به نام خداوند بخشنده مهربان

چگونه یک کامپایلر بنویسیم ؟

محمد رضا ذاکری نسب

زمستان ۱۳۸۰

مقدمه :

درس « اصول طراحی کامپایلرها » یکی از درس‌های اصلی دوره کارشناسی کامپیوتر در گرایش نرم‌افزار است. این درس در دانشگاه‌های سراسر جهان از دیرباز تدریس می‌شود و دلیل این توجه و اقبال عمومی نیز وابستگی شدید بخش نرم‌افزاری دانش کامپیوتر به این مقوله است. همان‌طور که می‌دانید توانایی یک مهندس نرم‌افزار وابسته به قدرت ابزارهایی است که به کار می‌برد و یکی از ابزارهای اولیه و اساسی، یک زبان برنامه‌نویسی قدرتمند است. در اینجاست که کامپایلرها به عنوان یکی از اجزاء اصلی و پایه‌های مهم علم نرم‌افزار مطرح می‌شوند. یکی دیگر از خصوصیات بارز این زمینه فضای گسترده‌ای است که برای نوآوری، خلاقیت و تلاش وجود دارد. بنا بر تجربه شخصی من و سایر دوستانم در دانشکده کامپیوتر دانشگاه علم و صنعت ایران، نوعی جذابیت خاص در زمینه کامپایلرها وجود دارد که شاید نتوان این جذابیت را در زمینه‌های دیگر مشاهده کرد.

در این مقاله ما ابتدا به بررسی مراحل نوشتن یک کامپایلر ساده برای تولید کد Jasmin می‌پردازیم و سپس تلاش می‌کنیم تا برای کد تولیدشده یک بهینه‌ساز (Optimizer) بنویسیم. انتخاب کد شبه اسمبلی Java (Jasmin) و

تمرکز پروژه روی آن به دلیل قابلیتهای ویژه ماشین مجازی Java (مانند قابلیت حمل کد و عدم وابستگی به سیستم سختافزاری) صورت پذیرفته است. امیدوارم این برگ سبز بتواند چراغی اگرچه کمسو در راه علم اندوزان سرزمین اسلامی ما، ایران بزرگ باشد. لازم به ذکر است که این پروژه بر مبنای آموختههای کلاسی دروس « اصول طراحی کامپایلرها » و « کامپایلرهای پیشرفته » ارائه شده توسط « دکتر سعید پارسا » میباشد. بنابراین ضمن تقدیر و تشکر از پشتیبانیها و زحمات بیشائبه ایشان، به علاقهمندان آموختن علم کامپایلرها مطالعه « جزوه کلاسی درس اصول طراحی کامپایلرها » که دربرگیرنده مطالب کلاسی ایشان است، توصیه میگردد.

۱- مراحل نوشتن یک کامپایلر

برای نوشتن یک کامپایلر شما باید مراحل زیر را به ترتیب انجام دهید (ترتیب ذکر شده برای تسهیل در نوشتن کد است ولی اجباری برای رعایت آن وجود ندارد) :

۱-۱- نوشتن یک تحلیلگر لغوی (Lexical Analyzer).

۱-۲- ایجاد یک جدول LR1 و یا LALR1 برای تشخیص صحت برنامههایی که قرار است بر اساس گرامر شما نوشته شوند.

۱-۳- نوشتن یک تجزیهگر (Parser).

۱-٤- نوشتن تابعی به نام MakeNode برای تولید درخت خلاصه نحوی (Abstract Syntax Tree) از روی کد برنامه.

۱-٥- نوشتن تابعی به نام GenCode که درخت ایجاد شده در مرحله قبل را به عنوان ورودی میگیرد و خروجی آن کد سطح پایین معادل برنامه است. در این قسمت میتوان برای خروجی تابع GenCode از کد اسمبلی مبتنی بر پردازندههای Intel استفاده

کرد که ما به جای این کد، از کد شبه اسمبلی ماشین مجازی Java (Jasmin) استفاده کرده‌ایم.

۱-۱- تحلیلگر لغوی

در ابتدای کار ما باید یک تحلیلگر لغوی برای گرامر در نظر گرفته‌شده بنویسیم. برای این کار ما در ابتدا یک ماشین خودکار (Automata) برای تحلیلگر خود ایجاد می‌کنیم و سپس این ماشین خودکار را با استفاده از یک ساختار انتخاب (مانند case در Delphi و یا switch در C) پیاده‌سازی می‌کنیم. در این مرحله باید به ازای هر یک از حالت‌های موجود در ماشین خودکار (که به صورت گره‌های آن ماشین نمایش داده می‌شوند) کد مربوط به آن نوشته شود. برای مثال فرض کنید شما می‌خواهید یک تحلیلگر لغوی ساده برای اعدادی که به صورت صحیح و یا اعشاری می‌باشند و با یک فضای خالی (blank) از هم جدا شده‌اند، بنویسید. این تحلیلگر را می‌توان در مراحل زیر پیاده‌سازی کرد (البته راه‌حل‌های متفاوت دیگری نیز برای این مسئله وجود دارد) :

۱ - ابتدا ببینیم تحلیلگر لغوی ما چه چیزی را باید بخواند؟ ما در ابتدا باید هر تعداد رقم که می‌بینیم بخوانیم تا به یک کاراکتر غیر رقم برسیم. اگر این کاراکتر blank بود، نوع عدد را integer در نظر می‌گیریم و عدد خوانده‌شده را ذخیره می‌کنیم. اگر این کاراکتر نقطه (point) بود، نوع عدد را double در نظر می‌گیریم و عدد را ذخیره می‌کنیم که در این صورت در ادامه قبل از رسیدن به blank یا EOF حق دیدن یک نقطه (point) دیگر را نداریم (زیرا عددی مانند 32.25.63 غیر معتبر است). با دیدن علامت EOF کار تحلیلگر لغوی پایان می‌یابد.

