附 录

附录 1 MiniJOOL 语言的词法记号类型及标识

词法记号类型标识	含义
CLASS	“class”
STATIC	“static”
FINAL	“final”
EXTENDS	“extends”
VOID	“void”
INT	“int”
BOOLEAN	“boolean”
STRING	“String”
IF	“if”
ELSE	“else”
WHILE	“while”
BREAK	“break”
CONTINUE	“continue”
RETURN	“return”
PRINT	“print”
READ	“read”
THIS	“this”
SUPER	“super”
NEW	“new”
DELETE	“delete”
INSTANCEOF	“instanceof”
LBRACE	“{”
RBRACE	“}”
LPAREN	“(”
RPAREN	“)”
LBRACK	“[”
RBRACK	“]”
SEMICOLON	“;”
COMMA	“,”
DOT	“.”

词法记号类型标识	含义
PLUS	“+”
MINUS	“–”
MULT	“*”
DIV	“/”
NOT	“!”
MOD	“%”
LT	“<”
GT	“>”
LTEQ	“<=”
GTEQ	“>=”
EQEQ	“==”
NOTEQ	“!=”
ANDAND	“&&”
OROR	“‖”
EQ	“=”
MULTEQ	“*=”
DIVEQ	“/=”
MODEQ	“%=”
PLUSEQ	“+=”
MINUSEQ	“– =”
TILDE	“~”
IDENTIFIER	标识符
INTEGER_LITERAL	整数，包括十进制、八进制(以 0 开头)、十六进制(以 0x 或 0X 开头)
BOOLEAN_LITERAL	“true”或“false”
STRING_LITERAL	双引号包含的字符序列
NULL_LITERAL	“null”

附录 2　运算符的优先级与结合性

优先级	类型	运算符	名称	结合性
1		( ) [ ] .	圆括号 下标 成员	自左至右
2	单目	+ − !	正 负 逻辑非	自右至左
3	算术(双目)	* / %	乘 除 取余	自左至右
4	算术(双目)	+ −	加 减	自左至右
5	数值比较 (双目)	< <= > >=	小于 小于或等于 大于 大于或等于	自左至右
6	数值相等 (双目)	== !=	等于 不等于	自左至右
7	逻辑 (双目)	&&	逻辑与	自左至右
8	逻辑 (双目)	||	逻辑或	自左至右
9	赋值 (双目)	= *=　/=　+=　−=　%=	赋值或 算术赋值	自右至左

附录 3　MLex 词法规范语言的 EBNF 表示[1]

你可以通过下面这个简洁的文法来快速了解 MLex 词法规范语言的语法特征，这个文法是二义的。

program	=	“class” IDENTIFIER “{” method_decl “}”
method_decl	=	“static” “void” “main” “(” “)” “{” { macro_decl } { rule } “}”
macro_decl	=	assignment “;”
rule	=	“if” “(” expression “)” block
block	=	“{” { statement } “}”
statement	=	| assignment “;”
		| “return” expression “;”
		| “print” “(” expression “)” “;”
expression	=	IDENTIFIER
		| literal
		| “(” expression “)”
		| expression binary_operator expression
		| assignment_expression
binary_operator	=	“*” | “/” | “+” | “–” | “|”
assignment	=	IDENTIFIER “=” expression
literal	=	INTEGER_LITERAL　(注：这里只考虑十进制非负整数)

[1] 根据 ISO/IEC 14977:1996(E)规定的 EBNF 语法（http://www.cl.cam.ac.uk/~mgk25/iso-ebnf.html），其中终结符用一对双引号括起，包含在一对方括号内的部分为可选的，包含在一对花括号内的部分表示该部分重复 0 次或多次。

