特別研究報告

題目

プログラム・スライサの試作を通じた「ソースコード 解析システム REFINE」の評価

指導教官
鳥居 宏次 教授

報告者
高田 智規

平成 7 年 2 月 24 日

大阪大学 基礎工学部 情報工学科

平成 12 年度 特別研究報告

プログラム・スライサの試作を通じた「ソースコード解析システム REFINE」の評価

高田 智規

内容梗概

プログラムスライスとは，プログラム中のある地点の一つの変数に着目し，その変数に影響する，もしくは影響される文のみを抜き出したもので，プログラム内の文の依存関係を解析して得られるグラフ (PDG) を用いて計算することができる．

本研究では，プログラム解析を用途として市販されているシステム REFINE が，プログラムスライスを求めるツール (スライサ) を作成する環境としてどれだけ有効であるかを評価するために，Pascal のサブセットをターゲット言語とするスライサの作成を実際に試みた．

試作に際しては，REFINE の持つプログラミング言語に備えられているデータ型や，サブシステムとして組み込まれた構文解析機能や視覚的ユーザインターフェース機能が，スライサを実現する上で，具体的にどのように利用可能なのかを明らかにすることを目指した．

試作したツールは REFINE 言語を用いて記述されており，サブセット Pascal のソースプログラムを読み込み，画面上に PDG をグラフ表示したうえで，マウスによる操作でスライスを求める基点と変数の指定を行なうことができる．その結果計算されたスライスは，PDG のサブグラフとして表示され，また，ファイルに保存することもできる．

本試作を通じて，PDG のノードや，依存関係を表すリンクの記述に，REFINE 言語の持つ object 型や tuple 型，map 型などのデータ型が利用可能であることを確認した．また，サブシステムの機能を有効に活用すれば，C 言語で記述する場合に比べて，より少ない労力と記述量でスライサを実現できることも確認できた．

主な用語

プログラム依存グラフ (Program Dependence Graph，PDG)
プログラムスライス (Program Slice)
視覚化 (Visualization)

目 次

1 まえがき **3**

2 プログラム依存グラフとスライス **5**

2.1 プログラム依存グラフ 5

2.2 制御依存とデータ依存 7

2.3 スライス 7

2.4 アルゴリズム 8

2.4.1 諸定義 9

2.4.2 プログラム全体の解析 9

2.4.3 関数および手続きの宣言部の処理 10

2.4.4 文の解析 11

2.4.5 スライスの計算法 13

3 REFINE システム **14**

3.1 REFINE システムの構成 14

3.2 REFINE 言語 14

3.3 DIALECT サブシステム 14

3.4 INTERVISTA サブシステム 15

4 試作したツールの概要 **18**

4.1 機能 18

4.2 入力言語 18

4.3 実行例 18

5 考察 **23**

6 あとがき **24**

謝辞 **25**

参考文献 **26**

1 まえがき

プログラムスライシング技術とは，プログラム中のある文 s におけるある変数 v の値に影響を与える全ての文，もしくは，s における変数 w の定義が影響を与える全ての文をプログラムから抽出する技術であり，その結果取り出された文の集合をプログラムスライスあるいは単にスライスと呼ぶ．この技術は Mark Weiser[2] によって考案され，最初はプログラムのデバッグを支援するものと考えられていが，現在ではデバッグだけでなく，テストや保守，プログラム合成などにも利用されている [4, 8]．

スライスには，全ての可能な入力に対するスライスである静的スライスと，特定の入力に対するスライスである動的スライスがある．本システムでは静的スライスのみ扱い，動的スライスは扱わない．従って，以下スライスといえば静的スライスのことを指すものとする．

プログラム依存グラフとは, プログラム内の文の間の依存関係を表す有向グラフであり, その節点は, プログラムに含まれる条件判定部分, 代入文, 入出力文, 手続き呼び出し文を表し, その有向辺は 2 つの節点の間の依存関係を表す．プログラム依存グラフの有向辺をたどり，依存関係のある文を抽出することによりスライスを求めることができる．

Reasoning Systems 社で開発された REFINE は，ソースコードの解析・変更を主目的としたシステムである．REFINE は，ソースコードの操作に有用な多くのデータ型を備えたプログラミング言語のインタプリタと，構文解析機能を備えたサブシステム (DIALECT) や，グラフィカル・ユーザ・インタフェースを提供するサブシステム (INTERVISTA) を備えている．これらを利用することにより，C 言語など，従来ソースコードの解析ツールの実現に利用されてきた言語では扱い難かった操作，例えば，構文木の構築・変更や，その図式表示などといった操作を，あらかじめ用意されたデータ型や関数を用いて比較的容易に行なうことができる．

本研究では，REFINE が，プログラムスライスを求めるツール (スライサ) を作成する環境としてどれだけ有効であるかを評価するために，Pascal のサブセットをターゲット言語とするスライサの作成を実際に試みた．

試作に際しては，REFINE の持つプログラミング言語に備えられているデータ型や，サブシステムとして組み込まれた構文解析機能や視覚的ユーザインタフェース機能が，スライサを実現する上で，具体的にどのように利用可能なのかを明らかにすることを目指した．

試作したツールは，REFINE 言語を用いて記述されており，サブセット Pascal のソースプログラムを解析し，画面上に PDG をグラフ表示したうえで，マウスによる操作でスライスを求める基点と変数の指定を行なうことができる．その結果計算されたスライスは，PDG のサブグラフとして表示され，また，ファイルに保存することもできる．

本試作を通じて，PDG のノードや，依存関係を表すリンクの記述に，REFINE 言語の持

つ object 型や tuple 型，map 型などのデータ型が利用可能であることを確認した．また，サブシステムの機能を有効に活用すれば，C 言語で記述する場合に比べて，より少ない労力と記述量でスライサを実現できることも確認できた．

本報告ではこれ以降，2 節でプログラム依存グラフ・スライスの概要およびスライスの計算方法を，3 節で REFINE の概要を，4 節で今回試作したシステムの説明を，5 節で考察を述べる．

