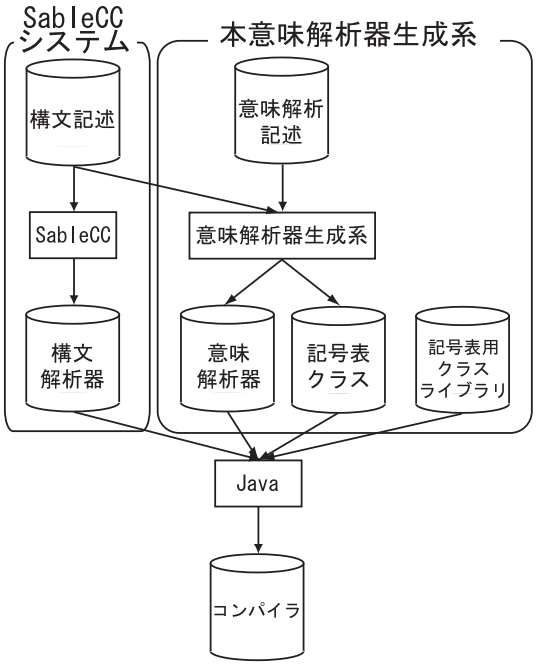

図 1: 本生成系を用いたコンパイラ作成の概要

生成した構文解析器のプログラムコードを Java を用いてコンパイルすることで対象とするプログラミング言語のコンパイラを得ることができる．

今回の意味解析器生成系では，手続き型言語及び手続き型言語の枠組みで扱うことができる Java 言語を対象のプログラミング言語としている．また，スコープルールは多くの手続き型言語で採用されている入れ子型ブロック構造を対象としている．

2.1 型とスコープの情報

これまでにも述べたように，型とスコープの情報はプログラミング言語ごとに異なるが記号表処理は主に記号表への情報の登録と検索により行なわれる．

そこで記号表は，記号表に格納する情報を表わすクラスである要素クラスと，要素クラスを記号表へ格納したり記号表から検索する処理を管理する管理クラスにより実現する．これらのクラスはあらかじめ記号表クラスライブラリとして用意する．

型や識別子などのプログラミング言語ごとに異なる情報については，要素クラスを継承しそれぞれ必要となる情報を持ったクラスを記述から生成する．

それぞれのクラス間の関係を図 2 に示す．

要素クラス 記号表に格納する基本的な情報を持つクラスとして要素クラスを作成し，あらかじめ記号

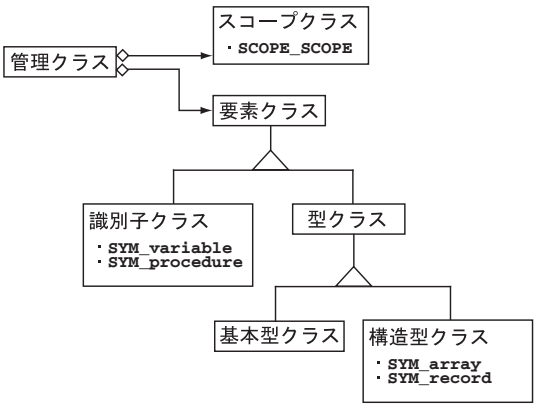

図 2: クラス間の関係図

表ライブラリとして用意する．記号表に格納する情報を表すクラスはすべてこのクラスを継承している．

型クラス　要素クラスを継承し，型の情報を持つ型クラスをライブラリとして用意する．

基本型クラス　プログラミング言語ごとの基本型の情報を持った基本型クラスを型クラスを継承して作成し，ライブラリとして用意する．意味解析記述で宣言された基本型についてそれぞれ基本型クラスのオブジェクトを作成し，意味解析を始める際にあらかじめ記号表へ登録するようにコードを生成する．

構造型クラス　構造型とは既存の型を型構成子を用いて組み合わせることで得られる型のことである．構造型では配列であれば配列の範囲，構造体型であればメンバの一覧などそれぞれ格納する情報が異なっている．これらの異なった情報は記述中に宣言し，宣言からそれぞれの情報を持ったクラスを基本型クラスを継承した構造型クラスとして生成する．生成する構造型クラスには接頭辞として SYM_を付けることとする．

識別子クラス　識別子の種類には変数や関数の他にも定数やラベルなどがあり，それぞれ記号表に格納する情報が異なっている．これらの種類及び格納する情報は記述中に宣言することにする．宣言された情報を用いてそれぞれの識別子の情報を格納する識別子クラスを生成する．生成する識別子クラスには接頭辞として SYM_を付けることとする．

管理クラス　管理クラスはスコープクラスを用いて要素クラスを記号表へ登録したり，記号表から要素クラスを検索する処理の管理を行なう．

スコープクラス　スコープクラスはスコープの規則に従って記号表へ情報を登録したり記号表からの情報を検索する処理を行なう．現段階では入れ子型ブロック構造のスコープである SCOPE_block クラスを記号表ライブラリとして用意した．