附录 4　SimpleMiniJOOL 语言语法的 EBNF 表示

你可以通过下面这个简洁的文法来快速了解 SimpleMiniJOOL 语言的语法特征，这个文法是二义的。你需要结合算符的优先级和结合性规则，以及 if-else 的就近匹配原则来理解这个文法的具体语义。

```
                program  =  class_declaration
      class_declaration  =  “class” IDENTIFIER “{” “static” “void” “main” “(” “)” block “}”
                  block  =  “{” { statement } “}”
              statement  =  “;”
                        | assignment_expression “;”
                        | “break” “;”
                        | “continue” “;”
                        | “return” “;”
                        | block
                        | “if” “(” expression “)” statement
                        | “if” “(” expression “)” statement “else” statement
                        | “while” “(” expression “)” statement
                        | “print” “(” expression “)” “;”
                        | “read” “(” IDENTIFIER “)” “;”
             expression  =  IDENTIFIER
                        | literal
                        |   “(” expression “)”
                        | unary_operator expression
                        | expression binary_operator expression
                        | assignment_expression
          unary_operator  =  “+” | “–” | “!”
         binary_operator  =  “*” | “/” | “%” | “+” | “–”
                        | “==” | “!=” | “<” | “<=” | “>” | “>=” | “||” | “&&”
assignment_expression  =  IDENTIFIER assignment_operator assignment_expression
      assignment_operator  =  “=” | “*=” | “/=” | “%=” | “+=” | “–=”
                  literal  =  INTEGER_LITERAL
```

附录 5　SkipOOMiniJOOL 语言语法的 EBNF 表示

```
                program = class_declaration
        class_declaration = “class” IDENTIFIER class_body
             class_body = “{” class_body_declaration { class_body_declaration} “}”
  class_body_declaration = field_declaration
                         | method_declaration
      field_declaration = [ “static” ] [ “final” ] type variable_declarators “;”
     variable_declarators = variable_declarator { “,” variable_declarator }
      variable_declarator = variable_declarator_id [ “=” variable_initializer ]
   variable_declarator_id = IDENTIFIER
    variable_initializer = expression | array_initializer
        array_initializer = “{” expression { “,” expression } “}”
       method_declaration = method_header    method_body
         method_header = [ “static” ] ( type | “void” ) method_declarator
       method_declarator = IDENTIFIER “(” [formal_parameter_list] “)”
   formal_parameter_list = formal_parameter { “,” formal_parameter }
      formal_parameter = type variable_declarator_id
           method_body = block
                 block = “{” [block_statements] “}”
       block_statements = block_statement { block_statement }
        block_statement = local_variable_declaration_statement
                        | statement
local_variable_declaration_statement = type variable_declarators “;”
             statement = “;”
                       | statement_expression “;”
                       | “break” “;”
                       | “continue” “;”
                       | “return” [expression] “;”
                       | block
                       | “if” “(” expression “)” statement
                       | “if” “(” expression “)” statement “else” statement
                       | “while” “(” expression “)” statement
                       | “print” “(” expression “)” “;”
                       | “read” “(” lvalue “)” “;”
  statement_expression = assignment_expression
                       | method_invocation
```

```
primary = literal
        | "(" expression ")"
        | method_invocation
        | array_access
method_invocation = IDENTIFIER "(" [ argument_list ] ")"
argument_list = expression { "," expression }
array_access = IDENTIFIER "[" expression "]"
expression = IDENTIFIER
           | IDENTIFIER "." "length"
           | primary
           | unary_operator expression
           | expression binary_operator expression
           | assignment_expression
unary_operator = "+" | "-" | "!"
binary_operator = "*" | "/" | "%" | "+" | "-"
                | "==" | "!=" | "<" | "<=" | ">" | ">=" | "||" | "&&"
lvalue = IDENTIFIER | array_access | "(" lvalue ")"
assignment_expression = lvalue assignment_operator assignment_expression
assignment_operator = "=" | "*=" | "/=" | "%=" | "+=" | "-="
type = primitive_type
     | "String"
     | array_type
primitive_type = "int" | "boolean"
array_type = ( primitive_type | "String") "[" [ constant_expression ] "]"
constant_expression = expression
literal = INTEGER_LITERAL
        | BOOLEAN_LITERAL
        | STRING_LITERAL
```

