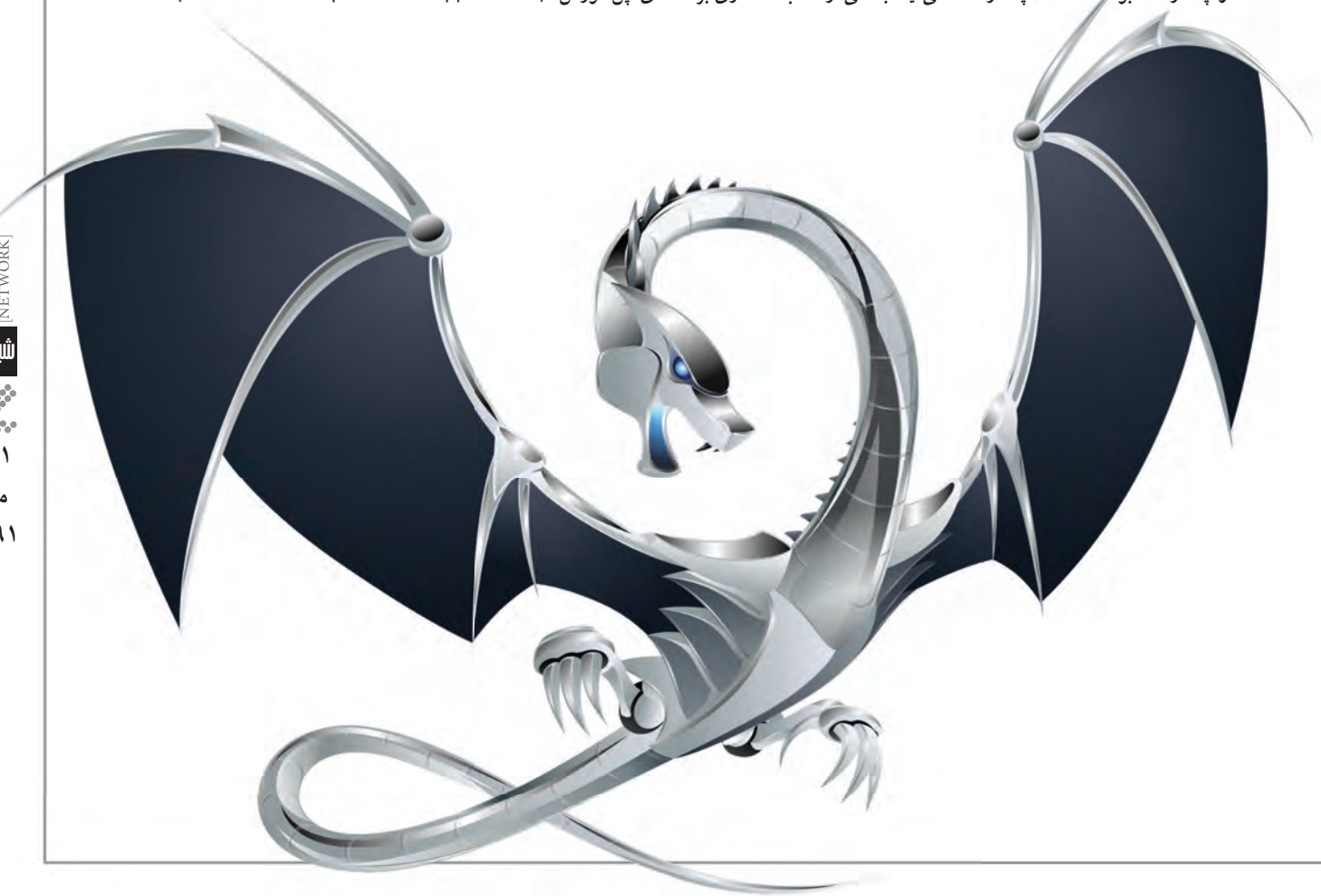
نگاهی ژرف به طراحی، ساختار و مزایای LLVM

بخش نخست

» نویسنده: کریس لاتنر » منبع: دکتر دابز » ترجمه: احمد شریف‌پور

تولید و توسعه کامپایلرها یکی از پیچیده‌ترین و سنگین‌ترین فرآیندهای برنامه‌نویسی است. پیاده‌سازی صحیح یک کامپایلر مستلزم داشتن درکی عمیق از پارادایم‌های برنامه‌نویسی، شناخت سخت‌افزار و نحوه کارکرد ماشین‌ها و همین‌طور توانایی و استعدادی زیاد در حوزه ریاضیات و منطق است. به همین دلیل، تعداد کامپایلرهای موفق و مطرح، به ویژه آن‌هایی که با هدف گرفتن پلتفرم‌های مختلف توانسته‌اند به جامعه کاربری گسترده‌ای دست یابند، چندان زیاد نیست. طبیعی است که توسعه و نگه‌داری چنین پروژه‌هایی نیز همواره بر عهده شرکت‌ها و نهادهای بزرگ و قدرتمند نرم‌افزاری بوده است.

