فهرست عناوین

عنوان | صفحه

فهرست شکل‌ها

عنوان **صفحه**

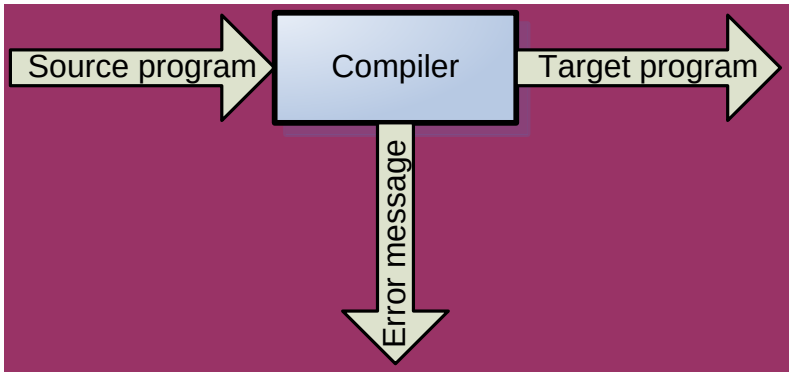
کامپایلر پیشرفته

مقدمه

در این درس به یادگیری روش های خلاصه سازی و تحلیل کد برنامه ها خواهیم پرداخت. ماحصل آن می تواند بهینه سازی کد باشد. همچنین روش های بهینه سازی کد، مورد بررسی قرار می گیرد. برای بهینه سازی کد کامپایلر باید قادر به تحلیل برنامه ها باشد. برای این منظور جریان کنترل اجرایی برنامه ها به صورت یک گراف مشخص می شود. سپس کامپایلر با استفاده از این گراف، جریان کنترل اجرایی وجریان انتقال داده را در داخل برنامه مورد بررسی قرار می دهد. نکته قابل توجه، ارائه روش هایی برای تبدیل برنامه به فرمی قابل تحلیل می باشد. گراف های جریان کنترلی و جریان داده ای، ابزاری برای تحلیل کد برنامه ها هستند. از این ابزار نه تنها برای بهینه سازی کد، بلکه می توان جهت آزمون صحت برنامه ها و تشخیص خودکار توازی کد نیز استفاده کرد.

یادآوری

کامپایلر، برنامه‌ای است که برنامه نوشته شده را به برنامه‌ای معادل به زبانی دیگر (زبان مقصد) ترجمه می‌کند.



شکل۱: کارکرد کلی کامپایلر

عملیات کامپایل در شش مرحله صورت می گیرد:

۱- تحلیل واژه‌ای

۲- تحلیل نحوی

۳- تحلیل معنایی

۴- تولید کد میانی

۵- بهینه‌سازی کد

۶- تولید کد نهایی

در مرحله تحلیل واژه‌ای، برنامه ورودی خوانده شده و به دنباله‌ای از توکن‌ها تبدیل می‌گردد.

در مرحله تحلیل نحوی، برنامه از نظر خطاهی نحوی مورد بررسی قرار می گیرد و با استفاده از نشانه‌های تولید شده در مرحله اول، یک درخت نحوی و یا درخت پارس، تشکیل می گردد.

در تحلیل معنایی، با استفاده از درخت تولید شده در مرحله قبل برنامه ورودی از نظر خطاهای مفهومی، چک می‌شود.

در مرحله تولید کدمیانی، یک برنامه که معادل برنامه اصلی است به یک زبان میانی تولید می گردد که با ایجاد کدمیانی، عملیات بعدی که کامپایلر انجام می‌دهد،ساده‌تر می گردد.

نهایتاً در بخش کد نهایی، کد برنامه، به زبان مقصد تولید می گردد. به عبارت دیگر هر کدام از کدهای میانی بهینه شده به مجموعه‌ای از دستورات ماشین که کار مشابهی انجام می‌دهند،مبدل می گردد.

کد میانی

هدف از بهینه سازی، تقلیل حجم کد و تسریع اجراء برنامه ها است. بهینه سازی بر روی کدمیانی انجام می گیرد زیرا کدمیانی ساده تر از کد عادی، قابل تبدیل به کد اسمبلی است. کدمیانی، کدی در سطح زبان ماشین است که مستقل از ماشین خاص می باشد. لذا به آن لفظ ماشین مجازی[۱] اطلاق می شود. کامپایلرنویس می تواند با نوشتن تولیدکننده برنامه برای هر سخت افزار امکان تولید برنامه برای آن سخت افزار را فراهم کند. بدین ترتیب قادر خواهیم بود تا برنامه را روی ماشین های مختلف به اجرا درآوریم. نتیجه ای که از آن می توان گرفت، قابلیت حمل[۲] برای کد میانی است.

برای نمونه زبان جاوا به عنوان زبان شبکه، از آنجاییکه نیاز بود برنامه هایی را بر روی سکوهای[۳] مختلف داشته باشد مجبور شد تا از روش کدمیانی استفاده نماید. کدمیانی که در زبان جاوا بکار می رود، Bytecode و ماشین مجازی آن JVM است. Bytecode را می توان با برنامه ساده ای به نام Jasmin بسادگی روی هر ماشین به اجرا در آورد. همچنین .NET. هم کد میانی IL یا CIL[۴] را مطرح کرد که ماشین مورد استفاده آن CLR نامیده می شود. اصولا IL و Bytecode زبان های شبه اسمبلی هستند و ماشین های مجازی آن ها مبنی بر ماشین پشته ای[۵] می باشند.

آنچه امروزه در کدمیانی رایج است، دستورالعمل های سه آدرسه و درخت های خلاصه نحوی[۶] می باشد. این ها به سادگی قابل تبدیل به کد اسمبلی هستند.

۱- Virtual Machine

۲- Portability

۳- Platform

۴- Common Intermidiate Language

۵- Stack Machine

۶- Abstract Syntax Tree

۲,۱ درخت خلاصه نحوی

درخت های خلاصه نحوی، درخت تجزیه ای است که در آن ترم های میانی حذف شده اند. درواقع شکل خلاصه شده از درخت تجزیه هستند که برای تولید کد میانی به کار رفته و به سادگی به کد اسمبلی تبدیل می شوند.

مقوله ترجمه با هدایت نحوی[۱] این امکان را فراهم آورده که بتوان همزمان با تحلیل نحوی و عمل پارسینگ , کد میانی را نیز ایجاد کنیم. برای تبدیل درخت تجزیه به درخت خلاصه نحوی به صورت زیر عمل می نماییم:

پرانتزها که بعنوان جداکننده و نشانگر اولویت هستند را از درخت تجزیه حذف می کنیم.

تک فرزندها را جایگزین پدرشان می نماییم.

ترم های میانی باقیمانده را با عملگرهایی جایگزین می کنیم که فرزند آن ترم باشد.

به این ترتیب مشاهده می شود که در یک درخت خلاصه نحوی برگها را عملوندها و گره ها را عملگر ها تشکیل می دهند.

برای نمونه گرامر زیر را درنظر بگیرید:

1. E → E + T
2. E →E – T
3. E → T
4. T → T * F
5. T → T / F
6. T → F
7. F → (E)
8. F → id
9. F → no

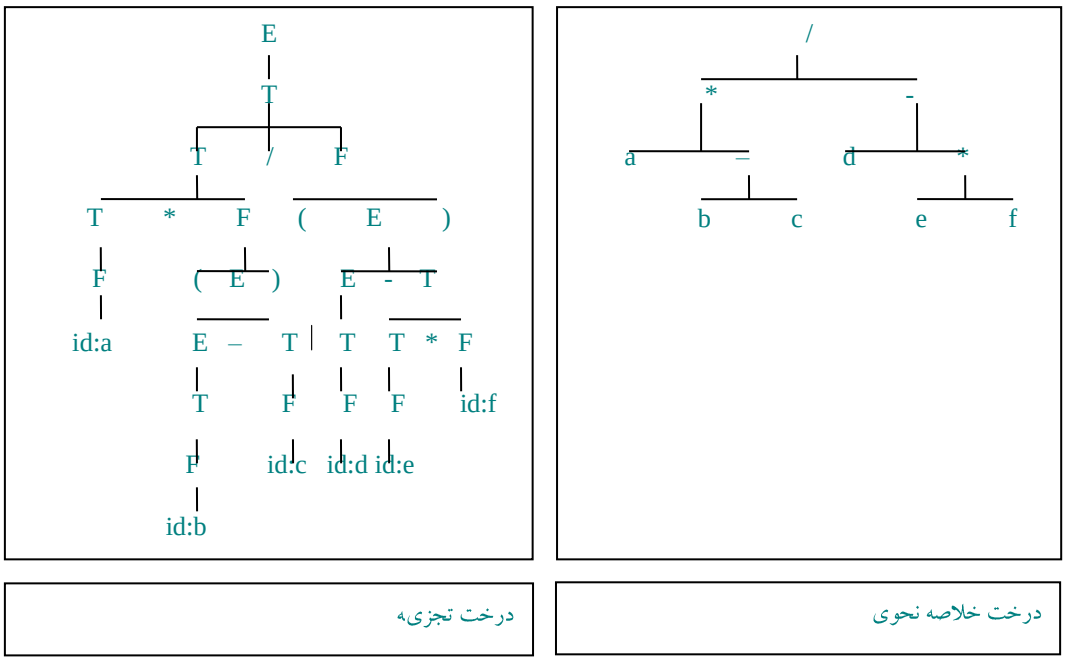
می خواهیم براساس گرامر فوق، برای عبارت زیر درخت خلاصه نحوی ایجاد نماییم.

a * (b – c) / (d – e * f)

ابتدا درخت تجزیه را رسم کرده، سپس طبق سه مرحله فوق عمل می نماییم. این مراحل در شکل زیر نشان داده شده است.

---

[۱] Sintax Direction



شکل ۱: درخت های تجزیه و خلاصه نحوی

البته باید توجه نمود که کلیه عملگرها لزوما باینری نیستند. درواقع باید گفت عملگرهای غیرباینری هم وجود دارند و برای آنها علامت های خاص یا عملگرهای ریاضی وجود ندارد. پس می توان به این نتیجه رسید که با پیمایش چپ، راست، ریشه روی آن، می توان کد ماشین ایجاد کرد. برای نمونه در مثال فوق خواهیم داشت:

abc-*def*-/

اکنون با استفاده از فرم پسوندی می توان بسادگی و با استفاده از ساختمان داده Stack کد اسمبلی را تولید کرد:

```
Mov ax, b
Sub ax, c
Mov bx, ax
Mov ax, a
Mul bx
Mov bx, ax
Mov ax, e
Mul f
Mov cx, ax
Mov ax, d
Sub ax, cx
Mov cx, ax
Mov ax, bx
Div cx
```

به گرامر زیر توجه نمایید. این گرامر یک زبان برنامه نویسی مانند پاسکال می باشد. این گرامر تنها دارای اپراتورهای باینری، علامت های خاص یا عملگرهای ریاضی نیست.

۱) S → Label Statement

۲) Label → id :

۳) | λ

۴) Statement → CompoundSt

۵) | AssignmentSt

۶) | CallSt

۷) | IfSt

۸) | ForSt

۹) | WhileSt

۱۰) Compoundst → begin Statements end

۱۱) Statements → Statement ';' S

۱۲) | S

۱۳) | λ

۱۴) AssignmentSt → id := E

۱۵) CallSt → id '(' ParameterList ')'

۱۶) WhileSt → While Condition do Statement

۱۷) ForSt → For id := E to E do Statement

۱۸) IfSt → If Condition then Statement Elsepart

۱۹) Elsepart → else Statement

۲۰) | λ

۲۱) ParameterList → ParameterList ';' id

۲۲) | id

۲۳) Condition → Condition or Cond

۲۴) | Cond

۲۵) Cond → Cond and C

۲۶) |C

۲۷) C → NOT Condition

۲۸) | '(' Condition ')'

۲۹) | Boolean

۳۰) Boolean → E Relop E

۳۱) | E

۳۲) | true

۳۳) | false

۳۴) Relop → <

۳۵) | <=

۳۶) | <>

۳۷) | >=

۳۸) | >

۳۹) E → E + T

۴۰) | E - T

۴۱) | T

۴۲) T → T * F

۴۳) | T / F

۴۴) | F

۴۵) F → id

۴۶) | No

۴۷) | '(' E ')'

۴۸) CaseSt → Case E of Caseparts Elsepart end

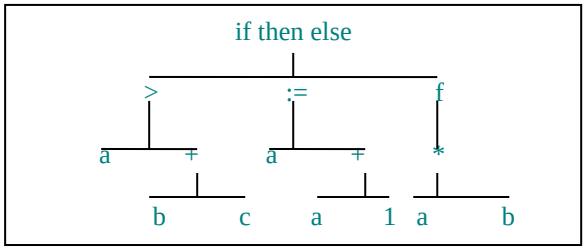
۴۹) Caseparts → Caseparts CasePart

۵۰) | CasePart

۵۱) CasePart → id : Statements

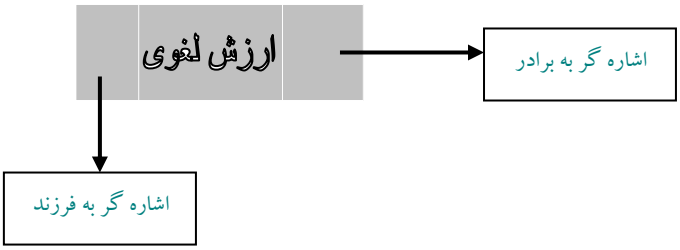
عبارت زیر را درنظر بگیرید:

```
if a > b + c
      then a := a + 1
      else f(a*b)
```



شکل ۲: درخت خلاصه نحوی عبارت If

اگر درخت تجزیه عبارت فوق را ترسیم کرده و طبق قوانین گفته شده، خلاصه نحوی می نمودیم حاصل همین درخت خلاصه نحوی می شد. در حالت کلی، برای پیاده سازی درخت خلاصه نحوی از ساختمان داده زیر استفاده می شود:



شکل۳: ساختمان داده عنصر درخت نحوی

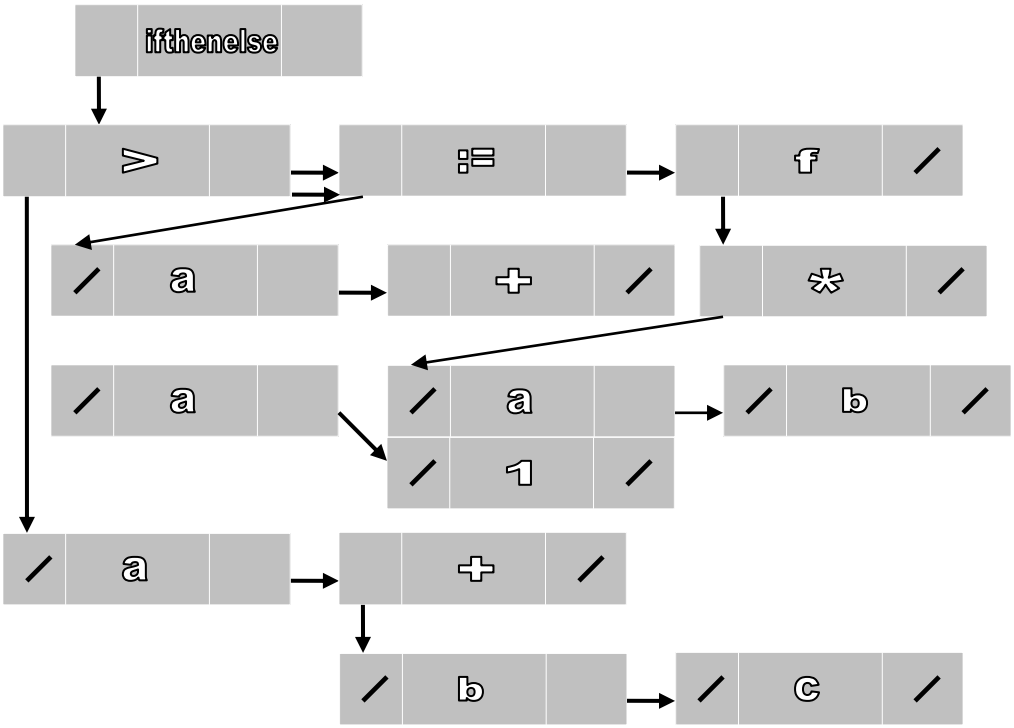
۲,۲ ترجمه با هدایت نحوی

منظور از ترجمه با هدایت نحوی تولید کد همگام با عمل تحلیل نحوی می باشد بعبارتی دیگر مراحل تحلیل نحوی خود شاخص مراحل تولید کد می باشد. برای این منظور برای هر ترم میانی یک ویژگی[۱] که شاخص کد میانی برای آن ترم میانی است , مشخص می شود . معمولا برای هر ترم میانی یک نشانگر بعنوان ویژگی مشخص می شود واین نشانگر در ضمت عمل تحلیل نحوی و معمولا در ترجمه پایین به بالا بعد از هر عمل reduce , به درختی اشاره خواهد کرد که تا آن زمان برای آن ترم میانی ایجاد شده است.

اما برای ایجاد این نشانگر ها تولید درختهای خلاصه نحوی عملیات[۲] باید انجام شود که این عملیات در کنار هر قاعده برای ترمهای میانی قرار داده می شود. در واقع به این عملیات ها که در داخل گرامر قرار داده می شود قواعد مفهومی[۳] گفته می شود بنابراین هر قاعده در گرامر شامل یک هم میگردد. برای نمونه به گرامر عبارات در ادامه توجه نمایید. با همراه کردن قواعد مفهومی با هر قاعده امکان تولید درخت خلاصه نحوی فراهم می گردد.

چگونه در ضمن تحلیل نحوی این اکشن ها به اجرا در می آیند و همگام با عمل تحلیل نحوی درخت خلاصه ایجاد می شود؟ معمولا بعد از هر عمل red اکشن مربوطه به اجرا در می آید و به این وسیله برای ترم میانی سمت چپ قاعده که عمل کاهش برای آن انجام شده, درخت خلاصه ایجاد می شود.

در حالت کلی همانگونه که گفته شد یک اپراتور زبان های برنامه سازی ممکن است چندین پارامتر داشته باشد. حال اگر بخواهیم مثال فوق را با طرح فوق ترسیم نماییم به شکل زیر می رسیم:



۱- attribute

۲- Action

۳- Semantic Rules

شکل ۴: ساختمان داده عبارت If در درخت خلاصه نحوی

مسئله ای که در اینجا مطرح است، چگونگی تولید کد میانی همگام با تولید درخت خلاصه نحوی است. بدین منظور برای ترم های میانی ویژگی هایی قرار داده که نشانگرهایی به درخت خلاصه نحوی می باشد. این تبدیلات بر اساس کد C++ ارائه می شود:

```
class CNode
{
  ValEnum Value;
  CNode * LeftChild;
  CNode * Sibling;
}



class AST
{
  CNode *Root;
  AST();
  void AddChild(CNode *  Father, CNode* Child);
  CNode* MakeNode(void * _Value);
  CNode* MakeNode(CNode *LeftChild , void * _Value, CNode *RightChild);
};


AST::AST()
{
   Root = NULL;
}

CNode*  AST::MakeNode(CNode *LeftChild , void * _Value, CNode *RightChild)
{
  CNode *TempNode = MakeNode(_Value);
  AddChild(TempNode,LeftChild);
  AddChild(TempNode,LeftChild);
  return TempNode;
}


CNode*  AST::MakeNode(void * _Value)
 {
   CNode *TempNode = new CNode;
   TempNode -> Value = _Value;
   TempNode -> LeftChild = NULL;
   TempNode -> Sibling = NULL;
   return TempNode;
 }


void AST::AddChild(CNode * Father, CNode* Child)
 {
   CNode *Temp = Father -> LeftChild;
   for(;;)
     {
       if(Temp == NULL)
        {
```

```
            Temp = Child;
            break;
           }
        Temp = Temp -> Sibling;
      }
    }


    CNode* AST::MakeNode(void *LeftChild ,void * _Value, CNode *RightChild )
    {
      CNode *TempNode = MakeNode(_Value);
      CNode *TempLeftChild = MakeNode(LeftChild);
      AddChild(TempNode,TempLeftChild);
      AddChild(TempNode,RightChild);
      return TempNode;
    }


    CNode* AST::MakeNode(CNode *LeftChild, void * _Value, void *RightChild)
    {
      CNode *TempNode = MakeNode(_Value);
      CNode *TempRightChild = MakeNode(RightChild);
      AddChild(TempNode,LeftChild);
      AddChild(TempNode,TempRightChild);
      return TempNode;
    }


    CNode* AST::MakeNode(void *LeftChild, void * _Value, void *RightChild)
    {
      CNode *TempNode = MakeNode(_Value);
      CNode *TempLeftChild = MakeNode(LeftChild);
      CNode *TempRightChild = MakeNode(RightChild);
      AddChild(TempNode,TempLeftChild);
      AddChild(TempNode,TempRightChild);
      return TempNode;
    }
```

```
S → label Statement          {Spntr = AST.MakeNode(labelpntr,S_Start,Statementpntr);}
label → id :                 {labelpntr = AST.MakeNode(id_lexval,S_label,S_colon); }
    |λ                       { labelpntr = NULL; }
Statement → Compoundst       {Statementpntr = Compoundstpntr;}
          |Assignmentst      {Statementpntr = Assignmentstpntr;}
          |Callst            {Statementpntr = Callstpntr;}
          |ifst              {Statementpntr = ifstpntr;}
          |forst             {Statementpntr = forstpntr;}
          |whilest           {Statementpntr = whilestpntr;}
Compoundst → begin           { Compoundstpntr = Statementspntr;}
Statements end

Statements → Statement ';' S {Statementspntr = AST.MakeNode(Statementpntr,S_Semicolon,Spntr);}
           |S                {Statementspntr = Spntr;}
           |λ                {Statementspntr = NULL;}
Assignmentst → id := E       {Assignmentpntr = AST.MakeNode(id_Lexval,S_assign,Epntr);}
Callst → id '(' ParameterList ')  { Callstpntr = AST. MakeNode(S_func);
                              AST.AddChild(Callstpntr,Parameterlistpntr);}
Whilest → while Condition    { AST.MakeNode(Conditionpntr,S_while,Statementpntr);}
do Statement

forst → for id := E to E do  { forstpntr = AST.MakeNode(S_for);
```

Statement	AssignTemppntr = AST.MakeNode(id_lexval, S_assign,Epntr); AST.AddChild(forstpntr, AssignTemppntr); AST.AddChild(forstpntr,Epntr); AST.AddChild(forstpntr,Statementpntr);}
ifst → if Condition do Statement elsepart	{ ifstpntr = AST.MakeNode(S_if); AST.AddChild(ifstpntr,Conditionpntr); AST.AddChild(ifstpntr,Statementpntr); AST.AddChild(ifstpntr,elsepartpntr);}
elsepart → else Statement	{elsepartpntr = Statementpntr;}
|λ	{ elsepartpntr = NULL;}
ParameterList → ParameterList ';' id	{ParameterList = AST.MakeNode(ParameterList,S_semicolon,id_lexval); }
| id	{ ParameterList = AST.MakeNode(id_lexval); }
Condition → Condition or Cond	{Conditionpntr = AST.MakeNode(Conditionpntr,S_or, Condpntr); }
| Cond	{Conditionpntr=Condpntr;}
Cond → Cond and C	{Condpntr = AST.MakeNode(Condpntr,S_and, Cpntr); }
|C	{Condpntr = Cpntr; }
C → NOT Condition	{ Cpntr = AST.MakeNode(S_Not); AST.AddChild(Cpntr,Conditionpntr); }
| '(' Condition ')'	{Cpntr = Conditionpntr;}
| boolean	{Cpntr = booleanpntr;}
boolean → E relop E	{ booleanpntr = AST.MakeNode(S_bool); AST.AddChild(booleanpntr,Epntr); AST.AddChild(booleanpntr,reloppntr); AST.AddChild(booleanpntr,Epntr);}
| E	{ booleanpntr = Epntr; }
| true	{ booleanpntr = AST.MakeNode(S_true); }
| false	{booleanpntr = AST.MakeNode(S_false);}
relop → <	{reloppntr = AST.MakeNode(S_l);}
|<=	{reloppntr = AST.MakeNode(S_le);}
|<>	{reloppntr = AST.MakeNode(S_NEq);}
|>=	{reloppntr = AST.MakeNode(S_ge);}
|>	{reloppntr = AST.MakeNode(S_lg;}
E → E + T	{Epntr = AST.MakeNode(Epntr,S_plus,Tpntr);}
| E – T	{Epntr = AST.MakeNode(Epntr,S_minus,Tpntr);}
| T	{Epntr = Tpntr;}
T → T * F	{Tpntr = AST.MakeNode(Tpntr,S_mul,Fpntr);}
| T / F	{Tpntr = AST.MakeNode(Tpntr,S_div,Fpntr);}
| F	{Tpntr = Fpntr;}
F → id	{Fpntr = AST.MakeNode(id_lexval);}
| No	{Fpntr = AST.MakeNode(No_lexval);}
| '(' E ')‘	{Fpntr = Epntr;}
Casest → Case E of Caseparts elsepart end	{ Casestpntr = AST.MakeNode(S_case); AST.AddChild(Casestpntr,Epntr); AST.AddChild(Casestpntr,Casepartspntr); AST.AddChild(Casestpntr,elsepart);}
Caseparts → Caseparts CasePart	{Casepartspntr = AST.MakeNode(Casepartspntr,S_CaseParts, Casepartpntr); }
| CasePart	{Casepartspntr = Casepartpntr;}
CasePart → id : Statements	{CasePartpntr = AST.MakeNode(id_lexval,S_semicolon,Statementspntr) ;}

جدول ۱: تبدیل گرامر به کد میانی از طریق درخت خلاصه نحوی

۲,۳ جملات سه آدرسه

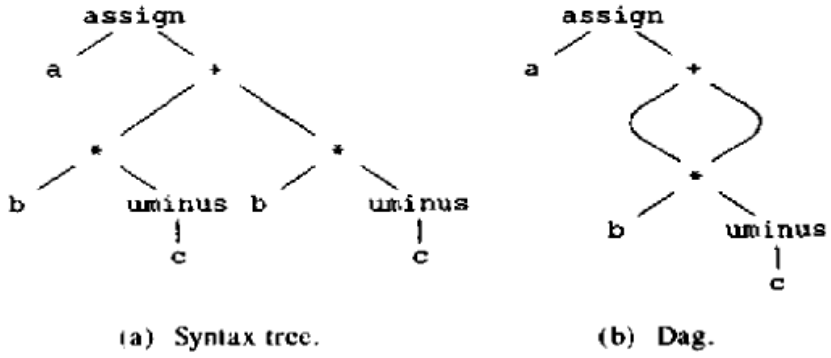
جملات سه آدرسه در حالت کلی x := y **op** z مشخص می شوند. در این حالت، x، y و z درواقع سه آدرس مختلف حافظه یا سه پارامتر را مشخص می کنند. لازم به ذکر است که در این قالب، حداکثر سه آدرس در انواع جملات باید وجود داشته باشد، اما ممکن است جملات تخصیصی نباشند. درصورتیکه برای مثال عبارت زیر را داشته باشیم:

x + y * z

عبارت فوق به جملات زیر ترجمه می گردد:

t1 := y * z
t2 := x + t1

t1 و t2 متغیرهای موقتی[۱]ای می باشند که توسط کامپایلر تولید می شوند. عبارت a := b * -c + b * -c را در نظر بگیرید. درخت خلاصه نحوی و برمبنای آن DAG برای این جمله تخصیصی بصورت زیر است:



(a) Syntax tree. (b) Dag.

شکل ۵: درخت نحوی و DAG

برای این جمله فرم سه آدرسه بشرح زیر می باشد:

Code for the Syntax tree :

t1 := -c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a := t5

Code for the DAG :

t1 := -c
t2 := b * t1
t5 := t2 + t2
a := t5

البته می توانستیم تعداد متغیرهای موقتی را کاهش دهیم و به حداقل ممکن برسانیم. به مثال زیر توجه نمایید:

a := b * (c – d) * e / f

t1 := c – d
t1 := b * t1
t1 := t1 * e
t1 := t1 / f
a := t1

[۱] – temporary variable

۲,۴ انواع دستورالعمل های سه آدرسه

دستورالعمل سه آدرسه مجموعه کوچکی هستند که می توان هر نوع دستورالعمل در زبان های برنامه سازی را برمبنای آنها بیان کرد. انواع آنها عبارتند از:

۱. جملات تخصصی در فرم کلی x := y **op** z هستند.

x := **op** y, x := y, x := &y, x := *y, x[i] := y, x := y[i], *x := y

در مورد آرایه ها لازم به ذکر است که باید بصورت دستی اشاره به حافظه را انجام دهید. یعنی اگر داده ای که در آرایه قرار گرفته است از نوع integer باشد، باید محتوای اندیس آرایه را در ۴ ضرب نمایید تا باندازه یک عدد باینری، به خانه حافظه بعدی اشاره نماید.

۲. جملات فراخوانی در فرم کلی P(x1, x2, …, xn) بصورت زیر تبدیل می شود.

push x1, push x2, …, push xn, call p, n

۳. جملات شرطی if در فرم پرش شرطی[۱] بصورت زیر مشخص می گردد.

if x **relop** y
goto l

۴. جملات انتقال کنترل goto در فرم کلی goto n.

تمرین: جمله زیر را به فرم سه آدرسه تبدیل کنید.

```
if a + b > c * d – e
  then
       begin
         a := a + b * c;
         write(a, b, c * d – e)
       end
else
     read(a, i, j);
```

```
  t1 := a + b
  t2 := c * d
  t2 := t2 – e
  if not (t1 > t2)   goto L1
  t3 := b * c
  a := a + t3
  push a
  push b
  push t2
  call write, 3
  goto L2
L1: push a
    push I
    push j
    call read, 3
L2:
```

[۱] – conditional jump

جمله زیر را به فرم سه آدرسه تبدیل کنید.

```
case a * b – c of
    1: if a > 2 * j – 1
          then
               a := b * c + d;
    2: writeln(a * b – 1, c / d);
    3: begin
          b: a * c – d;
          write(b * 2);
       end;
    else: a := a * b – c * d * e + f * y
```

```
  t1 := a * b
  t1 := t1 – c
  if not (t1 = 1) goto L1
  t2 := 2 * j
  t2 := t2 – 1
  if not (a > t2) goto L2
  t2:= b * c
  a := t2 + d
  goto L2
L1: if not (t1 = 2) goto L3
  t2 := a * b
  t2 := t2 – 1
  push t2
  t2 := c / d
  push t2
  call writeln, 2
  goto L2
L3: if not (t1 = 3) goto L4
  t2 := a * c
  t2 := t2 – d
  push t2
  call write, 1
  goto L2
L4: t1 := a * b
  t2 := c * d
  t2 := t2 * e
  t1 := t1 – t2
  t2 := f * g
  a := t1 + t2
L2:
```

البته می توان همگام با عمل تحلیل نحوی، تولید کد میانی را نیز انجام داد. در ادامه برای این منظور، گرامر جملات تخصیصی را ارائه داده و نشان می دهیم که چگونه می توان همگام با تحلیل نحوی، جملات کد میانی را نیز ایجاد کرد. بدین جهت ابتدا چند تابع ارائه می کنیم.

```
var
        TempNo, LabelNo: integer;

procedure Init;
begin
        TempNo := 0;
        LabelNo := 0;
        …
end;

function NewTemp: string;
begin
        TempNo := TempNo + 1;
        NewTemp := ‘T’ + int2str(TempNo);
end;

procedure RemoveTemp;
begin
```

```
        TempNo := TempNo - 1
end;

function NewLabel: string;
begin
        LabelNo := LabelNo + 1
        NewLabel := 'L' + int2str(LabelNo);
end;

function isTemp(T: string): boolean;
begin
        …
end;

procedure Emitln(S: string);
begin
        writeln(target, s);
end;
```

بین هر ترم میانی متغیری از نوع رشته ای تخصیص داده شده که ویژگی ترم میانی را مشخص می نماید. در عمل این ویژگی شاخص کدی است که برای ان ترم میانی تولید می گردد. تابعی به نام Emitln، کد میانی را ایجاد می کند. همچنین تابعی به نام isTemp مشخص می کند که آیا ویژگی در آن رشته ذخیره می شود حاوی یک Temporary است یا خیر؟

در عمل تجزیه پایین به بالا معمولا از پشته ای به نام پشته تجزیه استفاده می شود. هر بار وابسته به اینکه در بالای پشته تجزیه چه ترمی قرار گرفته و اینکه چه ترمی در سر ورودی باید از فایل منبع[۱]خوانده شود، برطبق جدول تجزیه تصمیم می گیرد که آیا عمل Shift، Reduce و یا Goto باید انجام دهد. بعد از هر عمل Reduce برطبق یک قاعده، عمل[۲]مربوط به آن قاعده اجرا می گردد.

اکنون در داخل گرامر، تعدادی عمل قرار می دهیم تا پس از هر عمل کاهش[۳]در تجزیه پایین به بالا، اگر عمل ها اجرا شوند، نهایتا کد سه آدرسه ایجاد گردد. به این نوع گرامرها در اصطلاح گرامر ویژه[۴]گفته می شود. درواقع گرامر ویژه برای هر ترم میانی[۵]یک مشخصه[۶]درنظر می گیرد. هر مشخصه، متغیری از نوع رشته[۷]است که حاصل کد تولید شده برای آن ترم میانی خواهد بود.

۱- Source File

۲- Action

۳- reduce

۴- attributed grammer

۵- non-terminal

۶- attribute

۷- string

Ast → id := E	{Emitln(id.Lexval + ‘:=’ + Eval); if isTemp(Eval) then RemoveTemp;}

E → E’ + T	{if isTemp(E’val) then begin
	Eval := E’val;
	if isTemp(Tval) then
	RemoveTemp;
	end;
	else if isTemp(Tval) then E’val := Tval;
	Else Eval := NewTemp;
	Emitln(Eval + ‘:=’ + E’val + ‘+’ + Tval);}

E →T	{Eval := Tval;}

T → T * F	{**if** isTemp(T^1val)
	then begin
	Tval := T^1val;
	if isTemp(Fval) then
	RemoveTemp;
	end;
	else if isTemp(Fval) then T^1val := Fval;
	Else Tval := NewTemp;
	Emitln(Tval + ‘:=’ + T^1val + ‘*’ + Fval);}

T → F	{Tval := Fval;}
F → (E)	{Fval := Eval;}
F → id	{Fval := id.Lexval;}

جدول ۲: تبدیل گرامر به کد میانی

حال مسئله این است که ما چگونه می توانیم بصورت خودکار کد تولید نماییم:

St → ASt | IfSt	
IfSt → If Condition	{Lthen := NewLabel ; Emitln(‘if ‘ + Conditionval + ‘ goto ‘ + Lthen) ; if isTemp(Conditionval) then RemoveTemp; Lelse := NewLabel; Emitln(‘goto ‘ + Lelse); Emitln(Lthen + ‘: ‘);}
Then St	{Lend := NewLabel; Emitln(‘goto ‘ + Lend); Emitln(Lend + ‘: ‘);}
ElsePart	{Emitln(Lend + ‘: ‘);}
ElsePart → None	
ElsePart → else St	

Condition → Condition’ AND B	{if isTemp then Condition’valm then
	Begin
	Conditionval := Condition’val;
	if isTemp(Bval) then RemoveTemp;
	End
	else if isTemp(Bval) then Conditionval := Bval;
	else Condition’val := NewTemp;
	Emitln(Conditionval + ‘:=’ + Condition’val + ‘ and ‘ + Bval);}

Condition → Condition’ OR B	{if isTemp then Condition’valm then
	begin
	Conditionval := Condition’val;
	if isTemp(Bval) then RemoveTemp;
	end;

	else if isTemp(Bval) then Conditionval := Bval;
	else Condition'val := NewTemp;
	Emitln(Conditionval + ':=' + Condition'val + ' or ' + Bval);}

Condition → NOT Condition'	Emitln(Conditionval + ':=' + 'not ' + Condition'val);}

B → E Relop E'	{if isTemp(E'val) then begin
	Bval := Eval;
	if isTemp(E'val) then
	RemoveTemp;
	end;
	else if isTemp(E'val) then Bval := E'val;
	Else Bval := NewTemp;
	Emitln(Bval + ':=' + Eval + Relop + E'val);}

B → True	{Bval := 'true';}
B → False	{Bval := 'false';}
B → id	{Bval := id.lexval;}

تمرین: گرامر جملات While و For را در زبان پاسکال نوشته، برای آن ابتدا با مثالی کد سه آدرسه، درخت خلاصه نحوی و سپس گرامر ویژه بنویسید.

۲,۵ ابزار ANTLR

ANTLR یک ابزار برای شناسایی زبان و دارای چارچوبی برای ساخت مفسرها، کامپایلرها و مترجم‌ها از یک سری توصیفات گرامری می باشد. ANTLR ساخت درخت تجزیه، پیمایش درخت، ترجمه، ترمیم خطا و گزارش خطا را به صورت کامل پشتیبانی و ابزاری برای آنها فراهم کرده است. ANTLR همچنین شامل یک محیط توسعه یافته برای گرامرها به نام ANTLRWORKES می باشد که اعمال یادشده را به صورت گرافیکی و با یک واسط کاربری ساده پشتیبانی می کند.

ANTLRWORKS یک محیط کامل برای گرامرهای ANTLR می باشد که تلاش می کندکه کاربران بتوانند به ایجاد و توسعه ی گرامرها بپردازند. اجزاء اصلی آن یک ویرایشگر گرامر با امکانات مختلف، یک مفسر گرامر و یک debugger گرامر می باشد. Debugger به صورت پویا یک جریان ورودی را نمایش داده ، درخت تجزیه را ایجاد و نمایش می دهد.

ANTLR یک تحلیل گر لغوی قدرتمند دارد که این پویشگر، جریان ورودی از کاراکترها را به سمبول‌ها و واژگان برای پارسر تبدیل می کند. از آنجا که ANTLR از یک سری از مکانیسم های شناسایی خاص برای تحلیل گر لغوی ، تجزیه و ایجاد درخت تجزیه استفاده می کند خروجی تحلیل‌گر لغوی ANTLR خیلی قدرتمندتر ازتحلیل گرهای لغوی بر مبنای DFA که در اغلب محصول استفاده می شود است.

۲,۵,۱ ایجاد درخت AST

ANTLR درخت های AST را ایجاد کرده و با افزودن یک سری اعمال و بازنویسی قوانین، کار را برای استفاده از AST آسان می کند. همچنین اجازه ی تعیین کردن ساختار گرامری AST را می دهد. البته تجزیه گر درخت در ANTLR می تواند هر درخت را که واسط AST را پیاده سازی کند پیمایش کند.

در اینجا تشخیص دهنده می تواند برای سه نوع ساختار ورودی ایجاد شود :

- جریان کاراکتری
- جریان توکن ها
- ساختار درخت های دو بعدی

این سه نوع ساختار توسط تحلیل گر لغوی، پارسرها و پیمایشگر درخت ها بررسی می شوند.

همه ی گرامرهای ANTLR زیر کلاس هایی از Lexer و Parser و TreeParser هستند. به عنوان مثال وقتی که می خواهیم گرامری برای قوانین نحوی بنویسیم باید از کلاس Parser ارث ببرد. برای تحلیل لغوی نیز باید از کلاس Lexer ارث برد. در مرحله ی بعد با استفاده از ANTLR برنامه ای به زبا ن تعیین شده مثل Java ایجاد می کنیم. این برنامه ی ایجاد شده برای هر قانون تجزیه یک متد به زبان Java ایجاد می کند. در نتیجه قوانین به فراخوانی های متد و توکن ها هم به فراخوانی تابع match(TOKEN) ترجمه می شوند. می توان یک برنامه به زبان Java نوشت و از این کلاس های ایجاد شده توسط ANTLR استفاده کرد. برای این کار در متد main() برنامه ابتدا یک نمونه از زیر کلاس Lexer که قبلا ایجاد شد ساخته و ورودی را که می خواهیم بررسی کنیم به عنوان پارامتر به سازنده ی آن می دهیم. سپس یک نمونه از زیر کلاس Parser که قبلا ایجاد کرده بودیم ساخته و نمونه ی زیر کلاس Lexer را به عنوان پارامتر به سازنده ارسال می کنیم. حال می توان متدهای Parser را روی آن فراخوانی کرده و ورودی را بررسی کنیم.

برای ایجاد درخت AST در زیر کلاس Parser نوشته شده و در قسمت option ایجاد درخت را هم با نوشتن BuildAST = true اضافه می کنیم. در اینجا نیز می توان یک برنامه به زبان Java نوشت و با فراخوانی متد getAST() از زیر کلاس Parser ایجاد شده و قرار دادن آن در یک شیء از کلاس AST از آن استفاده کرده و یا آنرا پیمایش کرد. البته می توان یک زیر کلاس از کلاس TreeParser ایجاد کرد و پیمایش یا اعمال دیگر را بر روی AST انجام داد.