۲ - حالا ما می‌توانیم با توجه به توضیحات بالا ماشین خودکار مربوط به این تحلیلگر لغوی را رسم کنیم :utomata1.gif همان‌طور که می‌بینید این ماشین خودکار، مراحل مربوط به تشخیص یک عدد را نشان می‌دهد. شایان ذکر است که در این ماشین فرض شده است ورودی دارای کاراکترهای غیرمجاز نمی‌باشد. کد مربوط به این ماشین درون یک حلقه و تا زمانی که کاراکتر خوانده‌شده مخالف EOF باشد ادامه می‌یابد. این ماشین به شما کمک می‌کند تا کد تحلیلگر نحوی را به مراتب سریع‌تر به پایان برسانید.

۳ - یکی دیگر از کارهایی که شما باید در تحلیلگر نحوی انجام دهید این است که ساختاری برای ذخیره کردن اطلاعات مربوط به لغات خوانده‌شده از کد منبع برنامه (source) در نظر بگیرید. این ساختار باید قابلیت نگهداری تمام اطلاعات مورد نیاز را داشته باشد. بخشی از این اطلاعات (مانند خود لغت خوانده‌شده) در فرآیند ایجاد کد سطح پایین مورد نیاز هستند و برخی دیگر (مانند شماره سطر و ستون لغت خوانده‌شده) در مواردی مانند اعلان خطا به کاربر مورد استفاده قرار می‌گیرند. برای مثال، برای ذخیره لغات ماشین اعداد بالا می‌توان از ساختاری به شکل زیر استفاده کرد:

```
type Token = record

  value: string;
  position: integer;
  numtype: TmyTypes;

end;
```

البته ساختاری که در پروژه ما استفاده شده است، مقداری با این ساختارمتفاوت است. این ساختار در یک unit به نام TypeDefinitions قراردارد.

٤ - بعد از این مراحل، شما باید کد مربوط به هر حالت (state) ماشین خودکار را بنویسید و سپس آن را در یک حلقه قرار دهید که تحلیلگر لغوی شما تا رسیدن به علامت EOF اجرای این کد را ادامه دهد و تمام لغات موجود در source را بخواند. برای مثال، در ماشین خودکار اعداد کد شما می‌تواند به صورت زیر باشد :

```
case state of
        0: Lexeme := Lexeme + NextCh;
           Read( input, NextCh);
           if NextCh = chr(26) then state := 0
           else if NextCh in ['0'..'9'] then state := 1
           else if NextCh = '.' then state := 2
           else exception.raise('Error in source file.');
        1: .....
        2: ....
```

خوانندگان عزیز برای درک عملکرد کلی یک تحلیلگر لغوی می توانند به بخش Educational Items -> Lexical Analyzing – Step by Step در برنامه Jacomizer مراجعه کنند.

۱-۲- ایجاد یک جدول LR1 و یا LALR1 :

برای ایجاد یک جدول LR1 و یا LALR1 شما می‌توانید به روشهای مختلفی اقدام کنید. شما می‌توانید این جدول را به طور دستی ایجاد کنید. اما به علت تعداد حالات زیادی که در یک جدول LR1 و یا LALR1 ایجاد می‌شود، این کار برای گرامرهایی که حداکثر ۲۰ یا ۲۵ قانون دارند، امکان‌پذیر است. این گرامرها فقط می‌توانند یک زبان بسیار ساده را مدل‌سازی کنند. راه دوم استفاده از برنامه‌هایی است که با گرفتن گرامر، جدول LR1 یا LALR1 را برای شما ایجاد می‌کنند. یکی از این برنامه‌ها برنامه LR1 Maker است که توسط آقای مجید رودی (از دانشجویان دانشکده کامپیوتر دانشگاه علم و صنعت

ایران) نوشته شده‌است. مزیت اصلی این برنامه رابط کاربر بسیار ساده آن و همچنین فرم ورودی قابل‌فهم آن می‌باشد. همچنین خروجی این برنامه به صورتی است که می‌توانید از آن مستقیما در برنامه‌تان استفاده کنید. از دیگر برنامه هایی که در این زمینه وجود دارند، می‌توان به برنامه‌هایی اشاره کرد که علاوه بر تولید جدول، قسمت‌هایی دیگر از کامپایلر شما را تولید می‌کنند، از جمله YACC ، JLEX و Parser Generator .