پیچیدگی کامپایلرها از یک سو و وابستگی اجزای مختلف آن از سوی دیگر باعث شده است تا دستکاری، ایجاد تغییر و مشارکت در توسعه کامپایلرها (حتی در نمونه‌های آزادی نظیر GCC) تنها محدود به درصد اندکی از توسعه‌دهندگان باشد. علاوه بر این، یکپارچگی کامپایلر، امکان بازنویسی و استفاده دوباره از یک کامپایلر برای پشتیبانی از زبان‌های برنامه‌نویسی جدید یا پلتفرم‌های سخت‌افزاری تازه را دشوار و گاهی ناممکن می‌سازد. اینجا است که طراحی ماجولار کامپایلری به‌نسبت جوان با نام LLVM مزیت‌هایی را به دنبال خواهد داشت که می‌تواند توسعه و بهینه‌سازی کامپایلرها را ساده‌تر کرده و دایره مشارکت‌کنندگان و استفاده‌کنندگان آن را گسترده‌تر سازد. پروژه LLVM مجموعه‌ای از فناوری‌های کامپایلر و زنجیره ابزارهای توسعه ماجولار است که برخلاف نامش، ارتباط چندانی با ماشین‌های مجازی ندارد. این کامپایلر به زبان C++ و برای بهینه‌سازی (Optimization) کدهای زبان ماشین در زمان کامپایل، زمان لینک کردن و حتی زمان اجرا نوشته شده است. این پروژه در سال ۲۰۰۰ در دانشگاه ایلی‌نویز و به رهبری ویکرام ادوه (Vikram Adve) و کریس لاتنر (Chris Lattner) کلید خورد و هدف اصلی آن ایجاد زیرساختی برای انجام تحقیقات در زمینه کامپایل دینامیک بود. این پروژه از ابتدا زیرپوشش مجوز اپن‌سورس دانشگاه ایلی‌نویز که شبیه مجوزهای BSD است، برای استفاده عموم عرضه شده است. اگرچه LLVM به زبان C++و در ابتدا برای خانواده زبان‌های C نوشته شده بود، اما ذات مستقل و ماجولار این کامپایلر که به زبان خاصی وابسته نبود، زمینه را برای ساخت Frontendهای متنوع آماده ساخته و امکان استفاده از LLVM برای کامپایل زبان‌های فراوانی از جمله آدا، هسکل، پایتون، روبی، فرترن و... روی پلتفرم‌های سخت‌افزاری مختلف را نیز فراهم کرد. برای درک تأثیر و اهمیت این پروژه کافی است در نظر داشته باشیم که شرکت اپل در سال ۲۰۰۵، لاتنر را به استخدام خود درآورد و گروهی را برای کار روی LLVM و استفاده از آن در توسعه نرم‌افزارهای اپل تشکیل داد. از آن زمان، LLVM یکی از اجزای اصلی ابزارهای توسعه، چه در سکوی Mac OSX و چه در iOS بوده است. آن‌چه در ادامه می‌آید، بخشی از کتاب «معماری برنامه‌های اپن‌سورس» (Architecture of Open Source Applications)است.



طی پنج سال گذشته، LLVM از پروژه‌ای آکادمیک به Back-end جهانی کامپایلرهای C، C++ و Objective C تبدیل شده است. کلید این موفقیت، کارایی، انعطاف‌پذیری و سازگاری آن است که هر دوی این خصوصیات از طراحی و پیاده‌سازی منحصربه‌فرد آن نشأت گرفته است.

پروژه LLVM چتری است که مجموعه‌ای از بخش‌های مرتبط و سطح پایین زنجیره ابزارهای توسعه (نظیر اسمبلرها، کامپایلرها، دیباگرها و...) را در سایه خود میزبانی کرده و توسعه می‌دهد. این ابزارها به‌گونه‌ای طراحی شده‌اند که با ابزارهای کنونی و موجود مورد استفاده در سکوی یونیکس سازگار باشند. نام LLVM زمانی سرنام «Low Level Virtual Machine» بود، اما اکنون به برندی برای پروژه مادر تبدیل شده است.

در حالی‌که LLVM ویژگی‌های منحصربه‌فردی را عرضه می‌کند و به‌واسطه استفاده گسترده از برخی ابزارهایش (نظیر Clang که کامپایلری برای C ،C++ و Objective C است و نسبت به GCC مزیت‌های خاص خود را دارد) شناخته می‌شود، اما مهم‌ترین وجه تمایز آن با سایر کامپایلرها، معماری درونی آن است.

از شروع پروژه در دسامبر ۲۰۰۰ میلادی، LLVM به صورت مجموعه‌ای از کتابخانه‌ها با قابلیت استفاده دوباره و رابط‌هایی (Interface) کاملاً تعریف شده، طراحی شد. در آن زمان پیاده‌سازی‌های اپن‌سورس زبان‌های برنامه‌نویسی، به‌عنوان یک ابزار تک منظوره طراحی می‌شدند.

به عنوان مثال، استفاده دوباره از parser یک کامپایلر استاتیک نظیر GCC، برای انجام تحلیل‌های آماری و refactoring بسیار دشوار بود. در حالی‌که زبان‌های اسکریپتی معمولاً راهی را برای تعبیه مفسر و کتابخانه‌های زمان اجرای‌شان در نرم‌افزارهای بزرگتر فراهم می‌کردند، این کتابخانه‌های زمان اجرا معمولاً قطعه‌ای حجیم و یکپارچه از کد بود که می‌توانست در نرم‌افزار قرار داده شده یا از آن حذف شود. راهی برای استفاده دوباره از قسمت‌های مختلف آن‌ها وجود نداشت و اشتراکات اندکی میان پیاده‌سازی‌های مختلف یک زبان دیده می‌شد.