در زیر کد مربوطه آمده است:

```
import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;
public class Main {
   public static void main(String args[]) throws Exception {
      SLexer lex = new SLexer(new
ANTLRFileStream("C:\\mohamad\\compiler\\java\\SParser\\input"));
      CommonTokenStream tokens = new CommonTokenStream(lex);
      SParser g = new SParser(tokens);
      try {
          SParser.program_return r= g.program();
          System.out.print(""+((Tree)r.getTree()).toStringTree());
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

در ادامه مثال هایی آمده است :

مثال : ورودی

```
Int  x;
Int  I;
For(i=0;i<10;i=i+1){
X=i+x;
}
```

خروجی :

```
Tree= (VAR_DEF int x) (VAR_DEF int i)(for(=i0)(<I 10)(=i(+I 1))(BLOCK(=x(+I
x))))
```

مثال : ورودی

```
Do{
X=x+1;
}While(x<5)
```

خروجی :

```
Tree: do{(=x(+x 1)}while((<x 5))
```

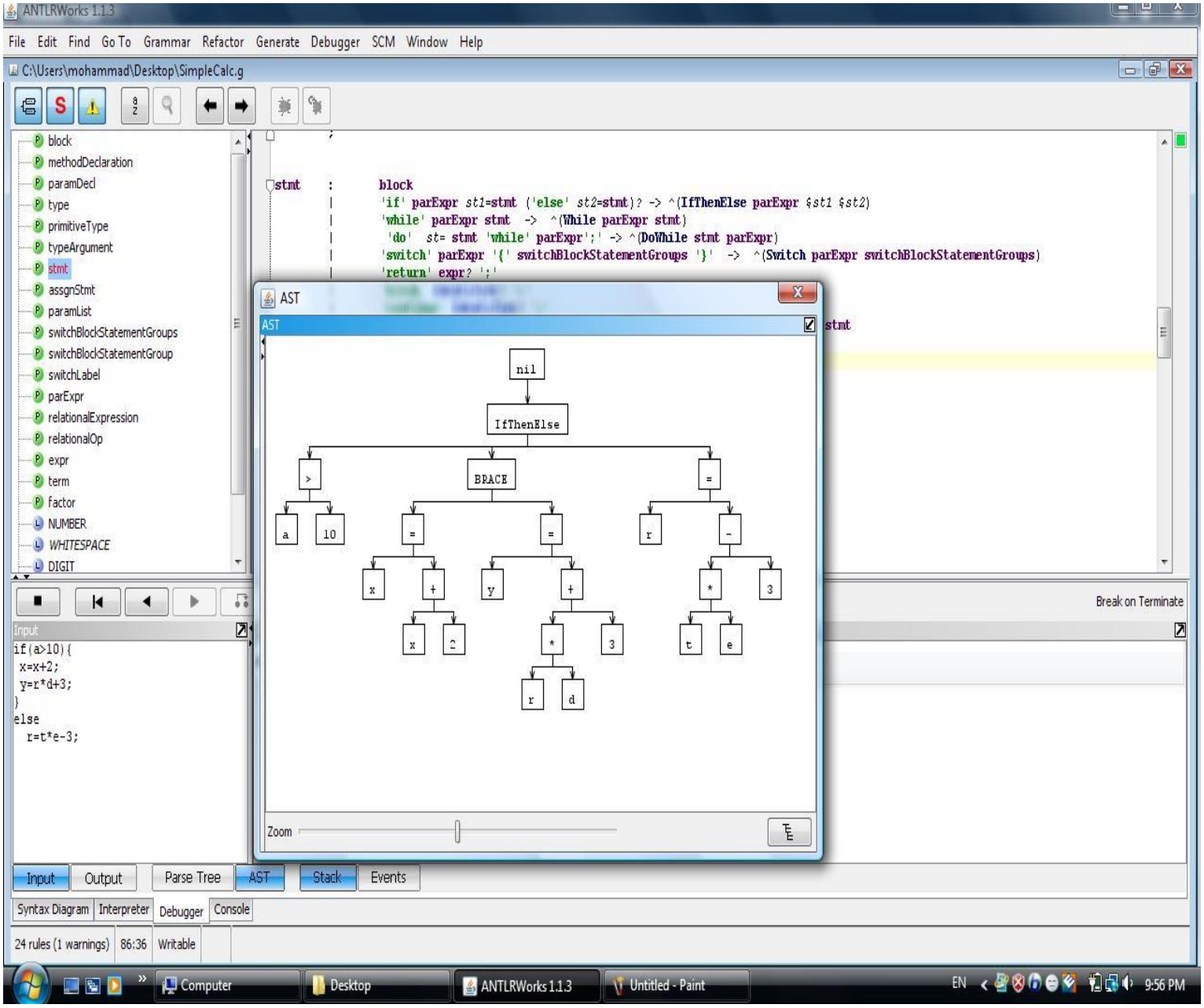
مثال : ورودی

```
char c;
int x;
void bar(int x);
int foo(int y, char d) {
  int i;
  for (i=0; i<3; i=i+1) {
    x=3;
    y=5;
  }
}
```

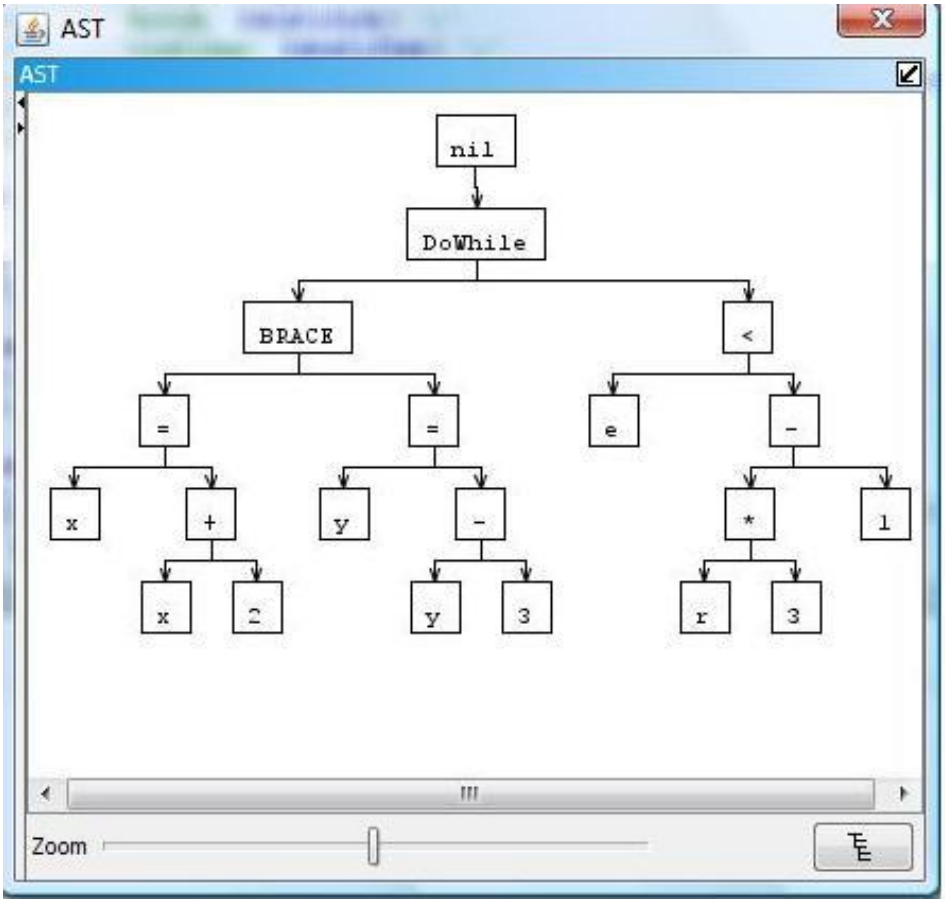
خروجی :

```
(VAR_DEF char c) (VAR_DEF int x) (FUNC_DECL (FUNC_HDR void bar
(ARG_DEF int x))) (FUNC_DEF (FUNC_HDR int foo (ARG_DEF int y)
(ARG_DEF char d)) (BLOCK (VAR_DEF int i) (for (= i 0) (< i 3) (= i (+ i 1))
(BLOCK (= x 3) (= y 5)))))
```

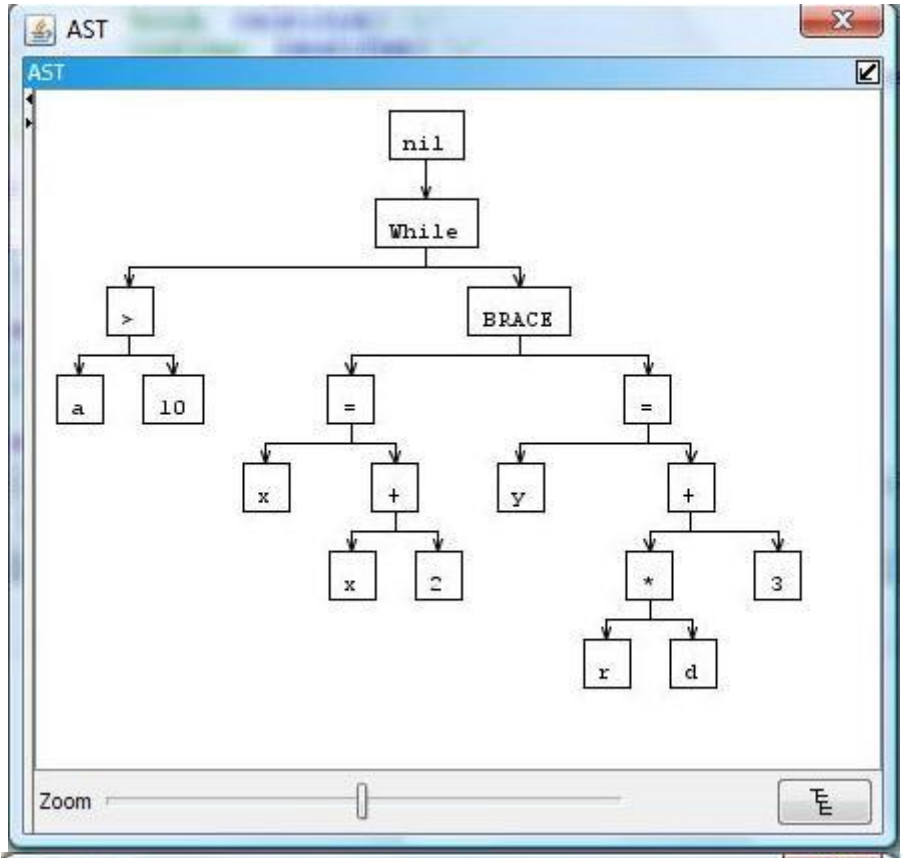
در ادامه مثال هایی با محیط گرافیکی آمده است :

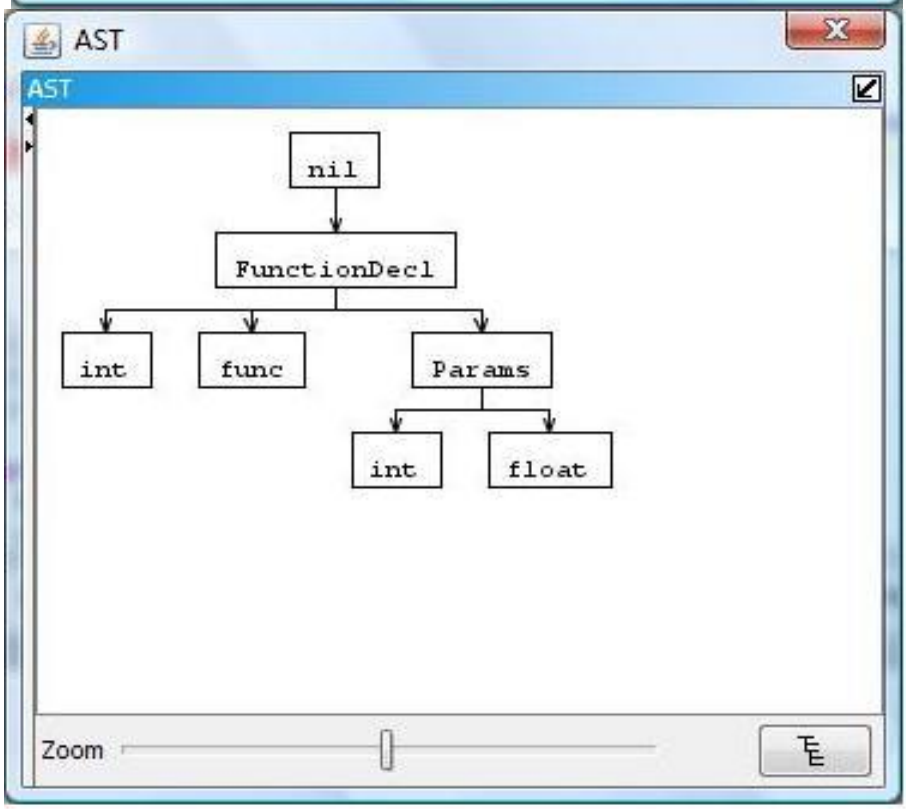

ANTLRWorks 1.1.3
File Edit Find Go To Grammar Refactor Generate Debugger SCM Window Help
C:\Users\mohammad\Desktop\SimpleCalc.g
block
methodDeclaration
paramDecl
type
primitiveType
typeArgument
stmt
assgnStmt
paramList
switchBlockStatementGroups
switchBlockStatementGroup
switchLabel
parExpr
relationalExpression
relationalOp
expr
term
factor
NUMBER
WHITESPACE
DIGIT
stmt : block
'if' parExpr st1=stmt ('else' st2=stmt)? -> ^(IfThenElse parExpr $st1 $st2)
'while' parExpr stmt -> ^(While parExpr stmt)
'do' st= stmt 'while' parExpr';' -> ^(DoWhile stmt parExpr)
'switch' parExpr '{' switchBlockStatementGroups '}' -> ^(Switch parExpr switchBlockStatementGroups)
'return' expr? ';'
stmt
AST
AST
nil
IfThenElse
>
BRACE
=
a
10
=
=
r
-
x
+
y
+
*
3
x
2
*
3
t
e
r
d
Zoom
Break on Terminate
Input
if(a>10){
x=x+2;
y=r*d+3;
}
else
r=t*e-3;
Input
Output
Parse Tree
AST
Stack
Events
Syntax Diagram
Interpreter
Debugger
Console
24 rules (1 warnings)
86:36
Writable
Computer
Desktop
ANTLRWorks 1.1.3
Untitled - Paint
EN
9:56 PM

مثال : حلقه ی DoWhile



مثال : در صفحه ی بعد مثالی از حلقه ی While و Function decleration آمده است.


AST
AST
nil
While
>
BRACE
a
10
=
=
x
+
y
+
x
2
*
3
r
d
Zoom


AST
AST
nil
FunctionDecl
int
func
Params
int
float
Zoom

تولید کد[1]

بعد از اینکه کد میانی تولید شد، مرحله تولید کد آغاز می گردد. در مرحله ی تولید کد در واقع ورودی کد میانی ، همچنین جدول نمادها است و خروجی در واقع برنامه ی مورد نظر است . نکته اساسی در اینجاست که برنامه تولید شده در حداقل زمان لازم به اجرا در آید. نکته ی قابل توجه در تخصیص ثبات هاست. استفاده از ثبات ها به عنوان مکان های داخلی cpu موجب می گردد که نه تنها اندازه کد تولیدی کمتر گردد بلکه سرعت اجرایی آن نیز افزایش یابد.

البته قبل از اینکه کد میانی به عنوان ورودی برای مرحله ی تولید کد مورد استفاده قرار گیرد ممکن است عمل بهینه سازی برای کد میانی انجام شود و سپس عمل تولید کد انجام گیرد و پس از تولید کد مرحله دیگری به نام peephole optimization که در واقع نوعی بهینه سازی وابسته به ساختار ماشین است انجام می گیرد. برای عمل تولید کد, کد میانی را در قالب گرافی به نام گراف جریان[2]یا گراف جریان کنترلی[3]تبدیل می کند. در این گراف هر گروه یک بلاک اولیه[4]نام دارد که شامل دستور العملهایی است که به صورت متوالی به اجرا در می آیند. در مورد گراف های جریان کنترلی بعدا توضیح خواهیم داد.

مساله ی اصلی که در این بخش با آن مواجه هستیم, مساله تشخیص ثبات هاست.

یک نکته در مورد مولدهای کد, طریقه ی نمایش کد میانی و نمایش جدول نمادها است. کد میانی ممکن است در قالب دستورالعمل سه آدرسه در فرم های سه تایی و چهارتایی نمایش داده شود, ممکن است در قالب درختهای خلاصه نحوی و یا نمایش خطی این درختها در قالب نمایش postfix مشخص کردد. کد میانی ممکن است برای مولد کد در قالب DAG مشخص شود, معمولا پس از تشخیص عبارات تکراری در داخل ASTاز تکرار خودداری نموده , محلهایی که به ریشه زیر درخت مربوط به آن عبارت را اشاره می کنند را همگی به یک محل اشاره می دهند, بدین

---

[1] - code generation

[2] - flow graph

[3] - control flow graph

[4] - basic block

ترتیب درخت به یک گراف جهتدار فاقد حلقه[1]تبدیل می شود. ممکن است ورودی در قالب Byte Code, IL و یا کد یک Stack machine برای مولد کد مشخص گردد. بنابراین می بینیم که ورودی با یک استاندارد خاص مشخص نمی گردد. این یک مساله و مشکل است. چگونه می توان ورودی این مرحله را استاندارد نمود تا مولد کدها بتوانند یکسان و برای پردازنده های خاص به صورت استاندارد مشخص شوند؟

شاید یک راه حل تبدیل AST و یا کد سه آدرسه به XML باشد, مساله دیگر و یا برنامه نهایی[2]است. برنامه ی نهایی ممکن است برنامه به زبان دیگر باشد. برای نمونه در سیستم عامل یونیکس برای کلیه کامپایلرها برنامه ی نهایی به زبان C می باشد. برنامه هدف ممکن است Object code باشد. در اینجاست که مجموعه دستورالعملها و ثبات ها نقش عمده ای در تولید کد بهینه به عهده دارند.

معمولا سه معماری برای پردازنده ها وجود دارد که عبارتند از: RISC, CISC و Stack- based machine.

در ماشین های RISC, تعداد ثبات ها نسبتا زیاد است. دستورالعملها سه آدرسه هستند. مجموعه ی دستورالعمل ها کم و دستور العمل ها ساده می باشند. ساختار RISC در انگلستان اولین بار مطرح شد.

در ماشین های CISC معمولا تعداد ثبات ها کم است, دستور العملها دو آدرسه هستند, آدرس دهیهای متفاوت وجود دارد مثل مستقیم, غیر مستقیم, index و .... registerها دسته بندی میشوند و چندین دستور العمل وجود دارد. دستور العملها output مختلف دارند. همچنین دستورالعملهایی که Side effect دارند تاثیرات جانبی دستور العملها ممکن است در قالب افزایش میزان انرژی مصرفی ظاهر شود که این مساله در پردازنده های RISC نیز مطرح است لذا در هنگام تولید کد شاید به عنوان یک Peephole optimization سعی بر آن شود که دستو العمل ها به ترتیبی قرار داده شوند که انرژی مصرفی به حداقل ممکن کاهش یابد.

در ماشین های Stack based عملیات بر روی یک پشته داخلی انجام میشود, بدین ترتیب که Operand ها در داخل Stack, push و برای انجام عملیات از آن pop می گردند و معمولا برای افزایش سرعت, top یا بالای Stack در بالای یک ثبات نگهداری می شد. به خاطر کندی عملکرد این نوع cpuها در عمل کنار گذاشته شد. البته این گونه معماری در ماشین های مجازی مانند JVM و CIL هنوز نیز به خاطر سادگی ساختاری و سهولت در تولید کد مورد استفاده قرار می گیرد. البته همانگونه که می دانید زبان Java و محیط net. مبتنی بر ماشن مجازی می باشند که در این ماشینهای مجازی که امروزه در قالب لایه های زیرین سیستم عامل مطرح شده اند, دستورالعمل ها interprete می گردند. مشکل interprete یا تفسیر تک به تک دستور العمل ها, مقوله context switch می باشد که زمان اجرایی را بسیار طولانی می کند. در مفسرها کد میانی مثل Byte code یا IL هربار با فعال شدن مفسر برای هر دستور العمل کد میانی به کد ماشین تبدیل و سپس به اجرا در آورده می شود. به این ترتیب اگر حلقه ای ۱۰۰۰۰۰ بار تکرار میشود, ۱۰۰۰۰۰ بار برای هر دستور العمل داخل حلقه در فرم Byte code و یا هر فرم دیگر مفسر باید فعال شود و عمل تولید

---

[1] - DAG(Directed Acycled Graph)

[2] - Target program

کد را انجام دهد و کد را به اجرا در آورده و با حفظ core Image برای برنامه اجرایی دوباره فعال گردد. برای حل این مشکل مقوله ای به نام Just intime compilers (JIT) مطرح شده است. JIT compiler در زمان اجرا, Byte Code را تبدیل به کد ماشین میکنند و از تکرار تبدیل کد جلوگیری به عمل می آورد. سوال اینجاست که چگونه میتوان کارآیی JIT را افزایش داد.

خروجی می تواند به صورت کد مطلق باشد. یعنی کدی که آدرس شروع آن مشخص و دستور العمل آدرسهایشان نسبی نیست مستقیما به خانه ی حافظه اشاره می کنند که در نتیجه سرعت اجرای دستورالعمل ها بالا می رود چرا که جهت آدرس دهی نیاز به عمل جمع با آدرس نیست. معمولا در سیستم های کنترلی و سیستمها ی توکار[1] این سیستم مورد استفاده قرار می گیرد. کد تولید شده ممکن است جابجا پذیر[2] باشد. تولید کد اسمبلی به جای Object code کار مرحله تولید کد را ساده تر می کند.

در جلسه قبل در مورد کلیت تولید کد و انواع دستور العملهای اسمبلی که در ساختارهای مختلف RISC, CISC و Stack machine ممکن است ظاهر شود به بررسی پرداختیم. در بخش ۸,۱,۳ کتاب در مورد انتخاب دستور العمل ها مطالبی ارائه شده. در واقع مجموعه دستور العمل ها در یک فرم میانی به نمایش در می آیند. مشکلی وجود دارد عدم وجود یک نمایش میانی استاندارد می باشد که همین امر موجب شده مرحله ی تولید کد در قالب خاصی در نیاید. پیچیدگی تولید کد که در واقع کار آن تصویر نمایش میانی به کد ماشین است وابستگی به پیچیدگی میانی خواهد داشت. همچنین مجموعه ی دستور العمل های اسمبلی امکاناتی که برای برنامه نویس فراهم میکند اینها همگی در میزان پیچیدگی مولد کد تاثیرگذارند. در کتاب دستور العملهای خاص اسمبلی ارائه شده است که می توان به جای آنها از اسمبلی سری x86 استفاده کرد. نکته اصلی در مرحله تولید کد مسئله ی مدیریت ثبات ها است. (اسمبلی mainframe)

۳,۱ مدیریت ثبات ها[3]

یک مساله اصلی در هنگام تولید کد استفاده بهینه از ثبات هاست و جایگزینی ثبات ها با متغییرها. برای مثال قطعه کد زیر را در نظر بگیرید:

```
i = 2 ;
i = i+i+1  ;
```

در اینجا مولد کد به جدول ثبات ها ارجاع میکند. فرض کنید مشاهده میکند ثبات CX آزاد است. در داخل این جدول ثبات Cx علامت "اشغال شد" و نام متغییر i را قرار میدهد. همچنین در جدول نمادها در ستونی مجزا محل نگهداری مقدار i را ثبات Cx مشخص می کند. حالا به جای اینکه در هنگام تولید کد از متغییر i استفاده شود از Cx استفاده می شود و قطعه کد فوق به صورت زیر تبدیل می شود:

```
mov  cx, 2   ; i = 2
mov  ax, cx   ; ax = i
mul  ax   ; ax = i × i
inc  ax  ; ax = i × i + 1
```

---

[1] - embedded system

[2] - relocatable

[3] - register management

mov i , ax ;

دستور العمل هایی که از ثبات ها استفاده می کنند کوتاه تر و سرعت اجرایی آنها نسبتا سریع تر از دستور العمل هایی است که به حافظه ارجاع می کنند. بنابراین بهره وری بهینه از ثبات ها نکته ای حائز اهمیت در مرحله ی تولید کد است. به کارگیری ثبات ها دو زیر مساله را مطرح می کند:

۱ _ تعیین ثبات[۱] که در ضمن آن مجموعه متغیرهایی که ثبات به آنها تخصیص باید داده شود مشخص می شود. البته این مجموعه در نقاط مختلف برنامه وابسته به شرایط تغییر می کند.

۲ _ تخصیص ثبات[۲] : در این مرحله متغییرهایی که تعیین شده باید در ثبات قرار گیرند برایشان ثباتی مشخص شده و تخصیص می یابد.

نکته ی قابل توجه، در مدیریت ثبات هاست که یک مساله ی بهینه سازی و بنا به گفته کتاب یک مساله NP-Complete است.

برخی از ثبات ها ممکن است به عنوان آکومولاتور مورد استفاده قرار گیرند یا اینکه ممکن است سیستم عامل آنها را رزرو کرده باشد. اینها مسائله تولید کد یا در واقع مساله مدیریت ثبات ها را پیچیده تر میکند. بنابراین مشاهده می شود که می توان به عنوان یک مساله بهینه سازی NP-Complete از روشهای مکاشفه ای[۳] و روشهای غیرقطعی برای حل این مساله استفاده کرد. البته مطالبی در مورد تحلیل کد نیز در اینجا مطرح است به خصوص مساله استفاده های بعدی[۴] از متغیرها. می توان با تحلیل کد به دست آمده که آیا متغیری مثل i بعد از این استفاده خواهد شد و میزان ارجاع به آن چیست. این اطلاعات در نگهداری ثبات ها برای متغییرها و تعیین اولویت برای نگهداری ثبات ها مورد استفاده قرار می گیرد.

۳,۲ مفسرهای ثبات و آدرس

مدیریت ثبات ها برای اینکه بتواند به عنوان یک کلاس مورد دسترسی مولد کد قرار گیرد باید بتواند وضعیت استفاده از ثبات ها را کنترل نماید. برای این منظور از مفسر ثبات[۵] و مفسر آدرس[۶] استفاده می کنند. مفسر آدرس در واقع یک جدول به صورت زیر است:

نام ثبات	نام متغیر	وضعیت	نام ثبات
CX		اشغال	AX
		ـــــ	BX
	i	اشغال	CX

۱- register allocation
۲- register assignment
۳- heuristic
۴- next-use
۵- register descriptor
۶- address descriptor

DX	——

جدول مفسر ثبات

نام	گروه	offset	مقدار	نوع	ثبات	نشانگر به مفسر نوع
i	S.Var	….	——	Integer	CX	Null
⋮	⋮	⋮	⋮	⋮	⋮	⋮

جدول نمادها(مفسر آدرس)

در واقع برای استفاده از این مفسرها تابعی به نام Getreg مطرح می شود. تابع Getreg برای دستور العملهای سه آدرسه ثبات تعیین می کند. تابع Getreg مفسرهای آدرس و ثبات برای هر بلاک اولیه دسترسی دارد. همچنین اطلاعاتی در مورد زنده بودن متغییرها در هنگام خروج از بلاک در دست دارد. زنده بودن متغیر ها الگو ریتم خاص خود را دارد که در مورد آن به بررسی خواهیم پرداخت و مشخص می کند که بعد از خروج از بلاک اولیه آیا متغییر مقدارش مورد استفاده قرار میگیرد یا خیر. یک بلاک اولیه به دنباله ای از دستور العمل های سه آدرسه اطلاق می گردد که اولا از داخل آنها پرشی به نقطه دیگر وجود ندارد و ثانیا از نقاط دیگر به وسط آنها پرش نمی شود. اگر پرشی وجود دارد از آخرین دستور العمل در داخل بلاک اولیه انجام میگیرد و یا به اولین دستور العمل در داخل بلاک اولیه پرش می شود بدین ترتیب گرافی به نام گراف جریان کنترلی از بطن برنامه ها استخراج می شود.

در دستور العمل سه آدرسه ی x=y+z و '+' در واقع یک عملگر جنریک است. (پارامترها می توانند هر نوعی باشند). فرض کنید برای عملگر + در ساده ترین حالت دستور العمل اسمبلی Add مورد استفاده قرار گرفته. مسلما y+z و z+y مقدارشان با یکدیگر فرق ندارد. حالا ببینیم که چگونه می توان با در نظر گرفتن وضعیت ثبات ها برای این دستور العمل یعنی x=y+z کد اسمبلی تولید کرد. الگوریتمی نوین حاوی چهار مرحله اصلی ارائه شده است است:

قبل از اینکه به بیان الگوریتم جدید GetReg بپردازیم, الگوریتم قبلی را که توسط Aho مطرح شده ارائه می کنیم. بر طبق الگوریتم تولید کد ارائه شده توسط Aho برای هر دستورالعمل سه آدرسه مثل x: = y op z عملیات زیر انجام می شود:

۱ _ تابع GetReg فراخوانی می شود تا مکان L که قرار است در آن حاصل y op z نگهداری شود مشخص شود. معمولا برای انجام یک عمل از آکومولاتور استفاده می شود اما این امکان وجود دارد که L یک مکان حافظه باشد.

۲ – با ارجاع به مفسر آدرس برای y, مکان 'y که ممکن است یک حافظه یا یک ثبات باشد و در آن مقدار y نگهداری میشود مشخص شود. مسلما ثبات ترجیح داده می شود. حالا دستور العمل mov y' , L تولید می شود تا مکان y در L ذخیره گردد.

۳ – در دستور العمل MOV , op z' , L ایجاد شود که در اینجا z' مشابه y' برای z است.

حالا مفسر آدرس برای x را باید به روز رسانی کرد تا مشخص شود که مقدار x در داخل L نگهداری می شود. اگر L ثبات باشد باید در داخل مفسر ثبات مشخص کرد که L مقدار x را نگهداری می کند.

۴ – چنانچه بر طبق تحلیلهایی که بعدا خواهید نمود مشخص شود که y یا z استفاده بعدی ندارند یا به عبارت دیگر در هنگام خروج از بلاک زنده نیستند در این صورت ثبات های مربوط به y یا z قابل آزاد سازی هستند. توجه کنید الگوریتم next use مشخص می کند که در داخل بلاک اولیه آیا مقدار متغییر مورد نظر بعد از x = y op z استفاده می شود . الگوریتم زنده بودن[۱]مشخص می کند که آیا متغییر مورد نظر در جریان اجرایی برنامه بعد از این در سایر بلاکها مورد استفاده قرار خواهد گرفت.

در دستور العمل x=y مسلما اگر y در داخل ثبات نگهداری شود حالا آن ثبات علاوه بر y مقدار x را هم نگهداری می کند و چنانچه y در داخل حافظه نگهداری شود صرفا در داخل مفسر آدرس مشخص می کنیم که مقدار x در همان مکانی است که مقدار y قرار گرفته. البته به خاطر پیچیده شدن الگوریتم می توانیم از تابع GetReg استفاده کنیم تا یک ثبات در ا ختیار ما قرار دهد و مقدار y را در آن ثبات قرار دهیم و آن ثبات را به x اختصاص دهیم. اما شکل سوم این است که دستورالعمل mov y, x را ایجاد کنیم این در صورتی است که x داخل بلاک , استفاده بعدی نداشته باشد. البته با استفاده از الگوریتم انتشار کپی که بعدا توضیح داده خواهد شد می توان تمام دستورالعملهای x = y را که کار کپی کردن را بر عهده دارند حذف کرد. در هنگام خروج از بلاک های اولیه می بایست ثبات ها را آزاد کرد. البته این به واسطه ساده تر شدن کار مولد کد است و می توانستیم این کار را نکنیم. برای این منظور تمام متغییرهایی که زنده هستند و پس از این مقدارشان مورد استفاده قرار می گیرد و ثباتی بدان ها تخصیص داده شده با ایجاد یک دستورالعمل Mov مقدار ثبات را در داخل متغییر ذخیره و ثبات ها را آزاد می کنیم.

۳,۴ تابع GetReg

تابع GetReg برای گرفتن ثبات مورد استفاده قرار می گیرد و در واقع کار مدیریت ثبات ها را بر عهده دارد در واقع این تابع برای دستور العملی مثل x = y op z مکان L را که می بایست مقدار x در آن نگهداری شود مشخص می کند.

---

[۱] – live

الگوریتم Get Reg به صورت زیر عمل می کند:

۱ – اگر مقدار y در داخل یک ثبات نگهداری می شود و y در داخل بلاک استفاده بعدی ندارد و بعد از این زنده نیست آنگاه تابع GetReg ثبات تخصیصی به y را برگشت می دهد, در ا ینصورت برای دستورالعمل x = y op z صرفا دستورالعمل op L , z' ایجاد می گردد که در اینجا L همان y' است. باید صرفا مفسر ثبات برای y' به روز رسانی شود و متغییر x را به عنوان اشغال کننده آن تعیین کنیم به همین ترتیب مفسر آدرس برای y می بایست تغییر کند و برای X.

۲ – در صورتی که در مرحله ۱ موفق نبودیم در صورت وجود، یک ثبات آزاد را به L باید تخصیص داد.

۳ – در صورتی که ثبات آزادی وجود نداشته باشد در این صورت می بایست چنانچه در داخل بلاک اولیه برای X مورد استفاده بعدی یا next use وجود داشته باشد آنگاه باید یک ثبات را آزاد کرد. برای مثال اگر ثبات R باشد و متغییری که مقدار آن را نگهداری می کند m, دستور العمل mov m, R باید ایجاد شود (در داخل فایل هدف که توسط مولد کد ایجاد می شود) و به این ترتیب ثبات R آزاد می شود البته در اینجا فرکانس استفاده از متغییرها در داخل بلاک اولیه میتواند عاملی برای دادن اولویت جهت آزاد کردن ثبات ها باشد.

۴ – در صورت عدم موفقیت در سه مرحله ی بالایی همان مکانی را که x اشغال کرده تابع Get Reg به عنوان هدف در نظر می گیرد. برای مثال دستورالعمل زیر را در نظر بگیرید:

d : = (a-b) + (a-c) + (a-c)

با در نظر گرفتن اینکه عمل تولید کد بر روی دستورالعمل های سه آدرسه انجام می گیرد اینکه t, u و v سه متغییر موقتی هستند این دستورالعمل به صورت زیر تبدیل خواهد شد:

```
t := a - b
u := a - c
v := t + u
d := v + u
```

البته در اینجا تولید کد اسمبلی وابسته به این است که مقدار ثبا ت ها چند عدد می باشد. در شکل ۹–۱۰ کد اسمبلی ایجاد شده با در نظر گرفتن دو ثبات R_0 و R_1 می باشد.

```
mov R0 , a
sub R0 , b          ; t = R0 = a - b
mov R1, a
sub R1, c; u = R1 = a - c
add R0 , R1         ; v = R0 = t + u
```

add R_0 , R_1 ; d = v + u
mov d , R_0

در کتاب جدید Aho این الگوریتم به صورتی دیگر بیان شده, مراحل آن در ادامه توضیح داده خواهد شد.

۳,۵ مدیریت مفسرهای ثبات و آدرس

در بخش ۶-۸ کتاب تحت عنوان یک مولد کد ساده الگوریتمی برای تولید کد در داخل بلاکهای اولیه ارائه شده است. همان طور که قبلا توضیح داده شد بلاک اولیه دنباله ای از جملات سه آدرسه می باشد که

الف) یا با نقطه شروع برنامه آغاز میشوند و یا اینکه اولین جمله ی آنها مقصد یک Go to است.

ب) آخرین جمله یا آخرین جمله برنامه است و یا خود یک Goto می باشد. (Goto ممکن است شرطی یا غیرشرطی باشد.)

ج) به وسط این دنباله از جملات پرشی وجود ندارد.

د) از وسط این جملات هیچ پرشی به جای دیگر وجود ندارد.

الگوریتم ساده ی تولید کد برای داخل هر بلاک اولیه مطرح شده است. هدف در این است که حداکثر استفاده از ثبات ها انجام می شود. در هنگام استفاده از ثبات ها چهار قانون اصلی را می بایست در نظر داشت:

۱ – در برخی از معماری ها برخی از عملوندها برای یک عمل می بایست در داخل ثبات نگهداری شوند. برای مثال در دستور العمل Mov یا بعضا load یکی از عملوندها می بایست در برخی از ماشین ها ثبات باشد. مثال

mov x,5 ⟶

mov Ax , 5
mov x , Ax

۲ – معمولا ثبات ها جایگزین خوبی برای متغییرهای موقتی هستند. معمولا زیر عبارات حاصلشان در داخل متغییرهای موقتی نگهداری می شد که در واقع هر زیر عبارت[۱] شاخص یک محاسبه می باشد و محاسبات در ثبات ها امکان پذیر می گردد.

۳ – ثبات ها معمولا توسط کامپایلرها برای حفظ مقادیر global یا سراسری و مقادیری که در بین block های اولیه رد و بدل می شوند مورد استفاده قرار می گیرند.

---

۱ – sub expression

۴ – برای مدیریت حافظه که به صورت پشته[۱]جهت فراخوانی های خود بازگشتی مطرح است معمولا آدرس بالای پشته در داخل ثبات نگهداری می شود.

در داخل کتاب دستور العمل ها در فرم کلی زیر مشخص شده اند:

LD reg , mem
ST mem, reg
OP reg, reg, reg

تعریف مفسرهای ثبات و آدرس قبلا ارائه شد که در بخش ۱-۶-۸ توضیح داده شده است. در صفحه ۵۴۴ کتاب بخش ۲-۶-۸ تابع GetReg(i) توضیح داده شده در واقع i در اینجا یک دستورالعمل سه آدرسه است و تابع GetReg در واقع ثبات ها را مشخص می کند. فرض بر این است که تابع GetReg اولاً به مفسرهای آدرس و ثبات برای کلیه متغییرها در هر بلاک اولیه دسترسی دارد و ثانیا به برخی از اطلاعات جریان داده[۲]در بین بلاک های اولیه مثل زنده بودن متغییرها در هنگام خروج از بلاک اولیه دسترسی دارد. منظور از زنده بودن یک متغیر دسترسی به مقدار آن می باشد. اگر در هنگام خروج از این بلاک اولیه تابع GetReg به این نتیجه برسد که متغیری زنده نیست این امر در تخصیص ثبات برای این تابع تاثیرگذار خواهد بود. الگوریتم تشخیص متغیرهای زنده در بخشهای بعدی توضیح داده خواهد شد. الگوریتم به صورت زیر است:

برای دستور العمل سه آدرسه ی x = y+z

۱) تابع GetReg را به صورت GetReg (x = y+z) در نظر بگیرید تا این تابع ثبات ها را برای x , y و z در قالب R_x, R_y و R_z مشخص کند.

۲) چنانچه مقدارy در داخل یک ثبات مثل R_y نگهداری نمی شود در آن صورت در خروجی یعنی در کد تولید شده دستور العمل y' و LD R_y ایجاد شود. در اینجا y' مکانی است که مقدار y را نگهداری می کند. حالا مقدار y' در داخل ثبات R_y کپی می شود.

۳) چنانچه مقدار z در داخل ثباتی مثل R_z نگهداری نمی شود آنگاه دستورالعمل z' و LD R_z درخروجی قرار داده می شود تا مقدار z نیز در داخل یک ثبات نگهداری شود.

۴) در خروجی دستور العمل R_z و R_yو ADD R_x قرار داده می شود تا بدین ترتیب عمل جمع انجام گردد. توجه نمایید که در این قسمت در الگوریتم جدیدی که ارائه شده هیچ گونه بهینه سازی در استفاده از ثبات ها وجود ندارد اما برای انجام بهینه سازی در داخل کتاب در صفحه ۵۴۵ چهار قانون کلی مطرح شده است.

قانون ۱: برای دستور العمل x و LD R:

---

۱ – stack

۲ – data flow

الف) مفسر ثبات برای R تغییر داده شود تا مشخص گردد که R مقدار x را در خود نگهداری میکند.

ب) مفسر آدرس برای x تغییر داده شود و R را به عنوان مکانی دیگر که مقدار x را نگهداری می کند مشخص کنیم.

قانون ۲: برای دستور العمل R و ST x, مفسر آدرس برای x تغییر داده شود تا صرفا x به آدرس تخصیصی به خود اشاره کند و تعیین شود که در off set مشخص شده برای x مقدارش نگهداری می شود.

قانون ۳: برای جمله سه آدرسه ی x = y+z دستور العمل R_z و R_y و Add R_x ایجاد میشود. در اینجا مفسر ثبات برای R_x که خود یک ثبات است مقدار x را به عنوان متغیری که مقدارش در ثبات R_x قرار داده شده نگهداری می کند و به همین ترتیب مفسر آدرس برای x فقط ثبات R_x را مشخص می کند. بدین ترتیب R_x صرفا در اختیار متغیر x قرار می گیرد.

قانون ۴: برای دستورالعملی مثل x=y , دستورالعمل y و LD R_y در صورتی که y در هیچ ثباتی قرار نداشته باشد ایجاد می شود و مفسر ثبات برای R_y به متغیرهای x و y اشاره می کند و بالعکس مفسر آدرس برای x و y هردو به R_y اشاره می کند.

نمونه هایی از مفسرهای ثبات و آدرس و کد اسمبلی ایجاد شده در شکل ۱۶-۸ برای دنباله ای از دستورالعملها در صفحه ۵۴۶ ارائه شده است. در این مثال با استفاده از توابع next use و تابع تشخیص متغیرهای زنده عمل مدیریت ثبات انجام شده است. برای دستور ا لعمل t=a-b کد ایجاد شده به صورت زیر است:

LD R_1, a
LD R_2, b
Sub R_2, R_1, R_2

در ا ینجا با توجه به اینکه برای a در جملات بعدی بلافاصله ارجاعی وجود دارد و از مقدار a استفاده می شود بنابراین ثبات R_1 نگهداری شده است و بالعکس چون b مورد استفاده بعدی ندارد ثبات مربوط به آن آزاد شده و حاصل تفریق a و b را که در t ذخیره می شود در R_2 نگهداری می کند.

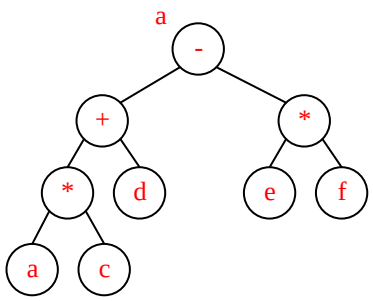
در انتهای کد تولید شده در شکل ۱۶-۸ مشاهده می کنیم که با اجرای دو دستور العمل St d , R_1 ST a وR_2 مقدار a و d از ثبات های مربوطه برداشته می شود و در Offset این دو متغیر در حافظه ذخیره میگردد اما برای متغیر v این چنین نیست. در اینجا مساله زنده بودن متغیرها بعد از خروج از بلاک اولیه مطرح است.

الگوریتم GetReg در بخش ۳-۶-۸ از صفحه ۵۴۷ کتاب جدید Aho ارائه شده است. این الگوریتم دقیقا مشابه همان الگوریتم قبلی است که در کتاب قدیمی تر Aho ارائه شده. مشکل در اینجاست که حتی در این الگوریتم نیز برخلاف مثالی که در اینجا ارائه شده مساله استفاده بعدی را مدنظر نداشته است.

البته بعد از اینکه کد اسمبلی ایجاد می شود بهینه سازی هایی بر روی آن انجام می گیرد تا در واقع با در نظر گرفتن سرعت اجرایی دستور العملها و امکانات سخت ا فزاری کد حاصل بهینه گردد. به اینگونه بهینه سازی ها لفظ peephole optimization اطلاق می شود. در واقع بهینه سازی به دوصورت انجام می گیرد: یک مرحله بهینه سازی بر روی کد میانی انجام می گیرد. برای مثال کدهای میانی سه آدرسه یا درختهای خلاصه با در نظر گرفتن دو معیار سرعت اجرایی و حجم فضای اشغالی بهینه سازی می شوند و علاوه بر این بعد از اینکه کد اسمبلی تولید شد برروی کد اسمبلی نیز بهینه سازی لازم با در نظر گرفتن امکان جایگزینی دستور العملها با دستورالعملهای سریعتر و حذف دستورالعمل های زاید انجام می گیرد. الگوریتم GetReg برای جملات سه آدرسه مطرح شد. اما نوع دیگر کد میانی در قالب درختهای خلاصه نحوی مطرح بود.

عبارت: a = b * c + d – e * f

```
mov  R1 , b
mul  R1, c          ; R1=b*c
add R1, d           ; R1= b * c + d
mov R2, e
mul R2, f           ; R2= e* f
sub R1, R2
mov a , R1
```



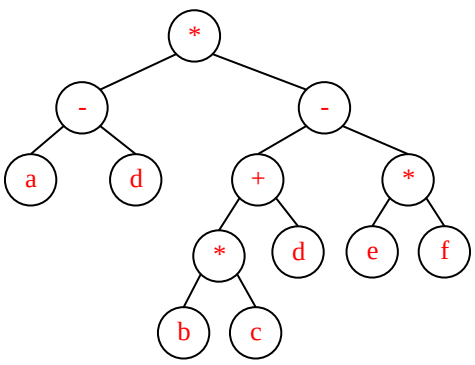
درخت خلاصه مربوط به عبارت

در کتاب قدیم Aho دستورالعملهای اسمبلی حتما می بایست پارامتر اولشان یک ثبات باشد اما پارامتر دوم لزوما یک ثبات نیست و می تواند بر خلاف اسمبلی سه آدرسه ارائه شده در کتاب جدید یک مکان حافظه باشد. نکته, استفاده بهینه از ثبات هاست که نشان داده خواهد شد طریق پیمایش درخت خلاصه ی نحوی می تواند در این مورد بسیار تاثیر گذار باشد. برای نمونه به مثال زیر توجه نمایید. در این مثلا نشان داده خواهد شد که نحوه ی پیمایش تاثیر بسزایی در اندازه و سرعت اجرایی کد اسمبلی حاصل خواهد داشت. فرض را بر این می گذاریم که تعداد ثبات ها دو عد باشد: R_1 و R_2

عبارت : a –b * b * c + d - e * f

۱ : پیمایش چپ به راست

```
mov  R1 , a
sub   R1 , a        ; a - b
mov  R2 , b
mul   R2 , c        ; b * c
add   R2 , d        ; b * c + d
mov  T1, R1
```



درخت خلاصه مربوط به عبارت

```
mov  R1 , e
mul   R1 , f        ; e * f
sub    R2 , R1      ; b * c + d - e * f
mov   R1 , T1
mul    R1 , R2      ; a –b * b * c + d - e * f
```

۲ : پیمایش بهینه

```
mov  R1 , b
mul   R1 , c        ; b * c
add   R1 , d        ; b * c + d
mov  R2 , e
mul   R2 , f        ; e * f
sub   R1 , R2       ; b * c + d - e * f
mov  R2 , a
sub    R2 , b       ; a - b
mul    R2 , R1      ; a –b * b * c + d - e * f
```

همان گونه که مشاهده می کنید با پیمایش جدید میزان کد اسمبلی تولید شده کمتر و سرعت اجرایی بسیار بیشتر است.

نشان داده شد که چگونه روش پیمایش درخت خلاصه نحوی می تواند در کارایی و اندازه کد تولید شده مؤثر باشد. اصولا هنگامی که در داخل یک بلاک اساسی مساله تولید کد مطرح است تابع next use می تواند مؤثر واقع گردد. مساله متغیرهای زنده هم در یک سیاست سراسری[۱] برای تخصیص ثبات ها مطرح است. در هنگام خروج از بلاک های اولیه متغیرهایی که پس از خروج از آن بلاک زنده می مانند اینها مقدارشان در داخل متغیرهای مربوطه ذخیره می شد و ثبات مربوطه برای آنها نگهداری میشد. به خصوص در ارتباط با حلقه هایی که شمارنده های حلقه در داخل ثبات نگهداری می شوند خروجی می تواند در سرعت اجرایی کد تاثیر گذار باشد. البته در زبانهایی مثل C این امکان وجود دارد که متغیرهایی را از نوع ثبات تعریف کنیم. در اینجا برنامه نویس اکیدا تاکید می کند که مقدار متغیر حتما باید در داخل ثبات حفظ شود.

۳,۶ تعیین میزان استفاده

Usage count در ارتباط با میزان استفاده از ثبات ها مطرح است برای این منظور هزینه ی دسترسی به حافظه و هزینه ی دسترسی به ثباتها را مطرح می کند تا به این وسیله مشخص شود که آیا متغیری در داخل ثبات نگهداری شود مقرون به صرفه است و یا در داخل حافظه. حال متغیر ممکن است شمارنده یک حلقه باشد و یا هرنوع متغیر دیگر. دسترسی به ثباتها را برایشان هزینه ۱ در نظر می گیرند و برای دسترسی به خانه های حافظه هزینه ۲ و سیاست را بر این قرار می دهند که اگر مقدار x در یک بلاک محاسبه گردد این مقدار در داخل یک ثبات نگهداری می شود. بنابراین اگر x مجددا مقدارش در داخل بلاک اولیه تعریف نگردد, آنگاه هر بار که به x دسترسی می شود و مقدار آن مورد استفاده قرار می گیرد مقدار واحد به نفع خواهد بود. بنابراین اگر این متغیر در داخل یک حلقه مورد استفاده قرار گیرد

[۱] - global

به تعداد تکرارهای حلقه منفعت وجود خواهد داشت.

اگر x در داخل یک بلاک اولیه مقدارش تعریف شود و بعد از آن بلاک اولیه x زنده باشد آنگاه مقدار ۲ واحد منفعت برای هر بلاک که x در هنگام خروج از آن زنده است و در داخل آن بلاک به x مقداری داده شده وجود خواهد داشت, از طرفی چنانچه x در هنگام ورود به یک حلقه زنده باشد قبل از ورود به حلقه باید x را در داخل یک ثبات نگهداری کنیم و این هزینه ۲ را در برخواهد داشت. به همین ترتیب طبق سیاست قبلی در هنگام خروج از هر بلاک اولیه در داخل یک حلقه در صورتی که x زنده باشد می بایست در انتهای بلاک اولیه مقدار x را از ثبات تخصیصی به X به داخل مکان تخصیصی به x در حافظه ذخیره نماییم. این هم ۲ واحد هزینه خواهد داشت. بنابراین میتوان منفعت تخصیص ثبات به متغیری مثل X در داخل حلقه را از رابطه زیر محاسبه کرد:

در رابطه فوق برای هر بلاک اولیه B در داخل حلقه L, use (x, B) تعداد دفعاتی که به x در داخل B, قبل از تعریف x استفاده شده را مشخص می کند. Live (x,B) در صورتی که x در هنگام خروج از B زنده باشد و در داخل B تعریف شده باشد مقدار ۱ را می گیرد. در غیر اینصورت مقدار ۰ را خواهد گرفت. البته باید توجه کرد که رابطه ی فوق یک رابطه تقریبی است زیرا اولا ممکن است در یک تکرار حلقه یک یا چند بلاک اولیه اصلا اجرا نشوند و ثانیا تعداد تکرارهای حلقه ممکن است مشخص نباشند.

میزان استفاده در سیاست تصمیم گیری برای آزاد سازی ثباتها ی تخصیصی به متغیرها مطرح است. در شکل ۱۳-۹ از کتاب قدیمی AHO مثالی در مورد میزان استفاده مطرح شده است.

قبل از اینکه مساله تخصیص ثبات ها و تولید کد برای درختهای خلاصه نحوی بپردازیم باید خاطرنشان شویم که جملات ۳ آدرسه صرفا جملات تخصیصی نیستند. در مورد جملات فراخوانی کد سه آدرسه حاصل مشابه با کد اسمبلی است اما در مورد جملات if و یا به عبارتی go to شرطی از دستور العمل CMP استفاده می گردد. معمولا حاصل مقایسه در داخل ثبات حالت تاثیر گذار است. برای مثال در اسمبلی که در کتاب در نظر گرفته شده دستور العمل if (x<y) go to z به صورت

```
CMP x, y
Cj< z
```

مشخص شده است که در پردازنده های گروه x86و پنتیوم معمولا عمل مقایسه بر روی محتوی ثبات انجام می گیرد و برای این منظور ثبات مورد نیاز است.

۳,۷ تولید کد به وسیله AST (درخت خلاصه نحوی)

همان طور که در جلسه قبل مشخص شد چگونگی پیمایش درخت خلاصه نحوی می تواند در اندازه کد اسمبلی تولید شده و سرعت اجرایی آن موثر باشد. برای این منظور درخت خلاصه نحوی را تبدیل به یک درخت برچسب دار

می کنند, بدین ترتیب که برای هر گروه در داخل درخت خلاصه تعداد ثبات هایی که مورد نیاز برای محاسبه زیردرختی که ریشه اش آن گره است می باشد مشخص می کند. اما اگر یک عملگر ۲ عملوند داشته باشد همیشه فرض بر این است که عملوند سمت چپ می بایست در داخل یک ثبات نگهداری شود. به عنوان مثال Add Ax , B در اینجا عملوند سمت چپ برای عملگر + در داخل یک ثبات است و برخی از اسمبلی ها این مساله را به عنوان یک محدودیت مطرح می کنند.

عمل برچسب گذاری از پایین درخت خلاصه به سمت ریشه انجام می گیرد و قبل از اینکه یک گره میانی بتوان تعداد ثبات های مورد نیاز و در نتیجه برچسب آن را مشخص کرد می بایست کلیه فرزندانش برچسب هایشان مشخص باشد. حال در داخل درخت می توان گفت برای درخت باینری فرزند سمت چپ اگر یک برگ باشد برچسبش ۱ و برگ سمت راستی برچسبش ۰ می باشد. برای گره میانی مثل n از رابطه ی زیر می توان تعداد ثبات های مورد نیاز را مشخص کرد:

$$label(n)= \begin{cases} max(l_1,l_2) & if\ l_1 \neq l_2 \\ l_1 + 1 & if\ l_1 = l_2 \end{cases}$$

همان گونه که توضیح داده شد ، نحوه پیمایش درخت خلاصه نحوی می تواند در میزان استفاده از ثبات ها و به کارگیری متغیرهای موقتی و در نتیجه اندازه و سرعت اجرایی کد اسمبلی حاصل مؤثر واقع گردد.

برچسب گذاری درخت خلاصه نحوی که در قالب DAG نیز جهت فاکتور گیری از بخشهای تکراری نمایش داده می شود می توان با تعیین تعداد ثبات های لازم جهت محاسبه هر زیر درخت, مشخص کرد که ترتیب پیمایش باید به چه ترتیبی باشد که بتوان به ایده آل فوق الذکر رسید. (حداقل استفاده از ثبات ها) (مرتب سازی درخت بر اساس label و بعد پیمایش postorder) بنابراین می بایست در صورت امکان بر اساس برچسب ها هر ردیف از درخت را مرتب و سپس در ضمن پیمایش پسوندی عمل تولید کد را انجام داد. البته باید توجه کنید که در برخی از موارد که ترتیب اجرای عملوند ها حائز اهمیت است این عمل امکان پذیر نیست.

در مورد درخت باینری در کتاب Aho قدیم در صفحه ۵۶۱ کتاب الگوریتم برچسب گذاری مطرح و با تابع زیر مشخص شده است:

$$label(n)= \begin{cases} max(l_1,l_2) & if\ l_1 \neq l_2 \\ l_1 + 1 & if\ l_1 = l_2 \end{cases}$$

اما در حالت کلی عملگرها n تایی هستند که n می تواند هر عدد قابل قبول صحیحی باشد. الگوریتم در حالت کلی در صفحه ۵۶۲ کتاب شکل ۲۳-۹ نشان داده شده است.

