---------- Action.gram -------------

//----------------------------------

// ACTION! compiler for the 6502

// Version 0.0.1 Nov 27, 2024

//----------------------------------

TERMINALS

"IDENT" = 256,

"NUMBER",

"STRING",

"VECTOR",

"MODULE",

"RETURN",

"PROC",

"FUNC",

"INTERRUPT",

"INTERRUPT\_IDENT",

"PROC\_IDENT",

"FOR",

"WHILE",

"DO",

"OD",

"BEGIN",

"END",

"EXIT",

"IF",

"THEN",

"ELSEIF",

"ELSE",

"FI",

"IFF",

"FFI",

"UNTILL",

"TO",

"STEP",

"OR", // Logical OR operator

"AND", // Logical AND operator or cpu AND opcode

"GTEQ",

"LTEQ",

"DEFINE",

"TYPE",

"RECORDTYPE",

"BOOL",

"CHAR",

"BYTE",

"CARD",

"INT",

"POINTER",

"ARRAY",

"PUSH",

"POP",

"BREAK",

"BITS",

"ACC",

"XREG",

"YREG",

"ASM",

//Assembler Tokens

"SECTION",

"START",

"SIZE",

"MODE",

"ZEROPAGE",

"READ\_ONLY",

"READ\_WRITE",

"true",

"false",

"ORG",

"DB",

"DW",

"DL",

"DAS", //ACTION! STRING

"DCS", // 'C' STRING

"DS", // DEFINE STORAGE, BLOCK OF MEMORY

"ASMPROC",

"EPROC",

"LOCAL\_LABLE",

"GLOBAL\_LABLE",

"GLOBAL\_LABLE",

"ADC",

// "AND", // see logical AND

"ASL",

"BCC",

"BCS",

"BEQ",

"BMI",

"BNE",

"BPL",

"BVC",

"BVS",

"BIT",

"BRK",

"CLC",

"CLD",

"CLI",

"CLV",

"CMP",

"CPX",

"CPY",

"DEC",

"DEX",

"DEY",

"EOR",

"INC",

"INX",

"INY",

"JMP",

"JSR",

"LDA",

"LDX",

"LDY",

"LSR",

"NOP",

"ORA",

"PHA",

"PLA",

"PHP",

"PLP",

"ROL",

"ROR",

"RTI",

"RTS",

"SBC",

"STA",

"SEC",

"SED",

"SEI",

"STX",

"STY",

"TAX",

"TAY",

"TXA",

"TYA",

"TXS",

"TSX",

"X",

"Y",

"A",

"SR", // Status Register

"@", // address of operator

".", // record field operator

"^", // pointer dereference

"!",

"XOR",

"&", // bitwise AND

"%", // bitwise OR

"#", // Not equals

">", // greater than

"<", // less than

"{",

"}",

"=", // assignment or equal to

"==+",

"==-",

"==\*",

"==/",

"==MOD",

"==&",

"==!",

"==XOR",

"==%",

"==LSH",

"==RSH",

"[",

"]",

",",

";",

"+",

"-",

"\*",

"/",

"MOD",

"LSH",

"RSH",

"(",

")",

"NEG", //cpu 'N'eg status register flage

"CARRY", // cpu 'C'arry status register flag

"ZERO", // cpu 'Z'ero status register flag

"OVERFLOW" // cpu 'O'verflow status register flag

GRAMMAR

//--------------------------------

// Program Structure

//--------------------------------

Action65 -> Modules;

Modules -> Vector Modules\_1;

Modules\_1 -> 'MODULE' Vector Modules\_1

-> .

;

Vector -> SysDecl Vector\_1;

Vector\_1 -> 'VECTOR' VectorAddress SysDecl Vector\_1

-> .

;

VectorAddress -> '(' CompConst ')' VectorData;

VectorData -> '=' CompConst;

//------------------------------------------

// Statements

//------------------------------------------

//---------------------------------------

// Proceedure/Function Call

//---------------------------------------

Statements ->ForStmt ProcCall\_1;

ProcCall\_1 -> 'PROC\_IDENT' ProcParams ForStmt ProcCall\_1

-> 'FUNC\_IDENT' ProcParams ForStmt ProcCall\_1

-> .

;

ProcParams -> '(' ValueList ')'

-> .

;

//------------------------------------------------

// FOR statement

//------------------------------------------------

ForStmt -> IfStmt ForStmt\_1;

ForStmt\_1 -> 'FOR' STEPoption DoStmt ForStmt\_1

-> .

;

STEPoption -> ForTO STEPoption\_1;

STEPoption\_1 -> 'STEP' ArithExpr

-> .

;

ForTO -> Itterrator ForTO\_1;

ForTO\_1 -> 'TO' ArithExpr

-> .

;

Itterrator -> MemContentsType Itterrator\_1;

Itterrator\_1 -> '=' ArithExpr

-> .

;

//------------------------------------------------

// IF statement

//------------------------------------------------

IfStmt -> Iff IfStmt\_1;

IfStmt\_1 -> 'IF' EndIF Iff IfStmt\_1

-> .

;

EndIF -> ElsePart EndIF\_1;

EndIF\_1 -> 'FI';

ElsePart -> ElseIfPart ElsePart\_1;

ElsePart\_1 -> 'ELSE' Statements

-> .

;

ElseIfPart -> ThenPart ElseIfPart\_1;

ElseIfPart\_1 -> 'ELSEIF' ThenPart ElseIfPart\_1

-> .

;

ThenPart -> RelOperation ThenPart\_1;

ThenPart\_1 -> 'THEN' Statements

-> .

;

//-------------------------------------------------

// IFF Statement

//-------------------------------------------------

Iff -> WhileStmt Iff\_1;

Iff\_1 -> 'IFF' IFFend WhileStmt Iff\_1

-> .

;

IFFend -> IFFelse IFFend\_1;

IFFend\_1 -> 'FFI';

IFFelse -> IFFthenpart IFFelse\_1;

IFFelse\_1 -> 'ELSE' Statements

-> .

;

IFFthenpart -> IffConditional IFFthenpart\_1;

IFFthenpart\_1 -> 'THEN' Statements

-> .

;

IffConditional -> Bits IffConditional\_1;

IffConditional\_1-> 'AREG' RelOper

-> 'XREG' RelOper

-> 'YREG' RelOper

-> .

;

RelOper -> '<' Value

-> 'GTEQ' Value

-> '=' Value

-> .

;

Bits ->StatusFlags Bits\_1;

Bits\_1 -> 'BITS' BitValue

-> .

;

BitValue -> '[' ArithExpr ']';

StatusFlags -> OptNot StatusFlags\_1;

StatusFlags\_1 -> 'NEG' //negative flag

-> 'CARRY' //Carry flag

-> 'ZERO' //Zero

-> 'OVERFLOW' //Overflow

;

OptNot -> '^'

-> .

;

//-------------------------------------------------

// WHILE Statement

//-------------------------------------------------

WhileStmt -> DoStmt WhileStmt\_1;

WhileStmt\_1 -> 'WHILE' RelOperation DoStmt WhileStmt\_1

-> .

;

//------------------------------------------------

// DO Statement

//------------------------------------------------

DoStmt -> EXITstmt DoStmt\_1;

DoStmt\_1 -> 'DO' DoEnd EXITstmt DoStmt\_1

-> .

;

DoEnd -> Statements DoEnd\_1;

DoEnd\_1 -> 'OD'

-> .

;

//---------------------------------------------------

// EXIT Statement

//---------------------------------------------------

EXITstmt -> RetStmt EXITstmt\_1;

EXITstmt\_1 -> 'EXIT' RetStmt EXITstmt\_1

-> .

;

//-------------------------------------------------

// RETURN Statement

//-------------------------------------------------

RetStmt -> InlineAssembly RetStmt\_1;

RetStmt\_1 -> 'RETURN' OptReturnValue InlineAssembly RetStmt\_1

-> .

;

OptReturnValue -> '(' ArithExpr ')'

-> .

;

//---------------------------------------------

// Inline Assembly Code

//---------------------------------------------

InlineAssembly -> CodeBlock InlineAssembly\_1;

InlineAssembly\_1 -> 'ASM' InlineAssBlock CodeBlock InlineAssembly\_1

-> .

;

InlineAssBlock -> EndAsmBlock InlineAssBlock\_1;

InlineAssBlock\_1 -> '{' EndAsmBlock InlineAssBlock\_1

-> .

;

EndAsmBlock -> AsmStmt EndAsmBlock\_1;

EndAsmBlock\_1 -> '}'

-> .

;

//-----------------------------------------------

// CODE (Kluge) BLOCK statement

//-----------------------------------------------

CodeBlock -> UntillStmt CodeBlock\_1;

CodeBlock\_1 -> '[' CodeBlockEnd UntillStmt CodeBlock\_1

-> .

;

CodeBlockEnd -> ConstList CodeBlockEnd\_1;

CodeBlockEnd\_1 -> ']'

-> .

;

//---------------------------------------------------

// UNTILE Statement

//---------------------------------------------------

UntillStmt -> Push UntillStmt\_1;

UntillStmt\_1 -> 'UNTILL' RelOperation Push UntillStmt\_1

-> .

;

//---------------------------------------------------

// PUSH Statement

//---------------------------------------------------

Push -> Pop Push\_1;

Push\_1 -> 'PUSH' PushSourceList Pop Push\_1

-> .

;

PushSourceList -> PushSource PushSourceList\_1;

PushSourceList\_1-> ',' PushSource PushSourceList\_1

-> .

;

PushSource -> ArithExpr PushSource\_1;

PushSource\_1-> 'AREG'

-> 'XREG'

-> 'YREG'

-> 'SR'

-> .

;

//---------------------------------------------------

// POP Statment

//---------------------------------------------------

Pop -> Break Pop\_1;

Pop\_1 -> 'POP' PopDestList Break Pop\_1

-> .

;

PopDestList -> PopDest PopDestList\_1;

PopDestList\_1 -> ',' PopDest PopDestList\_1

-> .

;

PopDest -> MemContentsType PopDest\_1;

PopDest\_1 -> 'AREG'

-> 'XREG'

-> 'YREG'

-> 'SR'

-> .

;

//--------------------------------------------------

// BREAK Statement (Assembler BRK) Software Interrupt

//--------------------------------------------------

Break -> Rti Break\_1;

Break\_1 -> 'BREAK' Rti Break\_1

-> .

;

//--------------------------------------------------

// RTI Return from interrupt statement

//--------------------------------------------------

Rti -> Assignment Rti\_1;

Rti\_1 -> 'RTI' Assignment Rti\_1

-> .

;

//---------------------------------------------------

// Assignment Statements

//---------------------------------------------------

Assignment -> MemContentsType Assignment\_1;

Assignment\_1-> '=' ArithExpr Assignment\_1

-> '==+' ArithExpr Assignment\_1

-> '==-' ArithExpr Assignment\_1

-> '==\*' ArithExpr Assignment\_1

-> '==/' ArithExpr Assignment\_1

-> '==MOD' ArithExpr Assignment\_1

-> '==&' ArithExpr Assignment\_1 // AND

-> '==!' ArithExpr Assignment\_1 //XOR

-> '==%' ArithExpr Assignment\_1 //OR

-> '==LSH' ArithExpr Assignment\_1

-> '==RSH' ArithExpr Assignment\_1

-> .

;

//----------------------------------------------

// Relation Operations

//----------------------------------------------

RelOperation -> LogicalOR RelOperation\_1;

RelOperation\_1 -> '>' LogicalOR RelOperation\_1

-> '<' LogicalOR RelOperation\_1

-> 'GTEQ' LogicalOR RelOperation\_1

-> 'LTEQ' LogicalOR RelOperation\_1

-> '=' LogicalOR RelOperation\_1

-> '#' LogicalOR RelOperation\_1 //not equal

-> .

;

LogicalOR -> LogicalAND LogicalOR\_1;

LogicalOR\_1 -> 'OR' LogicalAND LogicalOR\_1

-> .

;

LogicalAND -> ArithExpr LogicalAND\_1;

LogicalAND\_1 -> 'AND' ArithExpr LogicalAND\_1

-> .

;

//--------------------------------------

// Arithmetic Expressions

//--------------------------------------

ArithExpr -> BitwiseAND BitwiseOR\_1;

BitwiseOR\_1 -> '%' BitwiseAND BitwiseOR\_1

-> .

;

BitwiseAND -> BitwiseXOR BitwiseAND\_1;

BitwiseAND\_1 -> '&' BitwiseXOR BitwiseAND\_1

-> .

;

BitwiseXOR -> AddExpr BitwiseXOR\_1;

BitwiseXOR\_1 -> '!' AddExpr BitwiseXOR\_1

-> 'XOR' AddExpr BitwiseXOR\_1

-> .

;

AddExpr -> ShifExpr AddExpr\_1;

AddExpr\_1 -> '+' ShifExpr AddExpr\_1

-> '-' ShifExpr AddExpr\_1

-> .

;

ShifExpr -> MultExpr ShiftExpr\_1;

ShiftExpr\_1 -> 'LSH' MultExpr ShiftExpr\_1

-> 'RSH' MultExpr ShiftExpr\_1

-> .

;

MultExpr -> Unary MultExpr\_1;

MultExpr\_1 -> '\*' Unary MultExpr\_1

-> '/' Unary MultExpr\_1

-> 'MOD' Unary MultExpr\_1

-> .

;

Unary -> '-' Unary

-> Factor

;

Factor -> Value Factor\_1;

Factor\_1 -> '(' ArithExpr ')'

-> .

;

//-------------------------------------------

// Declarations

//-------------------------------------------

// DEFINE declaration Sort of a MACRO defination

//-------------------------------------------

SysDecl -> TypeDefDecl Define;

Define -> 'DEFINE' DefList TypeDefDecl Define

-> .

;

DefList -> Def DefList\_1;

DefList\_1 -> ',' Def DefList\_1

-> .

;

Def -> IDENT '=' CompConst;

//----------------------------------------------

// TYPE Defination Creates User define data

// object.

//----------------------------------------------

TypeDefDecl -> FundDecl TypeDefDecl\_1;

TypeDefDecl\_1 -> 'TYPE' EndTypeDef FundDecl TypeDefDecl\_1

-> .

;

EndTypeDef -> RecDefField EndTypeDef\_1;

EndTypeDef\_1 -> ']'

-> .

;

RecDefField -> Ident RecDefField\_1;

RecDefField\_1 -> '=' '[' LocalDecls

-> '[' LocalDecls

-> .

;

//--------------------------------------------

// Fundemental Declarations. These are the

// base types that every thing is based on

//--------------------------------------------

FundDecl ->FundPointerDecl FundDecl\_1;

FundDecl\_1 -> 'BOOL' FundPointerDecl FundDecl\_1

-> 'CHAR' FundPointerDecl FundDecl\_1

-> 'BYTE' FundPointerDecl FundDecl\_1

-> 'INT' FundPointerDecl FundDecl\_1

-> 'CARD' FundPointerDecl FundDecl\_1

-> 'RECORDTYPE' FundPointerDecl FundDecl\_1

-> .

;

FundPointerDecl -> FundArrayDecl FundPointerDecl\_1;

FundPointerDecl\_1 -> 'POINTER' FundArrayDecl

-> .

;

FundArrayDecl -> FunctionDecl FundArrayDecl\_1;

FundArrayDecl\_1 -> 'ARRAY' FunctionDecl

-> .

;

FunctionDecl -> IdentList FunctionDecl\_1;;

FunctionDecl\_1 -> 'FUNC' FuncDef

-> 'PROC' ProcDef

-> 'INTERRUPT' IrqDef

-> .

;

IrqDef -> IrqDecl IrqBody;

ProcDef -> ProcDecl ProcBody;

FuncDef -> FuncDecl FuncBody;

IrqDecl -> 'IDENT' OptInit '(' ')';

ProcDecl -> 'IDENT' OptInit '(' ParamList ')';

FuncDecl -> 'IDENT' OptInit '(' ParamList ')';

OptInit -> '=' CompConst

-> .

;

IrqBody -> LocalDecls Statements;

ProcBody -> LocalDecls Statements;

FuncBody -> LocalDecls Statements;

//----------------------------------

// Identifiers

//BYTE a,b;

//BYTE x=$a0

//BYTE y=[15]

//BYTE ARRAY a=$2000

//BYTE ARRAY b =["This is a string"]

//BYTE ARRAY c = [1 2 3 4 55 6]

//BYTE ARRAY d(156) =$2100

//----------------------------------

IdentList -> Ident IdentList\_1;

IdentList\_1 -> ',' Ident IdentList\_1

-> .

;

Ident -> IdentInit Ident\_1;

Ident\_1 -> 'IDENT' IdentInit

-> .

;

IdentInit -> ArrayDim IdentInit\_1;

IdentInit\_1 -> '=' Address

-> .

;

Address -> CompConst Address\_1;

Address\_1 -> '[' Data ']'

-> .

;

Data -> ConstList Data\_1;

Data\_1 -> 'STRING'

-> .

;

ArrayDim -> ArrayDim\_1;

ArrayDim\_1 -> '(' CompConst ')'

-> .

;

//-------------------------------------------

// Parameter Declarations

//-------------------------------------------

ParamList -> PramPointer Param\_1;

Param\_1 -> 'RECORDTYPE' PramPointer Param\_1

-> 'CHAR' PramPointer Param\_1

-> 'BYTE' PramPointer Param\_1

-> 'INT' PramPointer Param\_1

-> 'CARD' PramPointer Param\_1

-> 'BOOL' PramPointer Param\_1

-> ',' Param\_1

-> .

;

PramPointer -> ParamArray ParamPointer\_1;

ParamPointer\_1 -> 'POINTER' ParamArray

-> .

;

ParamArray -> IdentList ParamArray\_1;

ParamArray\_1 -> 'ARRAY' IdentList

-> .

;

;

//-----------------------------------------------

// Local Variableas

//-----------------------------------------------

LocalDecls -> LocalPointerDecl LocalDecls\_1;

LocalDecls\_1 -> 'CHAR' LocalPointerDecl LocalDecls\_1

-> 'BYTE' LocalPointerDecl LocalDecls\_1

-> 'INT' LocalPointerDecl LocalDecls\_1

-> 'CARD' LocalPointerDecl LocalDecls\_1

-> 'BOOL' LocalPointerDecl LocalDecls\_1

-> 'RECORDTYPE' LocalPointerDecl LocalDecls\_1

-> .

;

LocalPointerDecl -> LocalArrayDecl LocalPointerDecl\_1;

LocalPointerDecl\_1 -> 'POINTER' LocalArrayDecl

-> .

;

LocalArrayDecl -> IdentList LocalArrayDecl\_1;

LocalArrayDecl\_1 -> 'ARRAY' IdentList

-> .

;

//-------------------------------

// Compiler Constants

//-------------------------------

ConstList -> CompConst ConstList\_1;

ConstList\_1 -> ',' CompConst ConstList\_1

-> .

;

CompConst -> BaseCompConst CompConst\_1;

CompConst\_1 -> '+' BaseCompConst CompConst\_1

-> '-' BaseCompConst CompConst\_1

-> .

;

BaseCompConst -> 'NUMBER'

-> '\*'

-> '@' MemContentsType

-> 'INTERUPT\_IDENT'

-> 'FUNC\_IDENT'

-> 'PROC\_IDENT'

-> .

;

//----------------------------------

//Variable References

//Memory References

//----------------------------------

ValueList -> Value ValueList\_1;

ValueList\_1 -> ',' Value ValueList\_1

-> .

;

Value -> MemContentsType Value\_1;

Value\_1 -> 'NUMBER'

-> '\*' // current memroy location

-> '@' AddressOf

-> .

;

AddressOf -> MemContentsType AddressOf\_1;

AddressOf\_1 ->'PROC\_IDENT'

-> 'FUNC\_IDENT'

-> 'INTERRUPT\_IDENT'

-> .

;

MemContentsType -> MemContents MemContentsType\_1;

MemContentsType\_1 -> '(' ArithExpr ')' //array ref

-> '^' // Pointer Dereference

-> '.' MemContents MemContentsType\_1 //record ref

-> .

;

MemContents -> 'IDENT'

-> .

;

//----------------------------------------

// Inline assembly code

//----------------------------------------

//----------------------------------------

// Statements

//----------------------------------------

AsmStmt -> DefineMemory Processor\_1;

Processor\_1 -> 'PROCESSOR' ProcessorType DefineMemory Processor\_1

-> .

;

ProcessorType -> 'R6502'

-> 'WD65C02'

-> 'WD65C816'

;

//-----------------------------------------

// Code

// These statements are what actually

// do something that concerns the machine

// code that is executed

//-----------------------------------------

//-----------------------------------------

// SECITON

//-----------------------------------------

Section -> Org Section\_1;

Section\_1 -> 'SECTION' SectionName Org Section\_1

-> .

;

SectionName -> SectionDef SectionName\_1;

SectionName\_1 -> 'IDENT' SectionDef

-> .

;

SectionDef -> '[' SectionAttributesList ']'

-> .

;

SectionAttributesList -> SectionAtribute SectionAttributesList\_1;

SectionAttributesList\_1 -> ',' SectionAtribute SectionAttributesList\_1

-> .

;

SectionAtribute -> 'START' '=' AsmConstant

-> 'SIZE' '=' AsmConstant

-> 'MODE' '=' Modes

-> 'ZEROPAGE' '=' TrueFalse

-> .

;

Modes -> 'READ\_ONLY'

-> 'READ\_WRITE'

;

TrueFalse -> 'true'

-> 'false'

;

//-------------------------------------

// Org Sets the location counter

// for the current section

//-------------------------------------

Org -> DefineMemory Org\_1;

Org\_1 -> 'ORG' AsmConstant DefineMemory Org\_1

-> .

;

//-------------------------------------

// DefineMemeory

//-------------------------------------

DefineMemory -> DefineStorage DefineMemory\_1;

DefineMemory\_1 -> 'DB' AsmConstList DefineStorage DefineMemory\_1

-> 'DW' AsmConstList DefineStorage DefineMemory\_1

-> 'DL' AsmConstList DefineStorage DefineMemory\_1

-> 'DAS' 'STRING' DefineStorage DefineMemory\_1

-> 'DCS' 'STRING' DefineStorage DefineMemory\_1

-> .

;

//-------------------------------------

// DefineStorage

//-------------------------------------

DefineStorage -> Proceedure DefineStorage\_1;

DefineStorage\_1 -> 'DS' AsmConstant Proceedure DefineStorage\_1

-> .

;

//-------------------------------------

// Proceedure

//-------------------------------------

Proceedure -> Instruction Proceedure\_1;

Proceedure\_1 -> 'ASMPROC' Section 'EPROC' Instruction Proceedure\_1

-> .

;

//--------------------------------------

// Opcodes

//--------------------------------------

Instruction -> Labels Instruction\_1;

Instruction\_1 -> 'ADC' AluAdrModes Labels Instruction\_1 //ALU Opcodes

-> 'AND' AluAdrModes Labels Instruction\_1

-> 'EOR' AluAdrModes Labels Instruction\_1

-> 'ORA' AluAdrModes Labels Instruction\_1

-> 'CMP' AluAdrModes Labels Instruction\_1

-> 'SBC' AluAdrModes Labels Instruction\_1

-> 'LDA' AluAdrModes Labels Instruction\_1

-> 'STA' StaAddressingModes Labels Instruction\_1

-> 'ASL' ShiftAddressingModes Labels Instruction\_1 //shift addressing modes

-> 'ROL' ShiftAddressingModes Labels Instruction\_1

-> 'LSR' ShiftAddressingModes Labels Instruction\_1

-> 'ROR' ShiftAddressingModes Labels Instruction\_1

-> 'BCC' RelAddressingMode Labels Instruction\_1 // Branch Op Codes

-> 'BCS' RelAddressingMode Labels Instruction\_1

-> 'BEQ' RelAddressingMode Labels Instruction\_1

-> 'BMI' RelAddressingMode Labels Instruction\_1

-> 'BNE' RelAddressingMode Labels Instruction\_1

-> 'BPL' RelAddressingMode Labels Instruction\_1

-> 'BVC' RelAddressingMode Labels Instruction\_1

-> 'BVS' RelAddressingMode Labels Instruction\_1

-> 'BIT' BitAddressModes Labels Instruction\_1 //BIT opcode

-> 'BRK' Labels Instruction\_1 //Implied Addressing Mode

-> 'CLC' Labels Instruction\_1

-> 'CLD' Labels Instruction\_1

-> 'CLI' Labels Instruction\_1

-> 'CLV' Labels Instruction\_1

-> 'DEX' Labels Instruction\_1

-> 'DEY' Labels Instruction\_1

-> 'INX' Labels Instruction\_1

-> 'INY' Labels Instruction\_1

-> 'NOP' Labels Instruction\_1

-> 'PHA' Labels Instruction\_1

-> 'PLA' Labels Instruction\_1

-> 'PHP' Labels Instruction\_1

-> 'PLP' Labels Instruction\_1

-> 'RTI' Labels Instruction\_1

-> 'RTS' Labels Instruction\_1

-> 'SEC' Labels Instruction\_1

-> 'SED' Labels Instruction\_1

-> 'SEI' Labels Instruction\_1

-> 'TAX' Labels Instruction\_1

-> 'TAY' Labels Instruction\_1

-> 'TXA' Labels Instruction\_1

-> 'TYA' Labels Instruction\_1

-> 'TXS' Labels Instruction\_1

-> 'TSX' Labels Instruction\_1

-> 'INC' IncAddressingMOdes Labels Instruction\_1 //Inc/Dec Addressing Modes

-> 'DEC' IncAddressingMOdes Labels Instruction\_1

-> 'JMP' JumpAddressingModes Labels Instruction\_1 // Jump

-> 'JSR' CallAddressingMode Labels Instruction\_1 // Jump To Subroutine

-> 'CPX' CPX\_CPY\_AddressingMode Labels Instruction\_1

-> 'CPY' CPX\_CPY\_AddressingMode Labels Instruction\_1

-> 'LDX' LdxAddressingMode Labels Instruction\_1

-> 'LDY' LdyAddressingMode Labels Instruction\_1

-> 'STX' StxAddressingMode Labels Instruction\_1

-> 'STY' StyAddressingMode Labels Instruction\_1

-> .

;

//---------------------------------------------

// Lables

//---------------------------------------------

Labels -> 'IDENT' Label\_1 //Global

-> .

;

Label\_1 -> ':' //private lable

-> .

;

//-----------------------------------

//ALU Addressing Mode

//-----------------------------------

AluAdrModes -> '#' AsmConstant

-> '(' Indirect

-> AsmConstant OptIndexReg

;

Indirect -> AsmConstant Indirect\_1;

Indirect\_1 -> ')' ',' 'YREG'

-> ',' 'XREG' ')'

;

//---------------------------------------------

// STA addressing mode

//---------------------------------------------

StaAddressingModes -> AsmConstant OptIndexReg

-> '(' Indirect

;

//-----------------------------------------

// ASL LSR ROR and ROL addressing modes

//-----------------------------------------

ShiftAddressingModes -> 'AREG'

-> AsmConstant OptXReg

;

//------------------------------------------

// Branch Instructions Addressing Mode

//------------------------------------------

RelAddressingMode -> AsmConstant;

//------------------------------------------

// BIT Instructions Addressing Mode

//------------------------------------------

BitAddressModes -> AsmConstant;

//------------------------------------------

// INC DEC Instructions Addressing Mode

//------------------------------------------

IncAddressingMOdes -> AsmConstant OptXReg;

//------------------------------------------

// JMP Instructions Addressing Mode

//------------------------------------------

JumpAddressingModes -> JumpDirect JumpIndirect;

JumpIndirect -> '(' AsmConstant ')'

-> .

;

JumpDirect -> AsmConstant;

//------------------------------------------

// JSR Instructions Addressing Mode

//------------------------------------------

CallAddressingMode -> AsmConstant;

//------------------------------------------

// LDX Instructions Addressing Mode

//------------------------------------------

LdxAddressingMode -> '#' AsmConstant

-> AsmConstant OptYReg

;

//------------------------------------------

// CPX & CPY Instructions Addressing Mode

//------------------------------------------

CPX\_CPY\_AddressingMode -> '#' AsmConstant

-> AsmConstant

;

//------------------------------------------

// STX Instructions Addressing Mode

//------------------------------------------

StxAddressingMode -> AsmConstant OptYReg;

//------------------------------------------

// LDY Instructions Addressing Mode

//------------------------------------------

LdyAddressingMode -> '#' AsmConstant

-> AsmConstant OptXReg

;

//------------------------------------------

// STY Instructions Addressing Mode

//------------------------------------------

StyAddressingMode -> AsmConstant OptXReg;

//---------------------------------------------

// Optional Index Registers

//---------------------------------------------

OptIndexReg -> ',' OptIndexReg\_1

-> .

;

OptIndexReg\_1 -> 'XREG'

-> 'YREG'

;

OptXReg -> ',' 'XREG'

-> .

;

OptYReg -> ',' 'YREG'

-> .

;

//---------------------------------------

// Assembler Constants

//---------------------------------------

AsmConstList -> 'STRING'

-> AsmConstList\_1

;

AsmConstList\_1 -> AsmConstant AsmConstList\_2;

AsmConstList\_2 -> ',' AsmConstant AsmConstList\_2

-> .

;

AsmConstant -> AsmConstAddSub AsmConstant\_1;

AsmConstant\_1 -> '>' AsmConstAddSub //Lower Part

-> '<' AsmConstAddSub //Upper Part

-> .

;

AsmConstAddSub -> BaseAsmConstant AsmConstAddSub\_1;

AsmConstAddSub\_1 -> '+' BaseAsmConstant AsmConstAddSub\_1

-> '-' BaseAsmConstant AsmConstAddSub\_1

-> .