2 プログラム依存グラフとスライス

2.1 プログラム依存グラフ

プログラム依存グラフ (Program Dependence Graph, 以下 PDG) とは, プログラム内の文の間の依存関係を表す有向グラフである．PDG の節点は, プログラムに含まれる条件判定部分, 代入文, 入出力文, 手続き呼び出し文を表し，有向辺は 2 つの節点の間の依存関係を表している．2 つの節点の間の依存関係には，条件文や繰り返し文などの条件判定によるプログラムの制御の依存関係を表す制御依存関係と，変数の値の依存関係を表すデータ依存関係がある． 図 1 のプログラムに対する PDG の例を 図 2 に示す．図中の楕円が各節点を表し，その中に各文に対応する文字列が記されている． 2.2 節で述べるように，実線，破線で表された有向辺はそれぞれデータ依存関係，制御依存関係を示している．また，菱形で表された節点は以下で述べる特殊節点を示している．

```
program factorial(input,output);
  var n,f:integer;

  begin
    readln(n);
    f := 1;
    if (n <= 1)
      then
        writeln(f)
      else
        begin
          while (n > 1) do
            begin
              f := f * n;
              n := n - 1
            end;
          writeln(f)
        end
  end.
```

図 1: プログラム factorial

通常の文以外の節点として，手続き・関数内での大域変数の使用・定義による影響，パラメータの受渡しなどの解析を行なうためのいくつかの特殊節点を用いる．特殊節点は 表 1 に示すような役割を持っている．特殊節点が用いられている簡単な例を 図 3 に示す．関数 f

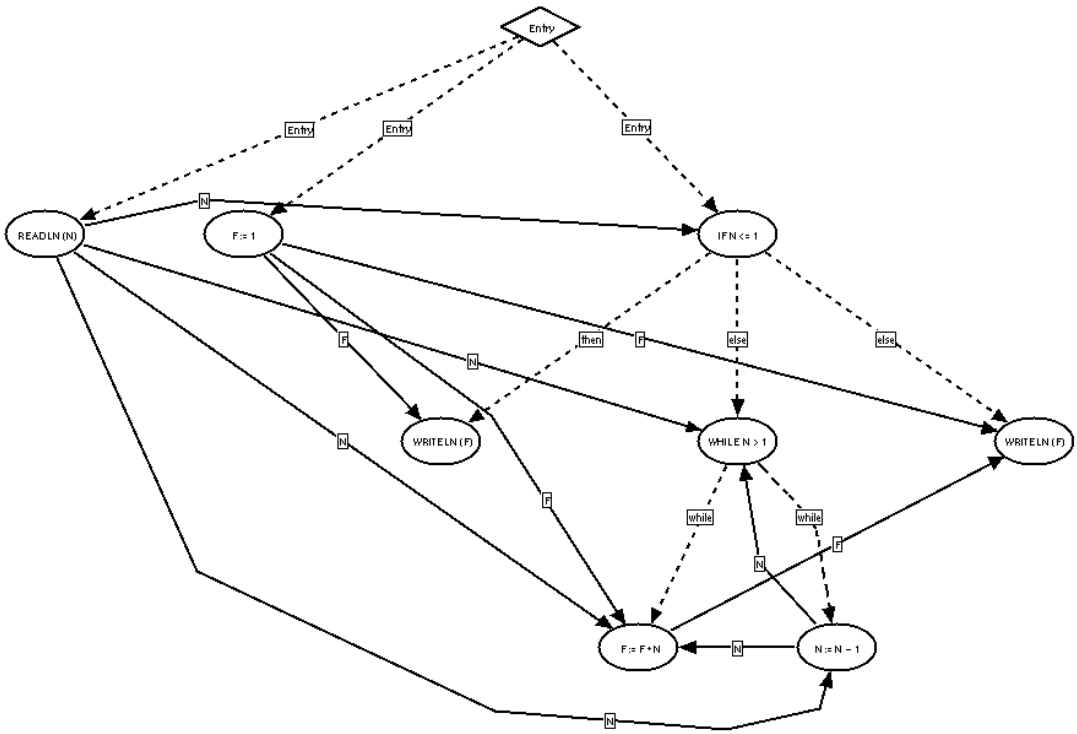

図 2: factorial の PDG

は，値引数 p をとり，関数内で大域変数 g を参照・定義しているものとする．ここで用いられている特殊節点は parameter-in 節点 f_p–in, global-in 節点 f_g–in, global-out 節点 f_g–out, exit 節点 f–$exit$ の 4 種類である．

PDG をもとめるアルゴリズムとしては， 文献 [9, 10] を参考とした．アルゴリズムについては 2.4 節に記した．

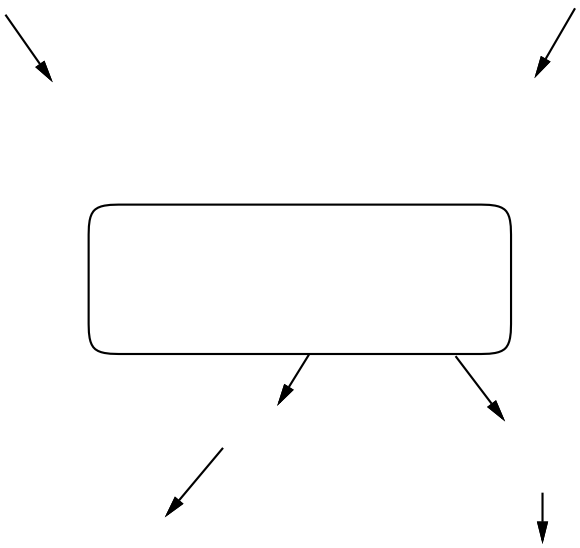

図 3: 関数 f に対する PDG の概略

2.2 制御依存とデータ依存

2 つの文の間の関係として制御依存 (Control Dependence, 以下 CD) 関係とデータ依存 (Data Dependence, 以下 DD) 関係がある. 文 s が if 文のような条件判定を含む文であり, かつ, s の条件判定部分 s_1 の実行結果が文 s_2 の実行の有無に直接影響を与える時, s_1 と s_2 の間に CD 関係が発生する. これを CD 関係辺 ($s_1 \dashrightarrow s_2$) と表す. また, 文 s_1 がある変数 v を定義し, v を参照している文 s_2 にこの定義が到達する[1] 時, s_1 と s_2 の間に DD 関係が発生する. これを DD 関係辺 ($s_1 \xrightarrow{v} s_2$) と表す.