```
if n is a leaf then
        if n is leftmost child of its parent then
                label(n):= 1
        else    label(n):= 0
```

else begin */* n is an interior node */*

let $n_1, n_2, ..., n_l$ be the child of ordered by label

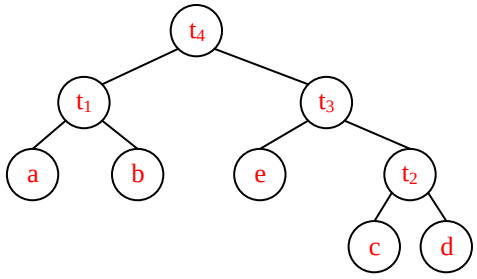
so $label(n_1) \geq label(n_2) \geq .... \geq label(n_l)$;

$label(n) := max_{1<i<l} (label(n_i) + i-1)$

end

فرض کنیم که مقادیر L_iها برای i بین ۱ تا k با یکدیگر مساوی نباشند در این صورت حداکثر تعداد ثبات مورد نیاز L_1 خواهد بود. حالا برای اثبات این مساله می گوییم $l_2<l_1$ بنابراین l_2 میتواند حداکثر مساوی $L_1 - 1$ وL_i می تواند حداکثر مساوی $L_1 - (i-1)$ باشد. بنابراین زمانی که L_2 را اجرا می کنیم یک ثبات حاصل را نگهداری کرده, L_2 ثبات هم به طور همزمان برای اجرای n_2 نیاز است بنابراین L_2+1 یعنی به تعداد L_1 ثبات مورد نیاز خواهد بود. زمانی که n_i را اجرا می کنیم i-1 ثبات مقادیر قبلی را نگهداری کرده اند وL_i ثبات هم برای ا جرای n_i مورد نیاز است. پس در جمع L_1+i-1 یعنی به تعدادL_i ثبات نیاز خواهد بود. حالا فرض کنید به این صورت نباشد و برای مثال L_i مساوی با L_i+1 باشد. در این صورت هنگامی کهL_i را انجام می دهیم به تعداد L_i+i-1 مساوی با L_1 ثبات نیاز خواهد بود و هنگامی که L_i+1 را انجام می دهیم به تعداد$L_{i+1}+i$ مساوی با $L_i+ i$ مساوی با L_1+1 ثبات نیاز خواهد بود. (اثبات به همین ترتیب ادامه دارد)

نمونه ای در شکل ۲۴ -۹ ارائه شده است.



درخت برچسب دار

۳,۸ تولید کُد از درخت برچسبدار

در اینجا بخش تابع خود بازگشتی[۱] به نام grencode (n) ارائه شده است. این الگوریتم از دو پشته استفاده می کند. یکی پشته Rstack که در ابتدا اسامی ثبات های R_0, R_1 و...و R_{2-1} را در خود نگهداری می کند. پشته ی دیگر به نام Fstack با طول نامتناهی وجود دارد که اسامی متغیرهای موقتی را در خود نگهداری می کند.

---

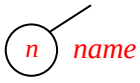
۱ - recursive

علاوه بر توابع push و pop که نام Rstack و Tstack را به عنوان پارامتر می گیرند, تابع Swap محتوای دو خانه بالای پشته ها را با هم عوض می کند.

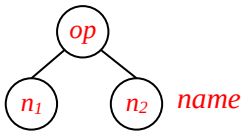
این تابع همان طور که توضیح داده شد از دو پشته ی Rstack و Tstack جهت تولید کد اسمبلی استفاده می کند. بدین ترتیب که در ضمن پیمایش درخت برچسب دار با مواجه شدن با هرگز n تابع gencode (n) مورد فراخوانی قرار می گیرد تا برای آن گره کد اسمبلی ایجاد گردد. در اینجا در داخل gencode وابسته به موقعیت گروه n , ۵ حالت مختلف مطرح می شود تا بتوان برای عبارات کد اسمبلی تولید کرد. البته این تابع را می توان به سادگی برای هر نوع جمله ای توسعه داد. در اینجا تابع gencode آورده شده.

```
procedure gencode (n);
begin
/* case 0 */
if n is a left leaf representing operand name and n is
                the leftmost child of its parent then
        print 'MOV' + name + ' , ' + top(rstack)
else if n is an Interior node with operator op, left child n1
              and right child n2 then
/* case 1 */
     if label (n2) = 0 then begin
           let name be the operand represented by n 2 ;
          gencode(n 1);
          print op + name + ', ' + top (rstack)
     end
/* case 2 */
       else if 1 <= Iabel (n1) < label (n2) and label(n1) < r then begin
           Swap ( rstack;)
          gencode (n2) ;
          R : = pop(rstack); /* n2 was evaluated into register R */
          Gencde(n1);
          print op + R + ‘,’ + top(rstack)
          push(rstack, R ) ;
          swap ( r stack)
      end
/* case 3 */
         else if 1 <= label(n2) <= label(nl) and label(n2) <  r then begin
              gencode( n1) ;
              R := pop(rstack);  /* n1 was evaluated into register R */
              Gencode( n2 ) ;
              print op + top(rstack) + ‘,’+R
              push(rstack,R )
         end
/* case 4, both labels >= r the total number of registers */
        else begin
             gencode(n2);
            T := pop(rstack);
             print 'MOV' + top (rstack) + ', ' + T;
             gencode( n1);
             push(t stack, T);
             print op + T + ‘,’ + top (rstack)
       end
end
```

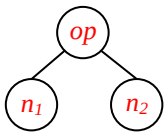
حالت ۱: که در داخل الگوریتم با case 0 مشخص شده گره n سمت چپ ترین فرزند پدر خود می باشد.از آنجایی که بر طبق ساختار اسمبلی مورد استفاده سمت چپ ترین عملگر می بایست در داخل یک ثبات قرار بگیرد دستور العمل "Mov" + top (Rstack) + ','+name قرار می گیرد.



حالت ۲: یا case 1 در داخل الگوریتم گره n یک عملگر op است با دو عملوند n_1 و n_2 و فرض بر این است که n_2 محتوایش یک name می باشد. (یک عملوند است).در اینجا در واقع label (n_2) مساوی با صفر است و تابع gen code همیشه فرض بر این است که نتیجه عملیات را در بالای پشته Rstack قرار می دهد.

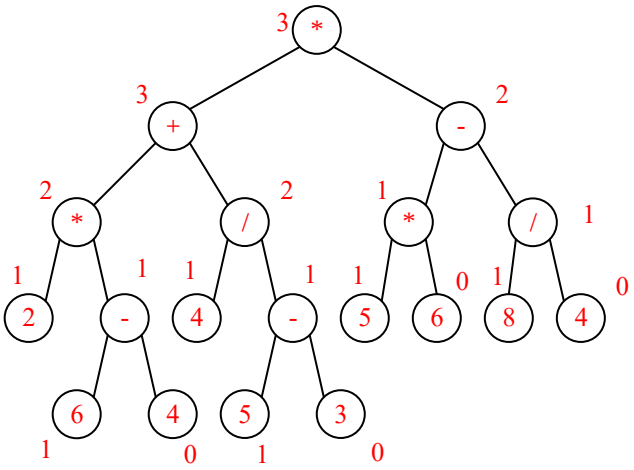


حالت ۳ و ۴ :به ترتیب یا n_1 از n_2 تعداد ثبات کمتر می خواهد و یا بالعکس. در هر حالت تعد کوچکتر از r یعنی تعداد ثبات های موجود است. باز هم در این دو حالت بدون نیاز به متغیر موقتی با در نظر گرفتن اینکه ابتدا کد میبایست برای زیردرختی حاصل گردد که تعداد ثبات های مورد نیازش بیشتر از دیگری است ابتدا برای n_1 و n_2 با فراخوانی gencode تولید کد می شود و سپس عملگر مورد نظر یعنی op بر روی ثبات هایی که حاصل n_1 و n_2 را نگهداری می کنند عمل می شود. توجه کنید که در اینجا تعداد باقیمانده ثبات هاست. عمل پیمایش در واقع با این فراخوانی های خود بازگشتی در داخل gencode انجام می گیرد .



حالت ۵: در آخرین حالت هر دو زیر درخت تعداد ثباتهای مورد نیازشان بزرگتر یا مساوی با تعداد ثبات های باقیمانده یعنی r کوچک می باشد که (حالت ۵ یا case4) در این صورت از متغیرهای موقتی و در واقع Tstack استفاده می شود.

برای نمونه به مثال زیر توجه کنید(فرض بر این است که تعداد ثبات ها در آغاز کار r = 2 است.):




```
Gencode (*);    //case4 , r=2
   Gencode (-);    //case 3 , r=2
      Gencode (*);   //case 1 , r=2
      Gencode (5);  // case 0 , r=2
      Mov R0,5
      Mol R0,6
   R=R0            //R=1
   Gencode (/);   //case1
      Gencode (8);
      Mov R1,8
      Div R1,4;
      Sub R0,R1;
      Push R0;
Gencode (+)  //case 4
   Gencode (/)
   Gencode (4)
   Gencode (-)
   Push R0    //R=2

Gencode (*)    //case 3
   Gencode (2);    //R=1
   Mov R0,2;
   Gencode (-);
   Gencode (6);
   Mov R1,6
  Sub R1,4;
  Mul R0,R1
  Add R0,t1
   Mul R0,t0
```

۳,۹ زمانبندی دستورات و تخصیص ثبات

مشکل ترین کار برای انجام محاسبات سنگین[۱] در کامپیوترها ، موازی سازی در سطح دستورالعمل‌ها است. با هر پیشرفتی درمعماری کامپیوترها، اشتراک و همکاری بین معماری و کامپایلرها خیلی آشکار می شود که نشان دهنده چالش جدی درعرصه کامپایل برنامه هاست.

این چالش‌ها پیچیدگی‌های جدیدی را معرفی می کنند که نشان دهنده موانعی است که برای به روز نمایی کامپایلرهای موجود سازگار با معماری‌های قبلی سر راه ما قرار دارند.

تمرکز اصلی دربهینه سازی کامپیلرها برای معماری ها، پشتیبانی از موازی سازی در سطح دستور عمل با سازماندهی دوباره (rearrangement) کد سطح پایین است. که با هدف افزایش توازی سازی می باشد.

خط لوله‌گی یک تکنیک معمول برای افزایش بهره وری در کامپیوترها است. خط لوله‌گی بهروری را با استفاده از با همپوش کردن اجرای دستورات بالا می برد. به طور معمول هدف در خط لوله‌های با مراحل زیاد، رسیدن به توان عملیاتی بالاتر است. وجود پرش ها و وابستگی داده‌ای بین دستورالعمل‌ها اثر بدی روی خط لوله‌گی می‌گذارد.گاهی روش‌های سخت افزاری برای حل این مشکل استفاده می‌شوند که به دلیل گران بودن و پیچیدگی آن که باعث کاهش نرخ کلاک می شود، بطور گسترده مورد استفاده قرار نمی‌گیرند.

از طرف دیگر زمانبندی که یک روش نرم افزاری است و توالی کد در حین کامپایل برنامه دوباره سازماندهی می کند که در نتیجه زمان اجرای آن کاهش می‌یابد. نشان داده شده این کار در بهبود بهروری پردازش گرهای خط لوله گی مؤثر است. زمانبندی برای ایجاد توازی در بلاک اولیه استفاده می شود و ترتیب اجرای چندین دستورعمل که مستقل هستند را در هم می‌آمیزد. در نتیجه تأخیر ناشی از رجوع به حافظه و اجرای آن ها کاهش می‌یابد. اما توازی سازی در بلاک اولیه به دو یا سه فاکتور محدود می شود و سعی می‌شود با تکنیک loop unrolling و trace scheduling بلاک‌های بزرگی تولید شود. معمولاً بلاک بزرگ در برنامه های علمی که برای سوپر کامپیوتر ها نوشته می‌شوند، وجود دارد و توازی سازی در آنها بهتر جواب می دهد.

---

[۱] Computing Power

۳,۹,۱ انواع مخاطرات[۱]

۳,۹,۱,۱ مخاطره داده‌ای[۲]

اغلب زمانبندی دستورات روی یک بلاک اولیه[۳]انجام می شود. به منظور تشخیص اینکه سازماندهی دستورات بلاک ، تغییر در عملکرد آن بوجود نمی آورد. ما به گراف وابستگی داده[۴]نیاز داریم. سه نوع وابستگی وجود دارد که باعث بوجود آمدن سه نوع مخاطره داده‌ای می‌شود.

1. Read After Write (RAW)
2. WAR
3. WAW

برای مطمئن شدن اینکه(در زمان بندی) سه نوع وابستگی در نظر گرفته شده گراف وابستگی را می سازیم که یک گراف جهت دار است که هر رأس گراف یک دستور است و بین دو رأس یال وجود دارد که نشان دهنده این است که رأس دوم (دستور دوم) بعد از اولی با خاطر وجود یک وابستگی بیاید.

۳,۹,۱,۲ مخاطره کنترلی[۵]

یعنی اجرای بک دستور نیاز به اجرا شدن دستور ما قبل باشد. وجود پرش در برنامه نیز باعع مخاطره کنترلی می‌شود.

۳,۹,۱,۳ مخاطره ساختاری[۶]

کاهش خط لوله‌گی با مخاطره ساختاری یعنی کمبود واحدها عملیاتی (منابع، فایل‌هاو...) برای دستوراتی که در خط لوله همزمان قرار دارند.

۳,۹,۲ زمانبندی دستورات[۷]

زمانبندی دستورات برای بهبود موازی سازی در سطح دستورات استفاده می شود. که کارآیی ماشین را با خط لوله‌گی افزایش می‌دهد. که کد را با کدی جایگزین می کند که هیچ تفاوتی با آن در معنی

---

۱ Hazards
۲ Data Hazard
۳ Basic Block
۴ Data Dependency Graph
۵ Control Hazard
۶ Structural Hazard
۷ Instruction Scheduling

ندارد و سعی می کند که با سازماندهی مجدد دستورات از کاهش خط لوله‌گی[1]جلوگیری می کند. همچنانکه زمانبندی کد در افزایش خط لوله‌گی و پنهان کردن تأخیر دسترسی به مؤثر است ولی یک مشکل برای تخصیص دهنده ثابت بوجود می آورد. زمانبند کد فاصله زمانی بین نوشتن و خواندن بعد از نوشتن را افزایش می‌دهد. داشتن متغیرهای با طول عمر زیاد باعث افزایش تعداد متغیر می شود که همزمان زنده هستند و باعث ایجاد برخورد[2]می‌شود. این امر تعویض تخصیص ثبات [3]را افزایش می دهد. پس زمانبندی کد باعث می شود که ثبات‌های یکسان به چند متغیر اختصاص داده شوند. این مهمترین دلیلی است که محققان زمانبند دیر[4]را به زمانبند زود[5]ترجیح می دهند. ولی در حین حال تخصیص دهنده ثبات نیز ممکن است سهواً وابستگی جدیدی را با تخصیص یک ثبات به دستور عمل های غیر مرتبط بوجود آورد. زمانبند کد دیر در مقایسه با زمانبند کد زود محدودیت دارد این محدودیت در بلاک های اولیه کوچک قابل ملاحظه نیست ولی در بلاک های بزرگ میزان بهره وری قابل ملاحظه است. الگوریتم های زمانبند کد در حالت کلی دستورالعمل ها را دوباره مرتب می کنند تا زمان اجرای برنامه بهبود بخشند. مرتب سازی دوباره باید ترتیب جزئی که توسط محدودیت تقدم عملگرها بوجود می آید را حفظ کند. برای نشان دادن محدودیت تقدم عملگرها از گراف جهتدار بدون دور[6]استفاده می‌شود. یک DAG ترتیب ارزیابی را درون بلاک اولیه مشخص می کند. رئوس نشان دهنده دستورالعمل هستند و یال‌ها نشان دهنده وابستگی ترتیبی بین آن ها می‌باشد. یک یال کشیده شده از رأس A برأس B نمایانگر آن است که باید دستورعمل A قبل از B در تبالی کد زمانبندی شده اجرا شود. در مرحله تخصیص ثبات باید تعداد زیادی ثبات نمادین و تعداد کمی از ثبات های واقعی نگاشت شوند.

۳,۹,۲,۱ الگوریتم ساده زمانبندی

ساده ترین الگوریتم این است که Topological Sort را بر روی DAG مکرراً به کار ببریم که این روش به عنوان List Scheduling شناخته می شود. به این صورت عمل می کند که منبع رأس اولیه از گراف را

---

[1] Pipeline Stall
[2] Interfere
[3] Register Spilling
[4] Post pass Scheduler
[5] Pre pass Scheduler
[6] DAG (Directed Acyclic graphs)

انتخاب کرده و به زمانبند دستورات جاری اضافه می کند و آن را ازگراف حذف مـی کنـد. الگـوریتم پایان می‌پذیرد اگر گراف خالی شود.

برای رسیدن به یک زمان بندی خوب باید از کاهش خط لوله‌گی جلوگیری کرد. انتخاب دستور عمل بعدی به الگوریتم زمانبندی بستگی دارد. چندروش اکتشافی[۱] که استفاده می‌شوند.

یک زمانبندی که سازماندهی دوباره کد درون یـک بـلاک جـدای از بـاقی برنامـه را انجـام مـی دهـد زمانبند محلی[۲] گفته می‌شود و زمانبندی که دستورات را درون بلاک‌های اولیه جابجا می‌کند بـا در نظـر گرفتن تأثیرات این جابجایی‌ها زمانبند عمومی[۳] نامیده می شود.

۳,۹,۳ تخصیص ثبات[۴]

در مبحث بهینه سازی کامپایلرها، تخصیص ثبات فرآیند نگاشت کردن تعداد زیادی متغیر در برنامـه بـه تعداد کمی از ثبات های پردازنده است. هدف این است تعداد زیادی متغیر (عملوند) را در ثبات ها نگه داری کنیم تا سرعت اجرای برنامه را افزایش دهیم. تخصیص ثبات ها می تواند

۱- برای یک بلاک اولیه

۲- برای یک تابع یا رویه

۳- برای چندین تابع صورت گیرد.

در بیشتر برنامه ها نیاز است تا تعدادی زیادی داده پردازش شـود. بـر هـر حـال بیشـتر پردازنـده‌ها روی بخش های کوچکی که ثبات نامیده می شود کار می‌کنند. حتی در ماشین های که از دستور عمل های با عملوند های حافظه‌ای پشتیبانی می کند دسترسی به ثبات خیلی سریعتر از دسترسی بـه حافظـه اسـت. متغیرهای که به آن ها ثبات تخصیص داده نشد باید به ثبات بار گذاری شوند و آخر سر در متغیر ذخیره شوند.

۳,۹,۳,۱ چالش‌ها

تخصیص ثبات به متغیرهای برنامه یک مسئله NP-complete می‌باشد تعداد متغیر های یک نمونه خیلـی زیادتر از ثبات های یک پردازشگر می باشد بنابراین مقدار برخی متغیر های باید در حافظه ذخیره شود.

---

۱ Heuristics

۲ Local Scheduler

۳ Global Scheduler

۴ Register Allocation

هزینه این انتقال با انتقال متغیر های که کمتر استفاده می شوند می تواند کاهش یابد. ولی تشخیص این متغیر ها کار آسانی نیست. بعلاوه سخت افزارها و سیستم عامل در استفاده از ثبات ها محدودیت‌هایی را اعمال می کنند.

۳,۹,۳,۲ الگوریتم کلی برای تخصیص ثبات

تخصیص دهنده های مرسوم برای تخصیص ثبات از الگوریتم رنگ کردن گراف[۱] استفاده می کنند. این الگوریتم به دو قسمت تقسیم می شود.

۱- کد ماشین با این فرض که تعداد بی نهایت ثبات وجود دارد تولید می شود. در این صورت همه متغیرها می‌توانند در ثبات‌های منطقی قرار داده شوند.

۲- ثبات‌های نمادین[۲] با ثبات های فیزیکی جایگزین می شوند با کاهش هزینه تعویض تخصیص ثبات.

در این دو مرحله یک گراف تداخل[۳] ساخته می شود که رئوس آن متغیرها و یال بین دو رأس نشان دهنده این است که دو متغیر در یک زمان زنده[۴] هستند. به طور دقیق تر اگر یک متغیر در یک زمانی زنده است و دیگر هم در آن زمان تعریف شد در این صورت آن دو متداخل گفته می‌شوند. اگر بتوان گراف را با R رنگ ، رنگ آمیزی کرد. متغیر ها می‌توانند در R ثبات ذخیره شوند.

می دانیم که مسئله رنگ آمیزی گراف یک مسئله NP-complete می باشد که این مسئله توسط JONE cock مطرح شد. این مسئله در الگوریتم chaitin با قاعده (degree < R) در نظر گرفته شده. الگوریتم یک گراف معلوم G را که شامل N رأس با درجه کمتر از R را می گیرد در زیر الگوریتم ارائه می‌شود.

```
While G cannot be R-colored
  While graph G has a node N with degree less than R
    Remove N and its associated edges from G and push N on a stack S
  End While
  If the entire graph has been removed then the graph is R-colorable
    While stack S contains a node N
      Add N to graph G and assign it a color from the R colors
    End While
  Else graph G cannot be colored with R colors
    Simplify the graph G by choosing an object to spill and remove its node
N from G
```

---

۱ Graph Coloring

۲ Symbolic Register

۳ Interference Graph

۴ Live

```
    (spill nodes are chosen based on object's number of definitions and
references)
End While
```

همان طور که معلوم است پیچیدگی زمانی این الگوریتم $O(n^2)$ می باشد.

۳,۹,۳,۳ تخصیص پویش خطی ثبات[۱]

تخصیص به گراف رنگ آمیزی شده که بهینه تولید می کند ولی زمان تخصیص آن خیلی بالا است.

درحالت کامپایل ایستا[۲]زمان تخصیص با اهمیت نیست.

ولی در حالت کامپایل پویا[۳]همچون کامپایلرهای JIT تخصیص سریع ثابت ها با اهمیت است. برای حل مشکل راه حل های دیگری ارئه شده و می‌شود.

یک تکنیک کارا به نام تخصیص پویش توسط server و poletto ارئه شده که در این روش فقط به یک مرحله برای تعیین حوزه متغیرهای زنده نیاز است.

و نیز در این روش حوزه (متغیرهای) با طول عمر کم به ثبات‌ها واگذار می شوند در صورتی که حوزه متغیرهای با طول عمر زیاد بخواهد در حافظه ساکن شوند.

این روش در مقایسه روش گراف رنگ آمیزی دارای کارآیی ۱۲ درصد کمتر است.

این الگوریتم به صورت زیر است:

```
LinearScanRegisterAllocation
Active  → {}
foreach live interval i, in order of increasing start point
        ExpireOldIntervals(i)
        if length(active) =R then
            SpillAtInterval(i)
       else
          register[i] a register removed from pool of free registers
          add i to active, sorted by increasing end point
ExpireOldIntervals(i)
     foreach interval j in active, in order of increasing end point
            if endpoint[j] _ startpoint[i] then
                    return
            remove j from active
            add register[j] to pool of free registers
SpillAtInterval(i)
spill last interval in active
      if endpoint[spill] > endpoint[i] then
          register[i] register[spill]
          location[spill] new stack location
          remove spill from active
         add i to active, sorted by increasing end point
      else
        location[i] new stack location
```

[۱] Liner Scan Register Allocation

[۲] Static Compilation

[۳] Dynamic Compilation

۳,۹,۴ تداخل اهداف

هدف متعالی تخصیص دهند ثبات ها این است که طوری ثبات‌ها را به مقادیر برنامه تخصیص دهد که تعداد دسترسی به حافظه در حین اجرای برنامه را کاهش دهد. تخصیص ثبات دو سه صورت محلی و عمومی و بین رویه‌ای است بستگی دارد به اینکه چگونه این ثبات ها به مقادیر اختصاص داده شود.
اهداف زمانبندی دستورات و تخصیص حافظه ، در تولید کد با هم در تضاد می باشد که عبارتند از:

۱- واگذاری ثبات فیزیکی یکسان به ثبات های مجازی مختلف باعث ایجاد وابستگی ها می شود که در کد اصلی وجود ندارد. اما مزیت آن هم این است که متغیر های دیگر را از انتقال حافظه محفوظ نگه می دارد.

۲- جابجایی دستورات[۱] در حالیکه میتواند باعث افزایش میزان توازی در برنامه شود ممکن است باعث افزایش طول عمر برخی از دستورات نیز شود که می توانند بار اضافی را به ثبات ها تحمیل می کند.

۳- زمانبندی دستورات تعداد کافی از ثبات های محلی را طلب می کند تا از استفاده مجدد دستورات جلوگیری کند. بدلیل اینکه استفاده مجدد فرصت توازی سازی در سطح دستور عمل را کاهش می‌دهد. اما تخصیص دهنده ثبات تعداد کافی ثبات عمومی را ترجیح می دهد تا از انتقال در محدود بلاک اولیه جلوگیری کند.

۴- یک زمانبند خوب درجه توازی در سطح دستورالعمل را که به آن رسیده را از دست می‌دهد وقتی که انتقال کد[۲] بعد از آن افزایش پیدا کند.

تکنیک هایی برای ایجاد ارتباط مورد نیاز و همکاری بین زمانبند دستور محلی و تخصیص دهنده ثبات محلی و همچنین همکاری بین زمانبند دستور محلی و تخصیص دهنده سراسری و همچنین همکاری دستورات سراسری و تخصیص دهنده ثبات تولید شده است. نتایج حاصل از آزمایش این گروه ها نشان داد که طرح همکاری به راستی که کارآیی را نسبت به تولید کننده کد مرسوم که در آن تخصیص دهنده ثبات و زمانبند دستورالعمل به صورت مجزا کار می کنند را دارد.

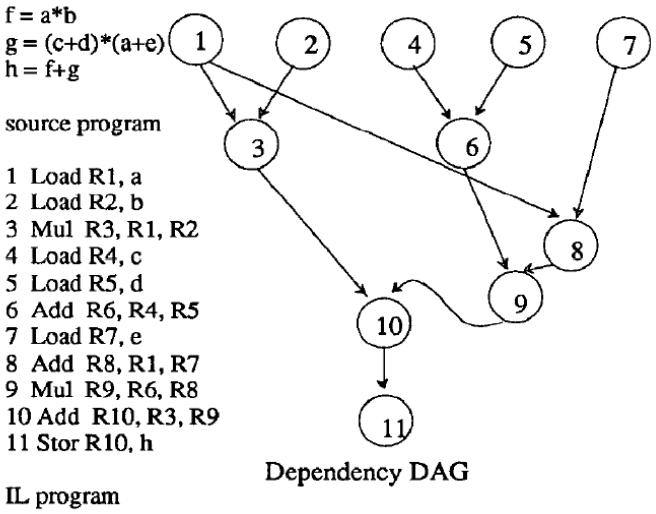
---

[۱] Instruction Reordering

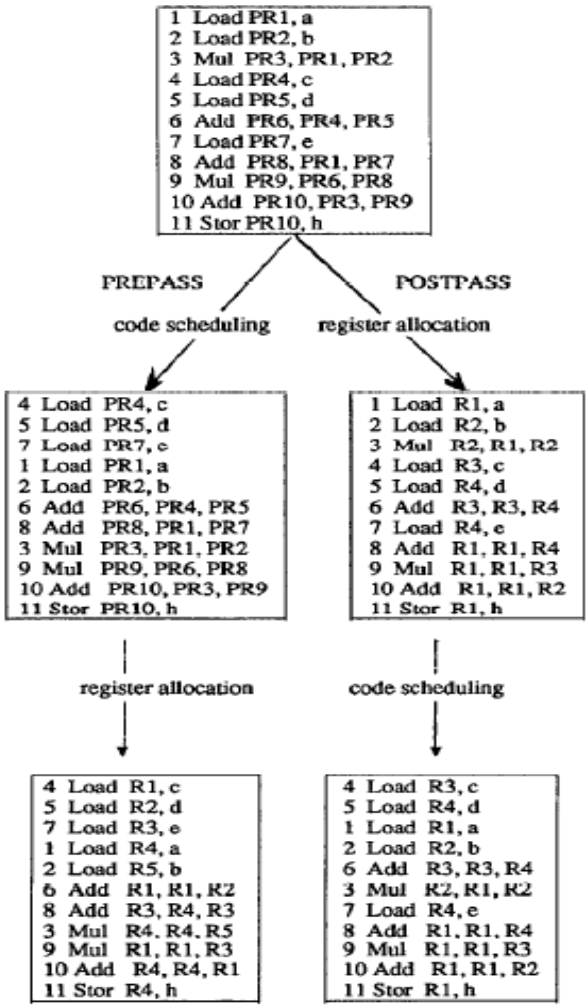
[۲] Spill Code

۳,۹,۵ زمانبند دیر یا زود؟

زمانبند کد می‌تواند برروی برنامه به صورت IL قبل از تخصیص ثبات و یا بعد از تخصیص ثبات به کار برده شود مزیت پیش زمانبند کد این است که توازی کاملی از برنامه را ممکن می‌سازد. عیب این روش این است که امکان استفاده مجدد ثبات ها موجب تعویض تخصیص ثبات پیش از اندازه شود پس زمانبند کد موجب این مشکل نمی‌شود ولی ممکن است قبلاً تخصیص دهنده ثبات یک ثبات را به چندین دستور عمل غیر مرتبط تخصیص دهد که موجب وابستگی های جدید شود در زیر فرض می کنیم که یک پشته داریم که برای مدیریت مجموعه ثبات ها از آن استفاده می‌کنیم ثبات های مرده به بالای پشته می‌آیند و ثبات های جدید از بالا اختصاص داده می‌شوند. (شکل ۲)

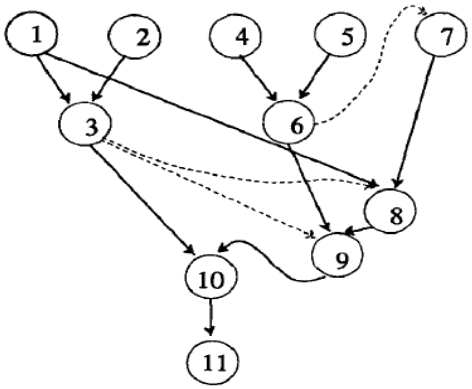


شکل ۱



شکل ۲ – prepass و postpass روی کد بالا شکل ۱

گراف شکل ۱ برای پیش زمانبند کد استفاده می شود. این DAG براساس وابستگی بین ثبات های کاذب است که باعث توازی بیشتر می‌شود. در زمانبند کد دیر بعد از تخصیص ثبات استفاده مجدد ثبات ها وابستگی های جدید را به وجود می آورد. برای مثال استفاده مجدد ثبات ۴ در دستور ۷ یک وابستگی WAR بین دستورات ۶ و ۷ ایجاد می کند (شکل ۳). این وابستگی جدید باعث می شود که دستور ۷ با دستورات ۴ و ۵ همپوش نشود. به هر حال در پیش زمانبد از ۵ ثبات استفاده شده است در حالی که در پس زمانبند از ۴ ثبات. اگر فقط ۴ ثبات در دسترس باشد پیش زمانبند نیاز دارد از دستورات load و store برای spill registers استفاده نماید.



solid lines -- original dependencies.

dashed lines -- dependencies added by register allocation

شکل ۳

۳,۹,۵,۱ دو تکنیک زمانبند متفاوت

در فاز بهینه سازی دو تکنیک سازماندهی مجدد می تواند به کار برده شود:

۱- زمانبند کدی که از تاخیر در ماشین خط لوله گی جلوگیری می کند که ما در باره آن صحبت کردیم و آن را به اختصار تکنیک (CSP(Code Scheduling for Pipelined processor می‌نامیم.

۲- زمانبند کدی که تعداد ثبات های مورد نیاز را کاهش می دهد و آن را CSR (Code Scheduling to minimize Register usage) می‌سنامیم.

CSP می تواند قبل با بعد از تخصیص ثبات به کار برده شود در حالی که به نظر می رسد CSR باید قبل از تخصیص ثبات باید به کار برده شود.

در زیر استفاده از CSR را نشان می دهیم. (شکل ۴)

```
4 Load PR4, c                          Load R1, c
5 Load PR5, d                          Load R2, d
6 Add  PR6, PR4, PR5                   Add  R1, R1, R2
1 Load PR1, a            after         Load R2, a
7 Load PR7, e        == register ==>   Load R3, e
8 Add  PR8, PR1, PR7     allocation    Add R3, R2, R3
9 Mul  PR9, PR8, PR6                   Mul  R1, R3, R1
2 Load PR2, b                          Load R3, b
3 Mul  PR3, PR1, PR2                   Mul R3, R2, R3
10 Add  PR10, PR3, PR9                 Add  R1, R3, R1
11 Stor PR10, h                        Stor R1, h
```

شکل ۴

در شکل ۲ با یک تخصیص ثبات معمولی روی کد IL به ۴ ثبات نیاز است در حالی که با استفاده از CSR

به ۳ ثبات نیاز است .

CSR و CSP با هم تداخل هدف دارند ما سعی داریم آنها را برای رسیدن به کد بهینه در یک فاز انجام دهیم.

۳,۹,۶ زمانبندی زود ترکیبی با تخصیص ثبات[۱]

عیب بزرگ زمانبندی زود این است که ممکن است استفاده زیادی از ثبات‌ها باعث تعویض تخصیص ثبات شود. در اینجا ادغام CSR و CSP در زمانبندی کد زود، برای کنترل تعویض تخصیص ثبات پیشنهاد می‌شود. ایده‌ی اصلی نگهداشتن رد تعداد ثبات‌های در دسترس طی زمانبندی کد است. از آنجا که هر دستورالعمل صادر شده ممکن است یک ثبات زنده‌ی جدید ایجاد و زمان زندگی تعدادی از ثباتها را خاتمه دهد، نگهداری تعداد ثباتهای در دسترس امکان‌پذیر است. وقتی که ثبات کافی وجود داشته باشد، زمانبند CSP را برای کاهش تاخیرهای خط لوله به کار می‌برد و وقتی کم باشد، زمانبند برای کنترل کاربرد ثباتها به CSR تغییر وضعیت می‌دهد. مثال زیر رهیافت را تشریح می‌کند.

مثال: برنامه‌ی ورودی یکسان با شکل ۱ و تعداد۴ ثبات در دسترس برای این برنامه را در نظر بگیرید. زمانبند، کد برنامه را در یک دنباله شبیه این زمانبندی می‌کند:

```
4   Load PR4, c
5   Load PRS, d
7   Load PR7, e
1   Load PR1, a
```

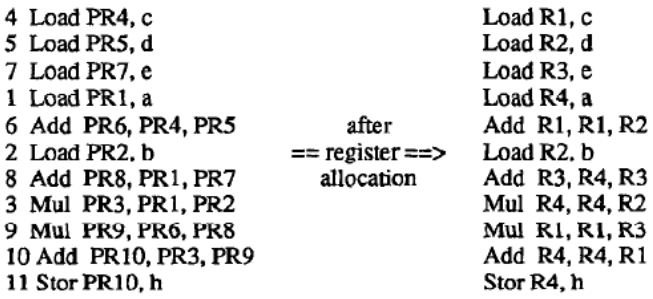
اکنون زمانبند باید بین صدور دستور۲ که یک ثبات را فعال می‌کند و دستور۶ که یک ثبات را آزاد می‌کند، انتخاب کند. از آنجا که ثباتهای در دسترس استفاده شده‌اند CSR کنترل زمانبندی را در دست می‌گیرد و دستور۶ را صادر می‌کند. سپس به CSP برمی‌گردیم و دستورالعمل۲ بعد از دستور۶ صادر می‌شود. دنباله کد سازماندهی شده‌ی کامل در شکل ۵ آمده است.

توجه کنید که این دنباله کد ۴ ثبات را به کار می‌برد، یعنی تعدادی یکسان با دنباله کد postpass. در مقایسه با دنباله کد postpass در شکل۲ ، این کد همبندیهای زمان اجرای کمتری دارد.

هدف CSR ، پیدا کردن دستورالعمل بعدی است که تعداد ثباتهای زنده را افزایش نمی‌دهد یا اگر ممکن است این تعداد را کاهش می‌دهد. CSR در مورد ترتیب ارزیابی کلی تصمیم نمی‌گیرد. رهیافت اصلی آن، پیدا کردن دستورالعملی است که ثباتهایی که آزاد می‌کند بیشتر از ثباتهای زنده‌ای باشد که

[۱] Integrated prepass scheduling

ایجاد می‌کند. اگر چنین دستورالعملی وجود نداشته باشد، زمانبند دنبال دستورالعملهایی روی مسیرهای جزئی ارزیابی شده می‌گردد که در صورت کامل شدن ارزیابی آن، ممکن است ثباتهایی آزاد شوند.



شکل ۵

سوئیچ بین CSP و CSR بر اساس تعداد ثباتهای در دسترس AVLREG انجام می‌شود. بیشتر اوقات CSP مسئول زمانبندی کد است. وقتی که AVLREG به زیر یک مقدار حدی افت می‌کند، CSR فراخوانی می‌شود. بعد از اینکه AVLREG بازیابی شده مقداری قابل پذیرش باشد CSP زمانبندی را از سر می‌گیرد. AVLREG ابتدا توسط تعداد کل ثباتها منهای تعداد ثباتهای زنده روی ورودی مقداردهی می‌شود. تحلیل جریان داده‌ی سراسری می‌تواند اطلاعات ثباتهای زنده روی ورودی را تامین کند. شمارگان ارجاع برای تعیین شبه ثباتهای مرده‌ای که می‌توانند آزاد شوند به کار می‌رود. AVLREG را زمانی افزایش می‌دهیم که ثباتهایی آزاد شوند و آن را کاهش می‌دهیم زمانی که دستورالعملها ثباتهای زنده ایجاد کنند.

۳,۹,۶,۱مجموعه leader ومجموعه ready

یک leader از یک DAG راسی است که هیچ جدی (predecessor) ندارد. یک دستورالعمل تا زمانی که leader نشده نمی‌تواند صادر شود. گره‌ی دستورالعملهایی که صادر می‌شود از DAG حذف می‌شود و تعدادی از گره‌های بعدی leader های جدید را تشکیل می‌دهند.همه‌ی leader ها در یک مجموعه‌ی leader نگهداری می‌شود. دستورالعملهای مجموعه‌ی leader فاقد همبندی با دستورالعملهای صادر شده‌ی قبلی که از مجموعه‌ی leader به مجموعه‌ی ready ارتقا یافته‌اند، می‌باشند. همه‌ی دستورالعملهای مجموعه‌ی ready آماده‌ی صدور هستند.

الگوریتم زمانبندی جامع

۱- شبه‌ثباتها را برای اجرای یک انتساب تغییر نام می‌دهد.

۲- وارد کردن یک بلاک اولیه، ایجاد DAG و محاسبه‌ی شمارگان ارجاع هر شبه‌ثبات

۳- محاسبه‌ی هزینه‌ی جمعی هر گره در ترتیب مرتب‌سازی مکانی معکوس

۴- صدور دستورالعملها در ترتیب مرتب شده‌ی مکانی

جزئیات گام ۴:

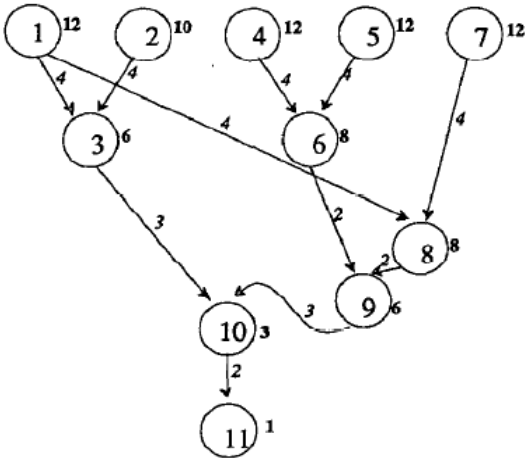
```
while (leader set or ready set is not empty) do

  4.1 Move nodes without interlocks from leader set to ready set.
  4.2 if (AVLREG > threshold value) then
        if (ready set is not empty) then
          select one node from ready set with maximum
          cumulative cost.
        else
          select one node from leader set with maximum
          cumulative cost.
        endif
      else {invoke CSR}
        if there are nodes in ready set that can free registers
        then
          select one node which frees the most registers.
          if there are more than one such node then
            select one with maximum cumulative cost.
          endif
        else
          if there are nodes in leader set that can free registers
             then
            select one which frees the most registers.
            if there are more than one such node then
              select one that has the fewest interlocks.
            endif
          else
            find a partially evaluated path, (for example,
               one of its RAW dependency has been lifted)
            select one node from the leaders of this path.
            if there are no such partially evaluated paths then
              select any one node from the ready set
              or from the leader set if the ready set is empty.
            endif
          endif
        endif
      endif

  4.3 Issue the selected instruction
    if the issued instruction creates one live register then
      decrement AVLREG by 1.
    for each pseudo-register referenced in this instruction do
      decrement its reference count by 1
      if the reference count drops to 0 then
        increment AVLREG by 1.
      endif
    end for
    Remove this instruction from the DAG
    Remove all dependencies caused by this instruction
    Reserve the destination register in a reservation table.