;

BaseAsmConstant -> 'LOCAL\_LABLE'

-> 'GLOBAL\_LABLE'

-> 'IDENT'

-> 'NUMBER'

-> 'CHAR\_CONSTANT'

-> '\*' //current location counter

;

---------- AccessModeStackItem.h -------------

#pragma once

class CAccessModeStackItem : public CStackItem

{

CSection::Mode m\_AccessMode;

public:

CAccessModeStackItem();

virtual ~CAccessModeStackItem();

bool Create();

CSection::Mode GetAccessMode() { return m\_AccessMode; }

void SetAccessMode(CSection::Mode AM) { m\_AccessMode = AM; }

};

---------- Act65ACC.h -------------

#pragma once

class CAct65ACC : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = ".A";

public:

CAct65ACC();

virtual ~CAct65ACC();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65ADD.h -------------

#pragma once

class CAct65ADD : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "ADD";

public:

CAct65ADD();

virtual ~CAct65ADD();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65ARRAY.h -------------

#pragma once

class CAct65ARRAY : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "ARRAY";

public:

CAct65ARRAY();

virtual ~CAct65ARRAY();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65ASM.h -------------

#pragma once

class CAct65ASM : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "ASM";

public:

CAct65ASM();

virtual ~CAct65ASM();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65ASMPROC.h -------------

#pragma once

class CAct65ASMPROC : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "ASMPROC";

public:

CAct65ASMPROC();

virtual ~CAct65ASMPROC();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65ASMstatement.h -------------

#pragma once

class CAct65ASMstatement : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "ASMstmt";

public:

CAct65ASMstatement();

virtual ~CAct65ASMstatement();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65AssignADD.h -------------

#pragma once

class CAct65AssignADD : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "=ADD";

public:

CAct65AssignADD();

virtual ~CAct65AssignADD();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65AssignAND.h -------------

#pragma once

class CAct65AssignAND : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "=AND";

public:

CAct65AssignAND();

virtual ~CAct65AssignAND();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65AssignDIV.h -------------

#pragma once

class CAct65AssignDIV : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "=DIV";

public:

CAct65AssignDIV();

virtual ~CAct65AssignDIV();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65AssignLSH.h -------------

#pragma once

class CAct65AssignLSh : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "=LSH";

public:

CAct65AssignLSh();

virtual ~CAct65AssignLSh();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65Assignment.h -------------

#pragma once

class CAct65Assignment : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "=";

public:

CAct65Assignment();

virtual ~CAct65Assignment();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65AssignMOD.h -------------

#pragma once

class CAct65AssignMOD : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "=MOD";

public:

CAct65AssignMOD();

virtual ~CAct65AssignMOD();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65AssignMULT.h -------------

#pragma once

class CAct65AssignMULT :public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "=MUL";

public:

CAct65AssignMULT();

virtual ~CAct65AssignMULT();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65AssignOR.h -------------

#pragma once

class CAct65AssignOR : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "=OR";

public:

CAct65AssignOR();

virtual ~CAct65AssignOR();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65AssignRSH.h -------------

#pragma once

class CAct65AssignRSH : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "=RSH";

public:

CAct65AssignRSH();

virtual ~CAct65AssignRSH();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65AssignSUB.h -------------

#pragma once

class CAct65AssignSUB : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "=SUB";

public:

CAct65AssignSUB();

virtual ~CAct65AssignSUB();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65AssignXOR.h -------------

#pragma once

class CAct65AssignXOR : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "=XOR";

public:

CAct65AssignXOR();

virtual ~CAct65AssignXOR();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65BEGIN.h -------------

#pragma once

class CAct65BEGIN : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "BEGIN";

public:

CAct65BEGIN();

virtual ~CAct65BEGIN();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65BIT.h -------------

#pragma once

class CAct65BIT : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "BIT";

public:

CAct65BIT();

virtual ~CAct65BIT();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65BitWiseAND.h -------------

#pragma once

class CAct65BitWiseAND : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "BW-AND";

public:

CAct65BitWiseAND();

virtual ~CAct65BitWiseAND();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65BitWiseOR.h -------------

#pragma once

class CAct65BitWiseOR : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "BW-OR";

public:

CAct65BitWiseOR();

virtual ~CAct65BitWiseOR();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65BOOL.h -------------

#pragma once

class CAct65BOOL : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "BOOL";

public:

CAct65BOOL();

virtual ~CAct65BOOL();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65BREAK.h -------------

#pragma once

class CActBREAK : public CAstNode

{

public:

CActBREAK();

virtual ~CActBREAK();

bool Create();

virtual CValue\* Process();

};

---------- Act65BYTE.h -------------

#pragma once

class CAct65BYTE : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "BYTE";

public:

CAct65BYTE();

virtual ~CAct65BYTE();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65CARD.h -------------

#pragma once

class CAct65CARD : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "CARD";

public:

CAct65CARD();

virtual ~CAct65CARD();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65CHAR.h -------------

#pragma once

class CAct65CHAR : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "CHAR";

public:

CAct65CHAR();

virtual ~CAct65CHAR();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65CharCibstabt.h -------------

#pragma once

class CAct65CharConstant : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "CHAR-CONST";

public:

CAct65CharConstant();

virtual ~CAct65CharConstant();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65CodeBlock.h -------------

#pragma once

class CAct65CodeBlock : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "CODE-BLOCK";

public:

CAct65CodeBlock();

virtual ~CAct65CodeBlock();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65CurrentLocation.h -------------

#pragma once

#include "AstNode.h"

class CAct65CurrentLocation : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "\*";

public:

CAct65CurrentLocation();

virtual ~CAct65CurrentLocation();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65DAS.h -------------

#pragma once

class CAct65DAS : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "DAS";

public:

CAct65DAS();

virtual ~CAct65DAS();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65DB.h -------------

#pragma once

class CAct65DB : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "DB";

public:

CAct65DB();

virtual ~CAct65DB();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65DCS.h -------------

#pragma once

class CAct65DCS : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "DCS";

public:

CAct65DCS();

virtual ~CAct65DCS();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65DEFINE.h -------------

#pragma once

class CAct65DEFINE : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "DEFINE";

public:

CAct65DEFINE();

virtual ~CAct65DEFINE();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65DL.h -------------

#pragma once

class CAct65DL : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "DL";

public:

CAct65DL();

virtual ~CAct65DL();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65DO.h -------------

#pragma once

class CAct65DO : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "DO";

public:

CAct65DO();

virtual ~CAct65DO();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65DS.h -------------

#pragma once

class CAct65DS : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "DS";

public:

CAct65DS();

virtual ~CAct65DS();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65DW.h -------------

#pragma once

class CAct65DW : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "DW";

public:

CAct65DW();

virtual ~CAct65DW();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65ELSE.h -------------

#pragma once

class CAct65ELSE : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "ELSE";

public:

CAct65ELSE();

virtual ~CAct65ELSE();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65ELSEIF.h -------------

#pragma once

class CAct65ELSEIF : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "ELSEIF";

public:

CAct65ELSEIF();

virtual ~CAct65ELSEIF();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65END.h -------------

#pragma once

class CAct65END : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "END";

public:

CAct65END();

virtual ~CAct65END();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65EPROC.h -------------

#pragma once

class CAct65EPROC : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "EPROC";

public:

CAct65EPROC();

virtual ~CAct65EPROC();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65EqualTO.h -------------

#pragma once

class CAct65EqualTO : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "EQU-TO";

public:

CAct65EqualTO();

virtual ~CAct65EqualTO();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65EXIT.h -------------

#pragma once

class CAct65EXIT : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "EXIT";

public:

CAct65EXIT();

virtual ~CAct65EXIT();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65FFI.h -------------

#pragma once

class CAct65FFI : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "FFI";

public:

CAct65FFI();

virtual ~CAct65FFI();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65FI.h -------------

#pragma once

class CAct65FI : public CAstNode

{

public:

CAct65FI();

virtual ~CAct65FI();

bool Create();

virtual CValue\* Process();

};

---------- Act65FlagCARRY.h -------------

#pragma once

class CAct65FlagCARRY : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = ".C";

public:

CAct65FlagCARRY();

virtual ~CAct65FlagCARRY();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65FlagNEG.h -------------

#pragma once

class CAct65FlagNEG : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = ".N";

public:

CAct65FlagNEG();

virtual ~CAct65FlagNEG();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65FlagOVERFLOW.h -------------

#pragma once

class CAct65FlagOVERFLOW : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = ".O";

public:

CAct65FlagOVERFLOW();

virtual ~CAct65FlagOVERFLOW();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65FlagZERO.h -------------

#pragma once

class CAct65FlagZERO : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = ".Z";

public:

CAct65FlagZERO();

virtual ~CAct65FlagZERO();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65FOR.h -------------

#pragma once

class CAct65FOR : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "FOR";

public:

CAct65FOR();

virtual ~CAct65FOR();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65ForItterator.h -------------

#pragma once

class CAct65ForItterator : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "ITRATR";

public:

CAct65ForItterator();

virtual ~CAct65ForItterator();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65ForSTEP.h -------------

#pragma once

class CAct65ForSTEP : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "STEP";

public:

CAct65ForSTEP();

virtual ~CAct65ForSTEP();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65ForTO.h -------------

#pragma once

class CAct65ForTO : public CAstNode

{

public:

CAct65ForTO();

virtual ~CAct65ForTO();

bool Create();

virtual CValue\* Process();

};

---------- Act65FuncCall.h -------------

#pragma once

class CAct65FuncCall : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "FUNC-CALL";

public:

CAct65FuncCall();

virtual ~CAct65FuncCall();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65GreaterTHAN.h -------------

#pragma once

class CAct65GreaterTHAN : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "THAN";

public:

CAct65GreaterTHAN();

virtual ~CAct65GreaterTHAN();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65GTEQ.h -------------

#pragma once

class CAct65GTEQ : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "GTEQ";

public:

CAct65GTEQ();

virtual ~CAct65GTEQ();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65IDENT.h -------------

#pragma once

class CAct65IDENT : public CAstNode

{

public:

CAct65IDENT();

virtual ~CAct65IDENT();

bool Create(CBin\* pSym);

virtual CValue\* Process();

};

---------- Act65IF.h -------------

#pragma once

class CAct65IF : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "IF";

public:

CAct65IF();

virtual ~CAct65IF();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65IFF.h -------------

#pragma once

class CAct65IFF : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "IFF";

public:

CAct65IFF();

virtual ~CAct65IFF();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65INT.h -------------

#pragma once

class CAct65INT : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "INT";

public:

CAct65INT();

virtual ~CAct65INT();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65INTERRUPT.h -------------

#pragma once

class CAct65INTERRUPT : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "IRQ";

public:

CAct65INTERRUPT();

virtual ~CAct65INTERRUPT();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65Label.h -------------

#pragma once

class CAct65Label : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "LABEL";

public:

CAct65Label();

virtual ~CAct65Label();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65LessTHAN.h -------------

#pragma once

class CAct65LessTHAN : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "LT";

public:

CAct65LessTHAN();

virtual ~CAct65LessTHAN();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65LogicalAND.h -------------

#pragma once

class CAct65LogicalAND : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "L-AND";

public:

CAct65LogicalAND();

virtual ~CAct65LogicalAND();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65LogicalOR.h -------------

#pragma once

class CAct65LogicalOR : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "L-OR";

public:

CAct65LogicalOR();

virtual ~CAct65LogicalOR();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65LowerPart.h -------------

#pragma once

class CAct65LowerPart : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "LowPart";

public:

CAct65LowerPart();

virtual ~CAct65LowerPart();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65LSH.h -------------

#pragma once

class CAct65LSH : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "LSH";

public:

CAct65LSH();

virtual ~CAct65LSH();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65LTEQ.h -------------

#pragma once

class CAct65LTEQ : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "LTEQ";

public:

CAct65LTEQ();

virtual ~CAct65LTEQ();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65Module.h -------------

#pragma once

class CAct65Module : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "MODULE";

public:

CAct65Module();

virtual ~CAct65Module();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65NotEquelTO.h -------------

#pragma once

class CAct65NotEquelTO : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "NEQ";

public:

CAct65NotEquelTO();

virtual ~CAct65NotEquelTO();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65OD.h -------------

#pragma once

class CAct65OD : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "OD";

public:

CAct65OD();

virtual ~CAct65OD();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65Opcode.h -------------

#pragma once

class CAct65Opcode : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "OP";

Token m\_OpcodeToken;

const char\* m\_pOpCodeName;

int m\_LineNumber;

int m\_ColumnNumber;

CSymbol\* m\_pSym; // Symbol for argument

CSymbol\* m\_pLabel; // Symbol for instruction location

AdrModeType m\_AdressMode;

CInstruction\* m\_pInstruction;

public:

CAct65Opcode();

virtual ~CAct65Opcode();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

//-----------------------------

// Node Processor

//-----------------------------

virtual CValue\* Process();

//-----------------------------

// Setter/Getter Methods

//-----------------------------

Token GetToken() { return m\_OpcodeToken; }

void SetToken(Token t) { m\_OpcodeToken = t; }

const char\* GetOpcodeName() { return m\_pOpCodeName; }

void SetOpcodeName(const char\* pN) { m\_pOpCodeName = pN; }

int GetLineNumber() { return m\_LineNumber; }

void SetLineNumber(int n) { m\_LineNumber = n; }

int GetColumnNumber() { return m\_ColumnNumber; }

void SetColumnNumber(int c) { m\_ColumnNumber = c; }

};

---------- Act65OptNOT.h -------------

#pragma once

class CAct65OptNOT : public CAstNode

{

public:

CAct65OptNOT();

virtual ~CAct65OptNOT();

virtual bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65ORG.h -------------

#pragma once

class CAct65ORG : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "ORG";

public:

CAct65ORG();

virtual ~CAct65ORG();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65POINTER.h -------------

#pragma once

class CAct65POINTER : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "POINTER";

public:

CAct65POINTER();

virtual ~CAct65POINTER();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65POP.h -------------

#pragma once

class CAct65POP : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "POP";

public:

CAct65POP();

virtual ~CAct65POP();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65PopDest.h -------------

#pragma once

class CAct65PopDest : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "POP-DST";

public:

CAct65PopDest();

virtual ~CAct65PopDest();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65Private.h -------------

#pragma once

class CAct65Private : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "PRIVATE";

public:

CAct65Private();

virtual ~CAct65Private();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65PROC.h -------------

#pragma once

class CAct65PROC : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "PROC";

public:

CAct65PROC();

virtual ~CAct65PROC();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65ProcCall.h -------------

#pragma once

class CAct65ProcCall : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "PROC-CALL";

public:

CAct65ProcCall();

virtual ~CAct65ProcCall();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65PROCESSOR.h -------------

#pragma once

class CAct65PROCESSOR : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "PROCESSOR";

public:

CAct65PROCESSOR();

virtual ~CAct65PROCESSOR();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65PROCname.h -------------

#pragma once

class CAct65PROCname : public CAstNode

{

public:

CAct65PROCname();

virtual ~CAct65PROCname();

bool Create();

virtual CValue\* Process();

};

---------- Act65ProcParams.h -------------

#pragma once

class CAct65ProcParams : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "PROC-PARAMS";

public:

CAct65ProcParams();

virtual ~CAct65ProcParams();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65PUSH.h -------------

#pragma once

class CAct65PUSH : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "PUSH";

public:

CAct65PUSH();

virtual ~CAct65PUSH();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65PushSource.h -------------

#pragma once

class CAct65PushSource : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "PUSH-SRC";

public:

CAct65PushSource();

virtual ~CAct65PushSource();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65RECTYPE.h -------------

#pragma once

class CAct65RECTYPE : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "RECTYPE";

public:

CAct65RECTYPE();

virtual ~CAct65RECTYPE();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65RETURN.h -------------

#pragma once

class CAct65RETURN : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "RETURN";

public:

CAct65RETURN();

virtual ~CAct65RETURN();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65ROL.h -------------

#pragma once

class CAct65ROL : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "ROL";

public:

CAct65ROL();

virtual ~CAct65ROL();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65ROR.h -------------

#pragma once

class CAct65ROR : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "ROR";

public:

CAct65ROR();

virtual ~CAct65ROR();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65RSH.h -------------

#pragma once

class CAct65RSH : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "RSH";

public:

CAct65RSH();

virtual ~CAct65RSH();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65RTI.h -------------

#pragma once

class CAct65RTI : public CAstNode

{

public:

CAct65RTI();

virtual ~CAct65RTI();

bool Create();

virtual CValue\* Process();

};

---------- Act65SecAtrbFALSE.h -------------

#pragma once

class CAct65SecAtrbFALSE : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "FALSE";

public:

CAct65SecAtrbFALSE();

virtual ~CAct65SecAtrbFALSE();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65SecAtrbMODE.h -------------

#pragma once

class CAct65SecAtrbMODE : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "MODE";

public:

CAct65SecAtrbMODE();

virtual ~CAct65SecAtrbMODE();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65SecAtrbREADONLY.h -------------

#pragma once

class CAct65SecAtrbREADONLY : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "RD-ONLY";

public:

CAct65SecAtrbREADONLY();

virtual ~CAct65SecAtrbREADONLY();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65SecAtrbREADWRITE.h -------------

#pragma once

class CAct65SecAtrbREADWRITE : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "R/W";

public:

CAct65SecAtrbREADWRITE();

virtual ~CAct65SecAtrbREADWRITE();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65SecAtrbSIZE.h -------------

#pragma once

class CAct65SecAtrbSIZE : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "SIZE";

public:

CAct65SecAtrbSIZE();

virtual ~CAct65SecAtrbSIZE();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65SecAtrbSTART.h -------------

#pragma once

class CAct65SecAtrbSTART : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "ATRB-START";

public:

CAct65SecAtrbSTART();

virtual ~CAct65SecAtrbSTART();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65SecAtrbZEROPAGE.h -------------

#pragma once

class CAct65SecAtrbZEROPAGE : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "ZERO-PAGE";

public:

CAct65SecAtrbZEROPAGE();

virtual ~CAct65SecAtrbZEROPAGE();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65SECTION.h -------------

#pragma once

class CAct65SECTION : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "SECTION";

public:

CAct65SECTION();

virtual ~CAct65SECTION();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65SUB.h -------------

#pragma once

class CAct65SUB : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "SUB";

public:

CAct65SUB();

virtual ~CAct65SUB();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65THEN.h -------------

#pragma once

class CAct65THEN : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "THEN";

public:

CAct65THEN();

virtual ~CAct65THEN();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65TRUE.h -------------

#pragma once

class CAct65TRUE : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "TRUE";

public:

CAct65TRUE();

virtual ~CAct65TRUE();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65TYPEDEF.h -------------

#pragma once

class CAct65tTYPEDEF : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "TYPEDEF";

public:

CAct65tTYPEDEF();

virtual ~CAct65tTYPEDEF();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65UNTILL.h -------------

#pragma once

class CAct65UNTILL : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "UNTILL";

public:

CAct65UNTILL();

virtual ~CAct65UNTILL();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65UpperPart.h -------------

#pragma once

class CAct65UpperPart : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "UpPart";

public:

CAct65UpperPart();

virtual ~CAct65UpperPart();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65VECTOR.h -------------

#pragma once

class CAct65VECTOR : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "VECTOR";

public:

CAct65VECTOR();

virtual ~CAct65VECTOR();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65WHILE.h -------------

#pragma once

class CAct65WHILE : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "ASM";

public:

CAct65WHILE();

virtual ~CAct65WHILE();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65XOR.h -------------

#pragma once

class CAct65XOR : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = "XOR";

public:

CAct65XOR();

virtual ~CAct65XOR();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65XREG.h -------------

#pragma once

class CAct65XREG : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = ".X";

public:

CAct65XREG();

virtual ~CAct65XREG();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- Act65YREG.h -------------

#pragma once

class CAct65YREG : public CAstNode

{

inline static const char\* m\_pNodeTyypeName = ".Y";

public:

CAct65YREG();

virtual ~CAct65YREG();

bool Create(CAstNode\* pChild, CAstNode\* pNext);

virtual CValue\* Process();

};

---------- ActionApp.h -------------

#pragma once

class CActionApp

{

char\* m\_pSourceFile;

char\* m\_pLogFile;

char\* m\_pObjectFile;

char\* m\_pBinaryFile;

FILE\* m\_pfSrc;

FILE\* m\_pfLog;

FILE\* m\_pfBin;

FILE\* m\_pfObj;

CParser m\_ActParse;

public:

CActionApp();

virtual ~CActionApp();

virtual bool Create(int argc, char\* argv[]);

virtual bool Run();

bool OpenSource();

bool OpenLog();

bool OpenObj();

bool OpenBin();

void CloseSource();

void CloseLog();

void CloseObj();

void CloseBin();

void CloseAll();

char\* GetSourceFileName() {

return m\_pSourceFile;

}

char\* GetLogFileName() {

return m\_pLogFile;

}

char\* GetObjFileName() {

return m\_pObjectFile;

}

char\* GetBinFileName() {

return m\_pBinaryFile;

}

virtual FILE\* SrcFile() {

return m\_pfSrc;

}

virtual FILE\* LogFile() {

return m\_pfLog;

}

virtual FILE\* ObjFile() {

return m\_pfObj;

}

virtual FILE\* BinFile() {

return m\_pfBin;

}

virtual CParser\* GetParser()

{

return &m\_ActParse;

}

char\* IndentString(char\* s, int Indent, int c);

static void Dump(

FILE\* pOut,

char\* pMem,

unsigned StartAddress,

unsigned Size

);

//-----------------------------------------

char\* CreateIndentString(char\* s, int n, int Indent, int c = ' ');

};

extern CActionApp\* Act();

---------- ActionAstTree.h -------------

#pragma once

class CActionAstTree : public CAstTree

{

public:

CActionAstTree();

virtual ~CActionAstTree();

bool Create();

virtual CAstNode\* MakeNewNode(

int Type,

CAstNode\* Child,

CAstNode\* Next

);

};

---------- ActStatusFlags.h -------------

#pragma once

class CActStatusFlags : public CAstNode

{

public:

CActStatusFlags();

virtual ~CActStatusFlags();

bool Create();

};

---------- AddressSizeStackItem.h -------------

#pragma once

class CAddressSizeStackItem : public CStackItem

{

CSection::AddressSize m\_OpRandAdrSize;

public:

CAddressSizeStackItem();

virtual ~CAddressSizeStackItem();

bool Create();

CSection::AddressSize GetOprandSize() { return m\_OpRandAdrSize; }

void SetOprandSize(CSection::AddressSize OpSize) { m\_OpRandAdrSize = OpSize; }

};

---------- AstNode.h -------------

#pragma once

class CAstNode

{

inline static int m\_NodeCount = 0;

int m\_NodeID;

int m\_NodeType;

const char\* m\_pNodeName;

CAstNode\* m\_pStart;

CAstNode\* m\_pNext;

CAstNode\* m\_pPrev;

CAstNode\* m\_pHead;

CAstNode\* m\_pTail;

CAstNode\* m\_pChild;

CAstNode\* m\_pParent;

CValue\* m\_pValue;

public:

CAstNode();

CAstNode(int NodeType);

virtual ~CAstNode();

bool CreateNode(

CAstNode\* pChild = 0,

CAstNode\* pNext = 0

);

virtual CValue\* Process() = 0;

virtual void Print(FILE\* pOut, int Indent, char\* s);

void AddToHead(CAstNode\* pN);

void AddToTail(CAstNode\* pN);

// Getter/Setter Methods

CAstNode\* GetHead() { return m\_pHead; }

CAstNode\* GetTail() { return m\_pTail; }

CAstNode\* GetStart() { return m\_pStart; }

CAstNode\* GetNext() { return m\_pNext; }

CAstNode\* GetPrev() { return m\_pPrev; }

CAstNode\* GetChild() { return m\_pChild; }

CAstNode\* GetParent() { return m\_pParent; }

void SetHead(CAstNode\* pAN) { m\_pHead = pAN; }

void SetTail(CAstNode\* pAN) { m\_pTail = pAN; }

void SetStart(CAstNode\* pAN) { m\_pStart = pAN; }

void SetNext(CAstNode\* pAN) { m\_pNext = pAN; }

void SetPrev(CAstNode\* pAN) { m\_pPrev = pAN; }

void SetChild(CAstNode\* pAN) { m\_pChild = pAN; }

void SetParent(CAstNode\* pAN) { m\_pParent = pAN; }

//---------------------------------------------------

int GetID() { return m\_NodeID; }

void SetID(int Id) { m\_NodeID = Id; }

int GetNodeType() { return m\_NodeType; }

void SetNodeType(int Nt) { m\_NodeType = Nt; }

const char\* GetNodeName() { return m\_pNodeName; }

void SetNodeName(const char\* pName) {

m\_pNodeName = pName;

}

bool SetSymbol(CBin\* pSym) {

bool rV = true;

if (m\_pValue)

m\_pValue->SetSymbol(pSym);

else

rV = false;

return rV;

}

CBin\* GetSymbol() {

CBin\* pSym = 0;

if (m\_pValue)

pSym = m\_pValue->GetSymbol();

return pSym; }

};

---------- AstTree.h -------------

#pragma once

class CAstTree

{

CAstNode\* m\_pRoot;

public:

CAstTree();

virtual ~CAstTree();

bool Create();

void Print(FILE\* pOut);

void TraverseTree(

FILE\* pOut,

CAstNode\* pNode,

char\* s

);

//--------------------------------

void SetRootNode(CAstNode\* pRoot) { m\_pRoot = pRoot; }

CAstNode\* GetRootNode() { return m\_pRoot; }

};

---------- Bin.h -------------

#pragma once

class CBin

{

public:

enum BinType {

ALL,

SYMBOL,

SECTION,

WHERE\_USED,

UNRESOLVED

};

private:

CBin\* m\_pNext;

CBin\* m\_pPrev;

CBin\* m\_pHead;

CBin\* m\_pTail;

char\* m\_pName;

BinType m\_Type;

Token m\_Token;

char m\_strName[MAX\_SYMBOL\_NAME\_LEN];

public:

CBin() {

m\_pHead = 0;

m\_pTail = 0;

m\_Token = Token(0);

m\_pNext = 0;

m\_pPrev = 0;

m\_pName = new char[MAX\_NAME\_LEN];

m\_Type = BinType(0);

m\_Token = Token(0);

for (int i = 0; i < MAX\_SYMBOL\_NAME\_LEN; ++i)

m\_strName[i] = 0;

}

CBin(BinType type) {

m\_pName = new char[MAX\_NAME\_LEN];

m\_pNext = 0;

m\_pPrev = 0;

m\_pHead = 0;

m\_pTail = 0;

m\_Token = Token::ENDOFTOKENS;

m\_Type = type;

for (int i = 0; i < MAX\_SYMBOL\_NAME\_LEN; ++i)

m\_strName[i] = 0;

}

virtual ~CBin() {

if (m\_pName)

delete[]m\_pName;

};

bool Create() { return true; }

CBin\* GetNext() { return m\_pNext; }

void SetNext(CBin\* pB) { m\_pNext = pB; }

CBin\* GetPrev() { return m\_pPrev; }

void SetPrev(CBin\* pB) { m\_pPrev = pB; }

virtual char\* GetName(void) { return m\_pName; }

virtual void SetName(const char\* pName) {

int l = (int)strlen(pName) + 1;

if (m\_pName) delete m\_pName;

m\_pName = new char[l];

strcpy\_s(m\_pName, l, pName);

}

virtual bool Compare(const char \*name, int aux = 0);

BinType GetType() { return m\_Type; }

// virtual bool Compare(const char\* name) {

// bool rV = false;

// if (strcmp(m\_pName, name) == 0) rV = true;

// return rV;

//}

void SetToken(Token t) { m\_Token = t; }

Token GetToken() { return m\_Token; }

virtual void Print(FILE\* pOut, const char\* s = 0);

//------------------------------------------

// SubList Methods

//------------------------------------------

void Add(CBin\* pSym);

CBin\* Find(const char\* pName, CBin::BinType Type);

void Delete(CBin\* pSym);

void Unlink(CBin\* pSym);

inline void SetHead(CBin\* pH) { m\_pHead = pH; }

inline CBin\* GetHead() { return m\_pHead; }

inline void SetTail(CBin\* pT) { m\_pTail = pT; }

inline CBin\* GetTail() { return m\_pTail; }

};

---------- BinStackItem.h -------------

#pragma once

class CBinStackItem : public CStackItem

{

CBin\* m\_pBin;

public:

CBinStackItem();

virtual ~CBinStackItem();

bool Create();

CBin\* GetBin() { return m\_pBin; }

void SetBin(CBin\* pB) { m\_pBin = pB; }

};

---------- Bucket.h -------------

#pragma once

class CBucket

{

CBin\* m\_pHead;

CBin\* m\_pTail;

public:

CBucket() {

m\_pHead = 0;

m\_pTail = 0;

}

virtual ~CBucket();

void Add(CBin\* pSym);

bool Create() { return true; }

CBin\* Find(const char\* pName, int scope);

void Delete(CBin\* pSym);

void Unlink(CBin\* pSym);

inline void SetHead(CBin\* pH) { m\_pHead = pH; }

inline CBin\* GetHead() { return m\_pHead; }

inline void SetTail(CBin\* pT) { m\_pTail = pT; }

inline CBin\* GetTail() { return m\_pTail; }

};

---------- DataSizeStackItem.h -------------

#pragma once

class CDataSizeStackItem : public CStackItem

{

int m\_ObjectSize;

public:

CDataSizeStackItem();

virtual ~CDataSizeStackItem();

bool Create();

void SetSize(int size) { m\_ObjectSize = size; }

int GetSize() { return m\_ObjectSize; }

};

---------- Instruction.h -------------

#pragma once

class CInstruction : public CStackItem

{

//-------------------------------

// Bytes are Little Endian Order

//-------------------------------

char m\_OpCode;

char m\_LowByte;

char m\_HighByte;

char m\_unused;

Token m\_OpCodeToken;

int m\_ByteCount;

bool m\_UnResolved;

public:

CInstruction();

~CInstruction();

bool Create();

void SetByteCount(int n) { m\_ByteCount = n; }

int GetByteCount() { return m\_ByteCount; }

void SetOpCodeToken(Token Op) { m\_OpCodeToken = Op; }

Token GetOpCodeToken() { return m\_OpCodeToken; }

void SetOpcode(unsigned op) { m\_OpCode = op; }

void AddToOpCode(unsigned v) { m\_OpCode += v; }

char GetOpCode() { return m\_OpCode; }

bool IsUnResolved() { return m\_UnResolved; }

bool IsResolved() { return !m\_UnResolved; }

void SetUnResolved() { m\_UnResolved = true; }

void SetOperandW(unsigned v) {

m\_LowByte = v & 0xff;

m\_HighByte = (v & 0x0ff00) >> 8;

}

unsigned GetOperandW() {

unsigned v;

v = (m\_LowByte & 0x0ff) | ((m\_HighByte & 0x0ff) << 8);

return v;

}

unsigned char GetLowByte() { return m\_LowByte; }

void SetLowByte(unsigned v) { m\_LowByte = v; }

unsigned char GetHighByte() { return m\_HighByte;}

void SetHighByte(unsigned v) { m\_HighByte = v; }

void AddToCurrentSection();

bool IsRelative();

};

---------- Lexer.h -------------

#pragma once

class CLexer

{

public:

private:

struct AdressModeItem {

AdrModeType Mode;

int inc;

AdressModeItem() {

Mode = AdrModeType::NA;

inc = 0;

}

AdressModeItem(AdrModeType T, int I) {

Mode = T;

inc = I;

}

int GetInc(AdrModeType ModeType) {

//---------------------------

// GetInc

// Compares the address mode

// type of this object to the

// one that is desired. If

// they are equal, we get

// back the increment value,

// If they are not equal, we

// get back -1

//---------------------------

int rV = -1;

if (ModeType == Mode)

rV = inc;

return rV;

}

};

struct AdressModeLUT {

CLexer::AdressModeItem\* ModeInc;

int m\_nElements;

AdressModeLUT() {

ModeInc = 0;

m\_nElements = 0;

}

AdressModeLUT(int n, AdressModeItem\* MI) {

ModeInc = MI;

m\_nElements = n;

}

int GetInc(AdrModeType Type) {

int i, rV = -1;

bool Loop = true;

for (i = 0; Loop && i < m\_nElements; ++i)

{

rV = ModeInc[i].GetInc(Type);

if (rV >= 0)

Loop = false;

}

return rV;

}

};

//alu access modes

static inline AdressModeItem Order8[8] = {

{AdrModeType::INDIRECT\_X\_ADR, 0},

{AdrModeType::ZERO\_PAGE\_ADR, 4},

{AdrModeType::IMMEDIATE\_ADR, 8},

{AdrModeType::ABSOLUTE\_ADR, 12},

{AdrModeType::INDIRECT\_Y\_ADR, 16},

{AdrModeType::ZERO\_PAGE\_X\_ADR, 20},

{AdrModeType::ABSOLUTE\_Y\_ADR, 24},

{AdrModeType::ABSOLUTE\_X\_ADR, 28}

};

static inline AdressModeLUT LUT\_Order8 = { 8, Order8 };

// sta address modes

static inline AdressModeItem Order7[7] = {

{AdrModeType::INDIRECT\_X\_ADR, 0},

{AdrModeType::ZERO\_PAGE\_ADR, 4},

{AdrModeType::ABSOLUTE\_ADR, 12},

{AdrModeType::INDIRECT\_Y\_ADR, 16},

{AdrModeType::ZERO\_PAGE\_X\_ADR, 20},

{AdrModeType::ABSOLUTE\_Y\_ADR, 24},

{AdrModeType::ABSOLUTE\_X\_ADR, 28}

};

static inline AdressModeLUT LUT\_Order7 = { 8, Order7 };

//---------------------------------------

// ASL, LSR, ROR, ROL

static inline AdressModeItem Order5[5] = {

{AdrModeType::ZERO\_PAGE\_ADR, 0},

{AdrModeType::ACCUMULATOR,4},

{AdrModeType::ABSOLUTE\_ADR, 8},

{AdrModeType::ZERO\_PAGE\_X\_ADR, 16},

{AdrModeType::ABSOLUTE\_X\_ADR, 24}

};

static inline AdressModeLUT LUT\_Order5 = { 5, Order5 };

//----------------------------------------------

static inline AdressModeItem Order5\_LDY[5] = {

{AdrModeType::IMMEDIATE\_ADR,0},

{AdrModeType::ZERO\_PAGE\_ADR, 4},

{AdrModeType::ABSOLUTE\_ADR, 12},

{AdrModeType::ZERO\_PAGE\_X\_ADR, 20},

{AdrModeType::ABSOLUTE\_X\_ADR, 28}

};

static inline AdressModeLUT LUT\_Order5\_LDY = { 5, Order5\_LDY };

//---------------------------------------------

static inline AdressModeItem Order5\_LDX[5] = {

{AdrModeType::IMMEDIATE\_ADR,0},

{AdrModeType::ZERO\_PAGE\_ADR, 4},

{AdrModeType::ABSOLUTE\_ADR, 12},

{AdrModeType::ZERO\_PAGE\_Y\_ADR, 20},

{AdrModeType::ABSOLUTE\_Y\_ADR, 28}

};

static inline AdressModeLUT LUT\_Order5\_LDX = { 5, Order5\_LDX };

//---------------------------------------------

static inline AdressModeItem Order4[4] = {

{AdrModeType::ZERO\_PAGE\_ADR, 0},

{AdrModeType::ABSOLUTE\_ADR, 8},

{AdrModeType::ZERO\_PAGE\_X\_ADR, 16},

{AdrModeType::ABSOLUTE\_X\_ADR, 24}

};

static inline AdressModeLUT LUT\_Order4 = { 4, Order4 };

//---------------------------------------------

static inline AdressModeItem Order3\_STX[3] = { //STX

{AdrModeType::ZERO\_PAGE\_ADR, 0},

{AdrModeType::ZERO\_PAGE\_Y\_ADR,16},

{AdrModeType::ABSOLUTE\_ADR, 8},

};

static inline AdressModeLUT LUT\_Order3\_STX = { 3, Order3\_STX };

//---------------------------------------------

static inline AdressModeItem Order3\_STY[3] = { //STY

{AdrModeType::ZERO\_PAGE\_ADR, 0},

{AdrModeType::ZERO\_PAGE\_X\_ADR,16},

{AdrModeType::ABSOLUTE\_ADR, 8},

};

static inline AdressModeLUT LUT\_Order3\_STY = { 3, Order3\_STY };

//---------------------------------------------

static inline AdressModeItem Order3\_CPY\_CPX[3] = { //CPY/CPX

{AdrModeType::IMMEDIATE\_ADR,0},

{AdrModeType::ZERO\_PAGE\_ADR, 4},

{AdrModeType::ABSOLUTE\_ADR, 12}

};

static inline AdressModeLUT LUT\_Order3\_\_CPY\_CPX = { 3, Order3\_CPY\_CPX };

//---------------------------------------------

static inline AdressModeItem Order2[2] = {

{AdrModeType::ABSOLUTE\_ADR,0 },

{AdrModeType::INDIRECT\_ADR,32 }

};

static inline AdressModeLUT LUT\_Order2 = { 3, Order2 };

//---------------------------------------------

static inline AdressModeItem Order2\_BIT[2] = {

{AdrModeType::ZERO\_PAGE\_ADR,0 },

{AdrModeType::ABSOLUTE\_ADR,8 }

};

static inline AdressModeLUT LUT\_Order2\_BIT = { 3, Order2\_BIT };

//---------------------------------------------

static inline AdressModeItem Order1[1] = {

{AdrModeType::ABSOLUTE\_ADR,0 }

};

static inline AdressModeLUT LUT\_Order1 = { 3, Order1 };

struct KeyWord {

Token m\_TokenID; //token value

const char\* m\_Name; //token name

int m\_MaxBytes; //Maximum number of bytes for instruction

int m\_OpCode; //base opcode

Processor m\_Processor; //for which processor

AdressModeLUT\* m\_pAddresModeLUT;

int NumOfAdrModes;

const char\* LookupToName(Token Toke);

int FindInc(AdrModeType AdrMode);

Token LookupToToken(const char\* pName);