همانطور که گفته‌شد، یکی از این برنامه‌ها Parser Generator است که ما در پروژه خود از این محصول استفاده کرده‌ایم. این برنامه گرامر ورودی را با پسوند y.* می‌گیرد و در پرونده‌ای با پسوند v.* و با یک قالب گزارش‌گونه جدول مربوط به آن را ایجاد می‌کند. چون گرامر ما در قوانین مربوط به else دارای اختلال در تصمیم‌گیری برای خواندن یا اعمال قانون (shift-reduce conflict) بود، ابتدا این اختلال به صورت دستی رفع شد و سپس چون قالب گزارش‌گونه خروجی این برنامه برای استفاده مناسب نیست، یک برنامه مبدل نوشته شد تا خروجی را به قالبی مناسب تبدیل کند. شما می‌توانید مراحل این کار را در زیر ببینید :

۱- ورودی برنامه Parser Generator (grammar.y).
۲- خروجی برنامه Parser Generator (grammar.v).
۳- خروجی برنامه Parser Generator پس از حذف قوانین از ابتدای آن و رفع مشکل قوانین مربوط به else (Parsed.txt).
٤- خروجی برنامه مبدل (ParsedLALR1.txt).

۱-۳- نوشتن یک تجزیه‌گر (Parser):

حالا ما تجزیه‌گر لغوی و جدول LALR1 خود را داریم. بنابراین می‌توانیم نوشتن تجزیه‌گر مورد نیاز را شروع کنیم. این تجزیه‌گر وظیفه دارد

صحت برنامه‌هایی را که بر اساس گرامر داده‌شده به آن نوشته‌شده‌اند، بررسی کند. در این مرحله تجزیه‌گر شما باید لغات خوانده شده توسط تحلیلگر لغوی را یک به یک بخواند و سپس بر اساس نوع لغت خوانده‌شده و شماره مرحله (state) ای که هم‌اکنون در آن قرار دارد، یکی از عملیات Shift ، Reduce و یا Goto را انجام دهد. این عملیات در جدول LALR1 برنامه ما (ParsedLALR1.txt) به ترتیب با استفاده از حروف S ، R ، و G مشخص شده‌اند. در جلوی هرکدام از این حروف یک عدد آمده است که یک شماره مرحله است، ولی در هر مورد معنای متفاوتی دارد. عدد بعد از حرف S یعنی « یک لغت دیگر بخوان و به حالت شماره # برو ». عدد بعد از حرف G یعنی « به حالت شماره # برو » و بعد از حرف R یعنی « طبق قانون شماره # یک عملیات کاهش انجام بده و سپس به حالتی برو که در مرحله قبل در جدول دیده‌ای ».

اجازه دهید این عملیات را با یک مثال ساده شرح دهیم. فرض کنید ما در ابتدای برنامه هستیم (بنابراین شماره مرحله جاری ما صفر است یا state = 0). حالا کلمه "program" را از خروجی تحلیلگر لغوی می‌گیریم (بنابراین نماد جاری ما S_Program است یا S_Program = CurrentSymbol). در جدول LALR1 در سطر مربوط به حالت صفر (state 0) به دنبال نماد S_Program می‌گردیم و می‌بینیم فرضا دستورالعمل S9 نوشته شده است (البته در جدول LALR1 ما S1 به این حالت اختصاص یافته است). بنابراین لغت بعدی را که یک شناسه (identifier) است می‌خوانیم و به حالت ۹ می‌رویم. این روند تا زمانی که تمام برنامه طبق یک قانون کاهش پیدا کند (لیست قوانین مورد استفاده را می‌توانید در grammar.y ببینید) ادامه می‌یابد. برای مطالعه بیشتر در مورد چگونگی اعمال قوانین می‌توانید به « جزوه درسی دکتر سعید پارسا » و یا یکی از کتابهای Compilers (نوشته Ullman و دیگران) و Modern Compiler Implementation in Java (نوشته Andrew W.Appel) مراجعه کنید.

توجه داشته باشید که تجزیه‌گر شما نیاز به یک پشته دارد. هر زمان که شما یک لغت را می‌خوانید، باید آن لغت و شماره حالتی را که آن لغت دیده شده را در پشته قرار دهید. به علاوه برای هر قانون شما باید بدانید که چند لغت باید از بالای پشته برداشته شوند تا آن قانون اعمال شود. ما این پشته را GenStack نامیده‌ایم و از متغیری به نام GenStackCount نیز برای نگه‌داشتن مکان بالای پشته استفاده کرده‌ایم. به علاوه برای اینکه از مشکلاتی که در اثر کار با یک پشته خالی به‌وجود می‌آیند اجتناب کنیم، پایین پشته را با "$$$$" علامت زده‌ایم.

لطفا در نوشتن تجزیه‌گر دقت لازم را مبذول کنید. تجزیه‌گر یکی از مهم‌ترین بخش های کامپایلر شماست. اگر شما بتوانید یک تجزیه‌گر دقیق و خوب بنویسید، از بسیاری از مشکلات بعدی جلوگیری کرده‌اید.

شما می‌توانید کد تجزیه‌گر Jacomizer را در unit ای به نام Parser ببینید. مراجعه به بخش Educational Items -> Parsing – Step by Step در برنامه Jacomizer هم می‌تواند کمک مؤثری در درک چگونگی کار با پشته تجزیه‌گر به شما بکند.