  4.4 Insert new leaders into the leader set

end while
```

برنامه‌ی ورودی به کار رفته در این مثال با برنامه‌ی بخش قبل یکسان است. به علاوه ما تنظیم وقت یا زمانگیری (timing) زیر را برای توابع مربوطه در نظر می‌گیریم:

Function	Timing(clock periods)

```
Load              4
Stor.              1
Add               2
Multiply        3
```

فرض می کنیم که مقدار اولیه‌ی AVLREG ، ۴ است. DAG وزن‌دار در شکل۶ نشان داده شده است.



boldface numbers associated with nodes are cumulative costs
italic numbers associated with edges are execution time estimates

شکل ۶

دنباله کد تولید شده توسط زمانبندیهای زود، دیر و الگوریتم شرح داده شده در شکل۷ نشان داده شده است. توجه کنید از آنجا که تعداد ثباتهای در دسترس ۴ است، زمانبندی کد زود موجب هزینه‌های تعویض تخصیص می‌شود.

Prepass	Postpass	Integrated
1 Load R1, a	4 Load R3, c	4 Load R2, c
4 Load R2, c	5 Load R4, d	5 Load R3, d
5 Load R3, d	1 Load R1, a	7 Load R4, e
7 Load R4, e	2 Load R2, b	1 Load R1, a
s Stor R4, temp1	6 Add R3, R3, R4	6 Add R2, R2, R3
2 Load R4, b	3 Mul R2, R1, R2	2 Load R3, b
6 Add R2, R2, R3	7 Load R4, e	8 Add R4, R1, R4
s Load R3, temp1	8 Add R1, R1, R4	3 Mul R1, R1, R3
8 Add R3, R1, R3	9 Mul R1, R1, R3	9 Mul R2, R2, R4
3 Mul R1, R1, R4	10 Add R1, R1, R2	10 Add R1, R1, R2
9 Mul R2, R2, R3	11 Stor R1, h	11 Stor R1, h
10 Add R1, R1, R2		
11 Stor R1, h		
(22 cycles)	(20 cycles)	(17 cycles)

شکل ۷

۳,۹,۷ روش ابتکاری برای حداقل استفاده از ثبات

در این روش به بررسی مشکل بهینه‌سازی تعداد ثبات‌ها[۱] برای دنباله‌ای از دستورات[۲] پرداخته‌ایم این مسئله نقش مؤثری در تولید کد کارا و بهینه دارد. صورت مسئله را به این صورت تعریف می‌کنیم:

ورودی ما یک گراف[۳] G است که بیانگر وابستگی داده‌ها (دستورات) می‌باشد. خروجی دنباله‌ای از دستورات است که از گراف G بدست می‌آیند و از نظر تعداد ثبات مورد نیاز بهینه می‌باشند.

این مسئله با تولید کد بهینه متفاوت می‌باشد. فرق اصلی این دو در این است که در روش سنتی بهینه‌سازی کد تلاش بر کوتاه کردن طول کد تولیدی است، در حالیکه در روش ما به کم کردن تعداد ثبات استفاده شده پرداخته شده است.

به عنوان مثال این مسئله در موارد زیر مطرح می‌شود:

- در سیستم‌های جاسازی شده[۴]. در صورتی که در این سیستم‌ها تعداد تعویض مدارها در حافظه پردازنده حداقل شود، مصرف انرژی کاهش پیدا می‌کند. معمولاً در این سیستم‌ها حافظه موقتی[۵] کمی وجود دارد و بنابراین احتمال پاک شدن داده‌ای که در آن بارگذاری شده است قبل از استفاد از آن زیاد است. در نتیجه در این معماری‌ها با کاهش تعداد ثبات‌های استفاده شده مصرف انرژی نیز کاهش پیدا می‌کند.
- در سیستم‌های موازی. مطالعات نشان داده که برای بالا بردن قابلیت اجرای موازی کد تولید شده در این سیستم‌ها کامپایلرها باید کدی با حداقل ثبات استفاده شده تولید کنند.

در این تحقیق یک روش ابتکاری برای حل این مشکل بیان شده است که بر اساس موارد زیر می‌باشد:

- آرایش(ساختار) سلسله دستورات[۶]: این مفهوم از مفهوم زنجیرهٔ دستورات[۷] حاصل شده است. مفهوم زنجیرهٔ دستورات قابلیت به اشتراک گذاری ثبات‌ها در یک زنجیره به هم وا بسته از دستورات را بیان می‌کرد. این در حالیست که روش سلسله دستورات به صورت دقیق‌تری به این موضوع پرداخته است که در ادامه توضیح داده خواهد شد.

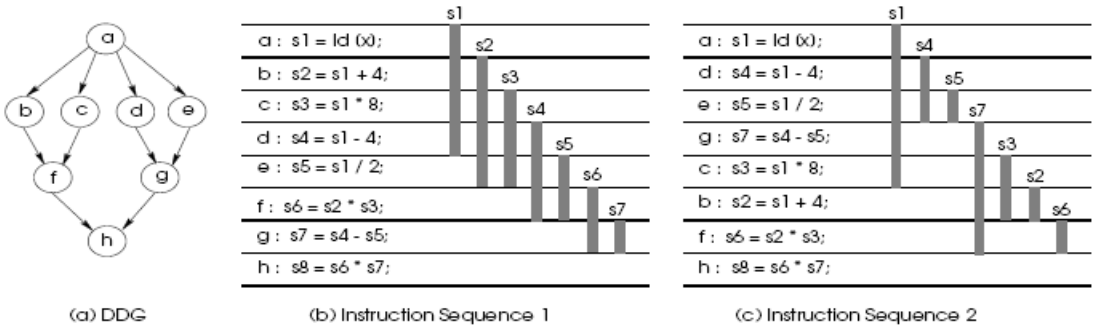
---

۱ Register
۲ Instruction
۳ Graph
۴ Embedded Systems
۵ Cache Memory
۶ Instruction Lineage Formation
۷ Instruction Chain

- مفهوم "گراف تداخل سلسله‌ها" که بیان کننده ارتباط بین سلسله‌ها می‌باشد و برای آسان کردن به اشتراک گذاری ثبات‌ها بین سلسله‌ها به کار برده می‌شود.
- مفهوم "وابستگی و پیوستگی گره‌ها" که برای مشخص کردن ترتیب و اولویت دستورات در تولید دنباله دستورات. این مفهوم میزان نزدیکی دستورات را در سلسله‌های مختلف نشان می‌دهد و باعث می‌شود تا دستورات وابسته‌تر با هم قرار گیرد. در نتیجه قابلیت به اشتراک گذاری ثبات‌ها ایجاد می‌شود و میزان استفاده از ثبات‌ها کاهش می‌یابد.

۳,۹,۷,۱ مثال عملی

در این بخش مثالی برای بیان مسئله آورده شده است. فرض کنید گراف وابستگی ورودی به صورت شکل ۱-a باشد.دو مورد از دنباله کدهای ممکن نیز در شکل آورده شده است. در دنبالهٔ اول حداکثر به ۴ متغیر به صورت همزمان مورد نیاز هستند. این اتفاق در دستورات d و e افتاده است. ولی در دنباله دوم فقط ۳ متغیر به صورت همزمان مورد نیاز هستند(زنده هستند).

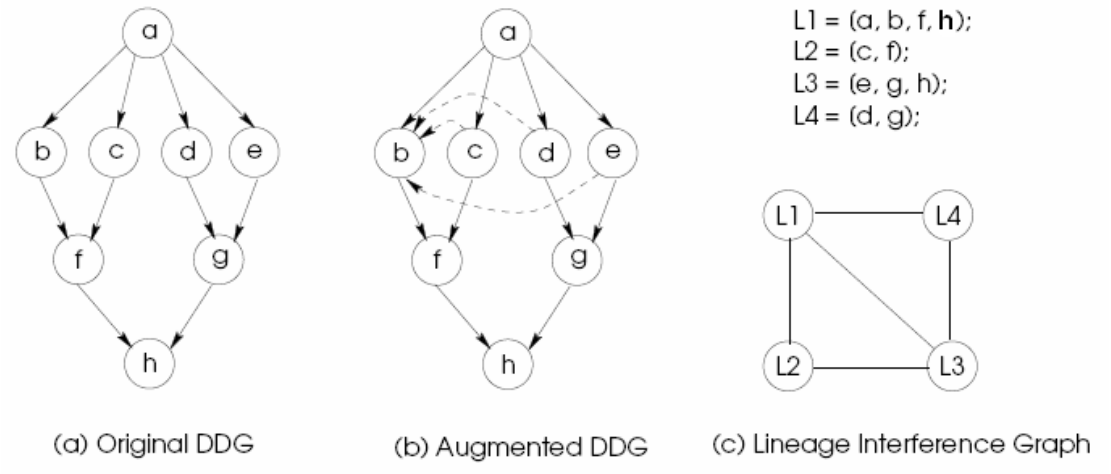


شکل ۸- مثال

برای حل مسئله فرض می‌کنیم ورودی گراف وابستگی داده است که برای یک بلاک اولیه داده شده است که باعث ترتیب نسبی بین دستورات می‌شود. همچنین این روش قابل اجرا روی superblockها هم می‌باشد.
در این گراف گره‌ها بیانگر دستورات و یال‌ها بیانگر وابستگی داده هستند. در این تحقیق تنها به گراف‌های بدون دور پرداخته شده است.
باید مشخص کنیم در یک دنباله مجاز کدام دستورات می‌توانند ثبات یکسان به اشتراک بگذارند. یک پاسخ برای این سؤال در گراف وابستگی داده‌ها موجود است. به عنوان مثال در شکل ۸.a از گره b به گره f وابستگی وجود دارد و گره دیگری به مقدار تولید شده توسط دستور b وابسته نیست، بنابراین ثبات مربوط به دستور b می‌تواند در دستور f نیز استفاده شود و به اشتراک گذاشته شود. به همین ترتیب برای بقیه گره‌ها این کار را می‌توان بررسی نمود. اما در مورد گره‌های f و g نمی‌توان این حرف را زد، چون مقادیر تولید شده توسط این دو دستور به صورت همزمان زنده هستند و نمی‌توان ثباتی بین این دو به اشتراک گذاشت.

۳,۹,۷,۲ روش حل

ما برای حل مشکل حداقل استفاده از ثبات‌ها از مفهوم سلسله دستورها استفاده کرده‌ایم که از مفهوم زنجیره دستورات مشتق شده است. اگر گره S در گراف وابستگی دارای فرزند باشد، حتماً یک مقدار تولید می‌کند و به یک ثبات نیاز دارد. اگر دنباله دستورات $S_1, S_2, S_3, \ldots, S_n$ را در گراف وابستگی داشته باشیم. که S_2 فرزند S_1 است و S_3 فرزند S_4 و به همین ترتیب برای بقیه. آنگاه می‌توانیم سلسله‌ای از دستورات را ایجاد کنیم به صورتی که تمامی دستورات سلسله بتوانند ثبات یکسانی را به اشتراک بگذارند.



شکل ۹

حال اگر S_1 بیش از یک فرزند داشت چه خواهیم کرد؟ برای اینکه S_2 از ثبات S_1 استفاده کند، باید مطمئن شویم که بقیه فرزندان S_1 قبل از S_2 در زمانبندی قرار دارند. از اینرو با محدودیت انتخاب جانشین برای ثبات استفاده شده در S_1 از میان فرزندانش مواجه هستیم و برای این کار باید شرایطی ایجاد کنیم. بعضی از این شرایط در گراف شکل b.۹ بوسیله یال‌های جهت‌دار به صورت خط چین نشان داده شده‌اند. برای مثال شکل ۹ را در نظر بگیرید. تعریف L1={a, b, f, h} بین فرزندان گره a شرایط زمانبندی را ایجاد می‌نماید. این شرایط در شکل b.۹ با خط چین مشخص شده‌اند.

کاملاً واضح است که تمامی گره‌ها در یک سلسله می‌توانند ثبات یکسانی را به اشتراک بگذارند. اما آیا دو سلسله می‌توانند ثبات یکسانی به اشتراک بگذارند؟ برای تعیین تداخل سلسله‌ها باید اشتراک داشتن بازه زنده بودن سلسله‌ها (مقادیر آنها) را چک کنیم. بازه زنده بودن یک سلسله دستور برابر مجموع (اتصال) بازه‌های زنده بودن مقادیر تعریف شده توسط دستورات آن سلسله است. اگر بازه زنده بودن دو سلسله روی هم باشد نمی‌توان بین آندو ثبات یکسانی را به اشتراک گذاشت. در غیر اینصورت اگر بازه زنده بودن آن دو در یکی از

زمانبندی‌های ممکن اشتراک زمانی نداشته باشد، می توان این سلسله‌ها را طوری زمانبندی کرد که بتوانند ثبات یکسان به اشتراک بگذارند.

۳,۹,۷,۳ روش ابتکاری برای حل مسئله

در این بخش روش ابتکاری برای حل مسئله پیدا کردن دنباله حداقل استفاده از ثبات در گراف وابستگی داده‌های بدون دور، بیان می‌شود.

۳,۹,۷,۳,۱ آرایش سلسله

ابتدا مفهوم سلسله دستورات تعریف می‌شود.

تعریف ۳,۱ سلسله دستورات، دنباله‌ای از گره های $\{v_1, v_2, \ldots, v_n\}$ است به طوریکه در گراف وابستگی داده‌ای یال‌های $(v_{n-1}, v_n), \ldots, (v_2, v_3), (v_1, v_2)$ وجود دارند. بعلاوه در سلسله بالا v_2 وارث[۱] v_1 و v_3 وارث v_2 است و به همین ترتیب.

همچنین ممکن است یک گره بیش از یک فرزند داشته باشد اما حداکثر یک وارث دارد. در این روش ما سلسله‌های بیشینه تشکیل می‌دهیم. به این صورت یک سلسله بیشینه می‌شود که اگر $\{v_1, v_2, \ldots, v_n\}$ یک سلسله باشد آنگاه هر v_n گره‌ای بدون فرزند است یا v_n متعلق به سلسله دیگری است. به این دلیل که وارث همیشه آخرین فرزندی است که زمانبندی می‌شود (در این گره آخرین استفاده از مقدار تولید شده توسط پدر وجود دارد)، در یک زمانبندی بدون دور، بازه زنده بودن گره‌ها در سلسله به هیچ وجه با هم تداخل ندارند و بنابراین تمامی گره‌ها در سلسله می‌توانند ثبات یکسانی به اشتراک بگذارند. اما به این منظور که مطمئن شویم که وارث آخرین استفاده را از مقدار تولید شده توسط پدر انجام می‌دهد، ما یال‌های زمانبندی را که از هر فرزند به وارث رسم می‌شود، در گراف وابستگی داده‌ای ایجاد می کنیم، مانند شکل b.۹.

اگر تولید یال زمانبندی باعث ایجاد دور در گراف شود دیگر زمانبندی دستوراتی که در گراف وابستگی داده‌ای آورده شده‌اند امکانپذیر نمی‌باشد. بنابراین وارث باید به دقت انتخاب شود. هنگام آرایش سلسله‌های دستورات برای انتخاب وارث به هر گره ارتفاعی داده می‌شود. ارتفاع گره به صورت زیر محاسبه می‌شود:

$$ht(u) = \begin{cases} 1 & \text{if } u \text{ has no descendents} \\ 1 + \max_v(ht(v)) & \text{for all descendents } v \text{ of } u \end{cases}$$

در گراف وابستگی شکل ۹، ارتفاع گره‌ها به این صورت است:

$$ht(a) = 4;\ ht(b) = 3;\ ht(c) = 3;\ ht(d) = 3;\ ht(e) = 3;\ ht(f) = 2;\ ht(g) = 2;\ ht(h) = 1;$$

زمان آرایش سلسله اگر یک گره v_i چند فرزند داشته باشد، فرزند با ارتفاع کمتر انتخاب می‌شود.اگر چند فرزند ارتفاع یکسانی داشتند، یک فرزند به طور تصادفی انتخاب می‌شود. هنگام آرایش سلسله برای اینکه مطمئن شویم دوری بوجود نمی‌آید، ارتفاع هر گره بعد از تولید یال‌های زمانبندی محاسبه می‌شود.

---

[۱] Heir

هر یال (u, v) در گراف وابستگی بیانگر یک وابستگی داده‌ای بین گره‌های u و v است، بنابراین هر یال وابستگی با یک ثبات معدل است. در تشکیل سلسله‌ها هدف پوشش دادن تمامی یال‌ها در گراف وابستگی است. هر گره با چند فرزند دارای چند یال خارج شونده است که هر یک از آنها به یک ثبات جداگانه مربوط می‌شود. بنابراین بهتر است هنگام تشکیل سلسله تنها یکی از این یال‌ها پوشش داده شوند و بهتر است آن یال، یال آخرین استفاده باشد. خوشبختانه، همینطور که یال‌های زمانبندی هنگام تشکیل سلسله تولید می‌شوند، یال بین گره و وارثش همان یال آخرین استفاده است. و تنها لازم است در آرایش سلسله این یال‌ها (وابستگی‌ها) پوشش داده شوند. بنابراین هر یال آخرین استفاده باید فقط در یک سلسله باشد. در تشکیل سلسله، هر سلسله بیشینه می‌شود که منجر می‌شود تا تعداد سلسله کمتری برای پوشش گراف وابستگی لازم باشد. این مسئله باعث می‌شود تا اندازه گراف تداخل سلسله‌ها کاهش پیدا کند.

آخرین گره در سلسله آخرین استفاده را از ثبات انجام می‌دهد که ممکن است متعلق به یک سلسله دیگر یا یک دستور ذخیره سازی باشد. بنابراین آخرین گره ثباتش را با بقیه ثبات‌ها به اشتراک نمی‌گذارد. این خاصیت به این صورت مشخص می‌شود که سلسله به صورت $[v_1, v_2, \ldots, v_{(n-1)}, v_n]$ نشان داده می‌شود. سلسله‌هایی که وابستگی‌های که آخرین استفاده هستند را در گراف وابستگی شکل ۹ را پوشش می‌دهند عبارتند از: $[a, b, f, h)$، $[c, f)$، $[e, g, h]$ و $[d, g)$.

الگوریتم[1] تشکیل سلسله‌ها یک الگوریتم پیمایش depth-first است که وارث‌ها مشخص می‌کند و سلسله‌های بیشینه را تشکیل می‌دهد.

تولید گره‌های زمانبندی حین عمل تولید آرایش سلسله‌ها در گراف حاصل دوری ایجاد نمی‌کند. که به این صورت اثبات می‌شود:

از آنجایی که کوچکترین فرزند هر گره به عنوان وارث انتخاب می‌شود، تمامی یال‌های اضافه شده از گره‌های با ارتفاع کم به گره‌های با ارتفاع زیاد هستند. بنابراین مسیری از گره با ارتفاع کمتر به گره با ارتفاع بیشتر نمی‌تواند وجود داشته باشد. بعلاوه ارتفاع گره‌ها بعد از اضافه کردن هر وارث دوباره محاسبه می‌شوند.

۳,۹,۷,۳,۲ گراف تداخل سلسله

برای مشخص کردن اینکه آیا دو سلسله می‌توانند ثبات به اشتراک بگذارند، احتیاج به مشخص کردن تداخل زمانی بازه زنده بودن آنها است. برای مشخص کردن بازه زنده بودن سلسله از این حقیقت استفاده می‌کنیم که هر دستور در زمانبندی مکانی مشخص و یکتا به نام t دارد.

اگر اولین دستور سلسله $L = \{v_1, v_2, \ldots, v_n\}$، v_1 در مکان t_1 و آخرین دستور v_n در مکان t_n در زمانبندی دستورات باشند، آنگاه بازه زنده بودن سلسله L در t_1 شروع و در t_n به پایان می‌رسد.

بازه زنده بودن سلسله پیوسته است. بنابراین با اولین دستور سلسله در زمانبندی بازه زنده بودن آن فعال می‌شود و با آخرین گره سلسله پایان می‌پذیرد.

بازه زنده بودن دو سلسله $L_u = [u_1, u_2, \ldots, u_m)$ و $L_v = [v_1, v_2, \ldots, v_n)$ حتما تداخل دارند اگر:

---

[1] Alorithm

مسیری از u_1 به v_n و v_1 به u_m در گراف افزوده شده وابستگی (گراف وابستگی با یال‌های زمانبندی) وجود داشته باشد.

اثبات این مسئله به این شکل است که اگر مسیر جهتداری از u_1 به v_n وجود داشته باشد، u_1 باید قبل از v_n در زمانبندی آورده شود. در نتیجه $t_{u1} < t_{vn}$ و به همین صورت $t_{um} < t_{v1}$. از طرفی چون L_u یک سلسله است داریم که $t_{u1} < t_{um}$ و به همین صورت $t_{v1} < t_{vn}$. از نامساوی‌های بالا واضح است که بازه زنده بودن L_u باید بعد از t_{v1} به پایان برسد و بازه زنده بودن L_v باید بعد از t_{u1} به پایان برسد. پس بازه‌های زنده بودن این دو سلسله حتماً تداخل دارند.

حال گراف تداخل سلسله‌ها که گراف بدون جهت می‌باشد را می‌سازیم. رئوس سلسله‌ها هستند و سلسله‌هایی که تداخل دارند به هم متصل می‌شوند. این گراف را می‌توان با استفاده از الگوریتم رنگ کردن گراف، رنگ نمود. تعداد رنگ‌های مورد نیاز برای رنگ کردن گراف برابر است با حد تعداد ثبات‌های مورد نیاز.

۳,۹,۷,۳,۳ تولید دنباله دستورات

رنگ کردن گراف تداخل سلسله‌ها برای هر سلسله یک ثبات مشخص می‌کند. برای مثال تخصیص ثبات A را که برای مثال قبلی گفته شده است مشاهده می‌کنید:

$$A = \{(a, R_1); (b, R_1); (f, R_1); (c, R_2); (d, R_2); (e, R_3); (g, R_3)\}$$

روش ترتیب‌دهی ما بر اساس الگوریتم زمانبندی لیستی[۱] است. این روش برای تخصیص ثبات از اطلاعات رنگ کردن گراف تداخل سلسله‌ها استفاده می‌کند. روش ترتیب‌دهی گره‌ها را از یک Ready List که بر اساس ارتفاع و موجود بودن ثبات مربوط به آن گره اولویت‌بندی شده است، لیست می‌کند. الگوریتم از گراف افزوده شده وابستگی (گراف وابستگی با یال‌های زمانبندی) استفاده می‌کند. موجو بودن ثبات تنها زمانی چک می‌شود که آن گره اولین گره در سلسله باشد. در غیر این صورت پدر آن گره ثباتش را در اختیار این گره می‌گذارد.

[۱] List scheduling

```
LineageFormation(V, E)
1.  mark all nodes in the DDG as not in any lineage
2.  compute the height of every node in the DDG
3.  while there is a node not in any lineage do
4.          recompute height ← false
5.          v_i ← highest node not in lineage
6.          start a new lineage containing v_i
7.          mark v_i as in a lineage
8.          while v_i has a descendent do
9.                  v_j ← lowest descendent of v_i that is not in any lineage
10.                 for each descendent v_k ≠ v_j of v_i do
11.                     add sequencing edge from v_k to v_j
12.                 endfor
13.                 if v_i has multiple descendents
14.                     recompute height ← true
15.                 endif
16.                 add v_j to lineage
17.                 if v_j is already marked as in a lineage
18.                   break;
19.                 mark v_j as in a lineage
20.                 v_i ← v_j
21.         end while
22.         if recompute height = true
23.                 recompute the height of every node in the DDG
24.         endif
25. end while
```

شکل ۱۰: الگوریتم تشکیل سلسله

```
SCHEDULING(G', L, A, N)
1.    ReadyList ← {(v_i, R_j) such that v_i has no predecessors }
2.    RegAvailable ← {R_1, R_2, ..., R_N}
3.    while ReadyList ≠ ∅ do
4.          for each node v_i in the ReadyList in decreasing height order do
5.              if (FirstNode(v_i) = False) or (((v_i, R_j) ∈ A) and (R_j ∈ RegAvailable))
6.                  Remove R_j from RegAvailable
7.                  Remove (vi, R_j) from ReadyList
8.                  Schedule(v_i, R_j)
9.                  Add to the ReadyList all successors of v_i
                      that have all its predecessors listed
10.                 if LastNode(v_i) and (v_i, R_j) ∈ A
11.                   Return R_j to RegAvailable
12.                 endif
13.             endif
14.         endfor
15.   end while
```

شکل ۱۱: الگوریتم زمانبندی

بهینه سازی کد

شاید بتوان گفت بهینه‌سازی کامپایلر، وقت‌گیرترین مرحله کامپایلر است و اگر کامپایلر بهینه‌سازی را انجام دهد، دیگر قابلیت پیاده‌سازی بصورت one pass را ندارد. از جمله مسایلی که باید در بهینه‌سازی مورد توجه قرار گیرد اینست که بهینه‌سازی نباید معنای برنامه را عوض کند چرا که هدف اصلی در بهینه‌سازی بالابردن سرعت برنامه‌ها تا حد قابل ملاحظه‌ای میباشد و اگر این افزایش سرعت باعث کارکرد غلط برنامه شود، نتنها مزیت نیست بلکه عیب محسوب می‌شود ضمن آنکه بهینه‌سازی باید ارزش وقتی را که صرف آن می‌شود را داشته باشد فلذا در مورد برنامه‌هایی که معمولاً لازم است بارها کامپایل شده ولی دفعات اجرای آن محدود است، بهینه‌سازی پیشنهاد نمی‌شود.

بهینه سازی وابسته به تحلیل کد برنامه می باشد. تحلیل کد برنامه در واقع یک روش استخراج مفاهیم از متن برنامه است که در زمینه های مختلفی از قبیل تست برنامه ها، تولید کد موازی، مهندسی معکوس[۱] و بالاخره بهینه سازی کد و مهندسی مجدد[۲] بسیار حائز اهمیت می باشد. جهت تحلیل کد برنامه‌ها،برنامه‌ها را خلاصه‌سازی می‌کنند؛ یک روش خلاصه‌سازی، استخراج گراف جریان کنترلی میباشد. در واقع گراف جریان کنترلی،چگونگی پرش‌ها و شاخه شاخه شدن جریاناجرایی در متن برنامه‌ها را مشخص می‌کند و چگونگی کنترل جریان اجرایی رادر داخل برنامه ها مدل می نماید.در این راستا کد برنامه ها را در سطح کل برنامه و متدها، بصورت مجزا مورد بررسی قرار می دهند. در سطح کل برنامه معمولا با استخراج گراف فراخوانی[۳] گراف وابستگی کلاس[۴] وابستگی بین متدها و کلاس ها از متن برنامه مشخص می شوند. با استفاده از گراف جریان فراخوانی، گراف جریان داده‌ای، گراف وابستگی داده‌ای مشخص شده و نودهای این گراف، هر ظهور یک متغیر را مشخص میکند. بعبارت دیگرشماره دستورالعملی که متغیر در آن مقدار گرفته و شماره دستورالعملی که مقدار متغیر استفاده شده، زنجیره‌هایی به نام زنجیره‌های وابستگی داده‌ای در داخل کد میانی مشخص می‌کند. برنامه های زیادی در قالب منابع آزاد[۵] جهت استخراج گراف جریان فراخوانی[۶] و گراف وابستگی کلاس وجود دارد. مشکل در ایجاد گراف فراخوانی رخ می دهد، تشخیص فراخوانی های چندریختی[۷] است. چراکه در این نوع فراخوانی ها، مقصد فراخوانی مشخص نمی باشد. امکانات پیوند پویا[۸] در زبان ها نیز آن را بسیار پیچیده تر می کند.برای تشخیص این نوع فراخوانی ها، روش هایی به نام CHS[۹] و RTA[۱۰] مطرح می باشد که در واقع، روش هایی ایستا هستند. یعنی در زمان کامپایل و بدون نیاز به اجرای برنامه، مسئله چندریختی بودن[۱۱] را به قسمی حل می کنند. اما در سطح متد و برنامه اصلی جهت تحلیل کد برنامه، گراف جریان کنترلی[۱۲] از متن برنامه ها استخراج می

۱- reverse engineering
۲- reengineering
۳- Call graph
۴- Class dependency graph
۵- Open Source
۶- Call flow graph
۷- Polymorphic
۸- Dynamic binding
۹- Class Hierarchy Analysis
۱۰- Rapid Type Analysis
۱۱- Polymorphism
۱۲- Control flow graph

شود.وابستگی‌ها در بین جملات صرفاً بواسطه استفاده مقادیر نیست. بلکه ممکن است وابستگی بصورت کنترلی باشد یعنی یک دستورالعمل، اجرای یک دستورالعمل دیگر را کنترل نماید. لازم به ذکر است دستورالعمل‌های وابسته بصورت موازی قابل اجرا نیستند. چرا که بعنوان مثال،دستور If مشخص میکند که آیا جملات بعدی آن یعنی thenpart و یا elsepart باید به اجرا درآیند لذا بصورت موازی با هم قابل اجرا نیستند. بنابراین برای اینکه وابستگی بین جملات را مشخص کند، گراف وابستگی کنترلی و گراف وابستگی داده‌ای را با یکدیگر ادغام کرده و گرافی به نام گراف وظیفه[۲] ایجاد شده و وبا استفاده از این گراف و بکارگیری الگوریتم‌های زمانبندی گراف وظایف، برنامه‌ها را بر روی سیستم-های چند پردازنده‌ای ،سوپر کامپیوترها،مالتیکورها،کلاسترها و شبکه‌های کامپیوتری بصورت اتوماتیک توزیع میکند.

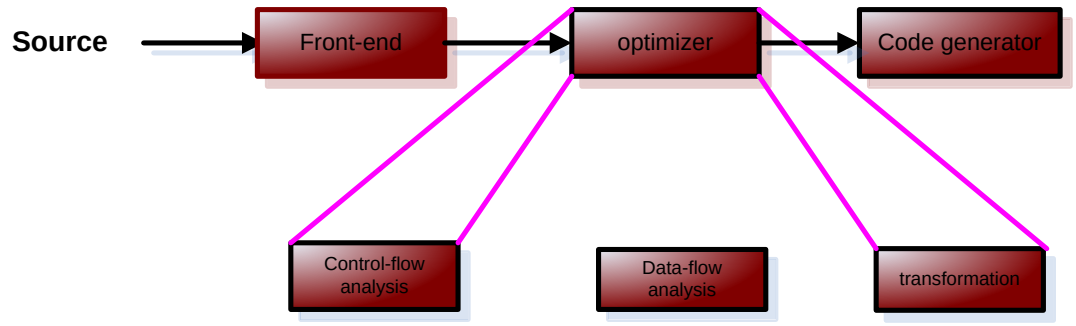
بهینه سازی کد برنامه ها شامل سه بخش است:

۱- تحلیل جریان کنترلی

۲- تحلیل جریان داده ها

۳- تکنیک های بهینه سازی

شکل زیر ساختار کلی یک بهینه‌ساز را نشان می‌دهد:



شکل : ساختار کلی یک بهینه ساز

تحلیل جریان کنترلی شامل بررسی و بهینه سازی دستورات و ساختارهای کنترل اجرایی است. تحلیل جریان داده ها، چگونگی استفاده، تغییر و نمایش داده ها را مورد بررسی قرار می دهد. همچنین برای بهینه سازی، سعی در استفاده بهینه از ثبات ها جهت دسترسی سریع به مقادیرمتغیرها دارد. تحلیل جریان داده ها و تحلیل جریان کنترلی در سطح برنامه اصلی و همچنین در زیر برنامه ها مطرح می باشند. تکنیک های بهینه سازی در دو سطح کد میانی و کد اسمبلی صورت می گیرد.

۴,۱ گراف جریان کنترلی

گراف جریان کنترلی از متن برنامه ها بخصوص دستورالعمل های سه آدرسه استخراج شده و شاخص جریان کنترل اجرایی و شاخه های آن در برنامه ها هستند. این گراف براساس کد میانی ایجاد می شود. گره های این گراف که گرافی جهت دار است را در اصطلاح بلاک اولیه[۱] می نامند.یک بلاک اولیه، دنباله ای است از دستورالعملها که

[۱] - Basic Block

بصورت متوالی بدون هیچ پرشی به اجرا در می‌آیند. لبه های گراف جریان کنترلی، تغییر جریان کنترلی و پرش از نقطه ای به نقطه دیگر را در داخل کد برنامه نمایش می دهند.

همانطورکه می دانید، وجود یک جمله If موجب می شود که جریان کنترل اجرایی برنامه به دو شاخه تقسیم شود. به این ترتیب هر گونه دستورالعمل انتقال کنترل مثل goto، حلقه های تکرار و جملات تصمیم گیری موجب می شوند که برنامه روند اجرایی خطی خود را حفظ نکند. گراف جریان کنترلی در دوسطح داخل برنامه و در بین زیر برنامه های آن ایجاد می شود. در اصطلاح گراف برنامه را گراف فراخوانی می نامند. هر جمله فراخوانی موجب یک جهش یا گذر از یک گره به گره دیگر در گراف فراخوانی می شود.

گره های گراف یا بعبارت بهتر بلاک های اولیه، دنباله ای از جملات یا دستورالعمل هایی هستند که چهار مشخصه زیر را دارند:

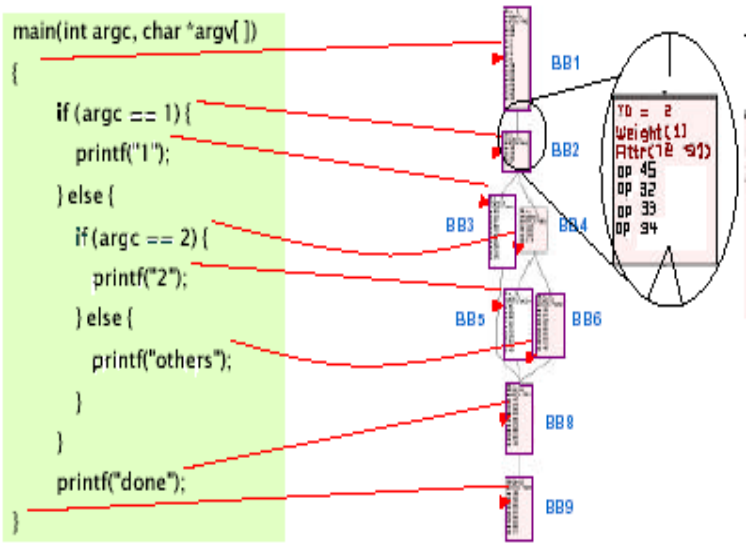
الف- بصورت متوالی بدون هیچگونه پرشی در بین جملات به اجرا درمی آیند.

ب – نقطه شروع بلاک اولیه، نقطه شروع برنامه یا مقصد goto است.

ج – آخرین جمله یک بلاک اولیه، آخرین جمله برنامه یا مبدا یک goto می باشد.

د – هیچگونه پرشی به وسط جملات وجود ندارد.

برای ایجاد گراف وابستگی کنترلی ابتدا درخت تحت تسلط هر گره باید ایجاد شود. به ابتدای هر گراف جریان کنترلی، یک گره خالی آن به نام Start اضافه نموده، یک گره خالی دیگر به نام End را در انتها آن درنظر می گیرند. بدین ترتیب برای هر گراف برنامه، یک نقطه شروع و یک نقطه خاتمه مشخص می شود.



شکل ۶: گراف جریان کنترلی بدون گره های ابتدایی و انتهایی

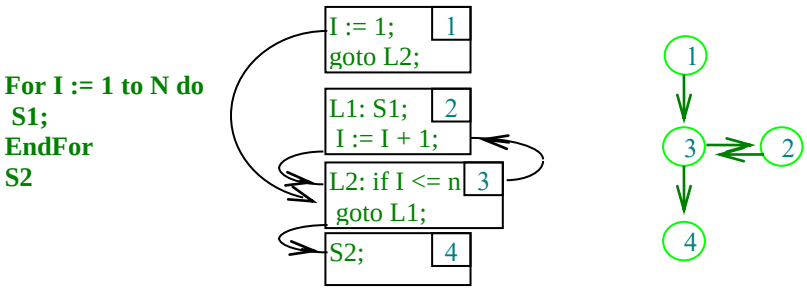
در شکل فوق، برنامه ای به همراه گراف جریان کنترلی آن مشخص شده است. البته معمولا برای سهولت در ایجاد گراف جریان کنترلی از دستورالعمل های سه آدرسه استفاده می شود. در دستورالعمل های سه آدرسه جملات بسیار ساده بوده و به سادگی می توان گره های گراف جریان کنترلی یا در اصطلاح بلاک های اولیه را تشخیص داد. البته بر

۲- task graph

۳- task graph schedualing

طبق شکل ۶ نیز می توان با کمی تلاش، الگوریتمی جهت استخراج بلاک های اولیه از متن یک برنامه ایجاد نمود. در این صورت نیازی به استفاده از دستورالعمل های سه آدرسه نخواهد بود.

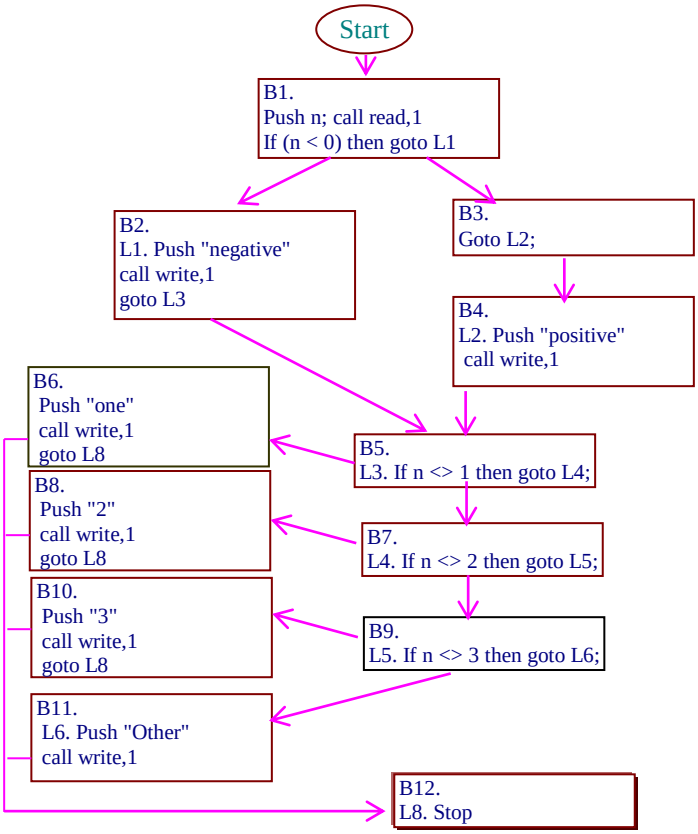
بهینه سازی بر گراف جریان کنترلی اعمال می شود. در یک گراف جریان، گره ها تحت عنوان بلاک های اولیه شناسایی شده اند. هر بلاک اولیه دنباله ای از جملاتی است که مقصد یا مبداء پرش در بین آنها وجود ندارد. آخرین جمله در یک بلاک اولیه یا یک جمله End و یا آخرین جمله برنامه است و اولین جمله یا مقصد یک goto و یا اولین جمله برنامه است. برای نمونه یک جمله ساده For فرم سه آدرسه و گراف جریان آن در شکل زیر ارائه شده است :



شکل ۷: گراف جریان کنترلی برای جمله for

برای نمونه به کد زیر توجه نمایید. البته روش صحیح این است که ابتدا کد برنامه را به کد سه آدرسه یا هر کد میانی دیگر تبدیل کرد و سپس گراف جریان کنترلی را برای کد میانی استخراج نمود.

```
Program Trivial
1. read(n);
2. if (n<0) then
3. write("negative");
 else
4. write("positive");
 endif
5. switch (n)
 case 1:
6. write("one");
 break;
 case 2:
7. write("two"); break;
 case 3:
8. write("three");
 break;
 default
9. write("other");
 endswitch;
end trivial
```



شکل ۸: گراف جریان کنترلی برای برنامه حاوی جمله case

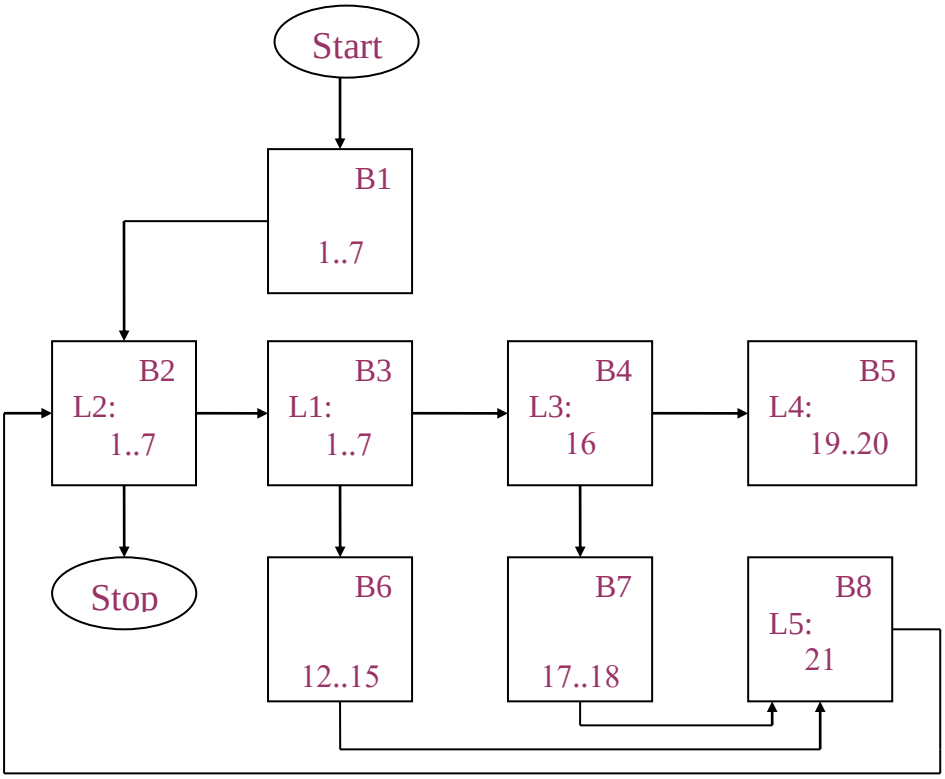
مثال:کد روبرو را در نظربگیرید.کد سه‌آدرسه آن را نوشته و گراف جریان کنترلی برایش ترسیم نمایید.

