

# MIPSSysY Compiler

## 词法分析

词法分析的本质是构造有限状态机扫描字符串，提取单词。本编译器采用正则表达式来提取。

基本架构是：

1. 对于每一种单词，构造一个对应的正则表达式；
2. 采用 `|` 将所有正则表达式按优先级串接起来，形成一个正则表达式：其中优先级是指
  1. 长串优先
  2. 关键字优先

首先构造映射 `Map<Token.Type, Pattern>`，随后遍历映射表并构造 `Pattern`：

```
final Optional<String> make = Token.typePatterns.entrySet().stream()
    .map(e -> "(?<" + e.getKey().toString() + ">" + e.getValue() + ")");
    .reduce((s1, s2) -> s1 + "|" + s2);
```

随后通过 `match` 方法即可提取内容。

## 语法分析

语法分析的本质是构造递归下降程序，分析语法成分。本编译器中的 `ParserController` 内置的所有方法就是完成了递归下降分析。通过递归下降，可以得到具体语法树。

## 语义分析（错误处理）

首先通过将具体语法树转化为抽象语法树。

错误处理：

1. 词法分析阶段，检查字符串的格式问题；
2. 语法分析阶段，检查文法匹配问题；
3. 通过维护符号表 `SymbolTable` 来检查标识符名的使用问题、函数传参的问题。

遍历 AST 的过程中，符号表是动态变化的，每一个语法要素都接受一个 `SymbolTable` 并返回一个新的 `SymbolTable`：

1. 声明、函数定义会使得符号表顶层发生变化；

2. 引入新的作用域时，将使得符号表引入新的层。

这个符号表的本质是栈式符号表。

## 语义分析（生成中间代码）

### 变量与值

中间代码中将使用三种“值”类型：

1. 字值 ( `WordValue` ) : 用于存储值，可用作左值和右值。字符串化后将以 `$` 为前缀。
2. 址值 ( `AddrValue` ) : 用于存储地址，可用作左值和右值。字符串化后将以 `&` 为前缀。
3. 数值 ( `ImmValue` ) : 直接表示数字，可用作右值，不可用作左值。字符串化后即为数值本身。

所有中间代码均直接操作这三种值，以字值和址值为变量。

### 中间代码设计与虚拟机

(类四元式) 中间代码中下述指令可在虚拟机上运行：

指令名	Java 类名	举例
空指令	<code>Nop</code>	<code>nop</code>
声明	<code>Declaration</code>	<code>global var &amp;array%1 [size = 3] 1, 2, 3</code>
二元运算赋值	<code>AssignBinaryOperation</code>	<code>save \$3 &lt;- \$2 add 5</code>
一元运算赋值	<code>AssignUnaryOperation</code>	<code>save \$4 &lt;- pos \$3</code>
直接赋值	<code>Move</code>	<code>temp &amp;5 &lt;- &amp;array%1</code>
条件分支	<code>Branch</code>	<code>branch @label_1 if \$3 eq 0</code>
无条件跳转	<code>Jump</code>	<code>jump @label_2</code>
函数调用	<code>CallFunction</code>	<code>call func</code>
函数入口	<code>FuncEntry</code>	<code>main(para_num: 0)</code>
取参数	<code>ParameterFetch</code>	<code>para \$name%2</code>
压参数	<code>PushArgument</code>	<code>push &amp;array%1</code>
函数返回	<code>Return</code>	<code>return \$ret%2</code>
取内存	<code>Load</code>	<code>temp \$5 &lt;- *&amp;array%1</code>

指令名	Java 类名	举例
存内存	Save	save *addr%2 <- \$7
输入	GetInt	call getInt
格式化输出	Print	printf "%d\n"
退出	Exit	exit

虚拟机 `IntermediateVirtualMachine` 模拟一个具有无限多寄存器的抽象机来运行中间代码，以方便调试。其本质是模拟运行栈。

需要注意的是，所有的中间代码指令均继承 `IntermediateCode` 抽象类，从而提供统一的模拟运行接口。

## 从语法树生成中间代码

事实上，抽象语法树已在错误处理前完成构建，其语法要素均独占一个类，派生自 `SyntaxNode`。转化接口：

```
package frontend;

import exceptions.SySEException;
import midend.LabelTable;
import utils.Pair;

import java.util.LinkedList;
import java.util.List;

public interface SyntaxNode {
    // ...

    Pair<SymbolTable, ICodeInfo> iCode(
        LabelTable lt, // table of labels
        SymbolTable st, // symbol table
        String lpBegin, // if parsing loop, should be the label of loop-begin
        String lpEnd, // if parsing loop, should be the label of loop-end
        int tc // count number of temp variables
    );
}
```