۱-٤-ایجاد درخت خلاصه نحوی (Abstract Syntax Tree) :

پس از آماده شدن تجزیه‌گر، حالا می‌توانیم کد مورد نیاز برای تولید درخت خلاصه نحوی را ایجاد کنیم. همانطور که قبلا هم گفته شد برای تولید گره‌های این درخت از تابعی به نام MakeNode استفاده می‌کنیم. شما می‌توانید کد این تابع و تابع دیگری به نام MakeLeaf را در unit ای به نام ASTBuilder ببینید. از تابع MakeLeaf برای تولید برگ‌های درخت استفاده شده است. همانطور که می‌بینید مقدار بازگشتی این دو تابع از نوع Pnode می‌باشد. اگر به پرونده TypeDefinitions مراجعه کنید، در آنجا ساختاری به نام ASTNode و بعد یک نوع اشاره‌گر به این ساختار به نام Pnode تعریف شده است. بنابراین نوع بازگشتی این دو تابع یک اشاره‌گر به یک گره درخت خلاصه

نحوی است. Pnode کلید اصلی تولید درخت خلاصه نحوی است. آسان‌ترین راه برای تولید این درخت، تولید آن همگام با تجزیه برنامه توسط تجزیه‌گر (Parser) است. روش کار هم به این صورت است که هر زمان شما با یکی از قوانین کاهش (Reduce) برخورد کردید، تابع مربوط به آن قانون را فراخوانی می‌کنید. این تابع با استفاده مناسب از توابع MakeNode و MakeLeaf نتیجه عملیات کاهش را بر روی درخت خلاصه نحوی نیز اعمال می‌کند. برای مشاهده روند کار می‌توانید به متد ParseProgram در unit Parser و همچنین به unit ای به نام RulesFunctions مراجعه کنید. همان‌طور که می‌بینید در اینجا از آرایه ای از function pointer ها (اشاره‌گرهایی به توابع مربوط به قوانین) استفاده شده است. این روش به شما اجازه می‌دهد تا عمل تولید درخت را همگام با عمل تجزیه انجام دهید که این مساله می‌تواند در بالابردن سرعت کامپایلر شما بسیار مؤثر باشد.

لطفا یک‌بار دیگر به RulesFunctions مراجعه کنید. همانطور که می‌بینید ما در اینجا ۹۵ تابع داریم که با نام‌های Rule1 تا Rule95 نامگذاری شده‌اند. هر کدام از این توابع در زمان اجرای قانون کاهش مربوط به خود باید فراخوانی شوند. برای آسانتر شدن دسترسی به این قوانین و اجتناب از به کاربردن یک دستور انتخاب برای ۹۵ گزینه ممکن (مثلا یک دستور case با ۹۵ خط انتخاب) در TypeDefinitions یک نوع داده‌ای به نام Tfunc تعریف شده است که یک اشاره‌گر به این توابع است و سپس در RulesFunctions یک آرایه ۹۵ تایی از این نوع با توابع مربوطه مقداردهی شده است. حالا برای فراخوانی قانون مربوط به ‘R23’ می‌توان به جای فراخوانی Rule23 از FuncArray[23] استفاده کرد .

حالا اجازه دهید با هم یکی از توابع مربوط به قوانین را بررسی کنیم. فرضا قانون زیر را در نظر بگیرید :

identifiers ➔ identifier virgule identifiers

برای این قانون تابعی به شکل زیر باید نوشته شود :

```
function Rule# ( Lexeme: string; Nodes: Tarray ): PNode;
begin
   result := MakeNode ( S_none, ntIds, Nodes[0], Nodes[1] );
end;
```

پارامترهایی که به این تابع فرستاده می‌شوند، یک رشته حرفی (مربوط به گره پدر) و گره‌های فرزندی هستند که باید به گره پدر متصل شوند. برای مثال فرض کنید می‌خواهیم برای قطعه کد زیر درخت را تولید کنیم و تا به حال b و c و d را در یک گره از نوع ntIds (که مخفف non-Terminal node, type: Identifiers می‌باشد) ذخیره کرده‌ایم و a نیز در یک گره از نوع ntId ذخیره شده است. حالا به شیوه فراخوانی و پارامترهای ارسالی به تابع مربوط به این قانون دقت کنید :

```
FuncArray [ RuleNumber ] ( " " , Nodes[0], Nodes[1] );
```

که در اینجا Nodes[0] اشاره‌گری به گره ntId و Nodes[1] اشاره‌گری به گره ntIds می‌باشد.

دقت کنید که اگر چه توابع MakeNode و MakeLeaf توابعی بسیار ساده هستند، اما با درست به کاربردن همین توابع ساده شما می‌توانید درخت خلاصه نحوی برنامه های پیچیده را نیز به سادگی بسازید. پیشنهاد می‌کنم برای بیشتر آشناشدن با شیوه نوشتن توابع مربوط به قوانین، چند قانون و توابع مربوط به آنها را در RulesFunctions بخوانید.