2.3 スライス

プログラムスライシング (Program Slicing) 技術とは，プログラム中のある文 s におけるある変数 v の値に影響を与える全ての文，もしくは，s における変数 w の定義が影響を与え

[1] 「s_1 から s_2 への, v の再定義を含まないような実行パスが存在する」と同義

表 1: 特殊節点

	特殊節点	表記例
entry 節点	プログラムおよび手続き (関数) にひとつずつあり，そのプログラム (関数・手続き) の中の全ての文は entry 節点からの CD 関係がある	f–$Entry$
exit 節点	関数の戻り値を通して伝わる影響を検出するための節点で, 各関数にひとつずつある	f–$exit$
global-in 節点	手続き外からの大域変数の影響を内部へ伝えるための節点で, 手続きに, 個々の大域変数に対して, ひとつずつある	f_g–in
global-out 節点	手続き内で定義された大域変数の影響をその外へ伝えるための節点で, 手続きに, 個々の大域変数に対して, ひとつずつある	f_g–out
paramter-in 節点	手続きの引数を通して伝わる影響を検出するための節点で, その引数それぞれにひとつずつある	f_p–par
parameter-out 節点	手続きの変数引数を通して伝わる影響を検出するための節点で, 変数引数それぞれにひとつずつある	f_p–par_{out}

る全ての文をプログラムから抽出する技術であり，その結果取り出された文の集合をプログラムスライス (Program Slice) あるいは単にスライス (Slice) と呼ぶ，スライスには，ソースコードを解析して得られる，全ての可能な入力に対するスライスである静的スライス (Static Slice) と，ある特定の入力のもとで実行した文を並べた実行系列を解析して得られる動的スライス (Dynamic Slice) がある．本システムでは静的スライスのみ扱い，動的スライスは扱わない．従って，以下スライスといえば静的スライスのことを指すものとする．

本システムでは，プログラムを PDG で表現し，その有向辺をたどることによりスライスを得る．PDG の有向辺を向き通りにたどることにより得られるスライスを Forward Slice，有向辺を逆向きにたどることにより得られるスライスを Backward Slice と呼ぶ．また，それらを計算することをそれぞれ Forward Slicing, Backward Slicing と呼ぶ．

2.4 アルゴリズム

ここで扱うプログラムの言語は，文として，条件文．繰り返し文．代入文．入出力文．手続き呼出文をもち，データ型はスカラー型のみでポインタ型はないものとする．また，手続きは自己再帰および相互再帰呼びだしが可能で，手続きの引数には，値渡しと，変数渡しの

二通りの渡し方が可能とする．

2.4.1 諸定義

変数 v と節点 n との組 d を $\langle v, n \rangle$と表し，変数部を d_{var} で，節点部を d_{vertex} で参照することにする．プログラムのある領域 S (連続する 0 個以上の文の集合) の到達定義集合 (Reaching Definition Set) を $RD_{in}(S)$ と書き，これを以下のように定義する [2] ．

$$RD_{in}(S) \equiv \{\langle v, n \rangle | \text{ 節点 } n \text{ で変数 } v \text{ が定義される} \wedge n \text{ が } S \text{ に到達する } \}$$

また，S を解析した後次の領域へ到達する定義集合を $RD_{out}(S)$ と書く．

プログラム中のある領域 S について，S を実行した時に (どのパスが実行されても) 必ず定義される変数とその変数が定義された節点との組の集合を確実定義集合と呼び，$SuDEF(S)$ と表す．また，定義される可能性のある変数とそれが定義された節点との組の集合を潜在定義集合と呼び，$PoDEF(S)$ と表す．

$$\begin{aligned} SuDEF(S) &\equiv \{\langle v, n \rangle | S \text{ の中の節点 } n \text{ で変数 } v \text{ は必ず定義される } \} \\ PoDEF(S) &\equiv \{\langle v, n \rangle | S \text{ の中の節点 } n \text{ で変数 } v \text{ は定義される可能性がある } \} \end{aligned}$$

また，手続き f の確実定義集合，潜在定義集合，をそれぞれ $SuDEF(f)$，$PoDEF(f)$ と表す．また，それ以外に，以下のような条件を満たす変数 v を「手続き f で暗に参照される大域変数」と呼び，その集合を $ImUSE(f)$ と表す．

- v は大域変数である．
- f 内で v を参照している地点に，f の外部からの v の定義が到達する．

2.4.2 プログラム全体の解析

プログラムは手続きを一つの単位として解析される．手続きの解析を開始する前に，全ての手続きに対して，以下のような初期化を行なう (f を手続きとする)．

$$\begin{aligned} SuDEF(f) &\leftarrow \phi \\ PoDEF(f) &\leftarrow \phi \\ ImUSE(f) &\leftarrow \phi \end{aligned}$$

また，P を手続きの集合として，以下のようにプログラム全体の解析を行なう．

[2] 到達定義集合についての詳細は [1] を参照されたい．

1. $P \leftarrow \{p | p$ はプログラム内の手続き $\}$
2. すべての $q \in P$ について q の解析を行なう .(2.4.3 節参照)
3. すべての $q \in P$ について q のすべての $g\text{–}in$ 節点に入ってくる，またはそこから出ていく DD 関係辺をすべて消去する．
4. $P \neq \phi$ の間，次の操作を繰り返す．
   - $P \leftarrow P - \{q\}$
   - q の解析をする .(2.4.3 節参照)
   - もし $SuDEF(q)$,$PoDEF(q)$,$ImUSE(q)$ のうちのどれかの値が変化したら，$P \leftarrow P \cup \{r | r$ は q を直接呼び出す手続き $\}$
5. 主プログラムを解析する

2.4.3 関数および手続きの宣言部の処理

手続き f は，図 4 のような構造で表される．ここでは，その本体の領域を B と表す．この f は以下のような手順で解析される．

$$
\begin{array}{l}
function \quad f\,(\ldots) : \cdots; \\
\quad var \cdots \\
\left.\begin{array}{l} \text{begin} \\ \quad \vdots \\ \text{end;} \end{array}\right\} B
\end{array}
$$