```
Read n,m;
For  i:=n to m do
Begin
  read j;
  switch j;
case 1:
        a:=a+m*n;
         j:=a-m*n;
          break;
case 2:
         j:=j-1;
           break;
 default 3:
             print j;
  end;
end.
```

حل:کد سه آدرسه کد فوق بصورت روبرو است :

```
1- push n;
2- push m;
3- call read,2;
4- pop n;
5- pop m;
6- i:=n;
7- goto L2;
8- L1:  push j;
9- call read,1;
10- pop j;
11- if j<>1 goto L2;
12- T1:=m*n;
13- a:=a+T1;
14- j:=j-1;
15- goto L5;
16- L3:
         If  j<>2 goto L4;
17- j:=j-1;
18- goto L5;
19- L4:
        Push j;
20- call  print,1;
21- L5:
           i:=i+1;
           if  i<=m  goto  L1;
22- L2:
```

اکنون با استفاده از موارد یاد شده،گراف جریان کنترلی را ایجاد می کنیم:



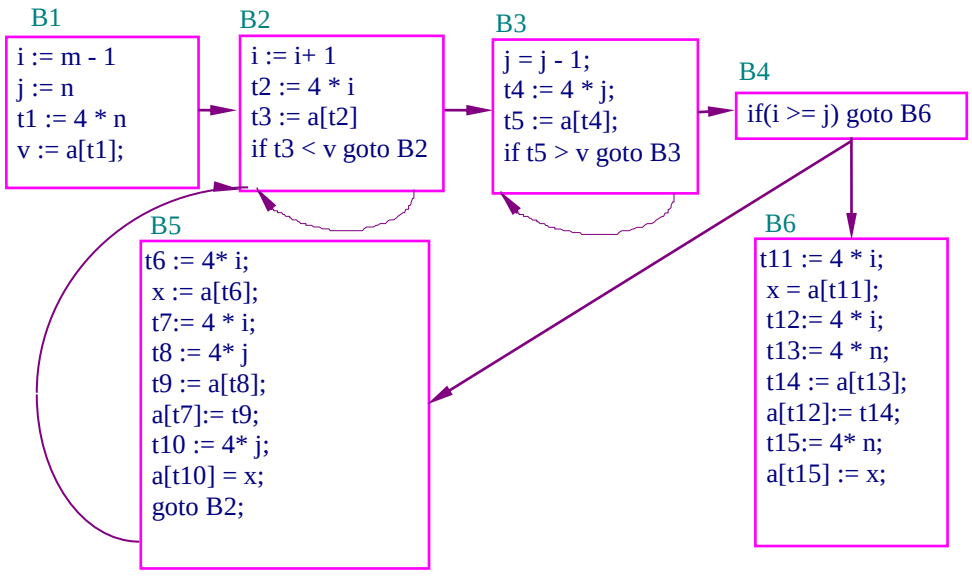
همانگونه که مشاهده می کنید، ابتدا کد برنامه به فرم سه آدرسه تبدیل شده، سپس برای کد سه آدرسه، بلاک های اولیه مشخص شده است. در ادامه تابع QuickSort مشخص شده است. جهت تولید گراف جریان کنترلی، ابتدا باید کد را تبدیل به دستورالعمل های سه آدرسه نمود.

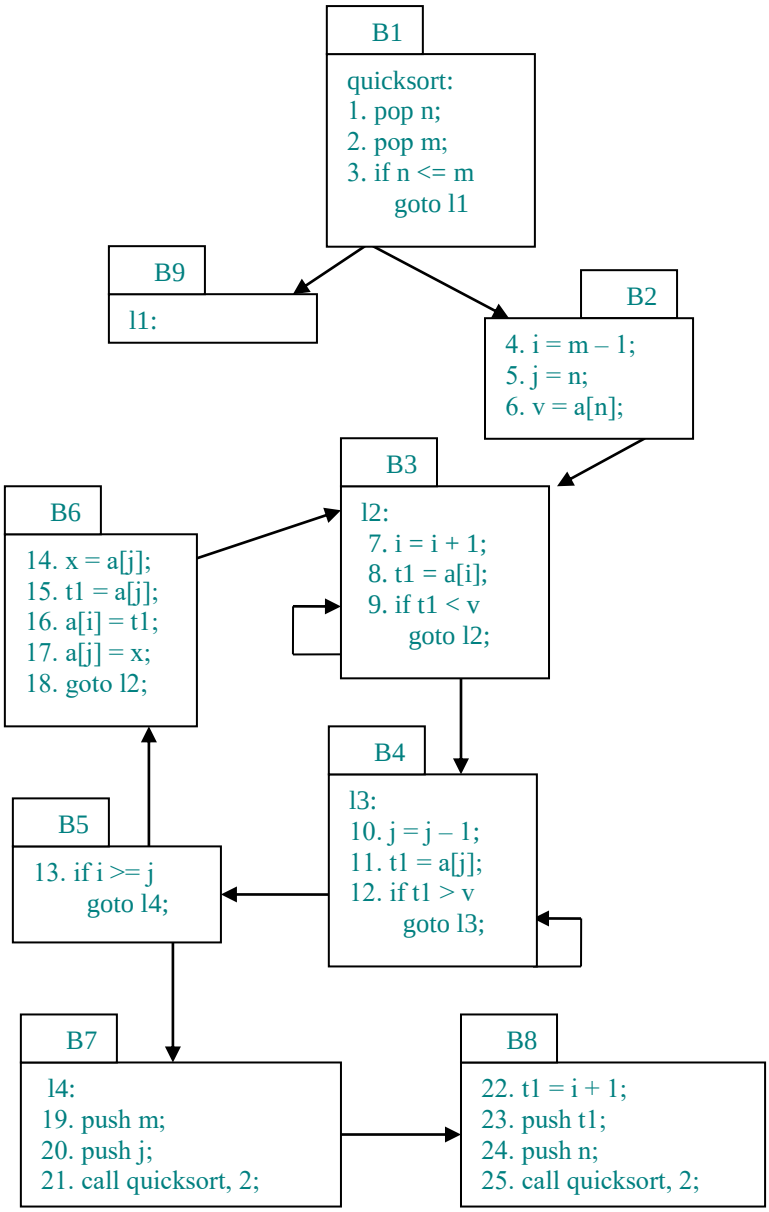
```
void quicksort(m, n)
int m, n
{
        int i, j;
        if (n <= m)
                return;
        i = m – 1;
        j = n;
        v = a[n];
        while (1)
        {
                do
                        i  = i + 1 ;
                while (a[i] < v);
                do
                        j = j – 1;
                while (a[j] > v);
                if (i >= j)
                        return a[j];
                x = a[i];
                a[i] = a[j];
                a[j] = x;
        }
        quicksort(m, j);
        quicksort(i + 1, n);
}
```

کد سه آدرسه مبتنی بر مرتب سازی کاراکتر بصورت زیر است:

```
quicksort:
        1. pop n;
        2. pop m;
        3. if n <= m goto l1;
        4. i = m – 1;
        5. j = n;
        6. v = a[n];
l2:
        7. i = i + 1;
        8. t1 = a[i];
        9. if t1 < v goto l2;
l3:
        10. j = j – 1;
        11. t1 = a[j];
        12. if t1 > v goto l3;
        13. if I >= j goto l4;
        14. x = a[j];
        15. t1 = a[j];
        16. a[i] = t1;
        17. a[j] = x;
        18. goto l2;
l4:     19. push m;
        20. push j;
        21. call quicksort, 2;
        22. t1 = i + 1;
        23. push t1;
        24. push n;
        25. call quicksort, 2;
l1:
```

سپس براساس کد میانی سه آدرسه گراف جریان کنترلی مطابق شکل های زیر تولید می شود.





شکل ۹: الف – گراف جریان کنترلی تابع QuickSort مبتنی بر عدد صحیح

ب – گراف جریان کنترلی تابع QuickSort مبتنی بر کاراکتر

بعنوان تمرین تفاوت بین این دو نوع مرتب سازی را بیابید؟

۴,۲ تکنیک های بهینه سازی

برمبنای گراف جریان کنترلی که خود نیز مبنایی جهت تحلیل کد برنامه ها است، می توان از تکنیک های بهینه سازی استفاده کرد. تکنیک های بهینه سازی، جهت کاهش حجم برنامه ها و تسریع کد اجرایی آن بکار می روند. تکنیک های بهینه سازی در دو سطح عمل می کنند، محلی و سراسری. تحلیل محلی در سطح بلاک اولیه انجام می شود. تکنیک هایی که در این بخش مورد بررسی قرار می گیرند عبارتند از:

الف – حذف عبارات مشترک[۱]

ب – انتشار کپی[۲]

ج – حذف کد زائد[۳]

د – جایگزینی مقادیر ثابت[۴]

ه – بهینه سازی حلقه ها[۵]

و – بهینه سازی بلاکهای اولیه[۶]

یکی از مواردی که در بهینه سازی کد مطرح می باشد، مقوله تحلیل جریان داده[۷]است، بدینترتیب که جریان داده ها در داخل برنامه یا تکنیک هایی مشخص می شود که ارائه خواهد شد. این تکنیک ها در مجموع تحلیل جریان داده نامیده می شوند. اصولا جهت تحلیل کد برنامه ها و بهینه سازی آن دو مقوله زیر مطرح می باشد:

الف – تحلیل جریان داده که مبتنی بر پیمایش گراف جریان داده یا گردآوری اطلاعات در مورد اتفاقاتی است که در زمان اجرا رخ می دهد.

ب – تبدیلات که براساس تحلیل های انجام شده جهت بهینه کردن کد اعمال می شود.

برای اینکه تحلیل جریان داده را انجام داد، می بایست زنجیره ای به نام زنجیره تعریف و استفاده[۸]را از کد برنامه استخراج کرد. برای این منظور مقوله تعاریف دسترسی شونده[۹]مطرح می گردد.

۴,۳ تعاریف دسترسی شونده

در بهینه سازی سراسری در سطح بین بلاک ها می بایست وابستگی داده ای بین جملات و در بین بلاک ها مشخص گردد. هنگامیکه به متغیری مقداری تخصیص داده می شود، در اصطلاح گفته می شود که آن متغیر تعریف[۱۰] شده است. سپس باید مشخص کرد که این مقدار تخصیص داده شده در کجاها استفاده[۱۱]شده است. بدین ترتیب، زنجیره ای از تعریف و استفاده های مختلف برای متغیر، در سطح برنامه بوجود می آید. درواقع، نام متغیر و مقدار متغیر را مشخص می کند.

۱- Common expretion elimination

۲- Copy propagation

۳- Dead code elimination

۴- Constant flooding

۵- Loop optimization

۶- Basic block optimization

۷- Data flow analysis

۸- Def-Use Chain

۹- Reaching definition

۱۰- Define

۱۱- Use

تعریف یاد شده در مورد متغیرهای غیرمبهم یا واضح صادق است مثل متغیر t که به آن مستقیما مقدار دهی شده باشد، اما تعریف ممکن است مبهم باشد. یعنی با نگاه کردن به کد برنامه و بدون تحلیل مفهومی، نتوان متوجه تعریف متغیر شد. برای نمونه، فرض کنید که متغیر t، متغیر عمومی[۱]است و جمله S فراخوانی به تابع می باشد. درون تابع ممکن است مقدار متغیر t تغییر کند، بدین ترتیب جمله S یک تعریف مبهم برای متغیر t می باشد.

نکته دیگری که در ابهام تعریف متغیر نقش دارد، در قالب اسامی مستعار[۲]بیان می شوند. بعنوان مثال به تعریف زیر توجه نمایید:

```
int *i;
j = &i;
*i++;
```

برای مثال در جمله a = f(b, c)، اگر توجه کنید این جمله، تعریفی مستقیم برای a بوده و می تواند تعریفی مبهم برای b و c باشد. درصورتیکه b و c از نوع پارامترهای ارجاعی[۳]باشند، مقدار آنها در f ممکن است تغییر نماید.

در ابتدا ساده ترین حالت را درنظر می گیریم. تعاریف در کد برنامه مشخص شده اند، جهت بدست آوردن زنجیره های تعریف و استفاده، باید چهار مجموعه با اسامی In، Out، Gen و Kill برای هر بلاک اولیه تعریف شوند. با استفاده از این مجموعه ها مبادرت به تشخیص تعاریف دسترسی شونده در داخل بلاک های اولیه می نماییم.

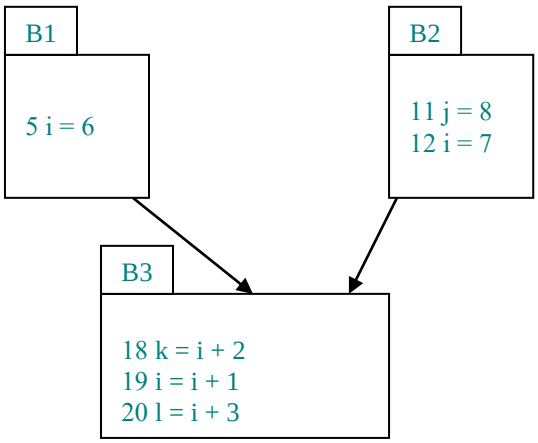
Gen = مجموعه تعاریفی که در داخل یک بلاک اولیه وجود دارد و با تعریف مجدد مواجه نشده است.

In = مجموعه تعاریفی که از خارج بلاک اولیه به آن بلاک اولیه می رسند.

Kill = مجموعه تعاریفی که به بلاک اولیه رسیده اند و بواسطه تعریف مجدد در داخل بلاک اولیه از میان رفته اند.

Out = مجموعه تعاریفی که از داخل یک بلاک اولیه خارج می شوند و به بلاک های بعدی آن می رسند. بنابراین می توان بطور خلاصه گفت:

$$In[B] = \cup Out[p] \ \forall \ p \in Predecessor(B)$$
$$Kill[B] = In[B] - Gen[B]$$
$$Out[B] = Gen[B] + (In[B] - Kill[B])$$



۱- global variable

۲- Aliasing

۳- Reference Parameter

شکل ۱۰: گرافی فرضی جهت تشخیص مجموعه های تعاریف دسترسی شونده

Gen[B3] = {(k, 18), (i, 19), (l, 20)}
In[B3] = {(i, 5), (i, 12), (j, 11)}
Kill[B3] = {(i, 5), (i, 12)}
Out[B3] ={(k, 18), (i, 19), (l, 20), (j, 11)}

برای مثال، گراف جریان کنترلی و مجموعه های Gen، Kill، In و Out قطعه کد زیر را بدست می آوریم.

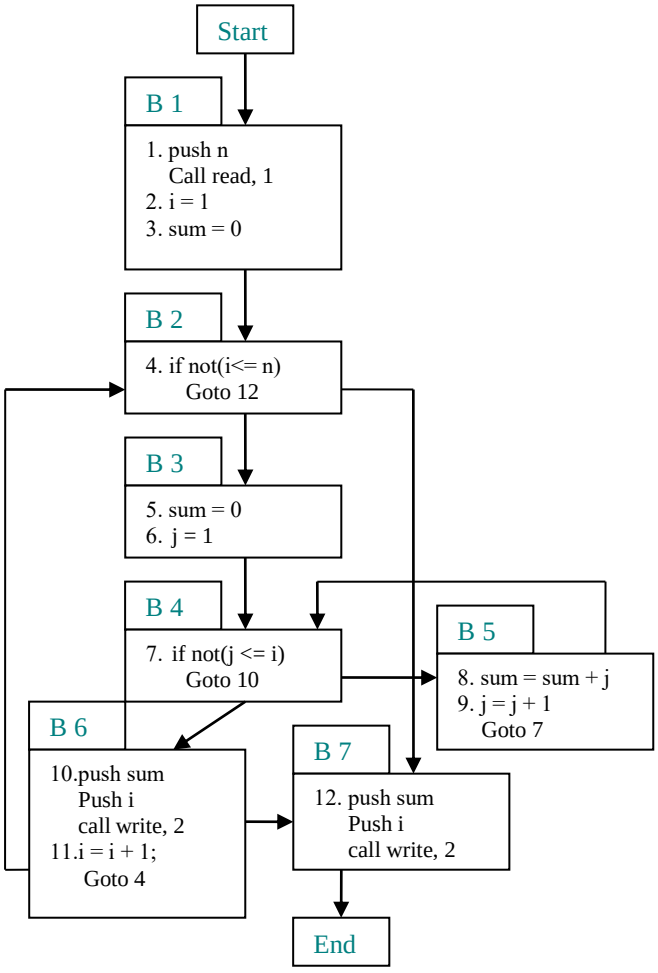
متن برنامه:

```
Program Sums
1. read(n);
2. i = 1;
3. sum = 0;
4. while (i<= n)
do
5.    sum = 0;
6.    j = 1;
7.    while (j <= i)
do
8.          sum =
sum + j;
9..         j = j + 1;
       endwhile
10..   write(sum,
i);
11..    i = i + 1;
     endwhile
12.. write(sum, i);
 end Sums
```

کد سه آدرسه:

```
1. push n
   Call read, 1
2. i = 1
3. sum = 0
4. if not(i<= n)
     Goto 12
5. sum = 0
6. j = 1
7. if not(j <= i)
     Goto 10
8. sum = sum + j
9. j = j + 1
   Goto 7
10.push sum
   Push i
   call write, 2
11.i = i + 1;
   Goto 4
12. push sum
   Push i
    call write, 2
```

گراف جریان کنترلی



شکل ۱۱: کد برنامه، کد سه آدرسه و گراف جریان کنترلی

برای اینکه الگوریتم را اجرا کنیم، در هر بلاک اولیه بصورت محلی نگاه می کنیم. کلیه تعاریفی که تا انتهای بلاک اولیه وجود دارند و تعریف مجدد[۱] نشده اند را در مجموعه Gen قرار می دهیم. به همین ترتیب مجموعه دیگری به نام

[۱] - Redefine

Kill را می سازیم، این کار را بصورت محلی و بدون اجرای الگوریتم انجام می دهیم. بدین منظور می گویم که مجموعه Kill برای بلاک اولیه مساوی با مجموعه کل تعاریفی است که برای متغیرهای مشابه مجموعه Gen در آن بلاک اولیه در بلاکهای اولیه دیگر وجود دارد. حال مهم نیست که آیا بین بلاک های اولیه مسیری وجود دارد یا خیر؟

برای مثال، اگر در بلاک اولیه ای داشته باشیم i = 5، آنگاه به مجموعه Kill در هر بلاک اولیه ی دیگری که i تعریف شده، اضافه می شود. بعبارت دیگر برای بدست آورن مجموعه Kill باید تمامی بلاک های اولیه لحاظ گردد. تعاریف دیگر برای i مرده و ازبین رفته اند.بعنوان مثال در شکل ۱۱،مشاهده می‌شود برای نمونه

$\{ In [B_2] = \cup out(p) , p= [B_1 , B_6] \}$

که $out[B_1]$ مشخص می‌باشد ولی بقیه موارد مشخص نیست. یعنی $out(B_6)$ را نمیتوان مستقیماً محاسبه کردو $out(B_6)$ تابعی از $In(B_6)$ و $out(B_2)$ است.

$Out[B_1]=\{ (n,1),(i,2),(sum ,3)\}$

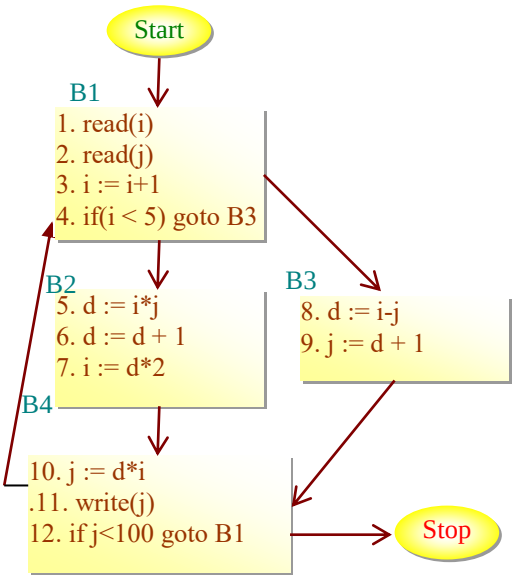
برای بدست آوردن Out و In می بایست از الگوریتم زیر استفاده نمایید.

Algorithm *Reaching Definition*

Input: Gen and kill sets for each basic block
Output: In and Out sets for each basic block

```
1. for each node n do
       OUT[n] = GEN[n]
       IN[n] = null
   endfor
2. change = true
3. while change do
4.     change = false
5.     for each node n in traverse(CFG) do
6.         IN[n] = ∪ OUT[P], where P is an immediate predecessor of n
7.         OLDOUT = OUT[n]
8.         OUT[n] =GEN[n] ∪ (IN[n] - KILL[n])
9.         if OUT[n] != OLDOUT
             then change = true
           endif
       endfor
   endwhile
```

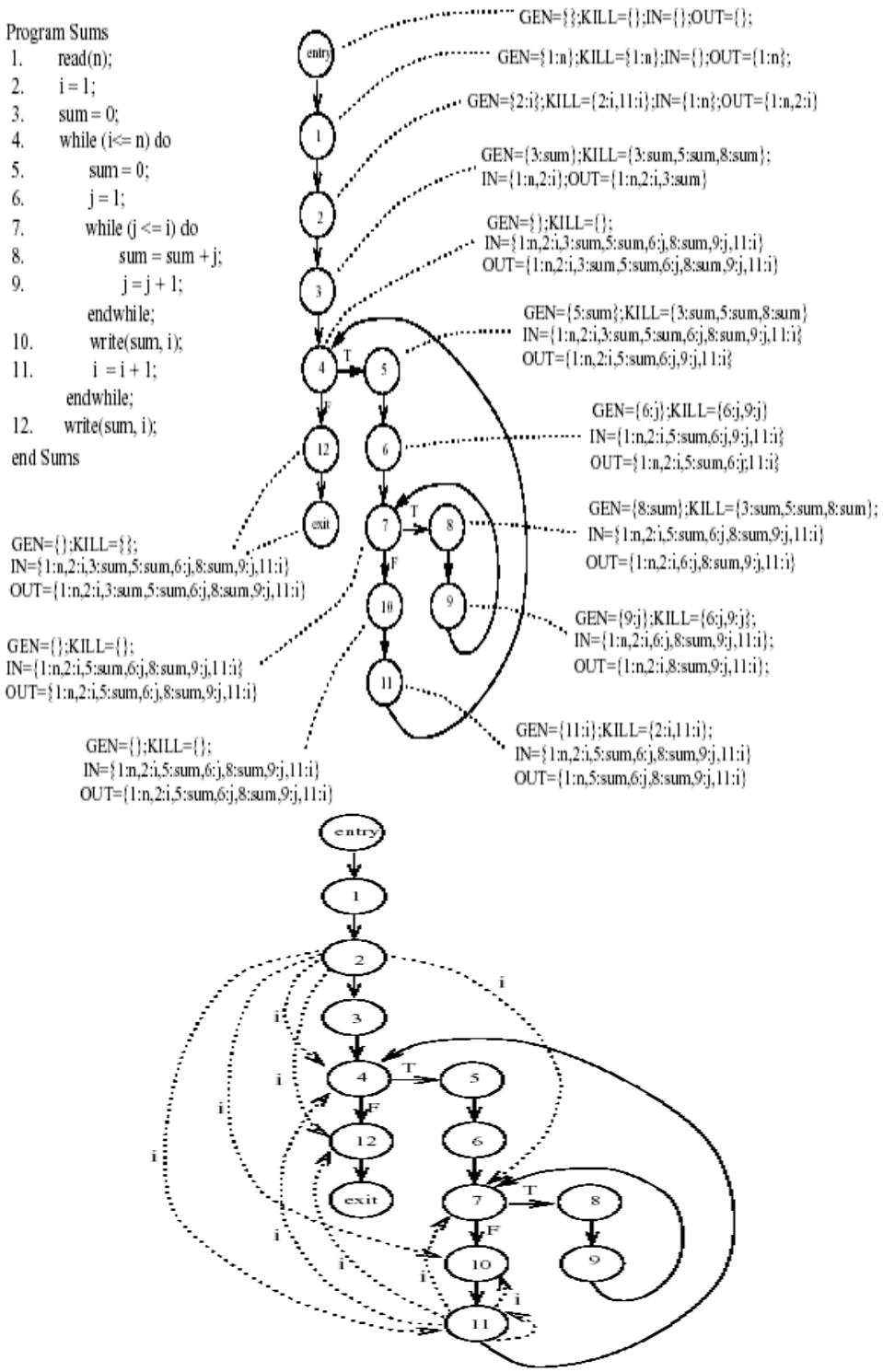
مشاهده می نمایید که برای هر بلاک اولیه مقدار Out یا Out جدید محاسبه شده با Out قبلی که OLDOut نامیده شده است مقایسه می گردد. درصورتیکه برابر نباشد، به این نتیجه می رسد که کار باید ادامه یابد. درصورتیکه مقدار Out برای کلیه بلاک های اولیه بدون تغییر باقی بماند، آنگاه کار الگوریتم به انتها می رسد. برای نمونه به گراف جریان کنترل زیر توجه نمایید.



	Gen	Kill	In	Out	In	Out	In	Out
B1	(j,2),(i,3)	(j,10),(i,7) (j,9)	(j,10)	(j,2),(i,3)	(j,10)	(j,2),(i,3)	j,10)(d,8) (d,6)(i,7) (i,3)	(j,2),(i, 3)
B2	(d,6),(i,7)	(i,3)(d,8)	(j,2)(i,3)	(d,6),(i,7)	(j,2),(i,3)	(d,6),(i,7) (j,2)	(j,2),(i,3)	
B3	(d,8),(j,9)	(j,2) (d,6) (j,10)	(j,2)(i,3)	(d,8),(j,9)	(j,2),(i,3)	(d,8),(j,9) (i,3)	(j,2),(i,3)	
B4	(j,10)	(j,2)(j,9)	(d,6),(i,7) (d,8),(j,9)	(j,10)	(d,8)(j,9) (d,6),(i,7) (j,2)(i,3)	(j,10)(d, 8) (d,6)(i,7) (i,3)	(d,8)(j,9) (d,6),(i,7) (j,2)(i,3)	

شکل ۱۲: گراف جریان کنترلی و جدول تعریف و استفاده

بدینترتیب مشاهده می نمایید که چگونه می توان برای هر بلاک اولیه تعاریف مورد استفاده را مشخص کرد. اگر توجه داشته باشید، الگوریتم فوق برای هر بلاک اولیه مشخص می کند که چه تعاریفی به بلاک اولیه می رسد و بدینصورت زنجیره های تعریف و استفاده فنا یا نابود می شوند. معمولا زنجیره ها در Symbol table ساخته می شوند. برای هر متغیر i سه زنجیره در داخل Symbol table قرار می گیرد. برای نمونه در زیر ابتدا گراف جریان کنترلی ترسیم گردیده و سپس مسیرها یا زنجیره تعریف و استفاده برای متغیر i بصورت خط چین مشخص شده است.


Program Sums
1. read(n);
2. i = 1;
3. sum = 0;
4. while (i<= n) do
5. sum = 0;
6. j = 1;
7. while (j <= i) do
8. sum = sum + j;
9. j = j + 1;
endwhile;
10. write(sum, i);
11. i = i + 1;
endwhile;
12. write(sum, i);
end Sums
entry
exit
T
F
GEN={};KILL={};IN={};OUT={};
GEN={1:n};KILL={1:n};IN={};OUT={1:n};
GEN={2:i};KILL={2:i,11:i};IN={1:n};OUT={1:n,2:i}
GEN={3:sum};KILL={3:sum,5:sum,8:sum};
IN={1:n,2:i};OUT={1:n,2:i,3:sum}
GEN={};KILL={};
IN={1:n,2:i,3:sum,5:sum,6:j,8:sum,9:j,11:i}
OUT={1:n,2:i,3:sum,5:sum,6:j,8:sum,9:j,11:i}
GEN={5:sum};KILL={3:sum,5:sum,8:sum}
IN={1:n,2:i,3:sum,5:sum,6:j,8:sum,9:j,11:i}
OUT={1:n,2:i,5:sum,6:j,9:j,11:i}
GEN={6:j};KILL={6:j,9:j}
IN={1:n,2:i,5:sum,6:j,9:j,11:i}
OUT={1:n,2:i,5:sum,6:j,11:i}
GEN={8:sum};KILL={3:sum,5:sum,8:sum};
IN={1:n,2:i,5:sum,6:j,8:sum,9:j,11:i}
OUT={1:n,2:i,6:j,8:sum,9:j,11:i}
GEN={9:j};KILL={6:j,9:j};
IN={1:n,2:i,6:j,8:sum,9:j,11:i};
OUT={1:n,2:i,8:sum,9:j,11:i};
GEN={11:i};KILL={2:i,11:i};
IN={1:n,2:i,5:sum,6:j,8:sum,9:j,11:i}
OUT={1:n,5:sum,6:j,8:sum,9:j,11:i}
GEN={};KILL={};
IN={1:n,2:i,3:sum,5:sum,6:j,8:sum,9:j,11:i}
OUT={1:n,2:i,3:sum,5:sum,6:j,8:sum,9:j,11:i}
GEN={};KILL={};
IN={1:n,2:i,5:sum,6:j,8:sum,9:j,11:i}
OUT={1:n,2:i,5:sum,6:j,8:sum,9:j,11:i}
GEN={};KILL={};
IN={1:n,2:i,5:sum,6:j,8:sum,9:j,11:i}
OUT={1:n,2:i,6:j,8:sum,9:j,11:i}
entry
exit
T
F
i

شکل ۱۳: زنجیره های تعریف و استفاده

خطوط خط چین در شکل فوق وابستگی داده ای را نشان داده، مشخص می کنند که مقدار i در کدام جملات تعریف و در کدام جملات استفاده شده است. معمولا زنجیره تعریف و استفاده جریان داده ها توسط خطوط خط چین مشخص می کند. نمونه ای از زنجیره های تعریف و استفاده برای متغیر i در شکل فوق ارائه شده است. گره ها، جملات برمامه هستند که می توان در مجموع بعنوان بلاک های اولیه آنها را دسته بندی نمود. زنجیره ها را در قالب زوج های تعریف و استفاده بصورت زیر هم مشخص می کنند.

بین گره ۲ و ۴، ۲ و ۱۲، ۲ و ۱۰، ۲ و ۷ و بالاخره ۲ و ۱۱ وابستگی داده ای وجود دارد. درواقع در جملـــه ۲ دستورالعمل i = 1 زوج i, 1 را ایجاد کرده که این زوج در جمله while با شماره ۷ در جمله ۱۲ از طریق write(sum, i) و غیره استفاده شده است. توجه داشته باشید که در جمله شماره ۱۱، i = i + 1 هم تعریف و هم استفاده از I را مشخص می کند. لذا در جمله شماره ۱۱ یک وابستگی داده ای وجود دارد. حال خود جمله ۱۱ که محل تعریف جدید I است وابستگی داده ای با جملات ۱۲، ۴، ۱۰ و بالاخره ۷ را ایجاد می کند.

اگر هیچ تعریفی برایv در این مسیر موجود نباشد، بر طبق تعریف مسیر([i: n_1], …, [n_m: j]) از گره i به گره j برای متغیر v خالی از تعریف است. برای نمونه مسیرهای ۱، ۲، ۳، ۵ و ۶ مسیری خالی از تعریف برای متغیر i هستند. مسیرهای ۷، ۸ و ۹، یک زنجیره تعریف و استفاده برای متغیر j هستند. به این ترتیب، اگر مسیر خالی از تعریف بین تعریف و استفاده از یک متغیر موجود باشد می توان مسیری را به زنجیره تعریف و استفاده متغیر افزود.

همانگونه که مشاهده خواهید کرد، برای هر سطر از برنامه مشخص شده که چه استفاده ای در آن سطر وجود دارد. همچنین تعیین می دارد که با کدام تعاریف و در کدام شماره خط ها وجود دارد. بدین ترتیب زنجیره تعریف و استفاده مشخص می شود. به طور خلاصه زنجیره تعریف و استفاده برای برنامه Sums به شرح ذیل است:

Node Deinition-Use Pairs
entry (none)
1 (none)
2 (none)
3 (none)
4 [1:n,4:n],[2:i,4:i],[11:i,4:i]
5 (none)
6 (none)
7 [6:j,7:j],[9:j,7:j],[2:i,7:i],[11:i,7:i]
8 [5:sum,8:sum],[8:sum,8:sum],[6:j,8:j],[9:j,8:j]
9 [6:j,9:j],[9:j,9:j]
10 [2:i,10:i],[11:i,10:i],[5:sum,10:sum],[8:sum,10:sum]
11 [2:i,11:i][11:i,11i]
12 [2:i,12i],[11:i,12:i],[3:sum,12:sum],[5:sum,12:sum],[8:sum,12:sum]
exit (none)

شکل ۱۴: نمونه ای از زنجیره تعریف و استفاده

وابسته به هدف دو نوع الگوریتم DU Chain و UDChain مطرح می شوند. در UD Chain مشخص می شود که یک استفاده به چه تعاریفی وابسته بوده و برای DU Chain، این نکته مشخص می شود که یک تعریف برای چه کاربردهایی وجود دارد. مسلما، اگر تعریف و استفاده ای وجود نداشته باشد، آن تعریف بدون استفاده[۱]می باشد.

در ادامه الگوریتم محاسبه زنجیره تعریف و استفاده به شرح ذیل ارایه می شود. ورودی این الگوریتم گراف جریان همراه با مجموعه In برای هر بلاک اولیه است و خروجی آن DUPair یا زوج های تعریف و استفاده است. این مشخص می کند که مقادیر مورد استفاده هر جمله در کجا تعریف شده اند. علاوه بر DUChain، UDChain هم ایجاد می کنند که ارتباط بین استفاده و تعریف را تعیین می نماید. بعبارت دیگر مشخص می کند که هر تعریفی در کجاها استفاده می شود. الگوریتم بشرح زیر است:

Algorithm *Def-Use Chain*

Input: A flow graph for which the IN sets for reaching definitions have been computed for each node *n*.
Output: DUPairs: a set of definition-use pairs.
Method: Visit each node in the control flow graph. For each node, use upwards exposed uses and reaching definitions to form definition-use pairs.

```
1. DUPairs = ϕ
2. for each node n do
3.     for each upwards exposed use U in n do
4.         for each reaching definition D in IN[n] do
5.             if D is a definition of v and U is a use of v
                 then DUPairs = DUPairs ∪ (D,U)
               endif
           endfor
       endfor
   endfor
```

شکل ۱۵: الگوریتم تعیین زنجیره تعریف و استفاده

[۱] – Dead Code

در اینجا در هر بلاک اولیه مجموعه In ها را درنظر می گیرند و با استفاده های مربوط در بلاک اولیه تبدیل به زوج می نماید. زوج را به مجموعه DUPairs اضافه می کنند. بدین ترتیب، جریان داده ها و زنجیره های مربوطه تعیین می شود. اگر مقداری به این متغیر داده شود، مشخص می گردد که در کدام نقاط استفاده شده اند. مسلما اگر مقداری بلااستفاده معرفی گردد، می توان آن را از داخل کد برنامه حذف کرد. لذا بهینه ساز با استفاده از این زنجیره ها کد زائد را از داخل برنامه حذف می کنند.

این زنجیره ها وابستگی داده ای بین جملات را مشخص می کند و در مجموعه یک گراف وابستگی برای جملات داخل برنامه ایجاد می کند. مسلما جملاتی که به یکدیگر وابسته نباشد، بصورت موازی قابل اجرا هستند. لذا زنجیره های تعریف و استفاده در ابرکامپیوترها جهت تشخیص توازی و اجرای کد مطرح هستند.

همانطور که گفته شد، مسئله اسامی مستعار کار تشخیص تعریف و استفاده را برای متغیرها دشوار می سازد. مشکل دیگر بواسطه فراخوانی توسط ارجاع بوجود می آید.

۴,۳,۱ حذف زیر عبارات مشترک(Common Subexpressions Elimination)

همانگونه که در مطالب فوق ذکر گردید، عبارات مشترک به آن دسته از عبارات اطلاق می شود که بدون اینکه پارامترهای آنها تغییری کرده باشد، بیش از یک بار استفاده گردد. با استفاده از این تکنیک عباراتی که تکراری هستند مشخص شده و صرفا یکبار محاسبه می گردند. الگوریتم زیر برای حذف عبارات مشترک ارائه می شود.

الگوریتم حذف زیرعبارات مشترک

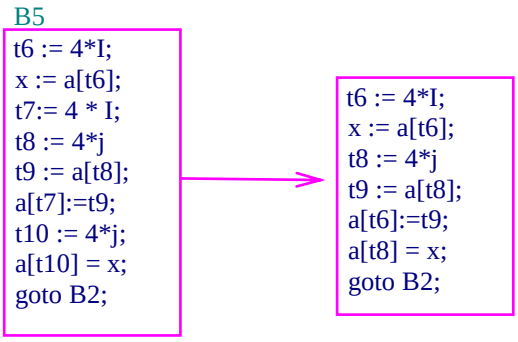
ورودی: یک گراف جریان که در آن بلاک اولیه عبارات قابل دسترس مشخص شده است.
خروجی: گراف جریان بهینه سازی شده است.
روش: برای هر جمله s در قالب x := y + z، چنانچه y + z در ورود به بلاک اولیه مربوطه قابل دسترس باشد و قبل از این جمله مقادیر y و z تغییر نکرده باشد، عملیات زیر را انجام می دهد.

۱. باید مشخص کرد که y + z در کدام بلاک های اولیه قبلی محاسبه شده اند که مقدار آنها به این بلاک های اولیه رسیده است.
۲. متغیر جدید به نام u از جنس y یا z بعنوان یک متغیر موقتی ایجاد کرده و به جدول نمادها می افزاییم.
۳. جملاتی که در مرحله ۱ پیدا شده اند در قالب w := y + z را بصورت زیر تغییر می دهیم:

```
u := y + z
w := u
```

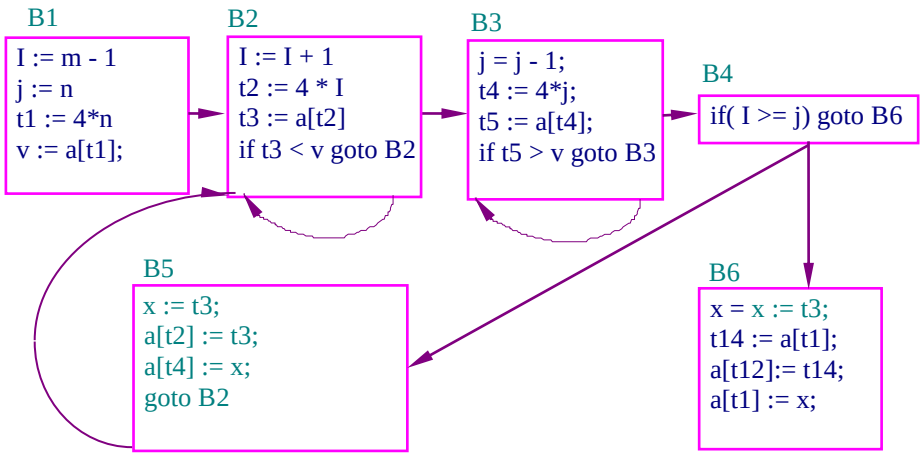
۴. جمله s را بصورت x := u تبدیل می نماید.

برای نمونه در ارتباط با کد میانی تولید شده در الگوریتم quickSort، عبارات 4 * i و 4 * j، چندین بـار و بـدون هـیچ تغییری در مقادیر i و j تکرار شده اند. می توان از محاسبه مجدد این عبارات جلـوگیری نمـود. درشـکل ۱۶، چگـونگی حذف عبارات مشترک در بلاک B5 مشخص شده است.



شکل ۱۶: حذف عبارات مشترک یا تکراری درون بلاک B5

مشکل در اینجا، ارائه الگوریتمی برای تعیین عبارات مشترک است. در بـلاک هـای B4 و B5 شـکل فـوق بـدون تغییـر مقادیر i یا j، چندین بار عبارات 4 * i و 4 * j بطورمکرر محاسبه شده اند، بنابراین می توان تکرار این عبارت را حـذف نمود. درنتیجه، گراف جریان به صورت زیر تبدیل می شود:

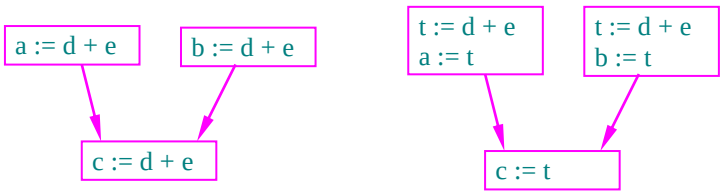


شکل ۱۷: حذف عبارات مشترک یا تکراری

۴,۳,۲ انتشار کپی (Copy Propagation)

منظور از انتشار کپی جملات تخصیصی این است که مقدار یـک متغیـر را در دیگـری کپـی نمایـد. در کامپایلرهـا سعی می شود این عمل کپی با جایگزینی مقدار مبدا بجای مقصد انجام پذیرد.

درواقع قانون کپی می گوید، در صورت امکان جمله ای مثل g := f را حذف کنید و در ادامه g را با f جایگزین نمائید. بنابراین چنانچه عبارت d + e در جملات مختلف تکرار شود، تا هنگامی که d و e تغییر نکرده اند می توان حاصل عبارت را مورد استفاده قرار داد.البته این بهینه سازی در صورتی موثر قرار می گیرد که f بعد ازg دوباره تعریف شود. نکته قابل ذکر این است که "انتشار کپی" بخودی خود برنامه را بهبود نمی‌دهد بلکه زمینه اجرای تبدیل بهینه سازی (حذف کد مرده) را فراهم می کند.برای نمونه درشکل زیر جهت حذف d + e از متغیر جدید d + e استفاده شده است. درست نیست که در اینجا عبارت c := a یا c := b را قرار دهیم، زیرا مشخص نیست که از کدام مسیر به تخصیص مقدار به c می توان رسید.



شکل ۱۸: نمونه ای از انتشار کپی

برای نمونه در بلاک B5 جمله x := t3 یک کپی برداری است. بنابراین، دنباله جملات:

```
x := t3;
a[t2] := t3;
a[t4] := x;
goto B2
```

به صورت زیر تبدیل می شوند:

```
x := t3;
a[t2] := t5;
a[t4] := t3;
goto B2
```

از انتشار کپی هدف این است که چنانچه داریم x=y حتی المقدور به جای x از y استفاده شود. این در تشخیص عبارات مشترک ممکن است مورد استفاده قرار گیرد..همچنین هر چه تعداد متغیرهای مورد استفاده در یک بلاک اولیه کمتر باشد مسلماً مسئله ی تخصیص ثبات و مدیریت ثباتها ساده تر انجام خواهد شد. نکته ی دیگر حذف جمله ی x=y می باشد . این در صورتی امکان پذیر است کع مقدار x مجدداً قبل از y تغییر نکند. برای این منظور اگر x=y به یک بلاک اولیه برسد و بلاک اولیه مقدار x را تغییر دهد ، حالا میتوان خلاف جهت در داخل بلاک اولیه به سمت بلاکی که تعریف x=y در آن قرا گرفته حرکت کردو در طول مسیر مقدار x را با y جایگزین نمود و نهایتاً جمله ی x=y را حذف کرد.

الف – برای مثال چنانچه عمل کپی در جمله P1 و محلی که از مقدار کپی شده در متغیر استفاده می کند را P2 بنامیم. اکنون P1 تعریفی از کپی بوده که برای استفاده در P2 قابل دسترسی است. بدین منظور می توان از الگوریتم Reaching Definition استفاده نمود. همچنین اطمینان حاصل کرد که برای مثال b = a در جمله p1 = 10 و p2 = 100 مقدار a را

مورد استفاده قرار داده، آنگاه می بایست تعریفی که در P2 برای b قابل دسترسی است همان تعریفی باشد کـه در P1 یـا جمله شماره ۱۰ وجود دارد. البته در این فاصله هیچ تعریفی برای a نباید مجددا وجود داشته باشد.

ب- در کلیه مسیرهای P1 و P2 نه مقدار b و نه مقدار a نباید تغییر کنند. بدین ترتیـب، مجموعـه هـای از قبـل مشـخص شده بصورت زیر خواهند بود:

۱. Gen[B]: هر یک از جملات کپی که در بلاک B وجود دارد و تا انتهای بلاک تعریف مجدد نشده باشد.

۲. Kill[B]: هر تعریفی که جمله کپی بصورت result = Variable را در سایر بلاک ها تغییر دهد و بـر آن تـاثیر گذارده باشد. توجه داشته باشید که در اینجا در بین بلاک ها بررسی می کنیم و کاری با داخل بلاک نداریم.

۳. In[B]: مجموعه کپی هایی که در ابتدای بلاک قابل دسترس می باشند و از رابطه زیر محاسبه می شوند:

$$In[B] = \cap (P \text{ a predecessor of } B)\ Out[P]$$

۴. Out[B]: مجموعه کپی هایی که از انتهای بلاک قابل دسترس باشند در این مجموعه قرار خواهند گرفت. این مجموعه از رابطه زیر محاسبه می شوند:

$$Out[B] = Gen[B] \cup (In[B] - Kill[B])$$

الگوریتم بصورت زیر می باشد:

Algorithm *Copy Propagation*

```
1. for each node n do
       GEN[n] = each copy in block
       KILL[n] = each redefine of copy in block
       OUT[n] = GEN[n]
   endfor
2. change = true
3. while change do
4.     change = false
5.     for each node n in traverse(CFG) do
6.         IN[n] = ∩ OUT[P], where P is an immediate predecessor of n
7.         OLDOUT = OUT[n]
8.         OUT[n] =GEN[n] ∪ (IN[n] - KILL[n])
9.         if OUT[n] != OLDOUT
             then change = true
           endif
       endfor
 endwhile
```

۴,۳,۳ انتشار ثابت

به زبان ساده، در مواردی که مقدار یک عبارت در زمان کامپایل قابل محاسبه باشد و برابر با یک مقدار ثابت مـی-شود، این تبدیل مقدار عبارت را محاسبه و جاگذاری می کند.

مسئله ی انتشار ثابتها که گاهی اوقات بصورت Constant Folding در قالب جایگزینی مقادیر ثابت می باشد بدین

ترتیب است که اگر برای مثال داشته باشیم

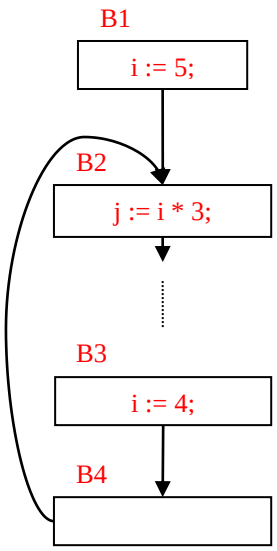
```
i=5;
j=i*2;
```

در اینجا کامپایلر به جای اینکه برای عمل ضرب کد ایجاد کند ، صرفاً دستورالعمل mov j,10 را تولید میکند.در زیر مثال دیگری برای مسئله ی انتشار ثابت مشاهده می کنید.