توجه :

در برنامه Jacomizer امکانی پیش‌بینی شده است که به شما اجازه می‌دهد درخت حاصل از برنامه‌تان را ببینید. این کار با استفاده از یک جزء سازنده به نام TtreeView (Delphi 5.0) انجام پذیرفته است. دقت کنید که اگر چه این مساله ابزار مناسبی برای آموزش می‌باشد، اما بر خلاف امکانات موجود در منوی Educational Items با هدف آموزش

انجام نپذیرفته است. **شما باید درخت تجزیه خود را ببینید**. بنابراین هنگام تولید درخت، حتما ابزاری برای دیدن درخت خود طراحی کنید. البته برای ساده‌تر شدن کار می‌توانید از اجزاء سازنده دیگری مانند Memo و یا ListBox نیز استفاده کنید.

پس از نوشتن توابع مربوط به قوانین و ایجاد درخت خلاصه نحوی با استفاده از این توابع، حالا می‌توانیم با استفاده از این درخت کد نهایی برنامه را تولید کنیم.

۱-۵- نوشتن تابع تولید کد

پس از ایجاد درخت خلاصه نحوی نوبت به تولید کد بر اساس این درخت می‌رسد. اگر شما قصد دارید کد نهایی خود را بر اساس در این مرحله باید عوامل زیادی مورد توجه قرارگیرند که در اینجا سعی می‌کنیم مهم‌ترین آنها را توضیح دهیم :

- کلاس Register Manager : اگر شما به یکی از کتابهای آموزشی کامپایلر که براساس تولید کد اسمبلی پردازنده‌های Intel نوشته شده است مراجعه کنید، در بخش تولید درخت خلاصه نحوی با مفهومی به نام الگوریتم برچسب‌گذاری (Labeling Algorithm) برخورد خواهید کرد. این الگوریتم به هر گره درخت خلاصه نحوی شما عددی اختصاص می‌دهد که بیانگر تعداد ثباتهای مورد نیاز برای پردازش آن گره و تمام زیرگره‌های آن است. البته در کامپایلر ما چون به جای اسمبلی Intel از کد شبه اسمبلی ماشین مجازی Java استفاده شده است و از آنجا که در ماشین مجازی Java مفهومی به نام ثبات (Register) وجود ندارد، ما از این موضوع صرف‌نظر کردیم. ولی اگر شما قصد تولید کد اسمبلی Intel را دارید برای اطلاع از چگونگی برچسب‌گذاری درخت و قوانین استفاده از برچسب‌ها هنگام ایجاد کد حتما به یکی از مراجع

« جزوه درسی دکتر سعید پارسا » و یا کتاب Compilers (نوشته Ullman و دیگران) مراجعه کنید. یک نمونه پروژه انجام‌شده درباره این مساله پروژه درس کامپایلر آقای سعید صدیقیان است که با استفاده از زبان Delphi 5.0 نوشته شده است و دارای یک Register Manager کامل و قوی می‌باشد.

◄◄ مدیریت پشته (Stack Management) : همانطور که در بالا ذکر شد، تولید کد بر اساس زبان شبه اسمبلی ماشین مجازی Java (Jasmin) نیازی به استفاده از ثبات‌ها (Registers) ندارد. دلیل این مساله ساختار مبتنی بر پشته ماشین مجازی Java است. بدین معنی که تمام عملیات مورد نیاز در این ماشین مجازی با استفاده از یک پشته صورت می‌پذیرد. به عنوان یک مقایسه ابتدایی دو قطعه کد زیر را که مربوط به اسمبلی Intel و زبان Jasmin می‌باشند در نظر بگیرید:

Assembly

```
Mov ax,5
Mov bx,6
Add ax,bx

; now result is stored
  in ax
```

Jasmin

```
bipush 5
bipush 6
iadd

; now result is stored
  on top of the stack
```

همانطور که در قطعه کد مربوط به Jasmin می‌بینید، در ابتدا با استفاده از دو دستور bipush 5 و bipush 6 دو عددی را که باید جمع زده شوند در بالای پشته ذخیره می‌شوند. سپس با فراخوانی دستور iadd پارامترهای مورد نیاز از بالای پشته برداشته شده و حاصل‌جمع آن بالای پشته قرار می‌گیرد. برای کسب اطلاعات بیشتر می‌توانید به « جزوه ضمیمه درس اصول طراحی کامپایلرها ، Jasmin » مراجعه کنید.

⏪ بدنه کد مورد نیاز برای یک برنامه Jasmin : از آنجا که برنامه‌هایی کـه بـه زبان Jasmin نوشته می‌شوند باید پس از کامپایل توسط کامپایلر Jasmin بـه یک پرونده class.* تبدیل شوند، شما نیاز دارید که یک بدنه کـد پیش‌فـرض را در خروجی تابع GenCode خود داشــته باشـید. ایـن بدنـه کـد پیش‌فـرض بـه صورت زیر است :

```
.class public YourProgramName
.super java/lang/Object

; specify the constructor method for the example class

.method public <init>()V
     ; just calls object constructor
     aload_0
     invokespecial java/lang/Object/<init>()V
     return
.end method

; specify the main method

.method public static main([Ljava/lang/String;)V

     ...
     ; your code must be placed here
     ...

     return
.end method
```