図 4: 手続き定義の概略

1. RD_{in} の初期値として，

   $RD_{in} \leftarrow \{\, \langle w, f_w\text{–}Pin\rangle \,| w$ は手続き f の仮引数変数 $\} \cup \{\langle v, f_v\text{–}Gin\rangle \,| v \in ImUSE(f)\}$

   とする．
2. 手続き本体 B を解析する．
3. $SuDEF(B)$,$PoDEF(B)$ から大域変数以外の要素を除く．

4. f において，

$$\begin{aligned} SuDEF(f) &== SuDEF(B) \wedge \\ PoDEF(f) &== PoDEF(B) \wedge \\ ImUSE(f) &== ImUSE(B)(2.4.4 \text{ 参照}) \end{aligned}$$

が成り立っていないなら，

$$\begin{aligned} SuDEF(f) &\leftarrow \{\langle v, f_v\text{–}Gout\rangle | \langle v, f_v\text{–}Gout\rangle \in SuDEF(B)\} \\ PoDEF(f) &\leftarrow \{\langle w, f_w\text{–}Gout\rangle | \langle w, f_w\text{–}Gout\rangle \in PoDEF(B)\} \\ ImUSE(f) &\leftarrow \{t | t \in ImUSE(B)\} \end{aligned}$$

とする．

5. $RD_{out}(f)$ の中から，関数の戻り値に関する定義を探し，その定義節点 n から関数の $exit$ 節点への DD 関係辺 ($n \xrightarrow{n} f\text{–}exit$) を作る．また，全ての大域変数 g に関する定義を探しだし，その定義節点 m から対応する $g\text{–}out$ 節点への DD 関係辺 ($m \xrightarrow{g} f_g\text{–}Gout$) を作る．同様に，全ての変数渡し仮パラメータ p に関する定義を探しだし，その定義節点 n から対応する手続きの $para\text{–}out$ 節点への DD 関係辺 ($n \xrightarrow{p} f_p\text{–}Pout$) を作る．

2.4.4 文の解析

条件文，繰り返し文，代入文，入出力文，手続き呼出文の解析は 文献 [9] のものに以下の修正を加えるだけである．

S : 文の集合もしくは式

新たに $ImUSE(S)$ なる変数の集合を定義し，

$ImUSE(S) \leftarrow \{v_{var} |$ v は S で参照される変数のうち外部からの影響を受ける大域変数で，$v \in RD_{in}(S)$ $\}$

手続き呼出文，式の解析については，引数として変数渡しも許したため少し異なる．それらの解析方法を以下に示す．

手続き呼出文の解析 $S : p(exp_1, ..., exp_n);$ とする．

1. $RD_{in}(exp_1) \leftarrow RD_{in}(S)$ として式 exp_1 の処理を行なう．
2. $RD_{in}(exp_k) \leftarrow RD_{out}(exp_{k-1})(1 < k \leq n)$ としてすべての引数についての処理をする．

3. $RD_{in}(S) \leftarrow RD_{out}(exp_n)$
4. $RD_{in}(S)$ 内の全ての大域変数 v の定義 d それぞれに対して d からそれに対応する p の $g - in$ 節点への DD 関係辺 $(d \xrightarrow{v} p_v\text{–}Gin$ を作る．
5. $RD_{out}(S) \leftarrow \{d | d \in RD_{in}(S) \wedge d_{var}$と同じ変数名の要素が $SuDEF(p)$ にない $\}$ $\cup$ $\{ \langle v, p_v\text{–}Gout \rangle | v$ と同じ名前の変数が $SuDEF(p)$ の中にある $\} \cup PoDEF(p)$ $\}$
6. $SuDEF(S) \leftarrow SuDEF(p) \cup \bigcup_{k=1}^{n} SuDEF(exp_k)$
7. $PoDEF(S) \leftarrow PoDEF(p) \cup \bigcup_{k=1}^{n} PoDEF(exp_k)$
8. $ImUSE(S) \leftarrow ImUSE(p) \cup \bigcup_{k=1}^{n} ImUSE(exp_k)$

式の解析 節点 n に含まれている式 exp は以下のようにして解析される．

1. $RD_{out}(exp) \leftarrow RD_{in}(exp)$
2. $$\begin{aligned} SuDEF(exp) &\leftarrow \phi \\ PoDEF(exp) &\leftarrow \phi \\ ImUSE(exp) &\leftarrow \phi \end{aligned}$$
   と初期化する．
3. 式を左から読み込み，
   - ある変数 v の参照があれば，その時点での $RD_{out}(exp)$ の中から v に関する定義 d を全て探して，その定義節点から節点 n への DD 関係辺 $(d \xrightarrow{v} n)$ をつくる．
   - ある関数 g の呼び出しがあれば，その引数を先に式の列として処理した後，g の $exit$ 節点から n への DD 関係辺 $(g\text{–}exit \xrightarrow{g} n)$ を作り，$RD_{out}(S)$ 内の全ての大域変数 v の定義 d それぞれに対して d からそれに対応する g の $g\text{–}in$ 節点への DD 関係辺 $(d \xrightarrow{v} g_v\text{–}Gin)$ を作る．さらに，以下のように各集合の値を更新する．
     $RD_{out}(exp) \leftarrow \{\langle v, n \rangle | v \in SuDEF(g)\} \cup \{PoDEF(f)\} \cup \{d | d \in RD_{out}(exp)$ $\wedge\ d_{var}$ と同じ名前の変数が $SuDEF(f)$ に存在しない $\}$
     $SuDEF(exp) \leftarrow SuDEF(exp) \cup SuDEF(g)$
     $PoDEF(exp) \leftarrow PoDEF(exp) \cup PoDEF(g)$
     $ImUSE(exp) \leftarrow ImUSE(exp) \cup ImUSE(g)$