فراتر از چیدمان اجزای خود کامپایلر، جامعه کاربری شکل گرفته پیرامون پیاده‌سازی‌های محبوب زبان‌ها نیز معمولاً به شدت دو قطبی بود. یک پیاده‌سازی خاص، یا یک کامپایلر استاتیک مانند GCC، Free Pascal یا Free BASIC را عرضه می‌کرد یا به عرضه یک کامپایلر زمان اجرا (در قالب یک مفسر یا کامپایلر JIT) می‌پرداخت. پیاده‌سازی‌هایی که هر دوی این قابلیت‌ها را عرضه کنند، به ندرت دیده می‌شدند و در صورت وجود، اشتراک اندکی میان کدهای نوشته شده برای این دو وضعیت وجود داشت.

در ده سال اخیر اما، LLVM این چشم‌انداز را از اساس دگرگون کرده است. اکنون از LLVM به عنوان زیرساختی برای پیاده‌سازی گستره وسیعی از کامپایلرهای استاتیک و زمان‌اجرا استفاده می‌شود. این گستره وسیع، خانواده زبان‌هایی را که توسط GCC، جاوا، دات‌نت، پایتون، روبی، اسکیم، هسکل و D پشتیبانی می‌شوند و همین‌طور بسیاری از زبان‌های کمتر شناخته‌شده دیگر را شامل می‌شود.

کامپایلر LLVM همچنین جایگزین بسیاری از کامپایلرهای با استفاده خاص نیز شده است. از میان این کامپایلرها می‌توان به موتور بهینه‌سازی زمان اجرای OpenGL در سیستم‌عامل اپل، کتابخانه‌های پردازش تصویر و پشته‌ها در After Effects ادوبی اشاره کرد. درنهایت LLVM برای تولید محصولات جدید و متنوعی نیز مورد استفاده قرار گرفته است که شناخته‌شده‌ترین آن‌ها شاید OpenCL و کتابخانه‌های زمان اجرای آن باشد.

آشنایی با طراحی کلاسیک کامپایلرها

معمول‌ترین طراحی برای یک کامپایلر استاتیک سنتی (مانند بیشتر کامپایلرهای C) یک طراحی سه مرحله‌ای است که اجزای اصلی آن Frontend، بخش بهینه‌سازی یا Optimizer و Backend هستند. (شکل ۱) بخش Frontend وظیفه parse کردن کد منبع، کنترل آن برای خطاها و ایرادات و ساخت یک «درخت نحوی انتزاعی» («AST» سرنام Abstract Syntax Tree که نمایشی انتزاعی از کد ورودی است) را برعهده دارد. در برخی شرایط خود AST برای بهینه‌سازی در بخش Optimizer به فرم جدیدی تبدیل می‌شود و Optimizer و Backend این کد را اجرا می‌کنند.(شکل ۱)

وظیفه Optimizer این است که در راستای بهبود کارایی کد در زمان اجرا، تغییرات متنوعی را در آن به وجود آورد؛ مثلاً محاسبات اضافی را حذف کند. این قسمت کم‌وبیش از زبان مورد استفاده و معماری ماشین مقصد مستقل است. پس از آن، قسمت Backend (که به مولد کد یا Code Generator نیز مشهور است) کد را به مجموعه دستورالعمل‌های ماشین مقصد نگاشت می‌کند. علاوه بر ایجاد کد صحیح و بدون خطا، ایجاد کدهای خوب و سریع نیز بر عهده این قسمت است. به این معنی که قسمت Backend باید بتواند از قابلیت‌های اختصاصی معماری موردنظر بهره برده و کدهای بهینه‌تری ایجاد کند. قسمت‌های معمول Backend یک کامپایلر، بخش‌های انتخاب دستورالعمل، تخصیص ثبات (Register) و زمان‌بندی دستورالعمل‌ها هستند. این مدل، درست به همین شکل برای مفسرها و کامپایلرهای JIT نیز صادق است. ماشین مجازی جاوا (JVM) نیز یکی از انواع پیاده‌سازی‌های این مدل است که از بایت‌کدهای جاوا به عنوان رابطی (Interface) میان Frontend و Optimizer بهره می‌برد.

پیاده‌سازی این طراحی

مهم‌ترین برگ برنده این طراحی کلاسیک، زمانی رو می‌شود که یک کامپایلر، بخواهد چندین زبان برنامه‌نویسی یا چندین معماری مقصد را پشتیبانی کند. همان‌طور که در شکل ۲ مشاهده می‌کنید، در چنین صورتی اگر کامپایلر از یک نمایش یا نحوه ارائه یکسان در Optimizer استفاده کند، می‌توان برای هر یک از زبان‌های موردنظر یک Frontend جداگانه نوشت که کدهای آن زبان را به فرم مورد استفاده در Optimizer تبدیل کند و پس از آن برای هر معماری موردنظر نیز

شکل ۱ بخش‌های اصلی یک کامپایلر سه بخشی

یک Backend جداگانه ایجاد کرد تا حاصل کار Optimizer را به کدهای آن معماری ترجمه کند. (شکل ۲)

با این سیستم طراحی، پورت‌کردن یک کامپایلر برای پشتیبانی از یک زبان جدید (مثل Algol یا BASIC) تنها به پیاده‌سازی یک Frontend جدید نیاز دارد، اما Optimizer و Backend موجود به همان شکل قابل استفاده هستند. اگر این بخش‌ها از هم جدا نشده بودند، کار با یک زبان برنامه‌نویسی جدید مستلزم ایجاد همه چیز از صفر بود و به این ترتیب برای پشتیبانی از n زبان برنامه‌نویسی و m معماری سخت‌افزاری به m*n کامپایلر نیاز بود.