};

static inline KeyWord KeyWords[] = {

{Token::ENDOFFILE,"End Of File"}, //1

{Token::IDENT,"IDENT"}, //2

{Token::NUMBER,"NUMBER"}, //3

{Token::MOD,"MOD"},

//--------- Assignment -------------

{ Token::ASSIGN\_ADD,"==+", 0 },

{ Token::ASSIGN\_SUB,"==-", 0 },

{ Token::ASSIGN\_MUL,"==\*", 0 },

{ Token::ASSIGN\_DIV,"==/", 0 },

{ Token::ASSIGN\_MOD,"==MOD", 0 },

{ Token::ASSIGN\_AND,"==&", 0 },

{ Token::ASSIGN\_OR,"==%", 0 },

{ Token::ASSIGN\_XOR,"==XOR", 0 },

{ Token::ASSIGN\_RSH,"==RSH", 0 },

{ Token::ASSIGN\_LSH,"==LSH", 0 },

//---------- Operators ------------

{Token::LSH,"LSH"},

{Token::RSH,"RSH"},

{Token::OR,"OR"},

{Token::XOR, "XOR"},

{Token::MOD,"MOD", 0 },

{Token::GTEQ,">=", 0 },

{Token::LTEQ,"<=", 0 },

//--------------- Functions --------------

{Token::PROC,"PROC"}, // Proceedure Declaration

{Token::FUNC,"FUNC"}, // Function Declaration

{Token::INTERRUPT,"INTERRUPT"},

{Token::PROC\_IDENT,"PROC Idnet"}, // Proceedure Declaration

{Token::FUNC\_IDENT,"FUNC Ident"}, // Function Declaration

{Token::INTERRUPT\_IDENT,"INTERRUPT Indent"},

//------ Data Types -----------------

{Token::BOOL,"BOOL"},

{Token::BYTE,"BYTE"},

{Token::CHAR,"CHAR"},

{Token::INT,"INT"},

{Token::CARD,"CARD"},

{Token::TYPE, "TYPE"},

{Token::RECORDTYPE,"Record Type (TYPE)"},

{Token::POINTER,"POINTER"},

{Token::ARRAY,"ARRAY"},

//--------- Statements --------------

{Token::MODULE,"MODULE"},

{Token::VECTOR,"VECTOR"},

{Token::FOR,"FOR"},

{Token::TO,"TO"},

{Token::STEP,"STEP"},

{Token::IF,"IF"},

{Token::IFF,"IFF"},

{Token::THEN,"THEN"},

{Token::ELSEIF,"ELSEIF"},

{Token::ELSE,"ELSE"},

{Token::FI,"FI"},

{Token::FFI,"FFI"},

{Token::WHILE,"WHILE"},

{Token::DO,"DO"},

{Token::OD,"OD"},

{Token::UNTIL,"UNTIL"},

{Token::EXIT,"EXIT"},

{Token::RETURN,"RETURN"},

{Token::ASM,"ASM"},

{Token::PUSH,"PUSH"},

{Token::POP,"POP"},

{Token::BREAK, "BREAK"},

{Token::BITS,"BITS"},

{Token::BEGIN,"BEGIN"},

{Token::END, "END"},

//--------- Compiler Directives -----------------

{Token::DEFINE,"DEFINE"},

{Token::SET,"SET"},

{Token::INCLUDE,"INCLUDE"},

//---------------------------------

// Assembler Tokens

//---------------------------------

//--------- Section ------------

{Token::SECTION,"SECTION", 0,0, Processor::ALL,NULL,0},

{Token::SECTION\_NAME,"SECTION NAME", 0,0, Processor::ALL,NULL,0},

{Token::START,"START", 0,0,Processor::ALL,NULL,0},

{Token::SIZE,"SIZE", 0,0,Processor::ALL,NULL,0},

{Token::NAME,"NAME", 0,0,Processor::ALL,NULL,0},

{Token::MODE,"MODE", 0,0,Processor::ALL,NULL,0},

{Token::READ\_WRTE,"READ\_WRITE", 0,0,Processor::ALL,NULL,0},

{Token::READ\_ONLY,"READ\_ONLY", 0,0,Processor::ALL,NULL,0},

{Token::ZEROPAGE,"ZEROPAGE", 0},

{Token::True,"true", 0},

{Token::False,"false", 0},

//----- Assembler Declarations

{Token::ORG,"ORG", 0,0, Processor::ALL,NULL,0},

{Token::DB,"DB", 0,0, Processor::ALL,NULL,0},

{Token::DW,"DW", 0,0, Processor::ALL,NULL,0},

{Token::DL,"DL", 0,0, Processor::ALL,NULL,0},

{Token::DAS,"DAS", 0,0, Processor::ALL,NULL,0},

{Token::DCS,"DCS", 0,0, Processor::ALL,NULL,0},

{Token::DS,"DS", 0,0, Processor::ALL,NULL,0},

//----------- Opcodes --------------

{Token::ADC,"ADC", 3 ,0X61, Processor::R6502, &LUT\_Order8,8},

{Token::AND,"AND", 3,0X21, Processor::R6502, &LUT\_Order8,8},

{Token::ASL,"ASL", 3,0X06, Processor::R6502, &LUT\_Order5,5},

{Token::BCC,"BCC", 2,0x90, Processor::R6502, NULL,0},

{Token::BCS,"BCS", 2,0XB0, Processor::R6502, NULL,0},

{Token::BEQ,"BEQ", 2,0XF0, Processor::R6502, NULL,0},

{Token::BMI,"BMI", 2,0x30, Processor::R6502, NULL,0},

{Token::BNE,"BNE", 2,0XD0, Processor::R6502, NULL,0},

{Token::BPL,"BPL", 2,0x10, Processor::R6502, NULL,0},

{Token::BVC,"BVC", 2,0x50, Processor::R6502, NULL,0},

{Token::BVS,"BVS", 2,0x70, Processor::R6502, NULL,0},

{Token::BIT,"BIT", 3,0X24, Processor::R6502, &LUT\_Order2\_BIT,2},

{Token::BRK,"BRK", 1,0x00, Processor::R6502, NULL,0},

{Token::CLC,"CLC", 1,0X18, Processor::R6502, NULL,0},

{Token::CLD,"CLD", 1,0xd8, Processor::R6502, NULL,0},

{Token::CLI,"CLI", 1,0X58, Processor::R6502, NULL,0},

{Token::CLV,"CLV", 1,0xb8, Processor::R6502, NULL,0},

{Token::CMP,"CMP", 3,0xc1, Processor::R6502, &LUT\_Order8,8},

{Token::CPX,"CPX", 3,0XE0, Processor::R6502, &LUT\_Order3\_\_CPY\_CPX,3},

{Token::CPY,"CPY", 3,0XC0, Processor::R6502, &LUT\_Order3\_\_CPY\_CPX,3},

{Token::DEC,"DEC", 3,0xc6, Processor::R6502, &LUT\_Order4,4},

{Token::DEX,"DEX", 1,0xca, Processor::R6502, NULL,0},

{Token::DEY,"DEY", 1,0x88, Processor::R6502, NULL,0},

{Token::EOR,"EOR", 3,0X41, Processor::R6502, &LUT\_Order8,8},

{Token::INC,"INC", 3,0xe6, Processor::R6502, &LUT\_Order4,4},

{Token::INX,"INX", 1,0xe8, Processor::R6502, NULL,0},

{Token::INY,"INY", 1,0xc8, Processor::R6502, NULL,0},

{Token::JMP,"JMP", 3,0X4C, Processor::R6502, &LUT\_Order2,2},

{Token::JSR,"JSR", 3,0x20, Processor::R6502, &LUT\_Order1,1},

{Token::LDA, "LDA", 3, 0xa1, Processor::R6502, & LUT\_Order8, 8},

{ Token::LDX,"LDX", 3,0xA2, Processor::R6502, &LUT\_Order5\_LDX,5 },

{ Token::LDY,"LDY", 3,0xa0, Processor::R6502, &LUT\_Order5\_LDY,5 },

{ Token::LSR,"LSR", 3,0X46, Processor::R6502, &LUT\_Order5,5 },

{ Token::NOP,"NOP", 1,0xea, Processor::R6502, NULL,0 },

{ Token::ORA,"ORA", 3,0X01, Processor::R6502, &LUT\_Order8,8 },

{ Token::PHA,"PHA", 1,0X48, Processor::R6502, NULL,0 },

{ Token::PLA,"PLA", 1,0x68, Processor::R6502, NULL,0 },

{ Token::PHP,"PHP", 1,0X08, Processor::R6502, NULL,0 },

{ Token::PLP,"PLP", 1,0X28, Processor::R6502, NULL,0 },

{ Token::ROL,"ROL",3,0X26, Processor::R6502, &LUT\_Order5,5 },

{ Token::ROR,"ROR",3,0X66, Processor::R6502, &LUT\_Order5,5 },

{ Token::RTI,"RTI", 1,0x40, Processor::R6502, NULL,0 },

{ Token::RTS,"RTS", 1,0x60, Processor::R6502, NULL,0 },

{ Token::SBC,"SBC",3,0xe1, Processor::R6502, &LUT\_Order8,8 },

{ Token::STA,"STA",3,0x81, Processor::R6502, &LUT\_Order7,7 },

{ Token::SEC,"SEC", 1,0x38, Processor::R6502, NULL,0 },

{ Token::SED,"SED", 1,0Xf8, Processor::R6502, NULL,0 },

{ Token::SEI,"SEI", 1,0x78, Processor::R6502, NULL,0 },

{ Token::STX,"STX",3,0x86, Processor::R6502, &LUT\_Order3\_STX,3 },

{ Token::STY,"STY",3,0x84, Processor::R6502, &LUT\_Order3\_STY,3 },

{ Token::TAX,"TAX", 1,0xaa, Processor::R6502, NULL,0 },

{ Token::TAY,"TAY", 1,0xa8, Processor::R6502, NULL,0 },

{ Token::TXA,"TXA", 1,0x8a, Processor::R6502, NULL,0 },

{ Token::TYA,"TYA", 1,0x98, Processor::R6502, NULL,0 },

{ Token::TXS,"TXS", 1,0x9a, Processor::R6502, NULL,0 },

{ Token::TSX,"TSX", 1,0xba, Processor::R6502, NULL,0 },

//--------------------------------------------

// Lower Case Opcodes

//--------------------------------------------

{ Token::ADC,"adc", 3 ,0X61, Processor::R6502, &LUT\_Order8,8 },

{ Token::AND,"and", 3,0X21, Processor::R6502, &LUT\_Order8,8 },

{ Token::ASL,"asl", 3,0X06, Processor::R6502, &LUT\_Order5,5 },

{ Token::BCC,"bcc", 2,0x90, Processor::R6502, NULL,0 },

{ Token::BCS,"bcs", 2,0XB0, Processor::R6502, NULL,0 },

{ Token::BEQ,"beq", 2,0XF0, Processor::R6502, NULL,0 },

{ Token::BMI,"bmi", 2,0x30, Processor::R6502, NULL,0 },

{ Token::BNE,"bne", 2,0XD0, Processor::R6502, NULL,0 },

{ Token::BPL,"bpl", 2,0x10, Processor::R6502, NULL,0 },

{ Token::BVC,"bvc", 2,0x50, Processor::R6502, NULL,0 },

{ Token::BVS,"bvs", 2,0x70, Processor::R6502, NULL,0 },

{ Token::BIT,"bit", 3,0X24, Processor::R6502, &LUT\_Order2\_BIT,2 },

{ Token::BRK,"brk", 1,0x00, Processor::R6502, NULL,0 },

{ Token::CLC,"clc", 1,0X18, Processor::R6502, NULL,0 },

{ Token::CLD,"cld", 1,0xd8, Processor::R6502, NULL,0 },

{ Token::CLI,"cli", 1,0X58, Processor::R6502, NULL,0 },

{ Token::CLV,"clv", 1,0xb8, Processor::R6502, NULL,0 },

{ Token::CMP,"cmp", 3,0xc1, Processor::R6502, &LUT\_Order8,8 },

{ Token::CPX,"cpx", 3,0XE0, Processor::R6502, &LUT\_Order3\_\_CPY\_CPX,3 },

{ Token::CPY,"cpy", 3,0XC0, Processor::R6502, &LUT\_Order3\_\_CPY\_CPX,3 },

{ Token::DEC,"dec", 3,0xc6, Processor::R6502, &LUT\_Order4,4 },

{ Token::DEX,"dex", 1,0xca, Processor::R6502, NULL,0 },

{ Token::DEY,"dey", 1,0x88, Processor::R6502, NULL,0 },

{ Token::EOR,"eor", 3,0X41, Processor::R6502, &LUT\_Order8,8 },

{ Token::INC,"inc", 3,0xe6, Processor::R6502, &LUT\_Order4,4 },

{ Token::INX,"inx", 1,0xe8, Processor::R6502, NULL,0 },

{ Token::INY,"iny", 1,0xc8, Processor::R6502, NULL,0 },

{ Token::JMP,"jmp", 3,0X4C, Processor::R6502, &LUT\_Order2,2 },

{ Token::JSR,"jsr", 3,0x20, Processor::R6502, &LUT\_Order1,1 },

{ Token::LDA,"lda", 3,0xa1, Processor::R6502, &LUT\_Order8,8 },

{ Token::LDX,"ldx", 3,0xA2, Processor::R6502, &LUT\_Order5\_LDX,5 },

{ Token::LDY,"ldy", 3,0xa0, Processor::R6502, &LUT\_Order5\_LDY,5 },

{ Token::LSR,"lsr", 3,0X46, Processor::R6502, &LUT\_Order5,5 },

{ Token::NOP,"nop", 1,0xea, Processor::R6502, NULL,0 },

{ Token::ORA,"ora", 3,0X01, Processor::R6502, &LUT\_Order8,8 },

{ Token::PHA,"pha", 1,0X48, Processor::R6502, NULL,0 },

{ Token::PLA,"pla", 1,0x68, Processor::R6502, NULL,0 },

{ Token::PHP,"php", 1,0X08, Processor::R6502, NULL,0 },

{ Token::PLP,"plp", 1,0X28, Processor::R6502, NULL,0 },

{ Token::ROL,"rol",3,0X26, Processor::R6502, &LUT\_Order5,5 },

{ Token::ROR,"ror",3,0X66, Processor::R6502, &LUT\_Order5,5 },

{ Token::RTI,"rti", 1,0x40, Processor::R6502, NULL,0 },

{ Token::RTS,"rts", 1,0x60, Processor::R6502, NULL,0 },

{ Token::SBC,"sbc",3,0xe1, Processor::R6502, &LUT\_Order8,8 },

{ Token::STA,"sta",3,0x81, Processor::R6502, &LUT\_Order7,7 },

{ Token::SEC,"sec", 1,0x38, Processor::R6502, NULL,0 },

{ Token::SED,"sed", 1,0Xf8, Processor::R6502, NULL,0 },

{ Token::SEI,"sei", 1,0x78, Processor::R6502, NULL,0 },

{ Token::STX,"stx",3,0x86, Processor::R6502, &LUT\_Order3\_STX,3 },

{ Token::STY,"sty",3,0x84, Processor::R6502, &LUT\_Order3\_STY,3 },

{ Token::TAX,"tax", 1,0xaa, Processor::R6502, NULL,0 },

{ Token::TAY,"tay", 1,0xa8, Processor::R6502, NULL,0 },

{ Token::TXA,"txa", 1,0x8a, Processor::R6502, NULL,0 },

{ Token::TYA,"tya", 1,0x98, Processor::R6502, NULL,0 },

{ Token::TXS,"txs", 1,0x9a, Processor::R6502, NULL,0 },

{ Token::TSX,"tsx", 1,0xba, Processor::R6502, NULL,0 },

{ Token::ENDOFOPCODES,"EOO", 0, 0, Processor::ALL,NULL,0 },

//------- Lables -------------

{ Token::LOCAL\_LABEL,"Local Lable", 0,0,Processor::ALL,NULL,0 },

{ Token::GLOBAL\_LABLE,"Global Lable", 0,0,Processor::ALL,NULL,0 },

//-------- Registers -------------

{ Token::AREG,"A", 0,0, Processor::ALL,NULL,0 },

{ Token::XREG,"X", 0,0, Processor::ALL,NULL,0 },

{ Token::YREG,"Y", 0,0, Processor::ALL,NULL,0 },

{ Token::SPREG,"S", 0,0, Processor::ALL,NULL,0 },

{ Token::PSREG,"P", 0,0, Processor::ALL,NULL,0 },

{ Token::AREG,"a", 0,0, Processor::ALL,NULL,0 },

{ Token::XREG,"x", 0,0, Processor::ALL,NULL,0 },

{ Token::YREG,"y", 0,0, Processor::ALL,NULL,0 },

{ Token::SPREG,"s", 0,0, Processor::ALL,NULL,0 },

{ Token::PSREG,"p", 0,0, Processor::ALL,NULL,0 },

//--------- Status Register Bits -------------

{ Token::NEG,"N" },

{ Token::ZERO,"Z" },

{Token::CARRY,"C"},

{Token::OVERFLOW,"O"},

{Token::DECIMAL\_MODE,"D"},

{Token::IRQENABLE,"I"},

//-------- Processor Selection -------

{ Token::PROCESSOR,"PROCESSOR", 0,0,Processor::ALL,NULL,0 },

{ Token::R6502,"R6502", 0,0, Processor::ALL,NULL,0 },

{ Token::WD65C02,"WD65C02", 0,0, Processor::ALL ,NULL,0 },

{ Token::WD65C816,"WD65C816", 0,0, Processor::ALL,NULL,0 },

//----------------------------------

// Misc

//----------------------------------

{ Token::EPROC,"EPROC", 0 },

{Token::CHAR\_CONSTANT,"Character Constant"},

{ Token('('),"(", 0 },

{ Token(')'),")", 0 },

{ Token('{'),"{", 0 },

{ Token('}'),"}", 0 },

{ Token('['),"[", 0 },

{ Token(']'),"]", 0 },

{ Token('<'),"<", 0 },

{ Token('>'),">", 0 },

{ Token('!'),"!", 0 },

{ Token('@'),"@", 0 },

{ Token('#'),"#", 0 },

{ Token('$'),"$", 0 },

{ Token('%'),"%", 0 },

{ Token('^'),"^", 0 },

{ Token('&'),"&", 0 },

{ Token('\*'),"\*", 0 },

{ Token('-'),"-", 0 },

{ Token('='),"=", 0 },

{ Token('+'),"+", 0 },

{ Token(','),",", 0 },

{ Token('.'),".", 0 },

{ Token('/'),"/", 0 },

{ Token('?'),"?", 0 },

{ Token('~'),"~", 0 },

{Token::ENDOFTOKENS,NULL}

};

CSymTab m\_SymbolTable;

int m\_UngetBuffer;

char m\_aLexBuff[256];

int m\_LexBuffIndex;

int m\_Line;

int m\_Col;

int m\_Number;

CSymbol\* m\_pLexSymbol;

char\* m\_pFileBuffeer;

int m\_InFileSize;

int m\_FileIndex;

bool m\_bAsmMode;

public:

CLexer();

virtual ~CLexer();

bool Create();

FILE\* LogFile();

bool IsAsmMode() { return m\_bAsmMode; }

void SetActionMode() { m\_bAsmMode = false; }

void SetAsmMode() { m\_bAsmMode = true; }

int LexGet();

int LexLook(int index);

void LexUnGet(int c);

bool IsValidHexNumber(int c);

bool IsValidNumber(int c);

bool IsValidNameChar(int c);

bool IsWhiteSpace(int c);

bool IsValidAssignmentOperator(int c);

Token Lex();

//------------------------------------------

// Symbol Table Methods

//------------------------------------------

CBin\* LookupSymbol(const char\* pName);

CSymTab\* GetSymTab() { return &m\_SymbolTable; }

//------------------------------------------

// Keyword Lookup Methods

//------------------------------------------

Token LookupKeyword(const char\* pKeyword);

KeyWord\* FindKeyword(Token KeywordToken);

Processor LookupProcessor(Token KeywordToken);

int LookupOpcode(Token OpcodeToken);

int GetOpcode(Token OpCodeToken);

KeyWord\* GetKeyWords() { return KeyWords; }

//--------------------------------------

// Symbol Methods

//--------------------------------------

CSymbol\* GetLexSymbol() { return m\_pLexSymbol; }

int GetNumber() { return m\_Number; }

char\* GetLexBuffer() { return m\_aLexBuff; }

int GetLineNumber() { return m\_Line; }

int GetColunm() { return m\_Col; }

};

---------- LHead.h -------------

#pragma once

class CLHead

{

public:

Token m\_Token;

CAstNode\* m\_pNode;

public:

CLHead();

CLHead(Token T, CAstNode\* pN) {

m\_Token = T;

m\_pNode = pN;

}

virtual ~CLHead();

bool Create();

void SetToken(Token T) { m\_Token = T; }

Token GetToken() { return m\_Token; }

void SetNode(CAstNode\* pN) { m\_pNode = pN; }

CAstNode\* GetNode() {

return m\_pNode;

}

void operator=(const CLHead & pLH);

};

---------- NumberStackItem.h -------------

#pragma once

class CSymbol;

class CNumberStackItem :public CStackItem

{

int m\_Value;

CSymbol\* m\_pSymbol;

public:

CNumberStackItem();

virtual ~CNumberStackItem();

bool Create();

void SetValue(int v) { m\_Value = v; }

int GetValue() { return m\_Value; }

CSymbol\* GetSymbol() { return m\_pSymbol; }

void SetSymbol(CSymbol\* pSym) { m\_pSymbol = pSym; }

//--------------------------------

// Operations

//--------------------------------

void Neg() { m\_Value = -m\_Value; }

void Add(CNumberStackItem\* pOperand) { m\_Value += pOperand->GetValue(); }

void Sub(CNumberStackItem\* pOperand) { m\_Value -= pOperand->GetValue(); }

};

#define VALUE(x) ((CNuberStackItem \*)(x))

---------- ObjFormatCode.h -------------

#pragma once

class CObjFormatCode : public CObjFormatSectionItem

{

unsigned short m\_BlockSize;

unsigned short m\_StartAddress;

char\* m\_pCode;

public:

CObjFormatCode();

virtual ~CObjFormatCode();

bool Create(int StartAddress, int BlockSize, const char\* pCode);

virtual void Out(FILE\* pOut);

virtual void Print(FILE\* pO, int Indent);

};

---------- ObjFormatData.h -------------

#pragma once

class CObjFormatData : public CObjFormatSectionItem

{

int m\_BlockSize;

int m\_Index;

char\* m\_pData;

public:

CObjFormatData();

virtual ~CObjFormatData();

bool Create(int BlockSize);

virtual void Out(FILE\* pOut);

virtual void Print(FILE\* pO, int Indent);

bool AddByte(char d);

};

---------- ObjFormatFile.h -------------

#pragma once

class CObjFormatFile

{

char m\_strName[512];

CObjFormatSection\* m\_pHead;

CObjFormatSection\* m\_pTail;

public:

CObjFormatFile();

virtual ~CObjFormatFile();

bool Create(const char\* pName);

CSymTab\* GetSymTab();

char\* GetObjectName() { return m\_strName; }

//-------------------------------------------------

CObjFormatSection\* GetHead() { return m\_pHead; }

void SetHead(CObjFormatSection\* pHead) { m\_pHead = pHead; }

CObjFormatSection\* GetTail() { return m\_pTail; }

void SetTail(CObjFormatSection\* pT) { m\_pTail = pT; };

//-------------------------------------------------

virtual void Print(FILE\* pO);

virtual void Out(FILE\* pO);

//-------------------------------------------------

void AddSection(CSection\* pSection);

void AddObjFormatSection(CObjFormatSection\* pAOFS);

};

---------- ObjFormatRelocation.h -------------

#pragma once

class CObjFormatRelocation : public CObjFormatSectionItem

{

public:

CObjFormatRelocation();

virtual ~CObjFormatRelocation();

bool Create();

virtual void Out(FILE\* pO);

virtual void Print(FILE\* pO, int Indent);

};

---------- ObjFormatSection.h -------------

#pragma once

class CObjFormatSection

{

struct SObjSectionData {

//-----------------------------------------------

// ‘SECTION’, \0 8 Bytes

// < Section Name> 32 bytes, 0 terminated

// < Attributes 2 bytes

// < Section Token> 2 bytes

// < File Offset From 0 of data> 4 bytes

//-----------------------------------------------

char m\_BlockTypeName[8];

char m\_SectionName[32];

unsigned short m\_Attributes;

unsigned short m\_SectionToken;

unsigned m\_FileOffset;

SObjSectionData() {

int i;

m\_Attributes = 0;

m\_SectionToken = 0;

m\_FileOffset = 0;

strcpy\_s(m\_BlockTypeName, 8, "SECTION");

for (i = 0; i < 32; ++i)

m\_SectionName[i] = 0;

}

};

CSection\* m\_pSection;

CObjFormatSection\* m\_pNext;

CObjFormatSection\* m\_pPrev;

CObjFormatSectionItem\* m\_pHead;

CObjFormatSectionItem\* m\_pTail;

public:

CObjFormatSection();

virtual ~CObjFormatSection();

bool Create(CSection\* pSection);

CSection\* GetSection() { return m\_pSection; }

//--------------------------------------------------

CObjFormatSectionItem\* GetHead() { return m\_pHead; }

void SetHead(CObjFormatSectionItem\* pH) { m\_pHead = pH; }

CObjFormatSectionItem\* GetTail() { return m\_pTail; }

void SetTail(CObjFormatSectionItem\* pT) { m\_pTail = pT; }

//---------------------------------------------------

void AddItem(CObjFormatSectionItem\* pItem);

//-------------------------------------------------

virtual void Print(FILE\* pO, int Indent);

virtual void Out(FILE\* pO);

//------------------------------------------------

void SetNext(CObjFormatSection\* pN) { m\_pNext = pN; }

CObjFormatSection\* GetNext() { return m\_pNext; }

void SetPrev(CObjFormatSection\* pP) { m\_pPrev = pP; }

CObjFormatSection\* GetPrev() { return m\_pPrev; }

};

---------- ObjFormatSectionItem.h -------------

#pragma once

class CObjFormatSectionItem

{

public:

enum class FormatType {

NONE,

CODE,

DATA,

RELOCATIONTABEL,

SYMBOLS,

SYMBOL

};

private:

struct FormatTypeLUT {

FormatType m\_LutType;

const char\* m\_pLutString;

};

inline static const FormatTypeLUT FormatStringLUT[] = {

{FormatType::NONE, "NONE"},

{FormatType::CODE, "CODE"},

{FormatType::DATA, "DATA"},

{FormatType::RELOCATIONTABEL, "RELOCATION TABEL"},

{FormatType::SYMBOLS, "SYMBOL Table"},

{FormatType::SYMBOL, "SYMBOL"},

{FormatType(-1), NULL},

};

FormatType m\_Type;

CObjFormatSectionItem\* m\_pNext;

CObjFormatSectionItem\* m\_pPrev;

public:

CObjFormatSectionItem();

virtual ~CObjFormatSectionItem();

bool Create(FormatType Type);

CObjFormatSectionItem\* GetNext() { return m\_pNext; }

void SetNext(CObjFormatSectionItem\* pN) { m\_pNext = pN; }

CObjFormatSectionItem\* GetPrev() { return m\_pPrev; }

void SetPrev(CObjFormatSectionItem\* pP) { m\_pPrev = pP; }

virtual void Out(FILE\* pOut);

virtual void Print(FILE\* pO, int Indent);

static const char\* LookupFormatSectionType(FormatType Type);

static FormatType LookupFormatSectionType(const char\* pName);

};

---------- ObjFormatSymbolItem.h -------------

#pragma once

class CObjFormatSymbolItem : public CObjFormatSectionItem

{

char m\_strName[MAX\_SYMBOL\_NAME\_LEN];

unsigned m\_DefinedAddress;

public:

CObjFormatSymbolItem();

virtual ~CObjFormatSymbolItem();

bool Create(CSymbol\* pSym);

virtual void Out(FILE\* pOut);

virtual void Print(FILE\* pO);

};

---------- ObjFormatSymbols.h -------------

#pragma once

class CObjFormatSymbols : public CObjFormatSectionItem

{

CObjFormatSymbolItem\* m\_pHead;

CObjFormatSymbolItem\* m\_pTail;

public:

CObjFormatSymbols();

virtual ~CObjFormatSymbols();

bool Create();

virtual void Print(FILE\* pO, int Indent);

virtual void Out(FILE\* pO);

virtual void Add(CObjFormatSymbolItem pSym);

};

---------- ObjType.h -------------

#pragma once

class CObjType

{

public:

enum class Spec {

NONE,

BYTE,

CHAR,

INT,

CARD,

BOOL,

TYPEDEF,

ARRAY,

POINTER,

PROC,

FUNC

};

private:

struct Types {

Spec m\_SpecType;

const char\* m\_pName;

Types() {

m\_SpecType = Spec::NONE;

m\_pName = 0;

}

Types(Spec T, const char\* pName) {

m\_SpecType = T;

m\_pName = pName;

}

static const char\* FindName(Spec T);

};

static inline Types TypesLUT[] = {

{Spec::NONE,"NONE"},

{Spec::BYTE,"BYTE"},

{Spec::CHAR,"CHAR"},

{Spec::INT,"INT"},

{Spec::CARD,"CARD"},

{Spec::BOOL,"BOOL"},

{Spec::TYPEDEF,"TYPEDEF"},

{Spec::ARRAY,"ARRAY"},

{Spec::POINTER,"POINTER"},

{Spec::PROC,"PROC"},

{Spec::FUNC,"FUNC"},

{Spec(-1),0}

};

Spec m\_SpecType;

CObjType\* m\_pNext;

CObjType\* m\_pPrev;

public:

CObjType();

virtual ~CObjType();

bool Create();

const char\* FindName(Spec T) {

return Types::FindName(T);

}

CObjType\* GetNext() { return m\_pNext; }

void SetNext(CObjType\* pN) { m\_pNext = pN; }

CObjType\* GetPrev() { return m\_pPrev; }

void SetPrev(CObjType\* pP) { m\_pPrev = pP; }

};

---------- Parser.h -------------

#pragma once

//-------------------------------------

// Allowed Addressing Modes

//-------------------------------------

constexpr auto ADR\_MODE\_ABSOLUTE = 0x0001;

constexpr auto ADR\_MODE\_ABSOLUTE\_X = 0x0002;

constexpr auto ADR\_MODE\_ABSOLUTE\_Y = 0x0041;

constexpr auto ADR\_MODE\_ZEROPAGE = 0x0008;

constexpr auto ADR\_MODE\_ZEROPAGE\_X = 0x0010;

constexpr auto ADR\_MODE\_ZEROPAGE\_Y = 0x0020;

constexpr auto ADR\_MODE\_IMMEDIATE = 0x0040;

constexpr auto ADR\_MODE\_INDIRECT\_Y = 0x0080;

constexpr auto ADR\_MODE\_INDIRECT\_X = 0x0100;

constexpr auto ADR\_MODE\_INDIRECT = 0x0200;

constexpr auto ADR\_MODE\_RELATIVE = 0x0400;

constexpr auto ADR\_MODE\_ACCUMULATOR = 0x0800;

constexpr auto ADR\_MODE\_IMPLIED = 0x1000;

class CParser

{

public:

enum class PHASE {

NONE,

GENERATE\_AST,

EXECUTE\_AST\_PASS1,

EXECUTE\_AST\_PASS2,

GENTERATE\_OUT\_FILES

};

private:

struct PHASE\_LUT {

PHASE m\_Phase;

const char\* m\_pName;

const char\* LookupPhaseName(PHASE phase);

PHASE LookupPhaseToken(const char\* pName);

};

static inline PHASE\_LUT ParsePhase[4] = {

{PHASE::GENERATE\_AST,"GENERATE\_AST"},

{PHASE::EXECUTE\_AST\_PASS1,"EXECUTE\_AST\_PASS1"},

{PHASE::EXECUTE\_AST\_PASS2,"EXECUTE\_AST\_PASS2"},

{PHASE::GENTERATE\_OUT\_FILES,"GENTERATE\_OUT\_FILES"},

};

CLexer\* m\_pLex;

PHASE m\_Phase; // Phase of the compiler operation

CSection\* m\_pCurrentSection;

Processor m\_Processor;

int m\_Recursion;

int m\_Bump;

CActionAstTree m\_AstTree;

CStack m\_ValueStack;

public:

CParser();

virtual ~CParser();

bool Create();

CStack\* GetValueStack() { return &m\_ValueStack; }

void SetCurrentSection(CSection\* pSection) {

m\_pCurrentSection = m\_pCurrentSection;

}

CSection\* GetCurrentSection() {

return m\_pCurrentSection;

}

CLexer\* GetLexer() { return m\_pLex; }

CActionAstTree\* GetAstTree() { return &m\_AstTree; }

FILE\* LogFile();

CLHead Run();

//---------------------------------

Token Expect(Token LookaheadToken, Token Expected);

bool Accept(Token LookaheadToken, Token Expected);

private:

void PrepareInstruction(CInstruction\*\* ppInst, Token Op);

//---------------------------------

// Parsing Methods

//---------------------------------

// Program Structure

//---------------------------------

CLHead Action65(CLHead LookaHead);

CLHead Modules(CLHead LookaHead);

CLHead Vector(CLHead LookaHead);

CLHead VectorAddress(CLHead LookaHead);

//------------------------------------------

// Statements

//------------------------------------------

CLHead Statements(CLHead LookaHead);

CLHead ProcParams(CLHead LookaHead);

//----FOR

CLHead ForStmt(CLHead LookaHead);

CLHead STEPoption(CLHead LookaHead);

CLHead ForTO(CLHead LookaHead);

CLHead Iterator(CLHead LookaHead);

//--- IF

CLHead IfStmt(CLHead LookaHead);

CLHead EndIf(CLHead LookaHead);

CLHead ElsePart(CLHead LookaHead);

CLHead ElseIfPart(CLHead LookaHead);

CLHead ThenPart(CLHead LookaHead);

//----IFF

CLHead IffStmt(CLHead LookaHead);

CLHead IFFend(CLHead LookaHead);;

CLHead IFFelse(CLHead LookaHead);;

CLHead IFFthenpart(CLHead LookaHead);;

CLHead IffConditional(CLHead LookaHead);;

CLHead RelOper(CLHead LookaHead);;

CLHead StatusFlags(CLHead LookaHead);;

CLHead OptNot(CLHead LookaHead);;

CLHead BitValue(CLHead LookaHead);;

CLHead Bits(CLHead LookaHead);;

//----WHILE

CLHead WhileStmt(CLHead LookaHead);

//----DO

CLHead DoStmt(CLHead LookaHead);

CLHead DoEND(CLHead LookaHead);

//----EXIT

CLHead EXITstmt(CLHead LookaHead);

//----RETURN

CLHead RetStmt(CLHead LookaHead);

CLHead OptReturnValue(CLHead LookaHead);

//----ASM

CLHead InlineAssembly(CLHead LookaHead);

CLHead InlineAssBlock(CLHead LookaHead);

CLHead EndAsmBlock(CLHead LookaHead);

//----CODE BLOCK

CLHead CodeBlock(CLHead LookaHead);

CLHead CodeBlockEnd(CLHead LookaHead);

//---- UNTIL

CLHead UntillStmt(CLHead LookaHead);

