طراحی کامپایلر

استاد درس:

مهران علیدوست‌نیا

دانشجویان:

علی لامعی رامندی

مریم پایدار اردکانی

ترم پاییز ۱۴۰۲

این پروژه در چند مرحله انجام می‌شود:

۱. تعریف زبان برنامه نویسی
   - نوشتن گرامر زبان
۲. ساختن Scanner
   - مشخص کردن توکن‌های زبان
   - نوشتن Lexer مربوط به زبان
۳. ساختن Parser
   - بررسی چگونگی قرار گرفتن توکن‌ها و ساخت AST
۴. بررسی Semantic درخت
۵. بهینه‌سازی کد
   - حذف کد مرده
۶. تبدیل درخت به زبان میانی llvm
۷. بررسی عملکرد کامپایلر

۱. تعریف زبان برنامه نویسی

این زبان شامل دستورات زیر است:

- دستورات ساده
  - تعریف متغیر
  - انتساب متغیر
  - عملیات ریاضی
- دستورات مرکب
  - شرط
  - حلقه

با توجه به توضیحاتی که در مورد هر دستور، در صورت پروژه نوشته شده است، گرامر زبان فوق به شکل زیر خواهد بود:

letter : a | . . . | z | A | . . . | Z

digit : 0 | . . . | 9

number : digit digit*

assignOp : −= | /= | *= | += | %= | ^= | =

relop : <= | < | > | >= | == | ! =

letdig : (digit | letter)*

Ident : letter letdig

---

program : (varDecl | ifStmt | iterStmt | assignStmt)*

varDecl : int (declWithAssign | declWithoutAssign);

declWithoutAssign : (Ident ,)* Ident

declWithAssign : Ident , declWithAssign , expr

| Ident , declWithAssign

| Ident assignOp expr

---

ifStmt : if logicalExpr : begin assignList end elifStmt

| if logicalExpr : begin assignList end elifStmt else: begin assignList end

elifStmt : (elif logicalExpr : begin assignList end)*

---

assignList : (assignStmt)*

assignStmt : Ident assignOp expr;

logicalExpr : comparison ((and | or) comparison)*

comparison : expr relop expr | (logicalExpr)

expr : term ((+| -) term)*

term : factor ((* | / | %) factor)*

factor : final (^ final)*

final : (expr) | Ident | number

---

iterStmt : loopc logicalExpr : begin assignList end

توضیحات:

حروف و علامت‌های آبی رنگ، پایانه‌ها را نشان می‌دهند.

برنامه‌ای که با استفاده از این زبان نوشته می‌شود، می‌تواند شامل ۴ دستور باشد:

۱. دستور تعریف متغیر: با ناپایانه varDecl تولید می‌شود.

به منظور تعریف چند متغیر باهم، دو حالت پیش می‌آید:

- همه متغیرها بدون مقدار اولیه تعریف می‌شوند (مانند int a, b, c;) که با ناپایانه declWithoutAssign تولید می‌شود.
- بعضی از متغیرها مقدار اولیه دارند (مانند int a, b, c = 2;) که با ناپایانه declWithAssign تولید می‌شوند.

۲. دستور شرط: با ناپایانه ifStmt تولید می‌شود.

۳. دستور حلقه: با ناپایانه iterStmt تولید می‌شود.

۴. دستور انتساب یک مقدار: با ناپایانه assignStmt تولید می‌شود.

عملیات ریاضی با کمک ناپایانه expr با رعایت اولویت عملگرها تولید می‌شوند.

۲. ساختن Scanner

زبان تعریف شده دارای انواع توکن زیر می‌باشد:

دیگر علامت‌ها	کلمه‌های کلیدی	علامت‌های دوحرفی	علامت‌ها تک‌حرفی
eoi	KW_int	minus_assign	comma
unknown	KW_if	plus_assign	semicolon
ident	KW_elif	star_assign	colon
number	KW_else	slash_assign	plus
	KW_begin	mod_assign	minus
	KW_end	exp_assign	star
	KW_loopc	eq	slash
	KW_and	neq	mod
	KW_or	gt	exp
		lt	l_paren
		gte	r_paren
		lte	assign

پس از مشخص کردن توکن‌ها در کلاس Token، توابع زیر را پیاده‌سازی می‌کنیم:

- getKind: نوع توکن را برمی‌گرداند.
- getText: خود توکن را برمی‌گرداند.
- is: در صورتی که توکن از جنس داده شده باشد، true برمی‌گرداند.
- isOneOf: اگر جنس توکن یکی از ورودی‌ها باشد، true برمی‌گرداند.