یکی دیگر از مزیت‌های طراحی سه مرحله‌ای (که به‌طور مستقیم نتیجه قابلیت هدف‌گیری چندین معماری مختلف است) این است که کامپایلر می‌تواند در خدمت مجموعه وسیع‌تری از برنامه‌نویسان باشد و نه تنها برنامه‌نویسان یک زبان خاص با معماری مقصد یکسان. برای یک پروژه اپن‌سورس، این به معنای جامعه بزرگتری از مشارکت‌کنندگان احتمالی است که خود باعث بهبود یافتن و پیشرفت سریع‌تر کامپایلر خواهد شد.

درست به همین دلیل است که کامپایلرهای اپن‌سورس (نظیر GCC) که توسط جوامع کاربری فراوانی مورد استفاده قرار می‌گیرند، معمولاً نسبت به کامپایلرهای خاص‌تر (نظیر Free Pascal) کدهای ماشین بهینه‌تری تولید می‌کنند. اما این قضیه در مورد کامپایلرهای غیر اپن‌سورس اختصاصی صحت ندارد. کیفیت چنین کامپایلرهایی مستقیماً به بودجه پروژه وابسته خواهد بود. به عنوان مثال کامپایلر ICC اینتل (هرچند طیف مخاطبان اندکی دارد) به واسطه کیفیت کدی که تولید می‌کند، بسیار مشهور شده است.

در نهایت، یکی از مهم‌ترین برتری‌های طراحی سه‌مرحله‌ای این است که مهارت‌های لازم برای ایجاد Frontend با مهارت‌های مورد نیاز برای ایجاد Optimizer و Backend کاملاً متفاوت هستند. با جداسازی این بخش‌ها، نگه‌داری و ارتقای بخش Frontend برای برنامه‌نویسان آن قسمت ساده‌تر خواهد بود. اگرچه این موردی اجتماعی به شمار می‌رود، در عمل بسیار حیاتی خواهد بود، به ویژه در پروژه‌های اپن‌سورس که سعی دارند موانع را به حداقل برسانند تا بتوانند بیشترین میزان مشارکت را جذب کنند.

پیاده‌سازی‌های زبان‌های موجود

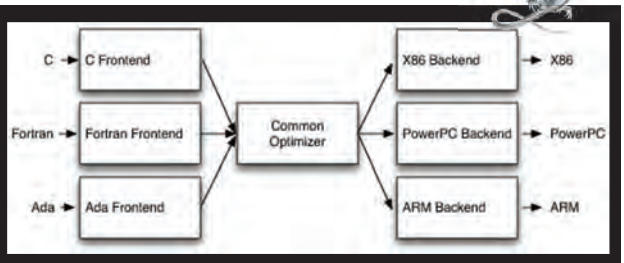
با این‌که مزایا و فواید این طراحی سه مرحله‌ای بسیار وسوسه‌کننده هستند و به‌خوبی در کتاب‌های آموزشی مربوط به کامپایلرها مدون شده‌اند، در عمل این طراحی هیچ‌گاه به صورت کامل پیاده نمی‌شود. با نگاهی به پیاده‌سازی‌های زبان‌های اپن‌سورس (البته در گذشته یعنی زمانی که LLVM کار خود را آغاز کرد) مشاهده خواهید کرد که پیاده‌سازی‌های پرل، پایتون، روبی و جاوا هیچ کد مشترکی ندارند! علاوه بر این، خواهید دید که پروژه‌هایی نظیر GHC (سرنام Glasgow Haskell Compiler) یا FreeBASIC می‌توانند پردازنده‌های مختلفی را هدف بگیرند، اما پیاده‌سازی آن‌ها تنها منحصر به پشتیبانی از یک زبان برنامه‌نویسی خاص است. همچنین فناوری‌های متنوعی برای پیاده‌سازی کامپایلرهای خاص و تک منظوره نظیر کامپایلرهای JIT مخصوص پردازش تصویر، RegEx، درایورهای کارت‌های گرافیک و سایر حوزه‌هایی که به انجام پردازش‌های فراوان توسط پردازنده وابسته هستند، مورد استفاده قرار می‌گیرد. با همه این‌ها، سه شیوه موفق استفاده از این مدل طراحی را می‌توان مشاهده کرد. شیوه نخست در جاوا و ماشین‌های مجازی دات‌نت قابل مشاهده است. این سیستم‌ها یک کامپایلر JIT، پشتیبانی از Runtime و یک قالب بایت‌کد با تعریف بسیار دقیق و مناسب را فراهم می‌کنند. در این صورت هر زبانی که بتواند به آن بایت‌کد کامپایل شود (که تعدادشان هم به شدت زیاد است) می‌تواند از توانایی‌ها و قابلیت‌های JIT، Optimizer و همین‌طور Runtime استفاده کند.

نکته منفی چنین پیاده‌سازی‌هایی این است که انعطاف‌پذیری اندکی در انتخاب Runtime دارند. هر دو مورد کامپایل Garbage JIT، Collection و استفاده از یک مدل خاص اشیا (Object Model) را به کاربر تحمیل می‌کنند. این امر باعث می‌شود که زبان‌هایی نظیر C (مثلاً در پروژه LLVM) که به خوبی با این مدل سازگار نیستند، به کارایی حداکثری خود دست نیابند. شیوه دوم (که شاید بدشانس‌ترین آن‌ها و در عین حال پرکاربردترین روش برای استفاده دوباره از فناوری کامپایلر باشد) ترجمه کد منبع به زبان C (یا سایر زبان‌ها) و ارسال آن به کامپایلرهای موجود C است.