生成中间代码时，应考虑作用域导致变量名重复的问题：

```
int main() {
    int x = 0;
    while (x < 10) {
        {
            int x = 3;
            printf("%d\n", x);
        }
        x = x + 1;
    }
    return 0;
}
```

对此，可作重命名：对于变量 `x`，应重命名为 `x%depth`，其中 `depth` 此变量定义、引用时此符号在栈式符号表中的深度。

## 短路求值

### 条件分支的短路

对于 `if` 条件分支语句的 `&&` 短路，可作如下变换：

```
// before conversion
if (a && b) {
    // then-block
} else {
    // else-block
}

// after conversion
if (a) {
    if (b) {
        // then-block
    } else {
        // else-block
    }
} else {
    // else-block
}
```

对于 `if` 条件分支语句的 `||` 短路，可作如下变换：

```
// before conversion
if (a || b) {
    // then-block
} else {
    // else-block
}

// after conversion
if (a) {
    // then-block
} else {
    if (b) {
        // then-block
    } else {
        // else-block
    }
}
```

若有嵌套，直接递归转化即可。

### 循环的短路

对于 `while` 循环语句的条件，可作如下变换：

```
// before conversion
while (cond) {
    // loop-body
}

// after conversion
while (1) {
    if (cond) {
        // loop-body
    } else {
        break;
    }
}
```

随后使用 `if` 的短路解决方案即可。

## 代码生成 (MIPS)

### 翻译中间代码为 MIPS 汇编

根据中间代码，可生成 MIPS 汇编。为保证中间代码的纯洁性——独立于目标代码存在——因此不在 `IntermediateCode` 抽象类中提供转换到 MIPS 的接口方法，而是在后端反射实现转换。

```

package backend;

import midend.*;
import utils.Pair;

import java.math.BigInteger;
import java.util.*;

public class Translator {
    @FunctionalInterface
    private interface Mapper {
        Pair<Pair<MIPSCode, MIPSCode>, IntermediateCode> trans(
            Translator translator,
            IntermediateCode code
        );
    }

    private static final Map<Class<? extends IntermediateCode>, Mapper> trans =
        Collections.unmodifiableMap(
            new HashMap<Class<? extends IntermediateCode>, Mapper>() {{
                put(AssignBinaryOperation.class, (t, c) -> {/* ... */});
                put(AssignUnaryOperation.class, (t, c) -> {/* ... */});
                put(Branch.class, (t, c) -> {/* ... */});
                put(CallFunction.class, (t, c) -> {/* ... */});
                put(Declaration.class, (t, c) -> {/* ... */});
                put(Exit.class, (t, c) -> {/* ... */});
                put(FuncEntry.class, (t, c) -> {/* ... */});
                put(GetInt.class, (t, c) -> {/* ... */});
                put(Jump.class, (t, c) -> {/* ... */});
                put(Load.class, (t, c) -> {/* ... */});
                put(Move.class, (t, c) -> {/* ... */});
                put(Nop.class, (t, c) -> {/* ... */});
                put(ParameterFetch.class, (t, c) -> {/* ... */});
                put(Print.class, (t, c) -> {/* ... */});
                put(PushArgument.class, (t, c) -> {/* ... */});
                put(Return.class, (t, c) -> {/* ... */});
                put(Save.class, (t, c) -> {/* ... */});
            }});

    // ...

    public String translate() {
        final StringJoiner sj = new StringJoiner("\n");
        // ...
        return sj.toString();
    }

    private Pair<MIPSCode, MIPSCode> emitMIPSCode() {
        // ...
        while (interP != null) {
            // ...
            try {
                nextCode = trans.get(interP.getClass()).trans(this, interP);
            } catch (ClassCastException e) {

```

```
        System.err.println(interP);
        e.printStackTrace();
        System.exit(1);
        return null;
    }
    // ...
}
return Pair.of(firstCode, q);
}

// ...
}
```