⏪ بدنه کـد مـورد نیـاز بـرای یـک Applet : یکـی از قابلیتـهای زیبـای برنامـه Jacomizer امکـان ایجـاد Applet هـای وب اسـت. ایـن برنامـه‌ها نـیز پـس از کامپایل توسط کامپایلر بـه یـک پرونـده class.* تبدیـل می‌شـوند. سـپس ایـن پرونده class.* درون بدنه یک پرونده html.* مورد ارجاع قـرار مـی‌گـیرد. در ایـن قسـمت می‌توانیـد کـد بدنـه پیش‌فـرض یـک Applet در زبـــان Jasmin و همچنین کد بدنه مورد نیاز برای پرونده html.* را ببینید:

```
.class public YourAppletName
```

```
.super java/applet/Applet

; paint() method - redraws applet window
.method public paint(Ljava/awt/Graphics;)V

     aload_1        ; Graphics object
     ; ...
     ; your code must be placed here
     ; ...

     return
.end method

; standard constructor - just calls Applet's constructor
.method public <init>()V
     aload_0
     invokespecial java/applet/Applet/<init>()V
     return
.end method
```

همانطور که می‌بینید، در این کد پیش‌فرض متد paint() به جای متد main() استفاده شده است. این کد پس از کامپایل پرونده‌ای به نام YourAppletName.class ایجاد می‌کند که در اصل حاوی یک Applet خالی است. شما می‌توانید از آن به صورت زیر در یک پرونده *.html استفاده کنید :

```
<HTML>
<HEAD>
<TITLE>My First Applet ( Empty )</TITLE>
</HEAD>
<BODY>
<applet code="YourAppletName.class" width=300 height=200>
</applet>
</BODY>
</HTML>
```

خاطرنشان می‌شود که امکان ایجاد اتوماتیک Applet و همچنین تست آن با استفاده از یک پرونده *.html در برنامه Jacomizer پیش‌بینی شده است.

چگونه یک بهینه‌ساز بنویسیم ؟

(Java Bytecode Peephole Optimizer)

مقدمه:

روش بهینه‌سازی Peephole روشی است که در آن برای بهینه‌سازی کد، در هر لحظه فقط قسمت نسبتا کوچکی از کد برنامه مورد بررسی قرار می‌گیرد. با این روش می‌توان یک برنامه را از روی کد سطح پایین آن (مانند کد اسمبلی و یا کد jasmin) و یا حتی شبه کد آن (pseudo code) بهینه‌سازی کرد. برای این کار از قوانین ساده‌ای استفاده می‌شود که این قوانین قطعه‌ای از کد را به کد دیگری تبدیل می‌کنند که همان کار را انجام می‌دهد ولی دارای حجم کمتر و یا سرعت بیشتر است.

از سالها پیش و از زمانی که از پردازنده‌های اولیه خانواده ۸۰۸۶ در رایانه‌ها استفاده می‌شد، به دلیل اهمیت فوق‌العاده زمان CPU و مقدار حافظه مصرفی برنامه، به صورت گسترده‌ای از بهینه‌سازهای Peephole استفاده شده است. در این پروژه ما می‌خواهیم یک بهینه ساز Peephole برای بایت کد

java بنویسیم که قابلیت تشخیص خودکار Applet های Java و تست Applet بهینه‌شده را نیز داشته باشد. دلایل عمده ما برای این کار عبارتند از:

۱-با از بین بردن کد مرده (dead code) و جایگزین کردن دستورات طولانی و یا تکراری با دستورات معادل که حجم کمتری دارند، می‌توان به فایل‌های class.* کوچکتری دست یافت که هم سریعتر از روی اینترنت بارگذاری می‌شوند و هم سریعتر توسط JVM پردازش و اجرا می‌شوند.
۲-با جایگزین کردن دستورات زمان‌بر با دستورات سریعتر سرعت اجرای کد افزایش می‌یابد.
۳-چون این بهینه‌سازی بر روی بایت کد صورت می‌پذیرد بهره سرعت به دست آمده روی هر سیستمی قابل مشاهده است، یعنی شما برای مشاهده افزایش سرعت نیازی به یک کامپایلر just-in-time (JIT) ندارید.
٤-تغییرات اعمال شده بر روی کد شما کار decompiler ها را سخت‌تر می‌کند و این به معنای امنیت بیشتر برای کد شما می باشد.
٥-تکنیک‌های به‌کاررفته در این پروژه به شما اجازه می‌دهند که هر فایل class.* را بدون اینکه به سورس آن دسترسی داشته باشید، بهینه کنید.

ابزارهای مورد نیاز:

برای شروع عملیات بهینه‌سازی بر روی بایت کد جاوا شما نیاز به چند ابزار اولیه دارید. ابزارهای آماده‌ای که در این پروژه استفاده شده‌اند عبارتند از:

۱–D-Java : اولین ابزار مورد نیاز برنامه‌ای است که یک فایل کلاس جاوا را به صورت قابل خواندن درآورد. یکی از کامل‌ترین و بهترین ابزارهایی که هم‌اکنون در این زمینه وجود دارد و غیرتجاری است، D-Java نام دارد. این نرم‌افزار در اصل به زبان C و برای Unix نوشته شده است اما یک نسخه تحت DOS آن نیز وجود دارد. البته چند باگ هم در این نرم‌افزار گزارش شده است. مشخصات این نرم‌افزار به صورت زیر است:

Name:	D-Java
Programmer:	Shawn Silverman (umsilve1@cc.umanitoba.ca)
Download:	http://www.cat.nyu.edu/meyer/jvm/djava/ (D-Java.exe, 43kb)

