C -Minus Language Documentation

Why C-Minus

Originally the reason for choosing choosing a C like language was for the familiarity. Our team wanted to work with languages we had seen before. C-minus was to be the target language, java the implementation language and ARM the output language. Each of us had some level of experience with each of those languages. Another reason for choosing a C like language for our input was to avoid some of the difficulties of compiling to arm from an even higher level language. The reason for the “minus” Was purely simplification for the type checker and code generation. A parser and lexer for full C was possible using a publicly available C grammar that is written for Antlr.

Code

This first snippet shows the types of statements that can be made in the language.

```
public void compileStatement(final Stmt stmt) {
        if (stmt instanceof VariableDeclarationInitializationStmt) {

compileVariableDeclarationInitializationStmt((VariableDeclarationInitializationStmt)stmt);
        } else if (stmt instanceof AssignmentStmt) {
            compileAssignmentStmt((AssignmentStmt)stmt);
        } else if (stmt instanceof SequenceStmt) {
            compileSequenceStmt((SequenceStmt)stmt);
        } else if (stmt instanceof PrintStmt) {
            compilePrintStmt((PrintStmt)stmt);
        } else if (stmt instanceof ReturnExpStmt) {
            compileReturnExpStmt((ReturnExpStmt)stmt);
        } else if (stmt instanceof ReturnVoidStmt) {
            compileReturnVoidStmt((ReturnVoidStmt)stmt);
        } else if (stmt instanceof FunctionCallStmt) {
            compileFunctionCallStmt((FunctionCallStmt)stmt);
        } else if (stmt instanceof IfStmt) {
            compileIfStmt((IfStmt)stmt);
        } else if (stmt instanceof WhileStmt) {
            compileWhileStmt((WhileStmt)stmt);
        } else {
```

```
            assert(false);
        }
    }
```

The next snippet shows the Types of Expressions

```
public void compileExpression(final Exp exp) {
        if (exp instanceof IntExp) {
            compileIntExp((IntExp)exp);
        } else if (exp instanceof BoolExp) {
            compileBoolExp((BoolExp)exp);
        } else if (exp instanceof BinopExp) {
            compileBinopExp((BinopExp)exp);
        } else if (exp instanceof VariableExp) {
            compileVariableExp((VariableExp)exp);
        } else if (exp instanceof DereferenceExp) {
            compileDereferenceExp((DereferenceExp)exp);
        } else if (exp instanceof AddressOfExp) {
            compileAddressOfExp((AddressOfExp)exp);
        }else if (exp instanceof FunctionCallExp) {
            compileFunctionCallExp((FunctionCallExp)exp);
        } else {
            assert(false);
        }
    }
```

This snippet bit shows the limited operations

```
 public void compileOp(final MIPSRegister destination,
                       final MIPSRegister left,
                       final Op op,
                       final MIPSRegister right) {
        if (op instanceof PlusOp) {
            add(new Add(destination, left, right));
        } else if (op instanceof EqualsOp) {
            add(new Seq(destination, left, right));
        } else {
            assert(false);
        }}
```

Known Limitations

The C-minus language has many limitations. In general the point of higher level languages is to abstract what is going on at a machine level and to allow the programmer to give instructions without needing to know exactly how they will be carried out. Our language does this in some ways, but in other ways it is more restrictive than the assembly language we compile to.
There are no subtraction, division or multiplication operators. Our language does have the boolean data type, however the only conditional is the '==' operator.
Another limitation of our language results from the data types used. The user must use integers for any arithmetic compilation. Essentially expressions in our language can be integer, boolean, variable or a function call.

What Would We Do Differently

The biggest challenge on this project was understanding the code generation when dealing with a stack in MIPS.
Starting out we chose assembly thinking that it would simplify our project. I naively had this thought. "Oh, that shouldn't be to bad. If we have x = 2 + 3, we just have to load a register with 2, load a register with 3 and then add those registers and have x point to the register that the result is stored in." (dumb)
At least for me personally I would have not done assembly if I could start over. I'm also on the fence about using Antlr only because of integration problems.
On a much smaller not I think I need to cough up the money for IntelliJ. I'm not a fan of netbeans, but I'm even less of a fan of Eclipse.

Compiling and Running

The project is a Maven project. All that needs to be done to compile and run it is to import the project, right click it and select build. As far as running an input file from the lexer all the way to the assembly output…...It doesn't.

Syntax

The lexer and parser are not integrated with the rest of the project. The grammar that was used for Antlr will follow the syntax that that better represents the type checker and code generation.