という処理を式の終りまで続ける．

4. exp が手続きの引数 p の時，現節点から対応する関数 g の $para\text{–}in$ 節点への DD 関係辺 $(n \xrightarrow{p} g_p\text{–}Pin)$ を作る．また，その引数が変数渡しをされている時には，対応する関数 f の $para\text{–}out$ 節点から現節点への DD 関係辺 $(f_p\text{–}Pout \xrightarrow{p} n)$ を作る．

2.4.5 スライスの計算法

スライスを計算する方法は，PDG を作成してから PDG 上での到達可能性を判定して計算するという 文献 [4] の方法と同じであるので，スライスの計算法も基本的には， 文献 [4] と同じとした．

3 REFINE システム

3.1 REFINE システムの構成

REFINE システムは，図 5 のように，2 つのサブシステムを含めて 3 つの部分から構成されている．REFINE システムは，ソースコードの解析・変更を主目的としたシステムであり，サブシステムを利用することにより，ソースコードの構文解析や図表の表示といった，従来ソースコードの解析に用いられている C などの言語では扱いにくい機能を比較的容易に実現できる．

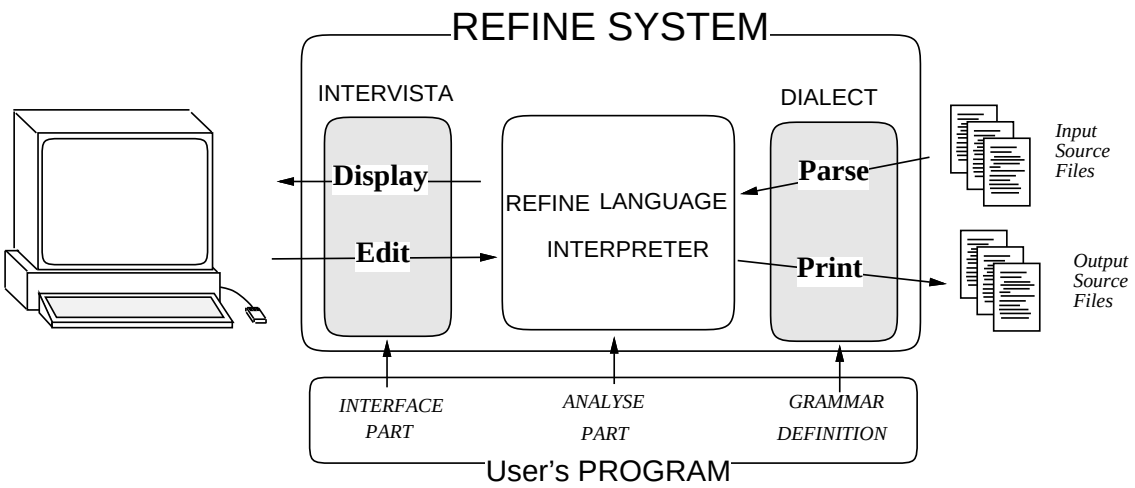

図 5: REFINE の構成図

3.2 REFINE 言語

REFINE 言語とは，REFINE システムにおいて使用する言語である．以下単に REFINE といえば REFINE 言語のことを指すものとする．REFINE は豊富なデータ型や機能を持った言語であり，集合や木構造を扱えるなどソースコードの解析を行なうのに適している．

3.3 DIALECT サブシステム

DIALECT は言語を処理するためのツールである．ある言語に対する文法を定義すれば，その言語に対する構文解析機能などを提供する．DIALECT によって作られた構文解析部は対象言語を構文解析し，Abstract Syntax Tree(以下 AST) と呼ばれる木構造を生成する．

例えば，図 6 で示すような文法定義によって提供された構文解析部で “$1 + 2 * 3$” という式を構文解析した時には， 図 7 のような AST[3] が生成される．生成された AST は REFINE のデータ型の一つである object 型によって表現されており，木の解析・変形などを行なうことによりソースコードの解析・変更などを行なうことができる．

```
!! in-package("RU")
!! in-grammar('user)

var CALC-EXPRESSION:  object-class subtype-of user-object
var IDENTIFIER:       object-class subtype-of calc-expression
var INTEGER-CONSTANT: object-class subtype-of calc-expression
var ADD-EXPRESSION:   object-class subtype-of calc-expression
var MULT-EXPRESSION:  object-class subtype-of calc-expression

var IDENTIFIER-NAME:  map(identifier, symbol) = {||}
var INTEGER-VALUE:    map(integer-constant, integer) = {||}
var ADD-ARG1:         map(add-expression, calc-expression) = {||}
var ADD-ARG2:         map(add-expression, calc-expression) = {||}
var MULT-ARG1:        map(mult-expression, calc-expression) = {||}
var MULT-ARG2:        map(mult-expression, calc-expression) = {||}

form DECLARE-CALC-TREE-ATTRIBUTES
  define-tree-attributes('identifier, {'identifier-name});
  define-tree-attributes('integer-constant, {'integer-value});
  define-tree-attributes('add-expression, {'add-arg1, 'add-arg2});
  define-tree-attributes('mult-expression, {'mult-arg1, 'mult-arg2})

!!in-grammar('syntax)
grammar CALC
  productions
    identifier       ::= [identifier-name]         builds identifier,
    integer-constant ::= [integer-value]           builds integer-constant,
    add-expression   ::= [add-arg1 "+" add-arg2]   builds add-expression,
    mult-expression  ::= [mult-arg1 "*" mult-arg2] builds mult-expression
  precedence
    for calc-expression brackets "(" matching ")"
      (same-level "+" associativity left),
      (same-level "*" associativity left)
end
```

図 6: 文法定義の例

3.4 INTERVISTA サブシステム

INTERVISTA は REFINE のアプリケーションのためのインターフェイス部分の機能を提供している．INTERVISTA を用いて，以下のような 3 種類の機能を実現できる．