附录 6　MiniJOOL 语言语法的 EBNF 表示

program	=	class_declaration　{ class_declaration }
class_declaration	=	“class” IDENTIFIER [“extends” IDENTIFIER] class_body
class_body	=	“{” class_body_declaration { class_body_declaration} “}”
class_body_declaration	=	field_declaration
		| method_declaration
		| constructor_declaration
		| destructor_declaration
field_declaration	=	[“static”] [“final”] type variable_declarators “;”
variable_declarators	=	variable_declarator { “,” variable_declarator }
variable_declarator	=	variable_declarator_id [“=” variable_initializer]
variable_declarator_id	=	IDENTIFIER
variable_initializer	=	expression | array_initializer
array_initializer	=	“{” [expression { “,” expression }] “}”
method_declaration	=	method_header method_body
method_header	=	[“static”] (type | “void”) method_declarator
method_declarator	=	IDENTIFIER “(” [formal_parameter_list] “)”
formal_parameter_list	=	formal_parameter { “,” formal_parameter }
formal_parameter	=	type variable_declarator_id
method_body	=	block
constructor_declaration	=	constructor_declarator constructor_body
constructor_declarator	=	IDENTIFIER “(” [formal_parameter_list] “)”
constructor_body	=	“{” [explicit_constructor_invocation] [block_statements] “}”
explicit_constructor_invocation	=	(“this” | “super”) “(” [argument_list] “)” “;”
destructor_declaration	=	destructor_declarator destructor_body
destructor_declarator	=	“~” IDENTIFIER “(” “)”
destructor_body	=	block
block	=	“{” [block_statements] “}”
block_statements	=	block_statement { block_statement }
block_statement	=	local_variable_declaration_statement | statement
local_variable_declaration_statement		type variable_declarators “;”
statement	=	“;” | statement_expression “;”
		| “break” “;”
		| “continue” “;”
		| “return” [expression] “;”
		| block

| “if” “(” expression “)” statement
| “if” “(” expression “)” statement “else” statement
| “while” “(” expression “)” statement
| “print” “(” expression “)” “;”
| “read” “(” lvalue “)” “;”
| “delete” expression “;”

statement_expression = assignment_expression
| method_invocation

primary = literal
| “(” expression “)”
| method_invocation
| array_access
| “this”
| class_instance_creation_expression
| field_access

class_instance_creation_expression = “new” class_type “(” [argument_list] “)”

field_access = (primary | “super”) “.” IDENTIFIER

method_invocation = (name | (“super” | primary) “.” IDENTIFIER)
“(” [argument_list] “)”

argument_list = expression { “,” expression }

array_access = (name | field_access) “[” expression “]”

expression = name | primary
| unary_operator expression
| expression binary_operator expression
| expression “instanceof” class_type
| assignment_expression
| cast_expression

cast_expression = “(” class_type “)” (name | primary)

unary_operator = “+” | “–” | “!”

binary_operator = “*” | “/” | “%” | “+” | “–”
| “==” | “!=” | “<” | “<=” | “>” | “>=” | “||” | “&&”

lvalue = name | field_access | array_access | “(” lvalue “)”

assignment_expression = lvalue assignment_operator assignment_expression

type = primitive_type | “String” | array_type | class_type

primitive_type = “int” | “boolean”

array_type = (primitive_type | “String”) “[” [constant_expression] “]”

class_type = IDENTIFIER

```
constant_expression = expression
literal = INTEGER_LITERAL
        | BOOLEAN_LITERAL
        | STRING_LITERAL
        | NULL_LITERAL
name = IDENTIFIER | name "." IDENTIFIER
```