این کار امکان استفاده دوباره از Optimizer و Backend یا مولد کد را فراهم آورده کنترل و انعطاف‌پذیری خوبی را برای Runtime ممکن می‌سازد و درک، پیاده‌سازی و نگه‌داری آن را برای برنامه‌نویسان Frontend بسیار ساده می‌کند. متأسفانه چنین کاری امکان پیاده‌سازی کارای سیستم مدیریت استثنا (Exception Handling) را از بین برده، تجربه دیباگ ناخوشایندی را برای کاربر به ارمغان می‌آورد و فرآیند کامپایل را کند می‌کند. علاوه‌بر این، اگر زبانی که مورد استفاده قرار می‌گیرد، نیازمند قابلیت‌هایی باشد که در C وجود ندارند (مثلاً Tail Callهای تضمین شده) این امر می‌تواند دردسر ساز باشد.

آخرین شیوه موفق پیاده‌سازی چنین مدلی، GCC (سرنام GNU Compiler Collection) است. کامپایلر GCC از Frontendها و Backendهای متنوعی پشتیبانی کرده و جامعه بزرگ و فعالی از مشارکت‌کنندگان را در کنار خود دارد. این کامپایلر مدت‌ها تنها یک کامپایلرC بود که می‌توانست معماری‌های متنوعی را هدف بگیرد و با هک‌های مختلف، تعداد اندکی از زبان‌های برنامه‌نویسی دیگر را هم پشتیبانی می‌کرد. با گذشت زمان، جامعه GCC به تدریج به طرحی تمیزتر و بهتر دست یافت.

در نسخه ۴/۴، GCC برای Optimizer از نحوه نمایش کدی استفاده می‌کند که GIMPLE Tuples نامیده می‌شود و بسیار بیش از نمونه‌های قبلی از Fontend مستقل شده است. همچنین Frontendهای آدا و فرترن آن از ASTهای تمیز و پالوده‌ای بهره می‌برند. این سه شیوه پیاده‌سازی در عین موفقیت



شکل ۲ هدف گرفتن چندین معماری مقصد

فراوان، محدودیت‌های زیادی در موارد اســتفاده و کاربرد دارند. دلیل این محدودیت‌ها این اســت که به عنــوان برنامه‌هایی یکپارچه طراحی شــده‌اند. به عنوان یک نمونه، در کاربردهای واقعی نمی‌توان GCC را به صــورت توکار در برنامه دیگری گنجانــد، از آن به عنوان یک کامپایلر JIT یــا Runtime اســتفاده کرد یا قســمت‌هایی از آن را بــدون نیاز به بخش‌های دیگر بیرون کشیده و مورد استفاده دوباره قرار داد. افرادی که می‌خواســتند از Frontend زبان C++ در GCC برای ایجاد مستندات، نشــانه‌گذاری کد، Refactoring و ابزارهای تحلیل آماری استفاده کنند، مجبــور بودنــد از GCC به عنوان یک برنامه یکپارچه اســتفاده کنند که اطلاعــات جالــب و مفید را در قالب XML بیرون مــی‌داد یا باید پلاگینی می‌نوشــتند که کدهای خارجی را در پردازه GCC تزریق می‌کرد. دلایل متعددی وجود دارد که باعث می‌شــود نتــوان از اجزای GCC به‌عنوان کتابخانه‌های مســتقل اســتفاده دوباره به عمل آورد. اســتفاده فراوان از متغیرهای جهانی، اســتفاده اختیاری از ثابت‌ها، ساختارهای داده‌ای بــا طراحی ضعیــف، کدهای مبنــای پراکنده و نابســامان و در نهایت اســتفاده از ماکروها، از جمله این دلایل هستند. به عنوان مثال، استفاده از ماکروها باعث می‌شود که کدهای مبنا تنها به گونه‌ای کامپایل شوند کــه در هر زمان تنها یک Frontend و یک Backend پشــتیبانی شــود. هرچند مشــکلی که اصلاح آن از همه ســخت‌تر اســت، مشکلات ارثی معماری است که از ریشه‌های اولیه و قدمت زیاد این کامپایلر سرچشمه می‌گیرند.

کامپایلــر GCC به صورت خــاص از لایه‌بندی و انتزاع‌های ضعیف رنج می‌برد. قسمت Backend کدهای AST تولید شده توسط Frontend را پیمایــش می‌کنــد تا اطلاعات دیباگ را تولید کند، قســمت Frontend ســاختارهای داده‌ای مرتبــط با Backend را به وجــود می‌آورد و کل کامپایلر به ســاختارهای داده جهانی (Global) متکی اســت که توسط رابط خط فرمان تنظیم می‌شوند.

نحوه نمایش کدهای LLVM یا LLVM IR

حال که از پس‌زمینه‌های تاریخی و شرایط موجود سخن گفتیم، باید به سراغ LLVM برویم. مهم‌ترین خصوصیت طراحی LLVM چیزی است کــه نحوه نمایش میانی، Intermediate Representation یا به اختصار IR نامیده می‌شــود و در واقع قالبی اســت که برای نمایش و ارائه کد در کامپایلر مورد استفاده قرار می‌گیرد.

