تولید کد میانی برای تسلنک

در تمرین عملی درس طراحی کامپایلر، هدف پیاده‌سازی قسمت «Front-end» از یک مترجم است که یک فایل در زبان تسلنک را می‌خواند و کد میانی معادل آن را تولید می‌کند. برای تولید کد نهایی از روی این کد میانی می‌توان از ابزارهای آماده‌ی دیگری استفاده نمود. یکی از این ابزارها «LLVM» می‌باشد. ورودی «LLVM» مشابه کد میانی در یک کامپایلر است و خروجی آن کد نهایی برای معماری‌های رایج است. بنابراین، مترجم شما می‌تواند به عنوان کد میانی، برنامه‌ای در زبان ورودی «LLVM» تولید نماید و با استفاده از این برنامه کد نهایی تولید شود. در ادامه این زبان معرفی می‌گردد و جزئیات لازم برای ترجمه‌ی زبان تسلنک به آن شرح داده می‌شوند. زبان ورودی «LLVM» دارای ویژگی‌های فراوانی است که این مستند خود را به گوشه‌ای از آنها (قسمت‌هایی که در ترجمه‌ی برنامه‌های زبان تسلنک مفید هستند) محدود می‌کند.

معرفی و راه‌اندازی

برنامه‌ی «LLVM» از آدرس http://llvm.org/ قابل دریافت است. در توزیع لینوکس «Ubuntu»، با دستور زیر می‌توانید آن را نصب کنید:

```
$ sudo apt-get install llvm
```

پس از نصب آن، دستور «llc» برنامه‌هایی که به زبان ورودی «LLVM» نوشته شده‌اند را به کد اسمبلی برای یک معماری ترجمه می‌کند و دستور «lli» آن برنامه‌ها را اجرا می‌نماید. برای مثال، در صورتی که فایل آزمایشی «test.ll» شامل عبارت‌های زیر باشد (این برنامه‌ی «LLVM» یک تابع با نام «main» تعریف می‌کند که اجرا از آن شروع می‌شود):

```
define i32 @main()
{
	ret i32 0
```

```
}
```

با دستور زیر، این برنامه اجرا می‌شود:

```
$ lli <test.ll
```

انواع ابتدایی و تعریف توابع

انواع عددی صحیح ابتدایی در «LLVM» به صورت «iX» نمایش داده می‌شوند که در آن «X» عددی است که تعداد بیت‌های آن نوع را نشان می‌دهد. برای مثال «i32»، اعداد صحیح ۳۲ بیتی را نشان می‌دهد. نوع «word» در زبان تسلنک را می‌توانید معادل «i32» یا «i64» در نظر بگیرید.

زبان ورودی «LLVM» مشابه یک زبان اسمبلی سطح بالا است که برخی از جزئیات معماری (برای مثال رجیسترها) در آن بیان نمی‌شوند. برنامه‌های ورودی «LLVM» شامل تعدادی تابع هستند؛ در تولید کد میانی برای برنامه‌های تسلنک، به ازای هر تابع در زبان تسلنک، یک تابع در زبان ورودی «LLVM» تعریف می‌کنید. تعریف توابع در «LLVM» به تعریف توابع در زبان‌های رایجی مثل «C» شباهت دارد. مثال زیر یک تابع با نام «testfunc» که دو ورودی، یکی از نوع اشاره‌گر به «i32» و دیگری از نوع «i32»، را دریافت می‌کند و یک عدد از نوع «i32» را بر می‌گرداند.

```
define i32 @testfunc(i32 *%A, i32 %n)
{
      ret i32 0
}

define i32 @main()
{
      call i32 @testfunc(i32 *null, i32 0)
      ret i32 0
}
```

در «LLVM» نام متغیرهای سراسری (مثل توابع) با علامت «@» شروع می‌شود و نام متغیرهای محلی (ورودی‌های یک تابع یا تعریف شده در بدنه‌ی آن) با علامت «%» شروع

می‌شود. بنابراین نام تابع «@testfunc» و نام ورودی‌های «%A» و «%n» قرار داده شده‌اند.

متغیرهای محلی و موقتی

متغیرهای موقتی و محلی در توابع را می‌توان به صورت زیر تعریف نمود.

```
define i32 @testmul(i32 %a, i32 %b)
{
	%ret = mul i32 %a, %b
	ret i32 %ret
}
```

این تابع حاصل ضرب دو عدد را بر می‌گرداند. متغیر «%ret» مقدار حاصل ضرب ورودی-های «%a» و «%b» را نگه خواهد داشت و مقدار حاصل از این تابع برگشت داده می‌شود. به این نکته دقت کنید که در «LLVM» نوع هر عمل باید به صراحت بیان شود؛ برای نمونه در این دستور پس از عملگر «mul» و «mul» نوع عملوندهای آنها «i32» تعریف شده‌اند.