۲– Jasmin : دومین ابزاری که شما نیاز دارید، برنامه‌ای است که با کمک آن بتوانید کد بهینه شده را به حالت اول آن (بایت کد جاوا) برگردانید. در این پروژه از اسمبلر Jasmin برای این کار استفاده شده است. این اسمبلر توسط آقای Meyer نوشته شده است و در اصل یکی از مواد کمک‌آموزشی کتاب Java Virtual Machine ایشان می‌باشد. از آنجا که D-Java می‌تواند خروجی کد جاسمین تولید کند، یک ایده خوب بهینه‌سازی کد جاسمین به دست آمده و سپس تبدیل آن به یک فایل *.class با استفاده از اسمبلر جاسمین است. تنها ایراد گزارش شده در اسمبلر جاسمین عدم پشتیبانی این اسمبلر از کاراکترهای Unicode است که اگر فایل کلاس شما در ابتدا دارای کاراکترهای Unicode بوده است، استفاده از این بهینه‌ساز موجب ایجاد اشکال خواهد شد. مشخصات اسمبلر جاسمین در جدول زیر آورده شده است:

Name:	Jasmin
Programmer:	Jon Meyer
Download:	http://mrl.nyu.edu/meyer/jvm/jasmin.html

(jasmin.zip, 584kb)

بهینه ساز:

پس از فراهم آوردن ابزارهای اولیه، نیاز به گـردآوری ترکیبـاتی اسـت که می‌توانند برای بهینه‌سازی مورد استفاده قرار بگیرند. در ایـن بهینه‌سـاز از ۸۶ ترکیب استفاده شده است که برای دیدن لیست کامل قوانیـن می‌توانیـد بـه فایل RulesEng.txt مراجعه کنید. کد مربوط به این بهینه‌ساز در دو پرونده بـا نام‌های Parser و Optimizer ذخـیره شده‌اسـت. در اینجـا چنـد مـورد از ایـن قوانین را به صورت تفصیلی مورد بررسی قرار می‌دهیم:

۱-push کردن مقادیر ثابت مساوی در بالای پشته:

کد زیر را که به زبان جاوا نوشته شده است در نظر بگیرید:

```
public class test {
      public void f(int i, int j) {
      }

      public void g() {
            f(146, 146) ;
      }
}
```

اگر این کد را ابتدا کامپایل کرده و سپس با استفاده از D-Java به کد جاسـمین برگردانید، فراخوانی ( ) f به صورت زیر در می‌آید:

```
aload_0
sipush 146
sipush 146
invokevirtual test/f(II)V
```

همان‌طور که می‌بینید در اینجـا عملیـات push دوبـار بـرای یـک مقـدار مساوی انجام گرفته است. به جای این کار می‌توان با یک دستور push و یـک دستور dup همین کار را انجام داد:

```
aload_0
sipush 146
dup
invokevirtual test/f(II)V
```

برای بیان کردن این جایگزینی از قانونی به شکل زیر استفاده می‌شود:

```
sipush %1
sipush %1
=
sipush %1
dup
```

با این جایگزینی دوبایت از حجم فایل class.* نهایی کم می‌شود. در مورد سرعت هم نتایج زیر با استفاده از یک سیستم با پردازنده Pentium 133MHz و JDK 1.0.2 و سیستم عامل Linux 2.0.33 به دست آمده‌اند. در این تست تابع ( ) f یک میلیون بار فراخوانی شده است:

مقدار ثابت	فضای صرفه‌جویی شده	زمان اجرا (قبل از بهینه‌سازی، ms)	زمان اجرا (بعد از بهینه‌سازی، ms)
١	٠ بایت	٣١١	٣٦٢
٦	١ بایت	٤٠٧	٤٠٩
١٤٦	٢ بایت	٤٨٦	٤٤٦

دقت کنید که در بعضی از موارد (مانند عدد ١) این بهینه‌سازی نتیجه خوبی ندارد. البته این مساله ربطی به تکنیک به کار رفته ندارد و دلیل آن این است که ماشین مجازی جاوا برای تعدادی از اعمال ابتدایی (مانند push کردن اعداد ٠ تا ٩) از مکانیزم‌های ویژه‌ای استفاده می‌کند که سرعت را بالا می‌برند. بنابراین در اینجا همانطور که می‌بینید استفاده از dup زمان عملیات را بالا برده است.

٢-push کردن متغیرهای یکسان در بالای پشته:

کد زیر را که به زبان جاوا نوشته شده است در نظر بگیرید:

```
public class test {
       int k = 0 ;

       public void f(int i, int j) {
       }

       public void g() {
           f(k, k);
       }
   }
```

اگر این کد را ابتدا کامپایل کرده و سپس با استفاده از D-Java به کد جاسمین برگردانید، فراخوانی f ( ) به صورت زیر در می‌آید:

```
aload_0
aload_0
getfield test/k I
aload_0
getfield test/k I
invokevirtual test/f(II)V
```

همان‌طور که می‌بینید در اینجا دوبار مقدار متغیر k بازیابی شده است. به جای این کار می‌توان با یک دستور getfield و یک دستور dup همین کار را انجام داد:

```
aload_0
aload_0
getfield test/k I
dup
invokevirtual test/f(II)V
```