کلاس Lexer نیز شامل توابع زیر است:

- formToken: توکن مورد نظر را می‌سازد.
- next: توکن بعدی را برمی‌گرداند:

بدون در نظر گرفتن white space ها، پس از خواندن یک char بررسی می‌شود که کلمه کلیدی، نام متغیر، عدد و یا عملگر می‌باشد تا توکن مربوط به آن را با کمک تابع formToken بسازد.

۳. ساختن Parser

این بخش از کامپایلر در فایل‌های AST.h، Parser.h و Parser.cpp پیاده‌سازی شده است.

کلاس ASTVisitor توابع مورد نیاز برای visit نودها را تعریف می‌کند تا در کلاس InputCheck برای بررسی سمنتیک و در کلاس ToIRVisitor برای ساخت IR متناظر، override شوند.

کلاس AST پایه کلاس‌های موجود برای پیاده‌سازی درخت است. تابع مهمی به نام accept در آن پیاده‌سازی شده است که به ASTVisitorها این اجازه را می‌دهد تا نود مربوطه را visit کنند.

همه نودهای دیگر به طور مستقیم یا غیرمستقیم از کلاس AST ارث‌بری می‌کنند که شامل:

- Program: رأس درخت می‌باشد و تنها یک نمونه از آن ساخته خواهد شد. فرزندان این نود از جنس AST خواهند بود.
- Declaration: برای تعریف متغیر از این نود استفاده می‌شود به طوری که می‌تواند هم زمان چند متغیر را تعریف و مقدار دهی کند. متغیری که برایش مقدار اولیه تعریف نشده باشد، برابر با صفر خواهد بود. این نود فرزندانی از جنس Expr و StringRef دارد، که همواره باید تعداد StringRefها از Exprها بیشتر باشد. همچنین این نود از کلاسprogram ارث‌بری می‌کند.
- Expr: نود پایه برای پیاده‌سازی عبارت‌های ریاضی است.
- Final: کوچکترین نود درخت می‌باشد. می‌تواند شامل number یا ident باشد. این نود از کلاس Expr ارث‌بری می‌کند.
- BinaryOp: این نود یک عملیات ریاضی دوعملوندی را مشخص می‌کند. دارای ۲ فرزند است که عملوندهای چپ و راست را مشخص می‌کنند و هر کدام می‌توانند از جنس Expr باشد. همچنین یک متغیر داخلی برای نوع عملیات نیز دارد. این نود از کلاس Expr ارث‌بری می‌کند.
- Assignment: این نود یک عبارت انتساب را مشخص می‌کند. دارای دو فرزند است که یکی از آن‌ها از جنس Factor می‌باشد تا اسم متغیر را مشخص کند و دیگری از جنس Expr می‌باشد تا مقدار متغیر را مشخص کند. همچنین این نود از کلاس Program ارث‌بری می‌کند.
- Logic: نود پایه برای پیاده‌سازی عبارت‌های منطقی است.