```
b := z + y                          b := z + y
a := 6          ⟶
x := 2 * a                          x := 2 * 6
```

شکل ۱۹: مثالی از انتشار ثابت

بنابراین مسئله ی انتشار ثابت شاید همان مسئله ی انتشار کپی باشد با این تفاوت که در اینجا مقداری ثابت به یک متغیر تخصیص داده می شود.البته جایگزینی ثابتها خود ممکن است موجب ایجاد و مقادیر ثابت جدیدتری گردد. برای نمونه در بالا با جایگزینی مقدار i ، j نیز به لیست ثابت ها اضافه می گردد.مسئله ی انتشار ثابت مسئله ی رو به جلویی است. یعنی اینکه مقادیر ثابت از بالا به پایین انتشار می یابند.مسئله بدست آوردن تابع انتقال ثباتهاست.الگوریتم انتشار ثباتها میتواند بسیار پیچیده باشد.علت این است که مجموعه ی Gen برای هر بلاک اولیه در حال تغییر و تحول است. برای نمونه به شکل زیر توجه کنید.



همانگونه که در اینجا مشاهده می کنید در آغاز کار مشخص نیست که آیا جایگزینی i:=5; به بلاک B2 می رسد یا نه که عملاً مطابق شکل این تعریف به این بلاک نمی رسد. بنابراین نیاز است که الگوریتم چندین بار تکرار شود و در هر تکرار ممکن است متغیرهای ثابت جدیدتری ظاهر شوند.

اما مجموعه های def و use را چگونه می توان مشخص کرد؟ الگوریتمی که پیشنهاد می گردد به ترتیب زیر می باشد:

Algorithm OnePathOf***ConstantPropagation***

1. **for** each node *n* in *CFG* **do**
   GEN[*n*] = set of (a, b) | a = b in Block *n* and b is Constant
   KILL[*n*] = Set of redefinition of Constants in the block n
   OUT[*n*] = GEN[*n*]

```
   endfor
2. change = true
3. while change do
4.     change = false
5.     for each node n in traverse(CFG) do
6.         IN[n] = ∩ OUT[P], where P is an immediate predecessor of n
7.         OLDOUT = OUT[n]
8.         OUT[n] =GEN[n] ∪ (IN[n] - KILL[n])
9.         if OUT[n] != OLDOUT
             then change = true
           endif
       endfor
   endwhile
```

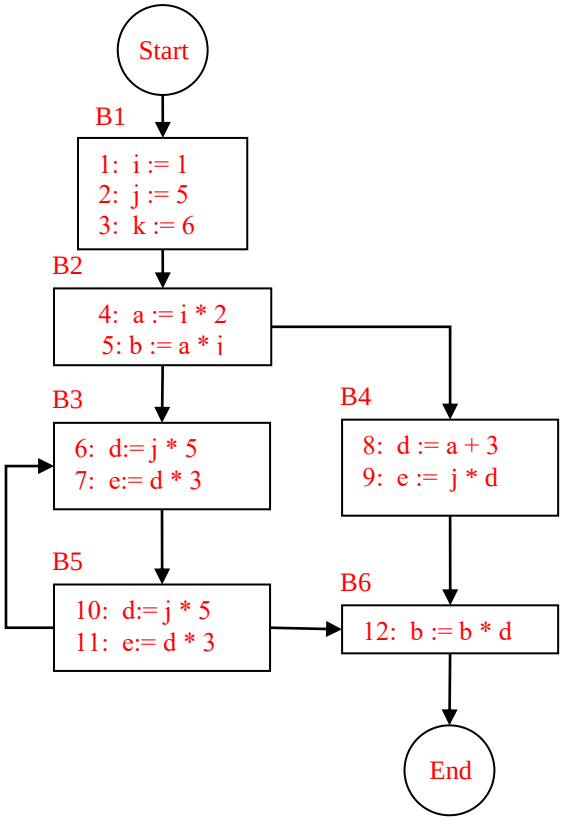
البته بعد از اجری الگوریتم فوق باید مقادیر ثابتی که به هر بلاک اولیه می رسد در محلهای استفاده شده درداخل بلاک اولیه جایگزین شوند. در صورتی که ثابت جدیدتری ایجاد شد ، مجدداً مجموعه های Gen و Kill برای بلاکهای اولیه محاسبه و الگوریتم تکرار می شود.

Algorithm *Constant Propagation*

```
      Input : Gen(n), Kill(n)
1.    NewConstants := true;
2.    While NewConstants do
3.      OnePathOfConstantPropagation;
4.       NewConstants := False;
5.      For each basic block n in CFG do
6.        Temp = IN(n)
9.        for s = first statement to last statement in n do
10.           if s = 'x := constantValue'
11.             then remove any existing definition of x from Temp
12.                  add 'x= constantValue' to  Temp
13.             else if s = 'x := y op z' then if both y=c1 and z = c2 belong to temp
14.                                          then remove any existing definition of x from Temp
12.                                               add 'x= c', where c = c1 op c2, to  Temp
13.                                          else remove any definition 'x = c' from Temp;
14.       end for;
15        NewConstans := In(n) <> Temp;
16.       Sunstitues any def
```

۱. الگوریتم معمول به نام Find Constants فراخوانی میشود.
   این الگوریتم برای هر بلاک اولیه مجموعه های IN و OUT را بدست می آورد.
۲. مقادیر ثابت که در مجموعه IN هر بلاک اولیه قراردارند ، درداخل بلاک اولیه جایگزین شوند.
۳. برای کلیه بلاکهای اولیه مجدداً مجموعه Gen محاسبه می گردد.
۴. برای کلیه بلاکهای اولیه مجدداً مجموعه Kill محاسبه می گردد.
۵. مرحله ی ۱ تا زمانی که ثابت های جدیدتری ایجاد میشود ، تکرار میگردد.

برای نمونه به مثال زیر توجه کنید.



با انجام الگوریتم انتشار ثابت روی گراف کنترلی فوق نتایج زیر بدست می آید:

B#	Gen	Kill	Initial	
			in	Out
B1	{1,2,3}	Null	Null	{1,2,3}
B2	Null	Null	Null	{1,2,3}
B3	Null	Null	Null	{1,2,3}
B4	Null	Null	Null	{1,2,3}
B5	Null	{1}	Null	{1,2,3}
B6	Null	Null	Null	{1,2,3}

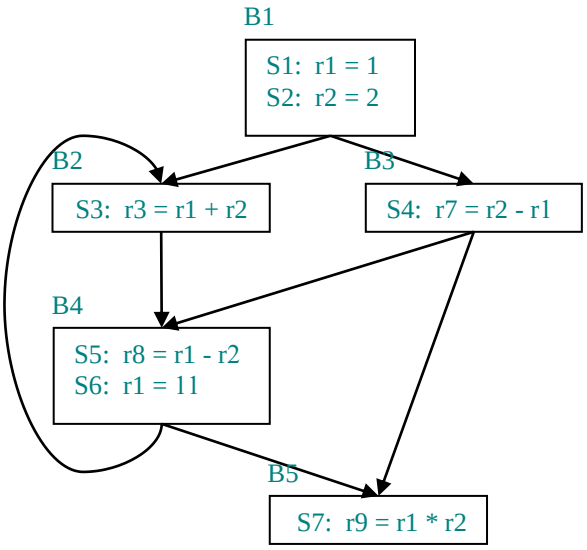
مجموعه ی OUT را در ابتدای کار برای همه ی بلاکهای اولیه برابر با کل تعاریف موجود در نظر می گیریم.

B#	In	Out	In	Out	....
B1	Null	{1,2,3}	Null	{1,2,3}	....
B2	{1,2,3}	{1,2,3}	{1,2,3}	{1,2,3}	....
B3	{1,2,3}	{1,2,3}	{2,3}	{2,3}	....
B4	{1,2,3}	{1,2,3}	{1,2,3}	{1,2,3}	....
B5	{1,2,3}	{2,3}	{2,3}	{2,3}	....
B5	{1,2,3}	{2,3}	{2,3}	{2,3}	....

شکل :- جداول انتشار ثابت

همانگونه که میبینید در تکرار اول الگوریتم خاتمه نمی یابد و تعاریف جدیدتری ایجادمی گردد.برای نمونه ، با جایگزینی مقدار I در B2 و B4 ، مقدار a ثابت ۲ میگردد. به همین ترتیب با جایگزینی مقدار j مقدار d ثابت ۲۵ می شود. حال باید الگوریتم تکرار شود و در تکرار سوم b و e هم به لیست ثباتها اضافه می شود.

بعنوان مثالی دیگر شکل صفحه بعد را در نظر بگیرید:



شکل ۲۰: نمونه ای از گراف جریان

حال الگوریتم انتشار ثابت را بر روی گراف کنترلی فوق اعمال می‌نمایم.

B#	Gen	Kill	Initial in	Out
B1	(r1, 1) (r2, 2)	(r1, -) (r2, -)	Null	(r1, 1) (r2, 2) (r1, 11)
B2	Null	(r3, -)	Null	(r1, 1) (r2, 2) (r1, 11)
B3	Null	(r7, -)	Null	(r1, 1) (r2, 2) (r1, 11)
B4	(r1, 11)	(r1, -) (r8, -)	Null	(r1, 1) (r2, 2) (r1, 11)
B5	Null	(r9, -)	Null	(r1, 1) (r2, 2) (r1, 11)

	In	Out	In	Out	In	Out
B1	Null	(r1, 1) (r2, 2)	Null	(r1, 1) (r2, 2)	Null	(r1, 1) (r2, 2)
B2	(r1, 1) (r2, 2)	(r1, 1) (r2, 2)	(r2, 2)	(r2, 2)	(r2, 2)	(r2, 2)
B3	(r1, 1) (r2, 2)	(r1, 1) (r2, 2)	(r1, 1) (r2, 2)	(r1, 1) (r2, 2)	(r1, 1) (r2, 2)	(r1, 1) (r2, 2)

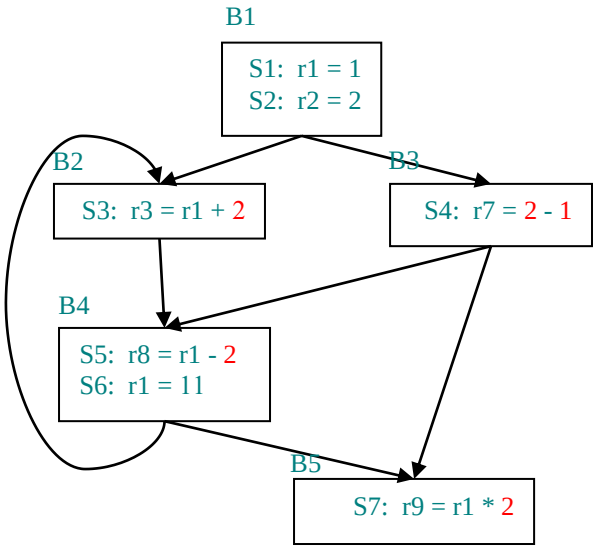
B4	(r1, 1) (r2, 2)	(r1, 11) (r2, 2)	(r2, 2)	(r1, 11) (r2, 2)	(r2, 2)	(r1, 11) (r2, 2)
B5	(r2, 2)	(r2, 2)	(r2, 2)	(r2, 2)	(r2, 2)	(r2, 2)

شکل ۲۱: جداول انتشار ثابت

بعد از اجرای الگوریتم می‌توان در هر بلاک تغییرات زیر را اعمال کرد:

- اگر (a, b) در In و Out یک بلاک باشد، می‌توان متغییر a را با مقدار ثابت b در آن بلاک جایگزین کرد.
- اگر (a, b) در Gen یک بلاک باشد، بعد از محل تعریف می‌توان متغییر a را با مقدار ثابت b در آن بلاک جایگزین کرد.
- اگر (a, b) در In یک بلاک باشد و در Out آن نباشد، امکان جایگزینی متغییر a با b تا مکانی وجود دارد که a مجدد تعریف می‌شود.

بعد از انجام تغییرات فوق شکل زیر حاصل می‌شود:



شکل ۲۲: اعمال انتشار ثابت در گراف جریان

سوال اینجاست که آیا روش دیگری برای حل این مسئله وجود دارد؟ مسئله ی فوق آیا یک مسئله NP-Hard می باشد یا NP-Complete ؟ آیا می توان با استفاده از تکنیکهای هوش مصنوعی که امروزه در بهینه سازی بسیار مطرح می باشند این مسئله ی ظاهراً ساده را که یک بهینه سازی بدیهی را انجام می دهد ، حل نمود؟ آیا با استفاده از زنجیره های تعریف و استفاده میتوان این مسئله را حل کرد؟

۴,۳,۴ حذف کدمرده یا کد غیرقابل دسترس (Unreachable Code and/or Dead Code Elimination)

کد مرده به دستوراتی گفته می‌شود که مقادیری را محاسبه میکنند ولی هیچگاه از آن استفاده نمی‌کنند. گاهی

اوقات نیز شروط دستورات شرطی بگونه ای است که در زمان کامپایل مشخص می گردد کـه بـه بخشـی از کـد هرگـز مراجعه‌ای صورت نمی‌گیرد که به آن بخش ، "کد غیرقابل دسترس" می‌گویند که توسط بهینه سـاز تشـخیص داده مـی شود. بدین ترتیب با حذف کد غیرقابل دسترس حجم کد کاهش می یابد.

چنانچه از یک نقطه به بعد در کد برنامه متغیری مورد استفاده یا زنده نباشد، متغیر مرده توصیف مـی شـود. فضـای تخصیصی به متغیر مرده بایدآزاد شود. با تعیین انتشار کپی متغیرها، می توان کار تشخیص کد مرده را ساده تر نمود. برای نمونه پس از تبدیل جملات از فرم:

```
x := t3;
a[t2] := t3;
a[t4] := x;
goto B2
```

بصورت:

```
x := t3;
a[t2] := t3;
a[t4] := t3;
goto B2
```

مشخص است که تخصیص مقدار به x در جمله x := t3; زائد است، لذا این جمله حذف می شود.

۴,۴ متغیرهای زنده

یک متغیر در یک فاصله زنده است اگر در آن فاصله مقدار آن متغیر مـورد اسـتفاده قـرار گیرد.زنـده بـودن متغیـر بنابرایت وابسته به استفاده بعدی است. بنا بر تعریف، متغیر A در صورتی در بلاک B زنده است که آن متغیـر در بـلاک B استفاده گردد. بنابر این، باید در ابتدا In(B) = Gen(B) در نظر گرفته شود. اما با تعریف مجدد یـک متغیـر، دیگـر آن متغیر زنده نخواهد بود زیرا مقدار آن در فاصله بین بلاک قبلی تا بلاک کنونی مورد استفاده قرار نگرفته اسـت. پـس در آغاز کار می توان مجموعه Kill را به صورت زیر تعریف نمود:

```
Kill[B] = Def(B) = { Set of variables defined in B}
Gen[B] = Use(B) = {Set of Variables used before a redefinition in B}
```

درواقع، مجموعه Kill مشخص می کند که چه مقادیر، برای کدام متغیرها عوض شده و در نتیجه مقدار قبلـی Kill شـده است یا مقدار جدیدی به متغیر تخصیص یافته است.

مجموعه Gen تعیین می کند که آیا متغیری در داخل بلاک استفاده شده یا خیر؟ همانطورکه بیان گردید، یک متغیـر در یک بلاک، درصورتی زنده است که بعد از آن نقطه و قبل از آنکه تعریـف مجـدد شـود اسـتفاده شـده باشـد. بنـابراین، مشاهده می کنید که زنده بودن به آینده مربوط است. در یک بلاک ممکن است متغیر i هرگز استفاده و یا تعریف نشده باشد، درصورتیکه در بلاک های بعدی استفاده شده باشد. دراینصورت، در داخل آن بلاک متغیر زنده بـوده و بنـابراین زنده بودن به وضعیت بعدی یا استفاده های بعدی مرتبط است.

مسلما اگر متغیری در یک فاصله زنده نباشد، دیگر نیازی به نگهداری مقدار آن نیست. این امر به ما در هنگام تولید کد و تخصیص ثبات ها جهت نگهداری مقدار متغیرها کمک می کند. معمولا در هنگام تولید کد سعی می شود تا بجای آنکه مقدار متغیری مثل i در حافظه نگهداری گردد در ثبات ذخیره شود. حال به جای ارجاع به متغیر i، به ثباتی که i در آن ذخیره شده ارجاع می شود. این امر موجب می گردد تا برنامه سریعتر اجرا شده و در نتیجه حجم آن کاهش یابد. دستورالعمل هایی که به جای ارجاع به حافظه، به ثبات ارجاع می کنند، هم کوتاهترند یعنی حافظه کمتری به خود اختصاص می دهند و هم سریعتر اجرا می گردند. این فایده زنده بودن است.

بنابراین تعیین زنده بودن باید از پائین به بالا حرکت نمود و مقدار Out برای بلک اولیه بصورت زیر محاسبه می شود:

$$Out[B] = \cup In[S]\ S \in Succ(B)$$

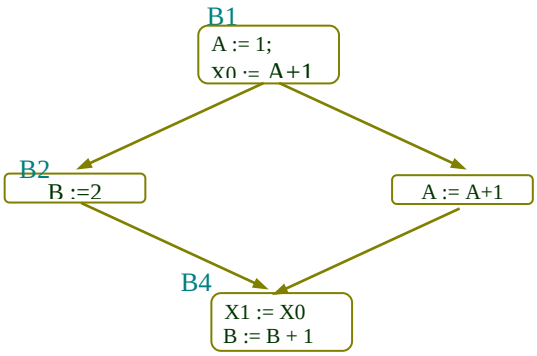
مقدار Out را برای هر بلاک تعیین کرده، اما حرکت از پائین به بالا را می توان در میان تکرارهای حلقه ایجاد نموده و ابتدا از بالا کار را آغاز کرد.

مقدار مجموعه kill که در واقع همان مجموعه تعاریف داخل بلاک است را می توان بلافاصله مشخص نمود. البته این ها شامل مجموعه kill هستند، اما اگر متغیری قبلا تعریف نشده باشد مسلما در مجموعه kill قرار نمی گیرد. الگوریتم بصورت زیر است:

Algorithm *Live Variable*

Input: Kill and Gen sets

1. **forall** Block *n* **do**
   IN[*n*] := NULL
   GEN[*n*] =Use(*n*)={ Set of Variables used before a redefinition in *n* }
   KILL[*n*] = Def(*n*)
   endfor
2. changes := true
3. **while** there are changes **do**
4. **for** each Block *n* **do**
5. OUT[*n*] := $\cup$ IN[*S*] S $\in$ Succ(*n*)
6. OLDIN := IN[*n*]
7. IN[*n*] := GEN[*n*] $\cup$ (OUT[*n*] – KILL[*n*])
9. **if** IN[*n*] != OLDIN
   then change = true
   endif
   endfor
   endwhile



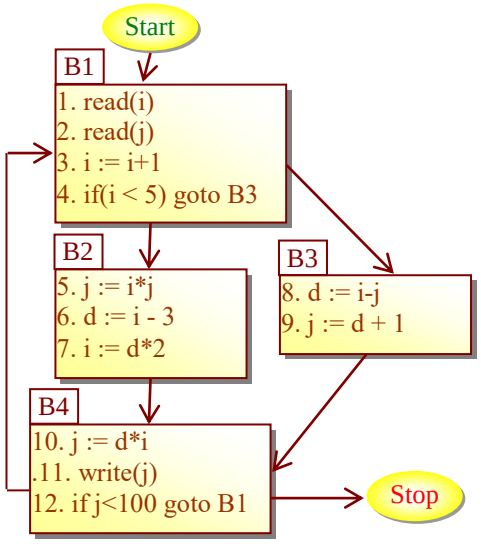
Initially:

Gen[B1] = Gen[B2] = ϕ
Gen[B3] = {A}
Gen[B4] = {X0, B}
Kill[B1] = ϕ
Kill[B2] = {B}
Kill[B3] = {A}

Kill[B4] = {X1,B}

شکل ۲۳: الگوریتم متغیرهای زنده

آنچه که مشاهده می نمایید، اطلاعات مربوط به زنده بودن از طریق In بلاک های بعدی بـه Out و از طریـق Out بـه In بلاک انتقال داده می شود. به مرور در طی اجرای حلقه While، این اطلاعات به بالا می رسد. برای نمونه در شـکل زیـر مجموعه های مورد نظر برای بلاکهای اولیه، بر طبق الگوریتم فوق مشخص شده است.



Initially:
Gen[B1] = ϕ
Kill[B1] = {(i, 1), (j, 2)}
Gen[B2] = {(i, 5), (i, 6)}
Kill[B2] = {(j, 5), (d, 6)}
Gen[B3] = {(i, 8)}
Kill[B3] = {(j, 8)}
Gen[B4] = {(d, 10)}
Kill[B4] = {(i, 10)}

	Gen	Kill	In	Out	In	Out	In	Out	In
B1	ϕ	i, j	ϕ	ϕ	ϕ	i, j	ϕ	i, j	ϕ
B2	i. j	j, d	ϕ	ϕ	i, j	d, i	i, j	d, i	i, j
B3	i, j	j, d	ϕ	ϕ	i, j	d, i	i, j	d, i	i, j
B4	d, i	j	ϕ	ϕ	d, i	ϕ	d, i	ϕ	d, i

شکل ۲۴: گراف جریان به همراه جدول

در آغاز همانند موارد قبل، بلاک های اولیه بصورت محلی مورد بررسی قرار می گیردکه مجموعه های Gen و Kill را برای آنها تعیین می نماییم. برای هر بلاک اولیه، ابتدا مقدار Gen را برابر مجموعه قابل استفاده درنظر می گیرند یعنی مجموعه متغیرهایی که در داخل آن بلاک اولیه قبل از تعریف مجدد، مورد استفاده قرار گرفته شده اند. مجموعه Kill برابر با مجموعه متغیرهایی است که در داخل آن بلاک اولیه تعریف شده است. توجه نمایید که صرفا با نام متغیرها کار داریم و بلاک مربوطه دیگر شماره جملات برای ما مهم نیست.

نکته اصلی که در ارتباط با متغیرهای زنده مطرح می باشد، انتقال اطلاعات از پایین به بالا است. متغیرهای زنده در تخصیص فضای حافظه بسیار حائز اهمیت هستند. معمولا در الگوریتم های تولید کد سعی بر آن است که اگر متغیری استفاده شود، در حد امکان بجای آدرس آن متغیر در حافظه مقدار متغیر را در ثبات نگهداری کند. اگر مقدارتخصیصی دیگر قابل استفاده نباشد نیازی به حفظ آن نیست.

به این ترتیب در داخل جدول نمادها، offset یک متغیر ممکن است صفر شود و یا اینکه در صورت نیاز آدرس جدیدی به آن تخصیص گردیده، یا یک ثبات به آن اختصاص یابد. وابسته به اینکه آیا متغیر پس از این زنده هست یا خیر، به آن فضایی تخصیص داده می شود. درصورتیکه متغیر زنده نباشدآن فضا آزاد می گردد. با زنده شدن مجدد، اگر ثباتی آزاد بود مقدار آن متغیر به ثبات تخصیص می یابد. ثباتی که دوباره مورد نیاز قرار گرفت اگر متغیر بیش از این زنده نبود آنگاه به آن فضایی در حافظه تخصیص داده نمی شود. بنابراین یک متغیرممکن است مکان های مختلفی از حافظه را وابسته به شرایط به خود اختصاص دهد. این مکانیزم برای درج برنامه ها در داخل حافظه های Drom بخصوص در برنامه های کنترلی رایج است.

۴,۵ عبارات موجود

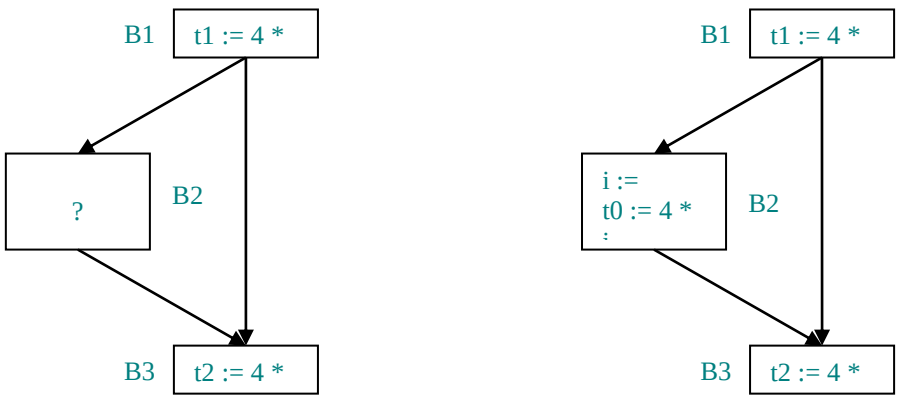
عبارت E با شکل کلی x op y در نقطه p1 از برنامه در نقطه دیگری مثل p2 موجود است اگر مقدار عبارت در این دو نقطه یکسان باشد و پارامترهای عبارت یعنی x و y در هیچ مسیری در فاصله p1 تا p2 تغییر نکرده باشند. درواقع با استفاده از این الگوریتم می توان عبارات تکراری را تشخیص داده، از محاسبه مجدد آنها ممانعت نمود. برای نمونه کد زیر را در نظر بگیرید:

```
If (condition) then
   X1 := A + BB* 12
Else X2 := A/2 + BB*12
…
/* no definitions for A & BB
…
X3 := BB * 12
```

در بالا چون BB و A قبل از محاسبه مقدار X3 تغییر نکرده اند، بنابراین نیازی به محاسبه مجدد عبارت BB*12 نیست.

Kill[B] = { (X op Y) ∈ Flow graph / X or Y are redefined in block B
Gen[B] = { (X op Y) / X or Y are not defined afterward in block B }
Out[B] = (In[B] – Kill[B]) ∪ Gen[B]
In[B] = ∩ Out[P], ∀ P ∈ Pred(P)

نمونه دیگری در شکل زیر ارائه شده است:



شکل ۲۵: زیرعبارت مشترک بین بلاک ها

در مثال فوق، در B1 می نویسیم:

```
T := 4 * i
T1 := T
```

در B2 می نویسیم:

```
T0 := 4 * i
T0 := T
```

و در B3 می نویسیم:

```
T2 := T
```

در حالت کلی چنانچه در نقطه P مجموعه A از عبارات قابل دسترسی وجود داشته باشد، آنگاه چنانچه q نقطه ای بعد از p باشد و در نقطه q جمله x := y + z مشاهده گردد. درصورتیکه بین p و q این جمله وجود داشته باشد، آنگاه برای اینکه بتوانیم عبارات موجود در q را بدست آوریم باید درصورتیکه مقدار y و z فرض نشده باشد در q به مجموعه A، y + z را اضافه نماییم اما چون مقدار x فرق کرده از داخل A می بایست هر عبارتی که شامل x باشد را حذف کنیم.

Statements	Available Experssions
	……. none
a := b + c	
	……. only b + c
b := a – d	
	……. only a – d
c := b + c	
	……. only b + c
d := a – d	
	……. none

شکل ۲۶: محاسبه عبارات موجود

برای نمونه در شکل فوق مثالی ارائه شده که می توانید نمونه هایی از عبارات قابل دسترسی را مشاهده نمایید. اما سوال اینجا است که مجموعه های Gen، In، Kill و Out به چه ترتیبی در اینجا محاسبه می شوند. مسلما برای هر بلاک اولیه مجموعه Gen عباراتی است که در داخل آن بلاک اولیه ظاهر شده و در هنگام خروج از بلاک اولیه مقدار آن عبارات فرقی نکرده باشد.

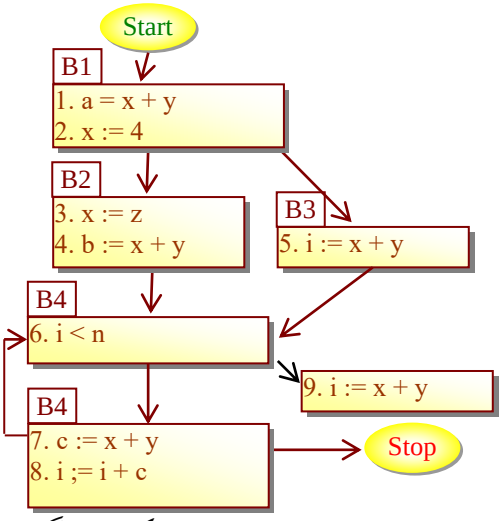
جنانچه برای مثال مقدار x در داخل بلاک اولیه تعریف شده باشد و هر عبارت قابل دسترس در داخل بلاک اولیه، یکی از پارامترهای آن x باشد، در مجموعه Kill برای آن بلاک اولیه قرار می گیرد. مجموعه های In و Out بصورت زیر محاسبه می گردد:

$$Out[B] = Gen[B] \cup (In[B] - Kill[B])$$
$$In[B] = \cap \text{ (P a predecessor of B) } Out[P]$$

همانگونه که مشاهده می کنید، روابط فوق برای محاسبه مجموعه های In و Out دقیقا مشابه روابط مورد استفاده برای محاسبه In و Out در الگوریتم تعاریف دسترسی شونده است . مقدار In[B1] در این الگوریتم، تهی درنظر گرفته می شود.

Algorithm Available Expressions

1. GEN(start) := NULL
2. IN(start) := {}
3. OUT(start) := GEN(start);
4. **for** each block *n* <> *start* **do**
   OUT[*n*] := U – KILL[*n*]
   endfor
5. change := true
6. **while** there are change **do**
7. change := false
8. **for** *n* <> *start* **do**
9. IN[n] := ∩ OUT[P], p a predecessor of *n*
10. OLDOUT := OUT[*n*]
11. OUT[n] := (IN[n] – KILL[n])
12. **if** OUT[*n*] != OLDOUT
    then change = true
    endif
    endfor
    endwhile



شکل ۲۷: الگوریتم تعیین عبارات موجود

هدف، مشخص کردن عباراتی است که مقدار آنها در برنامه تکرار می شود. هنگامیکه عبارات تکرار می شود به شرط معتبر بودن مقدار عبارت می تواند از مقدار محاسبه شده گرفته شود. این باعث بالا رفتن سرعت است چراکه محاسبات را کاهش می دهد.

همانگونه که مشاهده می کنید، الگوریتم زمانی خاتمه می یابد که برای کلیه بلاک های اولیه مقدار OLDOUT = Out[] گردد. یعنی خروجی تمامی بلاک های اولیه با مرتبه قبلی در حلقه While یکی باشد. اگر برای یکی از بلاک های اولیه آن مقدار مخالف باشد، کار باید ادامه یابد. همانگونه که بیان شد و مشاهده کردید، مقادیر ورودی به این الگوریتم باز هم مقادیر Gen و Kill می باشند که برای هر بلاک مستقلا قبل از شروع الگوریتم قابل محاسبه هستند.

برای مثال، در بلاک B1 عبارت x + y بدلیل اینکه بعد از این عبارت، مقدار متغیر x تغییر نموده، در مجموعه Kill قرار می گیرد. مجموعه Gen برای بلاک اولیه B1، Null است.

در بلاک B2، بعد از عبارت x + y، جمله x = z آمده است. بنابراین، x + y در مجموعه های Gen و Kill قرار می گیرد. بعبارتی دیگر، اگرقبل از این x + yیی وجود داشته باشد، Kill می شود. اکنون x + y جدید در مجموعه Gen قرار می گیرد.

در بلاک های اولیه ۵ و ۶، عبارت x + y مجددا آمده و در مجموعه Gen قرار می گیرد. عبارت i + c مطرح نمی باشد، چراکه فقط در یک جا ظاهر شده و بلافاصله مقدار i عوض شده است. توجه داشته باشید که در بلاک اولیه B5، عبارت

x + y موجود است. اگر از بلاک اولیه B3 به بلاک اولیه B5 برسیم، مقدار x + y متفاوت است. از هنگامیکه از بلاک اولیه B2 به بلاک اولیه B5 برسیم، جهت بهینه سازی، x + y را در خطوط ۴ و ۵ در متغیری با یک نام مشابه T1 قرار داده و در سطر ۷ می نویسیم c := T1. همین عمل در سطر ۹ نیز با ید انجام شود و می نویسیم i := T1.

۴,۶ اسامی مستعار[1]

اگر یک یا چند عبارت بصورت مشترک مکانی از حافظه را مقداردهی کند، عبارات اسامی مستعار هستند. معمولا استفاده از اشاره گرها[2]و ارسال پارامترها بصورت فراخوانی با ارجاع[3]مساله اسامی مستعار را بوجود می آورد. اصولا وجود اشاره گر کار تحلیل جریان داده ها[4]را پیچیده می کند چراکه استفاده از اشاره گرها مقداردهی و استفاده از مقادیر را پنهان می سازد. درواقع این آدرس دهی غیرمستقیم است که مشکل ساز می باشد.

در حالت کلی x := *p می تواند از طریق p به هر مکانی از حافظه دسترسی داشته باشد و مقدار آن را استفاده کندکه در این حالت الگوریتم های متغیرهای زنده و تعاریف دسترسی شونده دستخوش مشکلاتی می شوند. بنابراین باید از تکنیک های تحلیل جریان داده ها استفاده نموده، مشخص نماییم که اشاره گر به بخش هایی از حافظه می تواند اشاره کند. اما سوال اینجا است که از انواع آدرس دهی های غیرمستقیم به چه صورت است.

قاعدتا تنها متغیرهایی که بصورت اشاره گر تعریف شده اند، این امکان را دارند که بطور غیرمستقیم به متغیرها ارجاع کنند. البته Temporaryهایی در داخل برنامه تعریف می شوند نیز ممکن است در قالب اشاره گر عمل نماید. اصولا تصمیم گیری د رمورد اینکه اشاره گر به چه چیزهایی ممکن است اشاره کند در قالب سه مورد زیر خلاصه می شود:

۱. مقدار آدرس یک متغیر در داخل اشاره گر ذخیره می شود در اینصورت از طریق این اشاره گر می توان متغیر را تغییر داد. برای مثال چنانچه داشته باشیم s: p = &a، در اینجا p می تواند بعنوان یک اسم مستعار برای a عمل نماید. درصورتیکه a یک آرایه باشد، آنگاه p = &a ± c بازهم به خانه ای از آرایه a اشاره می کند اگر c مقدار صحیحی باشد، p درواقع اسم مستعار برای a – c خواهد بود.
۲. چنانچه داشته باشیم s: p = q ± c، دراینصورت پس از اجرای s، p به هر آرایه ای که q اشاره کند اشاره خواهد کرد.
۳. چنانچه داشته باشیم p = q و p و q دو اشاره گر باشند، واضح است که پس از اجرای این دستورالعمل p و q می توانند به یک مکان حافظه اشاره کنند و اسم مستعار باشند.

حال مسئله پیداکردن اسامی مستعار مطرح می شود. برای این منظور مجموعه In[B] برای هر بلاک اولیه بدین ترتیب تعریف می کنیم که در In[B] برای هر اشاره گر مثل p متغیرهایی را مشخص می نماییم که اشاره گر P می تواند به آن اشاره کند. بعبارت دیگر در In[B] برای هر اشاره گر p مشخص می کنیم که برای کدام متغیرها p می تواند بعنوان

---

[1] - Aliasing
[2] - Pointer
[3] - Call by Reference
[4] - Data flow analysis

اسامی مستعار درنظر گرفته شود. بنابراین In[B] بصورت جفت های (p, a) درنظر گرفته می شود که در اینجا p درواقع بعنوان اسم مستعار برای a مطرح است. به همین ترتیب برای هر مجموعه اولیه، مجموعه Out را مشخص می کنیم.

برای هر بلاک اولیه مانند B، تابعی به نام $trans_B$ تعریف می نماییم که این تابع In[B] را به Out[B] تبدیل می کند. درواقع، تابع trans نشان می دهد که اسامی مستعار قبل از ورود به بلاک، چه ترتیب بوده و در خروج آن به چه صورت تبدیل شده اند. برای مثال:

۱. چنانچه داشته باشیم s: p = &a ± c که در آن a یک آرایه است. دراینصورت:

$$Trans_s(s) = (s - \{(p, b)| \text{ any variable b}\}) \cup \{(p, a)\}$$

درواقع این عبارت می گوید که از مجموعه trans که در واقع نشان می دهد p به کجا اشاره می کرده، هر جای قبلی مثل b که p به آن اشاره می کرد حذف می شود. حال در این مجموعه (p, a) اضافه گردیده، این جفت نشان می دهد که p بعنوان اسامی مستعار برای a هستند.

۲. چنانچه داشته باشیم p = q + c که در آن q یک اشاره گر و c مقدار صحیحی است. آنگاه:

$$Trans_s(s) = (s - \{(p, b)| \text{ any variable b}\}) \cup \{(p, b)| (q, b) \text{ is in s and b is array variable}\}$$

دراینجا p قبلا به مکان b اشاره می کرد که مکان b خود خانه ای در آرایه q است.

۳. چنانچه p = q باشد آنگاه:

$$Trans_s(s) = (s - \{(p, b)| \text{ any variable b}\}) \cup \{(p, b)| (q, b) \text{ is in s}\}$$

۴. اگر p یک اشاره گر یا هر عبارت دیگر باشد:

$$Trans_s(s) = s - \{(p, b)| \text{ any variable b}\}$$

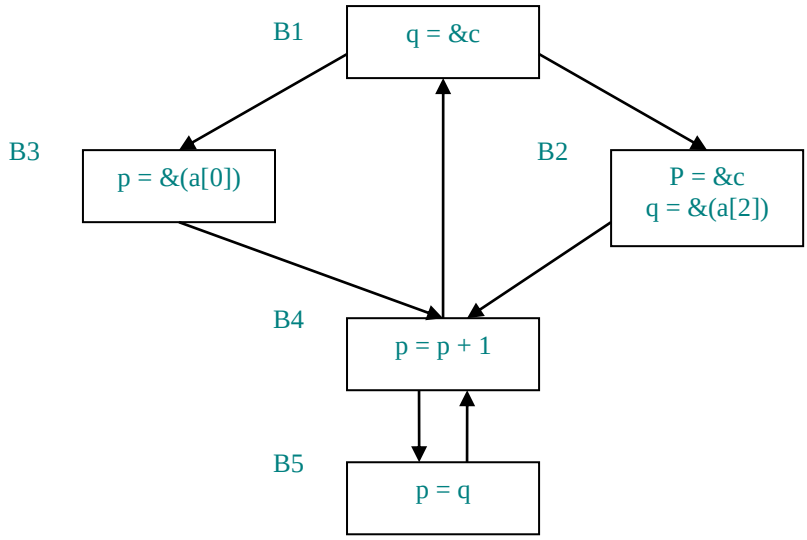
۵. اگر اشاره گر نباشد:

$$Trans_s(s) = s$$

بنابراین الگوریتم ساده فوق می خواهد بگوید که مقدار In[B] تحت تاثیر تخصیص مقادیر به اشاره گرها در طول اجرای بلاک اولیه B دستخوش تغییراتی می شود و نهایتا به دنبال این تغییرات Out[B] مشخص می گردد:

$$Out[B] = trans_B(In[B])$$
$$In[B] = \cup \text{ (p a predecessor of B) } Out[B]$$

برای نمونه به گراف زیر توجه نمایید:



شکل ۲۸: گراف جریان

گراف جریان فوق را درنظر بگیرید. در اینجا a یک آرایه و c مقداری صحیح است. p و q دو اشاره گر می باشند. در ابتدای کار فرض می شود که In[B] برای کلیه بلاک های اولیه مجموعه تهی است. اولین گراف B1 می باشد. در داخل B1، اگر توجه کنید جمله q = &c قرار گرفته است، بنابراین داریم: $Out[B1] = trans_{B1}(\varphi) = \{(q, c)\}$. از آنجاییکه تنها بلاکی که قبل از B2 قرار گرفته، B1 می باشد، پس $In[B2] = Out[B1]$ ست. تاثیر p = &c، توسط زوج (q, a) مشخص می شود. درواقع q = &(a[2]) معادل جمله q = &a + 2 است بنابراین مشاهده می نماییم که $Out[B2] = trans_{B2}(\{(q, c)\}) = \{(p, c), (q, a)\}$. ورودی یا $In[B2] = Out[B1] = \{(q, c)\}$ است که این ورودی در خروجی با مجموعه {(p, c), (q, a)} جایگزین می شود.

به همین ترتیب $Out[B3] = \{(p, a), (q, c)\}$ است. اما اگر به شکل توجه نمایید:

$$In[B4] = Out[B2] \cup Out[B3] \cup Out[B5]$$

اما در ابتدای کار Out[B5] را نداریم و فرض بر این است که در آغاز کار مجموعه Out برای کلیه بلاک های اولیه Null می باشد. از طرف دیگر داریم: $In[B5] = Out[B4]$ یعنی یک وابستگی دو طرفه. بنابراین در ابتدا با درنظر گرفتن اینکه Out[B5] برابر φ می باشد داریم: $In[B4] = \{(p, a), (p, c), (q, a), (q, c)\}$.

یک اشاره گر ممکن است که در زمان کامپایل تشخیص داده شود تا به چند مکان مختلف اشاره کند و برای انها اسم مستعار می باشد. البته در زمان اجرا همانطور که می دانید یک اشاره گر درزبان تنها به یک سوال می تواند اشاره نماید. برای هر بلاک اولیه In[B] در قالب زوج های (p, a)، مشخص می شود که در اینجا p اشاره گر و a یک متغیر است. این دو نشان می دهد که p ممکن است در زمان اجرا به متغیر a اشاره می نماید. درواقع در هنگام ورود به بلاک رخ می دهد و Out[B] به همین ترتیب برای هر اشاره گری مثل p مشخص می کند که در هنگام خروج از بلاک اولیه به کدام aها ممکن است اشاره نماید.

$trans_B$ درواقع تاثیر بلاک B را برای زوج های (p, a) مشخص می کند و برای هر مجموعه In مثل s، $trans_B(s)$ شاخص هر جمله s در داخل بلاک B تاثیر اسامی مستعار را مشخص می کند. همانطور که قبلا توضیح داده شد:

$Out[B] = trans_B(In[B])$
$In[B] = \cup$ (p a predecessor of B) Out[p]

مسلما برای محاسبه مقادیر In و Out از آنجاییکه مقدار In براساس Out و مقدار Out براساس In درداخل یک حلقه محاسبه می شود. بازهم می بایست الگوریتم تکراری قبلی را مورد استفاده قرار دهیم و آنقدر تکرار نماییم تا برای مثال مقدار Out جدید با Out قبلی یک شود. برای نمونه بین B4 و B5 و همچنین B4 و B1 حلقه ای وجود دارد. مسلما باید آنقدر حلقه را تکرار نماییم تا مقادیر جدید محاسبه شده برای Out با مقادیر قبلی یکی شود.

۴,۷ بهینه سازی حلقه ها

بهینه سازی حلقه ها از اهمیت ویژه ای در تسریع اجراء برنامه ها برخوردار است، بدین خاطرکه معمولا کد درون یک حلقه چندین بار اجرا می شود و برای تسریع اجرای حلقه ها معمولا عملیات زیر انجام می گردد:

۱. انتقال کد: هرگاه دستوری در یک حلقه عمل ثابتی را انجام دهد که تأثیر خاصی در اجرای حلقه نداشته باشد، آن دستور به خارج حلقه منتقل می‌شود. که به این عمل "انتقال کد" گویند. کد جملاتی که مستقل از اجرای حلقه می باشند را به قبل از نقطه شروع حلقه انتقال می دهند تا در هر تکرار حلقه مجددا محاسبه نشوند. برای مثال:

```
for i := 1 to 5 do
  begin
        j := 5;
        write(i)
  end;
```

⟶

```
j := 5;
for i := 1 to 5 do
   write(i);
```

شکل ۲۹: مثالی از بهینه سازی حلقه

۲. متغیرهای استقرایی[۱]: اگر در یک حلقه مقدار یک متغیر، بطور خطی افزایش یا کاهش یابد، به آن متغیر، متغیر استقرایی گفته می‌شود.

که معمولاً مقدار آن متغیر بر اساس متغیرهای دیگر به سادگی قابل محاسبه است. لذا می توان بعضا این متغیر را حذف نمود.

۳. کاهش توان[۲]: منظور از کاهش توان، جایگزینی جملات با جملاتی است که نسبتا سریعتر انجام می شود. برای نمونه چنانچه درون یک حلقه دو جمله بصورت زیر موجود باشد:

```
Loop:
j := j – 1
t4 := j*4
  ….
```

۱- Induction Variable

۲- Strength Reduction

Go to Loop

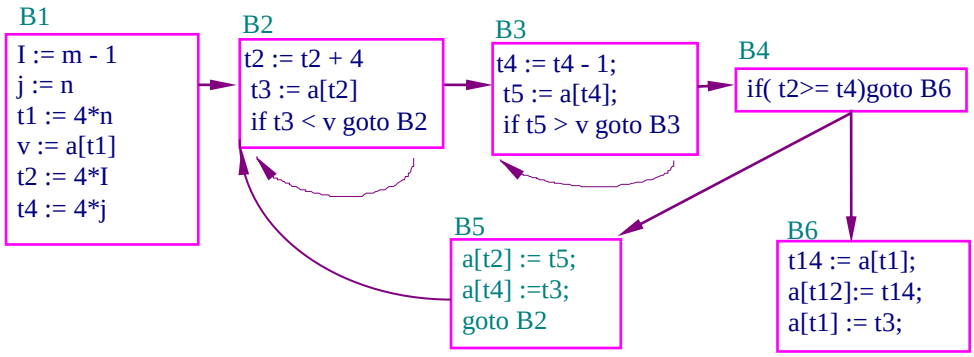
می توان جمله j := j - 1 را حذف نموده و سپس جمله t4 := j * 4 - 4 را نوشت. با افزایش جمله t4 := j * 4 در قبل از نقطه شروع حلقه، مقدار اولیه برای t4 معین می گردد. دو جمله فوق بصورت زیر تبدیل می شوند:

```
t4 := j * 4
Loop:
j := j - 1
t4 := t4 - 4
 ….
Go to Loop
```

دراینصورت در هر بار تکرار حلقه، عمل ضرب در جمله t4 := j * 4 با عمل تفریق به صورت t4 := t4 - 4 جایگزین می شود. با در نظر گرفتن روابط t2 := 4 * i، t4 := 4 * j و اینکه مورد استفاده بعدی از این دو متغیر در جمله

if(i >= j) then goto B6

می توان جمله if را بصورت if(t2 >= t4) then goto B6 تبدیل نمود. حال، پس از حذف متغیرهای استقرایی و کاهش توان، در بلاک های B2 به بعد دیگر نیازی به دو متغیر i و j نبوده، در نتیجه می توان آنها را حذف نمود.به این ترتیب برای نمونه، گراف جریان کنترلی برای تابع مرتب سازی quickSort به صورت زیر تبدیل می شود.



شکل ۳۰: بهینه سازی حلقه ها

نکته قابل توجه در مورد حلقه ها، چگونگی تشخیص حلقه می باشد. در این راستا باید ابتدا و انتهای حلقه را مشخص نمود. بطور طبیعی هر حلقه یک نقطه ورود و یک یا چند نقطه خروج دارد. به این دسته از حلقه ها حلقه طبیعی گفته می شود.

۴,۷,۱ تشخیص حلقه در گراف جریان

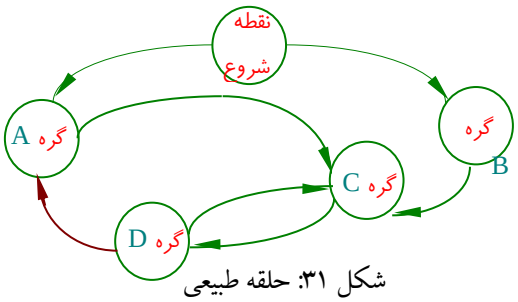
برای انتقال کد بدون استفاده از درون حلقه های موجود، باید ابتدا و انتهای حلقه را مشخص نمود. بخصوص اینکه به دلیل تکرار بدنه حلقه، بهینه سازی درون حلقه ها تاثیری چند برابر دارد. برای تشخیص حلقه ابتدا حلقه ها را بصورت ساده، یا در اصطلاح طبیعی در نظر می گیرند. حلقه های طبیعی دارای ویژگی های زیر هستند:

۱- فقط از طریق گره آغازین حلقه بتوان وارد حلقه شد.

۲- گره آغازین حلقه بر کلیه گره های حلقه باید مسلط باشد.

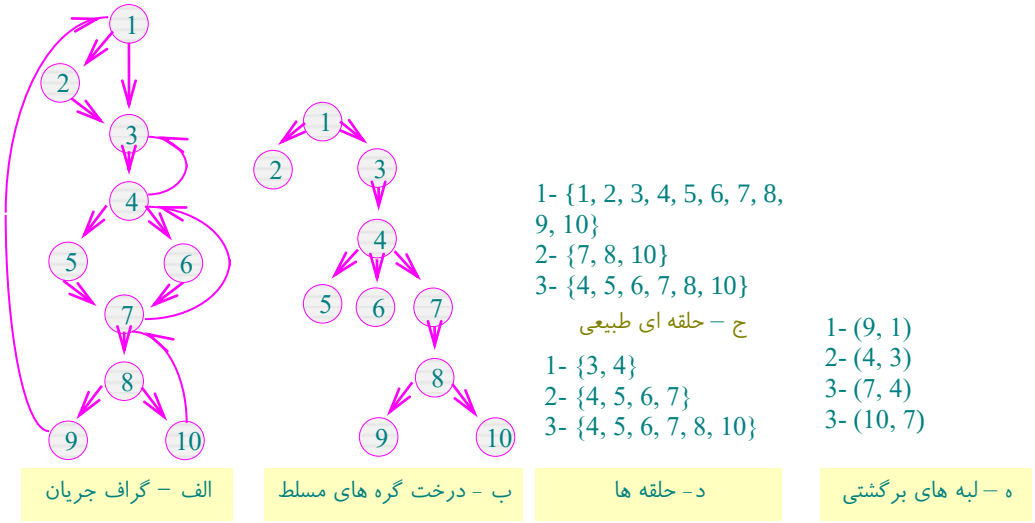
۳- حداقل یک مسیر یا لبه برگشتی برای تکرار حلقه موجود داشته باشد.

گره A بر گره B مسلط است اگر هر مسیر از گره شروع به گره B، حتما از A عبور کند. به شکل زیر توجه نمائید:



شکل ۳۱: حلقه طبیعی

در شکل فوق، گره A بر هیچ گره دیگری مسلط نیست. لذا، گره A با C و D تشکیل یک حلقه طبیعی را نمی دهند. حلقه طبیعی تنها در بین C و D وجود دارد. در شکل زیر، حلقه طبیعی و درخت گره های مسلط برآن مشخص شده است.



شکل ۳۲: حلقه طبیعی و درخت گره های مسلط

حلقه های طبیعی فقط یک نقطه ورودی و چند نقطه خروجی دارند. البته برای ساختارهای ساختیافته باید بتوان کلیه نقاط خروج را به یک نقطه وصل نموده تا در ظاهر یک نقطه خروج برای حلقه مشخص گردد. برای بدست آوردن حلقه های طبیعی باید لبه های برگشتی گراف جریان را مشخص کرد. در یک لبه برگشتی، گره انتهایی بر گره ابتدایی لبه مسلط است. برای نمونه، لبه بین دو گره ۹ و ۱ در گراف جریان ارائه شده در شکل ۳۲ یک لبه برگشتی است. با در دست داشتن لبه های برگشتی می توان طبق الگوریتم زیر حلقه های طبیعی را مشخص نمود.

برای یافتن گره های درون حلقه طبیعی، آخرین گره درون حلقه را حذف می کنند. گره های قبلی آن را نیز حذف کرده تا نهایتا به ابتدای حلقه برسند. برای نگهداری موقتی گره ها در ضمن اجرای حلقه، از پشته استفاده می شود. در زیر الگوریتم تعییین گره های درون حلقه طبیعی ارائه شده است. در این الگوریتم $n \rightarrow h$ یک لبه برگشتی است.