//---- PUSH

CLHead Push(CLHead LookaHead);

CLHead PushSourceList(CLHead LookaHead);

CLHead PushSource(CLHead LookaHead);

//---- POP

CLHead Pop(CLHead LookaHead);

CLHead PopDestList(CLHead LookaHead);

CLHead PopDest(CLHead LookaHead);

//---- BREAK

CLHead Break(CLHead LookaHead);

//---- RTI

CLHead Rti(CLHead LookaHead);

//---- Assogmemt

CLHead Assignment(CLHead LookaHead);

//--------------------------------------

// Logical Expressions

//--------------------------------------

CLHead RelOperation(CLHead LookaHead);

CLHead LogicalOR(CLHead LookaHead);

CLHead LogicalAND(CLHead LookaHead);

//--------------------------------------

// Arithmetic Expressions

//--------------------------------------

CLHead ArithExpr(CLHead LookaHead); //bitwise OR

CLHead BitwiseAND(CLHead LookaHead);

CLHead BitwiseXOR(CLHead LookaHead);

CLHead AddExpr(CLHead LookaHead);

CLHead ShifExpr(CLHead LookaHead);

CLHead MultExpr(CLHead LookaHead);

CLHead Unary(CLHead LookaHead);

CLHead Factor(CLHead LookaHead);

//-------------------------------------------

// Declarations

//-------------------------------------------

CLHead SysDecl(CLHead LookaHead);

CLHead DefList(CLHead LookaHead);

CLHead Def(CLHead LookaHead);

//--------------------------------------

// TYPEdef Definition

//--------------------------------------

CLHead TypeDefDecl(CLHead LookaHead);

CLHead EndTypeDef(CLHead LookaHead);

CLHead RecDefField(CLHead LookaHead);

//----------------------------------------

// Fundamental Declarations

//----------------------------------------

CLHead FundDecl(CLHead LookaHead);

CLHead FundPointerDecl(CLHead LookaHead);

CLHead FundArrayDecl(CLHead LookaHead);

CLHead FunctionDecl(CLHead LookaHead);

CLHead IrqDef(CLHead LookaHead);

CLHead ProcDef(CLHead LookaHead);

CLHead FuncDef(CLHead LookaHead);

CLHead IrqDecl(CLHead LookaHead);

CLHead ProcDecl(CLHead LookaHead);

CLHead FuncDecl(CLHead LookaHead);

CLHead IrqBody(CLHead LookaHead);

CLHead ProcBody(CLHead LookaHead);

CLHead FuncBody(CLHead LookaHead);

CLHead OptInit(CLHead LookaHead);

//----------------------------------

// Identifiers

//----------------------------------

CLHead IdentList(CLHead LookaHead);

CLHead Ident(CLHead LookaHead);

CLHead IdentInit(CLHead LookaHead);

CLHead Address(CLHead LookaHead);

CLHead Data(CLHead LookaHead);

CLHead ArrayDim(CLHead LookaHead);

//-------------------------------------------

// Parameter Declarations

//-------------------------------------------

CLHead ParamList(CLHead LookaHead);

CLHead PramPointer(CLHead LookaHead);

CLHead ParamArray(CLHead LookaHead);

//-----------------------------------------------

// Local Variableas

//-----------------------------------------------

CLHead LocalDecls(CLHead LookaHead);

CLHead LocalPointerDecl(CLHead LookaHead);

CLHead LocalArrayDecl(CLHead LookaHead);

//-------------------------------

// Compiler Constants

//-------------------------------

CLHead ConstList(CLHead LookaHead);

CLHead CompConst(CLHead LookaHead);

CLHead BaseCompConst(CLHead LookaHead);

//----------------------------------

//Variable References

//Memory References

//----------------------------------

CLHead ValueList(CLHead LookaHead);

CLHead Value(CLHead LookaHead);

CLHead AddressOf(CLHead LookaHead);

CLHead MemContentsType(CLHead LookaHead);

CLHead MemContents(CLHead LookaHead);

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//----------------------------------------

// Inline assembly code

//----------------------------------------

// Statements

//----------------------------------------

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

CLHead AsmStmt(CLHead LookaHead);

CLHead Processor\_1(CLHead LookaHead);

CLHead ProcessorType(CLHead LookaHead);

//-----------------------------------------

// Code

// These statements are what actually

// do something that concerns the machine

// code that is executed

//-----------------------------------------

//-----------------------------------------

// SECITON

//-----------------------------------------

CLHead Section(CLHead LookaHead);

CLHead SectionName(CLHead LookaHead);

CLHead SectionDef(CLHead LookaHead);

CLHead SectionAttributesList(CLHead LookaHead);

CLHead SectionAtribute(CLHead LookaHead);

CLHead Modes(CLHead LookaHead);

CLHead TrueFalse(CLHead LookaHead);

//-------------------------------------

// Org Sets the location counter

// for the current section

//-------------------------------------

CLHead Org(CLHead LookaHead);

//-------------------------------------

// Define Memeory

//-------------------------------------

CLHead DefineMemory(CLHead LookaHead);

//-------------------------------------

// DefineStorage

//-------------------------------------

CLHead DefineStorage(CLHead LookaHead);

//-------------------------------------

// Proceedure

//-------------------------------------

CLHead Proceedure(CLHead LookaHead);

//--------------------------------------

// Opcodes

//--------------------------------------

CLHead Instruction(CLHead LookaHead);

//---------------------------------------------

// Lables

//---------------------------------------------

CLHead Labels(CLHead LookaHead);

CLHead LocalGlobal(CLHead LookaHead);

//-----------------------------------

//ALU Addressing Mode

//-----------------------------------

CLHead AluAdrModes(CLHead LookaHead);

CLHead Indirect(CLHead LookaHead);

//---------------------------------------------

// STA addressing mode

//---------------------------------------------

CLHead StaAddressingModes(CLHead LookaHead);

//-----------------------------------------

// ASL LSR ROR and ROL addressing modes

//-----------------------------------------

CLHead ShiftAddressingModes(CLHead LookaHead);

//------------------------------------------

// Branch Instructions Addressing Mode

//------------------------------------------

CLHead RelAddressingMode(CLHead LookaHead);

//------------------------------------------

// BIT Instructions Addressing Mode

//------------------------------------------

CLHead BitAddressModes(CLHead LookaHead);

//------------------------------------------

// INC DEC Instructions Addressing Mode

//------------------------------------------

CLHead IncAddressingMOdes(CLHead LookaHead);

//------------------------------------------

// JMP Instructions Addressing Mode

//------------------------------------------

CLHead JumpAddressingModes(CLHead LookaHead);

//------------------------------------------

// JSR Instructions Addressing Mode

//------------------------------------------

CLHead CallAddressingMode(CLHead LookaHead);

//------------------------------------------

// LDX Instructions Addressing Mode

//------------------------------------------

CLHead LdxAddressingMode(CLHead LookaHead);

//------------------------------------------

// CPX & CPY Instructions Addressing Mode

//------------------------------------------

CLHead CPX\_CPY\_AddressingMode(CLHead LookaHead);

//------------------------------------------

// STX Instructions Addressing Mode

//------------------------------------------

CLHead StxAddressingMode(CLHead LookaHead);

//------------------------------------------

// LDY Instructions Addressing Mode

//------------------------------------------

CLHead LdyAddressingMode(CLHead LookaHead);

//------------------------------------------

// STY Instructions Addressing Mode

//------------------------------------------

CLHead StyAddressingMode(CLHead LookaHead);

//---------------------------------------------

// Optional Index Registers

//---------------------------------------------

CLHead OptIndexReg(CLHead LookaHead);

CLHead OptIndexReg\_1(CLHead LookaHead);

CLHead OptXReg(CLHead LookaHead);

CLHead OptYReg(CLHead LookaHead);

//---------------------------------------

// Assembler Constants

//---------------------------------------

CLHead AsmConstList(CLHead LookaHead);

CLHead AsmConstList\_1(CLHead LookaHead);

CLHead AsmConstant(CLHead LookaHead);

CLHead AsmConstAddSub(CLHead LookaHead);

CLHead BaseAsmConstant(CLHead LookaHead);

CLHead Immediate(CLHead LookaHead, CInstruction\* pInst);

void PageZero(CInstruction\* pInst, int Address, AdrModeType ModeType);

void Absolute(CInstruction\* pInst, int Address, AdrModeType ModeType);

bool CheckZeroPageAddress(int A);

//---------------- Utillity ----------------------

void PrintLookahead(

FILE\* pLog,

CLHead token,

const char\* pS,

int RecursionLevel,

int Bump = 0

);

//---------------------------------

};

---------- pch.h -------------

#ifndef PCH\_\_H

#define PCH\_\_H

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#include <sys\stat.h>

#include <sys\types.h>

constexpr auto MAX\_SYMBOL\_NAME\_LEN = 256;

constexpr auto MAX\_NAME\_LEN = 64;

constexpr auto MAX\_STRING\_LEN = 512;

enum class Token {

ENDOFFILE = -1, // 1

IDENT = 256, //2

STRING, //3

NUMBER, //4

//------ Assignment

ASSIGN\_ADD, //5

ASSIGN\_SUB, //6

ASSIGN\_MUL, //7

ASSIGN\_DIV, //8

ASSIGN\_MOD, //9

ASSIGN\_AND, //10

ASSIGN\_OR,

ASSIGN\_XOR,

ASSIGN\_RSH,

ASSIGN\_LSH,

//------- Operators

MOD,

LSH,

RSH,

GTEQ,

LTEQ,

OR, //logical OR

XOR, //bitwise Exclusive OR

//-------- Functions --------

PROC,

FUNC,

INTERRUPT,

INTERRUPT\_IDENT,

FUNC\_IDENT,

PROC\_IDENT,

//------ Data Types -----------------

BOOL,

BYTE,

CHAR,

INT,

CARD,

POINTER,

TYPE,

RECORDTYPE,

ARRAY,

//--------Statements----------

MODULE,

VECTOR,

FOR,

TO,

STEP,

IF,

IFF,

THEN,

ELSEIF,

ELSE,

FI,

FFI,

WHILE,

DO,

OD,

UNTIL,

EXIT,

RETURN,

ASM,

PUSH,

POP,

BREAK,

BITS,

BEGIN,

END,

// Compiler Directives

DEFINE,

SET,

INCLUDE,

//---------- Assembler Tokens ------------

//--------- Section --------

SECTION,

SECTION\_NAME,

START,

SIZE,

NAME,

MODE,

READ\_WRTE,

READ\_ONLY,

ZEROPAGE,

True,

False,

//----- Assembler declarators

ORG,

DB,

DW,

DL,

DAS, //define action! string

DCS, //define C String

DS, //define storage

//------- Opcodes

ADC,

AND, //Logical Action AND or ASM Opcode

ASL,

BCC,

BCS,

BEQ,

BMI,

BNE,

BPL,

BVC,

BVS,

BIT,

BRK,

CLC,

CLD,

CLI,

CLV,

CMP,

CPX,

CPY,

DEC,

DEX,

DEY,

EOR,

INC,

INX,

INY,

JMP,

JSR,

LDA,

LDX,

LDY,

LSR,

NOP,

ORA,

PHA,

PLA,

PHP,

PLP,

ROL,

ROR,

RTI,

RTS,

SBC,

STA,

SEC,

SED,

SEI,

STX,

STY,

TAX,

TAY,

TXA,

TYA,

TXS,

TSX,

ENDOFOPCODES,

//----- Registers

AREG, //accumulator

XREG,

YREG,

SPREG, //stack pointer

PSREG, //processor status register

//---------- Lables

LOCAL\_LABEL,

GLOBAL\_LABLE,

//-------- Status Register bits

NEG,

ZERO,

CARRY,

OVERFLOW,

IRQENABLE,

DECIMAL\_MODE,

//------- Processor Selector

PROCESSOR,

R6502,

WD65C02,

WD65C816,

//------- Misc

CHAR\_CONSTANT,

EPROC, // proceedure end

EOL,

//------------------------

ENDOFTOKENS = 0

};

enum class IdentType {

NEW\_SYMBOL,

LABEL,

PROC,

FUNC,

FUNC\_BYTE,

FUNC\_CHAR,

FUNC\_CARD,

FUNC\_INT,

FUNC\_POINTER\_BYTE,

FUNC\_POINTER\_CHAR,

FUNC\_POINTER\_CARD,

FUNC\_POINTER\_INT,

MEM\_GLOBAL\_BYTE,

MEM\_GLOBAL\_CHAR,

MEM\_GLOBAL\_CARD,

MEM\_GLOBAL\_INT,

MEM\_GLOBAL\_POINTER\_BYTE,

MEM\_GLOBAL\_POINTER\_CHAR,

MEM\_GLOBAL\_POINTER\_CARD,

MEM\_GLOBAL\_POINTER\_INT,

MEM\_PARAM\_BYTE,

MEM\_PARAM\_CHAR,

MEM\_PARAM\_CARD,

MEM\_PARAM\_INT,

MEM\_PARAM\_POINTER\_BYTE,

MEM\_PARAM\_POINTER\_CHAR,

MEM\_PARAM\_POINTER\_CARD,

MEM\_PARAM\_POINTER\_INT,

MEM\_LOCAL\_BYTE,

MEM\_LOCAL\_CHAR,

MEM\_LOCAL\_CARD,

MEM\_LOCAL\_INT,

MEM\_LOCAL\_POINTER\_BYTE,

MEM\_LOCAL\_POINTER\_CHAR,

MEM\_LOCAL\_POINTER\_CARD,

MEM\_LOCAL\_POINTER\_INT

};

enum class AdrModeType {

NA,

IMPLIED,

INDIRECT\_X\_ADR,

ZERO\_PAGE\_ADR,

IMMEDIATE\_ADR,

ABSOLUTE\_ADR,

INDIRECT\_Y\_ADR,

ZERO\_PAGE\_X\_ADR,

ABSOLUTE\_Y\_ADR,

ABSOLUTE\_X\_ADR,

ZERO\_PAGE\_Y\_ADR,

INDIRECT\_ADR,

ACCUMULATOR

};

enum class AddressModesClass { //ToDo: Delete?

IMPLIED, //0

ONE\_MODE, //1

TWO\_MODES, //2

THREE\_MODES, //3

FOUR\_MODES, //4

FIVE\_MODES, //5

SIX\_MODES, //6

SEVEN\_MODES, //7

EIGHT\_MODES //8

};

enum class Processor {

ALL,

R6502,

WD65C02,

WD65C816

};

#include "StackItem.h"

#include "DataSizeStackItem.h"

#include "RegisterStackItem.h"

#include "Instruction.h"

#include "StackSectionItem.h"

#include "NumberStackItem.h"

#include "ObjType.h"

#include "Bin.h"

#include "WhereSymbolIsUsed.h"

#include "UnResolvedSymbol.h"

#include "Bucket.h"

#include "Symbol.h"

#include "Value.h"

constexpr auto MAX\_EXCEPTION\_STRING\_LEN = 512;

class Exception {

public:

enum class ExceptionType {

WHOKNOWS,

UNEXPECTED\_TOKEN,

SECTION\_ADDRES\_RANGE\_EXCEEDED,

SECTION\_UNDEFINED,

NOSECTION\_DEFINED,

LEXER\_STUMPTED,

ILLEGAL\_ADDRESSING\_MODE,

TOKEN\_OUT\_OF\_PLACE,

VALUE\_EXCEEDS\_RANGE,

STACK,

INTERNAL\_ERROR

};

enum class ExceptionSubType {

WHOKNOWS,

//--------------------

// Stack Exceptions Subtypes

//--------------------

STACK\_UNEXPECTED\_NULL,

STACK\_EMPTY,

STACK\_ITEM\_MISMATCH

};

private:

ExceptionType m\_Type;

ExceptionSubType m\_SubType;

const char\* m\_Name;

Token badToken;

CBin\* m\_pSymbol;

char ErrorString[MAX\_EXCEPTION\_STRING\_LEN];

inline static int MaxStringLen = MAX\_EXCEPTION\_STRING\_LEN;

//------------------------------------------------

struct ExceptionSubTypeStrings {

ExceptionSubType m\_Type;

const char\* m\_Name;

ExceptionSubTypeStrings() {

m\_Type = ExceptionSubType::WHOKNOWS;

m\_Name = 0;

}

ExceptionSubTypeStrings(ExceptionSubType Type, const char\* pName) {

m\_Type = Type;

m\_Name = pName;

}

const char\* FindSubTypeString(ExceptionSubType Type);

};

//------------------------------------------------

struct ExcepTypeToString {

ExceptionType m\_Type;

const char\* m\_Name;

ExcepTypeToString() {

m\_Type = ExceptionType::WHOKNOWS;

m\_Name = 0;

}

ExcepTypeToString(ExceptionType Type, const char\* pName) {

m\_Type = Type;

m\_Name = pName;

}

const char\* FindString(ExceptionType Type);

};

//-------------------------------------------------------

inline static ExcepTypeToString ExceptionTypesLUT[] = {

{ExceptionType:: WHOKNOWS, "WHO KNOWS, Whatever" },

{ ExceptionType::UNEXPECTED\_TOKEN, "UNEXPECTED TOKEN" },

{ ExceptionType::SECTION\_ADDRES\_RANGE\_EXCEEDED, "SECTION ADDRES RANGE EXCEEDED" },

{ ExceptionType::SECTION\_UNDEFINED, "SECTION UNDEFINED" },

{ ExceptionType::NOSECTION\_DEFINED, "NO SECTION DEFINED" },

{ExceptionType::LEXER\_STUMPTED, "LEXER is STUMPTED!" },

{ ExceptionType::ILLEGAL\_ADDRESSING\_MODE, "ILLEGAL ADDRESSING MODE" },

{ ExceptionType::TOKEN\_OUT\_OF\_PLACE, "TOKEN OUT OF PLACE" },

{ ExceptionType::VALUE\_EXCEEDS\_RANGE, "VALUE EXCEEDS RANGE" },

{ ExceptionType::INTERNAL\_ERROR, "This is too much, INTERNAL ERROR!" },

{ ExceptionType(-1), NULL}

};

//--------------------------------------------------

inline static ExceptionSubTypeStrings XcepSubTypesLUT[] =

{

{ExceptionSubType::WHOKNOWS,"Who Knows, Not me!"},

{ExceptionSubType::STACK\_UNEXPECTED\_NULL,"Top of Stack is NULL"},

{ExceptionSubType::STACK\_EMPTY,"Stack is Empty"},

{ExceptionSubType::STACK\_ITEM\_MISMATCH,"Mismatch"},

{ExceptionSubType(-1),0}

};

public:

Exception() {

int i;

m\_Name = 0;

m\_Type = ExceptionType::WHOKNOWS;

m\_SubType = ExceptionSubType::WHOKNOWS;

badToken = Token(0);

m\_pSymbol = 0;

for (i = 0; i < MAX\_EXCEPTION\_STRING\_LEN; ++i)

ErrorString[i] = 0;

}

const char\* GetExceptionTypeString(ExceptionType xType);

ExceptionType GetXCeptType() { return m\_Type; }

void SetXCeptType(ExceptionType Type) { m\_Type = Type; }

Token GetGotToken() { return badToken; }

void SetGotToken(Token t) { badToken = t; }

char\* GetErrorString() { return ErrorString; }

int GetMaxStringLen() {

return MaxStringLen;

}

void SetSymbol(CBin\* pSym) { m\_pSymbol = pSym; }

CBin\* GetSymbol() { return m\_pSymbol; }

const char\* FindXceptnSubType(ExceptionSubType Type) {

return XcepSubTypesLUT->FindSubTypeString(Type);

}

const char\* FindXceptnType(ExceptionType Type) {

return ExceptionTypesLUT->FindString(Type);

}

};

#include "BinStackItem.h"

#include "StackSymbolItem.h"

#include "Stack.h"

#include "SymTab.h"

//--------------------------------------

// Object File Format

//--------------------------------------

#include "ObjFormatSectionItem.h"

#include "ObjFormatSymbolItem.h"

#include "ObjFormatSymbols.h"

#include "ObjFormatCode.h"

#include "ObjFormatData.h"

#include "ObjFormatRelocation.h"

#include "ObjFormatSection.h"

#include "ObjFormatFile.h"

//--------------------------------------

#include "Section.h"

//---------------------------------------

//----------------Value Stack -----------

//---------------------------------------

#include "Stack.h"

#include "StackItem.h"

#include "StackSectionItem.h"

#include "StackSymbolItem.h"

#include "AddressSizeStackItem.h"

#include "AccessModeStackItem.h"

//-------------- AST Base Class ----------

#include "AstNode.h"

//----------------------------------------

//------------- AST Node Classes ---------

//----------------------------------------

//------------ Program Structure --------------

#include "Act65Module.h"

#include "Act65VECTOR.h"

#include "Act65INTERRUPT.h"

#include "Act65PROC.h"

//----------- Declaration AST Nodes -----------

//----- Types ------

#include "Act65BYTE.h"

#include "Act65CHAR.h"

#include "Act65CARD.h"

#include "Act65BOOL.h"

#include "Act65INT.h"

#include "Act65RECTYPE.h"

#include "Act65POINTER.h"

//----- Modifiers --------------

#include "Act65ARRAY.h"

#include "Act65DEFINE.h"

#include "Act65TYPEDEF.h"

//-----------------------------------------

//---- Logical/Arithmetic AST Nodes -------

//-----------------------------------------

#include "Act65GreaterTHAN.h"

#include "Act65LessTHAN.h"

#include "Act65GTEQ.h"

#include "Act65LTEQ.h"

#include "Act65EqualTO.h"

#include "Act65NotEquelTO.h"

#include "Act65LogicalOR.h"

#include "Act65LogicalAND.h"

#include "Act65BitWiseOR.h"

#include "Act65BitWiseAND.h"

#include "Act65XOR.h"

#include "Act65ADD.h"

#include "Act65SUB.h"

#include "Act65LSH.h"

#include "Act65RSH.h"

#include "Act65ROR.h"

#include "Act65ROL.h"

//---------------------------------------------

//----------------Statement AST Nodes ---------

//---------------------------------------------

#include "Act65ProcCall.h"

#include "Act65FuncCall.h"

#include "Act65ProcParams.h"

#include "Act65FOR.h"

#include "Act65ForSTEP.h"

#include "Act65ForTO.h"

#include "Act65ForItterator.h"

#include "Act65IF.h"

#include "Act65FI.h"

#include "Act65THEN.h"

#include "Act65ELSEIF.h"

#include "Act65ELSE.h"

#include "Act65IFF.h"

#include "Act65BEGIN.h"

#include "Act65END.h"

#include "Act65FFI.h"

#include "Act65FlagNEG.h"

#include "Act65FlagCARRY.h"

#include "Act65FlagZERO.h"

#include "Act65FlagOVERFLOW.h"

#include "Act65OptNOT.h"

#include "Act65BIT.h"

#include "Act65WHILE.h"

#include "Act65DO.h"

#include "Act65OD.h"

#include "Act65EXIT.h"

#include "Act65RETURN.h"

#include "Act65ASM.h"

#include "Act65ASMstatement.h"

#include "Act65CodeBlock.h"

#include "Act65UNTILL.h"

#include "Act65PUSH.h"

#include "Act65PushSource.h"

#include "Act65POP.h"

#include "Act65PopDest.h"

#include "Act65RTI.h"

#include "Act65BREAK.h"

//--------- Assignment Node Classes ------

#include "Act65Assignment.h"

#include "Act65AssignADD.h"

#include "Act65AssignSUB.h"

#include "Act65AssignMULT.h"

#include "Act65AssignDIV.h"

#include "Act65AssignMOD.h"

#include "Act65AssignOR.h"

#include "Act65AssignAND.h"

#include "Act65AssignXOR.h"

#include "Act65AssignLSH.h"

#include "Act65AssignRSH.h"

//------------------------------------

// Assembler AST Nodes

//------------------------------------

#include "Act65PROCESSOR.h"

#include "Act65SECTION.h"

#include "Act65SecAtrbSTART.h"

#include "Act65SecAtrbSIZE.h"

#include "Act65SecAtrbMODE.h"

#include "Act65SecAtrbZEROPAGE.h"

#include "Act65SecAtrbREADONLY.h"

#include "Act65SecAtrbREADWRITE.h"

#include "Act65TRUE.h"

#include "Act65SecAtrbFALSE.h"

#include "Act65ORG.h"

#include "Act65DB.h"

#include "Act65DW.h"

#include "Act65DL.h"

#include "Act65DAS.h"

#include "Act65DCS.h"

#include "Act65DS.h"

#include "Act65ASMPROC.h"

#include "Act65EPROC.h"

#include "Act65Opcode.h"

#include "Act65Label.h"

#include "Act65Private.h"

#include "Act65XREG.h"

#include "Act65YREG.h"

#include "Act65ACC.h"

#include "Act65CharCibstabt.h"

#include "Act65LowerPart.h"

#include "Act65UpperPart.h"

#include "Act65CurrentLocation.h"

#include "ActStatusFlags.h"

//------------------------------------

#include "Act65PROCname.h"

#include "Act65IDENT.h"

//------------------------------------

#include "AstTree.h"

#include "ActionAstTree.h"

#include "LHead.h"

#include "Lexer.h"

#include "Parser.h"

#include "ActionApp.h"

extern Exception ThrownException;

#endif // ! PCH\_\_H

---------- RegisterStackItem.h -------------

#pragma once

class CRegisterStackItem : public CStackItem

{

public:

enum RegType {

NONE,

X,

Y

};

private:

RegType m\_RegType;

public:

CRegisterStackItem();

virtual ~CRegisterStackItem();

bool Create();

RegType GetRegType() { return m\_RegType; }

void SetRegType(RegType t) { m\_RegType = t; }

};

---------- Section.h -------------

#pragma once

class CSection : public CBin

{

public:

enum class Mode {

MODE\_READ\_ONLY,

MODE\_READ\_WRITE

};

enum class AddressSize {

ADDRESSSIZE\_WORD,

ADDRESSSIZE\_ZEROPAGE

};

private:

unsigned m\_StartAddress; //where section starts in memory

unsigned m\_Size; //max size of the memory block

unsigned m\_LocationCounter; //current location to store data

char\* m\_pSectionData; //where the data is

Mode m\_AccessMode; //read only or read write

AddressSize m\_ZeroPageAddressSize;

public:

CSection();

virtual ~CSection();

bool Create();

unsigned GetStartAddress() { return m\_StartAddress; }

void SetStartAddress(unsigned SA);

unsigned GetSectionSize() { return m\_Size; }

void SetSectionSize(unsigned S);

unsigned short GetLocationCounter() { return m\_LocationCounter; }

unsigned short IncrementLocationCounterBy(unsigned short v) {

m\_LocationCounter += v;

return m\_LocationCounter;

}

void SetLocationCounter(unsigned short NewAddress);

unsigned short AddData(unsigned ObjectSize, int Value);

void AddDataAt(unsigned Adr, unsigned ObjectSize, int valu);

unsigned short AllocateDataBlock(unsigned short size); //define Storage

void SetAccessMode(Mode Am) { m\_AccessMode = Am; }

CSection::Mode GetAccessMode() { return m\_AccessMode; }

void SetZeroPageFlag(AddressSize SizeFlag) { m\_ZeroPageAddressSize = SizeFlag; }

CSection::AddressSize GetZeroPageFlag() { return m\_ZeroPageAddressSize; }

virtual void Print(FILE\* pOut, const char\* s = 0);

void Info();

};

---------- Stack.h -------------

#pragma once

class CStack

{

static inline char m\_sExceptionString[MAX\_STRING\_LEN];

const char\* m\_pName;

CStackItem\* m\_pHead;

int m\_ItemCount;

public:

CStack();

virtual ~CStack();

bool Create(const char\* pName);

virtual void Push(CStackItem\* pItem);

virtual CStackItem\* Pop(CStackItem::ItemType Type);

virtual CStackItem\* Look(int Depth, CStackItem::ItemType Type);

virtual CStackItem\* GetHead() { return m\_pHead; }

virtual void SetHead(CStackItem\* pSI) { m\_pHead = pSI; }

virtual bool IsTopOfType(CStackItem::ItemType Type);

virtual void Print(

FILE\* pLog,

const char\* pName,

int Indent = 0,

CStackItem\* pRV = 0

);

void StackException(const char\* pDesc, Exception::ExceptionSubType SubType);

virtual const char\* GetStackName() { return m\_pName; }

int GetItemCount() { return m\_ItemCount; }

};

---------- StackItem.h -------------

#pragma once

class CStackItem

{

public:

enum class ItemType {

NONE,

SYMBOL,

SECTION,

INTVALUE,

INSTRUCTION,

REGTYPE,

ADDRESSSIZE,

ACCESS\_MODE,

DATA\_SIZE,

BIN,

NODE

};

struct ItemTypeLUT {

ItemType m\_Type;

const char\* m\_pName;

ItemTypeLUT() {

m\_Type = ItemType::NONE;

m\_pName = 0;

}

ItemTypeLUT(ItemType T, const char\* s) {

m\_Type = T;

m\_pName = s;

}

static const char\* FindItemName(CStackItem::ItemType Type);

};

private:

inline static ItemTypeLUT TypeString[] = {

{ItemType::NONE,"NONE"},

{ItemType::SYMBOL,"SYMBOL"},

{ItemType::SECTION,"SECTION"},

{ItemType::INTVALUE,"INT VALUE"},

{ItemType::INSTRUCTION,"INSTRUCTION"},

{ItemType::REGTYPE,"REGTYPE"},

{ItemType::ADDRESSSIZE,"ADDRESS SIZE"},

{ItemType::ACCESS\_MODE,"ACCESS MODE"},

{ItemType::DATA\_SIZE,"DATA SIZE"},

{ItemType::BIN,"BIN"},

{ItemType::NODE,"NODE"},

{ItemType(-1),NULL}

};

ItemType m\_Type;

CStackItem\* m\_pNext;

public:

CStackItem();

virtual ~CStackItem();

bool Create(ItemType Type);

void SetNext(CStackItem\* pSI) { m\_pNext = pSI; }

CStackItem\* GetNext() { return m\_pNext; }

ItemType GetStackItemType() { return m\_Type; }

bool Verify(CStackItem::ItemType Type);

static const char\* LookupItemName(CStackItem::ItemType Type);

};

---------- StackNodeItem.h -------------

#pragma once

class CAstNode;

class CStackNodeItem : public CStackItem

{

CAstNode\* m\_pAstNode;

public:

CStackNodeItem();

virtual ~CStackNodeItem();

bool Create(CAstNode\* pNode);

void SetNode(CAstNode\* pNode) { m\_pAstNode = pNode; }

CAstNode\* GetNode() { return m\_pAstNode; }

};

---------- StackSectionItem.h -------------

#pragma once

class CSection;

class CStackSectionItem : public CStackItem

{

CSection\* m\_pSection;

public:

CStackSectionItem();

~CStackSectionItem();

bool Create(CSection\* pSect);

void SetSection(CSection\* pSect) { m\_pSection = pSect; }

CSection\* GetSection() { return m\_pSection; }

};

---------- StackSymbolItem.h -------------

#pragma once

class CSymbol;

class CStackSymbolItem : public CStackItem

{

CSymbol\* m\_pSymbol;

public:

CStackSymbolItem();

~CStackSymbolItem();

bool Create(CSymbol\* pSym);

void SetSymbol(CSymbol\* pSym) { m\_pSymbol = pSym; }

CSymbol\* GetSymbol() { return m\_pSymbol; }

};

---------- Symbol.h -------------

#pragma once

constexpr auto SYMBOL\_SCOPE\_ANY = 0;

constexpr auto SYMBOL\_SCOPE\_GLOBAL = 1;

constexpr auto SYMBOL\_SCOPE\_LOCAKL = 2;

class CSection;

class CSymbol: public CBin

{

unsigned m\_Address;

int m\_Value;

int m\_Scope;

CSection\* m\_pSection;

IdentType m\_IdentType;

bool m\_UnResolved;

//---------------------------

// Type Chain

//---------------------------

CObjType\* m\_pTcHead;

CObjType\* m\_pTcTail;

public:

CSymbol() {

m\_Address = 0;

m\_Value = 0;

m\_Scope = SYMBOL\_SCOPE\_ANY;

m\_pSection = 0;

m\_IdentType = IdentType::NEW\_SYMBOL;

m\_UnResolved = true;

m\_pTcHead = 0;

m\_pTcTail = 0;

}

virtual ~CSymbol() {}

bool Create() { return true; }

virtual bool Compare(const char\* name, int scope);

virtual void Print(FILE\* pOut, const char\* s);

//-----------------------------

// Accessor Methods

//-----------------------------

unsigned GetAddress() { return m\_Address; }

void SetAddress(unsigned A) { m\_Address = A; }

int GetValue() { return m\_Value; }

void SetValue(int v) { m\_Value = v; }

int GetScope() { return m\_Scope; }

void SetScope(int S) { m\_Scope = S; }

CSection\* GetSection() { return m\_pSection; }

void SetSection(CSection\* pS) { m\_pSection = pS; }

void SetIdentType(IdentType IT) { m\_IdentType = IT; }

IdentType GetIdentType() { return m\_IdentType; }

bool IsUnResolved() {

return m\_UnResolved;

}

bool IsResolved() {

return !m\_UnResolved;

}

void SetResolved() { m\_UnResolved = false; }

void SetUnResolved() { m\_UnResolved = true; }

//-------------------------------------------------

// Backfill unresolved references

//-------------------------------------------------

void BackFillUnresolved();

//-----------------------------------

// Type chain methods

//-----------------------------------

void AddAtHead(CObjType\* pObjType);

void AddAtTail(CObjType\* pObjType);

void Delete(CObjType\* pObjType);

void Unlink(CObjType\* pObjType);

void SetTcHead(CObjType\* pOT) { m\_pTcHead = pOT; }

CObjType\* GetTcHead() { return m\_pTcHead; }

void SetTcTail(CObjType\* pOT) { m\_pTcTail = pOT; }

CObjType\* GetTcTail() { return m\_pTcTail; }

};

---------- SymTab.h -------------

#pragma once

class CSymTab

{

int Hash(const char\* name);

protected:

int m\_nElements; //total number of elements in table

int m\_tSize; //depth of table

CBucket\*\* m\_ppTab; //pointer to table

int m\_GeneralSymbolCount;

int m\_SectionSymbolCount;

public:

CSymTab() {

m\_ppTab = 0;

m\_nElements = 0;

m\_tSize = 0;

m\_GeneralSymbolCount = 0;

m\_SectionSymbolCount = 0;

}

virtual ~CSymTab();

bool Create(int TableDepth);

int GetSectionCount() { return m\_SectionSymbolCount; }

int GetSymbolCount() { return m\_GeneralSymbolCount; }

virtual CBin\* FindSymbol(const char\* name, int scope);

virtual void AddSymbol(CBin\* pSym);

virtual void DelSymbol(CBin\* pSym);

virtual void PrintTable(FILE\* pOut);

inline int GetNumElements() { return m\_nElements; }

inline int GetTableSize() { return m\_tSize; }

inline CBucket\*\* GetTable() { return m\_ppTab; }

bool FindAll(

CBin::BinType Type,

int NumberOfObject,

CBin\*\* ppObjects

);

};

---------- TreeNodeValue.h -------------

---------- UnResolvedSymbol.h -------------

#pragma once

class CUnResolvedSymbol: public CBin

{

unsigned m\_Address;

CSection\* m\_pSection;

CWhereSymbolIsUsed::UnResolvedType m\_UnResRefType;

public:

CUnResolvedSymbol();

virtual ~CUnResolvedSymbol();

bool Create();

void SetAddress(unsigned a);

};

---------- Value.h -------------

#pragma once

class CValue

{

CBin\* m\_pSym;

public:

CValue();

virtual ~CValue();

bool Create(CBin\* pSym);

void SetSymbol(CBin\* pSym) { m\_pSym = pSym; }

CBin\* GetSymbol() { return m\_pSym; }

};

---------- WhereSymbolIsUsed.h -------------

#pragma once

class CWhereSymbolIsUsed : public CBin

{

public:

enum UnResolvedType {

ABSOLUTE\_REFERENCE,

RELATIVE\_REFERENCE

};

private:

struct UnResolvedSymItem {

UnResolvedType m\_Type;

const char\* m\_pName;

UnResolvedSymItem() {

m\_Type = ABSOLUTE\_REFERENCE;

m\_pName = 0;

}

UnResolvedSymItem(UnResolvedType t, const char\* pS) {

m\_Type = t;

m\_pName = pS;

}

};

inline static UnResolvedSymItem UnResolvedTypeLUTLUT[2] = {

{ABSOLUTE\_REFERENCE,"ABSOLUTE REFERENCE"},

{RELATIVE\_REFERENCE,"RELATIVE REFERENCE"}

};

int m\_Address; //address location where found

CSection\* m\_pSection; // section block where found

UnResolvedType m\_UnResRefType; //type of reference

public:

CWhereSymbolIsUsed();

virtual ~CWhereSymbolIsUsed();

bool Create();

void SetAddress(int a);

int GetAddress() { return m\_Address; }

void SetSection(CSection\* pS) {

m\_pSection = pS;

}

CSection\* GetSection() {

return m\_pSection;

}

void SetUnResType(UnResolvedType type) { m\_UnResRefType = type; }

UnResolvedType GetUnResType() { return m\_UnResRefType; }

};

---------- AccessModeStackItem.cpp -------------

#include "pch.h"

CAccessModeStackItem::CAccessModeStackItem()

{

m\_AccessMode = CSection::Mode::MODE\_READ\_WRITE;

}

CAccessModeStackItem::~CAccessModeStackItem()

{

}

bool CAccessModeStackItem::Create()

{

return CStackItem::Create(CStackItem::ItemType::ACCESS\_MODE);

}

---------- Act65ACC.cpp -------------

#include "pch.h"

CAct65ACC::CAct65ACC()

{

}

CAct65ACC::~CAct65ACC()

{

}

bool CAct65ACC::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild,pNext);

}

CValue\* CAct65ACC::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65ADD.cpp -------------

#include "pch.h"

CAct65ADD::CAct65ADD()

{

}

CAct65ADD::~CAct65ADD()

{

}

bool CAct65ADD::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65ADD::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

/\*

static value\* binary(NODE\* n, int op)

{

NODE\* n1, \* n2; /\* child nodes \*/

// value\* v1, \* v2; /\* values returned from processor nodes \*/

// value\* rV = NULL;

// value\* Lv; //left value

//

// if (Debug) printf("BINOP=%s\n", binop\_strings[op]);

// n1 = n->down;

// n2 = n1->next;

// Lv = (value\*)n->aux;

// if (Lv) fprintf(stderr, "Levft Value is %s\n", Lv->name);

// v1 = (node\_proc[n1->id])(n1); /\* process nodes \*/

// v2 = (node\_proc[n2->id])(n2);

// if (v1 && v2) //good values?