این نحوه نمایش به گونه‌ای طراحی شده است که میزبان تغییر شکل‌ها و تحلیل‌های میانی باشــد که در بخش Optimizer کامپایلرهای معمول مشاهده می‌شود. در طراحی LLVM IR اهداف خاصی مدنظر قرار گرفته‌اند که پشتیبانی از بهینه‌سازی‌های زمان اجرا (Runtime Optimization)، تحلیل‌هــای مرتبــط بــا کل برنامــه، بهینه‌ســازی‌های میــان تابعی (Cross-function/Interprocedural Optimization)، تغییرهــــای ســاختاری تهاجمــی (Aggressive Restructuring Transformation) از آن جمله‌انــد. مهم‌تریــن جنبه چنین رویکردی این اســت که خود IR بــه صــورت زبانــی در رده اول (First Class Language) زبانی اســت کــه بتواند با توابع به‌عنوان آرگومان ورودی یــا مقدار خروجی توابع دیگــر کار کنــد. در این زبان‌ها میان تابــع و متغیر تفاوتی وجود ندارد) با ســمانتیکی کاملاً تعریف شــده، طراحی شده اســت. برای واضح‌تر شــدن ایــن مطلب بــه نمونه‌ای که در فهرســت ۱ آورده شــده اســت توجه کنید.

```
define i32 @add1(i32 %a, i32 %b) {
entry:
 %tmp1 = add i32 %a, %b
 ret i32 %tmp1
}
define i32 @add2<i32 %a, i32 %b) {
entry:
 %tmp1 = icmp eq i32 %a, 0
 br i1 %tmp1, label %done, label %recurse
recurse:
 %tmp2 = sub i32 %a, 1
 %tmp3 = add i32 %b, 1
 %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
 ret i32 %tmp4
done:
 ret i32 %b
}
```

فهرست۱ نمونه‌ای از کدهای LLVM IR

این کد LLVM IR، ترجمه تابعی به زبان C است که آن را در فهرست ۲ مشــاهده می‌کنید. این قطعه کد دو روش متفاوت را برای جمع کردن دو عدد صحیح تعریف می‌کند.

```
unsigned add1(unsigned a, unsigned b) {
 return a+b;
}

// Perhaps not the most efficient way to add two
numbers.
unsigned add2(unsigned a, unsigned b) {
 if (a == 0) return b;
 return add2(a-1, b+1);
}
```

فهرست۲ کد C متناظر با LLVM IR فهرست شماره ۱

همان‌طــور کــه در این مثال مشــاهده می‌کنیــد، LLVM IR در واقع نوعی مجموعه دســتورالعمل مجازی، شبیه مجموعه دستورالعمل‌های RISC اســت. درســت همانند یک مجموعه دســتورالعمل RISC واقعی، LLVM IR نیــز از توالی ســاده دســتورالعمل‌هایی نظیــر جمع، تفریق، مقایســه و انشعاب پشــتیبانی می‌کند. این مجموعه دستورالعمل‌ها در قالب دستورالعمل‌های سه آدرسی یا Three Address قرار دارند به این معنــا که می‌توانند ورودی‌های متعددی را از چندین رجیســتر گرفته و نتیجه را در ثباتی دیگر ثبت کنند. این درســت برخلاف روشــی است که در معماری‌های دو آدرسی نظیر x86 استفاده می‌شود.

در این معماری‌ها، اطلاعات رجیســتر ورودی بازنویســی شده و از بین می‌رود. در ماشــین‌های تک آدرســی سیســتم تنها یک عملوند مشخص را دریافت کرده و عملیات را روی یک آکومولاتور یا یک پشته (Stack) به‌انجام می‌رساند. علاوه بر این LLVM IR از برچسب‌ها (Label) پشــتیبانی می‌کند و به صورت کلی مانند فــرم عجیب و غریبی از زبان

اسمبلی به نظر می‌رسد. برخلاف بیشتر مجموعه دستورالعمل‌های RISC، در دستورالعمل‌های LLVM،انتخاب نوع داده مورد استفاده از میان یک سیستم نوع داده ساده اجباری است. به عبارت دیگر LLVM را می‌توان Strongly Typed دانست. برای مثال i32 نشانه یک عدد صحیح ۳۲ بیتی و i32** نشانه اشاره‌گری (Pointer) به یک اشاره‌گر دیگر است که اشاره‌گر دوم به یک عدد صحیح ۳۲ بیتی اشاره می‌کند. در این سیستم برخی جزئیات مربوط به معماری ماشین مورد استفاده به صورت انتزاعی از سر راه برداشته می‌شوند. مثلاً سیستم فراخوانی و جواب‌دهی ماشین، تنها با استفاده از دستورالعمل‌های call و ret با آرگومان‌های اجباری انتزاعی می‌شوند. تفاوت اساسی دیگری که میان LLVM IR و کدهای زبان ماشین، وجود دارد این است که LLVM IR از تعداد مشخصی ثبات نام‌گذاری شده استفاده نمی‌کند، بلکه مجموعه‌ای نامتناهی از متغیرهای موقتی را به کار می‌برد که نام آن‌ها با علامت % شروع می‌شود.