برای بیان کردن این جایگزینی از قانونی به شکل زیر استفاده می‌شود:

```
aload%1
getfield %2 I
aload%1
getfield %2 I
=
aload%1
getfield %2 I
dup
```

نتایج زیر از یک میلیون بار فراخوانی تابع f ( ) به دست آمده اند:

مقدار ثابت	فضای صرفه‌جویی شده	زمان اجرا (قبل از بهینه‌سازی، ms)	زمان اجرا (بعد از بهینه‌سازی، ms)
۰ (int)	۳ بایت	۱۲۷۲	۸۱۲
۱۵ (int)	٤ بایت	۱۳۸۰	۸۷۱
۰ (long)	۳ بایت	۱۳٤۹	۸٤۲
۱۵ (long)	٤ بایت	۱۹۷۳	۹۲۱

۳- بازیابی مقدار یک متغیر بلافاصله پس از ذخیره:

کد زیر را در نظر بگیرید:

```
public class test {

        public void g() {
            int i1, i2, i3, i4 ;

            i1 = i2 = i3 = 0 ;

            i4 = 9 ;
            i1 = i4;
        }
    }
```

اگر این کد را ابتدا کامپایل کرده و سپس با استفاده از D-Java به کد جاسمین برگردانید، دو خط آخر کد به صورت زیر در می‌آیند:

```
bipush 9
istore 4
iload 4
istore_1
```

همان‌طور که می‌بینید در اینجا بلافاصله پس از ذخیره ٤ مقدار آن بازیابی شده است. می‌توان همین نتیجه را با استفاده مناسب از یک دستور dup به دست آورد:

```
bipush 9
dup
istore 4
istore_1
```

برای بیان کردن این جایگزینی از قانونی به شکل زیر استفاده می‌شود:

```
istore %1
iload %1
=
dup
istore %1
```

نتایج زیر از یک میلیون بار اجرای دو خط آخر تابع ( ) g به دست آمده اند:

دستورات	فضای صرفه‌جویی شده	زمان اجرا (قبل از بهینه‌سازی، ms)	زمان اجرا (بعد از بهینه‌سازی، ms)
istore/iload	۱ بایت	٤٤٥	٤٣٩
fstore/fload	۱ بایت	٤٤٣	٤٣٥
astore/aload	۱ بایت	٤٨٦	٤٣٩
lstore/lload	۱ بایت	٤٥٣	٤٩٩
dstore/dload	۱ بایت	٥٠٠	٤٩٩

۴- بهینه سازی عبارات منطقی :

کد زیر را در نظر بگیرید:

```
public class test {
       public boolean b ;

       public void f() {
       }

       public void g() {
           if ( b == true )
               f() ;

           if ( b )
               f() ;

           if ( b == false )
               f() ;

           if ( !b )
               f() ;
       }
   }
```

اگر دقت کنید، در ایـن کـد سـاختارهای (b == false) و !b وهمچنیـن (b == true) و b به کاررفته‌اند. اگر چه این ساختارها از لحاظ منطقی هم معنا هستند، اما کدی که کامپایلر جاوا برای آنها تولید می‌کند متفاوت است:

```
b == true

aload_0
getfield test/b Z
iconst_1
if_icmpne Label1
```

```
b

aload_0
getfield test/b Z
ifeq Label2
```

```
b == false

aload_0
getfield test/b Z
iconst_0
if_icmpne Label3
```

```
!b

aload_0
getfield test/b Z
ifne Label4
```

بنابراین ما می‌توانیم با استفاده از قوانین زیر کد مربـوط بـه دو حـالت سمت چپ را بهینه کنیم:

```
getfield %1 Z
iconst_1
if_icmpne %2
=
getfield %1 Z
ifeq %2

getfield %1 Z
iconst_0
if_icmpne %2
=
getfield %1 Z
ifne %2
```

چگونه از این بهینه ساز استفاده کنیم؟

همانطور که قبلا گفته شد هدف از نوشتن بهینه ساز، بهینه کردن کد یک فایل *.class جاوا بوده است. در این بهینه‌ساز شما می‌توانید یک فـایل *.class

جاوا و یا یک برنامه به زبان جاسمین را باز کنید و پس از انجام عمل بهینه سازی، خروجی را در پایین صفحه ببینید.این خروجی خود به صورت یک برنامه به زبان جاسمین است، اما شما می‌توانید وابسته به نیازتان آن را به صورت فایل جاسمین و یا یک فایل class.* جاوا ضبط کنید.

کد کامل پروژه

این پروژه شامل unit های زیر می‌باشد:

1. About.pas (About Form)
2. ASTBuilder.pas
3. GenCodeUnit.pas
4. Htable.pas
5. LexemeAnalyser.pas
6. Optimizer.pas
7. Parser.pas (main unit, contains a lot of code related to other sections rather than parser)
8. RulesFunctions.pas
9. TypeDefinitions.pas
10. Unit2.pas (Form)
11. Unit3.pas (Form)
12. Unit4.pas (Form)

هرگونه پیشنهاد خود درباره این مستندات و یا خود پروژه

را با ما در میان بگذارید.

rzakery2001@yahoo.com

اجرای Jacomizer