// {

// //printf("VALUE1 %s VALUE2 %s\n",v1->name,v2->name);

// if (do\_binary\_const(&v1, binop\_ops[op], &v2)) /\* was this a const \*/

// {

// release\_value(v2);

// if (Debug) printf("VALUE=%d\n", v1->type->V\_INT);

// rV = v1;

// }

// else

// {

// if ((SizeOfRef(v1->type) != SizeOfRef(v2->type)) && (!IS\_CONSTANT(v1->type) && !IS\_CONSTANT(v2->type)))

// //----------------------------

//OK, these guys are not the

//smae size and they need to

// be in order to combine them

//----------------------------

// if (SizeOfRef(v1->type) > SizeOfRef(v2->type))

// {

//// v2 = ConvertTypeUp(OutFile, v2, v1->type);

// //---------------------------

// // this operation leaves the

//// // in registers if v2 is not

// // a long. so we need to save

// v2 to a temp otherwise

//--------------------------

// }

// else if (SizeOfRef(v2->type) > SizeOfRef(v1->type))

//// {

// v1 = ConvertTypeUp(OutFile, v1, v2->type);

// }

// }

// if (ValInMem(v1) && !ValInMem(v2))

// {

// if (NotRelOp(op)) //we can swap operands

// {

// value\* t;

// t = v1;

// v1 = v2;

// v2 = t; //swap around the args

// }

// else

// {

// v2 = SaveToTemp(OutFile, v2); //put in temp

// }

// }

// else if (!ValInMem(v1) && !ValInMem(v2))

// {

// v2 = SaveToTemp(OutFile, v2); //put V2 in a temp

// }

// rV = DoBinary(OutFile, v1, op, v2, 1, 1, NULL, NULL, Lv);

// release\_value(v2);

// }

// }

// return rV;

//}

---------- Act65ARRAY.cpp -------------

#include "pch.h"

CAct65ARRAY::CAct65ARRAY()

{

}

CAct65ARRAY::~CAct65ARRAY()

{

}

bool CAct65ARRAY::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65ARRAY::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65ASM.cpp -------------

#include "pch.h"

CAct65ASM::CAct65ASM()

{

}

CAct65ASM::~CAct65ASM()

{

}

bool CAct65ASM::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65ASM::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65ASMPROC.cpp -------------

#include "pch.h"

CAct65ASMPROC::CAct65ASMPROC()

{

}

CAct65ASMPROC::~CAct65ASMPROC()

{

}

bool CAct65ASMPROC::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65ASMPROC::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65ASMstatement.cpp -------------

#include "pch.h"

CAct65ASMstatement::CAct65ASMstatement()

{

}

CAct65ASMstatement::~CAct65ASMstatement()

{

}

bool CAct65ASMstatement::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65ASMstatement::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65AssignADD.cpp -------------

#include "pch.h"

CAct65AssignADD::CAct65AssignADD()

{

}

CAct65AssignADD::~CAct65AssignADD()

{

}

bool CAct65AssignADD::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65AssignADD::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65AssignAND.cpp -------------

#include "pch.h"

CAct65AssignAND::CAct65AssignAND()

{

}

CAct65AssignAND::~CAct65AssignAND()

{

}

bool CAct65AssignAND::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65AssignAND::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65AssignDIV.cpp -------------

#include "pch.h"

CAct65AssignDIV::CAct65AssignDIV()

{

}

CAct65AssignDIV::~CAct65AssignDIV()

{

}

bool CAct65AssignDIV::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65AssignDIV::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65AssignLSh.cpp -------------

#include "pch.h"

CAct65AssignLSh::CAct65AssignLSh()

{

}

CAct65AssignLSh::~CAct65AssignLSh()

{

}

bool CAct65AssignLSh::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65AssignLSh::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65Assignment.cpp -------------

#include "pch.h"

CAct65Assignment::CAct65Assignment()

{

}

CAct65Assignment::~CAct65Assignment()

{

}

bool CAct65Assignment::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65Assignment::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65AssignMOD.cpp -------------

#include "pch.h"

CAct65AssignMOD::CAct65AssignMOD()

{

}

CAct65AssignMOD::~CAct65AssignMOD()

{

}

bool CAct65AssignMOD::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65AssignMOD::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65AssignMULT.cpp -------------

#include "pch.h"

CAct65AssignMULT::CAct65AssignMULT()

{

}

CAct65AssignMULT::~CAct65AssignMULT()

{

}

bool CAct65AssignMULT::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65AssignMULT::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65AssignOR.cpp -------------

#include "pch.h"

CAct65AssignOR::CAct65AssignOR()

{

}

CAct65AssignOR::~CAct65AssignOR()

{

}

bool CAct65AssignOR::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65AssignOR::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65AssignRSH.cpp -------------

#include "pch.h"

CAct65AssignRSH::CAct65AssignRSH()

{

}

CAct65AssignRSH::~CAct65AssignRSH()

{

}

bool CAct65AssignRSH::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65AssignRSH::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65AssignSUB.cpp -------------

#include "pch.h"

CAct65AssignSUB::CAct65AssignSUB()

{

}

CAct65AssignSUB::~CAct65AssignSUB()

{

}

bool CAct65AssignSUB::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65AssignSUB::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65AssignXOR.cpp -------------

#include "pch.h"

CAct65AssignXOR::CAct65AssignXOR()

{

}

CAct65AssignXOR::~CAct65AssignXOR()

{

}

bool CAct65AssignXOR::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65AssignXOR::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65BEGIN.cpp -------------

#include "pch.h"

CAct65BEGIN::CAct65BEGIN()

{

}

CAct65BEGIN::~CAct65BEGIN()

{

}

bool CAct65BEGIN::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65BEGIN::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65BIT.cpp -------------

#include "pch.h"

CAct65BIT::CAct65BIT()

{

}

CAct65BIT::~CAct65BIT()

{

}

bool CAct65BIT::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65BIT::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65BitWiseAND.cpp -------------

#include "pch.h"

CAct65BitWiseAND::CAct65BitWiseAND()

{

}

CAct65BitWiseAND::~CAct65BitWiseAND()

{

}

bool CAct65BitWiseAND::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65BitWiseAND::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65BitWiseOR.cpp -------------

#include "pch.h"

CAct65BitWiseOR::CAct65BitWiseOR()

{

}

CAct65BitWiseOR::~CAct65BitWiseOR()

{

}

bool CAct65BitWiseOR::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65BitWiseOR::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65BOOL.cpp -------------

#include "pch.h"

CAct65BOOL::CAct65BOOL()

{

}

CAct65BOOL::~CAct65BOOL()

{

}

bool CAct65BOOL::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65BOOL::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65BREAK.cpp -------------

#include "pch.h"

CActBREAK::CActBREAK()

{

}

CActBREAK::~CActBREAK()

{

}

bool CActBREAK::Create()

{

return false;

}

CValue\* CActBREAK::Process()

{

return nullptr;

}

---------- Act65BYTE.cpp -------------

#include "pch.h"

CAct65BYTE::CAct65BYTE()

{

}

CAct65BYTE::~CAct65BYTE()

{

}

bool CAct65BYTE::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65BYTE::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65CARD.cpp -------------

#include "pch.h"

CAct65CARD::CAct65CARD()

{

SetNodeName(m\_pNodeTyypeName);

}

CAct65CARD::~CAct65CARD()

{

}

bool CAct65CARD::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65CARD::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65CHAR.cpp -------------

#include "pch.h"

CAct65CHAR::CAct65CHAR()

{

}

CAct65CHAR::~CAct65CHAR()

{

}

bool CAct65CHAR::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65CHAR::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65CharCibstabt.cpp -------------

#include "pch.h"

CAct65CharConstant::CAct65CharConstant()

{

}

CAct65CharConstant::~CAct65CharConstant()

{

}

bool CAct65CharConstant::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65CharConstant::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65CodeBlock.cpp -------------

#include "pch.h"

CAct65CodeBlock::CAct65CodeBlock()

{

}

CAct65CodeBlock::~CAct65CodeBlock()

{

}

bool CAct65CodeBlock::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65CodeBlock::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65CurrentLocation.cpp -------------

#include "pch.h"

CAct65CurrentLocation::CAct65CurrentLocation()

{

}

CAct65CurrentLocation::~CAct65CurrentLocation()

{

}

bool CAct65CurrentLocation::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65CurrentLocation::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65DAS.cpp -------------

#include "pch.h"

CAct65DAS::CAct65DAS()

{

}

CAct65DAS::~CAct65DAS()

{

}

bool CAct65DAS::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65DAS::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65DCS.cpp -------------

#include "pch.h"

CAct65DCS::CAct65DCS()

{

}

CAct65DCS::~CAct65DCS()

{

}

bool CAct65DCS::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65DCS::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65DEFINE.cpp -------------

#include "pch.h"

CAct65DEFINE::CAct65DEFINE()

{

SetNodeName(m\_pNodeTyypeName);

}

CAct65DEFINE::~CAct65DEFINE()

{

}

bool CAct65DEFINE::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65DEFINE::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65DL.cpp -------------

#include "pch.h"

CAct65DL::CAct65DL()

{

}

CAct65DL::~CAct65DL()

{

}

bool CAct65DL::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65DL::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65DO.cpp -------------

#include "pch.h"

CAct65DO::CAct65DO()

{

}

CAct65DO::~CAct65DO()

{

}

bool CAct65DO::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65DO::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65DW.cpp -------------

#include "pch.h"

CAct65DW::CAct65DW()

{

}

CAct65DW::~CAct65DW()

{

}

bool CAct65DW::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65DW::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65ELSE.cpp -------------

#include "pch.h"

CAct65ELSE::CAct65ELSE()

{

}

CAct65ELSE::~CAct65ELSE()

{

}

bool CAct65ELSE::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65ELSE::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65ELSEIF.cpp -------------

#include "pch.h"

CAct65ELSEIF::CAct65ELSEIF()

{

}

CAct65ELSEIF::~CAct65ELSEIF()

{

}

bool CAct65ELSEIF::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65ELSEIF::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65END.cpp -------------

#include "pch.h"

CAct65END::CAct65END()

{

}

CAct65END::~CAct65END()

{

}

bool CAct65END::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65END::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65EPROC.cpp -------------

#include "pch.h"

CAct65EPROC::CAct65EPROC()

{

}

CAct65EPROC::~CAct65EPROC()

{

}

bool CAct65EPROC::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65EPROC::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65EqualTO.cpp -------------

#include "pch.h"

CAct65EqualTO::CAct65EqualTO()

{

SetNodeName(m\_pNodeTyypeName);

}

CAct65EqualTO::~CAct65EqualTO()

{

}

bool CAct65EqualTO::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65EqualTO::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65EXIT.cpp -------------

#include "pch.h"

CAct65EXIT::CAct65EXIT()

{

}

CAct65EXIT::~CAct65EXIT()

{

}

bool CAct65EXIT::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65EXIT::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65FFI.cpp -------------

#include "pch.h"

CAct65FFI::CAct65FFI()

{

}

CAct65FFI::~CAct65FFI()

{

}

bool CAct65FFI::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65FFI::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65FI.cpp -------------

#include "pch.h"

CAct65FI::CAct65FI()

{

}

CAct65FI::~CAct65FI()

{

}

bool CAct65FI::Create()

{

return false;

}

CValue\* CAct65FI::Process()

{

return nullptr;

}

---------- Act65FlagCARRY.cpp -------------

#include "pch.h"

CAct65FlagCARRY::CAct65FlagCARRY()

{

}

CAct65FlagCARRY::~CAct65FlagCARRY()

{

}

bool CAct65FlagCARRY::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65FlagCARRY::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65FlagNEG.cpp -------------

#include "pch.h"

CAct65FlagNEG::CAct65FlagNEG()

{

}

CAct65FlagNEG::~CAct65FlagNEG()

{

}

bool CAct65FlagNEG::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65FlagNEG::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65FlagOVERFLOW.cpp -------------

#include "pch.h"

CAct65FlagOVERFLOW::CAct65FlagOVERFLOW()

{

}

CAct65FlagOVERFLOW::~CAct65FlagOVERFLOW()

{

}

bool CAct65FlagOVERFLOW::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65FlagOVERFLOW::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65FlagZERO.cpp -------------

#include "pch.h"

CAct65FlagZERO::CAct65FlagZERO()

{

}

CAct65FlagZERO::~CAct65FlagZERO()

{

}

bool CAct65FlagZERO::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65FlagZERO::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65FOR.cpp -------------

#include "pch.h"

CAct65FOR::CAct65FOR()

{

}

CAct65FOR::~CAct65FOR()

{

}

bool CAct65FOR::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65FOR::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65ForItterator.cpp -------------

#include "pch.h"

CAct65ForItterator::CAct65ForItterator()

{

}

CAct65ForItterator::~CAct65ForItterator()

{

}

bool CAct65ForItterator::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65ForItterator::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65ForSTEP.cpp -------------

#include "pch.h"

CAct65ForSTEP::CAct65ForSTEP()

{

}

CAct65ForSTEP::~CAct65ForSTEP()

{

}

bool CAct65ForSTEP::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65ForSTEP::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65ForTO.cpp -------------

#include "pch.h"

CAct65ForTO::CAct65ForTO()

{

}

CAct65ForTO::~CAct65ForTO()

{

}

bool CAct65ForTO::Create()

{

return false;

}

CValue\* CAct65ForTO::Process()

{

return nullptr;

}

---------- Act65FuncCall.cpp -------------

#include "pch.h"

CAct65FuncCall::CAct65FuncCall()

{

}

CAct65FuncCall::~CAct65FuncCall()

{

}

bool CAct65FuncCall::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65FuncCall::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65GreaterTHAN.cpp -------------

#include "pch.h"

CAct65GreaterTHAN::CAct65GreaterTHAN()

{

SetNodeName(m\_pNodeTyypeName);

}

CAct65GreaterTHAN::~CAct65GreaterTHAN()

{

}

bool CAct65GreaterTHAN::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65GreaterTHAN::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65GTEQ.cpp -------------

#include "pch.h"

CAct65GTEQ::CAct65GTEQ()

{

SetNodeName(m\_pNodeTyypeName);

}

CAct65GTEQ::~CAct65GTEQ()

{

}

bool CAct65GTEQ::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65GTEQ::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65IDENT.cpp -------------

#include "pch.h"

CAct65IDENT::CAct65IDENT()

{

}

CAct65IDENT::~CAct65IDENT()

{

}

bool CAct65IDENT::Create(CBin\* pSym)

{

bool rV = true;

CValue\* pV = new CValue;

pV->Create(pSym);

CAstNode::CreateNode();

return rV;

}

CValue\* CAct65IDENT::Process()

{

return nullptr;

}

---------- Act65IF.cpp -------------

#include "pch.h"

CAct65IF::CAct65IF()

{

}

CAct65IF::~CAct65IF()

{

}

bool CAct65IF::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65IF::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65IFF.cpp -------------

#include "pch.h"

CAct65IFF::CAct65IFF()

{

}

CAct65IFF::~CAct65IFF()

{

}

bool CAct65IFF::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65IFF::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65INT.cpp -------------

#include "pch.h"

CAct65INT::CAct65INT()

{

SetNodeName(m\_pNodeTyypeName);

}

CAct65INT::~CAct65INT()

{

}

bool CAct65INT::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65INT::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65INTERRUPT.cpp -------------

#include "pch.h"

CAct65INTERRUPT::CAct65INTERRUPT()

{

}

CAct65INTERRUPT::~CAct65INTERRUPT()

{

}

bool CAct65INTERRUPT::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65INTERRUPT::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65Label.cpp -------------

#include "pch.h"

CAct65Label::CAct65Label()

{

}

CAct65Label::~CAct65Label()

{

}

bool CAct65Label::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65Label::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65LessTHAN.cpp -------------

#include "pch.h"

CAct65LessTHAN::CAct65LessTHAN()

{

SetNodeName(m\_pNodeTyypeName);

}

CAct65LessTHAN::~CAct65LessTHAN()

{

}

bool CAct65LessTHAN::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65LessTHAN::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65LogicalAND.cpp -------------

#include "pch.h"

CAct65LogicalAND::CAct65LogicalAND()

{

}

CAct65LogicalAND::~CAct65LogicalAND()

{

}

bool CAct65LogicalAND::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65LogicalAND::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65LogicalOR.cpp -------------

#include "pch.h"

CAct65LogicalOR::CAct65LogicalOR()

{

}

CAct65LogicalOR::~CAct65LogicalOR()

{

}

bool CAct65LogicalOR::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65LogicalOR::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65LowerPart.cpp -------------

#include "pch.h"

CAct65LowerPart::CAct65LowerPart()

{

}

CAct65LowerPart::~CAct65LowerPart()

{

}

bool CAct65LowerPart::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65LowerPart::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65LSH.cpp -------------

#include "pch.h"

CAct65LSH::CAct65LSH()

{

}

CAct65LSH::~CAct65LSH()

{

}

bool CAct65LSH::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65LSH::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65LTEQ.cpp -------------

#include "pch.h"

CAct65LTEQ::CAct65LTEQ()

{

}

CAct65LTEQ::~CAct65LTEQ()

{

}

bool CAct65LTEQ::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65LTEQ::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65Module.cpp -------------

#include "pch.h"

CAct65Module::CAct65Module()

{

}

CAct65Module::~CAct65Module()

{

}

bool CAct65Module::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65Module::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65NotEquelTO.cpp -------------

#include "pch.h"

CAct65NotEquelTO::CAct65NotEquelTO()

{

}

CAct65NotEquelTO::~CAct65NotEquelTO()

{

}

bool CAct65NotEquelTO::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65NotEquelTO::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65OD.cpp -------------

#include "pch.h"

CAct65OD::CAct65OD()

{

}

CAct65OD::~CAct65OD()

{

}

bool CAct65OD::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65OD::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65Opcode.cpp -------------

#include "pch.h"

CAct65Opcode::CAct65Opcode()

{

m\_OpcodeToken = Token(-1);

m\_pOpCodeName = 0;

m\_pInstruction = 0;

m\_LineNumber = 0;

m\_ColumnNumber = 0;

m\_pSym = 0;

m\_pLabel = 0;

m\_AdressMode = AdrModeType::NA;

}

CAct65Opcode::~CAct65Opcode()

{

if (m\_pInstruction)

delete m\_pInstruction;

}

bool CAct65Opcode::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65Opcode::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65OptNOT.cpp -------------

#include "pch.h"

CAct65OptNOT::CAct65OptNOT()

{

}

CAct65OptNOT::~CAct65OptNOT()

{

}

bool CAct65OptNOT::Create(CAstNode\* pChild, CAstNode\* pNext)

{

return false;

}

CValue\* CAct65OptNOT::Process()

{

return nullptr;

}

---------- Act65ORG.cpp -------------

#include "pch.h"

CAct65ORG::CAct65ORG()

{

}

CAct65ORG::~CAct65ORG()

{

}

bool CAct65ORG::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65ORG::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65POINTER.cpp -------------

#include "pch.h"

CAct65POINTER::CAct65POINTER()

{

}

CAct65POINTER::~CAct65POINTER()

{

}

bool CAct65POINTER::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65POINTER::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65POP.cpp -------------

#include "pch.h"

CAct65POP::CAct65POP()

{

}

CAct65POP::~CAct65POP()

{

}

bool CAct65POP::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65POP::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65PopDest.cpp -------------

#include "pch.h"

CAct65PopDest::CAct65PopDest()

{

}

CAct65PopDest::~CAct65PopDest()

{

}

bool CAct65PopDest::Create(CAstNode\* pChild, CAstNode\* pNext)

{

return false;

}

CValue\* CAct65PopDest::Process()

{

return nullptr;

}

---------- Act65Private.cpp -------------

#include "pch.h"

CAct65Private::CAct65Private()

{

}

CAct65Private::~CAct65Private()

{

}

bool CAct65Private::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65Private::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65PROC.cpp -------------

#include "pch.h"

CAct65PROC::CAct65PROC()

{

}

CAct65PROC::~CAct65PROC()

{

}

bool CAct65PROC::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65PROC::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65ProcCall.cpp -------------

#include "pch.h"

CAct65ProcCall::CAct65ProcCall()

{

}

CAct65ProcCall::~CAct65ProcCall()

{

}

bool CAct65ProcCall::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65ProcCall::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65PROCESSOR.cpp -------------

#include "pch.h"

CAct65PROCESSOR::CAct65PROCESSOR()

{

}

CAct65PROCESSOR::~CAct65PROCESSOR()

{

}

bool CAct65PROCESSOR::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65PROCESSOR::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65PROCname.cpp -------------

#include "pch.h"

CAct65PROCname::CAct65PROCname()

{

}

CAct65PROCname::~CAct65PROCname()

{

}

bool CAct65PROCname::Create()

{

return false;

}

CValue\* CAct65PROCname::Process()

{

return nullptr;

}

---------- Act65ProcParams.cpp -------------

#include "pch.h"

CAct65ProcParams::CAct65ProcParams()

{

}

CAct65ProcParams::~CAct65ProcParams()

{

}

bool CAct65ProcParams::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65ProcParams::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65PUSH.cpp -------------

#include "pch.h"

CAct65PUSH::CAct65PUSH()

{

}

CAct65PUSH::~CAct65PUSH()

{

}

bool CAct65PUSH::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65PUSH::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65PushSource.cpp -------------

#include "pch.h"

CAct65PushSource::CAct65PushSource()

{

}

CAct65PushSource::~CAct65PushSource()

{

}

bool CAct65PushSource::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65PushSource::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65RECTYPE.cpp -------------

#include "pch.h"

CAct65RECTYPE::CAct65RECTYPE()

{

}

CAct65RECTYPE::~CAct65RECTYPE()

{

}

bool CAct65RECTYPE::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65RECTYPE::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65RETURN.cpp -------------

#include "pch.h"

CAct65RETURN::CAct65RETURN()

{

}

CAct65RETURN::~CAct65RETURN()

{

}

bool CAct65RETURN::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65RETURN::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65ROL.cpp -------------

#include "pch.h"

CAct65ROL::CAct65ROL()

{

}

CAct65ROL::~CAct65ROL()

{

}

bool CAct65ROL::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65ROL::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65ROR.cpp -------------

#include "pch.h"

CAct65ROR::CAct65ROR()

{

}

CAct65ROR::~CAct65ROR()

{

}

bool CAct65ROR::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65ROR::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65RSH.cpp -------------

#include "pch.h"

CAct65RSH::CAct65RSH()

{

}

CAct65RSH::~CAct65RSH()

{

}

bool CAct65RSH::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65RSH::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65RTI.cpp -------------

#include "pch.h"

CAct65RTI::CAct65RTI()

{

}

CAct65RTI::~CAct65RTI()

{

}

bool CAct65RTI::Create()

{

return false;

}

CValue\* CAct65RTI::Process()

{

return nullptr;

}

---------- Act65SecAtrbFALSE.cpp -------------

#include "pch.h"

CAct65SecAtrbFALSE::CAct65SecAtrbFALSE()

{

}

CAct65SecAtrbFALSE::~CAct65SecAtrbFALSE()

{

}

bool CAct65SecAtrbFALSE::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65SecAtrbFALSE::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65SecAtrbMODE.cpp -------------

#include "pch.h"

CAct65SecAtrbMODE::CAct65SecAtrbMODE()

{

}

CAct65SecAtrbMODE::~CAct65SecAtrbMODE()

{

}

bool CAct65SecAtrbMODE::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65SecAtrbMODE::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65SecAtrbREADONLY.cpp -------------

#include "pch.h"

CAct65SecAtrbREADONLY::CAct65SecAtrbREADONLY()

{

}

CAct65SecAtrbREADONLY::~CAct65SecAtrbREADONLY()

{

}

bool CAct65SecAtrbREADONLY::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65SecAtrbREADONLY::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65SecAtrbREADWRITE.cpp -------------

#include "pch.h"

CAct65SecAtrbREADWRITE::CAct65SecAtrbREADWRITE()

{

}

CAct65SecAtrbREADWRITE::~CAct65SecAtrbREADWRITE()

{

}

bool CAct65SecAtrbREADWRITE::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65SecAtrbREADWRITE::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65SecAtrbSIZE.cpp -------------

#include "pch.h"

CAct65SecAtrbSIZE::CAct65SecAtrbSIZE()

{

}

CAct65SecAtrbSIZE::~CAct65SecAtrbSIZE()

{

}

bool CAct65SecAtrbSIZE::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65SecAtrbSIZE::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65SecAtrbSTART.cpp -------------

#include "pch.h"

CAct65SecAtrbSTART::CAct65SecAtrbSTART()

{

}

CAct65SecAtrbSTART::~CAct65SecAtrbSTART()

{

}

bool CAct65SecAtrbSTART::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65SecAtrbSTART::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65SecAtrbZEROPAGE.cpp -------------

#include "pch.h"

CAct65SecAtrbZEROPAGE::CAct65SecAtrbZEROPAGE()

{

}

CAct65SecAtrbZEROPAGE::~CAct65SecAtrbZEROPAGE()

{

}

bool CAct65SecAtrbZEROPAGE::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65SecAtrbZEROPAGE::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65SECTION.cpp -------------

#include "pch.h"

CAct65SECTION::CAct65SECTION()

{

}

CAct65SECTION::~CAct65SECTION()

{

}

bool CAct65SECTION::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65SECTION::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65SUB.cpp -------------

#include "pch.h"

CAct65SUB::CAct65SUB()

{

}

CAct65SUB::~CAct65SUB()

{

}

bool CAct65SUB::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65SUB::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65THEN.cpp -------------

#include "pch.h"

CAct65THEN::CAct65THEN()

{

}

CAct65THEN::~CAct65THEN()

{

}

bool CAct65THEN::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65THEN::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65TRUE.cpp -------------

#include "pch.h"

CAct65TRUE::CAct65TRUE()

{

}

CAct65TRUE::~CAct65TRUE()

{

}

bool CAct65TRUE::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65TRUE::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65TYPEDEF.cpp -------------

#include "pch.h"

CAct65tTYPEDEF::CAct65tTYPEDEF()

{

}

CAct65tTYPEDEF::~CAct65tTYPEDEF()

{

}

bool CAct65tTYPEDEF::Create(CAstNode\* pChild, CAstNode\* pNext)

{

return false;

}

CValue\* CAct65tTYPEDEF::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65UNTILL.cpp -------------

#include "pch.h"

CAct65UNTILL::CAct65UNTILL()

{

}

CAct65UNTILL::~CAct65UNTILL()

{

}

bool CAct65UNTILL::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65UNTILL::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65UpperPart.cpp -------------

#include "pch.h"

CAct65UpperPart::CAct65UpperPart()

{

}

CAct65UpperPart::~CAct65UpperPart()

{

}

bool CAct65UpperPart::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65UpperPart::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65VECTOR.cpp -------------

#include "pch.h"

CAct65VECTOR::CAct65VECTOR()

{

}

CAct65VECTOR::~CAct65VECTOR()

{

}

bool CAct65VECTOR::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65VECTOR::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65WHILE.cpp -------------

#include "pch.h"

CAct65WHILE::CAct65WHILE()

{

}

CAct65WHILE::~CAct65WHILE()

{

}

bool CAct65WHILE::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65WHILE::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65XOR.cpp -------------

#include "pch.h"

CAct65XOR::CAct65XOR()

{

}

CAct65XOR::~CAct65XOR()

{

}

bool CAct65XOR::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65XOR::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65XREG.cpp -------------

#include "pch.h"

CAct65XREG::CAct65XREG()

{

}

CAct65XREG::~CAct65XREG()

{

}

bool CAct65XREG::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65XREG::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- Act65YREG.cpp -------------

#include "pch.h"

CAct65YREG::CAct65YREG()

{

}

CAct65YREG::~CAct65YREG()

{

}

bool CAct65YREG::Create(CAstNode\* pChild, CAstNode\* pNext)

{

SetNodeName(m\_pNodeTyypeName);

return CAstNode::CreateNode(pChild, pNext);

}

CValue\* CAct65YREG::Process()

{

CAstNode\* pChild = 0, \* pNext = 0;

CValue\* pValueChild = 0, \* pValueNext = 0

;

pChild = GetChild();

if (pChild)

{

pNext = pChild->GetNext();

}

if (pChild)

{

pValueChild = pChild->Process();

}

if (pNext)

{

pValueNext = pNext->Process();

}

//-----------------------

// Code Generation

//-----------------------

return pValueChild;

}

---------- ActionApp.cpp -------------

#include "pch.h"

CActionApp ActApp;

CActionApp\* Act()

{

return &ActApp;

}

int main(int argc, char\* argv[])

{

fprintf(stderr, "The Action! Compiler for the 6502 Ver 0.0.1\nNov 27, 2024\n");

ActApp.Create(argc, argv);

ActApp.Run();

fprintf(stderr, "Done\n");

}

//--------------------------------------

// Action App Class

//--------------------------------------

CActionApp::CActionApp()

{

m\_pSourceFile = 0;

m\_pLogFile = 0;

m\_pObjectFile = 0;

m\_pBinaryFile = 0;

m\_pfSrc = 0;

m\_pfLog = 0;

m\_pfBin = 0;

m\_pfObj = 0;

}

CActionApp::~CActionApp()

{

if (m\_pSourceFile)delete[]m\_pSourceFile;

if (m\_pLogFile)delete[]m\_pLogFile;

if (m\_pObjectFile)delete[]m\_pObjectFile;

if (m\_pBinaryFile)delete[]m\_pBinaryFile;

CloseAll();

}

bool CActionApp::Create(int argc, char\* argv[])

{

//--------------------------------

// parse the command line

//

// Command line format:

// action65 <source file> options

//

// options:

// -L <log file name>

// -O <object file output

// -B <binary file output for burning rom>

//--------------------------------

int i;

int l=0;

if (argc == 1)

{

fprintf(stderr, "Usage:\n");

fprintf(stderr, "action65 <source file> options\n");

fprintf(stderr, "Options:\n");

fprintf(stderr, "-L <Log File Name>\n");

fprintf(stderr, "-B <Binary File Name>\n");

fprintf(stderr, "-O <Object File Name>\n");