در پس پیاده‌سازی به صورت یک زبان، LLVM IR در واقع در سه فرم متناظر (Isomorphic) تعریف می‌شود. در بالاترین لایه قالب متنی (Textual Format) قرار دارد. در لایه میانی، ساختار داده درون حافظه (In-Memory Data Structure) وجود دارد که توسط بهینه‌سازها ایجاد و ویرایش می‌شود و در لایه زیرین یک قالب کارا و فشرده به صورت بیت‌کد (BitCode) روی دیسک وجود خواهد داشت. پروژه LLVM همچنین ابزارهایی را برای تبدیل فرمت نوشتاری دیسک از حالت متنی به حالت باینری فراهم می‌کند. برنامه llvm-as فایل‌های متنی با پسوند ll. را به فایلی با پسوند bc.تبدیل می‌کند که حاوی بیت‌کدها است و برنامه llvm-dis دقیقاً عکس این کار را انجام می‌دهد. نحوه نمایش میانی یا Intermediate Representation یک کامپایلر از این جهت حائز اهمیت است که می‌تواند دنیایی کامل برای Optimizer آن کامپایلر باشد؛ چراکه برخلاف Frontend و Backend کامپایلر، Optimizer به یک زبان خاص یا یک معماری سخت‌افزاری خاص محدود نیست. از طرف دیگر، این نمایش واسطه باید به‌خوبی به هر دو طرف خدمت‌رسانی کند. یعنی باید به گونه‌ای باشد که تولید کدهای آن برای Frontend ساده باشد و در عین حال به اندازه کافی گویا و توصیفی باشد که بهینه‌سازی‌های مهم و حیاتی را برای ماشین‌های مقصد مختلف امکان‌پذیر کند.

نوشتن یک بهینه‌ساز برای LLVM IR

برای داشتن درکی ساده از این‌که بهینه‌سازی چگونه کار می‌کند، بهتر است از چندین مثال استفاده کنیم. انواع مختلف و بسیار متفاوتی از بهینه‌سازی در سطح کامپایلر وجود دارد، بنابراین تجویز نسخه‌ای قطعی برای هر مشکل دلخواهی بسیار دشوار است. با این حال، بیشتر بهینه‌سازی‌ها از ساختاری سه مرحله‌ای به این شرح استفاده می‌کنند:

- به دنبال الگویی می‌گردند که قابل تبدیل باشد.
- کنترل می‌کنند که آیا این تبدیل در مورد این نمونه خاص صحیح و امن است.
- با به‌روزرسانی کد، تبدیل را انجام می‌دهند.

ساده‌ترین نوع این بهینه‌سازی‌ها، تطبیق الگو روی شناسه‌های ریاضی است. به عنوان مثال، به ازای هر عدد صحیح x، حاصل x-x برابر صفر، 0-x برابر x و حاصل x-(2*x) برابر x خواهد بود. نخستین پرسش این است که چنین مواردی در LLVM IR چگونه نمایش داده می‌شوند؟ نمونه‌هایی از این موارد را در فهرست ۳ مشاهده می‌کنید.

```
%example1 = sub i32 %a, %a
%example2 = sub i32 %b, 0
%tmp = mul i32 %c, 2
%example3 = sub i32 %tmp, %c
```

فهرست ۳ نمونه‌هایی از الگوهای ریاضی قابل بهینه‌سازی

برای انجام این نوع از تبدیل‌های جزئی، LLVM رابطی را برای ساده‌سازی دستورالعمل‌ها فراهم کرده است که به‌عنوان ابزاری در سایر تبدیل‌های سطح بالا نیز مورد استفاده قرار می‌گیرد. این تبدیل‌های خاص در تابعی با عنوان SimplifySubInst نگه‌داری می‌شوند و همانند فهرست ۴ هستند.

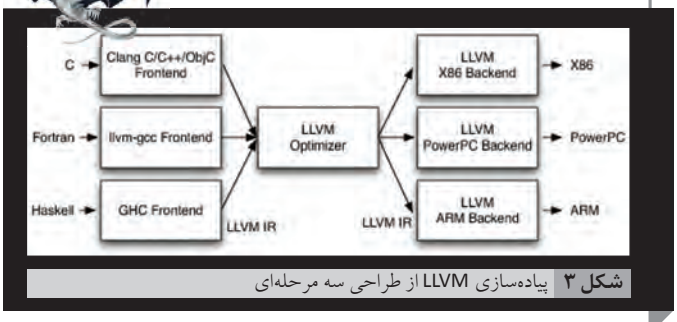
```
// X - 0 -> X
if (match(Op1, m_Zero()))
 return Op0;
// X - X -> 0
if (Op0 == Op1)
 return Constant::getNullValue(Op0->getType());
// (X*2) - X -> X
if (match(Op0, m_Mul(m_Specific(Op1), m_Constan-
tInt<2>())))
 return Op1;
…
return 0; // Nothing matched, return null to indi-
cate no transformation.
```

فهرست ۴ نمونه‌ای از تبدیل‌های موجود در تابع SimplifySubInst

در این کدها، Op0 و Op1 به عملوندهای سمت چپ و راست یک عمل تفریق اعداد صحیح اشاره می‌کنند. نکته مهم این است که این عملوندها لزوماً حاوی مقادیر ممیز شناور براساس استاندارد IEEE نیستند. خود LLVM براساس C++ توسعه‌داده شده است که قابلیت‌های تشخیص و انطباق الگوی آن (در مقایسه با زبان‌های تابعی نظیر Objective Caml) چندان مشهور نیستند، اما این زبان یک سیستم قالب‌های بسیار کلی (Very General Template System) دارد که امکان پیاده‌سازی قابلیت‌هایی مشابه را فراهم می‌کند.

تابع match و توابع با پیشوند _m امکان انجام عملیات‌های انطباق الگوی توصیفی را روی کدهای LLVM IR فراهم می‌کنند. به عنوان مثال تابع m_Specific تنها تطابق‌هایی را می‌یابد که در آن‌ها عملوند سمت چپ یک عملیات ضرب همانند Op1 باشد.