[3] AST の詳細については 文献 [6] を参照されたい．

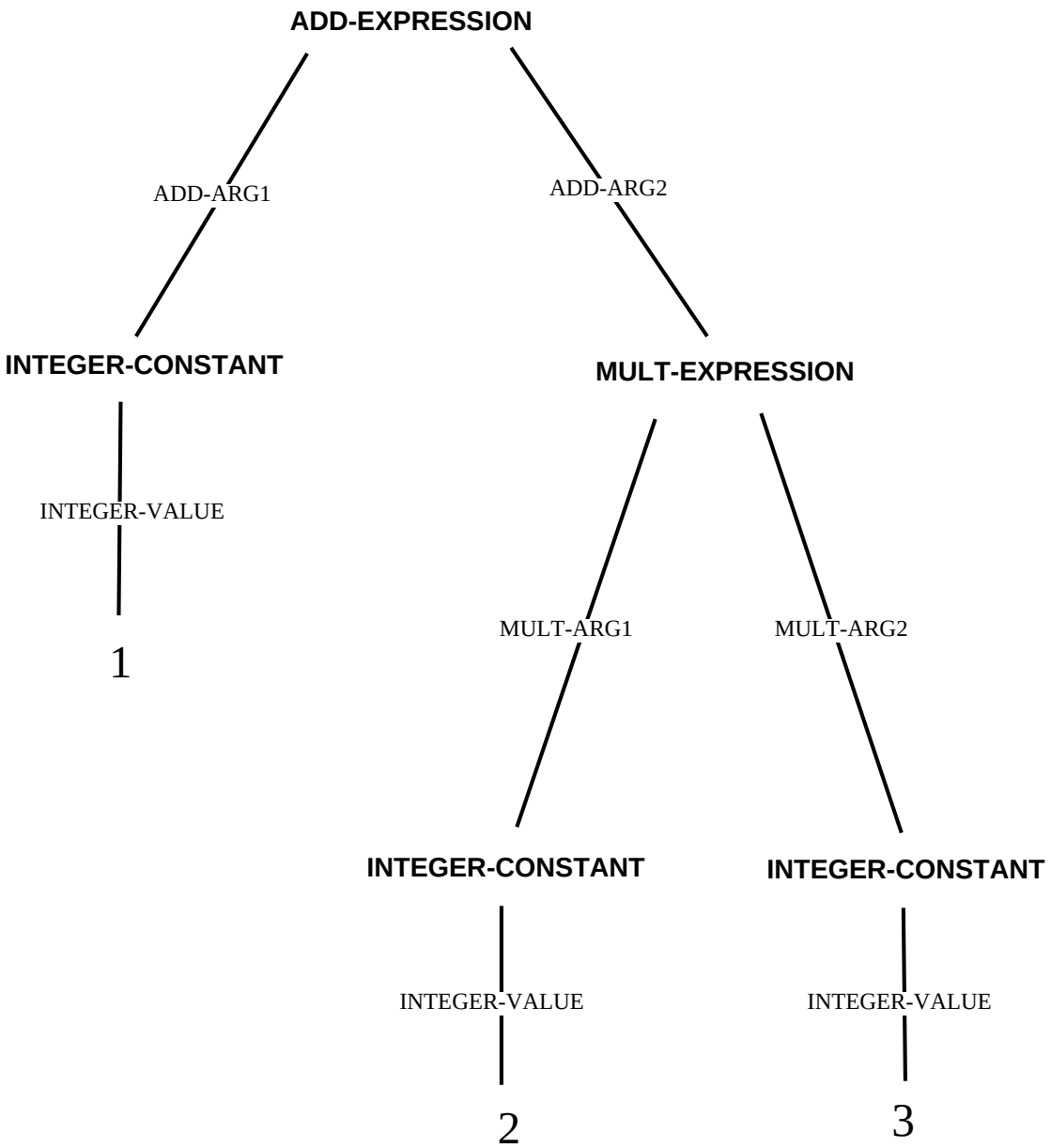
ADD-EXPRESSION
ADD-ARG1
ADD-ARG2
INTEGER-CONSTANT
MULT-EXPRESSION
INTEGER-VALUE
1
MULT-ARG1
MULT-ARG2
INTEGER-CONSTANT
INTEGER-CONSTANT
INTEGER-VALUE
INTEGER-VALUE
2
3

図 7: AST の例 $(1 + 2 * 3)$

1. 図表
   icon と呼ばれる「箱」と,link と呼ばれる「矢印」によって図形を表す．また，表示された図形に対し，マウスを使って操作を行なうことができる．

2. メニュー
   メニューを表示し，入力を受け付ける．

3. マウス操作可能なテキストウィンドウ
   テキストを表示し，指定した文に対し，マウスを使って操作することができる．

4　試作したツールの概要

本ツールは REFINE 言語によるプログラムとして記述され，REFINE システム上で動作する．ツールの大きさは本体 (解析部・表示部・マウス操作部) が約 1600 行，文法定義部が約 1000 行の，合計約 2600 行である．

4.1　機能

Pascal のサブセットで書かれたプログラムを入力として読み込み，PDG を画面上に表示する．また，節点，変数およびスライスの種類 (Forward Slice, Backward Slice) を指定することにより，スライスの計算を行なうことができる．スライスの計算結果は，必要な節点と辺のみによって表された PDG のサブグラフとして表示される．節点，変数，スライスの種類の指定は全てマウス操作によって行なう．また，PDG およびスライスの計算結果は PS 形式のファイルとして保存することが可能である．

4.2　入力言語

本ツールの入力言語として, 以下で述べるような Pascal 風言語を扱った. この言語は現実の言語に比べ簡単化されてはいるが, 本質的な構造をすべて含んでおり, 一般の言語に拡張できる.