- Comparison: این نود یک عملیات مقایسه‌ای دوعملوندی را مشخص می‌کند. دارای ۲ فرزند است که عملوندهای چپ و راست را مشخص می‌کنند و هر کدام می‌توانند از جنس Expr باشند. همچنین یک متغیر داخلی برای نوع عملیات نیز دارد. این نود از کلاس Logic ارث‌بری می‌کند.
- LogicalExpr: یک عبارت منطقی را تشکیل می‌دهد که شامل دو فرزند از جنس Logic است و میان آن‌ها And یا Or قرار دارد. این نود از کلاس Logic ارث‌بری می‌کند.
- IfStmt: این نود دستور شرط را پیاده‌سازی می‌کند و از Program ارث‌بری می‌کند. فرزندان آن:
  - Cond: عبارت منطقی مشخص کننده شرط را در خود ذخیره می‌کند. بنابراین از جنس Logic است.
  - ifAssignments: وکتوری حاوی Assignmentهای بلاک if
  - elseAssignments: وکتوری حاوی Assignmentهای بلاک else
  - elifStmts: وکتوری حاوی elifStmtها
- elifStmt: همانند نود IfStmt است با این تفاوت که بلاک else ندارد.
- IterStmt: پیاده‌سازی حلقه به عهده این نود است که از Program ارث‌بری می‌کند. فرزندان آن:
  - Cond: عبارت منطقی مشخص کننده شرط را در خود ذخیره می‌کند. بنابراین از جنس Logic است.
  - assignments: وکتوری حاوی Assignmentهای بلاک loopc

parser با توجه به ورودی خود، که از lexer دریافت کرده است، درخت مربوط به آن را می‌سازد. به این طریق که از نود Program شروع به پارس می‌کند و تمام فرزندان آن را نیز به طور بازگشتی با توجه به قواعد زبان پارس می‌کند. اگر مشکلی در ساخت نودهای درخت وجود داشت به معنی Syntax error است و آن را به کاربر اعلام می‌کند.

۴. بررسی Semantic درخت

این بخش از کامپایلر در فایل‌های Sema.h و Sema.cpp پیاده‌سازی شده‌است. پس از ساخت درخت لازم است به بررسی درستی معنایی ورودی پرداخت. برای این کار نودهایی که امکان دارد در آن‌ها اشتباهات منطقی رخ دهد بررسی می‌شود و بقیه نودها نیز فرزندان خود را فراخوانی می‌کنند. اشتباهات منطقی محتمل در زبان عبارتند از:

۱. استفاده از متغیر قبل از تعریف آن:

برای بررسی متغیرها نیاز است متغیرهای تاکنون تعریف شده را در حافظه ذخیره کنیم. به همین دلیل یک StringSet تعریف می‌کنیم. هر متغیری که در نود Assignment به کار برده می‌شود را در صورت عدم وجود در StringSet، غیرمجاز شمرده و Semantic error رخ خواهد داد.

۲. تعریف متغیر با نام تکراری:

هنگام تعریف یک متغیر در نود Declaration نام آن را در StringSet گفته شده سرچ می‌کنیم. در صورت وجود، تعریف دوباره آن غیر مجاز بوده و در غیر این صورت آن را به StringSet اضافه می‌کنیم.

۳. تقسیم بر صفر:

در نود BinaryOp و Assignment هنگام استفاده از علمگرهای % ، / ، =/ و =% عملوند سمت راست را چک می‌کنیم تا درصورت وجود مقدار صفر خطا رخ دهد.

۶. بهینه‌سازی کد

به منظور بهینه‌سازی کد در کامپایلر، کدهای مرده آن را حذف می‌کنیم. کد مرده به کدهایی گفته می‌شوند که مقدار متغیر result وابسته به آن‌ها نیست. بنابراین می‌توانند حذف شوند.

برای این کار از تابع Optimize استفاده می‌کنیم. این تابع با گرفتن نود Tree که شامل یک llvm::SmallVector<AST *> است، AST های مرده را حذف کرده و llvm::SmallVector جدیدی برمی‌گرداند. به منظور تغییر llvm::SmallVector، از کلاس OptimizerVisitor کمک می‌گیریم. این کلاس نوعی ASTVisitor می‌باشد و با visit نودهای درخت بررسی می‌کند که آن کد مرده است یا خیر.

یک خط از کد زمانی live به شمار می‌رود که شرایط زیر را داشته باشد:

- اگر Assignment باشد، متغیری که در سمت چپ تساوی قرار می‌گیرد، متغیر live باشد.
- اگر Declaration باشد، متغیری که تعریف می‌شود، متغیر live باشد.