附录 7 与本书有关的 Eclipse AST 节点类及其含义

AST 节点类	用处	说明
ArrayAccess	数组元素访问表达式	例如：a[10]
ArrayInitializer	数组初始化表达式	例如：int[3] i = {1,2,3}中的{1,2,3}。
ArrayType	数组类型	SkipOOMiniJOOL 和 MiniJOOL 语言中可能需要指定数组长度，这时可以在节点上增加名为 LENGTH 的自定义属性。
Assignment	赋值表达式	例如：a = 2 或者 a+=2。
Block	语句块	用于表示一个由花括号括起来的语句序列。
BooleanLiteral	布尔型常量	表示 true 或 false。
BreakStatement	break 语句	表示一个 break 语句。
CastExpression	cast 表达式	表示一个强制类型转换表达式，如：(A)oB
ClassInstanceCreation	类实例创建表达式	表示一个 new 表达式，例如：new A(p1, p2)。
CompilationUnit	表示整个程序	是 AST 的根节点类型
ConstructorInvocation	构造器调用表达式	指形如 this(p1,p2)的构造器调用。
ContinueStatement	continue 语句	表示一个 continue 语句。
EmptyStatement	空语句	表示一个空语句。
Expression	表达式节点类的抽象基类	
ExpressionStatement	表达式语句	将一个表达式封装成语句。
FieldAccess	域访问表达式	表示对实例变量的访问，对于类变量的访问则使用 QualifiedName。FieldAccess 类还可以用于表示“<数组名>.length”。
FieldDeclaration	域声明语句	表示 SkipOOMiniJOOL 语言和 MiniJOOL 语言中的域声明，前者只有类变量，后者有类变量和实例变量。
IfStatement	if 语句	表示一个 if 语句。
InfixExpression	中缀表达式	表示中缀表达式。
InstanceofExpression	instanceof 表达式	例如：a instanceof A。
MethodDeclaration	方法声明	用于表示方法声明、构造器声明和析构器声明。方法声明同时包括了方法的定义。
MethodInvocation	方法调用表达式	例如：o.m(a1, a2)。
Modifier	修饰符	在 SkipOOMiniJOOL 和 MiniJOOL 中 modifier 包括：static 和 final。
Name	名字节点类的基类	是 QualifiedName 和 SimpleName 的基类。
NullLiteral	null 常量	表示 null。

AST 节点类	用处	说明
NumberLiteral	数字常量	在 SkipOOMiniJOOL 和 MiniJOOL 语言中仅仅指整型常量。
ParenthesizedExpression	括号表达式	例如：(a+b)。
PrefixExpression	前缀表达式	表示一个前缀表达式，如+5 等。
PrimitiveType	基本类型	描述 SkipOOMiniJOOL 和 MiniJOOL 语言中的 int、boolean 和 void 类型。
QualifiedName	限定名	在 MiniJOOL 语言中指形如 A.id 的表达式。其中 A 是类型名，id 是 A 中的成员。
ReturnStatement	return 语句	表示一个 return 语句。
SimpleName	简单名称	通常表示变量名、方法名等。
SimpleType	简单类型	在 SimpleMiniJOOL 和 MiniJOOL 语言中用来表示不在 PrimitiveType 中的类型，如 String、类类型。
SingleVariableDeclaration	单变量声明表达式	可用于描述一个形参声明。
Statement	语句节点类的抽象基类	
StringLiteral	字符串常量	表示字符串常量，如："abd"。
SuperConstructorInvocation	超类构造器调用表达式	指形如 super(p1, p2)的构造器调用。
SuperFieldAccess	超类变量访问表达式	指形如 super.id 的超类变量访问。
SuperMethodInvocation	超类方法调用表达式	指形如 super.f( )的超类方法调用。
ThisExpression	this 表达式	表示“this”。
Type	各种类型的抽象基类	Primitivetype、SimpleName、ArrayType 的抽象基类
TypeDeclaration	类型声明	在 SkipOOMiniJOOL 和 MiniJOOL 语言中用于表示一个类声明。
VariableDeclarationExpression	变量声明表达式	表示一个变量声明表达式，例如：表示“int a=10,b,c=1;”中的“int a=10,b,c=1”。
VariableDeclarationFragment	变量声明中声明的单个变量及其初始化	例如：表示“int a=10,b,c=1;”中的“a=10”。
VariableDeclarationStatement	变量声明语句	表示一个变量声明语句，例如：表示“int a=10,b,c=1;”。
WhileStatement	while 语句	表示一个 while 语句。