}

else

{

for (i = 1; i < argc; ++i)

{

if (i == 1)

{

//---------------------

// this had better be

// the input source

// file

//---------------------

l = strlen(argv[i]) + 1;

m\_pSourceFile = new char[l];

strcpy\_s(m\_pSourceFile, l, argv[i]);

}

else if (argv[i][0] == '-')

{

switch (argv[i][1])

{

case 'L':

++i;

l = strlen(argv[i]) + 1;

m\_pLogFile = new char[l];

strcpy\_s(m\_pLogFile, l, argv[i]);

break;

case 'O':

++i;

l = strlen(argv[i]) + 1;

m\_pObjectFile = new char[l];

strcpy\_s(m\_pObjectFile, l, argv[i]);

break;

case 'B':

++i;

l = strlen(argv[i]) + 1;

m\_pBinaryFile = new char[l];

strcpy\_s(m\_pBinaryFile, l, argv[i]);

break;

}

}

}

}

return true;

}

bool CActionApp::Run()

{

bool rV = false;

OpenLog();

m\_ActParse.Create();

m\_ActParse.Run();

return rV;

}

bool CActionApp::OpenSource()

{

bool rV = false;

errno\_t err;

if (m\_pSourceFile)

{

err = fopen\_s(&m\_pfSrc, m\_pSourceFile, "r");

if (err == 0)

rV = true;

else

{

fprintf(stderr, "Could not open %s\n", m\_pSourceFile);

CloseAll();

exit(1);

}

}

return rV;

}

bool CActionApp::OpenLog()

{

bool rV = false;

errno\_t err;

if (m\_pLogFile)

{

err = fopen\_s(&m\_pfLog, m\_pLogFile, "w");

if (err == 0)

rV = true;

else

{

fprintf(stderr, "Unable to open %s as the log file\n", m\_pLogFile);

}

}

return rV;

}

bool CActionApp::OpenObj()

{

bool rV = false;

errno\_t err;

if (m\_pObjectFile)

{

err = fopen\_s(&m\_pfObj, m\_pObjectFile, "wb");

if (err == 0)

rV = true;

}

return rV;

}

bool CActionApp::OpenBin()

{

bool rV = false;

errno\_t err;

if (m\_pBinaryFile)

{

err = fopen\_s(&m\_pfBin, m\_pBinaryFile, "wb");

if (err == 0)

rV = true;

}

return rV;

}

void CActionApp::CloseSource()

{

if(m\_pfSrc)

fclose(m\_pfSrc);

m\_pfSrc = 0;

}

void CActionApp::CloseLog()

{

if(m\_pfLog)

fclose(m\_pfLog);

m\_pfLog = 0;

}

void CActionApp::CloseObj()

{

if(m\_pfObj)

fclose(m\_pfObj);

m\_pfObj = 0;

}

void CActionApp::CloseBin()

{

if(m\_pfBin)

fclose(m\_pfBin);

m\_pfBin = 0;

}

void CActionApp::CloseAll()

{

CloseSource();

CloseLog();

CloseBin();

CloseObj();

}

char\* CActionApp::IndentString(char\* s, int Indent, int c)

{

int i;

for (i = 0; i < Indent;++i)

s[i] = c;

s[i] = 0;

return s;

}

void CActionApp::Dump(

FILE\* pOut,

char\* pMem,

unsigned StartAddress,

unsigned Size

)

{

unsigned Loc = 0;

bool bDone = false;

char\* s = new char[256];

char strPad[64];

int Pad;

char Ascii[10];

int l, i, c, n;

Ascii[8] = 0;

while (!bDone)

{

l = 0;

l = sprintf\_s(s, 256, "%04X ", Loc + StartAddress);

if ((Size - Loc) <= 0)

{

bDone = true;

}

else

{

if ((Size - Loc) < 8)

n = Size - Loc;

else

n = 8;

for (i = 0; i < n; ++i)

{

l += sprintf\_s(&s[l], 256 - l,

"%02X ", unsigned(pMem[Loc + i]) & 0x0ff);

}

if (n < 8)

{

Pad = (8 - n) \* 3;

Act()->IndentString(strPad, Pad, ' ');

l += sprintf\_s(&s[l], 256 - l, "%s", strPad);

}

for (i = 0; i < n; ++i)

{

c = pMem[Loc + i];

if (c < 0x20)

c = '.';

else if (c > 0x7f)

c = '.';

Ascii[i] = c;

}

fprintf(pOut, "%s %s\n", s, Ascii);

Loc += 8;

}

}

delete[] s;

}

char\* CActionApp::CreateIndentString(char\* s, int n, int Indent, int c)

{

int i;

for (i = 0; (i < Indent) && (i < (n - 1)); ++i)

s[i] = c;

s[i] = 0;

return s;

}

---------- ActionAstTree.cpp -------------

#include "pch.h"

CActionAstTree::CActionAstTree()

{

}

CActionAstTree::~CActionAstTree()

{

}

bool CActionAstTree::Create()

{

return false;

}

CAstNode\* CActionAstTree::MakeNewNode(

int Type,

CAstNode\* Child,

CAstNode\* Next

)

{

return nullptr;

}

---------- ActStatusFlags.cpp -------------

#include "pch.h"

CActStatusFlags::CActStatusFlags()

{

}

CActStatusFlags::~CActStatusFlags()

{

}

bool CActStatusFlags::Create()

{

return false;

}

---------- Actt65DB.cpp -------------

#include "pch.h"

CAct65DB::CAct65DB()

{

}

CAct65DB::~CAct65DB()

{

}

bool CAct65DB::Create(CAstNode\* pChild, CAstNode\* pNext)

{

return false;

}

CValue\* CAct65DB::Process()

{

return nullptr;

}

---------- Actt65DS.cpp -------------

#include "pch.h"

CAct65DS::CAct65DS()

{

}

CAct65DS::~CAct65DS()

{

}

bool CAct65DS::Create(CAstNode\* pChild, CAstNode\* pNext)

{

return false;

}

CValue\* CAct65DS::Process()

{

return nullptr;

}

---------- AddressSizeStackItem.cpp -------------

#include "pch.h"

CAddressSizeStackItem::CAddressSizeStackItem()

{

m\_OpRandAdrSize = CSection::AddressSize::ADDRESSSIZE\_WORD; //default

}

CAddressSizeStackItem::~CAddressSizeStackItem()

{

}

bool CAddressSizeStackItem::Create()

{

return CStackItem::Create(CStackItem::ItemType::ADDRESSSIZE);

}

---------- AstNode.cpp -------------

#include "pch.h"

CAstNode::CAstNode()

{

m\_NodeID = ++m\_NodeCount;

m\_NodeType = 0;

m\_pStart = 0;

m\_pNext = 0;

m\_pPrev = 0;

m\_pChild = 0;

m\_pParent = 0;

m\_pNodeName = 0;

m\_pValue = 0;

m\_pHead = 0;

m\_pTail = 0;

}

CAstNode::CAstNode(int NodeType)

{

m\_NodeID = ++m\_NodeCount;

m\_NodeType = NodeType;

m\_pStart = 0;

m\_pNext = 0;

m\_pPrev = 0;

m\_pChild = 0;

m\_pParent = 0;

m\_pNodeName = 0;

m\_pValue = 0;

m\_pHead = 0;

m\_pTail = 0;

}

CAstNode::~CAstNode()

{

}

bool CAstNode::CreateNode(

CAstNode\* pChild,

CAstNode\* pNext

)

{

SetChild(pChild);

SetNext(pNext);

SetChild(pChild);

if (pChild)

{

pChild->AddToTail(pNext);

}

if(pNext)GetNext()->SetPrev(this);

SetStart(Act()->GetParser()->GetAstTree()->GetRootNode());

return true;

}

void CAstNode::Print(FILE\* pOut, int Indent, char\* s)

{

int ChildID, NextID;

// char\* pIndentString;

// pIndentString = new char[256];

// Act()->IndentString(pIndentString, Indent, ' ');

if (GetChild())

ChildID = GetChild()->GetID();

else

ChildID = -1;

if (GetNext())

NextID = GetNext()->GetID();

else

NextID = -1;

fprintf(

pOut,

";\t %4d %4d %4d %s%s %s\n",

GetID(),

ChildID,

NextID,

s,

this->GetNodeName(),

GetSymbol()?GetSymbol()->GetName():"..."

);

// delete[] pIndentString;

}

void CAstNode::AddToHead(CAstNode\* pN)

{

if (GetHead())

{

GetHead()->SetPrev(pN);

pN->SetNext(GetHead());

SetHead(pN);

}

else

{

SetTail(pN);

SetHead(pN);

}

}

void CAstNode::AddToTail(CAstNode\* pNode)

{

if (GetHead())

{

GetTail()->SetNext(pNode);

pNode->SetPrev(GetTail());

SetTail(pNode);

}

else

{

SetTail(pNode);

SetHead(pNode);

}

}

---------- AstTree.cpp -------------

#include "pch.h"

CAstTree::CAstTree()

{

}

CAstTree::~CAstTree()

{

}

bool CAstTree::Create()

{

return false;

}

void CAstTree::Print(FILE\* pOut)

{

char\* s = 0;

if (GetRootNode() && pOut)

{

s = new char[4096];

memset(s, 0, 4096);

fprintf(

pOut,

"\t NUMB DOWN NEXT\n"

);

TraverseTree(pOut, GetRootNode(), s);

delete[] s;

}

if(Act()->LogFile())

fprintf(Act()->LogFile(),"Root Node is NULL");

}

void CAstTree::TraverseTree(FILE\* pOut, CAstNode\* pNode, char\* s)

{

//-------------------------------

// Navigate the Syntax Tree

//-------------------------------

static int Recursions = 0;

if (++Recursions > 100)

{

fprintf(pOut, "Too Many Recursions\n");

Act()->CloseAll();

exit(2);

}

while (pNode->GetNext()) /\* while next pointer points to valid node \*/

{

strcat\_s(s, 4096, "+-");

pNode->Print(pOut, 0,s);

s[strlen(s) - 2] = 0;

if (pNode->GetChild())

{

strcat\_s(s, 4096, "| ");

TraverseTree(pOut, pNode->GetChild(),s);

s[strlen(s) - 2] = 0;

}

pNode = pNode->GetNext();

}

strcat\_s(s, 4096, "+-");

pNode->Print(pOut, 0,s);

s[strlen(s) - 2] = 0;

if (pNode->GetChild())

{

strcat\_s(s, 4096, " ");

TraverseTree(pOut, pNode->GetChild(),s);

s[strlen(s) - 2] = 0;

}

}

---------- Bin.cpp -------------

#include "pch.h"

bool CBin::Compare(const char\* name, int aux)

{

bool rV = false;

if (strcmp(GetName(), name) == 0)

rV = true;

return rV;

}

void CBin::Print(FILE\* pOut, const char\* s)

{

fprintf(pOut, "CBin::%s\n", m\_pName);

}

//-----------------------------------------------

// Sub list methods

//-----------------------------------------------

void CBin::Add(CBin\* pSym)

{

if (GetHead())

{

GetTail()->SetNext(pSym);

pSym->SetPrev(GetTail());

SetTail(pSym);

}

else

{

SetTail(pSym);

SetHead(pSym);

}

}

CBin\* CBin::Find(const char\* pName, CBin::BinType Type)

{

CBin\* pBin = 0;

bool loop = true;

pBin = GetHead();

while (pBin && loop)

{

if (pBin->Compare(pName, Type))

loop = false;

else

pBin = pBin->GetNext();

}

return pBin;

}

void CBin::Delete(CBin\* pSym)

{

Unlink(pSym);

delete pSym;

}

void CBin::Unlink(CBin\* pSym)

{

if (GetHead() == pSym)

{

SetHead(pSym->GetNext());

if (GetHead())

GetHead()->SetPrev(0);

}

else if (GetTail() == pSym)

{

SetTail(pSym->GetPrev());

if (GetTail())

GetTail()->SetNext(0);

}

else

{

pSym->GetNext()->SetPrev(pSym->GetPrev());

pSym->GetPrev()->SetNext(pSym->GetNext());

}

}

---------- BinStackItem.cpp -------------

#include "pch.h"

CBinStackItem::CBinStackItem()

{

m\_pBin = 0;

}

CBinStackItem::~CBinStackItem()

{

}

bool CBinStackItem::Create()

{

return CStackItem::Create(CStackItem::ItemType::BIN);

}

---------- Bucket.cpp -------------

#include "pch.h"

CBucket::~CBucket()

{

CBin\* pBin;

pBin = m\_pHead;

while (pBin)

{

pBin->Print(stdout);

m\_pHead = pBin->GetNext();

delete pBin;

pBin = m\_pHead;

}

}

void CBucket::Add(CBin\* pSym)

{

if (m\_pHead)

{

m\_pTail->SetNext(pSym);

pSym->SetPrev(m\_pTail);

m\_pTail = pSym;

}

else

{

m\_pTail = pSym;

m\_pHead = pSym;

}

}

CBin\* CBucket::Find(const char\* pName, int scope)

{

CBin\* pBin = 0;

bool loop = true;

pBin = GetHead();

while (pBin && loop)

{

if (pBin->Compare(pName, scope))

loop = false;

else

pBin = pBin->GetNext();

}

return pBin;

}

void CBucket::Delete(CBin\* pSym)

{

Unlink(pSym);

delete pSym;

}

void CBucket::Unlink(CBin\* pSym)

{

if (m\_pHead == pSym)

{

m\_pHead = pSym->GetNext();

m\_pHead->SetPrev(0);

}

else if (m\_pTail == pSym)

{

m\_pTail = pSym->GetPrev();

m\_pTail->SetNext(0);

}

else

{

pSym->GetNext()->SetPrev(pSym->GetPrev());

pSym->GetPrev()->SetNext(pSym->GetNext());

}

}

---------- DataSizeStackItem.cpp -------------

#include "pch.h"

CDataSizeStackItem::CDataSizeStackItem()

{

}

CDataSizeStackItem::~CDataSizeStackItem()

{

}

bool CDataSizeStackItem::Create()

{

return CStackItem::Create(CStackItem::ItemType::DATA\_SIZE);;

}

---------- Instruction.cpp -------------

#include "pch.h"

CInstruction::CInstruction()

{

m\_OpCode = 0;

m\_LowByte = 0;

m\_HighByte = 0;

m\_unused = 0;

m\_ByteCount = 0;

m\_OpCodeToken = Token::BRK;

m\_UnResolved = false;

}

CInstruction::~CInstruction()

{

m\_OpCodeToken = Token(0);

m\_OpCode = 0;

m\_LowByte = 0;

m\_HighByte = 0;

m\_unused = 0;

m\_ByteCount = 1;

}

bool CInstruction::Create()

{

CStackItem::Create(CStackItem::ItemType::INSTRUCTION);

return 0;

}

void CInstruction::AddToCurrentSection()

{

CSection\* pSec;

unsigned short Address;

pSec = Act()->GetParser()->GetCurrentSection();

Address = (unsigned short)pSec->GetLocationCounter();

pSec->AddData(1, GetOpCode());

if (GetByteCount() == 1)

{

// fprintf(

// Act()->LogFile(),

// "Add Instruction %04x: %02x\n",

// Address & 0x0ffff,

// GetOpCode() & 0x0ff

// );

}

else if (GetByteCount() == 2)

{

pSec->AddData(1, GetLowByte());

// fprintf(

// Act()->LogFile(),

// "Add Instruction %04hx: %02x,%02x\n",

// Address & 0x0ffff,

// GetOpCode() & 0x0ff,

// GetLowByte() & 0x0ff);

}

else

{

// fprintf(

// Act()->LogFile(),

// "Add Instruction %04hX: %02X, %04X\n",

// Address &0x0ffff,

// GetOpCode() & 0x0ff,

// GetOperandW() & 0x0ffff

// );

pSec->AddData(2, GetOperandW());

}

}

bool CInstruction::IsRelative()

{

bool rV = false;

switch (m\_OpCodeToken)

{

case Token::BCC:

case Token::BCS:

case Token::BEQ:

case Token::BMI:

case Token::BNE:

case Token::BPL:

case Token::BVC:

case Token::BVS:

rV = true;

break;

}

return rV;

}

---------- Lexer.cpp -------------

#include "pch.h"

CLexer::CLexer()

{

int i;

for (i = 0; i < 256; ++i)

m\_aLexBuff[i] = 0;

m\_UngetBuffer = 0;

m\_LexBuffIndex = 0;

m\_Line = 1;

m\_Col = 0;

m\_Number = 0;

m\_pLexSymbol = 0;

m\_pFileBuffeer = 0;

m\_FileIndex = 0;

m\_InFileSize = 0;

m\_bAsmMode = false;

}

CLexer::~CLexer()

{

}

bool CLexer::Create()

{

struct \_stat32 FileStats;

unsigned BytesRead = 0;

//---------------------------------

// Open Input File

//---------------------------------

\_stat32(Act()->GetSourceFileName(), &FileStats);

m\_InFileSize = FileStats.st\_size;

Act()->OpenSource();

m\_pFileBuffeer = new char[m\_InFileSize + 1];

if(m\_pFileBuffeer && Act()->SrcFile())

BytesRead = fread(m\_pFileBuffeer, 1, m\_InFileSize, Act()->SrcFile());

Act()->CloseSource();

if (BytesRead)

m\_InFileSize = BytesRead;

//----------------------------

// Create Symbol Table

//----------------------------

m\_SymbolTable.Create(101);

if(LogFile())

fprintf(LogFile(), "File:%s has %d Bytes\n", Act()->GetSourceFileName(), m\_InFileSize);

return true;

}

FILE\* CLexer::LogFile()

{

return Act()->LogFile();

}

int CLexer::LexGet()

{

int c = EOF;

if (m\_pFileBuffeer && (m\_FileIndex < m\_InFileSize))

c = m\_pFileBuffeer[m\_FileIndex++];

else if (m\_FileIndex == m\_InFileSize)

{

c = EOF;

}

m\_Col++;

return c;

}

int CLexer::LexLook(int index)

{

int c = EOF;

int i;

i = index + m\_FileIndex;

if (m\_pFileBuffeer && (i < m\_InFileSize))

c = m\_pFileBuffeer[i];

else if (m\_FileIndex == m\_InFileSize)

{

c = EOF;

}

return c;

}

void CLexer::LexUnGet(int c)

{

if ((m\_FileIndex > 0) && (c != EOF))

{

--m\_FileIndex;

if (m\_Col > 0)

m\_Col--;

}

}

bool CLexer::IsValidHexNumber(int c)

{

bool IsValid = false;

switch (c)

{

case '0': case '1': case '2': case '3': case '4':

case '5': case '6': case '7': case '8': case '9':

case 'a': case 'b': case 'c': case 'd': case 'e': case 'f':

case 'A': case 'B': case 'C': case 'D': case 'E': case 'F':

IsValid = true;

break;

}

return IsValid;

}

bool CLexer::IsValidNumber(int c)

{

bool IsValid = false;

switch (c)

{

case '0': case '1': case '2': case '3': case '4':

case '5': case '6': case '7': case '8': case '9':

IsValid = true;

break;

}

return IsValid;

}

bool CLexer::IsValidNameChar(int c)

{

bool IsValid = false;

if (isalnum(c) || c == '\_')

IsValid = true;

return IsValid;;

}

bool CLexer::IsWhiteSpace(int c)

{

bool IsValid = false;

switch (c)

{

case '\n':

case '\r':

case '\t':

case ' ':

IsValid = true;

break;;

}

return IsValid;

}

bool CLexer::IsValidAssignmentOperator(int c)

{

bool IsValid = false;

switch (c)

{

case '+':

case '-':

case '\*':

case '/':

case 'M':

case '&':

case '%':

case '!':

case 'X':

case 'L':

case 'R':

IsValid = true;

}

return IsValid;

}

Token CLexer::Lex()

{

bool Loop = true;

bool auxLoop = true;

int c = 0,c1 = 0, c2 = 0;

Token TokenValue = Token(0);

m\_LexBuffIndex = 0;

while (Loop)

{

c = LexGet();

switch (c)

{

case EOF:

TokenValue = Token::ENDOFFILE;

Loop = false;

break;

case '\n': //white space

m\_Col = 0;

m\_Line++;

break;

case '\r': //more white space

case '\t':

case ' ':

break;

case '0': case '1': case '2': case '3': case '4':

case '5': case '6': case '7': case '8': case '9':

//Decimal Number

m\_aLexBuff[m\_LexBuffIndex++] = c;

auxLoop = true;

while (auxLoop)

{

c = LexGet();

if (IsValidNumber(c))

m\_aLexBuff[m\_LexBuffIndex++] = c;

else

auxLoop = false;

}

m\_aLexBuff[m\_LexBuffIndex] = 0;

m\_Number = atoi(m\_aLexBuff);

LexUnGet(c);

Loop = false;

TokenValue = Token::NUMBER;

break;

case '$': //Hexadecimal Number

auxLoop = true;

while (auxLoop)

{

c = LexGet();

if (IsValidHexNumber(c))

m\_aLexBuff[m\_LexBuffIndex++] = c;

else

auxLoop = false;

}

m\_aLexBuff[m\_LexBuffIndex] = 0;

m\_Number = strtol(m\_aLexBuff, NULL, 16);

LexUnGet(c);

Loop = false;

TokenValue = Token::NUMBER;

break;

case '=': //assignment operators

c = LexGet();

if (IsWhiteSpace(c) || c != '=')

{

TokenValue = Token('=');

Loop = false;

LexUnGet(c);

}

else

{

c = LexGet();

if (IsValidAssignmentOperator(c))

{

switch (c)

{

case '+':

TokenValue = Token::ASSIGN\_ADD;

break;

case '-':

TokenValue = Token::ASSIGN\_SUB;

break;

case '\*':

TokenValue = Token::ASSIGN\_SUB;

break;

case '/':

TokenValue = Token::ASSIGN\_DIV;

break;

case 'M':

//throw away next two characters

LexGet();

LexGet();

TokenValue = Token::ASSIGN\_MOD;

break;

case '&':

TokenValue = Token::ASSIGN\_AND;;

break;

case '%':

TokenValue = Token::ASSIGN\_OR;

break;

case '!':

TokenValue = Token::ASSIGN\_XOR;

break;

case 'X':

TokenValue = Token::ASSIGN\_XOR;;

//throw away next two characters

LexGet();

LexGet();

break;

case 'L':

//throw away next two characters

LexGet();

LexGet();

TokenValue = Token::ASSIGN\_LSH;

break;

case 'R':

//throw away next two characters

LexGet();

LexGet();

TokenValue = Token::ASSIGN\_RSH;

break;

}

}

}

Loop = false;

break;

case '[':

case ']':

case '(':

case ')':

case'{':

case '}':

case '+': //add

case '-': //sub

case '\*': //mul

case '!': //XOR

case '&': //Bitwise AND

case '%': //Bitwise OR

case '#': //Not Equal/Immediate Data

case '@': //Address of

case '^': //Pointer dereference

case ',':

TokenValue = Token(c);

Loop = false;

break;

case '>': //Greader Than

c = LexGet();

if (c == '=')

TokenValue = Token::GTEQ;

else

{

TokenValue = Token('>');

LexUnGet(c);

}

Loop = false;

break;

case '<': //Less Than

c = LexGet();

if (c == '>')

TokenValue = Token('#');

else if (c == '=')

TokenValue = Token::LTEQ;

else

{

TokenValue = Token('<');

LexUnGet(c);

}

Loop = false;

break;

case '.':

c1 = LexLook(1);

c2 = LexLook(2);

if (!IsValidNameChar(c2))

{

switch (c1)

{

case 'A': //accumulator

TokenValue = Token::AREG;

m\_LexBuffIndex++;

break;

case 'P': //proecssor status

TokenValue = Token::PSREG;

m\_LexBuffIndex++;

break;

case 'S': // stack pointer

TokenValue = Token::SPREG;

m\_LexBuffIndex++;

break;

case 'X': // X index register

TokenValue = Token::XREG;

m\_LexBuffIndex++;

break;

case 'Y': // Y index register

TokenValue = Token::YREG;

m\_LexBuffIndex++;

break;

case 'C': // carry flag

TokenValue = Token::CARRY;

m\_LexBuffIndex++;

break;

case 'D':

TokenValue = Token::DECIMAL\_MODE;

m\_LexBuffIndex++;

break;

case 'I': // interrupt flag

TokenValue = Token::IRQENABLE;

m\_LexBuffIndex++;

break;

case 'N': // negative flag

TokenValue = Token::NEG;

m\_LexBuffIndex++;

break;

case 'O':

TokenValue = Token::OVERFLOW;

m\_LexBuffIndex++;

break;

default:

TokenValue = Token('.');

break;

}

Loop = false;

}

else

{

TokenValue = Token('.');

}

Loop = false;

break;

case '/':

if (LexLook(1) == '/') //comment?

{

auxLoop = true;

while (auxLoop)

{

c = LexGet();

if (c == '\n')

{

auxLoop = false;

m\_Line++;

m\_Col = 0;

}

}

}

else

{

TokenValue = Token(c);

Loop = false;

}

break;

case ';': // Comment

auxLoop = true;

while (auxLoop)

{

c = LexGet();

if (c == '\n')

{

auxLoop = false;

m\_Line++;

m\_Col = 0;

}

}

break;

default: //Keywords and Identifiers

m\_aLexBuff[m\_LexBuffIndex++] = c;

auxLoop = true;

while (auxLoop)

{

c = LexGet();

if (IsValidNameChar(c))

{

m\_aLexBuff[m\_LexBuffIndex++] = c;

}

else

{

auxLoop = false;

m\_aLexBuff[m\_LexBuffIndex] = 0;

LexUnGet(c);

}

} //END OF collecting characters for word

//---------------------------------

// First check to see if it is a

// Keyword

//---------------------------------

TokenValue = LookupKeyword(m\_aLexBuff);

if (int(TokenValue))

{

//--------------------------

// Keyword has matched

//--------------------------

Loop = 0;

}

else

{

//-------------------------------------

// Is it a Identifier?

//-------------------------------------

m\_pLexSymbol = (CSymbol\*)LookupSymbol(m\_aLexBuff);

if (m\_pLexSymbol)

{

TokenValue = Token::IDENT;

Loop = false;

}

else

{

//-------------------------------------

// Identifier is New/Undefined

//-------------------------------------

m\_pLexSymbol = new CSymbol;

m\_pLexSymbol->Create();

TokenValue = Token::IDENT;

m\_pLexSymbol->SetIdentType(IdentType::NEW\_SYMBOL);

m\_pLexSymbol->SetName(GetLexBuffer());

Loop = false;

}

}

break; // end of default:

}

}

return TokenValue;

}

CBin\* CLexer::LookupSymbol(const char\* pName)

{

return GetSymTab()->FindSymbol(pName,1);

}

Token CLexer::LookupKeyword(const char\* pKeyword)

{

Token Toke;

Toke = KeyWords->LookupToToken(pKeyword);

return Toke;

}

CLexer::KeyWord\* CLexer::FindKeyword(Token KeywordToken)

{

return 0;

}

Processor CLexer::LookupProcessor(Token KeywordToken)

{

Processor AppropriateProcessor = Processor::ALL;

int i = 0;

bool Loop = true;

while (Loop)

{

if (KeyWords[i].m\_TokenID == KeywordToken)

{

AppropriateProcessor = KeyWords[i].m\_Processor;

Loop = 0;

}

else

{

++i;

if (KeyWords[i].m\_TokenID == Token::ENDOFTOKENS)

{

Loop = false;

}

}

}

return AppropriateProcessor;

}

int CLexer::LookupOpcode(Token OpcodeToken)

{

int OpCode = 0;

int i;

bool Loop = true;

for (i = 0; KeyWords[i].m\_OpCode >= 0 && Loop; ++i)

{

if (KeyWords[i].m\_TokenID == OpcodeToken)

{

Loop = false;

OpCode = KeyWords[i].m\_OpCode;

}

}

return OpCode;

}

int CLexer::GetOpcode(Token OpCodeToken)

{

int OpCode = -1; //indicates error

int i = 0;

bool Loop = true;

while (KeyWords[i].m\_TokenID != Token::ENDOFTOKENS && Loop)

{

if (KeyWords[i].m\_TokenID == OpCodeToken)

{

OpCode = KeyWords[i].m\_OpCode;

Loop = false;

}

else

++i;

}

return OpCode;

}

const char\* CLexer::KeyWord::LookupToName(Token Toke)

{

bool Loop = true;

Token T = Token(0);

int i = 0;

const char\* S =0;

while (Loop)

{

T = KeyWords[i].m\_TokenID;

if (Toke == T)

{

Loop = false;

S = KeyWords[i].m\_Name;

}

else if (T == Token::ENDOFTOKENS)

{

Loop = false;

S = 0;

}

else

{

++i;

}

}

return S;

}

Token CLexer::KeyWord::LookupToToken(const char\* pName)

{

int i = 0;

bool Loop = true;

while (Loop)

{

if (strcmp(pName, KeyWords[i].m\_Name) == 0)

Loop = false;

else

i++;

if (KeyWords[i].m\_TokenID == Token::ENDOFTOKENS)

{

Loop = false;

}

}

return KeyWords[i].m\_TokenID;

}

int CLexer::KeyWord::FindInc(AdrModeType AdrMode)

{

int IncValue = -1;

IncValue = m\_pAddresModeLUT->GetInc(AdrMode);

if (IncValue < 0)

{

sprintf\_s(

ThrownException.GetErrorString(),

ThrownException.GetMaxStringLen(),

"Houston, we have a problem Line:%d",

Act()->GetParser()->GetLexer()->GetLineNumber()

);

ThrownException.SetXCeptType(Exception::ExceptionType::INTERNAL\_ERROR);

throw(ThrownException);

}

return IncValue;

}

---------- LHead.cpp -------------

#include "pch.h"

CLHead::CLHead()

{

m\_Token = Token(0);

m\_pNode = 0;

}

CLHead::~CLHead()

{

}

bool CLHead::Create()

{

return false;

}

void CLHead::operator=(const CLHead& pLH)

{

m\_Token = pLH.m\_Token;

m\_pNode = pLH.m\_pNode;

}

---------- NumberStackItem.cpp -------------

#include "pch.h"

CNumberStackItem::CNumberStackItem()

{

m\_Value = 0;

}

CNumberStackItem::~CNumberStackItem()

{

}

bool CNumberStackItem::Create()

{

return CStackItem::Create(CStackItem::ItemType::INTVALUE);

}

---------- ObjFormatCode.cpp -------------

#include "pch.h"

CObjFormatCode::CObjFormatCode()

{

m\_BlockSize = 0;

m\_pCode = 0;

m\_StartAddress = 0;

}

CObjFormatCode::~CObjFormatCode()

{

if (m\_pCode) delete m\_pCode;

}

bool CObjFormatCode::Create(int StartAddress, int BlockSize, const char\* pCode)

{

m\_pCode = new char[BlockSize];

m\_BlockSize = (unsigned short)BlockSize;

m\_StartAddress = (unsigned short)StartAddress;

memcpy(m\_pCode, pCode, BlockSize);

return CObjFormatSectionItem::Create(CObjFormatSectionItem::FormatType::CODE);

}

void CObjFormatCode::Out(FILE\* pOut)

{

}

void CObjFormatCode::Print(FILE\* pO, int Indent)

{

}

---------- ObjFormatData.cpp -------------

#include "pch.h"

CObjFormatData::CObjFormatData()

{

m\_BlockSize = 0;

m\_Index = 0;

m\_pData = 0;

}

CObjFormatData::~CObjFormatData()

{

if (m\_pData) delete[] m\_pData;

}

bool CObjFormatData::Create(int BlockSize)

{

m\_BlockSize = BlockSize;

m\_pData = new char[BlockSize];

return CObjFormatSectionItem::Create(CObjFormatSectionItem::FormatType::DATA);

}

void CObjFormatData::Out(FILE\* pOut)

{

}

void CObjFormatData::Print(FILE\* pO, int Indent)

{

}

bool CObjFormatData::AddByte(char d)

{

bool rV = true;

if (m\_Index + 1 < m\_BlockSize)

{

m\_pData[m\_Index++] = d;

}

else

rV = false;

return 0;

}

---------- ObjFormatFile.cpp -------------

#include "pch.h"

CObjFormatFile::CObjFormatFile()

{

int i;

for (i = 0; i < 512; ++i)

{

m\_strName[i] = 0;

}

m\_pHead = 0;

m\_pTail = 0;

}

CObjFormatFile::~CObjFormatFile()

{

}

bool CObjFormatFile::Create(const char\* pName)

{

bool rV = true;

int NumberOfSections;

CSection\*\* ppSections;

int i;

fprintf(Act()->GetParser()->LogFile(), "File:%s\n", pName);

strcpy\_s(m\_strName, 512, pName);

NumberOfSections = GetSymTab()->GetSectionCount();

ppSections = new CSection \* [NumberOfSections];

GetSymTab()->FindAll(CBin::BinType::SECTION, GetSymTab()->GetSectionCount(),(CBin\*\*) ppSections);

for (i = 0; i < NumberOfSections; ++i)

{

AddSection(ppSections[i]);

}

Print(Act()->LogFile());

return rV;

}

CSymTab\* CObjFormatFile::GetSymTab()

{

return Act()->GetParser()->GetLexer()->GetSymTab();

}

void CObjFormatFile::Out(FILE\* pOut)

{

FILE\* pLog;

pLog = Act()->LogFile();

Print(pLog);

fprintf(pLog, "Number of section:%d\n", GetSymTab()->GetSectionCount());

fprintf(pLog,"Number of Symbols:%d", GetSymTab()->GetSymbolCount());

}

void CObjFormatFile::Print(FILE\* pO)

{

CObjFormatSection\* pOFS;

fprintf(pO, "Name:%s\n" ,m\_strName);

pOFS = GetHead();

while (pOFS)

{

pOFS->Print(pO, 2);

pOFS = pOFS->GetNext();

}

}

void CObjFormatFile::AddSection(CSection\* pSection)

{

CObjFormatSection\* pOFS;

pOFS = new CObjFormatSection;

pOFS->Create(pSection);

AddObjFormatSection(pOFS);

}

void CObjFormatFile::AddObjFormatSection(CObjFormatSection\* pOFS)

{

if (GetHead())

{

GetTail()->SetNext(pOFS);

pOFS->SetPrev(GetTail());

SetTail(pOFS);

}

else

{

SetTail(pOFS);