متغیر live، متغیری است که مقدار result وابسته به آن باشد. بنابراین نیاز است دستورات را از انتها مورد بررسی قرار داد. اگر یکی از شروط بالا برقرار بود، متغیرهای سمت راست تساوی نیز live می‌شوند. در صورتی که در یک Assignment، از "=" استفاده شده باشد، متغیر سمت چپ دیگر live نخواهد بود زیرا به مقدار قبلی خودش وابسته نیست و تنها باید منتظر تعریف آن باشیم. اما اگر از اشکال دیگر مثل "+=" استفاده می‌شد، آن متغیر همچنان live می‌ماند.

پیاده‌سازی این قسمت در فایل‌های Optimizer.h و Optimizer.cpp صورت گرفته است. نمونه ای از عملکرد این بهینه‌سازی را در تصویر زیر مشاهده می‌کنید:

```
1 int a = 0;
2 int b = 1;
3 int result;
4 result = a + b;
5 result = 0;
6
7
```

```
marie@ubuntu:~/Desktop/phase2/build/s
; ModuleID = 'simple-compiler'
source_filename = "simple-compiler"

declare void @gsm_write(i32)

define i32 @main(i32 %0, i8** %1) {
entry:
  %2 = alloca i32, align 4
  store i32 0, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  store i32 0, i32* %2, align 4
  call void @gsm_write(i32 0)
  ret i32 0
}
```

۵. تبدیل درخت به زبان میانی llvm

این بخش از کد در فایل‌های CodeGen.h و CodeGen.cpp پیاده‌سازی شده‌است. در این بخش درخت AST به کمک ToIRVisitor پیمایش می‌شود و برای هر نود IR متناظر آن ایجاد می‌شود:

- ابتدا نود Program، ویزیت را برای فرزندانش accept می‌کند تا IR آن‌ها ساخته شود.
- در نود Declaration به وسیله تابع CreateAlloca برای تایپ Int32Ty یک حافظه اختصاص میابد و پوینتر آن را nameMap ذخیره می‌کنیم. در صورت وجود مقدار اولیه برای متغیرها، به وسیله تابع CreateStore آن را در حافظه تازه تخصیص داده شده، ذخیره می‌کنیم.
- در نود Assignment مانند مقداردهی اولیه در Declaration، مقدار انتساب در حافظه ذخیره می‌شود. همچنین با فراخوانی تابع gsm_write مقداری که در متغیر ذخیره می‌شود، چاپ می‌شود تا درستی عملکرد کامپایلر بررسی شود.
- در نود BinaryOp عملوندها را بازیابی کرده و سپس با استفاده از switch case عملگر میان آن‌ها تشخیص داده خواهد شد تا instruction مورد نظر در IR ساخته شود.

  پیاده‌سازی توان با استفاده از ضرب صورت گرفته است.

- اگر نود Final از جنس عدد باشد، به وسیله‌ی تابع ConstantInt مقدار ثابت متناظر با آن ساخته و برگردانده می‌شود. در صورتی که Final از جنس ident باشد، مقدار Value آن را، به وسیله تابع CreateLoad از nameMap بازیابی کرده و برگردانده می‌شود.
- در نود LogicalExpr عملوندها را بازیابی کرده و سپس با استفاده از switch case عملگر میان آن‌ها تشخیص داده خواهد شد تا instruction مورد نظر در IR ساخته شود.
- در نود Comparison نیز عملگرهای مقایسه‌ای تشخیص داده خواهد شد.
- برای پیاده‌سازی IfStmt به ازای بلاک‌های if، elif، else بیسیک بلاک‌های متفاوتی ساخته می‌شود و برای هرکدام یک لیبل گذاشته می‌شود تا در صورت برقراری Cond ها به لیبل مربوطه jump کند.
- در نود IterStmt به ازای بلاک لوپ و Cond بیسیک بلاک ساخته می‌شود و لیبل مخصوص خود را می‌گیرند تا در انتهای لوپ به Cond برگردد و مسیر ادامه برنامه را مشخص کند.