附录 8　MIPS-SPIM 汇编语言的 EBNF 定义

注：该文法只包含 MIPS-SPIM 汇编语言中常用的伪指令以及汇编指令。

Program	=	{Line} EOF
Line	=	Statement [Comment] EOL
Statement	=	ε | Directive | Label | Instruction
Comment	=	以#开始的任意字符串
Directive	=	Director [Argument {“,” Argument}]
Director	=	“.align” | “.ascii” | “.asciiz” | “.byte” | “.comm” | “.data” | “.end” | “.ent” | “.globl” | “.space” | “.text” | “.word”
Argument	=	Integer | String | Symbol
Label	=	Symbol “:”
Instruction	=	Opcode [Operand [“,” Operand [“,” Operand]]]
Opcode	=	“move” | “la” | “li” | “lw” | “sw” | “add” | “addu” | “sub” | “subu” | “mulo” | “mulou” | “div” | “divu” | “rem” | “remu” | “abs” | “neg” | “negu”| “and” | “or” | “xor” | “nor” | “not” | “rol” | “ror” | “sll” | “sra” | “srl” | “seq” | “sge” | “sgt” | “sle” | “slt” | “sne” | “b” | “beq” | “bge” | “bgt” | “ble” | “blt” | “bne” | “j” | “jal” | “callout” | “nop” | “syscall”
Operand	=	Register | Integer | Symbol | Address
Register	=	“$zero” \| “$at” | “$v0” \| “$v1” | “$v2” \| “$a0” | “$a1” \| “$a2” | “$a3” \| “$t0” | “$t1” \| “$t2” | “$t3” \| “$t4” | “$t5” \| “$t6” | “$t7” \| “$t8” | “$t9” \| “$s0” | “$s1” \| “$s2” | “$s3” \| “$s4” | “$s5” \| “$s6” | “$s7” \| “$k0” | “$k1” \| “$gp” | “$sp” \| “$fp” | “$ra” \| “$” RegNo
RegNo	=	0~31 的整数
Integer	=	[“-”] Dec {Dec} | “0” (“x”|”X”) (Dec|Hex) {Dec|Hex}
Dec	=	“0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”
Hex	=	“A” | “B” | “C” | “D” | “E” | “F” | “a” | “b” | “c” | “d” | “e” | “f”
String	=	由一对双引号括起的任意字符串，其中特殊字符使用转义字表示： newline　\n tab　\t quote　\” slash　\\ null　\0
Symbol	=	((Letter | “_” | “$” \| “_”) {Letter \| Dec \| “_” \| “$” | “_” }) – (Opcode | Director | Register)
Address	=	[Integer | ε] “(” Register “)” | Symbol [(“+” | “-”) [Integer | ε] “(” Register “)”]
Letter	=	“A”～“Z”和“a”～“z”之一
EOL	=	表示行结束
EOF	=	表示文件结尾

附录 9　采用 AT&T 语法的 x86 汇编语言的 EBNF 定义

注：该文法只包含常用的伪指令以及汇编指令。

Program = {Line} EOF

Line = Statement [Comment] EOL

Statement = ε | Directive | Label | Instruction

Comment = 以#开始的任意字符串

Directive = Director [Argument {“,” Argument}]

Director = “.align” | “.ascii” | “.asciiz” | “.balign” | “.byte” | “.comm” | “.data” | “.end” | “.endfunc” | “.func” | “.globl” | “.long” | “.size” | “.space” | “.string” | “.text” | “.word”

Argument = Integer | String | Symbol

Label = Symbol “:”

Instruction = Opcode [Operand [“,” Operand [“,” Operand]]]