SetHead(pOFS);

}

}

---------- ObjFormatRelocation.cpp -------------

#include "pch.h"

CObjFormatRelocation::CObjFormatRelocation()

{

}

CObjFormatRelocation::~CObjFormatRelocation()

{

}

bool CObjFormatRelocation::Create()

{

return 0;

}

void CObjFormatRelocation::Out(FILE\* pO)

{

}

void CObjFormatRelocation::Print(FILE\* pO, int Indent)

{

}

---------- ObjFormatSection.cpp -------------

#include "pch.h"

CObjFormatSection::CObjFormatSection()

{

m\_pSection = 0;

m\_pNext = 0;

m\_pPrev = 0;

m\_pHead = 0;

m\_pTail = 0;

}

CObjFormatSection::~CObjFormatSection()

{

}

bool CObjFormatSection::Create(CSection\* pSection)

{

bool rV = true;

m\_pSection = pSection;

//--------------------------------

// Create a list of locations

// that need to be updated in

// this section

//--------------------------------

return rV;

}

void CObjFormatSection::AddItem(CObjFormatSectionItem\* pItem)

{

if (m\_pHead)

{

m\_pTail->SetNext(pItem);

pItem->SetPrev(m\_pTail);

m\_pTail = pItem;

}

else

{

m\_pTail = pItem;

m\_pHead = pItem;

}

}

void CObjFormatSection::Print(FILE\* pO, int Indent)

{

char\* pStr = new char[256];

fprintf(pO, "%sSection:%s\n",

Act()->IndentString(pStr,Indent,' '),

GetSection()->GetName()

);

delete[] pStr;

}

void CObjFormatSection::Out(FILE\* pO)

{

}

---------- ObjFormatSectionItem.cpp -------------

#include "pch.h"

CObjFormatSectionItem::CObjFormatSectionItem()

{

m\_Type = FormatType::NONE;

m\_pNext = 0;

m\_pPrev = 0;

}

CObjFormatSectionItem::~CObjFormatSectionItem()

{

}

bool CObjFormatSectionItem::Create(FormatType Type)

{

bool rV = true;

m\_Type = Type;

return rV;

}

void CObjFormatSectionItem::Out(FILE\* pOut)

{

}

void CObjFormatSectionItem::Print(FILE\* pO, int Indent)

{

}

const char\* CObjFormatSectionItem::LookupFormatSectionType(FormatType Type)

{

const char\* pStr = 0;;

int i;

bool Loop = true;

for (i = 0; FormatStringLUT[i].m\_pLutString && Loop; ++i)

{

if (Type == FormatStringLUT[i].m\_LutType)

{

pStr = FormatStringLUT[i].m\_pLutString;

Loop = false;

}

}

return pStr;

}

CObjFormatSectionItem::FormatType CObjFormatSectionItem::LookupFormatSectionType(const char\* pName)

{

FormatType Type = FormatType::NONE;

int i;

bool Loop = true;

for (i = 0; FormatStringLUT[i].m\_pLutString && Loop; ++i)

{

if (strcmp(FormatStringLUT[i].m\_pLutString, pName) == 0)

{

Type = FormatStringLUT[i].m\_LutType;

Loop = false;

}

}

return Type;

}

---------- ObjFormatSymbolItem.cpp -------------

#include "pch.h"

CObjFormatSymbolItem::CObjFormatSymbolItem()

{

int i;

m\_DefinedAddress = 0;

for (i = 0; i < MAX\_SYMBOL\_NAME\_LEN; ++i)

m\_strName[i] = 0;

}

CObjFormatSymbolItem::~CObjFormatSymbolItem()

{

}

bool CObjFormatSymbolItem::Create(CSymbol\* pSym)

{

return 0;

}

void CObjFormatSymbolItem::Out(FILE\* pOut)

{

}

void CObjFormatSymbolItem::Print(FILE\* pO)

{

}

---------- ObjFormatSymbols.cpp -------------

#include "pch.h"

CObjFormatSymbols::CObjFormatSymbols()

{

}

CObjFormatSymbols::~CObjFormatSymbols()

{

}

bool CObjFormatSymbols::Create()

{

return 0;

}

void CObjFormatSymbols::Print(FILE\* pO, int Indent)

{

}

void CObjFormatSymbols::Out(FILE\* pO)

{

}

void CObjFormatSymbols::Add(CObjFormatSymbolItem pSym)

{

}

---------- ObjType.cpp -------------

#include "pch.h"

CObjType::CObjType()

{

}

CObjType::~CObjType()

{

}

bool CObjType::Create()

{

return false;

}

---------- Parser.cpp -------------

#include "pch.h"

CParser::CParser()

{

m\_Phase = PHASE::NONE;

m\_pLex = 0;

m\_Processor = Processor::R6502;

m\_pCurrentSection = 0;

m\_Recursion = 0;

m\_Bump = 0;

}

CParser::~CParser()

{

if (m\_pLex)

delete m\_pLex;

}

bool CParser::Create()

{

bool rV = true;

m\_pLex = new CLexer;

m\_pLex->Create();

if (LogFile())

fprintf(LogFile(), "Parser Created\n");

return true;

}

FILE\* CParser::LogFile()

{

return Act()->LogFile();

}

CLHead CParser::Run()

{

CLHead LookaHead = CLHead(Token(0),0);

FILE\* ErrorDest = 0;

if (LogFile())

ErrorDest = LogFile();

else

ErrorDest = stderr;

try {

LookaHead.m\_Token = GetLexer()->Lex();

LookaHead = Action65(LookaHead);

GetLexer()->GetSymTab()->PrintTable(LogFile());

// GetAstTree()->Print(LogFile());

fprintf(LogFile(), "Lines Compiled:%d\n", GetLexer()->GetLineNumber());

}

catch (Exception& BooBoo)

{

char\* s = new char[256];

Exception::ExceptionType ExcptType;

ExcptType = BooBoo.GetXCeptType();

switch (ExcptType)

{

case Exception::ExceptionType::WHOKNOWS:

fprintf(ErrorDest,

"%s Line=%d Col=%d\n",

BooBoo.GetExceptionTypeString(BooBoo.GetXCeptType()),

GetLexer()->GetLineNumber(),

GetLexer()->GetColunm()

);

break;

case Exception::ExceptionType::UNEXPECTED\_TOKEN:

fprintf(ErrorDest,

"%s %d:%s Line:%d Col:%d\n",

BooBoo.GetExceptionTypeString(BooBoo.GetXCeptType()),

int(BooBoo.GetGotToken()),

GetLexer()->GetKeyWords()->LookupToName(BooBoo.GetGotToken()),

GetLexer()->GetLineNumber(),

GetLexer()->GetColunm()

);

break;

case Exception::ExceptionType::SECTION\_ADDRES\_RANGE\_EXCEEDED:

fprintf(ErrorDest,

"Section:Address Range Exceeded %d:%s\n%s\n Line:%d Col:%d\n",

int(BooBoo.GetGotToken()),

GetLexer()->GetKeyWords()->LookupToName(BooBoo.GetGotToken()),

BooBoo.GetErrorString(),

GetLexer()->GetLineNumber(),

GetLexer()->GetColunm()

);

break;

case Exception::ExceptionType::SECTION\_UNDEFINED:

fprintf(ErrorDest,

"Section Undefined %d:%s Line:%d Col:%d\n",

int(BooBoo.GetGotToken()),

GetLexer()->GetKeyWords()->LookupToName(BooBoo.GetGotToken()),

GetLexer()->GetLineNumber(),

GetLexer()->GetColunm()

);

break;

case Exception::ExceptionType::NOSECTION\_DEFINED:

fprintf(ErrorDest,

"No Section Defined %d:%s Line:%d Col:%d\n",

int(BooBoo.GetGotToken()),

GetLexer()->GetKeyWords()->LookupToName(BooBoo.GetGotToken()),

GetLexer()->GetLineNumber(),

GetLexer()->GetColunm()

);

break;

case Exception::ExceptionType::LEXER\_STUMPTED:

fprintf(

ErrorDest,

"%s",

BooBoo.GetErrorString()

);

break;

case Exception::ExceptionType::ILLEGAL\_ADDRESSING\_MODE:

fprintf(ErrorDest,

"%s",

BooBoo.GetErrorString()

);

break;

case Exception::ExceptionType::INTERNAL\_ERROR:

fprintf(

ErrorDest,

"Internal Error:%s Line:%d Col:%d\n",

BooBoo.GetErrorString(),

GetLexer()->GetLineNumber(),

GetLexer()->GetColunm()

);

break;

case Exception::ExceptionType::STACK:

fprintf(

ErrorDest,

"Error:%s\n",

BooBoo.GetErrorString()

);

break;

default:

fprintf(ErrorDest, "Unknown Exception\n");

break;

}

Act()->CloseAll();

exit(1);

}

return LookaHead;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Expect

//

// This function is used to check that we get

// the token that we Expect.

//

// parameters:

// Lookahead..Current lookahead token

// token....this is the token we Expect

// return value:

// reutns the next Lookahead token (>0)

// reutnrs 0 or negative if we did not get what we Expected

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Token CParser::Expect(Token LookaHeadToken, Token Expected)

{

char\* pExpectedToken = 0;

char\* pLookaheadToken = 0;

int number = 0;

if (LogFile())

{

switch (LookaHeadToken)

{

case Token::IDENT:

pLookaheadToken = GetLexer()->GetLexSymbol()->GetName();

if (LookaHeadToken == Expected)

{

pExpectedToken = pLookaheadToken;

}

else

pExpectedToken = (char\*)"Unexpected";

fprintf(LogFile(), "%d::Expected CLHead: %s Lookahead = %s Line:%d\n",

m\_Recursion,

pExpectedToken,

pLookaheadToken,

GetLexer()->GetLineNumber()

);

break;

case Token::NUMBER:

number = GetLexer()->GetNumber();;

pExpectedToken = (char\*)GetLexer()->GetKeyWords()->LookupToName(Expected);

fprintf(LogFile(), "%d::Expected CLHead: %s Lookahead = %d Line:%d\n",

m\_Recursion,

pExpectedToken,

number,

GetLexer()->GetLineNumber()

);

break;

default:

pLookaheadToken = (char\*)GetLexer()->GetKeyWords()->LookupToName(LookaHeadToken);

pExpectedToken = (char\*)GetLexer()->GetKeyWords()->LookupToName(Expected);

fprintf(LogFile(), "%d::Expected CLHead: %s LookaHeadToken = %s Line:%d\n",

m\_Recursion,

pExpectedToken,

pLookaheadToken,

GetLexer()->GetLineNumber()

);

break;

}

}

if (Accept(LookaHeadToken, Expected))

LookaHeadToken = GetLexer()->Lex();

else

{

ThrownException.SetXCeptType(Exception::ExceptionType::UNEXPECTED\_TOKEN);

sprintf\_s(

ThrownException.GetErrorString(),

ThrownException.GetMaxStringLen(),

"Line %d: Unexpected CLHead:Got %d Expected %d\n",

GetLexer()->GetLineNumber(),

LookaHeadToken,

Expected

);

throw(ThrownException);

}

return LookaHeadToken;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Accept

//

// This function compares the token you want

// versus the token that is current. If they

// match, then we get another token.

// If not, then just return.

//

// parameter:

// Lookahead..The current lookahead token

// token.....this is the token we want

//

// return value:

// returns the new token value (>0)

// returns 0 or negative if we don't get the token we want

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

bool CParser::Accept(Token LookaheadToken, Token Expected)

{

bool rv = false;

if (Expected == LookaheadToken)

rv = true;

return rv;

}

void CParser::PrepareInstruction(CInstruction\*\* ppInst, Token Op)

{

\*ppInst = new CInstruction;

(\*ppInst)->Create();

(\*ppInst)->SetOpCodeToken(Op);

(\*ppInst)->SetOpcode(

GetLexer()->LookupOpcode(Op)

);

(\*ppInst)->SetByteCount(

GetLexer()->FindKeyword(Op)->m\_MaxBytes

);

}

//---------------------------------

// Parsing Methods

//---------------------------------

// Program Structure

//---------------------------------

CLHead CParser::Action65(CLHead LookaHead)

{

//--------------------------------------------

// Action65->Modules;

//

//--------------------------------------------

PrintLookahead(LogFile(), LookaHead, "Enter Action65", ++m\_Recursion);

LookaHead = Modules(LookaHead);

PrintLookahead(LogFile(), LookaHead, "Exit Action65", --m\_Recursion);

return LookaHead;

}

CLHead CParser::Modules(CLHead LookaHead)

{

//--------------------------------------------

// Modules -> Vector Modules\_1;

//--------------------------------------------

// Modules\_1 -> 'MODULE' Vector Modules\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN = 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Modules", ++m\_Recursion);

LHChild = Vector(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::MODULE:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::MODULE);

LHNext = Vector(LHNext);

pN = new CAct65Module;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

case Token::EOL:

Loop = false;

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit Module", --m\_Recursion);

return LHChild;

}

CLHead CParser::Vector(CLHead LookaHead)

{

//--------------------------------------------

// Vector ->SysDecl Vector\_1;

// Vector\_1 -> 'VECTOR' VectorAddress SysDecl Vector\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN = 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Vector", ++m\_Recursion);

LHChild = SysDecl(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::VECTOR:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::VECTOR);

LHNext = VectorAddress(LHNext);

pN = new CAct65VECTOR;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

//--------------------------------------

LHChild = SysDecl(LHChild);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit Vector", --m\_Recursion);

return LHChild;

}

CLHead CParser::VectorAddress(CLHead LookaHead)

{

//--------------------------------------------

// VectorAddress -> '(' CompConst ')' VectorData;

// VectorData -> '=' CompConst;

//--------------------------------------------

CAstNode\* pN = 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter VectorAddress", ++m\_Recursion);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('('));

LHChild = CompConst(LookaHead); // Vector Address

LHNext.m\_Token = Expect(LHChild.GetToken(), Token(')'));

LHNext.m\_Token = Expect(LHNext.GetToken(), Token('='));

LHNext = CompConst(LHNext); // Vector Data

pN = new CAct65VECTOR;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.m\_Token = LHNext.GetToken();

PrintLookahead(LogFile(), LookaHead, "Exit VectorAddress", --m\_Recursion);

return LHChild;

}

//------------------------------------------

// Statements

//------------------------------------------

CLHead CParser::Statements(CLHead LookaHead)

{

//--------------------------------------------

// Statements ->ForStmt ProcCall\_1;

//--------------------------------------------

// ProcCall\_1 -> 'PROC\_IDENT' ProcParams ForStmt ProcCall\_1

// -> 'FUNC\_IDENT' ProcParams ForStmt ProcCall\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN = 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Statements", ++m\_Recursion,1);

LHChild = ForStmt(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::PROC\_IDENT:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::PROC\_IDENT);

LHNext = ProcParams(LHNext);

pN = new CAct65ProcCall;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

//--------------------------------------------

LHChild = ForStmt(LHChild);

break;

case Token::FUNC\_IDENT:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::FUNC\_IDENT);

LHNext = ProcParams(LookaHead);

pN = new CAct65FuncCall;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

//-----------------------------------------

LookaHead = ForStmt(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit Statements", --m\_Recursion,-1);

return LHChild;

}

CLHead CParser::ProcParams(CLHead LookaHead)

{

//--------------------------------------------

// ProcParams -> '(' ValueList ')'

// -> .

// ;

//--------------------------------------------

CAstNode\* pN = 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter ProcParams", ++m\_Recursion);

switch (LHChild.GetToken())

{

case Token('('):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('('));

LHNext = MemContents(LookaHead);

pN = new CAct65FuncCall;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

LHChild.m\_Token = Expect(LHNext.GetToken(), Token(')'));

break;

default:

break;

}

PrintLookahead(LogFile(), LHChild, "Exit ProcParams", --m\_Recursion);

return LHChild;

}

CLHead CParser::ForStmt(CLHead LookaHead)

{

//--------------------------------------------

// ForStmt -> IfStmt ForStmt\_1;

// ForStmt\_1 -> 'FOR' STEPoption DoStmt ForStmt\_1

// - > .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN = 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter ForStmt", ++m\_Recursion);

LHChild = IfStmt(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::FOR:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::FOR);

LookaHead = STEPoption(LookaHead);

LookaHead = DoStmt(LookaHead);

pN = new CAct65FOR;;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

//-------------------------------------------------

LHChild = IfStmt(LHChild);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LHChild, "Exit ForStmt", --m\_Recursion);

return LHChild;

}

CLHead CParser::STEPoption(CLHead LookaHead)

{

//--------------------------------------------

// STEPoption ->ForTO STEPoption\_1;

// STEPoption\_1 -> 'STEP' ArithExpr

// -> .

// ;

//--------------------------------------------

CAstNode\* pN = 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Step", ++m\_Recursion);

LHChild = ForTO(LookaHead);

switch (LHChild.GetToken())

{

case Token::STEP:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::STEP);

LHNext = ArithExpr(LHNext);

pN = new CAct65ForSTEP;;;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

default:

break;

}

PrintLookahead(LogFile(), LHChild, "Exit Step", --m\_Recursion);

return LHChild;

}

CLHead CParser::ForTO(CLHead LookaHead)

{

//--------------------------------------------

// ForTO ->Itterrator ForTO\_1;

// ForTO\_1 -> 'TO' ArithExpr

// -> .

// ;

//--------------------------------------------

CAstNode\* pN = 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Finish", ++m\_Recursion);

LHChild = Iterator(LookaHead);

switch (LHChild.GetToken())

{

case Token::TO:

LookaHead.m\_Token = Expect(LHChild.GetToken(), Token::TO);

LHNext = ArithExpr(LHChild);

pN = new CAct65ForTO;;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit Finish", --m\_Recursion);

return LookaHead;

}

CLHead CParser::Iterator(CLHead LookaHead)

{

//--------------------------------------------

// Itterrator ->MemContentsType Itterrator\_1;

// Itterrator\_1 -> '=' ArithExpr

// -> .

// ;

//--------------------------------------------

CAstNode\* pN = 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Iterator", ++m\_Recursion);

LHChild = MemContentsType(LHChild);

switch (LHChild.GetToken())

{

case Token('='):

LHChild.m\_Token = Expect(LHChild.GetToken(), Token('='));

LHNext = ArithExpr(LHChild);

pN = new CAct65Assignment;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

default:

break;

}

PrintLookahead(LogFile(), LHChild, "Exit Itterator", --m\_Recursion);

return LHChild;

}

//-----------------------------------------------

// If Statement

//-----------------------------------------------

CLHead CParser::IfStmt(CLHead LookaHead)

{

//--------------------------------------------

// IfStmt -> Iff IfStmt\_1;

// IfStmt\_1 -> 'IF' EndIF Iff IfStmt\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter IfStmt", ++m\_Recursion);

LHChild = IffStmt(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::IF:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::IF);

LHNext = EndIf(LHNext);

pN = new CAct65IF;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

//-------------------------------------------

LHChild = IffStmt(LHChild);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LHChild, "Exit IfStmt", --m\_Recursion);

return LHChild;

}

CLHead CParser::EndIf(CLHead LookaHead)

{

//--------------------------------------------

// EndIF ->ElsePart EndIF\_1;

// EndIF\_1 -> 'FI';

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter EndIf", ++m\_Recursion);

LHChild = ElsePart(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::FI:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::FI);

pN = new CAct65FI;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

//------------------------------------------------

Loop = false;

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LHChild, "Exit EndIf(FI)", --m\_Recursion);

return LHChild;

}

CLHead CParser::ElsePart(CLHead LookaHead)

{

//--------------------------------------------

// ElsePart ->ElseIfPart ElsePart\_1;

// ElsePart\_1 -> 'ELSE' Statements

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter ElsePart", ++m\_Recursion);

LHChild = ElseIfPart(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::ELSE:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::ELSE);

LHNext = Statements(LHNext);

pN = new CAct65ELSE;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LHChild, "Exit ElsePart", --m\_Recursion);

return LHChild;

}

CLHead CParser::ElseIfPart(CLHead LookaHead)

{

//--------------------------------------------

//ElseIfPart ->ThenPart ElseIfPart\_1;

//ElseIfPart\_1 -> 'ELSEIF' ThenPart ElseIfPart\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter ElseIfPart", ++m\_Recursion);

LHChild = ThenPart(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::ELSEIF:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::THEN);

LHNext = ThenPart(LHNext);

pN = new CAct65ELSEIF;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

//----------------------------------------------------

LHChild = ElsePart(LHChild);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LHChild, "Exit ElseIfPart", --m\_Recursion);

return LHChild;

}

CLHead CParser::ThenPart(CLHead LookaHead)

{

//--------------------------------------------

// ThenPart ->RelOperation ThenPart\_1;

// ThenPart\_1 -> 'THEN' Statements

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Then Part", ++m\_Recursion);

LHChild = RelOperation(LookaHead);

switch (LHChild.GetToken())

{

case Token::THEN:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::THEN);

LHNext = Statements(LHNext);

pN = new CAct65THEN;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

default:

break;

}

PrintLookahead(LogFile(), LHChild, "Exit Then Part", --m\_Recursion);

return LHChild;

}

//------------------------------------------------

// IFF Statement

//------------------------------------------------

CLHead CParser::IffStmt(CLHead LookaHead)

{

//--------------------------------------------

// Iff ->WhileStmt Iff\_1;

// Iff\_1 -> 'IFF' IFFend WhileStmt Iff\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter IFFStmt", ++m\_Recursion);

LHChild = WhileStmt(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::IFF:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::IFF);

LHNext = IFFend(LHNext);

pN = new CAct65IFF;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

//-------------------------------------------

LHChild = WhileStmt(LHChild);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LHChild, "Exit IFFStmt", --m\_Recursion);

return LHChild;

}

CLHead CParser::IFFend(CLHead LookaHead)

{

//--------------------------------------------

// IFFend ->IFFelse IFFend\_1;

// IFFend\_1 -> 'FFI';

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter IFFend", ++m\_Recursion);

LHChild = IFFelse(LookaHead);

switch (LHChild.GetToken())

{

case Token::FFI:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::FFI);

pN = new CAct65IFF;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

Loop = false;

break;

default:

Loop = false;

break;

}

PrintLookahead(LogFile(), LHChild, "Exit IFFend", --m\_Recursion);

return LHChild;

}

CLHead CParser::IFFelse(CLHead LookaHead)

{

//--------------------------------------------

// IFFelse ->IFFthenpart IFFelse\_1;

// IFFelse\_1 -> 'ELSE' Statements

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter IFFelse", ++m\_Recursion);

LHChild = IFFthenpart(LookaHead);

switch (LHChild.GetToken())

{

case Token::ELSE:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::ELSE);

LHNext = Statements(LHNext);

pN = new CAct65ELSE;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

default:

break;

}

PrintLookahead(LogFile(), LHChild, "Exit IFFelse", --m\_Recursion);

return LHChild;

}

CLHead CParser::IFFthenpart(CLHead LookaHead)

{

//--------------------------------------------

// IFFthenpart ->IffConditional IFFthenpart\_1;

// IFFthenpart\_1 -> 'THEN' Statements

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter IFFthenpart", ++m\_Recursion);

LHChild = IffConditional(LookaHead);

switch (LHChild.GetToken())

{

case Token::THEN:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::THEN);

LookaHead = Statements(LookaHead);

pN = new CAct65THEN;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

//---------------------------------------------

LHChild = Statements(LHChild);

break;

default:

break;

}

PrintLookahead(LogFile(), LHChild, "Exit IFFthenpart", --m\_Recursion);

return LHChild;

}

CLHead CParser::IffConditional(CLHead LookaHead)

{

//--------------------------------------------

// IffConditional ->Bits IffConditional\_1;

// IffConditional\_1 -> 'AREG' RelOper

// -> 'XREG' RelOper

// -> 'YREG' RelOper

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter IffConditional", ++m\_Recursion);

LHChild = Bits(LookaHead);

switch (LHChild.GetToken())

{

case Token::AREG:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::AREG);

LHNext = RelOper(LHNext);

pN = new CAct65ACC;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

case Token::XREG:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::XREG);

LHNext = RelOper(LHNext);

pN = new CAct65XREG;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

case Token::YREG:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::YREG);

LHNext = RelOper(LHNext);

pN = new CAct65YREG;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

default:

break;

}

PrintLookahead(LogFile(), LHChild, "Exit IffConditional", --m\_Recursion);

return LHChild;

}

CLHead CParser::RelOper(CLHead LookaHead)

{

//--------------------------------------------

// RelOper -> '<' Value

// -> 'GTEQ' Value

// -> '=' Value

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter RelOper", ++m\_Recursion);

switch (LHChild.GetToken())

{

case Token('<'):

LHNext.m\_Token = Expect(LHChild.GetToken(), Token('<'));

LHNext = Value(LHNext);

pN = new CAct65LessTHAN;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

case Token::GTEQ:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::GTEQ);

LHNext = Value(LHNext);

pN = new CAct65GTEQ;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

case Token('='):

LHNext.m\_Token = Expect(LHChild.GetToken(), Token('='));

LHNext = Value(LHNext);

pN = new CAct65EqualTO;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

default:

break;

}

PrintLookahead(LogFile(), LHChild, "Exit RelOper", --m\_Recursion);

return LHChild;

}

CLHead CParser::Bits(CLHead LookaHead)

{

//--------------------------------------------

// Bits ->StatusFlags Bits\_1;

// Bits\_1 -> 'BITS' BitValue

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN = 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Bits", ++m\_Recursion);

switch (LHChild.GetToken())

{

case Token::BITS:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::BITS);

LookaHead = BitValue(LookaHead);

break;

default:

Loop = false;

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit Bits", --m\_Recursion);

return LHChild;

}

CLHead CParser::BitValue(CLHead LookaHead)

{

//--------------------------------------------

// BitValue -> '[' ArithExpr ']';

//--------------------------------------------

CAstNode\* pN = 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter BitValue", ++m\_Recursion);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('['));

LHChild = ArithExpr(LookaHead);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(']'));

PrintLookahead(LogFile(), LookaHead, "Exit BitValue", --m\_Recursion);

return LHChild;

}

CLHead CParser::StatusFlags(CLHead LookaHead)

{

//--------------------------------------------

// StatusFlags ->OptNot StatusFlags\_1;

// StatusFlags\_1 -> 'NEG' //negative flag

// -> 'CARRY' //Carry flag

// -> 'ZERO' //Zero

// -> 'OVERFLOW' //Overflow

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter StatusFlags", ++m\_Recursion);

LHChild = OptNot(LookaHead);

switch (LHChild.GetToken())

{

case Token::NEG:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::NEG);

break;

case Token::ZERO:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::ZERO);

break;

case Token::CARRY:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::CARRY);

break;

case Token::OVERFLOW:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::OVERFLOW);

break;

default:

break;

}

PrintLookahead(LogFile(), LHChild, "Exit StatusFlags", --m\_Recursion);

return LHChild;

}

CLHead CParser::OptNot(CLHead LookaHead)

{

//--------------------------------------------

// OptNot -> '^'

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter OptNot", ++m\_Recursion);

switch (LookaHead.GetToken())

{

case Token('^'):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('^'));

pN = new CAct65OptNOT;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LookaHead.m\_pNode = pN;

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit OptNot", --m\_Recursion);

return LookaHead;

}

//-------------------------------------------------

// WHILE Statement

//-------------------------------------------------

CLHead CParser::WhileStmt(CLHead LookaHead)

{

//--------------------------------------------

// WhileStmt -> DoStmt WhileStmt\_1;

// WhileStmt\_1 -> 'WHILE' RelOperation DoStmt WhileStmt\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter WhileStmt", ++m\_Recursion);

LHChild = DoStmt(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::WHILE:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::WHILE);

LookaHead = RelOperation(LookaHead);

LookaHead = DoStmt(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit WhileStmt", --m\_Recursion);

return LookaHead;

}

//------------------------------------------------

// DO Statement

//------------------------------------------------

CLHead CParser::DoStmt(CLHead LookaHead)

{

//--------------------------------------------

// DoStmt ->EXITstmt DoStmt\_1;

// DoStmt\_1 -> 'DO' DoEnd EXITstmt DoStmt\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter DO Stmt", ++m\_Recursion);

LHChild = EXITstmt(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::DO:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::DO);

LookaHead = Statements(LookaHead);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::OD);

LookaHead = EXITstmt(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit DoStmt", --m\_Recursion);

return LookaHead;

}

CLHead CParser::DoEND(CLHead LookaHead)

{

//--------------------------------------------

// DoEnd ->Statements DoEnd\_1;

// DoEnd\_1 -> 'OD'

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter DoEND", ++m\_Recursion);

LHChild = Statements(LookaHead);

switch (LHChild.GetToken())

{

case Token::OD:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::OD);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit DoEND", --m\_Recursion);

return LHChild;

}

//---------------------------------------------------

// EXIT Statement

//---------------------------------------------------

CLHead CParser::EXITstmt(CLHead LookaHead)

{

//--------------------------------------------

// EXITstmt -> RetStmt EXITstmt\_1;

// EXITstmt\_1 -> 'EXIT' EXITstmt\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter EXITstmt", ++m\_Recursion);

LHChild = RetStmt(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::EXIT:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::EXIT);

LookaHead = RetStmt(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit EXITstmt", --m\_Recursion);

return LookaHead;

}

//-----------------------------------------------

// RETURN statement

//-----------------------------------------------

CLHead CParser::RetStmt(CLHead LookaHead)

{

//--------------------------------------------

// RetStmt -> InlineAssembly RetStmt\_1;

// RetStmt\_1 -> 'RETURN' OptReturnValue InlineAssembly RetStmt\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter RetStmt", ++m\_Recursion);

LHChild = InlineAssembly(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::RETURN:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::RETURN);

LookaHead = OptReturnValue(LookaHead);

LookaHead = InlineAssembly(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit RetStmt", --m\_Recursion);

return LHChild;

}

CLHead CParser::OptReturnValue(CLHead LookaHead)

{

//--------------------------------------------

// OptReturnValue -> '(' ArithExpr OptReturnValue

// -> ')'

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter OptRetValue", ++m\_Recursion);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('('));

LookaHead = ArithExpr(LookaHead);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(')'));

PrintLookahead(LogFile(), LookaHead, "Exit OptReturnValue", --m\_Recursion);

return LookaHead;

}

//---------------------------------------------

// Inline Assembly Code

//---------------------------------------------

CLHead CParser::InlineAssembly(CLHead LookaHead)

{

//--------------------------------------------

// InlineAssembly ->CodeBlock InlineAssembly\_1;

// InlineAssembly\_1 -> 'ASM' InlineAssBlock CodeBlock InlineAssembly\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter InLineAssembly", ++m\_Recursion);

LHChild = CodeBlock(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::ASM:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::ASM);

LookaHead = InlineAssBlock(LookaHead);

LookaHead = CodeBlock(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit InLineAssembly", --m\_Recursion);

return LHChild;

}

CLHead CParser::InlineAssBlock(CLHead LookaHead)

{

//--------------------------------------------

// InlineAssBlock ->EndAsmBlock InlineAssBlock\_1;

// InlineAssBlock\_1 -> '{' EndAsmBlock InlineAssBlock\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter InLineAsmBlock", ++m\_Recursion);

LHChild = EndAsmBlock(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token('{'):

GetLexer()->SetAsmMode();

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('{'));

GetLexer()->SetAsmMode();

LookaHead = EndAsmBlock(LookaHead);

break;

default:

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit InLineAsmBlock", --m\_Recursion);

return LHChild;

}

CLHead CParser::EndAsmBlock(CLHead LookaHead)

{

//--------------------------------------------

// EndAsmBlock ->AsmStmt EndAsmBlock\_1;

// EndAsmBlock\_1 -> '}'

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter EndAsmBlock", ++m\_Recursion);

LHChild = AsmStmt(LookaHead);

switch (LHChild.GetToken())

{

case Token('}'):

GetLexer()->SetActionMode();

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('}'));

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit EndAsmBlock", --m\_Recursion);

return LHChild;

}

//-----------------------------------------------

// CODE (Kluge) BLOCK statement

//-----------------------------------------------

CLHead CParser::CodeBlock(CLHead LookaHead)

{

//--------------------------------------------

// CodeBlock ->UntillStmt CodeBlock\_1;

// CodeBlock\_1 -> '[' CodeBlockEnd UntillStmt CodeBlock\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter CodeBlock", ++m\_Recursion);

LHChild = UntillStmt(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token('['):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('['));

LookaHead = CodeBlockEnd(LookaHead);

LookaHead = UntillStmt(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit CodeBlock", --m\_Recursion);

return LHChild;

}

CLHead CParser::CodeBlockEnd(CLHead LookaHead)

{

//--------------------------------------------

// CodeBlockEnd ->ConstList CodeBlockEnd\_1;

// CodeBlockEnd\_1 -> ']'

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter CodeBlockEnd", ++m\_Recursion);

LHChild = ConstList(LookaHead);

switch (LHChild.GetToken())

{

case Token(']'):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(']'));

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit CodeBlockEnd", --m\_Recursion);

return LHChild;

}

//---------------------------------------------------

// UNTILE Statement

//---------------------------------------------------

CLHead CParser::UntillStmt(CLHead LookaHead)

{

//--------------------------------------------

// UntillStmt ->Push UntillStmt\_1;

// UntillStmt\_1 -> 'UNTILL' RelOperation Push UntillStmt\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter UntillStmt", ++m\_Recursion);

LHChild = Push(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::UNTIL:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::UNTIL);

LookaHead = RelOperation(LookaHead);

LookaHead = Push(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit UntilStmt", --m\_Recursion);

return LHChild;

}

//-------------------------------------------------

// PUSH statment

//-------------------------------------------------

CLHead CParser::Push(CLHead LookaHead)

{

//--------------------------------------------

// Push ->Pop Push\_1;

// Push\_1 -> 'PUSH' PushSourceList Pop Push\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Push", ++m\_Recursion);

LHChild = Pop(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::PUSH:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::PUSH);

LookaHead = PushSourceList(LookaHead);

LookaHead = Pop(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit Push", --m\_Recursion);

return LHChild;

}

CLHead CParser::PushSourceList(CLHead LookaHead)

{

//--------------------------------------------

// PushSourceList ->PushSource PushSourceList\_1;

// PushSourceList\_1 -> ',' PushSource PushSourceList\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter PushSourceList", ++m\_Recursion);

LHChild = PushSource(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token(','):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(','));