با استفاده از این سه مورد در کنار یکدیگر، تمام الگوها تطابق یافته و تابع، جایگزینی را باز می‌گرداند یا در صورت نبود امکان جایگزینی یک اشاره‌گر خالی را باز می‌گرداند. فراخواننده این تابع (Simplify Instruction) یک توزیع‌کننده (Dispatcher) است که دستورالعمل‌ها را جابه‌جا کرده و آن‌ها را میان توابع کمکی توزیع می‌کند. این تابع توسط بهینه‌سازهای مختلف فراخوانده می‌شود. یک نمونه از این فراخوانی‌ها را می‌توانید در فهرست ۵ ببینید.



شکل ۳ پیاده‌سازی LLVM از طراحی سه مرحله‌ای

```
for (BasicBlock::iterator I = BB->begin(), E
  = BB->end(); I != E; ++I)
  ((if (Value *V = SimplifyInstruction(I
I->replaceAllUsesWith(V);
```

فهرست ۵ نمونه فراخوانی تابع SimplifyInstruction

این کد به‌سادگی تمام دستورالعمل‌های یک بلوک را در یک حلقه بررسی می‌کند تا ببیند آیا هیچ یک از آن‌ها قابل بهینه‌سازی هست یا نه. اگر پاسخ مثبت باشد، یعنی SimplifyInstruction مقداری غیر صفر را بازگرداند، از replace All Uses With برای به‌روزرسانی کد و تغییر عملیات قابل ساده‌سازی استفاده می‌کند.

پیاده‌سازی LLVM از طراحی سه مرحله‌ای

در تمام کامپایلرهای مبتنی بر LLVM، قسمت Frontend وظیفه parse کردن، صحت‌سنجی و یافتن خطاها در کد ورودی و سپس ترجمه کد parse شده به LLVM IR را بر عهده دارد. ترجمه کد به LLVM IR معمولاً (و نه همیشه) از طریق ایجاد یک AST و سپس تبدیل آن به LLVM IR انجام می‌گیرد. این کد IR در صورت نیاز ممکن است چندین بار برای تحلیل و بهینه‌سازی بررسی شود که این کار باعث بهبود کد خواهد شد. این کد، همان‌طور که در شکل ۳ می‌بینیم، پس از آن به مولد کد داده می‌شود تا کدهای محلی زبان ماشین را تولید کند. این یک پیاده‌سازی بسیار سرراست از طراحی سه‌مرحله‌ای است، اما برخی توانایی‌ها و انعطاف‌پذیری‌هایی که معماری LLVM به‌واسطه LLVM IR می‌تواند کسب کند، در این حالت از دست خواهند رفت. (شکل ۳)

LLVM IR نمایشی کامل از کد (Complete Code Representation)

LLVM IR هم به خوبی تعریف شده است و هم تنها رابط موجود به Optimizer است. به این معنا که برای نوشتن یک Frontend برای LLVM تنها چیزی که باید بدانید این است که LLVM IR چیست، چگونه کار می‌کند و انتظار استفاده از چه ثابت‌هایی را دارد. به این دلیل که LLVM IR فرمی متنی از رده اول دارد، ممکن و منطقی است که Frontend ایجاد شده توسط شما خروجی LLVM IR خود را به صورت متنی تولید کند و سپس از پایپ‌های یونیکس برای ارسال آن به مراحل بعدی Optimizer و مولد کد دلخواه خود استفاده کنید.

هرچند ممکن است عجیب به نظر برسد، اما این قابلیت در واقع یک ویژگی نوین در LLVM است و یکی از دلایل اصلی موفقیت و کاربردهای گسترده LLVM به شمار می‌رود. حتی کامپایلر به شدت موفق GCC که معماری بسیار خوبی هم دارد، از چنین ویژگی سودمندی برخوردار نیست. مدل نمایش میانی GCC یعنی GIMPLE یک نمایش کاملاً مستقل و کامل نیست. به عنوان مثال، هنگامی که مولد کد GCC می‌خواهد از اطلاعات دیباگ DWARF صرف نظر کند، به عقب باز می‌گردد و فرم «درختی» کد منبع را بررسی می‌کند.

خود GIMPLE برای نمایش عملیات‌ها در کد از Tuple استفاده می‌کند، اما (حداقل در GCC نسخه ۴/۵) برای نمایش عملوندها، ارجاع‌هایی به فرم درختی در سطح کد منبع را مورد استفاده قرار می‌دهد.

چنین پیاده‌سازی‌ای به این معنا است که برنامه‌نویسان برای ایجاد Frontend باید ساختار داده درختی GCC را بشناسند و در کنار GIMPLE این ساختار درختی را نیز تولید کنند. قسمت‌های Backend در GCC نیز از همین مشکل رنج می‌برند، به همین دلیل نویسندگان Backend هم باید بدانند که Backendهای RTL چگونه کار می‌کنند. درنهایت GCC راهی برای نوشتن GIMPLE یا هر نوع نمایش دیگری از کد به‌صورت متنی را ندارد. نتیجه این است که کسب تجربه با GCC به نسبت دشوار است و به همین دلیل Frontendهای محدودی دارد.

در قسمت بعدی این مقاله مزیت‌هایی که کتابخانه‌های LLVM به ارمغان خواهند آورد، تعریف مولدهای کد و فایل توصیف معماری مقصد برای LLVM مورد بررسی قرار خواهند گرفت و در نهایت خواهیم دید که طراحی ماجولار LLVM چه مزیاتی را به‌همراه خواهد داشت.