2.2　意味解析処理

構文木のどのノードにおいてどのような処理を行なうかについて，ノードに対応する非終端記号名と処理の内容とを組にして記述することで意味解析の処理を生成する．処理の記述には以下に示すような二重宣言や未宣言の識別子の検査，型の同定検査といった機能についてあらかじめ用意したプリミティブを使用することができる．

- 識別子の宣言時には，宣言が現われたスコープの規則に従って記号表に識別子の情報を登録する．記述中では %make() というプリミティブを用いて識別子の種類，名前，情報を指定することで記号表へ登録を行なう．この際，すでに記号表内に同じ名前と同じ情報を持つ要素が存在する場合には，二重宣言となるため適切なエラーメッセージを出力する．

- 識別子の使用時には，その識別子が現われたスコープの規則に従って記号表内から識別子の情報を検索する．もし記号表内に目的の情報が見つからない場合は未宣言でありエラーメッセージを出力する．記述中では %search() というプリミティブを用いて記号表から識別子の検索を行なう．

- 型検査などのために，記号表から検索された情報の比較を行なう．記述中では比較演算子や条件演算子を用いて情報の比較を行なう．

2.3　合成属性

SableCC では構文木の各ノードで行なわれる意味解析において，それぞれの解析結果の値を参照するための機構が用意されていない．ノード間で解析結果を参照するためには，例えば大域的なハッシュ表を用意しそれぞれのノードの値をキーとすることでノードごとの値を保持しておく機構をユーザが用意しなければならない．属性評価器生成系 Rie[8] では，

各生成規則における意味解析の結果を継承属性や合成属性を用いることで相互に参照することができる．

本生成系でも親子関係にあるノード間で解析結果を大域的なハッシュ表に保存し，他のノードで簡単に参照できるような機構を用意することにした．

現在のノードの解析結果を保存するには`$result`という変数に解析結果を代入する．あるノードにおける意味解析時にそのノードの子となるノードの解析結果を使用したい場合には子ノードの文法記号名 (例えば `ident` など) の頭に`$`記号を付けた変数 (`$ident`) を使用することにより値を参照することができる．

一般的な属性文法ではすべての生成規則において属性の評価を記述しなければならないが，本生成系では解析結果について何も指定されなかった親子のノード間では，子のノードの解析結果を親のノードへ伝播するコードを自動的に生成する．

3 生成系の仕組みと実現

まず `Pascal-S`[1] のための意味解析記述を図 3 に示す．この節ではこの記述を例に用いて今回作成した生成系の仕組みについて簡単に説明する．次に SableCC を用いて生成系を実現し実際に `Pascal-S` の意味解析記述から自動的に作成した意味解析器と，他の生成系における意味解析器の部分との記述量を比較した結果について述べる．

```
 1 %adapter DepthFirstAdapter
 2 %grammar pascal-S.grammar
 3
 4 %prim   integer,real,bool,char
 5
 6 %struct record,array
 7
 8 %ident procedure,function,variable,
 9        constant,param
10
11 %scope block      :SCOPE_block
12
13 %%
14
15 proc_decl_head:
16  %make(procedure,
17        name:$ident,
18        param:$formal_param_list);
19
20 param_list:
21  %make(param, name:$ident);
22
23 var_decl:
24  %make(variable,
25        name:$ident_list,
26        type:$type);
27
28 proc_call:
29  {actual_param_list}
30      %search(name:$ident,
31              mode:procedure);
32      %check($ident.param,
33             $actual_param_list);
34 |{ident}
35      %search(name:$ident,
36              mode:procedure);
37
38 %%
```

図 3: Pascal-S のための意味解析記述の一部

3.1 生成系の仕組み

意味解析記述は型やスコープなどの情報を記述する部分と，解析木のどのノードでどのようなアクションを行なうかを記述する部分とで構成されている．それぞれの部分は`%%`により分けられている．

図 3 の 1 行目から 13 行目にある`%%`までは型やスコープなどの情報を記述する部分である．

図 3 の 1 行目は抽象構文木の走査順の宣言である．SableCC では構文木のノードのクラスを作成すると同時に構文木の走査用のクラスとして深さ優先で走査を行なうクラス (DepthFirstAdapter) と構文木を逆に走査するクラス (ReverseDepthFirstAdapter) の 2 つのクラスを生成する．ここでは作成する意味解析器においてどちらの走査法を使用するかを宣言する．

図 3 の 2 行目は対象のプログラミング言語の構文記述ファイルの指定である．指定された構文記述ファイルを解析し，目的のプログラミング言語におけるトークンや非終端記号，生成規則を収集する．この情報は以降の処理で利用する．

図 3 の 4 行目は基本型を宣言している．宣言された基本型を記号表に登録するコードを生成する．図 3 の 6 行目は構造型の宣言である．宣言された名前から，前節で述べた構造型の情報を保持する構造型クラスを生成する．図 3 の 8，9 行目は識別子の種類の宣言である．宣言された名前から，前節で述べた記号表の要素クラスを生成する．