I is an integer
var is a variable
fn is a function name
type ::= int | void | bool | // basic primitive types
        type* // pointers
op ::= + | ==
lhs ::= var | *lhs // used on left-hand side of assignment
exp ::= i | true | false | var |
       exp op exp |
       fn(exp*) | // calls a function
       (type)exp | // cast
       &lhs | // address-of (reference)
       *exp | // dereference
VarDec ::= type var
stmt ::= if (exp) { stmt } else { stmt } |
        while (exp) { stmt } |
        varDec = exp | // combined variable declaration and initialization
        lhs = exp | // assignment
        return | // return void
        return exp | // return a value
        stmt ; stmt // one statement followed by another
structDec ::= sn { varDec* }
fDef ::= type fn(varDec*) { stmt }

Grammar for Antlr starts on next page

```
grammar Cminus;

startRule
	:	type main LEFTCURLY progStatements RIGHTCURLY EOF;

main
	:	MAIN;

progStatements
	:	statement| statement progStatements| loops| loops progStatements;

statement
	:	(varDec|varAssign|printStatement) NULLCHAR | (funcDecParam|
funcCallParam| funcPointerDec);

funcPointerDec
	:	type LEFTPAREN POINTDEC varDec RIGHTPAREN ASSIGN POINT var;

type	:
		TYPE;

funcCallParam
	:	var LEFTPAREN (var|exp) RIGHTPAREN | funcPointer;

funcPointer
	:	POINTDEC var LEFTPAREN (var|exp) RIGHTPAREN;

funcDecParam
	:	type var LEFTPAREN varDec RIGHTPAREN LEFTCURLY progStatements
		RETURN (var|exp|funcPointer) NULLCHAR RIGHTCURLY
	|	VOID var LEFTPAREN varDec RIGHTPAREN LEFTCURLY progStatements
RIGHTCURLY;

printStatement
	:	PRINT LEFTPAREN STRING RIGHTPAREN;

varDec
	:	type var|type var NULLCHAR;
```

```
varAssign
	:	var assignment mathExpr;

var	:	VAR;

assignment
	:	ASSIGN| assignOp;

mathExpr
	:	exp| exp op exp| var op exp| exp op var| var;

op
	:	PLUS| MINUS | DIVIDE | MULT;

exp
	:	INT|CHAR;

assignOp
	:	PLUSEQUAL| MINUSEQUAL | MULTEQUAL | DIVIDEQUAL;

loops
	:	Ifloop LEFTPAREN ifConditions RIGHTPAREN LEFTCURLY
progStatements RIGHTCURLY
	|	Ifloop LEFTPAREN ifConditions RIGHTPAREN LEFTCURLY progStatements
RIGHTCURLY
		Elseloop LEFTCURLY progStatements RIGHTCURLY
	|	Forloop LEFTPAREN forConditions RIGHTPAREN LEFTCURLY
progStatements RIGHTCURLY;

ifConditions
	:	var compare (exp|var);

forConditions
	:	var compare exp;
compare
	:	LESSTHAN| GREATERTHAN| LESSEQUAL| GREATEREQUAL;


TYPE :  'int'|'char';
```

```
VOID : 'void';
RETURN : 'return'
       ;
POINTDEC :'&'
         ;
POINT : '@'
      ;
LEFTPAREN : '('
          ;
RIGHTPAREN : ')'
           ;
Ifloop  : 'if'
           ;
Elseloop : 'else'
            ;
Forloop          : 'for'
           ;
PRINT    : 'printf'
           ;
MAIN: 'main';
VAR: [a-z]+;
LEFTCURLY: '{';
RIGHTCURLY: '}';
PLUS : '+';
MINUS : '-';
MULT : '*';
DIVIDE : '/';
PLUSEQUAL : '+=';
MINUSEQUAL :'-=';
MULTEQUAL :'*=';
DIVIDEQUAL : '/=';
LESSTHAN : '<';
GREATERTHAN : '>';
LESSEQUAL : '<=';
GREATEREQUAL : '>=';
ASSIGN    : '=';

ID  :    ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
    ;
```

```
INT :   '0'..'9'+
    ;
COMMENT
    :   '//' ~('\n'|'\r')* '\r'? '\n' ->skip
    ;
WS  :   ( ' '
        | '\t'
        | '\r'
        | '\n'
        ) ->skip
    ;
STRING
    :  '"' ( ESC_SEQ | ~('\\'|'"') )* '"'
    ;
CHAR:  '\'' ( ESC_SEQ | ~('\''|'\\') ) '\''
    ;
fragment
HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;
fragment
ESC_SEQ
    :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\\')
    |   UNICODE_ESC
    |   OCTAL_ESC
    ;
fragment
OCTAL_ESC
    :   '\\' ('0'..'3') ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7')
            ;
fragment
UNICODE_ESC
    :   '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
    ;
NULLCHAR : ';';
skip    :
            ;
```