از متغیرهای محلی که نام آنها یک عدد است، می‌توان برای نگهداری مقادیر موقتی استفاده کرد؛ در استفاده از این متغیرها باید دقت کرد که آنها باید همواره باید به ترتیب تعریف شوند. مثال زیر عبارت «a + b + c × d» را محاسبه می‌نماید.

```
define i32 @etc(i32 %a, i32 %b, i32 %c, i32 %d)
{
	%1 = add i32 %a, %b
	%2 = mul i32 %c, %d
	%3 = add i32 %1, %2
	ret i32 %3
}
```

متغیرهای ذخیره شده در پشته

یکی از نکات مهم در «LLVM» که می‌تواند منشاء ابهام باشد، تفاوت بین مفهوم متغیرهای محلی «LLVM» (تعریف شده در تابع) و متغیرهای ذخیره شده در پشته (مثل متغیرهای محلی توابع تسلنک) می‌باشد. متغیرهای محلی «LLVM» (که نام آنها با علامت «%»

شروع می‌شود) تنها یک بار می‌توانند تعریف شوند و پس از تعریف، مقدار آنها تغییر نمی‌کند ولی متغیرهای ذخیره شده در پشته این محدودیت را ندارند.

برای تعریف متغیرهای ذخیره شده در پشته، می‌توان از عملگر «alloca» در «LLVM» استفاده نمود. آدرس متغیری که با «alloca» ایجاد می‌شود را می‌توان در یک متغیر محلی «LLVM» نگهداری نمود. سپس، برای خواندن و نوشتن به متغیرهای پشته باید از عملگر «load» و «store» استفاده کرد. در مثال زیر، متغیر محلی «%ret» آدرس یک متغیر پشته را نگهداری می‌کند.

```
define i32 @five()
{
	%ret = alloca i32
	store i32 5, i32 *%ret
	%1 = load i32 *%ret
	ret i32 %1
}
```

در این مثال ابتدا مقدار پنج به متغیر پشته‌ای که آدرس آن در «%ret» قرار گرفته است، انتقال داده می‌شود و سپس خوانده و از تابع برگشت داده می‌گردد.

آرایه‌ها

در نوع آرایه‌ها در «LLVM» اندازه و نوع عناصر آن ذکر می‌شود. برای نمونه «[20 x i32]» نوع آرایه‌ای است که ۲۰ عنصر با نوع «i32» را نگهداری می‌کند. برای دسترسی به عناصر آرایه، می‌توان از عملگر «getelementptr» استفاده نمود. در مثال زیر یک آرایه روی پشته تخصیص می‌یابد و به عنصر پنجم آن مقدار ۱۰ اختصاص می‌یابد.

```
define i32 @testarrays()
{
	%A = alloca [20 x i32]
	%1 = getelementptr [20 x i32] *%A, i32 0, i32 5
	store i32 10, i32 *%1
	ret i32 0
}
```

جهش‌ها و حلقه‌ها

اجرای شرطی دستورات در «LLVM» و جهش‌ها با یک مثال نشان داده می‌شوند. در مثال زیر مقدار بیشینه‌ی دو ورودی در تابع «@max» محاسبه می‌شود.

```
define i32 @max(i32 %a, i32 %b)
{
      %ret = alloca i32
      %1 = icmp sgt i32 %a, %b
      br i1 %1, label %.seta, label %.setb
.seta:
      store i32 %a, i32 *%ret
      br label %.done
.setb:
      store i32 %b, i32 *%ret
      br label %.done
.done:
      %2 = load i32 *%ret
      ret i32 %2
}
```

با استفاده از عملگر «icmp» می‌توان دو مقدار عددی صحیح را مقایسه کرد. عبارت بعد از «icmp» می‌تواند یکی از این کلمه‌ها ظاهر شوند: «eq» برای تساوی، «ne» برای عدم تساوی، «slt» برای کوچک‌تر بودن علامت‌دار، «sgt» برای بزرگ‌تر بودن علامت‌دار، «sle» برای تساوی یا کوچک‌تر بودن علامت‌دار، «sge» برای تساوی یا بزرگ‌تر بودن علامت‌دار. همان طور که دیده می‌شود، برچسب‌هایی (Label) در بدنه‌ی تابع تعریف می‌شوند. با استفاده از عملگر «br» می‌توان به یکی از این برچسب‌ها جهید. در صورتی که فقط یک ورودی به «br» داده شود، یک جهش غیر شرطی انجام می‌شود و در صورتی که سه ورودی به آن داده شوند، یک جهش شرطی انجام می‌شود (ورودی اول شرط، ورودی دوم مقصد جهش در صورت موفق بودن آن و ورودی سوم مقصد جهش در صورت ناموفق بودن آن هستند). توجه به این نکته لازم است که قبل از هر برچسب حتما باید یک جهش باشد.

توابع ورودی و خروجی در تسلنک

برای پیاده‌سازی توابع ورودی و خروجی در تسلنک می‌توان از توابع کتابخانه‌ی زبان «C» استفاده کرد. پیاده‌سازی دو تابع «printword» و «readword» تسلنک در ادامه نشان داده شده‌اند.

```
; declarations
declare i32 @printf(i8*, ...)
declare i32 @scanf(i8*, ...)

; print a word
@.msg1 = internal constant [4 x i8] c"%d\0A\00"
define i32 @printword(i32 %val)
{
      %1 = call i32 (i8*, ...)*
            @printf(i8* getelementptr([4 x i8]* @.msg1,
                        i32 0, i32 0), i32 %val)
      ret i32 0
}

; read a word
@.msg2 = internal constant [4 x i8] c"%d\00\00"
define i32 @readword()
{
      %1 = alloca i32
      %2 = call i32 (i8*, ...)*
            @scanf(i8 *getelementptr([4 x i8] *@.msg2,
                        i32 0, i32 0), i32* %1)
      ret i32 0
}
```