```
-----------------------------------------------
Procedure Insert ( m )
If m is not in loop then begin
Loop = loop ∪ m
Push m into the stack
End
-----------------------------------------------
Stack = empty
Loop = {h}
Insert ( n )
While Stack is not empty do
Pop m
For each predecessor of m do Insert( m)
End while
End procedure
-----------------------------------------------
```

شکل ۳۳: دو الگوریتم مجزا برای ساختن حلقه های طبیعی

۴,۸ تحلیل جریان داده ها در بین روال ها

در یک برنامه از تابع یا روال نیز استفاده می شود. معمولا برای تشخیص آنها، گراف فراخوانی[۱] تشکیل می دهند. در صورتیکه یک تابع یا روالی دیگری را فراخوانی کند، هر کدام از آنها بعنوان یک گره گراف فراخوانی محسوب شده و بین هر دو گره یک لبه قرار می دهند. سوالی که در اینجا مطرح می شود این است که اگر مقصد فراخوانی مشخص نباشد، چه باید کرد؟

درواقع فراخوانی بصورت پلی مرفیسم می باشد. در فراخوانی های پلی مرفیسم مقصد فراخوانی معمولا وابسته به نوع شی، متفاوت بوده و نوع شی در زمان اجرا مشخص می شود. بنابراین از آنجاییکه در زمان کامپایل، نوع شی مشخص نیست و وابسته به شرایط اجرایی می باشد، لذا کلیه مقاصد مختلف یک فراخوانی پلی مرفیسم را باید درنظر گرفت.

توابع مجازی[۲] نیز می تواند مشکل ساز باشد. برای نمونه به شکل ۳۴ توجه نمایید.

بنابراین یک مسئله تشخیص ایستای گراف فراخوانی یا گراف جریان فراخوانی[۳] است. یعنی اینکه نشخیص دهیم که کدام متدها یکدیگر را فراخوانی کرده و گرافی از این فراخوانی ها ایجاد نماییم. بدین منظور روش هایی مانند CHA و روش بهینه شده RTA مطرح می باشند.

۱- Call graph

۲- Virtual function

۳- Call flow graph

معرفی	استفاده

class SUP
{
virtual int vf(…)
{ … }
}
class Sub1
{ …
int vf(…) { … }
… }
class Sub2
{ …
int vf(…) { … }
… }
	SUP *s;
	Sup1 s1;
	Sup2 s2;
	
	if I > 5
	s = Sub1
	else
	s = Sub2;
	s.vf(…)

شکل ۳۴: بکار گیری تابع مجازی

مسئله دیگری که مطرح می باشد، اسامی مستعار است که بواسطه پارامترهای ارجاعی[۱] در روش فراخوانی توسط ارجاع[۲] ایجاد می گردد. بعد از اینکه توانستیم گراف فراخوانی را تشخیص داده و ایجاد نماییم، می توانیم مساله اسامی مستعار را دنبال کنیم.

۴,۹ اسامی مستعار و گراف فراخوانی

هنگامیکه تابعی فراخوانی می شود، پارامترهایی که بصورت ارجاعی در داخل تابع طراحی شده اند، ممکن است مقدار آنها تغییر کند. بنابراین هر فراخوانی با ارجاع را می توان بعنوان یک تعریف درنظر گرفت. البته ممکن است در داخل تابع این پارامتر مقداردهی نشود. در اینصورت، این فراخوانی تعریفی برای پارامتر نمی باشد.

[۱] Reference parameter –

[۲] Call by reference –

متغیرهای عمومی[1] نیز ممکن است در داخل توابع مقدار آنها تغییر کند. در شکل زیر مثالی در این رابطه ارائه گردیده، در ادامه نیز الگوریتم بسیار ساده ای آورده شده است.

```
global g, h;
procedure main();
  local i;
  g := …;
  one(h, i)
end;

procedure one(w, x);
  x := …;
  two(w, w);
  two(g, x)
end;

procedure two(y, z);
  local k;
  h := …;
  one(k, y)
end;
```

Algorithm *Interprocedural analysis of changed variables*

Input: A collection of procedures P1, P2, …, Pn. If the calling graph in a cyclic, we assume Pi calls Pj only if j<i. Otherwise, we make no assumption about which procedures call which.
Output: For each procedure P, we procedure change[p], the set of global variables and formal parameters of P that may be changed explicity by P with no aliasing..

Method**: Compute def[p] for each procedure P by inspection and Execute the program.**

```
1. for each procedure P do
     change[p] := def[p]
   endfor
2. while changes to any change[p] occure do
3.    for i := 1 to n do
4.       for each procedure q called by Pi do
5.          add any global variables in change[q] to change[pi];
6.          for each formal parameter X (the jth) of q do
7.             if X is in change[q]
8.             then for each call of q by Pi do
9.                     if A  the jth actual parameter of the call is a global or formal parameter  of Pi
10.                          then add A to change[pi]
                       endif
                    endfor
             endif
          endfor
       endfor
    endfor
endwhile
```

[1] Glibal Variable

۴,۱۰ تشخیص وابستگی های کنترلی

اصولا ممکن است دو دسته وابستگی بین جملات وجود داشته باشد. یک دسته بعنوان وابستگی داده ای مطرح بوده و نوع دیگر وابستگی کنترلی است.

وابستگی داده ای بین دو مکان A و B با بکارگیری نتایج محاسبه شده مکان A، در مکان B مطرح است. محل محاسبه مقدار را در اصطلاح مکان تعریف می نامند. به این ترتیب زنجیره هایی برای ایجاد ارتباط بین مکان های تعریف مقادیر و مکان استفاده از مقادیر تعریف شده ایجاد می کنند. در اینصورت می توانند وابستگی داده ای را نمایش دهند. به این زنجیره در اصطلاح زنجیره تعریف و استفاده گفته می شود، که قبلا بطور اجمال توضیح داده شده است.

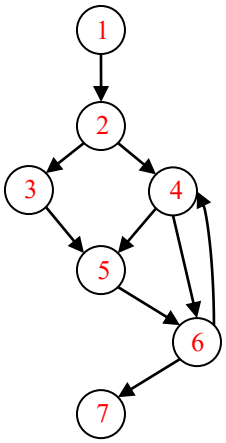
وابستگی کنترلی بدین معنی است که اجرای یک جمله، وابسته به تصمیمی است که در جمله دیگر گرفته می شود. برای مثال تمام جملات داخل بدنه for یا while همگی به ابتدای اسن جملات وابستگی منترلی دارند. همچنین یک جمله if را درنظر بگیرید. جملات داخل بدنه if همگی چه در بخش thenpart و چه در بخش elsepart وابستگی کنترلی به شرط if دارند.

۴,۱۱ وابستگی های کنترلی

اصولا ممکن است دو دسته وابستگی بین جملات وجود داشته باشد. یک دسته بعنوان وابستگی داده ای مطرح بوده و نوع دیگر وابستگی کنترلی است.

وابستگی داده ای بین دو مکان A و B با بکارگیری نتایج محاسبه شده مکان A، در مکان B مطرح است. محل محاسبه مقدار را در اصطلاح مکان تعریف می نامند. به این ترتیب زنجیره هایی برای ایجاد ارتباط بین مکان های تعریف مقادیر و مکان استفاده از مقادیر تعریف شده ایجاد می کنند. در اینصورت می توانند وابستگی داده ای را نمایش دهند. به این زنجیره در اصطلاح زنجیره تعریف و استفاده گفته می شود، که قبلا بطور اجمال توضیح داده شده است.

وابستگی کنترلی بدین معنی است که اجرای یک جمله، وابسته به تصمیمی است که در جمله دیگر گرفته می شود. برای مثال تمام جملات داخل بدنه for یا while همگی به ابتدای این جملات وابستگی کنترلی دارند. همچنین یک جمله if را درنظر بگیرید. جملات داخل بدنه if همگی چه در بخش thenpart و چه در بخش elsepart وابستگی کنترلی به شرط if دارند. در واقع نمیتوان شرط if را به صورت موازی با بدنه ی جمله ی if به اجرا در آورد. برای نمونه به شکل زیر توجه کنید:



گراف جریان کنترلی

همانگونه که در شکل مشاهده می کنید , گره شماره ۲ یک مرکز تصمیم گیری برای اجرای ۳ و ۴ است. لذا نمیتوان ۲ و ۳ و ۴ را بصورت موازی به اجرا در آورد اما وابستگی های داده ای در بین جملات یا دستورالعملها مطرح است.

۴,۱۱,۱ گراف جریان کنترلی

همانگونه که قبلا بیان گردید، گراف جریان کنترلی شاخص چندشاخه شدن مسیر اجرایی برنامه ها می باشد. معمولا دستورالعمل ها بصورت متوالی اجرا می شوند. وجود جملاتی مانند if، while و for موجب می شود که برنامه به چندشاخه تبدیل گردد. درواقع این نوع جملات تصمیم می گیرند که کدام شاخه به اجرا درآید. بنابراین جملات در شاخه اجرای آنها وابسته به شرط این نوع جملات می باشد. نقاط تصمیم گیری در برنامه ها، نوعی وابستگی ایجاد کرده و کنترل اجرایی وابسته به تصمیم اتخاذ شده تغییر می کند.

بنابراین، علاوه بر گراف وابستگی داده، گراف دیگری به نام گراف وابستگی کنترلی مطرح می شود. از ترکیب این دو گراف، گرافی به نام گراف وظایف[۱] مشخص می گردد. اصولا وظیفه[۱] به بخشی از برنامه ارجاع می شود که بصورت

[۱] - Task graph

موازی با بخش های دیگر قابل اجرا باشد. درصورتیکه هیچگونه وابستگی – وابستگی داده ای یا کنترلی - بین دو جمله وجود نداشته باشد، آنگاه آن دو جمله می توانند بصورت همروند یا موازی اجرا شوند.

کامپایلرهای موازی ساز، این وظیفه را برعهده دارند. این ها با ایجاد گراف وظایف و سپس زمانبندی کد ترتیبی را به کد موازی تبدیل می کنند. در داخل گراف وظایف، هر گره یک جمله کد میانی است. لبه های این گراف شاخص وابستگی بین جملات می باشد. این گراف جهت دار است. دو نوع لبه در آن وجود دارد. یکی شاخص وابستگی کنترلی است که با خط پر مشخص شده و دیگری وابستگی داده ای است که با خط چین مشخص می گردد.

۴,۱۱,۲ درخت تسلط

جهت یافتن لبه های برگشتی باید، لبه هایی را مشخص کرد که انتهای لبه بر ابتدا آن مسلط باشد. پس از افزودن گره های ابتدایی Start و انتهایی Stop به گراف جریان کنترلی، می توان مجموعه تحت تسلط را تعریف نمود. اگر هر مسیر از گره N1 به Stop حتما از N2 عبور کند، بر طبق تعریف گره N1 تحت تسلط یا پس تسلط توسط گره N2 است. بالعکس گره N1 مسلط بر N2 است اگر هر مسیر از Start به N2 حتما از N1 عبور نماید. برای بدست آوردن مجموعه گره هایی که هر گره گراف بر آنها مسلط است را می توان با استفاده از الگوریتم ذیل مشخص کرد:

Algorithm *Dominant*

```
1. Dominant(Start) = {Start}
2. forall node <> Start in ControlFlowGraph do
     Dominant(node) = All Nodes
   endfor
3. Changes = true
4. while Changes do
5.    Changes = false
6.    forall node <> Start in ControlFlowGraph do
7.       NewDominant = {node} ∪ { ∩ Dominatnt(p), ∀ p ε {predecessors(node)}}
8.       if Dominant(node) <> NewDominant
9.          then changes = true
         endif
10.      Dominat(node) = NewDominant
      endfor
   endwhile
```

شکل ۳۵: الگوریتم یافتن مجموعه گره های مسلط بر هر گره در یک گراف

الگوریتم فوق برای هر گره، مجموعه گره هایی که بر آنها مسلط است را درون Dominant مشخص می کند. برای نمونه به مثال زیر توجه نمایید:

---

[۱] Task -

D(1) = {1}
D(2) = {2} ∪ D(1) = {2, 1}, idom(2) = 1
D(3) = {3} ∪ D(2) = {3, 2, 1}, idom(3) = 2
D(4) = {4} ∪ D(2) = {4, 2, 1}, idom(4) = 2
D(5) = {5} ∪ D(4) = {5, 4, 2, 1}, idom(5) = 4
D(6) = {6} ∪ D(4) = {6, 4, 2, 1}, idom(6) = 4
D(7) = {7} ∪ {D(5) ∩ D(6)} = {7, 4 ,2 ,1}, idom(7) = 4

DF(1) = {} DF(2) = {2} DF(3) = {2} DF(4) = {2} DF(5) = {7} DF(6) = {7} DF(7) = {}

شکل ۳۶: گراف جریان کنترلی و درخت تسلط

در شکل فوق براساس مجموعه D مجموعه idom برای هر گره محاسبه می شود و براساس مجموعه idom می توان درخت تسلط را ایجاد کرد. مجموعهDF شاخص مجموعه ای تحت عنوان صف جلوی تسلط ها می باشد. این مجموعه برای گره x شامل مجموعه گره های w است که x بر یکی از گره های قبلی آنها مسلط است و x بر w مسلط نیست. برای نمونه در شکل فوق گره x=5 را در نظر بگیرید. مجموعه{7} = w می باشد. زیرا {5} = pred(7) = y است و x = 5 بر این گره مسلط است. بنابراین {7} = Df(5) می باشد.

۴,۱۱,۳ درخت پس تسلط[۱]

برطبق تعریف گره y بر z پس تسلط دارد – البته در داخل گراف جریان – اگر و تنها اگر برای اینکه از گره z به گره stop در داخل گراف جریان کنترل برسیم مجبور باشیم که حتما از گره y عبور کنیم. توجه داشته باشید که شاخه شاخه شدن اجرای برنامه ها در داخل گرافی به نام گراف جریان کنترل[۲]مشخص می شود.

هر گره این گراف یک بلاک اولیه است. دو گره خالی به آن اضافه کردیم، یکی به ابتدا که Start نامیده شده و دیگری به انتهای گراف یا نقطه خاتمه برنامه که Stop نامیده شده است. حال می گوییم در داخل این گراف جریان کنترلی، بلاک Bi به بلاک Bj تسلط دارد اگر و تنها اگر هر مسیری از start به Bj ناچارا از Bi عبور کند. بالعکس بیان می داریم که گره Bl بر گره Bj پس تسلط دارد اگر و تنها اگر هر مسیری از Bj به stop لزوما از Bl عبور نماید.

برای بدست آوردن درخت پس تسلط، ابتدا درخت گراف جریان را برعکس کرده – جهت لبه ها را عوض می نماییم– سپس براساس مجموعه idom می توان درخت پس تسلط را ایجاد کرد. منظور از پس تسلط در واقع مسلط بودن بر یک

۱- Post-dominate

۲- Control Flow Graph - CFG

گره برای مسیرهای آن گره به Stop است. در ادامه خلاصه الگوریتمی برای یافتن مجموعه های تحت تسلط ارائه گردید:

1. $D(n_0) = \{n_0\}$
2. **for** each node n in $N - \{n_0\}$ **do**
   D(n) = N
   endfor
3. **while** changes to any $D(n)$ occur **do**
4. **for** n in $N - \{n_0\}$ **do**
5. $D(n) = \{n\} \cup (\cap D(p)$ for all immediate predecessors p of n
   endfor
   endwhile

شکل ۳۷: خلاصه الگوریتم

الگوریتم فوق را با تغییر کوچکی برای محاسبه پس تسلط ها می توان مورد استفاده قرار داد. برای ایجاد درخت تحت تسلط می توان از الگوریتم ذیل استفاده نمود:

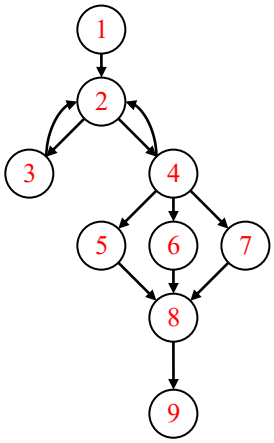
algorithm *BuildDtree*

Input. A set of nodes N for CFG G, with n_0 the entry node for G, and $D(n)$, the set of *nodes that dominaten*, for each node n in N.
Output. Dominator tree DT for G.

1. let n_0 be the root of DT;
2. put n_0 on queue Q;
3. **for** each node n in N **do** D(n) = D(n)-n **enddo**;
4. **while** Q is not empty **do**
5. m = the next node on Q (remove it from Q);
6. **for** each node n in N such that D(n) is nonempty **do**
7. **if** D(n) contains m
8. D(n) = D(n)-m;
9. **if** D(n) is now empty
10. *add n to DT as a child of m;*
11. add n to Q;
    endif
    endif
    endfor
    endwhile

شکل ۳۸: الگوریتم ایجاد درخت

همانگونه که در بالا متذکر شدیم، برای ایجاد درخت پس تسلط می توان گراف جریان معکوس کنترلی را با معکوس نمودن جهت لبه ها در گراف جریان کنترلی ایجاد نمود و سپس الگوریتم فوق را برای پس تسلط ها اجرا کرد.
در ذیل مثالی از یک گراف جریان کنترلی برای بدست آوردن درخت تسلط ، درخت پس تسلط و گراف وابستگی کنترلی آورده شده.



گراف جریان کنترلی

در این شکل اگر توجه نمایید گره ۴ بر ۵ و ۶ و ۷ تسلط دارد و گره ۴ بر گره های ۱و۲و۳ پس تسلط دارد.

گره ۲ بر ۴ تسلط دارد. همچنین بر ۳ هم تسلط دارد و همچنین بر گره ۴.(در واقع بر همه)

Dom(1)={1}

Dom(2)={1,2}

Dom(3)={1,2,3}

Dom(4)={1,2,4}

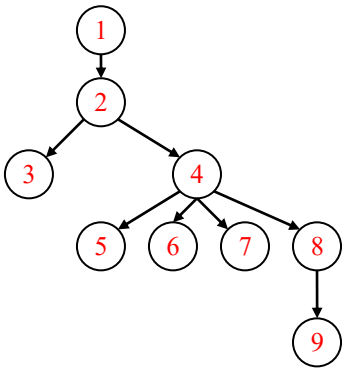
Dom(5)={1,2,4,5}

Dom(6)={1,2,4,6}

Dom(7)={1,2,4,7}

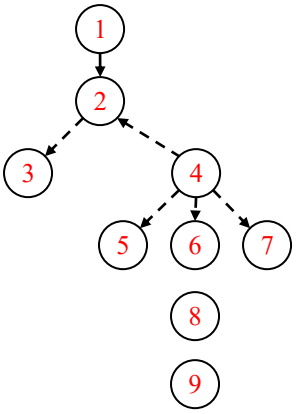
Dom(8)={1,2,4,8}

درخت پس تسلط بصورت زیر نبیجه می شود:

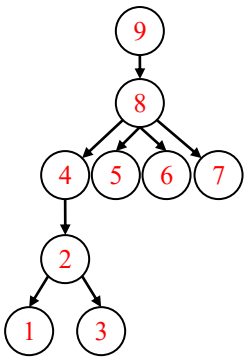


گراف تسلط

در ادامه درخت پس تسلط و گراف وابستگی مربوطه مشاهده می شود:



گراف وابستگی کنترلی



گراف پس تسلط

۴,۱۰,۴ ایجاد گراف وابستگی کنترلی

گراف وابستگی کنترلی یک گراف جهت دار است. گره ها، جملاتی در قالب کد میانی بوده و لبه ها شاخص وابستگی کنترلی می باشند. برروی بر چسب لبه ها جهت مشخص می شود. با استفاده از درخت تحت تسلط و گراف جریان کنترلی می توان گراف وابستگی کنترلی را ایجاد نمود. در واقع این عمل به نوعی، یک مرتب سازی گراف است. بدین منظور برای هر لبه (x, y) باید نزدیکترین پدر مشترک دو گره x و y را در داخل درخت تحت تسلط مشخص نمود. چنانچه این گره L باشد، L مسلما نمی تواند y باشد. L می تواند x و یا پدر x در درخت تحت تسلط باشد.

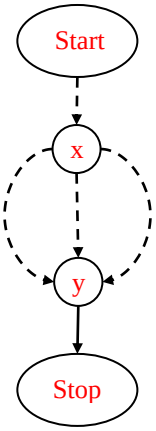
بر طبق تعریف گره y وابسته کنترلی به گره x است اگر در x شرایطی اجرا می شود که مشخص می کند آیا گره y باید اجرا شود یا خیر. به این ترتیب مشخص می شود که آیا حتی علیرغم عدم وجود وابستگی داده ای، گره y باید در انتظار برای خاتمه کار x باقی مانده و نمی تواند بصورت موازی با آن اجرا شود. این یک نوع وابستگی کنترلی است. بطور دقیقتر گراف وابستگی کنترلی به صورت زیر تعریف می شود.

گره y وابسته کنترلی به گره x است، اگر و تنها فقط اگر :

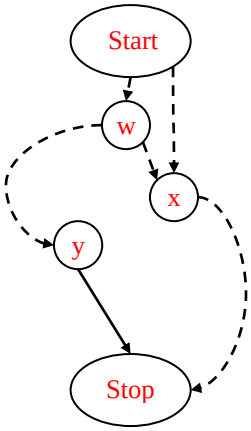
۱- مسیری از x به y در داخل برنامه وجود داشته باشد که با لبه ای با برچسب L آغاز شود و y بر هر گره w که بین گره های x و y است، حالت پس-تسلط داشته باشد.

۲- گره y بر x پس-تسلط نداشته باشد.

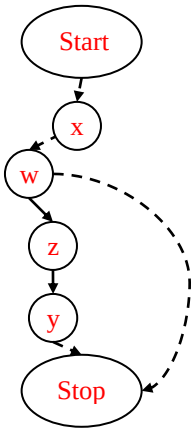
نکته قابل توجه در اینجا است که اگر گره y وابسته کنترلی به x داشته باشد، آنگاه گره x باید مستقیما قادر به تصمیم گیری در مورد اجرا یا عدم اجرای گره y داشته باشد. به عبارت دیگر، یکی از چند خروجی از گره x باید به y برسد.
برای درک مفهوم فوق , اولاً باید دید گره y بر x پس تسلط داشته باشد:



با توجه به شکل ,x نمیتواند یک مرکز تصمیم گیری برای اجرای y باشد, چرا که لزوماً با اجرای x, y هم اجرا می شود.چاره ی دیگری نیست . بنابراین یک شرط لازم و نه کافی برای اینکه y وابسته کنترلی به x باشد اینست که y , x را پس تسلط نکند.اما شرط دوم این بود که مسیری حتماً بین x و y وجود داشته باشد.برای نمونه به شکل زیر توجه کنید:



در این شکل مشاهده می کنید که y بر x پس تسلط ندارد و از آنجایی که هیچ مسیری بین x و y وجود ندارد , x یک نقطه تصمیم گیری برای اجرای y نیست. اما سومین شرط لازم این بود که مسیری بین x و y وجود داشته باشد که y بر همه ی گره های آن مسیر به جز x پس تسلط داشته باشد. فرض کنیم این گونه نباشد و گره ای مثل w در این مسیر وجود داشته باشد که y بر آن پس تسلط نداشته باشد. x در اینجا هم یک مرکز تصمم گیری برای اجرای y نیست.



الگوریتم ایجاد گراف وابستگی کنترلی

ورودی : گراف جریان کنترلی
خروجی : گراف وابستگی کنترلی

۱. گره آغازین Start را به ابتدای گراف جریان کنترلی اضافه نمایید
۲. درخت پس-تسلط را ایجاد کنید
۳. وابستگی های کنترلی را در طی مراحل زیر مشخص کنید:

الف - مجموعه لبه های (A,B) را مشخص کنید به قسمی که B جزء اجداد A در درخت پس تسلط نباشد.

ب - برای هر لبه (A,B) در S، گره L نزدیکترین جد مشترک A و B در درخت پس تسلط را پیدا کن.

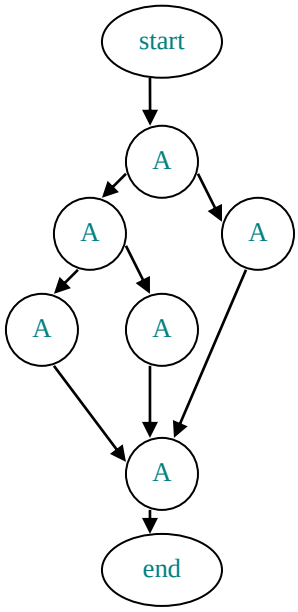
ج - درضمن پیمایش خلاف جهت از L به A ، هر گره در مسیر را علامت "مشاهده شد" بزنید. اگر L = A باشد، L را نیز علامت بزنید.

یا اینکه می توانید در درخت پس تسلط از B به سمت L درخت را خلاف جهت پیمایش نمایید.

د - تمام جمله های علامت خورده وابسته کنترلی به A هستند.

شکل ۳۹: الگوریتم ایجاد گراف وابستگی کنترلی

در شکل زیر یک گراف وابستگی جریانی ارائه شده است.



شکل ۴۰: نمونه ای از گراف جریان

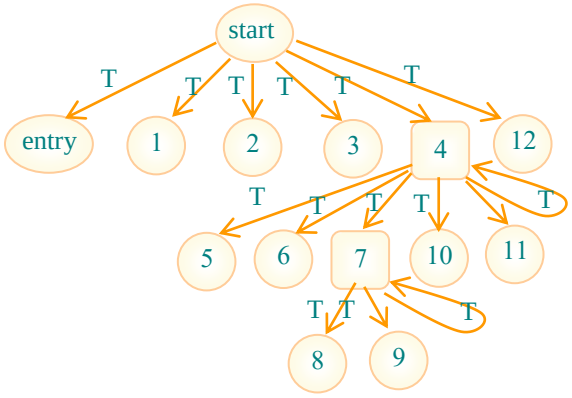
همانگونه که در شکل فوق مشاهده می نمایید، از گره A به گره B مسیر یا لبه ای وجود دارد. پس گره B بر گره A پس تسلط ندارد، یعنی گره A حتما یک نقطه تصمیم گیری است چراکه برای رسیدن از گره A به stop حتما مسیر دیگری نیز وجود دارد. لذا گره A یک نقطه تصمیم گیری است. اما گره A برای چه نقاطی تصمیم گیری می نماید؟ یکی از آن نقاط حتما گره B است. سوالی که در اینجا مطرح می شود این است که سایر نقاط به چه ترتیبی بدست می آیند؟

جد مشترک این دو در داخل درخت پس تسلط را درنظر بگیرید. مسلما گره L است. برای اینکه از A و B به stop برسیم، حتما می بایست از L عبور نماییم. اما اگر سایر مسیرها از گره A به L را درنظر بگیرید، سایر مسیرهایی که با لبه (A, B) شروع نمی شوند A یک نقطه تصمیم گیری برای اجرای آنها است. برای مثال گره C، E یا D را درنظر بگیرید. چون مسیر A به stop حتما از گره L عبور می کند، بنابراین A یک نقطه تصمیم گیری برای اجرای این سه نقطه هم می باشد.

توجه کنید که برای کلیه مسیرهای A به L، گره A نقطه تصمیم گیری می باشد. واضح است که در اینجا مسیر اجرایی شاخه به شاخه شده و این شاخه ها همگی به گره L برگشته اند.

در واقع از A به L بیش از یک مسیر وجود دارد. یک مسیر از حتما از B عبور می کند و مسیر دیگری هم حتما وجود دارد، زیرا در غیر اینصورت B بر A پس تسلط دارد. بدین ترتیب با ترسیم درخت وابستگی کنترلی مشخص می گردد که کدام جملات شرطی اجرا شده و کدام جملات حتما اجرا می شوند.

```
Program Sums
   1. read(n);
   2. i = 1;
   3. sum = 0;
   4. while (i<= n) do
   5.          sum = 0;
   6.          j = 1;
   7.          while (j <= i) do
   8.             sum = sum + j;
   9.             j = j + 1;
              endwhile;
  10.             write(sum, i);
  11.             i = i + 1;
      endwhile;
  12.          write(sum, i);
end
```



شکل ۴۱: گراف وابستگی جریانی

در حالت کلی چنانچه x بر گره y مسلط باشد، این رابطه را بصورت $(x\nabla_d y)$ نمایش می دهند. چناچه x بر y مسلط نباشد، رابطه بصورت $(x\nabla/_d y)$ نمایش داده می شود. رابطه پس تسلط یا در واقع تحت تسلط بودن گره y تحت x را بصورت $(x\nabla_p y)$ مشخص می کنند. در واقع این رابطه نشان می دهد که هر مسیری از گره y به stop حتما از گره x عبور می کند.

همانگونه که در بالا توضیح داده شد، مجموعه گره هایی که x بر آنها مسلط است، مجموعه گره هایی است که در گراف معکوس جریان کنترلی تحت تسلط گره x قرار دارند. می توان درخت تسلط را برای گراف معکوس نیز ترسیم کرده و بدین ترتیب گراف تحت تسلط ها را ایجاد نمود.

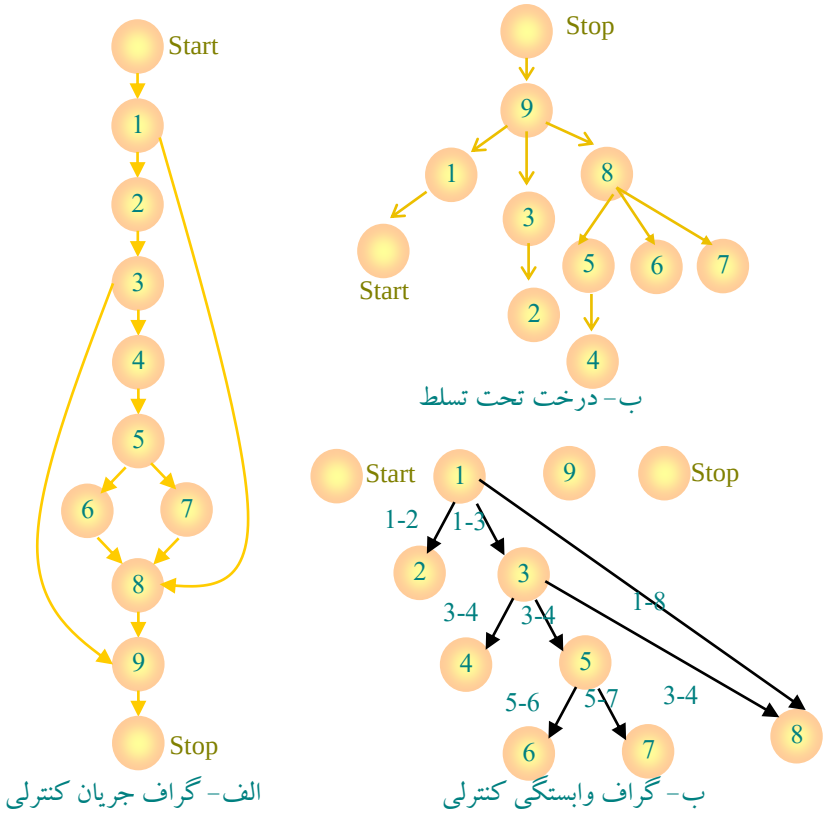
با استفاده از تعاریف فوق برای رابطه تحت تسلط بودن، می توان رابطه وابستگی کنترلی $(x\nabla_c y)$ را به صورت ذیل تعریف نمود:

گره y وابسته کنترلی گره x است اگر و فقط اگر:

۱. $(y\nabla/_p x)$ یعنی y پس تسلط بر x نداشته باشد.

۲. مسیری غیرتهی مثل p بین x و y وجود دارد، یعنی $P = <x \dots a \dots y>$ به قسمی که برای هر $z \in P$ و $z \notin x$, y خواهیم داشت: $(y\nabla_p z)$.

در شکل زیر نمونه ای از گراف جریان کنترلی، درخت تسلط و گراف وابستگی کنترلی مشخص شده است.



شکل ۴۲: مراحل ایجاد گراف جریان

۴,۱۲ الگوریتم های استخراج گراف فراخوانی

در این بخش جهت استخراج گراف فراخوانی الگوریتم های زیر را شرح می دهیم.

۱.آنالیز سلسله مراتب کلاس ها[۱]

۲.انالیز سریع نوع[۲]

۳.آنالیز استای نوع[۳]

ساختن یک گراف فراخوانی کم هزینه و دقیق برای یک برنامه شیء گرا توسط آنالیز ایستای کد منبع, یکی از مهمترین اهداف تحقیقات در سالهای اخیر به شمار می آید. این تحقیقات معمولا در زمینه بهینه سازی کامپایلر انجام می گیرد. در دو بخش بعدی ساختن یک گراف فراخوانی برای تابع Foo() شرح داده می شود.

```
Static Void foo (shape s)    {
                                    s.draw ();
                                    }
```

[1]- Class Hiearchly Analysis – CHA

[2]- Rapid Type Analysis – RTA

[3]- Static Type Analysis – STA

هر زمان که تابع Foo() اجرا می شود، مقصد عبارت s.draw() برحسب نوع حقیقی شی که به پارامتر رسمی bound,s شده است، تغییر می کند.نوع اعلان شده shape است، اما نوع حقیقی می تواند هر زیر نوعی از shape باشد. بعلاوه، نوع حقیقی ممکن است پیاده سازی را از نوع implementing، ارث ببرد که می تواند هر نوعی مافوق[1]نوع حقیقی باشد(شامل super-type نوع اعلان شده). فرض کنید Foo() همراه با ارجاعی به کد زیر نوشته شده باشد:

```
Abstract class shape{
        Abstract Void draw();
}
class Circle extend shape{
        Void draw() {printf("Circle");}
}
class Triangle extends shape {
        Void draw() {printf("Triangle");}
}
class Rectangle extend shape {
        Void draw() {printf("Rectangle ");}
}
class square extends Rectangle{}
```

حال با توجه به قطعه کد بالا دو الگوریتم CHA و RTA را توضیح می دهیم.

۴,۱۲,۱ الگوریتم آنالیز سلسله مراتب کلاس ها

این الگوریتم برای استخراج گراف فراخوانی، کل متن کد سیستم تحت مطالعه را نیاز داشته و نیز انواع داده ای پارامترهای واقعی و پیاده سازی شده را استخراج کرده، ساختار کلی برنامه را وارسی می کند. اما می دانیم که همواره به دلایلی تمام کد سیستم را نمی توان در اختیار گرفت که در مورد مهندسی معکوس در برخی موارد می توان با دردست داشتن قطعه هایی از کد نیز عملیات مورد نظر را انجام داد. در مثال زده شده، الگوریتم CHA برای s.draw() سه یال فراخوانی ircle.draw() و traingle.draw() و rectangle.draw() را ایجاد می کند. این که این الگوریتم کل کد برنامه را تحلیل کرده و اطلاعات لازم را استخراج می کند. یال shape.draw() را دز لیست خروجی خود ندارد چراکه طبق داده های جمع آوری شده کلاس shape کلاسی انتزاعی است و فاقد کد و هر نوع پیاده سازی است.

۴,۱۲,۲ آنالیز سریع نوع RTA

این الگوریتم با درنظر گرفتن دیگر اطلاعاتی که از متن برنامه به دست می آیند نتیجه ی به دست آمده از CHA را بهینه کرده و یال های اضافی به دست آمده از CHA را حذف می کند. منظور از اطلاعات داده هایی در مورد نوع و تعارف اشیا در متن برنامه هستند. در مثال فوق چون هیچ شی از نوع Triangular ایجاد نشده پر واضح است که Triangular.draw() نیز هرگز فراخوانی نخواهد شد. RTA از تمام نقاط ورودی برنامه شروع به وارسی کرده به صورت افزایشی گراف فراخوانی و اطلاعاتی در مورد نام و نوع اشیا استخراج می کند. در مثال زده شده، با بررسی متد main می بینیم تنها زیر نوعی از shape() که برای تعریف اشیا به کار رفته نوع square() است و لذا square نوع واقعی برای پارامتر s از تابع Foo() می باشد. اما نوع پیاده سازی کننده نه Square که Rectagle است. به این معنی که کلاس

[1]- Super-type

square پیاده سازی تابع ()draw را از کلاس Rectangle به ارث برده است. RTA نیز همانند CHA مستقل از متن بوده و از لحاظ زمانی و مکانی کار آمد است و برای استخراج گراف فراخوانی نیازمند کل برنامه است.

۴,۱۲,۳ الگوریتم آنالیز ایستای نوع STA

در این الگوریتم که ساده ترین آنها نیز می باشد، مقصد هر فراخوانی o.m() تنها با توجه به نوع اعلان شده برای متغیر o تعیین می شود. در واقع در اینجا در تعیین مقصد یک فراخوانی به ارجاعی که در زمان اجرا در آن متغیر قرار داده شده و مقصد واقعی فراخوانی را مشخص می کند، توجه نمی کنیم و نوع متغیر را به عنوان کلاس مقصد فراخوانی در نظر می گیریم. برای نمونه به مثال زیر توجه کنید.

```
Class A{
Public Void method1(){System.out.print("This is A");}
}
Class B extends A {
Public Void method1(){System.out.print("This is B");}
}
Class C extends A {
Public Void method1() {System.out.print("This is C");}
}
```

حال اگر قطعه کد زیر را داشته باشیم:

```
A a;
B b ;
a=new B();
a.method1()
```

از آنجایی که نوع اعلان شده برای a کلاس A بوده مقصد فراخوانی کلاس A در نظر گرفته می شود.

اگر بخواهیم الگوریتم CHA را در این مثال به کار ببریم، با توجه به این نکته که این الگوریتم، ترکیبی از اعلان استاتیک نوع و سلسله مراتب کلاس را جهت تعیین مجموعه مقصدهای احتمالی یک فراخوانی چندریختی به کار می برد، و با دانستن این حقیقت که در قطعه کد بالا a متغیری است که نوع اعلان شده آن کلاس A می باشد در نتیجه a می تواند حاوی ارجاعات به اشیائی از کلاس های A، B و C باشد.

به نظر می رسد که الگوریتم RTA با اینکه عملکرد بسیار خوبی نسبت به پیچیدگی زمانی کم آن داراست، اما در تعیین مقصد فراخوانی درشت دانه[۱]عمل می کند. بدین معنی که کافیست در کل برنامه حداقل یک نمونه از کلاسی ساخته شده باشد تا آن کلاس به عنوان نوع مقصد احتمالی در گراف باقی بماند. در حقیقت RTA می گوید که نوع A به گیرنده[۲] o می رسد،اگر یک نمونه از شئ با نوع A در هرجای برنامه موجود باشد(یعنی عبارت new A()) و A یک نوع محتمل و موجه برای o با استفاده از آنالیز سلسله مراتبی می باشد.

---

۱ – Coarse-grain

۲ – Receiver

۴,۱۲ گراف وظایف

بعدازاینکه وابستگی های داده ای و کنترلی را بدست آوردیم، گراف وظایف ایجاد می شود. در گراف وظایف، هر گره یک دستورالعمل سه آدرسه می باشد. هر لبه، شاخص وابستگی بین دو گره است که وابستگی داده ای با خط چین و وابستگی کنترلی با خط پر مشخص می گردند. درصورتیکه بین دو گره لبه ای در گراف وظایف وجود نداشته باشد، آن دو گره بصورت موازی قابل اجرا هستند.

کار کامپایلرهای موازی ساز[۱]و یا ابرکامپایلرها این است که کد ترتیبی را به کدی با قابلیت اجرای موازی تبدیل نماید. این ها برای این کار گراف وظایف را ایجاد کرده، آن را با مکان شناسی مرتب[۲]می کنند. در هر لایه، گره ها بصورت موازی قابل اجرا هستند.

از آنجاییکه گره ها در گراف وظایف، هر کدام یک دستورالعمل مستقل هستند –آن هم از نوع سه آدرسه–، لذا مقرون بصرفه نیست که هر دستورالعمل را بر روی یک پردازنده و بصورت موازی به اجرا درآورند. مسئله ای به نام دانه بندی[۳] در اینجا مطرح می گردد. در این حالت دانه بندی بسیار کم و در حد یک دستورالعمل بوده و درواقع دانه بندی بصورت ریز دانه[۴]می باشد. دانه بندی ممکن است درشت دانه[۵]باشد.

سوال اینجا است که چه تعداد دستورالعمل باید در یک گره قرار گیرند تا سربار ارتباط[۶]به وظیفه[۷]کم شود؟ مسلما هر چه تعداد وظایفی که بصورت ترتیبی اجرا می شوند بیشتر باشد، سربار ارتباط کمتر خواهد بود. اما از طرف دیگر، اگر همه دستورالعمل ها در یک گره قرار گیرند، اگرچه سربار ارتباط صفر می شود اما دیگر موازی سازی وجود ندارد.

از سوی دیگر مسئله توازن بار[۸]مطرح است. یعنی می بایست دستورالعمل ها را چنان بین پردازنده ها یا وظایف توزیع نماییم تا بار کار پردازنده ها یکسان شده، در تمام وقت همگی فعال بوده و هیچکدام منتظر باقی نمانند. این امر ایده آل موازی سازی بوده و بدین منظور الگوریتم زمانبندی بکار می آید.

۴,۱۳ حذف وابستگی های اضافی

همانگونه قبلا توضیح داده شد، در گراف وظایف هر گره شاخص جمله ای در قالب کد میانی و هر لبه در قالب خط چین شاخص وابستگی داده ای و خط پر شاخص وابستگی کنترلی گره انتهای لبه به گره ابتدای لبه است. چنانچه

۱- Parallelism Compiler
۲- Topological Sort
۳- Granularity
۴- Fine grain
۵- Coarse grain
۶- Communication
۷- Task
۸- Load balancing

گره b به گره a وابستگی داده ای داشته و گره c به گره b، واضح است که گره c به a وابستگی داده ای دارد. بنابراین در گراف وظایف لبه ای از گره c به گره a وجود داشته، می توان آن لبه را حذف کرد.

در مورد وابستگی های کنترلی اینچنین نیست. اگر گره b به گره a وابسته کنترلی باشد، گره c به گره a وابستگی کنترلی دارد. واضح است که گره c به گره a وابستگی کنترلی دارد. اما اگر لبه ای در این میان وجود داشته باشد آیا می توان آن را حذف کرد؟

درصورتیکه گره b به گره a و گره c به گره b حتما وابستگی کنترلی داشته باشد، این کار قابل انجام است.درصورتیکه گره c به گره b وابستگی داده ای داشته باشد، آنگاه نمی توان وابستگی کنترلی گره c به گره a را حذف کرد. خصلت وابستگی های کنترلی و داده ای با هم متفاوت بوده، لذا نمی توان یکی را با دیگری جایگزین کرد.

پس ممکن است گراف وظایف حاصل از تحلیل کنترلی و داده ای دارای لبه های اضافه شود. برای حذف لبه های اضافه هر لبه را حذف می کنند. اگر در مسیرهای موجود بین هر دو نقطه تغییری داده نشد، مسیر حذف شده اضافه بوده است. برای تعیین و حذف مسیرهای اضافه با استفاده از ماتریس مجاورت، ماتریس مسیر را بدست آوریم. اگر ماتریس مسیرها متفاوت از ماتریس قبلی باشد، لبه نباید حذف شود. چراکه با حذف آن، یکی از مسیرها از بین می رود. اما اگر مسیرها همچنان بجای خود باقی بمانند، نیازی به آن لبه وجود ندارد.

درواقع ماتریس مجاورت، ماتریسی است که بازای هر گره از گراف یک سطر و یک ستون تخصیص می دهند. درصورتیکه بین دو گره، لبه ای وجود داشته باشد، مقدار آن ماتریس در آن سطر و ستون مقدار ۱ و در غیراینصورت مقدار صفر خواهد داشت. پس، ماتریس مجاورت برای گراف G با m گره، ماتریسی است m*m بطوریکه هر سطر آن شاخص یک گره و همچنین بازای آن گره ستونی با همان شماره گره درنظر گرفته می شود. چنانچه ماتریس را M بنامیم، آنگاه درصورتی $M_{ij} = 1$ می باشد که از v_i به v_j لبه ای در گراف وجود داشته باشد، در غیراینصورت صفر است. حال ماتریس مسیر را می توان از ماتریس مجاورت بدست آورد.

الگوریتم بدینصورت است که مسیری را از داخل ماتریس M حذف می کنند –طول مسیر ۱ است– ماتریس مسیر را بدست می آورند. اگر با مسیر قبلی فرق نکرده باشد، آن مسیر زائد بوده و می توان آن را حذف کرد. درغیراینصورت وجود لبه ضروری است. در الگوریتم زیر فوق M ماتریس مجاورت است و *M ماتریس مسیر می باشد. بدینترتیب الگوریتم بصورت ذیل است:

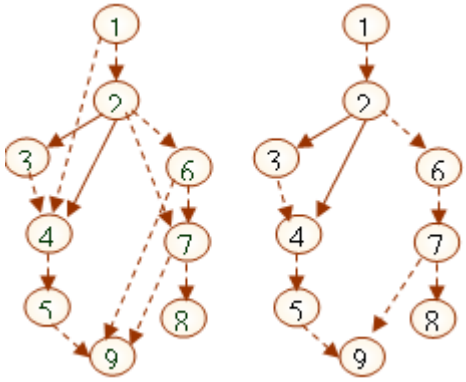
Algorithm *Reduce Dependency*

```
1. Given task graph, G, Construct Adjancey Matrix  M
2. Compute All Path Matrix  M*
3. for each elemnet M[i, j]  do
4.    if M[i, j] = 1
5.    then M[i,j] = 0
6.          Construct all path matrix MM* for M
7.          if  M* <> MM*
8.          then M[I,j] = 1
```

```
        endif
    endif
endfor
```

شکل ۴۳: الگوریتم حذف وابستگی های اضافی

اگر به شکل ۴۴ توجه نمایید، بین گره های ۱ و ۴ وابستگی داده ای وجود دارد. اما از طرف دیگر گره ۲ به گره ۱ وابسته داده ای و به گره ۴ به ۲ وابسته کنترلی دارد. بنابراین لبه ۱ به ۴ را می توان حذف نمود. توجه داشته باشید که گره ۳ به گره ۲ وابستگی کنترلی و گره ۴ به گره ۳ وابستگی داده ای دارد. رابطه ۴ به ۲ حذف نشده است، چرا؟



شکل ۴۴:نمونه ای از یک گراف وظایف

درشکل فوق همانگونه که مشاهده می کنید، رابطه ۴ به ۲ نشان می دهد که اجرای ۴ وابسته به اجرای ۲ است. بعبارت دیگر، در ۲ شرطی وجود دارد که وابسته به آن، ۴ اجرا می شود. حال، اگر این لبه را حذف نمایید مشکل ایجاد می شود. ۳ به ۲ وابسته کنترلی است، بدین معنی که در ۲ شرطی وجود دارد که اجرای آن برای ۳ تصمیم گیرنده است. اما ۴ به ۳ وابستگی داده ای دارد، درصورتیکه ۳ اجرا نشود، همچنان ۴ قابل اجرا خواهد بود. وابستگی داده ای همیشه حتما اجرای متوالی را نشان نمی دهد. برای مثال به جمله if زیر توجه کنید:

```
if i > j
then i = 5
else i = 6
k = i * 2;
```

بعد از جمله if فوق، جمله k = i * 2 به هر دو جمله i = 5 و i = 6 وابستگی داده ای دارد. این مطلب لزوما بیان کننده اجرای i = 5 و i = 6 قبل از اجرای k = i * 2 نمی باشد.

۴,۱۴ پیداکردن رشته های وظایف

بعد از اینکه لبه های اضافی گراف حذف شد، رشته های وظایف اجرا می شوند. رشته وظایف دنباله ای از وظایف است که اگر با یکدیگر ادغام شوند، درجه توازی در برنامه هیچگونه تغییری نمی کند. بعد از حذف لبه های اضافی، رشته وظایف سریال می تواند تشخیص داده می شوند. چون هر گره از گراف تنها یک دستورالعمل را شامل می شود، اندازه آن حداقل بوده و ریزترین اندازه را دارا است. لذا باید آن را درشت دانه تر نمود. گراف حاصل از ترکیب، درجه

توازی برابر درجه توازی گراف اولیه را دارا می باشد. وظایف سریال در یک وظیفه تکی کپسوله شده و وابستگی های کنترلی به وضوح بین گره های جدید وجود خواهد داشت.

یک رشته وظایف سریال، مجموعه ای از وظایف است که اجرا و تکمیل آنها تضمین شده است. این درصورتی است که اولین گره این رشته اجرا شود و وظایف به صورت سریال به یکدیگر وابسته باشند. تمامی گره ها در یک رشته وظایف باید وابستگی کنترلی مشابهی داشته باشند. تمامی وابستگی های داده ای وارد شده به رشته وظایف باید به گره اول این رشته وارد و وابستگی های داده ای خروجی باید از گره آخر رشته خارج شوند. الگوریتم تشخیص رشته وظایف در ادامه ارائه می شود:

Algorithm *Finding String(graph)*

```
1. Strings = ϕ
2. Possible_headers = {all tasks without predecessors}
3. While not all tasks done do
4.     New_headers = ϕ
5.     for each task head  ∈  possible_headers do
6.         string = <head>
7.         cur  =  head
8.         mark cur as done
9.         last = cur
10.        cur = next_string_node(cur)
11.        if  cur = nil
12.          goto  15
           endif
13.        string =  <string, cur>
14.        goto 8
15.        strings = strings  ∪  {string}
16.        new_headers = new_headers  ∪  Successrs(last)
       endfor
17.    possible_headers = new_headers
   endwhile
```

شکل۴۵: الگوریتم جستجوی رشته

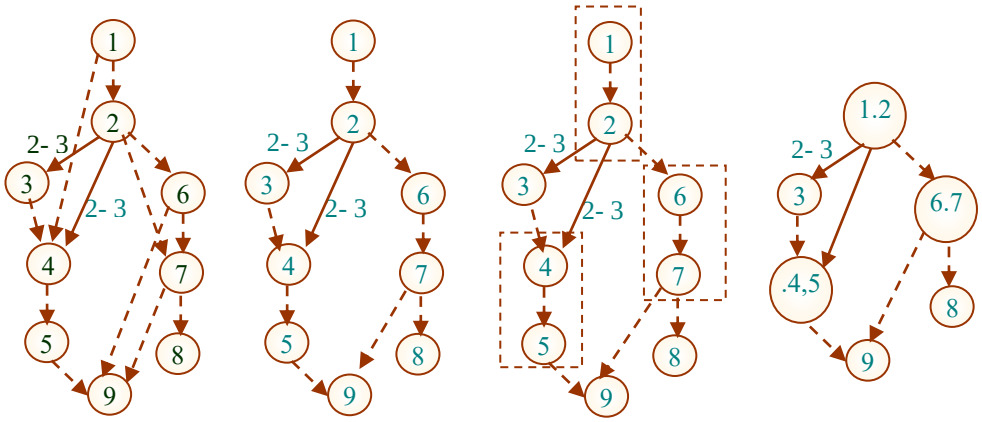
اگر به الگوریتم توجه نمایید، در ابتدا به دنبال وظایفی می گردد که هیچگونه وظیفه قبلی ندارد یعنی نقطه شروع وظایف است. سپس با استفاده از تابع next_string_node() سعی می کند گره بعدی در رشته وظایف را بدست آورد که با این گره بدون وظیفه قبلی شروع شده باشد. درصورتیکه گره اول هیچ فرزند یا وظیفه بعدی نداشته باشد مسلما رشته ای هم نمی تواند ایجاد یا آغاز کند. درغیراینصورت، چنانچه بیش از یک وظیفه بعدی داشته باشد، باز هم رشته وظایف تولید نمی شود.

بنابراین، به دنبال تک فرزندها می گردیم. رشته وظایف رشته ای است که به وسط آن پرشی وجود ندارد. اگر اولی شروع شد تا آخر ادامه می یابد. پس، گره های داخل یک رشته وظایف همگی وابستگی های کنترلی یکسانی دارند و به وسط آنها پرشی نمی شود.

وابستگی های داده ای باید در تمامی گره های یک رشته وظایف یکسان بوده، و بدین ترتیب رشته وظایف دنباله ای از وظایفی خواهد بود که تک فرزند، پی درپی، بدون تاخیر و انتظار هستند.