۶. بررسی عملکرد کامپایلر

ابتدا با ران کردن فایل build.sh، پروژه را build می‌کنیم. سپس کدی که می‌خواهیم کامپایل کنیم را در فایل input.txt قرار می‌دهیم. در آخر با ران کردن فایل run.sh خروجی را دریافت می‌کنیم.

Input:

```
int a, b, c, d, e;
a = 1+2^4+3*3;
b = 3*3+2^4+1;
c = 2^4+3*3+1;
d = 2^(4+3)*3+1;
e = 2^4+3*(3+1);
```

Output:

```
ali@vostro:~/Desktop/Compiler/Compiler-Project-main$ ./run.sh
The result is: 26
The result is: 26
The result is: 26
The result is: 385
The result is: 28
```

```
; ModuleID = 'simple-compiler'
source_filename = "simple-compiler"

declare void @gsm_write(i32)

define i32 @main(i32 %0, i8** %1) {
entry:
  %2 = alloca i32, align 4
  store i32 0, i32* %2, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %3, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %4, align 4
  %5 = alloca i32, align 4
  store i32 0, i32* %5, align 4
  %6 = alloca i32, align 4
  store i32 0, i32* %6, align 4
  %7 = load i32, i32* %2, align 4
  store i32 26, i32* %2, align 4
  call void @gsm_write(i32 26)
  %8 = load i32, i32* %3, align 4
  store i32 26, i32* %3, align 4
  call void @gsm_write(i32 26)
  %9 = load i32, i32* %4, align 4
  store i32 26, i32* %4, align 4
  call void @gsm_write(i32 26)
  %10 = load i32, i32* %5, align 4
  store i32 385, i32* %5, align 4
  call void @gsm_write(i32 385)
  %11 = load i32, i32* %6, align 4
  store i32 28, i32* %6, align 4
  call void @gsm_write(i32 28)
  ret i32 0
}
```

Input:

```
int a, b = 10;
if a % 3 == 0: begin
    b = 10;
end
elif a % 3 == 1: begin
    b = 20;
end
else: begin
    b = 30;
end
```

Output:

```
ali@vostro:~/Desktop/Compiler/Compiler-Project-main$ ./run.sh
The result is: 20
```

```
; ModuleID = 'simple-compiler'
source_filename = "simple-compiler"

declare void @gsm_write(i32)

define i32 @main(i32 %0, i8** %1) {
entry:
  %2 = alloca i32, align 4
  store i32 10, i32* %2, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %3, align 4
  br label %if.cond

if.cond:                                          ; preds = %entry
  %4 = load i32, i32* %2, align 4
  %5 = srem i32 %4, 3
  %6 = icmp eq i32 %5, 0
  br i1 %6, label %if.body, label %elif.cond

if.body:                                          ; preds = %if.cond
  %7 = load i32, i32* %3, align 4
  store i32 10, i32* %3, align 4
  call void @gsm_write(i32 10)
  br label %after.if

after.if:                                         ; preds = %else.body, %elif.body, %if.body
  ret i32 0

elif.cond:                                        ; preds = %if.cond
  %8 = load i32, i32* %2, align 4
  %9 = srem i32 %8, 3
  %10 = icmp eq i32 %9, 1
  br i1 %10, label %elif.body, label %else.body
```

```
elif.body:                                        ; preds = %elif.cond
  %11 = load i32, i32* %3, align 4
  store i32 20, i32* %3, align 4
  call void @gsm_write(i32 20)
  br label %after.if

else.body:                                        ; preds = %elif.cond
  %12 = load i32, i32* %3, align 4
  store i32 30, i32* %3, align 4
  call void @gsm_write(i32 30)
  br label %after.if
}
```