今回扱った言語では文として, 条件文 (if), 代入文, 繰り返し文 (while), 入力文 (readln), 出力文 (writeln), 手続き呼びだし文, 複合文 (begin–end) がある．変数の型としてはスカラ型のみでポインタ型はない．プログラムは．大域変数・定数宣言，手続き・関数定義，メインプログラムからなり，ブロック構造はない．手続き (関数) 内では，内部で宣言された局所変数と引数，および大域変数のみが参照可能で，他の手続き (関数) 内の局所変数は参照できない．手続き (関数) の引数は値引数，変数引数の両方を扱い，相互再帰・自己再帰ともに使用可能である．

4.3　実行例

本ツールを実行すると， 図 8 に示すようなウィンドウが現れ，ファイル名の入力を行なう．ファイル名が入力されると PDG が表示される．PDG の節点上でマウスのボタンを押すとメニューが現れ，そこで Slicing を選ぶと 図 9　図 10 で示すようにスライスの種類，変数名をメニューにより選ぶことができる．スライスの種類，変数名を選ぶとスライスの計算を行ない，結果を表示する．

図 12 は， 図 11 を解析して得られた PDG である． 2 節では CD 関係辺にはラベルが付けられていなかったが，本ツールにおいては視覚的な理解を助けるために，CD 関係辺にも

図 8: ファイル名の入力ウィンドウ

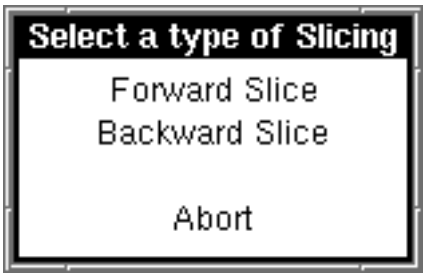

図 9: スライスの種類の選択

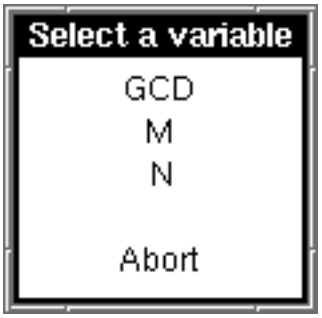

図 10: 変数の選択

その性質を表すラベルをつけた．

また，図 13 は，プログラムの 20 行目の `writeln(kz)` から変数 kz に関する Backward Slice を計算した結果である．

```
 1  program check(input,output);
 2   var date,day:integer;
 3       kz:boolean;
 4
 5  begin
 6   readln(date);
 7   readln(day);
 8   kz := false;
 9   if (date mod 5) = 0
10    then
11     begin
12      kz := not kz;
13      writeln(kz)
14     end
15    else
16     if (day = 6) or (day = 7)
17      then
18       begin
19        kz := not kz;
20        writeln(kz)
21       end
22      else
23        writeln(kz)
24  end.
```