Opcode = “mov” | “movl” | “movb” | “mow” | “movsbw” | “movsbl” | “movswl” | “movzbw” | “movzbl” | “movzwl” | “lea” | “leal” | “leaw” | “push” | “pushl” | “pushw”| “pop” | “popl” | “popw” | “pusha” | “pushal” | “pushaw” | “popa” | “popal” | “popaw” | “lahf” | “sahf” | “pushf” | “pushfl” | “pushfw” | “popf” | “popfl” | “popfw” | “incb” | “incw” | “inc” | “incl” | “decb” | “decw” | “dec” | | “decl” | “neg” | “negl” | “negb” | “negw” | “not” | “notl” | “notb” | “notw” | “add” | “addl” | “addb” | “addw” | “sub” | “subl” | “subb” | “subw” | “imul” | “imull” | “imulb” | “imulw” | “idiv” | “idivl” | “idivb” | “idivw” | “xor” | “xorl” | “xorb” | “xorw” | “or” | “orl” | “orb” | “orw” | “and” | “andl” | “andb” | “andw” | “sal” | “sall” | “salb” | “salw” | “shl” | “shll” | “shlb” | “shlw” | “sar” | “sarl” | “sarb” | “sarw” | “shr” | “shrl” | “shrb” | “shrw” | “test” | “testl” | “testb” | “testw” | “cmp” | “cmpl” | “cmpb” | “cmpw” | “cmc” | “clc” | “stc” | “jmp” | “jl” | “jg” | “jge” | “jeq” | “jne” | “jecxz” | “call” | “ret” | “loop” | “loopz” | “loopnz” | “loope” | “loopne” | “int” | “nop”

Operand = Register | Integer | Symbol | Address

Register = “%eax” | “%ebx” | “%ecx” | “%edx” | “%esi” | “%edi” | “%ebp” | “%esp” | “%ax” | “%bx” | “%cx” | “%dx” | “%si” | “%di” | “%bp” | “%sp” | “%ah” | “%al” | “%bh” | “%bl” | “%ch” | “%cl” | “%dh” | “%dl” | “%eflags” | “%flags” | “%cs” | “%ds” | “%ss” | “%es” | “%fs” | “%gs”

Integer = [“-”] Dec {Dec}
| “0” (“x”|”X”) (Dec|Hex) {Dec|Hex}

Dec = “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”

Hex = “A” | “B” | “C” | “D” | “E” | “F” | “a” | “b” | “c” | “d” | “e” | “f”

String = 由一对双引号括起的任意字符串，其中特殊字符使用转义字表示：

newline	\n
tab	\t
quote	\”
slash	\\
null	\0

Symbol = ((Letter | “_” | “$” | “_”) {Letter | Dec | “_” | “$” | “_” }) – (Opcode | Director | Register)

Address = [Integer | ε] “(” Register “)”
| Symbol [(“+” | “-”) [Integer | ε] “(” Register “)”]

Letter = “A”～“Z”和“a”～“z”之一

EOL = 表示行结束

EOF = 表示文件结尾

附录 10　实验软件包提供的可用编译器组件

这些可用编译器组件统一打包在实验软件包的 lib/compiler.jar 中。下面分别列出各编译器组件接口所关联的具体可用组件。

分析器接口：edu.ustc.cs.compile.platform.interfaces.ParserInterface

接口方法：public InterRepresent doParse(File src) throws ParserException

可用分析器组件类名	说明
edu.ustc.cs.compile.parser.simpleminijool.Parser	SimpleMiniJOOL 语言语法分析器
edu.ustc.cs.compile.parser.skipoominijool.Parser	SkipOOMiniJOOL 语言语法分析器
edu.ustc.cs.compile.parser.minijool.Parser	MiniJOOL 语言语法分析器

变换器接口：edu.ustc.cs.compile.platform.interfaces.TransformerInterface

接口方法：public InterRepresent transform(InterRepresent ir) throws TransformerException

可用变换器组件类名	说明
edu.ustc.cs.compile.ast2lir.skipoominijool.AST2LIR	SkipOOMiniJOOL 语言的 AST 到 LIR 的转换器
edu.ustc.cs.compile.ast2lir.minijool.AST2LIR1	MiniJOOL-I 语言的 AST 到 LIR 的转换器
edu.ustc.cs.compile.ast2lir.minijool.AST2LIR2	MiniJOOL-II 语言的 AST 到 LIR 的转换器
edu.ustc.cs.compile.ast2lir.minijool.AST2LIR3	MiniJOOL-III 语言的 AST 到 LIR 的转换器
edu.ustc.cs.compile.ast2lir.minijool.AST2LIR	MiniJOOL 语言的 AST 到 LIR 的转换器