Input:

```
int i = 0;
loopc i < 5: begin
    i += 1;
end
```

Output:

```
ali@vostro:~/Desktop/Compiler/Compiler-Project-main$ ./run.sh
The result is: 1
The result is: 2
The result is: 3
The result is: 4
The result is: 5
```

```
; ModuleID = 'simple-compiler'
source_filename = "simple-compiler"

declare void @gsm_write(i32)

define i32 @main(i32 %0, i8** %1) {
entry:
  %2 = alloca i32, align 4
  store i32 0, i32* %2, align 4
  br label %loopc.cond

loopc.cond:                                       ; preds = %loopc.body, %entry
  %3 = load i32, i32* %2, align 4
  %4 = icmp slt i32 %3, 5
  br i1 %4, label %loopc.body, label %after.loopc

loopc.body:                                       ; preds = %loopc.cond
  %5 = load i32, i32* %2, align 4
  %6 = add nsw i32 %5, 1
  store i32 %6, i32* %2, align 4
  call void @gsm_write(i32 %6)
  br label %loopc.cond

after.loopc:                                      ; preds = %loopc.cond
  ret i32 0
}
```

Input:

```
int a, b, c, result = 1, 10, 3;
if a == 1 and b < 10: begin
    result = 0;
end
elif b != 10 or a >= 2: begin
    result = 1;
end
elif b % c == a and a + c == 4 and b <= 10: begin
    result = 2;
end
else: begin
    result = 3;
end
```

```
; ModuleID = 'simple-compiler'
source_filename = "simple-compiler"

declare void @gsm_write(i32)

define i32 @main(i32 %0, i8** %1) {
entry:
  %2 = alloca i32, align 4
  store i32 1, i32* %2, align 4
  %3 = alloca i32, align 4
  store i32 10, i32* %3, align 4
  %4 = alloca i32, align 4
  store i32 3, i32* %4, align 4
  %5 = alloca i32, align 4
  store i32 0, i32* %5, align 4
  br label %if.cond

if.cond:                                          ; preds = %entry
  %6 = load i32, i32* %2, align 4
  %7 = icmp eq i32 %6, 1
  %8 = load i32, i32* %3, align 4
  %9 = icmp slt i32 %8, 10
  %10 = and i1 %7, %9
  br i1 %10, label %if.body, label %elif.cond

if.body:                                          ; preds = %if.cond
  %11 = load i32, i32* %5, align 4
  store i32 0, i32* %5, align 4
  call void @gsm_write(i32 0)
  br label %after.if

after.if:                                         ; preds = %else.body, %elif.body2, %elif.body, %if.body
  ret i32 0

elif.cond:                                        ; preds = %if.cond
  %12 = load i32, i32* %3, align 4
  %13 = icmp ne i32 %12, 10
```

```
  %14 = load i32, i32* %2, align 4
  %15 = icmp sge i32 %14, 2
  %16 = or i1 %13, %15
  br i1 %16, label %elif.body, label %elif.cond1

elif.body:                                        ; preds = %elif.cond
  %17 = load i32, i32* %5, align 4
  store i32 1, i32* %5, align 4
  call void @gsm_write(i32 1)
  br label %after.if

elif.cond1:                                       ; preds = %elif.cond
  %18 = load i32, i32* %3, align 4
  %19 = load i32, i32* %4, align 4
  %20 = srem i32 %18, %19
  %21 = load i32, i32* %2, align 4
  %22 = icmp eq i32 %20, %21
  %23 = load i32, i32* %2, align 4
  %24 = load i32, i32* %4, align 4
  %25 = add nsw i32 %23, %24
  %26 = icmp eq i32 %25, 4
  %27 = and i1 %22, %26
  %28 = load i32, i32* %3, align 4
  %29 = icmp sle i32 %28, 10
  %30 = and i1 %27, %29
  br i1 %30, label %elif.body2, label %else.body

elif.body2:                                       ; preds = %elif.cond1
  %31 = load i32, i32* %5, align 4
  store i32 2, i32* %5, align 4
  call void @gsm_write(i32 2)
  br label %after.if
```

```
else.body:                                        ; preds = %elif.cond1
  %32 = load i32, i32* %5, align 4
  store i32 3, i32* %5, align 4
  call void @gsm_write(i32 3)
  br label %after.if
}
```

Output:

```
ali@vostro:~/Desktop/Compiler/Compiler-Project-main$ ./run.sh
The result is: 2
```