図 3 の 11 行目はスコープの宣言である．対象のプログラミング言語で使用するスコープが，どの非終端記号名に対応しているかを，終端記号名と使用するスコープクラスを組にして記述する．この記述から宣言された非終端記号に対応する解析木のノードの解析を行なっている時は指定されたスコープクラスを用いて記号表へ情報の登録を行なったり，記号表から情報を検索するコードを生成する．

図 3 の 13 行目にある`%%`から 38 行目にある`%%`まで

の部分は解析木のどのノードでどのようなアクションを行なうかを記述する部分である．

図 3 の 15 行目から 18 行目はコンパイル中に手続きの宣言が見つかった場合にその情報を記号表へ登録する処理を記述している部分である．前述した`%make()`記述を使用することで，二重宣言の検査及び記号表への登録を行なうコードを生成する．

図 3 の 28 行目から 36 行目はコンパイル中に手続きの使用が見つかった場合にその情報が記号表に登録されているか検査を行なう意味処理を記述している部分である．`%search()` 記述を使用することで，記号表内を検索し識別子の宣言が行なわれているか検査を行なうコードを生成する．検索の結果は`$entry`という変数に格納され，他の処理の際に参照することができる．

3.2 生成系の実現と検証

以上のような意味解析記述から，意味解析器を生成する意味解析器生成系を SableCC を用いて作成した．本生成系と他の生成系のそれぞれを用いて実際に `Pascal-S` のコンパイラを作成し意味解析器の記述量を比較した．その結果，本生成系を用いた場合に `Pascal-S` ために記述した意味解析記述は全体で 200 行，生成された意味解析器のプログラムコードの行数は 1400 行であった．また，属性評価器生成系 Rie を用いて `Pascal-S` コンパイラを作成した場合に意味解析器にあたる部分のコード断片は 1200 行，SableCC を用いた場合の意味解析器のプログラムコードは 1300 行であった．

以上のことから，本生成系を用いることでコンパイラの意味解析器の作成にかかる労力を大幅に削減できることが確認できた．

4 まとめと今後の課題

本論文では生成系を用いてコンパイラを作成する場合に，識別子の検索方法や扱う識別子の種類，あらかじめ用意されている型の種類などといった対象とするプログラミング言語ごとに異なっている部分を指定することでそれらの意味解析器を自動的に生成することを目的とした．その一例として SableCC を用いてコンパイラを作成する際に意味解析部を自動的に生成する生成系を作成した．

本生成系のために記述した意味解析記述は全体で 200 行であり他の生成系と比較して 1/6 程度の記述量であった．これらの結果から，本生成系を用いることで意味解析部の作成にかかる労力を大幅に削減できることを確認した．

今後の課題として以下のことが挙げられる．

- 現在の記号表用ライブラリには入れ子型ブロック構造を持つスコープを扱うクラスしか用意していない．スコープルールはプログラミング言語により異っているため，登録や探索方法などスコープごとの違いを明示的に宣言することでスコープクラスを自動的に生成できると考える．
- 型検査の処理について，現在は型検査を行なうためのプリミティブを用意しているが，それぞれの型の間の強弱関係などを明示的に宣言することで，型検査機構を自動的に生成できると考える．
- 現在の意味解析記述は形式的ではなく冗長な部分が多い．同じ内容の処理を行なっている部分については，さらに簡潔な記述を使用できるようにする．

参考文献

[1] R. E. Berry. *Programming Language Translation*. Ellis Horwood limited, 1981.
(邦訳) 武市正人:『プログラム言語の処理系 Pascal S の翻訳系・通訳系』, 近代科学社 (1983).

[2] E. Gagnon. SableCC, An Object Oriented Compiler Framework. Master's thesis, McGill University, 1996. `http://www.sablecc.org`.

[3] J. Gosling, B. Joy, and G. Steel. *The Java Language Specification Second Edition*. Addison Wesley, 2000.

[4] S. E. Hudson. *CUP Parser Generator for Java*, 0.10j edition, Mar. 1998.
`http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[5] S. C. Johnson. Yacc - yet another compiler compiler. Computing science technical report 32, AT&T Bell Laboratories, 1975.

[6] M. E. Lesk. Lex - A Lexical Analyzer Generator. Computing science technical report 39, AT&T Bell Laboratories, 1975.

[7] T. Parr. *The Purdue Compiler Construction Tool Set*, version 2.22 released edition, Mar. 2000.
`http://www.polhode.com/pccts.html`.

[8] M. Sassa, H. Ishizuka, and I. Nakata. Rie, a Compiler Generator Based on a One-pass-type Attribute Grammar. *Software – Practice and Experience*, 25(3):229–250, 1995.

[9] P.-C. Wu, J.-H. Lin, and F.-J. Wang. Designing a Reusable Symbol Table Library. Technical report csie-93-1010, National Chiao Tung University, Nov. 1993.