図 11: プログラム goto

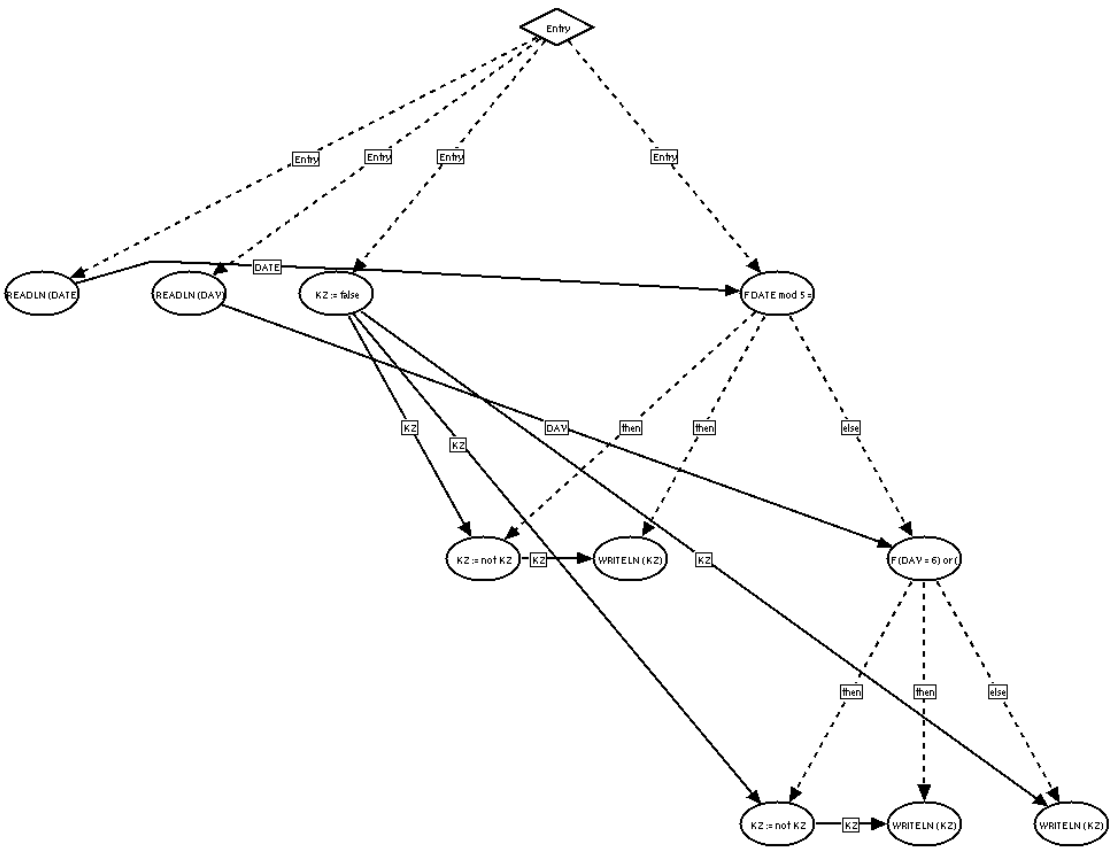

図 12: プログラム goto の PDG

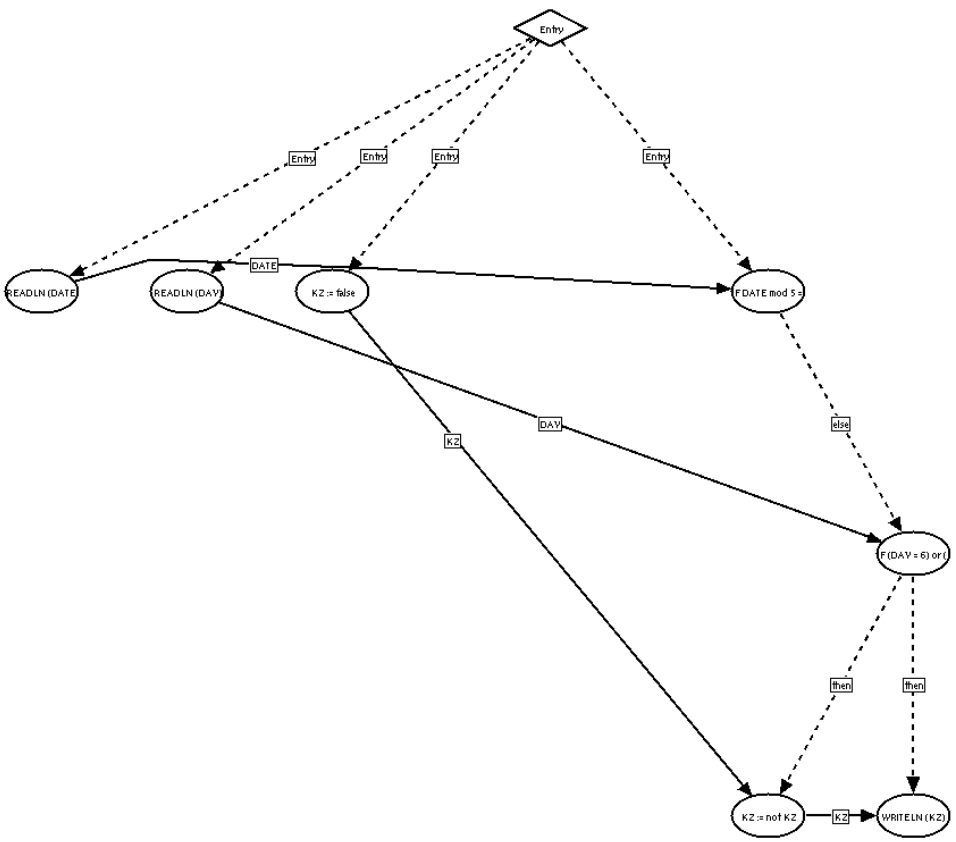

図 13: スライスの計算結果の例

5　考察

本ツールでは，Pascal のサブセットを入力言語として使用している．この言語は基本的な文しか扱っていないが，本質的な構造は全て含んでいる．さらに，REFINE のサブシステムである DIALECT を利用することにより構文解析における開発の労力を削減できるため，ポインタなど複雑な構造を使用しないと仮定すれば，C などの実用的言語を入力言語として適用することも可能であると考えられる，

現在のツールの問題点として，入力するプログラムが大きくなると PDG が複雑になり，PDG からプログラムの構造や依存関係を読みとることが困難になってくることがあげられる．改善策としては，プログラム全体を一つの PDG として表すだけでなく，関数単位で表示する機能をつける，節点の配置法を変更する，などが考えられる．

また，PDG の表示とスライスの計算の他に，計算したスライスの結果を利用した機能やソースコードの編集等の機能を付加することで，作業効率をより高めることができると考えられる．

6 あとがき

本研究では，入力されたプログラムの PDG を生成し，スライスの計算を行なうツールを試作した．本ツールは REFINE システム上で動作し，REFINE システムによって提供されている機能を利用することにより，生成した PDG を画面上に表示し，マウスによる簡単な操作でスライスを計算・表示する．PDG およびスライスを画面上に表示することにより，直観的にプログラムの構造を理解することができる．しかし，本ツールはまだ試作段階であるため，実用的なツールを作成するには以下に述べるような改善点が残されている．

一つには，PDG の表示方法の問題がある． 5 節で述べたように，サイズの大きなプログラムに対してはどうしても PDG が複雑になるため，プログラムの直観的な理解が困難になる．

二つ目には，付加機能の問題である．本ツールでは，スライスを視覚的に処理することを第一の目的としていたので，計算したスライスを利用する部分については実現できなかった．

以上の点が改善されれば実用的なツールになると考えられる．

謝辞

本論文を作成するにあたり，常に適切な御指導を賜わりました大阪大学 基礎工学部 情報工学科 (奈良先端科学技術大学院大学 情報科学研究科 併任) 鳥居 宏次 教授に心より深く感謝致します．

本論文の作成において，常に適切な御指導を頂きました大阪大学 基礎工学部 情報工学科 井上 克郎 助教授に深く感謝致します．

本論文の作成において，常に適切な御指導および御助言を頂きました同 飯田 元 助手に深く感謝致します．

本論文の作成において，適切な御指導を頂きました奈良先端科学技術大学院大学 情報科学研究科 島 和之 助手に深く感謝致します．

本論文を作成するにあたり，プログラムスライスに関する事項において多大な御助言を頂きました大阪大学基礎工学部 情報工学科 佐藤 慎一 氏，小林 孝規 氏に感謝致します．

最後に，その他様々な御指導，御助言等を頂いた大阪大学 基礎工学部 情報工学科 鳥居研究室の皆様に深く感謝致します．

参考文献

[1] Aho, A.V., Sethi, S. and Ullman, J.D.:“Compilers : Principles, Techniques, and Tools”, Addison-Weseley, 1986.

[2] Weiser,M.: “Program Slicing”,Proceeding of the Fifth International Conference on Software Engineering, pp.439–449(1981).

[3] Horwitz,S., Reps,T. and Binkley,D.: “Interprocedural Slicing Using Dependence Graphs”, Proceedings of the SIGPLAN ’88 Conference on Programming Language Design and Implementation, pp. 35–46(1988).

[4] Horwitz,S. and Reps,T.: “The Use of Program Dependence Graphs in Software Engineering”,Proceedings of the 14th International Conference on Software Engineering, pp.392–411(1992).

[5] Reasoning Systems, Palo Alto, CA, *REFINE User’s Guide* (1990).

[6] Reasoning Systems, Palo Alto, CA, *DIALECT User’s Guide* (1990).

[7] Reasoning Systems, Palo Alto, CA, *INTERVISTA User’s Guide* (1991).

[8] 下村 隆夫： “Program Slicing 技術とテスト，デバッグ，保守への応用”，情報処理，Vol.33, No.9, pp.1078–1086(1992).

[9] 植田 良一: “再帰を含むプログラムのスライスを求めるアルゴリズムの提案” ，大阪大学大学院基礎工学研究科修士学位論文 (1994).

[10] 佐藤 慎一: “プログラムスライスを用いたデバッグ支援システムの試作”，大阪大学基礎工学部情報工学科特別研究報告 (1994).