## 寄存器分配

这里使用到一个后端接口 `RegScheduler`，用于调度寄存器。接口定义为：

```

package backend;

import midend.IntermediateCode;
import midend.Value;
import utils.Pair;

import java.util.*;

public interface RegScheduler {
    List<Reg> regs = Arrays.asList(/* ... */);
    /**
     * @return first: var-name, second: register-name
     */
    Pair<Value, Reg> overflow(IntermediateCode code, Collection<Reg> holdRegs);

    /**
     * @param name var-name
     * @param holdRegs reg-ignored
     * @return <code>Optional.empty()</code> iff full,
     * <code>Optional.of(reg-name)</code> iff not full
     */
    Optional<Reg> allocReg(Value name, Collection<Reg> holdRegs);

    /**
     * remove the usage of the specified register
     * @param reg the specified register
     */
    void remove(Reg reg);

    /**
     * clear all registers in use
     */
    void clear();

    /**
     * find the register already mapped to <code>name</code>
     * @param name variable name
     * @return if found, return <code>Optional.of(reg)</code>; else,
     * <code>Optional.empty()</code>
     */
    Optional<Reg> find(Value name);

    /**
     * get current mapping between registers and variables
     * @return map between registers and variables
     */
    Map<Reg, Value> current();

    /**
     * switch context to the specified context
     * @param context the specified context
     */
    void switchContext(String context);
}

```

```

    /**
     * tell whether the variable <code>value</code> is active(liveness
     * analysis) at <code>code</code>
     * @param code the specified intermediate code
     * @param value the variable name
     * @return if the variable <code>value</code> is active(liveness
     * analysis) at <code>code</code>, return <code>true</code>;
     * else, return <code>false</code>
     */
    boolean active(IntermediateCode code, Value value);

    /**
     * tell whether the register is used as global register
     * @param reg the specified registers
     * @return if the register is used as global register, return
     * <code>true</code>; else, return <code>false</code>
     */
    boolean isGlobalReg(Reg reg);
}

```

通过这些接口，可以调度 MIPS 寄存器。

## 寄存器溢出与内存控制

在下述情况下，须考虑将寄存器中的值放回内存，并取消调度器中寄存器与变量的绑定：

1. 基本块结束
2. 函数调用

## 代码优化

在 MIPSSyY Compiler 中，实现了下述优化：

- 常量传播
- 复写传播
- 单个中间代码的基本简化
- 死代码删除
- 基于活跃分析的图着色寄存器分配
- 除法优化

## 基本铺垫

在所有优化开始前，均需要做一些基本准备工作：

1. 消除多余标签
2. 简化连续跳转

在做完这两个准备工作后，即可划分基本块。

## 基于数据流分析的优化

在此编译器中，数据流分析的优化，只考虑函数内的优化。对于一个给定的函数，可以划分基本块并建立其流图。建立好流图后，即可进行到达定义分析。随后，即可完成常量传播、复制传播。进行活跃分析，即可完成死代码删除。

需要注意的是，若要进行死代码删除，应将指令替换为 `nop` 并让原有标签仍指向此 `nop`。

删除完死代码后，可以简化一些常数运算表达式（单个中间代码的基本简化）：

1. 一元、二元运算的操作数为常数
2. 条件跳转的条件永真
3. 无条件跳转到下一语句

## 图着色寄存器分配

根据活跃分析，定义局部变量的冲突：在同一点处活跃。因此可由此建立局部变量冲突图。随后使用图着色分配算法即可，对于不得不溢出图中结点的时刻，选用下述函数选择使得函数值最小的对应结点：

$$f(v) = \frac{2^{\text{depth}(v)}}{\deg(v)}$$

其中  $v$  为图中结点， $\text{depth}(\cdot)$  获取此变量的循环深度（若处于多个循环中，则取最大值）， $\deg(\cdot)$  获取此变量在冲突图中的度数。

## 面向 MIPS 体系结构的优化

除法优化参考《Division by Invariant Integers using Multiplication》

Initialization(given constant **sword**  $d$  with  $d \neq 0$ ):

```
int l = max(ceil(log2 |d|), 1);
udword m = 1 + floor(2^{N+l-1}/d);
sword m' = m - 2^N;
sword d_{sign} = XSIGN(d);
int sh_{post} = l - 1;
```

For  $q = \text{TRUNC}(n/d)$ , all **sword**:

```
sword q_0 = n + MULSH(m', n);
q_0 = SRA(q_0, sh_{post}) - XSIGN(n);
q = EOR(q_0, d_{sign}) - d_{sign};
```

其中：

$$\text{XSIGN}(x) = \begin{cases} 0 & , x \geq 0 \\ -1 & , x < 0 \end{cases}$$

$\text{EOR}(x, y) = x$  bitwise-xor  $y$  (x xor y)

$\text{SRA}(x, y) = x$  shift-right-arithmetically  $y$  (x sra y)

$\text{MULSH}(x, y) = x$  mult-signed-high  $y$  ((x \* y)[63..32])