```
Next_String_Node( node )
// 1-  عدم وجود گره بعدی
10a_   if  node has no successors  return nil
//2-  وجود بیش از یک گره بعدی
10b_   if node has more than one successor return nil
//3-  تعیین گره بعدی
10c     succ  =  node's unique successor
//4-  اگر وابستگی کنترلی بین گره و بعدی آن یکسان نیست مقدار تهی برگردان
10d     if node and succ not identically control dependent return nil
10g     otherwise  if succ is not data dependent on a node on which the given Node is
                    Not dependent return succ
```

همانگونه که در الگوریتم فوق توجه می نمایید، در واقع رشته های اولیه را با در نظر گرفتن وابستگی داده ای ایجاد نمودیم. برای نمونه به شکل زیر توجه نمایید:



شکل ۴۶: نمایش چگونگی تشخیص و ترکیب رشته وظایف

همانگونه که در شکل فوق مشاهده می نمایید، تنها فرزند گره ۱، گره ۲ می باشد. همیچکدام از این دو گره وابستگی کنترلی ندارند. بنابراین گره های ۱ و ۲ می توانند تشکیل یک رشته وظایف بدهند. درصورتیکه گره ۲ انتهای رشته بوده و بیش از یک فرزند دارد، فرزندهای گره ۲ را به حالت ادغام شده می دهند.

اگر گره های ۳ و ۴ را درنظر بگیرید، گره ۴ تنها فرزند گره ۳ است و هر دوی آنها یعنی گره های ۳ و ۴ وابستگی کنترلی یکسانی دارند. هر دو به گره ۲ وابسته اند، لذا گره های ۳ و ۴ نیز می توانند یک رشته وظایف باشند. اما گره های ۴ و ۵ اگرچه گره ۵ تنها فرزند گره ۴ است ولی وابستگی کنترلی آنها یکسان نیست. بنابراین گره ۵ به این رشته افزوده نمی شود.

سوالی که مطرح می شود این است که چرا گره های ۵ و ۹ تشکیل یک رشت هوظایف را نمی دهند؟ درصورتیکه وابستگی بین گره های ۵ و ۹ را درنظر بگیرید، مشاهده می نمایید که تمام شرایط برای تشکیل یک رشته وظایف وجود دارد، اما در عمل امکان پذیر نیست. بنابراین باید الگوریتم بصورت زیر اصلاح شود:

دو گره در مجموع باید یک نوع وابستگی داشته باشند، بعبارت دیگر در مجموع به یک سری نقاط وابسته باشند. اگر توجه داشته باشید، گره های ۵ و ۹، به گره های ۴ و ۷ وابسته اند. برای گره ۵ وابستگی به گره ۴ و برای ۹ وابستگی به گره ای متفاوت وجود دارد. درصورتیکه گره های ۳ و ۴ هر دو به گره ۲ وابستگی دارند. اصولا مجموعه دستورالعمل هایی که یک وظیفه را تشخیص می دهند، وابستگی های داده ای به ابتدای آنها وارد شده و از انتهای آنها خارج می گردد. در اینجا در گره های ۹ و ۵ وابستگی داده ای باید به گره ۵ و یا به گره ۹ وارد شود. پس نمی توان آنها را یک رشته وظایف درنظر گرفت.

۴,۱۵ ادغام وظایف

جهت افزایش اندازه وظایف یا به عبارت دیگر افزایش دانه بندی، وظایف را با یکدیگر ادغام می کنند. باید تا اندازه ای که سربار ارتباط بیشتر از میزان موازی سازی در اجرای قطعات کد نشود، اندازه وظایف را افزایش داد. البته هر افزایشی به قیمت ازدست دادن توازی است. در اینجا تعداد پردازنده هایی که در دسترس هستند بسیار مطرح است.

درواقع باید دو گره ای را در یکدیگر ادغام نمود که حداقل میزان انتظار را نسبت به یکدیگر جهت اجرا داشته باشند. به عبارت دیگر وظیفه ترکیبی باید منتظر شود تا شرایط اجرایی وظایف تشکیل دهنده آن بر قرار شود. لذا، باید دو وظیفه ای را با یکدیگر ادغام نمود که اختلاف زمان آغاز آنها با یکدیگر حداقل باشد. به همین ترتیب تا زمانیکه دو وظیفه ادغام شده خاتمه نیابند گره های بعدی را نمی توان فعال نمود.

الگوریتم ادغام به شرح ذیل است.

Algorithm *Merge Grain*

1. **while** objective not met **do**
2. **for** each task T_i **do**
3. compute min and max Start times of T_i
   endfor
4. **for** each candidate task pair T_i and T_j **do**
5. P_{ij} = Penalty of combining T_i and T_j
   endfor
6. combine task pair T_i and T_j with the minimum Penalty P_{ij}
   endwhile

Penalty(T_1, T_2) = Max(min _start(T_1), min_start(T_2)) -

$$Min(max\_start(T_1) - exec\_time(T_2) , Max\_start(T_2) - exec\_time(T_1))$$

در اینجا min_start و max_start، زمان های حداقل و حداکثر شروع وظایف هستند. Penalty درواقع همان جریمه یا میزان انتظاری است که برای شروع وظایف بواسطه ادغام آنها در یک پردازنده تحمیل می شود. برای محاسبه پنالتی یا در واقع تاخیر زمانی که بخاطر برقراری شرایط جهت اجرا دو وظیفه ادغامی ایجاد می شود، نیاز است که ابتدا هر دو وظیفه که امکان ادغام شدن را دارند مشخص نمود. شرایط انتخاب به شرح ذیل است:

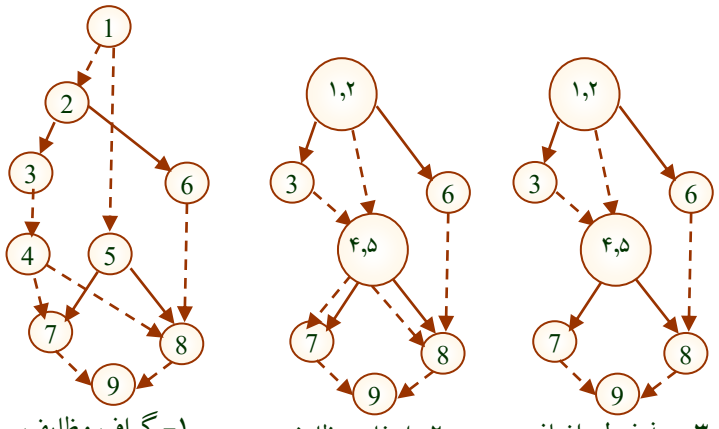
۱- دو گره ادغام شونده T_i و T_j می بایستی که وابستگی های کنترلی ورودی یکسان داشته باشند.

۲- در گراف وظایف نبایستی گره سومی مثل T_k در مسیر بین گره T_i و T_j موجود باشد.

۳- گره های T_i و T_j نباید هر دو با هم شامل شرایط کنترلی خروج شرطی باشند.

ممکن است موازی سازی وظایف مقرون بصرفه نباشد. درواقع سربار ارتباط بین پردازنده ها می تواند مشکل ساز باشد. لذا سعی می کنیم تا روشی ارائه دهیم که وظایف را درصورت امکان با یکدیگر ادغام نماییم. درواقع ادغام وظایف بواسطه افزایش دانه بندی برای پردازش موازی مشخص می کنند.

اصولا دو وظیفه ای را باید با هم ادغام نمود که اختلاف زمان آغاز آنها با یکدیگر حداقل باشد. ازطرفی دیگر تازمانیکه دو وظیفه ادغام شده خاتمه نیابند، گره های بعدی را نمی توان فعال کرد. همین امر را باید در نظر داشته تا اولا وظایف بدون تاخیر زیاد آغاز شوند. البته این ها بر روی یک پردازنده قرار می گیرند اما اگر زمان شروع یکی خیلی عقبتر از خاتمه دیگری باشد، مقرون بصرفه نیست که این ها را با هم ادغام نماییم. به همین ترتیب، زمان خاتمه ها را باید مدنظر داشت. برای نمونه در شکل زیر ادغام گره ها با در نظر گرفتن شرایط فوق مشخص شده است.



۱- گراف وظایف ۲- ادغام وظایف ۳- حذف لبه اضافی

شکل ۴۷: انتخاب وظایف ادغامی

همانطورکه مشاهده می کنید، بعد از ادغام وظایف ممکن است لبه های اضافی ایجاد شود که این لبه های اضافی دوباره باید حذف شود. البته بدلیل ادغام وظایف خیلی واضح نیست، چون زمان های اجرایی را در اینجا نداریم.

۴,۱۶ تبدیل گراف وظایف به کد HTGIL

زبان HTGIL یک نمایش سطح بالا برای گراف سلسله مراتبی برنامه به صورت متن است. این زبان، یک نمایش میانی بین مراحل تولید گراف وظایف برنامه و تولید کد است. بواسطه این زبان، گراف وظایف در قالبی خوانا و قابل تبدیل به کد اجرایی ارائه می شود. در واقع این زبان، ابزاری برای تبدیل گراف به متن است. در این زبان یک گراف وظایف در ساختار زیر گنجانده می شود. در گرامر ذیل براکت به مفهوم اختیاری بودن است.

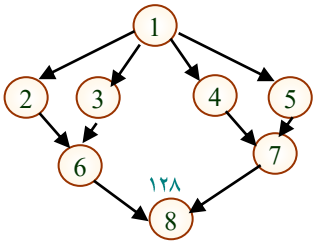
```
DAG
      [Task List]
ENDDAG
```

در داخل این ساختار، وظایف قرار می گیرد. هر وظیفه با کمک گرامر ذیل بیان شود:

```
Task Identifier ([dependent list])
            [statement list]
ENDTASK
```

با استفاده از پارامتر dependent list، لیست گره هایی از گراف مشخص می شود که در ارتباط با گره یا وظیفه با نام مشخص شده توسط پارامتر identifier هستند. این وابستگی ها ممکن است وابستگی داده ای به یک وظیفه دیگر و یا وابستگی کنترلی باشد. برای نمونه به شکل زیر توجه نمایید:

```
DAG
        TASK 1( )
         ENDTASK
        TASK 2(1)
         ENDTASK
        TASK 3(1)
         ENDTASK
        TASK 4(1)
         ENDTASK
        TASK 5(1 )
         ENDTASK
        TASK 6(2,3)
         ENDTASK
        TASK 7(4,5)
```



```
    ENDTASK
    TASK 8(6,7)
    ENDTASK
ENDDAG
```

شکل ۴۸: گراف کلی وظایف و کد HTGIL

TASK 8(6, 7) یعنی اینکه TASK 8 به TASKهای ۶ و ۷ وابسته است. در اینجا بدنه TASKها مشخص نیست.گراف فوق در واقع نمایانگر گراف وظایف برای یک برنامه ضرب ماتریسی برای اعداد موهومی است. فرض کنید X و Y دو ماتریس n×n از اعداد موهومی باشند.

$$X_{n\times n} = A_{n\times n} + j\, B_{n\times n}$$
$$Y_{n\times n} = C_{n\times n} + j\, D_{n\times n}$$

در اینجا A و B دو ماتریس n×n از اعداد صحیح هستند. به این ترتیب حاصل ضرب دو ماتریس Z = X * Y به صورت زیر محاسبه می شود:

$$Z = X * Y = (A + j * B) * (C + j * D) = (A * C - B * D) + j * (A * D + B * C)$$

به این ترتیب مشاهده می شود که ضرب ماتریسی شامل چهار ضرب ماتریس اعداد صحیح است. با استفاده از چهار تابع MATMUL[A, B, C, n]، MATSUB[A, B, C, n] و MATADD[A, B, C, n] که به ترتیب حاصل ضرب، تفاضل و مجموع دو ماتریس را بدست می آورند می توان گراف وظایف شکل فوق را به صورت ذیل در قالب جمله های زبان HTGIL بیان نمود. باید توجه داشته باشید که گره های شماره ۱ و شماره ۸ برای این گراف به ترتیب نمایانگر دو گره Start و Stop می باشند.

```
UBROUTINE MATCMUL(A, B, C, D, P, Q, N)

// declare the parameters

INTEGER n                                   //  matrices size
REAL A[n, n], B[n, n], C[n, n], D[n, n]     // input matrices
REAL P[n, n], Q[n, n]                       //  output matrices

// declare the local variables

REAL T[n, n], U[n, n], V[n, n], W[n, n]

DAG
    TASK 1( )
        // initialize local matrices
        T = 0.0
        U = 0.0
        V = 0.0
        W = 0.0
    ENDTASK
    TASK 2(1)
        CALL MATMUL[A, C, T, n]
    ENDTASK
    TASK 3(1)
        CALL MATMUL[B, D, U, n]
```

```
        ENDTASK
        TASK 4(1)
            CALL MATMUL[A, D, V, n]
        ENDTASK
        TASK 5(1)
            CALL MATMUL[B, C, W, n]
        ENDTASK
        TASK 6(2, 3)
            CALL MATSUB[T, U, P, n]
        ENDTASK
        TASK 7(4, 5)
            CALL MATADD[V, W, Q, n]
        ENDTASK
        TASK 8(6, 7)
ENDDAG
```

شکل ۴۹: کد HTGIL.

با استفاده از دستورالعمل های cobegin و coend می توان همزمانی در اجرای وظایف را مشخص نمود. انتظار برای رویداد a دستورالعمل wait(a) ایجاد می شود. به این ترتیب اجرای یک وظیفه تا زمانی که دستورالعمل post(a) توسط یک وظیفه دیگر به اجرا در نیامده است متوقف می ماند. با اجرا clear(a) کلیه اتفاقات مربوط به a یا به عبارت دیگر کلیه Post(a)ها پاک می شوند.

۴,۱۷ زمانبندی وظایف

زمانبندی هنگامی صورت می گیرد که وظایف مشخص و نیز دقیقا پردازنده ها و تعداد آنها معین باشند. می بایست با استفاده از روش های ایستا و براساس نوع و تعداد جملات داخل هر وظیفه، زمان اجرایی وظیفه را تخمین زد. به همین ترتیب باید با توجه به ارتباطات بین وظایف، زمان ارتباط را با سخت افزار مورد استفاده پیش بینی کرد.

بدینترتیب گراف وظایف به گرافی وزن دار تبدیل شده، و بصورت های ایستا و پویا می توان آن را زمانبندی نمود. برای اجرا یک وظیفه می بایست لااقل یکی از وابستگی های کنترلی و کلیه وابستگی های داده ای آن وظیفه ارضاء شوند. شرایط کنترلی لازم برای اجرا وظیفه x در قالب مجموعه گره هایی که X به آنها وابسته کنترلی است به صورت زیر در قالب مجموعه عناصر با وابستگی داده ای dPred و وابستگی کنترلی cPred به x و مجموعه عناصری مشخص می شود که x به آنها وابسته است.

$cPred(x) = \{x_i \in X / x_i \rightarrow x \in Ac\}$
$dPred(x) = \{x_i \in X / x_i \rightarrow x \in Ad\}$
$Pred(x) = cPred(x) \cup dPred(x)$
$cSucc(x) = \{x_i \in X / x \rightarrow x_i \in Ac\}$
$dSucc(x) = \{x_i \in X / x \rightarrow x_i \in Ad\}$
$Succ(x) = cSucc(x) \cup dSucc(x)$

مجموعه پرش هایی که گره x به آنها وابسته است در قالب مجموعه BranPred(x) و به صورت زیر مشخص می شود:

$BranPred(x) = \cup \{b \in B_i / x \rightarrow x_i \in Ac, \text{ labeled b and } i \in 1..n\}$

با استفاده از متغیر بولین ξn مشخص می کنندکه آیا وظیفه n به اجرا درآمده است؟ به همین ترتیب برای هر پرش در داخل گراف وظایف $b_i \in B_i$ متغیر بولین β_i با مقدار اولیه False تخصیص داده می شود. همچنین برای تعیین مجموعه گره های xi مجموعه $branNeg(b_i)$ مشخص می شود که با انجام پرش bi به اجرا درنخواهند آمد. این مجموعه از الگوریتم ذیل استفاده می کند. با استفاده از مجموعه $branNeg(b_i)$ در این الگوریتم مجموعه $Neg(x_i)$ که شاخص شاخه هایی را می توان محاسبه نمود که هنگامی که اجرا شوند گره x_i به اجرا در نمی آید.

```
set function BranNeg (bl)
{
   (* compute the BranNeg set of branch bl  *)
     B =  set to which bl belongs
   (* N will be the set of branches not taken *)
     N = B - bl
  (* S is the set of nodes bypassed  *)
    S = 0;
    do  {
           S' = S
          foreach (xi   ∈ X ) {
                C = branches on which xi is control dependent
                if (( C ⊆ N ) ∧ (C ≠  0)) {
                                        S = S + xi
                                        N = N – Bi } }

       } while (S' ≠ S )
return S  }
```

شکل ۵۰: الگوریتم زمانبندی

باید توجه نمایید که B_i نمایانگر مجموعه شاخه هایی است که از گره x_i خارج می شوند. بنابراین برای بدست آوردن مجموعه شاخه هایی که عبور از آنها موجب می شود که گره X به اجرا در نیاید، مجموعه Neg(x) را به صورت زیر می توان تعریف نمود:

$$Neg(x) = \{ b_i \in \cup B_i \ / \ x \in BranPred(b_i), \forall i \in 1..n \}$$

شرط اجرایی $\varepsilon(x)$ برای گره x به صورت ترکیب عطفی شرط کنترلی $\varepsilon_c(x)$ و شرط داده ای $\varepsilon_d(x)$ به صورت زیر تعریف می شود:

$$\varepsilon(x) = \varepsilon_c(x) \wedge \varepsilon_d(x)$$

برای محاسبه $\varepsilon_c(x)$ نی توان گفت که باید یکی از شاخه هایی اجرا شود که به x منتهی می گردد.

ازآنجاییکه زمانبندی گراف وظایف، مسئله ای از نوع NPhard می باشد. نمی توان این مسئله را با استفاده از روش های قطعی[1]حل نمود، لذا از الگوریتم های ژنتیک بدین منظور استفاده می کنند.

۴,۱۸ الگوریتم های ژنتیک

الگوریتم های ژنتیک اصولا روشی غیرقطعی مبتنی بر فرآیند تکامل ژنتیکی در موجودات مطرح شده است. اصل تکامل بیانگر بقای بهترین ها است. در پی تغییرات شرایط زیست محیطی تنها برخی از گونه های موجودات که شرایط مناسب محیط را داشته اند توانسته اند از طریق پدیده های ژنتیکی مثل تبادل متقاطع[2]و جهش[3]خود را با محیط وفق دهند بدین ترتیب گونه های جدید از موجودات به مرور جایگزین موجودات قبلی شده اند.

الگوریتم های ژنتیک سیر تکاملی موجودات را الگو قرار داده اند به قسمی از بین جماعتی از پاسخ ها که بعضا بصورت تصادفی[4]برای مسئله ای داده شده اند انتخاب شده و پاسخ ها مناسب تر را براساس اصل بقا بهترین ها را انتخاب می کند. انتخاب[5]براساس تابع هدف انجام می شود.

برای نمونه می خواهیم هر لبه زمان ارتباط گراف وظایف داده شده ای را که درواقع یک گراف وزن دار است را بین دو وظیفه مشخص می کنند. بر روی تعدادی مشخص از پردازنده ها به اجرا در آوریم. ابتدا بصورت تصادفی و با درنظرگرفتن نکاتی از قبیل ترتیب اجرای وظایف براساس وابستگی ها وظایف را به پردازنده ها تخصیص می دهیم. این تخصیص تصادفی است. حال جمعیتی از این پاسخ هایی که بصورت تصادفی انتخاب شده اند ایجاد می شود و از میان این پاسخ ها خوبتر ها را انتخاب می نماییم.

بدین منظور یک تابع هدف وجود دارد که کیفیت هر زمانبندی را مشخص می نماید.در نتیجه تعدادی کاندید برای ایجاد نسل جدید انتخاب می شود و از میان آنها تعدادی با استفاده از عملگر تبادل متقاطع که در تکثیر سلولی موجودات رایج است تکثیر می شود. تعدادی با درنظر گرفتن شرایط محیطی جهش داده می شوند و نسل جدید ایجاد می گردد. این عمل آنقدر تکرار می شود تا اینکه در یک مسیر تکاملی بهترین زمانبندی یا راه حل متبلور گردد. برای نمونه در ادمه یک زمانبندی ارائه می نماییم.

---

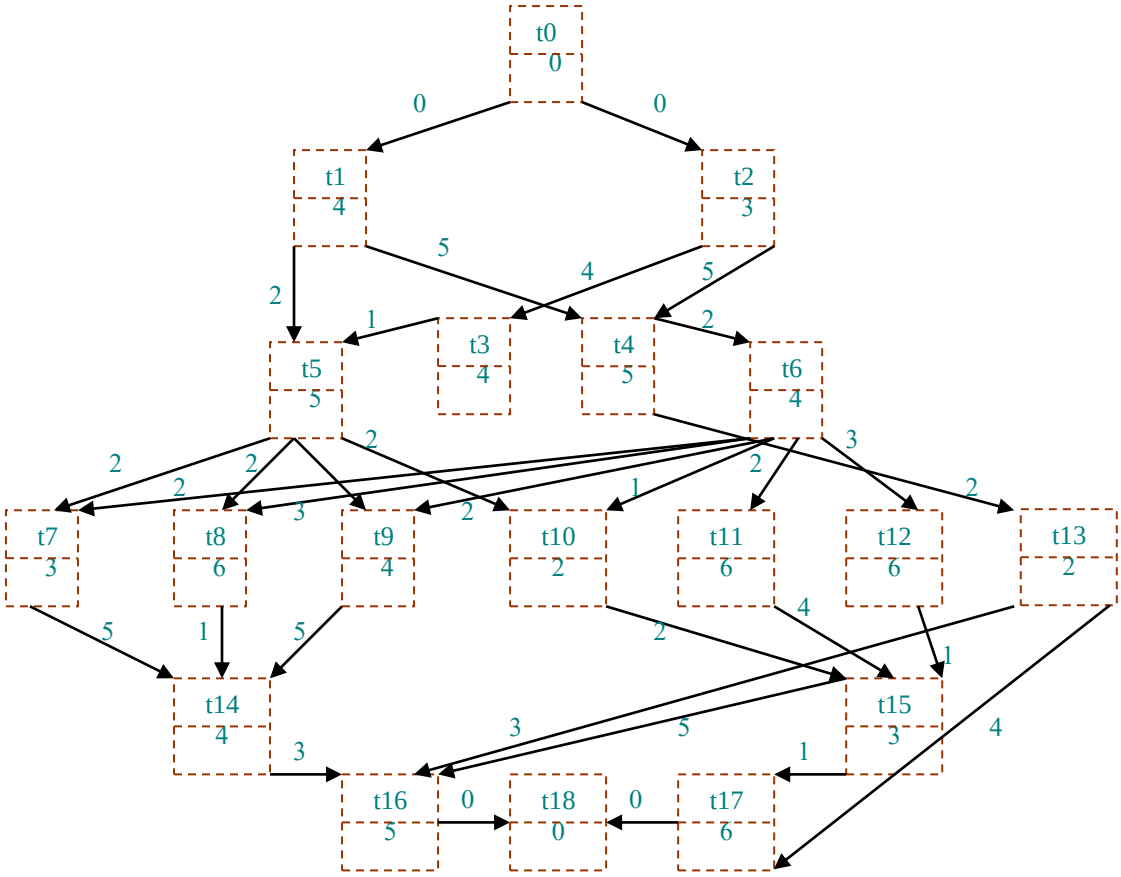
[1] - Detremenestic

[2] - Cross Over

[3] - Mutation

[4] - Random

[5] - Selection



شکل ۵۱: گراف وظایف

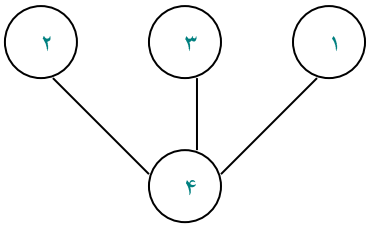
یک پاسخ انتخابی برای این مسئله در شکل زیر مشخص شده است. در این شکل وظایف بر روی پردازنده های مختلف زمانبندی گردیده است. تعداد پردازنده ها سه عدد می باشد و وظایف به ترتیب با درنظر گرفتن اینکه هیچگونه وظیفه قبلی برای آنها وجود ندارد که اجرا نشده باشد به پردازنده ها تخصیص داده شده اند.

هدف این است که تخصیص، بصورتی باشد که حداکثر موازی سازی در عملکرد همزمان پردازنده ها حاصل گردد تا بدینوسیله زمان اجرای برنامه که در قالب گراف وظایف مشخص شده را به حداقل ممکن برسد.

t0	t1	t2	t3	t4	t5	t8	t9	t10	T11	t7	t12	t14	t13	t15	t17	t16	t18
P0	P1	P2	P1	P2	P2	P2	P2	P1	P0	P0	P0	P0	P1	P0	P0	P0	P0

شکل ۵۲: نمونه ای از زمانبندی با سه پردازنده

درصورتیکه هر وظیفه بخواهد به پردازنده تخصیص داده شود حتما باید کار وظایف قبلی تخصیص شده به پردازنده تمام شده باشد.



شکل ۵۳: تخصیص وظیفه

باید کار[۱]۱، ۲ و ۳ با پردازنده تمام شود تا پردازنده به کار ۴ داده شود. مثال فوق در واقع پاسخی برای مسئله زمانبندی است. درواقع تبدیل گراف وظایف به رشته فوق را بوسیله عملگر Encoding انجام می دهند و بلعکس، از عملگر Encoding استفاده می شود.

در اینجا راه حل ارائه شده براساس اصطلاحات ژنتیکی کروموزوم گویند. کروموزوم ها در داخل هسته سلول ها قرار می گیرند. ساختار سلول بدین ترتیب است که دارای غشای سلولی بوده، در داخل آن مایعی به نام سیتوپلاسم قرار گرفته و در داخل سیتوپلاسم هسته قرار می گیرد. هسته دارای ۲۴ جفت کروموزوم می باشد. بر روی کروموزوم ها، ژن ها قرار گرفته اند و ژن ها مسئول انتقال خصایص موروثی هستند. بنابراین برای اینکه بتوانیم از روند تکامل در طبیعت استفاده نماییم، راه حل های مختلفی برای مسئله زمانبندی را در قالب کروموزوم تشبیه می نماییم. ژنها در اینجا زوج های کار و پردازنده می باشند.

اصولا در اولین گام می بایست جمعیت اولیه کروموزوم ها را ایجاد کرد. درواقع فرآیند تکاملی با یک جمعیت اولیه آغاز می شود. بنابراین باید ببینیم که چگونه می توان جمعیت اولیه را ایجاد کرد تا فرآیند تکاملی به بهترین وجهی انجام گیرد؟ بدین منظور دو نکته را درنظر می گیریم. درصورتیکه بتوانیم کروموزوم های اولیه را طوری انتخاب نماییم که جواب های نسبتا قابل قبول و خوبی ارائه دهد، مسلما فرآیند تکامل را تسریع نموده ایم. اما اصل تضاد را نباید فراموش کرد. لذا لازمه این جمعیت کروموزوم های بد هم هستند. کروموزوم هایی که بصورت تصادفی انتخاب شده اند.

بنابراین پیشنهاد می شودکه تعدادی از کروموزوم ها بصورت تصادفی ایجاد شوند و تعدادی دیگر با کیفیت خوب ایجاد شوند. برای این منظور با الگوریتمی که در ادامه ارائه می گردد سریعترین زمان شروع[۲]برای کارها در ابتدا محاسبه می شود. کارها بصورت نزولی براساس زودترین زمان شروع مرتب می شوند.

الگوریتم زیر، برای هر گره Ni مقدار Ttlevel(ni) که در واقع زودترین زمان آغاز برای Ni است را مشخص می کند. این الگوریتم همچنین گره هایی را که برروی طولانی ترین مسیر در داخل گراف وظایف قرار دارند را مشخص می کند. در واقع هدف این الگوریتم، به حداقل رساندن مسیر گراف است. مسلما زمان اجرای این گراف وظایف معادل با طولانی ترین زمان بلندترین مسیر در گراف است و هدف کوتاه کردن طولانی ترین مسیر می باشد.

---

[۱] - Task

[۲] - Earliest Start Time

Algorithm *Evaluate earliest start time and critical path*

1. **Initialize** length list of nodes and number of nodes on critical path as folows:
   LengthofCriticalPath := 0
   NodesonCriticalPath := 0
   NumberofNodesonCriticalPath := 0
2. **for** each node ni in v such that G = (V, E) **do**
3. Ttlevel(ni) := 0
4. NodesonLengthPathto(ni) := 0
5. NumberofNodesonLengthPathto(ni) := 0
   endfor
6. **for** each node ni in v such that G = (V, E) **do**
7. Ttlevel(ni) := 0
   endfor
8. **for** each node ni in v set the reference count equal to the number of its parents **do**
9. ReferenceCount(ni) := #parents(ni), ∀ ni ∈ v
   endfor
10. **for** each node with no parents then add it to ready list **do**
11. ReadyList := {ni ∈ v| ReferenceCount(ni) := 0}
    endfor
12. **while** the ReadyList is not empty **do**
13. select a node nk from the ReadyList, random list
14. **for** each child ni of node nk **do**
15. ReadyList := ReferenceCount(ni) – 1, ∀ ni ∈ v
16. **if** reference count of ni becomes zero then ni is ready to execute
17. **then** append ni to the ReadyList
    endif
    endfor

the earliest start time(EST) of ni, Ttlevel(ni), is computed as follow:

18. Ttlevel(ni) = max(Ttlevel(ni), Ttlevel(nk) + Execution(nk) + CommuniccationCost(nk, ni))
19. LengthofCriticalPath = max(LengthofCriticalPath, Ttlevel(ni) + ExecuteTime(ni))
20. **if** Ttlevel(ni) = Ttlevel(nk) + ExecutionTime(nk) + CommunicationCost(nk, ni)
21. **then** copy NodeonLongestPathto(nk) onto NodesonLongestPathto(ni)
22. NumberofNodesonLongestPathto(ni) = NumberofNodesonLongestPathto(ni) + 1
    endif
23. **if** Ttlevel(ni)>LengthofcriticalPath
24. **then** LengthofCriticalPath := Ttlevel(ni)
25. Copy the NodesonLongestPathto(ni) onto NodeonCriticalPath
26. Put the nodes ni at the end of NodesonCriticalPath
27. LengthofCriticalPath := Ttlevel(ni) + ExecutionTime(ni)
28. NumberofNodesoncriticalPath := NumberofNodesoncriticalPath + 1
    endif
    endwhile

شکل ۵۴: الگوریتم ارزیابی طولانی ترین زمان شروع و مسیر بحرانی

در الگوریتم فوق همانگونه که مشاهده می کنید، Referencecount برای هر وظیفه درواقع شاخص تعداد وظایفی است که آن وظیفه به آنها وابسته است. اما برای هر کروموزوم یا جوابی که با استفاده از الگوریتم فوق و یا بصورت تصادفی ایجاد شده میزان سازگاری[۱] می بایست محاسبه گردد.

[۱] - Fitness

درواقع سازگاری، شاخص کیفیت کروموزوم با درنظر گرفتن هدف الگوریتم می باشد. در اینجا هدف بدست آوردن پاسخ با کمترین زمان ممکن است. بعبارتی دیگر سازگاری برای هر کروموزوم زمان کامل شدن یا زمان اجرای کل گراف وظایف است که بوسیله کروموزوم مشخص شده است. این برابر با طول طولانی ترین مسیر در داخل گراف می باشد. برای اینکه این زمان را کاهش دهیم، سعی می نماییم تا وظایفی را به یک پردازنده تخصیص دهیم که به یکدیگر وابسته اند و بر روی مسیر بحرانی قراردارند. در شکل زیر الگوریتم مربوطه ارائه شده است. برای تعیین میزان سازگاری یا کیفیت کروموزوم ها ارائه می شود.

```
1. for each task, in a chromosome do
      set it reference count aqual to the number of immediate parents of the task in the task graph
      corresponding to the chromosome
2. for each processor Pi do
      set its local Timer, Si to Zero
3. set the global Timer S to Zero
4. starting with the leftmost task t in the chromosome
      repeat
         if reference count of t is not Zero
         then the chromosome is not acceptable
         let Pi be the processor to which t is assigned in the chromosome
         read the value of Timer, Si of Pi from the chromosome
         if S >= Si
         then the processoe is idle
              add the sum of S and execution time, t, of the task, t, to Si
              add one unit of time to the global Timer, S
         elseif S = Si then the processor has finished with the task
            then reduce one from the reference count of each child of the task
         take the next task t from the chromosome
      until all the tasks are scheduled
5. set fitness equal to the maximum value of the timer Si, of all processor, Pi
6. for each task ti, assigned to processor Pi in the chromosome do
      if processors of ti are not assigned to Pi
      then add the communication costs between the task and each of its predecessors to fitness
7. return fitness as the fitness of the chromosome
```

شکل ۵۵: الگوریتم محاسبه میزان سازگاری

بنابراین آموختیم که چگونه جمعیت اولیه ایجاد می شود و جمعیت اولیه تا حدود ۴۰ درصد بوسیله الگوریتم شکل 54 مشخص می گردد و ۶۰ درصد باقی بصورت تصادفی پردازنده ها را به کارها تخصیص می دهد. البته برای این منظور گراف وظایف بصورت سطح به سطح پیمایش می شود و کارهایی که در این سطح قرار می گیرند بطور تصادفی انتخاب و بطور تصادفی پردازنده ها به آنها تخصیص داده می شود.

آنهایی که در یک سطح باشند با هم پیمایش می شوند. در یک سطح تصادفی انتخاب شده و پردازنده ها بصورت تصادفی به آنها تخصیص می یابند. بدین ترتیب جمعیت اولیه ایجاد شده و در مرحله ابتدایی برای تولید نسل جدید کروموزوم ها انتخاب و با هم ادغام می گردند.

مسئله بدین ترتیب است که می خواهیم گراف وظایف خود را که نمایانگر چگونگی وابستگی بین قطعات کد است را بر روی پردازنده های موازی به اجرا درآوریم. هدف این است که در کوتاهترین زمان کل گراف وظایف که درواقع کل برنامه را نشان می دهد به اجرا درآید. راه حل این است که حداکثر همروندی یا توازی را در عملکرد پردازنده های موازی داشته باشیم.

بدین منظور از الگوریتم های زمانبندی استفاده می نماییم اما مشکلی داریم. برای زمانبندی باید مدت زمان لازم برای اجرای هر کار را مشخص کنیم. این به دو روش ایستا و پویا عمل می کنند. در روش ایستا به هر نوع جمله فراخوانی، if و غیره، وزنی داده می شود. بدین ترتیب براساس جمله هایی زمان آن تقریب زده می شود که درون یک کار قرار می گیرند. در روش پویا نمودارهایی با استفاده از تکنیک های گراف جریان و پیمایش کامل[1]زمان اجرایی وظایف را در عمل تقریب می زنند. اما از آنجاییکه مسئله زمانبندی گراف وظایف یک مسئله NPhard است. با روش هایی غیرقطعی مثل الگوریتم های ژنتیک مبادرت به حل مسئله می نماییم.

الگوریتم ژنتیک، در قالب کلی الگوریتم شکل ۵۵ مشخص شده است. این الگوریتم دارای دو گام اصلی است گام های شماره ۱ و ۲ ایجاد جمعیت اولیه را برعهده دارند. پس، الگوریتم های ژنتیکی با ایجاد جمعیتی از پاسخ هایی به مسئله آغاز می شود. یک چهارم از این جمعیت تصادفی انتخاب می شوند. این به دلیل ایجاد تضاد در نسل اول است. از کیفیت جواب ها یا در اصطلاح کروموزوم هایی که بصورت تصادفی انتخاب می شوند از قبل اطلاعی نداریم ممکن است بد یا خوب باشد. اما سعی داریم سه چهارم جمعیت اولیه را با پاسخ های نسبتا قابل قبول پر کنیم.

برطبق الگوریتم شکل ۵۵، برای هر وظیفه یا کار با درنظر گرفتن مسیرهای قابل دسترسی به آن در داخل گراف زودترین زمان آغاز وظیفه را مشخص کرده و سپس برای کل گراف آن را تعیین می نماییم. آنگاه وظایف را براساس مقدار زودترین زمان شروع مرتب تا برای وظایفی را که زودتر آغاز می شوند سریعتر پردازنده انتخاب نماییم.

معمولا روش کار پس از تشخیص زودترین زمان شروع بدین ترتیب است که وظایفی را که هیچ وظیفه قبلی برای آنها وجود ندارد به آن پردازنده ای تخصیص داده می شود و یا اصلا برای آن وظیفه ای وجود نداشته باشد این ها را در لیستی به نام Ready to schedule در الگوریتم قرار داده اند.

بعد از تخصیص پردازنده به این وظایف به ترتیب Reference Count با تعداد وظایفی که وظیفه های بعدی آنها به آن وابسته هستند را یک واحد کم می نماییم. اگر وظیفه ای Reference Count برای آن برابر صفر شد آن را به Ready list اضافه می کنیم.

برای تخصیص پردازنده ابتدا درنظر می گیریم که آیا وظیفه انتخاب شده برای طولانی ترین مسیر یا مسیر بحرانی قرار دارد، اگر قرار داشت وظیفه پدر این وظیفه را در مسیر بحرانی مشخص می نماییم و وظیفه را به همان پردازنده ای تخصیص می دهیم که کار یا وظیفه قبلی آن قرار است بر روی همان پردازنده اجرا شود.

---

[1] - White box testing

مزیت آن این است که هنگامیکه دو وظیفه بر روی یک پردازنده اجرا شوند زمان ارتباط برای آنها صفر می شود. اما چرا وظایف روی مسیر بحرانی است؟ علت این است که مسیر بحرانی بلندترین مسیر از لحاظ زمان است. با این تکنیک زمان اجرای مسیر بحرانی را تقلیل می دهیم. این تاثیر مستقیم در کوتاه شدن زمان اجرایی گراف وظایف دارد. اما اگر وظیفه بر روی مسیر بحرانی نباشد، آنرا به پردازنده ای تخصیص می دهیم که وظیفه قبلی آن که بیشترین زمان تبادل داخلی با آنرا دارد به آن تخصیص داده شده است. بدین ترتیب این زمان کاهش داده می شود و متن اولیه برای مثال به تعداد ۱۰۰ پاسخ طبق این الگوریتم انتخاب می شوند.

در مرحله ۱، جمعیت اولیه بدست آمده و در مرحله ۲ در داخل یک حلقه هر بار جمعیت جدیدی را ایجاد می کنیم. برای اینکار در مرحله ۲ در شکل ۵۵ جمله while termination criteria not satisfied do ظاهر شده است. درواقع این معیار خاتمه الگوریتم چند چیز می تواند باشد. برای مثال ۱۰۰ بار تکرار حلقه یعنی ایجاد نسل، چراکه هربار تکرار حلقه یک نسل جدید ایجاد می کند. البته زمانیکه جمعیت یکنواخت شد باز هم ممکن است به انتهای کار رسیده باشیم. معمولا در جمعیت ها به واسطه تضاد تکامل ایجاد می شود تا حدی که هم جمعیت جمعیت کاملی گردد یا تکامل یافته ای گردد. در اینصورت دیگر تکاملی ایجاد نمی شود (آنقدر جمعیت ها ادغام می شود که هم یکجور شوند و اگر جمعیت یکسان شد دیگر پیشرفتی نیست).

برای ایجاد نسل ها برطبق الگوریتمی که در شکل ۵۵ ارائه شده ، میزان کیفیت مشخص می شود. همچنین در ضمن پیمایش گراف بلندترین مسیر تا خاتمه گراف را مشخص می کند. بدین ترتیب برای هر جواب بدست آمده در داخل این الگوریتم بلندترین مسیر با درنظر گرفتن پردازنده هایی که به هر وظیفه تخصیص داده شده محاسبه می شود. توجه کنید که چگونه تخصیص وظایف به پردازنده ها در مدت زمان کل انجام وظایف گراف تاثیر مستقیم دارد.

در مرحله بعد، از بین این وظایف هر بار یک زوج انتخاب می شود. برای انتخاب والدین، یک روش این است که یک دور دایره را که ۴۰۰ رادیان است را به نسبت میزان عکس سازگاری یا مدت زمان اجرایی در کروموزوم های نسل موجود تقسیم می کنیم. مسلما کروموزومی که زمان لازم برای اجرای آن کمتر باشد قطاع بزرگتری را بخود تخصیص می دهد. حال عددی را بصورت تصادفی بین صفر تا ۴۰۰ رادیان انتخاب می کنیم. بر روی قطاع مربوط به هر کروموزومی که افتاد آن کروموزوم را انتخاب می کنیم برای انجام تبادل متقاطع که درواقع توسط آن والدین انتخاب شده با یکدیگر ادغام می شوند تا فرزندان جدید ایجاد گردند.

بدین ترتیب عمل می کنیم که طبق شکل 55 دو نقطه را بصورت تصادفی در دو کروموزوم والد انتخاب شده مشخص می کنیم. اکنون در فاصله این دو نقطه در داخل کروموزوم ها به جلو می رویم و پردازنده ها را با یکدیگر جایگزین می نماییم. برای مثال در شکل 25 t11 به پردازنده P1 در Parent1 تخصیص داده شده وظیفه t5 در Parent2 به P2 تخصیص داده شد. بدین ترتیب نسل جدید ایجاد می شود.

اما ۸۰ درصد یا درصدی از کروموزوم ها را برای تبادل متقاطع انتخاب می کنند چراکه در دنیای واقعی نیز همگی فرزند ایجاد نمی کنند اما هنگامی که کروموزم ها یکسان می شوند دیگر تکاملی ایجاد نمی شود. بدین منظور در جمعیت جهش ایجاد می نماییم. جهش درواقع ممکن است منجر به سرطان و یا پدیده ای مثبت شود.

الگوریتم های ژنتیکی باید Tune شوند. برای مثال تعداد نسل، اندازه جمعیت، احتمال تبادل متقاطع و احتمال جهش، این ها اعدادی هستند که بصورت تصادفی انتخاب می شوند.