LookaHead = PushSource(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit PushSourceList", --m\_Recursion);

return LHChild;

}

CLHead CParser::PushSource(CLHead LookaHead)

{

//--------------------------------------------

// PushSource ->ArithExpr PushSource\_1;

// PushSource\_1 -> 'AREG'

// -> 'XREG'

// -> 'YREG'

// -> 'SR'

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter PushSource", ++m\_Recursion);

LHChild = ArithExpr(LookaHead);

switch (LHChild.GetToken())

{

case Token::AREG:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::AREG);

break;

case Token::XREG:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::XREG);

break;

case Token::YREG:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::YREG);

break;

case Token::PSREG: //processor status register

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::PSREG);

break;

default:

Loop = false;

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit PushSource", --m\_Recursion);

return LHChild;

}

//--------------------------------------------------

// POP Statement

//--------------------------------------------------

CLHead CParser::Pop(CLHead LookaHead)

{

//--------------------------------------------

// Pop ->Break Pop\_1;

// Pop\_1 -> 'POP' PopDestList Break Pop\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Pop", ++m\_Recursion);

LHChild = Break(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::POP:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::POP);

LookaHead = PopDestList(LookaHead);

LookaHead = Break(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit Pop", --m\_Recursion);

return LHChild;

}

CLHead CParser::PopDestList(CLHead LookaHead)

{

//--------------------------------------------

// PopDestList ->PopDest PopDestList\_1;

// PopDestList\_1 -> ',' PopDest PopDestList\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter PopDestList", ++m\_Recursion);

LHChild = PopDest(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token(','):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(','));

LookaHead = PopDest(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit PopDestList", --m\_Recursion);

return LHChild;

}

CLHead CParser::PopDest(CLHead LookaHead)

{

//--------------------------------------------

// PopDest ->MemContentsType PopDest\_1;

// PopDest\_1 -> 'AREG'

// -> 'XREG'

// -> 'YREG'

// -> 'PSREG'

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter PopDest", ++m\_Recursion);

LHChild = MemContentsType(LookaHead);

switch (LHChild.GetToken())

{

case Token::AREG:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::AREG);

break;

case Token::XREG:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::XREG);

break;

case Token::YREG:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::YREG);

break;

case Token::PSREG:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::PSREG);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit PopDest", --m\_Recursion);

return LHChild;

}

//--------------------------------------------------

// BREAK Statement

//--------------------------------------------------

CLHead CParser::Break(CLHead LookaHead)

{

//--------------------------------------------

// Break ->Rti Break\_1;

// Break\_1 -> 'BREAK' Rti Break\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Break", ++m\_Recursion);

LHChild = Rti(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::BREAK:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::BREAK);

LookaHead = Rti(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit Break", --m\_Recursion);

return LHChild;

}

//--------------------------------------------------

// RTI Statement

//--------------------------------------------------

CLHead CParser::Rti(CLHead LookaHead)

{

//--------------------------------------------

// Rti ->Assignment Rti\_1;

// Rti\_1 -> 'RTI' Assignment Rti\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Rti", ++m\_Recursion);

LHChild = Assignment(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::RTI:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::RTI);

LookaHead = Assignment(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit Rti", --m\_Recursion);

return LHChild;

}

//---------------------------------------------------

// Assignment Statements

//---------------------------------------------------

CLHead CParser::Assignment(CLHead LookaHead)

{

//--------------------------------------------

// Assignment -> MemContents Assignment\_1;

//--------------------------------------------

// Assignment\_1 -> '=' ArithExpr Assignment\_1

// -> '==+' ArithExpr Assignment\_1

// -> '==-' ArithExpr Assignment\_1

// -> '==\*' ArithExpr Assignment\_1

// -> '==/' ArithExpr Assignment\_1

// -> '==MOD' ArithExpr Assignment\_1

// -> '==&' ArithExpr Assignment\_1

// -> '==!' ArithExpr Assignment\_1 //XOR

// -> '==%' ArithExpr Assignment\_1 //OR

// -> '==LSH' ArithExpr Assignment\_1

// -> '==RSH' ArithExpr Assignment\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

CAstNode\* pChildNode = 0, \* pNextNode = 0;

union NodeTypes {

CAstNode\* m\_pAstNode;

CAct65AssignADD\* m\_pAssAdd;

CAct65AssignAND\* m\_pAssAnd;

CAct65AssignDIV\* m\_pASSDiv;

CAct65AssignLSh\* m\_pAssLSH;

CAct65Assignment\* m\_pAssignment;

CAct65AssignMOD\* m\_pAssMod;

CAct65AssignMULT\* m\_pAssMult;

CAct65AssignOR\* m\_pAssOr;

CAct65AssignRSH\* m\_pAssRSH;

CAct65AssignSUB\* m\_pAssSub;

CAct65AssignXOR\* m\_pAssXor;

}Nodes = NodeTypes(0);

PrintLookahead(LogFile(), LookaHead, "Enter Assignment", ++m\_Recursion);

LHChild = MemContents(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token('='):

pChildNode = LookaHead.GetNode();

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('='));

LookaHead = ArithExpr(LookaHead);

Nodes.m\_pAssignment = new CAct65Assignment;

pNextNode = LookaHead.GetNode();

Nodes.m\_pAssignment->CreateNode(pChildNode, pNextNode);

LookaHead.SetNode(Nodes.m\_pAstNode);

LookaHead = MemContents(LookaHead);

break;

case Token::ASSIGN\_ADD:

pChildNode = LookaHead.GetNode();

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::ASSIGN\_ADD);

LookaHead = ArithExpr(LookaHead);

Nodes.m\_pAssAdd = new CAct65AssignADD;

pNextNode = LookaHead.GetNode();

Nodes.m\_pAssignment->CreateNode(pChildNode, pNextNode);

LookaHead.SetNode(Nodes.m\_pAstNode);

LookaHead = MemContents(LookaHead);

break;

case Token::ASSIGN\_AND:

pChildNode = LookaHead.GetNode();

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::ASSIGN\_AND);

LookaHead = ArithExpr(LookaHead);

Nodes.m\_pAssAnd = new CAct65AssignAND;

pNextNode = LookaHead.GetNode();

Nodes.m\_pAssignment->CreateNode(pChildNode, pNextNode);

LookaHead.SetNode(Nodes.m\_pAstNode);

LookaHead = MemContents(LookaHead);

break;

case Token::ASSIGN\_DIV:

pChildNode = LookaHead.GetNode();

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::ASSIGN\_DIV);

LookaHead = ArithExpr(LookaHead);

Nodes.m\_pASSDiv = new CAct65AssignDIV;

pNextNode = LookaHead.GetNode();

Nodes.m\_pAssignment->CreateNode(pChildNode, pNextNode);

LookaHead.SetNode(Nodes.m\_pAstNode);

LookaHead = MemContents(LookaHead);

break;

case Token::ASSIGN\_LSH:

pChildNode = LookaHead.GetNode();

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::ASSIGN\_LSH);

LookaHead = ArithExpr(LookaHead);

Nodes.m\_pAssLSH = new CAct65AssignLSh;

pNextNode = LookaHead.GetNode();

Nodes.m\_pAssignment->CreateNode(pChildNode, pNextNode);

LookaHead.SetNode(Nodes.m\_pAstNode);

LookaHead = MemContents(LookaHead);

break;

case Token::ASSIGN\_MOD:

pChildNode = LookaHead.GetNode();

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::ASSIGN\_MOD);

LookaHead = ArithExpr(LookaHead);

Nodes.m\_pAssMod = new CAct65AssignMOD;

pNextNode = LookaHead.GetNode();

Nodes.m\_pAssignment->CreateNode(pChildNode, pNextNode);

LookaHead.SetNode(Nodes.m\_pAstNode);

LookaHead = MemContents(LookaHead);

break;

case Token::ASSIGN\_MUL:

pChildNode = LookaHead.GetNode();

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::ASSIGN\_MUL);

LookaHead = ArithExpr(LookaHead);

Nodes.m\_pAssMult = new CAct65AssignMULT;

pNextNode = LookaHead.GetNode();

Nodes.m\_pAssignment->CreateNode(pChildNode, pNextNode);

LookaHead.SetNode(Nodes.m\_pAstNode);

LookaHead = MemContents(LookaHead);

break;

case Token::ASSIGN\_OR:

pChildNode = LookaHead.GetNode();

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::ASSIGN\_OR);

LookaHead = ArithExpr(LookaHead);

Nodes.m\_pAssOr = new CAct65AssignOR;

pNextNode = LookaHead.GetNode();

Nodes.m\_pAssignment->CreateNode(pChildNode, pNextNode);

LookaHead.SetNode(Nodes.m\_pAstNode);

LookaHead = MemContents(LookaHead);

break;

case Token::ASSIGN\_RSH:

pChildNode = LookaHead.GetNode();

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::ASSIGN\_RSH);

LookaHead = ArithExpr(LookaHead);

Nodes.m\_pAssRSH = new CAct65AssignRSH;

pNextNode = LookaHead.GetNode();

Nodes.m\_pAssignment->CreateNode(pChildNode, pNextNode);

LookaHead.SetNode(Nodes.m\_pAstNode);

LookaHead = MemContents(LookaHead);

break;

case Token::ASSIGN\_SUB:

pChildNode = LookaHead.GetNode();

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::ASSIGN\_SUB);

LookaHead = ArithExpr(LookaHead);

Nodes.m\_pAssSub = new CAct65AssignSUB;

pNextNode = LookaHead.GetNode();

Nodes.m\_pAssignment->CreateNode(pChildNode, pNextNode);

LookaHead.SetNode(Nodes.m\_pAstNode);

LookaHead = MemContents(LookaHead);

break;

case Token::ASSIGN\_XOR:

pChildNode = LookaHead.GetNode();

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::ASSIGN\_XOR);

LookaHead = ArithExpr(LookaHead);

Nodes.m\_pAssXor = new CAct65AssignXOR;

pNextNode = LookaHead.GetNode();

Nodes.m\_pAssignment->CreateNode(pChildNode, pNextNode);

LookaHead.SetNode(Nodes.m\_pAstNode);

LookaHead = MemContents(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit Assignment", --m\_Recursion);

return LHChild;

}

//----------------------------------------------

// Relation Operations

//----------------------------------------------

CLHead CParser::RelOperation(CLHead LookaHead)

{

//--------------------------------------------

// RelOperation -> LogicalOR RelOperation\_1;

// RelOperation\_1 -> '>' LogicalOR RelOperation\_1

// -> '<' LogicalOR RelOperation\_1

// -> 'GTEQ' LogicalOR RelOperation\_1

// -> 'LTEQ' LogicalOR RelOperation\_1

// -> '=' LogicalOR RelEquals\_1

// -> '#' LogicalOR RelEquals\_1 //not equal

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN = 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter RelOperations", ++m\_Recursion);

LHChild = LogicalOR(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::GTEQ:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::GTEQ);

LHNext = LogicalOR(LHNext);

pN = new CAct65GTEQ;;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

case Token::LTEQ:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::LTEQ);

LHNext = LogicalOR(LHNext);

pN = new CAct65LTEQ;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

case Token('<'):

LHNext.m\_Token = Expect(LHChild.GetToken(), Token('<'));

LHNext = LogicalOR(LHNext);

pN = new CAct65LessTHAN;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

case Token('>'):

LHNext.m\_Token = Expect(LHChild.GetToken(), Token('>'));

LHNext = LogicalOR(LHNext);

pN = new CAct65GreaterTHAN;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

case Token('='):

LHNext.m\_Token = Expect(LHChild.GetToken(), Token('='));

LHNext = LogicalOR(LHNext);

pN = new CAct65EqualTO;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

case Token('#'): // not equals

LHNext.m\_Token = Expect(LHChild.GetToken(), Token('#'));

LHNext = LogicalOR(LHNext);

pN = new CAct65NotEquelTO;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit RelOperations", --m\_Recursion);

return LHChild;

}

CLHead CParser::LogicalOR(CLHead LookaHead)

{

//--------------------------------------------

// LogicalOR -> LogicalAND LogicalOR\_1;

// LogicalOR\_1 -> 'OR' LogicalAND LogicalOR\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN = 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter LogicalOR", ++m\_Recursion);

LHChild = LogicalAND(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::OR:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::OR);

LHNext = LogicalOR(LHNext);

pN = new CAct65GreaterTHAN;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit LogicalOR", --m\_Recursion);

return LHChild;

}

CLHead CParser::LogicalAND(CLHead LookaHead)

{

//--------------------------------------------

// LogicalAND -> RelOperation LogicalAND\_1;

// LogicalAND\_1 -> 'AND' RelOperation LogicalAND\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN = 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter LogicalAND", ++m\_Recursion);

LHChild = ArithExpr(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::AND:

LHNext.m\_Token = Expect(LookaHead.GetToken(), Token::AND);

LHNext = ArithExpr(LookaHead);

pN = new CAct65LogicalAND;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit LogicalAND", --m\_Recursion);

return LHChild;

}

//--------------------------------------

// Arithmetic Expressions

//--------------------------------------

CLHead CParser::ArithExpr(CLHead LookaHead)

{

//--------------------------------------------

// ArithExpr -> BitwiseAND BitwiseOR\_1;

// BitwiseOR\_1 -> '%' BitwiseAND BitwiseOR\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN = 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter ArithExpr/BitwiseOR", ++m\_Recursion);

LHChild = BitwiseAND(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token('%'):

LHNext.m\_Token = Expect(LHChild.GetToken(), Token('%'));

LHNext = BitwiseAND(LHNext);

pN = new CAct65BitWiseOR;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit ArithExpr/BitwiseOR", --m\_Recursion);

return LHChild;

}

CLHead CParser::BitwiseAND(CLHead LookaHead)

{

//--------------------------------------------

// BitwiseAND -> BitwiseXOR BitwiseAND\_1;

// BitwiseAND\_1 -> '&' BitwiseXOR BitwiseAND\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN = 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter BitwiseAND", ++m\_Recursion);

LHChild = BitwiseXOR(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token('&'): // not equals

LHNext.m\_Token = Expect(LHChild.GetToken(), Token('&'));

LHNext = BitwiseXOR(LHNext);

pN = new CAct65BitWiseAND;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit BitwiseAND", --m\_Recursion);

return LHChild;

}

CLHead CParser::BitwiseXOR(CLHead LookaHead)

{

//--------------------------------------------

// BitwiseXOR -> AddExpr BitwiseXOR\_1;

// BitwiseXOR\_1 -> '!' AddExpr BitwiseXOR\_1

// -> 'XOR' AddExpr BitwiseXOR\_1

// -> 'EOR' AddExpr BitwiseXOR\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN = 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter BitwiseXOR", ++m\_Recursion);

LHChild = AddExpr(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token('!'):

case Token::EOR:

case Token::XOR:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::XOR);

LHNext = AddExpr(LHNext);

pN = new CAct65XOR;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit BitwiseXOR", --m\_Recursion);

return LHChild;

}

CLHead CParser::AddExpr(CLHead LookaHead)

{

//--------------------------------------------

// AddExpr -> ShifExpr AddExpr\_1;

// AddExpr\_1 -> '+' ShifExpr AddExpr\_1

// -> '-' ShifExpr AddExpr\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN = 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter AddExpr", ++m\_Recursion);

LHChild = ShifExpr(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token('+'):

LHNext.m\_Token = Expect(LHChild.GetToken(), Token('+'));

LHNext = ShifExpr(LHNext);

pN = new CAct65ADD;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

case Token('-'):

LHNext.m\_Token = Expect(LHChild.GetToken(), Token('-'));

LHNext = ShifExpr(LookaHead);

pN = new CAct65SUB;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit AddExpr", --m\_Recursion);

return LHChild;

}

CLHead CParser::ShifExpr(CLHead LookaHead)

{

//--------------------------------------------

// ShifExpr -> MultExpr ShiftExpr\_1;

// ShiftExpr\_1 -> 'LSH' MultExpr ShiftExpr\_1

// -> 'RSH' MultExpr ShiftExpr\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN = 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter ShiftExpr", ++m\_Recursion);

LHChild = MultExpr(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::LSH:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::LSH);

LHNext = MultExpr(LHNext);

pN = new CAct65LSH;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

case Token::RSH:

LHNext.m\_Token = Expect(LHChild.GetToken(), Token::RSH);

LHNext = MultExpr(LookaHead);

pN = new CAct65RSH;

pN->CreateNode(LHChild.GetNode(), LHNext.GetNode());

LHChild.m\_pNode = pN;

LHChild.SetToken(LHNext.GetToken());

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit ShiftExpr", --m\_Recursion);

return LHChild;

}

CLHead CParser::MultExpr(CLHead LookaHead)

{

//--------------------------------------------

// MultExpr -> Unary MultExpr\_1;

// MultExpr\_1 -> '\*' Unary MultExpr\_1

// -> '/' Unary MultExpr\_1

// -> 'MOD' Unary MultExpr\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter MultExpr", ++m\_Recursion);

LHChild = Unary(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token('\*'):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('\*'));

LookaHead = Unary(LookaHead);

break;

case Token('/'):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('/'));

LookaHead = Unary(LookaHead);

case Token::MOD:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::MOD);

LookaHead = Unary(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit MultExpr", --m\_Recursion);

return LHChild;

}

CLHead CParser::Unary(CLHead LookaHead)

{

//--------------------------------------------

// Unary -> '-' Unary

// -> Factor

// ;

// I really wanted to call this Urinary

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Urnary", ++m\_Recursion);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token('-'):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('-'));

break;

default:

Loop = false;

break;

}

}

LookaHead = Factor(LookaHead);

PrintLookahead(LogFile(), LookaHead, "Exit Unary", --m\_Recursion);

return LookaHead;

}

CLHead CParser::Factor(CLHead LookaHead)

{

//--------------------------------------------

// Factor ->Value Factor\_1;

// Factor\_1 -> '(' ArithExpr ')'

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Factor", ++m\_Recursion);

LHChild = MemContents(LookaHead);

switch (LHChild.GetToken())

{

case Token::FUNC\_IDENT:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::FUNC\_IDENT);

LookaHead = ProcParams(LookaHead);

break;

case Token('('):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('('));

LookaHead = RelOperation(LookaHead);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(')'));

break;

default:

Loop = false;

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit Factor", --m\_Recursion);

return LHChild;

}

//--------------------------------------------

// Fundemental Declarations. These are the

// base types that every thing is based on

//--------------------------------------------

//-------------------------------------------

// Declarations

//-------------------------------------------

CLHead CParser::SysDecl(CLHead LookaHead)

{

//--------------------------------------------

// SysDecl -> TypeDefDecl Define;

// Define -> 'DEFINE' DefList TypeDefDecl Define

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter SysDecl", ++m\_Recursion);

LHChild = TypeDefDecl(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::DEFINE:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::DEFINE);

LookaHead = DefList(LookaHead);

LookaHead = TypeDefDecl(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit SysDecl", --m\_Recursion);

return LHChild;

}

CLHead CParser::DefList(CLHead LookaHead)

{

//--------------------------------------------

// DefList -> Def DefList\_1;

// DefList\_1 -> ',' Def DefList\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter DefList", ++m\_Recursion);

LHChild = Def(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token(','):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(','));

LookaHead = Def(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit DefList", --m\_Recursion);

return LHChild;

}

CLHead CParser::Def(CLHead LookaHead)

{

//--------------------------------------------

// Def -> IDENT '=' CompConst;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter DEF", ++m\_Recursion);

switch (LHChild.GetToken())

{

case Token::IDENT:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::IDENT);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('='));

LookaHead = CompConst(LookaHead);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit DEF", --m\_Recursion);

return LHChild;

}

//--------------------------------------

// TYPE Definition

//--------------------------------------

CLHead CParser::TypeDefDecl(CLHead LookaHead)

{

//--------------------------------------------

// TypeDefDecl ->FundDecl TypeDefDecl\_1;

// TypeDefDecl\_1 -> 'TYPE' EndTypeDef FundDecl TypeDefDecl\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter TypeDefDecl", ++m\_Recursion);

LHChild = FundDecl(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::TYPE :

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::TYPE);

LookaHead = EndTypeDef(LookaHead);

LookaHead = FundDecl(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit TypeDefDecl", --m\_Recursion);

return LHChild;

}

CLHead CParser::EndTypeDef(CLHead LookaHead)

{

//--------------------------------------------

// EndTypeDef ->RecDefField EndTypeDef\_1;

// EndTypeDef\_1 -> ']'

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter EndTypeDef", ++m\_Recursion);

LHChild = RecDefField(LookaHead);

switch (LHChild.GetToken())

{

case Token(']'):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(']'));

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit EndTypeDef", --m\_Recursion);

return LHChild;

}

CLHead CParser::RecDefField(CLHead LookaHead)

{

//--------------------------------------------

// RecDefField ->Ident RecDefField\_1;

// RecDefField\_1 -> '=' '[' LocalDecls

// -> '[' LocalDecls

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter RecDefField", ++m\_Recursion);

LHChild = Ident(LookaHead);

switch (LHChild.GetToken())

{

case Token('='):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('='));

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('['));

LookaHead = LocalDecls(LookaHead);

break;

case Token('['):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('['));

LookaHead = LocalDecls(LookaHead);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit RecDefField", --m\_Recursion);

return LHChild;

}

//--------------------------------------------

// Fundemental Declarations. These are the

// base types that every thing is based on

//--------------------------------------------

CLHead CParser::FundDecl(CLHead LookaHead)

{

//--------------------------------------------

// FundDecl ->FundPointerDecl FundDecl\_1;

// FundDecl\_1 -> 'BOOL' FundPointerDecl FundDecl\_1

// -> 'CHAR' FundPointerDecl FundDecl\_1

// -> 'BYTE' FundPointerDecl FundDecl\_1

// -> 'INT' FundPointerDecl FundDecl\_1

// -> 'CARD' FundPointerDecl FundDecl\_1

// -> 'RECORDTYPE' FundPointerDecl FundDecl\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter FundDecl", ++m\_Recursion);

LHChild = FundPointerDecl(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::RECORDTYPE:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::RECORDTYPE);

LookaHead = FundPointerDecl(LookaHead);

break;

case Token::CHAR:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::CHAR);

LookaHead = FundPointerDecl(LookaHead);

break;

case Token::BYTE:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::BYTE);

LookaHead = FundPointerDecl(LookaHead);

break;

case Token::CARD:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::CARD);

LookaHead = FundPointerDecl(LookaHead);

break;

case Token::INT:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::INT);

LookaHead = FundPointerDecl(LookaHead);

break;

case Token::BOOL:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::BOOL);

LookaHead = FundPointerDecl(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit FundDecl", --m\_Recursion);

return LHChild;

}

CLHead CParser::FundPointerDecl(CLHead LookaHead)

{

//--------------------------------------------

// FundPointerDecl ->FundArrayDecl FundPointerDecl\_1;

// FundPointerDecl\_1 -> 'POINTER' FundArrayDecl

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter FundPointerDecl", ++m\_Recursion);

LHChild = FundArrayDecl(LookaHead);

switch (LHChild.GetToken())

{

case Token::POINTER:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::POINTER);

LookaHead = FundArrayDecl(LookaHead);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit FundPointerDecl", --m\_Recursion);

return LHChild;

}

CLHead CParser::FundArrayDecl(CLHead LookaHead)

{

//--------------------------------------------

// FundArrayDecl ->FunctionDecl FundArrayDecl\_1;

// FundArrayDecl\_1 -> 'ARRAY' FunctionDecl

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter FundArrayDecl", ++m\_Recursion);

LHChild = FunctionDecl(LookaHead);

switch (LHChild.GetToken())

{

case Token::ARRAY:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::ARRAY);

LookaHead = FunctionDecl(LookaHead);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit FundArrayDecl", --m\_Recursion);

return LHChild;

}

CLHead CParser::FunctionDecl(CLHead LookaHead)

{

//--------------------------------------------

// FunctionDecl ->IdentList FunctionDecl\_1;;

// FunctionDecl\_1 -> 'FUNC' FuncDef

// -> 'PROC' ProcDef

// -> 'INTERRUPT' IrqDef

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter FunctionDecl", ++m\_Recursion);

LHChild = IdentList(LookaHead);

switch (LHChild.GetToken())

{

case Token::FUNC:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::FUNC);

LookaHead = FuncDef(LookaHead);

break;

case Token::PROC:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::PROC);

LookaHead = ProcDef(LookaHead);

break;

case Token::INTERRUPT:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::INTERRUPT);

LookaHead = IrqDef(LookaHead);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit FunctionDecl", --m\_Recursion);

return LHChild;

}

CLHead CParser::IrqDef(CLHead LookaHead)

{

//--------------------------------------------

// IrqDef->IrqDecl IrqBody;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter IrqDef", ++m\_Recursion);

LookaHead = IrqDecl(LookaHead);

LookaHead = IrqBody(LookaHead);

PrintLookahead(LogFile(), LookaHead, "Exit IrqDef", --m\_Recursion);

return LHChild;

}

CLHead CParser::ProcDef(CLHead LookaHead)

{

//--------------------------------------------

// ProcDef->ProcDecl ProcBody;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter ProcDef", ++m\_Recursion);

LookaHead = ProcDecl(LookaHead);

LookaHead = ProcBody(LookaHead);

PrintLookahead(LogFile(), LookaHead, "Exit ProcDef", --m\_Recursion);

return LookaHead;

}

CLHead CParser::FuncDef(CLHead LookaHead)

{

//--------------------------------------------

// FuncDef->FuncDecl FuncBody;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter FuncDef", ++m\_Recursion);

LookaHead = FuncDecl(LookaHead);

LookaHead = FuncBody(LookaHead);

PrintLookahead(LogFile(), LookaHead, "Exit FuncDef", --m\_Recursion);

return LookaHead;

}

CLHead CParser::IrqDecl(CLHead LookaHead)

{

//--------------------------------------------

// IrqDecl -> 'IDENT' OptInit '(' ')';

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter IrqDecl", ++m\_Recursion);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::IDENT);

LookaHead = OptInit(LookaHead);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('('));

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(')'));

PrintLookahead(LogFile(), LookaHead, "Exit IrqDecl", --m\_Recursion);

return LHChild;

}

CLHead CParser::ProcDecl(CLHead LookaHead)

{

//--------------------------------------------

// ProcDecl -> 'IDENT' OptInit '(' ParamList ')';

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter ProcDecl", ++m\_Recursion);

switch (LHChild.GetToken())

{

case Token::IDENT:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::IDENT);

LookaHead = OptInit(LookaHead);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('('));

LookaHead = ParamList(LookaHead);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(')'));

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit ProcDecl", --m\_Recursion);

return LHChild;

}

CLHead CParser::FuncDecl(CLHead LookaHead)

{

//--------------------------------------------

// FuncDecl -> 'IDENT' OptInit '(' ParamList ')';

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter FuncDecl", ++m\_Recursion);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::IDENT);

LookaHead = OptInit(LookaHead);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('('));

LookaHead = ParamList(LookaHead);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(')'));

PrintLookahead(LogFile(), LookaHead, "Exit FuncDecl", --m\_Recursion);

return LHChild;

}

CLHead CParser::OptInit(CLHead LookaHead)

{

//--------------------------------------------

// OptInit -> '=' CompConst

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter OptInit", ++m\_Recursion);

switch (LHChild.GetToken())

{

case Token('='):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('='));

LookaHead = CompConst(LookaHead);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit OptInit", --m\_Recursion);

return LookaHead;

}

CLHead CParser::IrqBody(CLHead LookaHead)

{

//--------------------------------------------

// IrqBody->LocalDecls Statements;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter IrqBody", ++m\_Recursion);

LookaHead = LocalDecls(LookaHead);

LookaHead = Statements(LookaHead);

PrintLookahead(LogFile(), LookaHead, "Exit IrqBody", --m\_Recursion);

return LHChild;

}

CLHead CParser::ProcBody(CLHead LookaHead)

{

//--------------------------------------------

// ProcBody -> LocalDecls Statements;

//--------------------------------------------

PrintLookahead(LogFile(), LookaHead, "Enter ProcBody", ++m\_Recursion);

LookaHead = LocalDecls(LookaHead);

LookaHead = Statements(LookaHead);

PrintLookahead(LogFile(), LookaHead, "Exit ProcBody", --m\_Recursion);

return LookaHead;

}

CLHead CParser::FuncBody(CLHead LookaHead)

{

//--------------------------------------------

// FuncBody -> LocalDecls Statements;

//--------------------------------------------

PrintLookahead(LogFile(), LookaHead, "Enter FuncBodt", ++m\_Recursion);

LookaHead = LocalDecls(LookaHead);

LookaHead = Statements(LookaHead);

PrintLookahead(LogFile(), LookaHead, "Exit FuncBody", --m\_Recursion);

return LookaHead;

}

//----------------------------------

// Identifiers

//----------------------------------

CLHead CParser::IdentList(CLHead LookaHead)

{

//--------------------------------------------

// IdentList -> Ident IdentList\_1;

// IdentList\_1 -> ',' Ident IdentList\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter IdentList", ++m\_Recursion);

LHChild = Ident(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token(','):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(','));

LookaHead = Ident(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit IdenrtList", --m\_Recursion);

return LHChild;

}

CLHead CParser::Ident(CLHead LookaHead)

{

//--------------------------------------------

// Ident ->IdentInit Ident\_1;

// Ident\_1 -> 'IDENT' IdentInit

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Ident", ++m\_Recursion);

LHChild = IdentInit(LookaHead);

switch (LHChild.GetToken())

{

case Token::IDENT:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::IDENT);

LookaHead = IdentInit(LookaHead);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit Ident", --m\_Recursion);

return LHChild;

}

CLHead CParser::IdentInit(CLHead LookaHead)

{

//--------------------------------------------

// IdentInit ->ArrayDim IdentInit\_1;

// IdentInit\_1 -> '=' Address

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter IdentInit", ++m\_Recursion);

LHChild = ArrayDim(LookaHead);

switch (LHChild.GetToken())

{

case Token('='):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('='));

LookaHead = Address(LookaHead);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit IdentInit", --m\_Recursion);

return LHChild;

}

CLHead CParser::Address(CLHead LookaHead)

{

//--------------------------------------------

// Address ->CompConst Address\_1;

// Address\_1 -> '[' Data ']'

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Address", ++m\_Recursion);

LHChild = CompConst(LookaHead);

switch (LHChild.GetToken())

{

case Token('['):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('['));

LookaHead = Data(LookaHead);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit Address", --m\_Recursion);

return LHChild;

}

CLHead CParser::Data(CLHead LookaHead)

{

//--------------------------------------------

// Data ->ConstList Data\_1;

// Data\_1 -> 'STRING'

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Data", ++m\_Recursion);

LHChild = ConstList(LookaHead);

switch (LHChild.GetToken())

{

case Token::STRING:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::STRING);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit Data", --m\_Recursion);

return LHChild;

}

CLHead CParser::ArrayDim(CLHead LookaHead)

{

//--------------------------------------------

// ArrayDim ->ArrayDim\_1;

// ArrayDim\_1 -> '(' CompConst ')'

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter ArrayDim", ++m\_Recursion);

switch (LHChild.GetToken())

{

case Token('('):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('('));

LookaHead = CompConst(LookaHead);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(')'));

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit ArrayDim", --m\_Recursion);

return LHChild;

}

//-------------------------------------------

// Function Parameters Declarations

//-------------------------------------------

CLHead CParser::ParamList(CLHead LookaHead)

{

//--------------------------------------------

// ParamList ->PramPointer Param\_1;

// Param\_1 -> 'RECORDTYPE' PramPointer Param\_1

// -> 'CHAR' PramPointer Param\_1

// -> 'BYTE' PramPointer Param\_1

// -> 'INT' PramPointer Param\_1

// -> 'CARD' PramPointer Param\_1

// -> 'BOOL' PramPointer Param\_1

// -> ',' Param\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter ParamList", ++m\_Recursion);

LHChild = PramPointer(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::RECORDTYPE:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::RECORDTYPE);

LookaHead = PramPointer(LookaHead);

break;

case Token::CHAR:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::CHAR);

LookaHead = PramPointer(LookaHead);

break;

case Token::BYTE:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::BYTE);

LookaHead = PramPointer(LookaHead);

break;

case Token::CARD:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::CARD);

LookaHead = PramPointer(LookaHead);

break;

case Token::INT:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::INT);

LookaHead = PramPointer(LookaHead);

break;

case Token::BOOL:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::BOOL);

LookaHead = PramPointer(LookaHead);

break;

case Token(','):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(','));

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit ParamList", --m\_Recursion);

return LHChild;

}

CLHead CParser::PramPointer(CLHead LookaHead)

{

//--------------------------------------------

// PramPointer ->ParamArray ParamPointer\_1;

// ParamPointer\_1 -> 'POINTER' ParamArray

// -> .

// ;

//-------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter PramPointer", ++m\_Recursion);

LHChild = ParamArray(LookaHead);

switch (LHChild.GetToken())

{

case Token::POINTER:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::POINTER);

LookaHead = ParamArray(LookaHead);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit PramPointer", --m\_Recursion);

return LHChild;

}

CLHead CParser::ParamArray(CLHead LookaHead)

{

//--------------------------------------------

// ParamArray ->IdentList ParamArray\_1;

// ParamArray\_1 -> 'ARRAY' IdentList

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter ParamArray", ++m\_Recursion);

LHChild = IdentList(LookaHead);

switch (LHChild.GetToken())

{

case Token::ARRAY:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::ARRAY);

LookaHead = IdentList(LookaHead);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit ParamArray", --m\_Recursion);

return LHChild;

}

//-----------------------------------------------

// Local Variableas

//-----------------------------------------------

CLHead CParser::LocalDecls(CLHead LookaHead)

{

//--------------------------------------------

// LocalDecls ->LocalPointerDecl LocalDecls\_1;

// LocalDecls\_1 -> 'CHAR' LocalPointerDecl LocalDecls\_1

// -> 'BYTE' LocalPointerDecl LocalDecls\_1

// -> 'INT' LocalPointerDecl LocalDecls\_1

// -> 'CARD' LocalPointerDecl LocalDecls\_1

// -> 'BOOL' LocalPointerDecl LocalDecls\_1

// -> 'RECORDTYPE' LocalPointerDecl LocalDecls\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter LocalDecls", ++m\_Recursion);

LHChild = LocalPointerDecl(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::CHAR:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::CHAR);

LookaHead = LocalPointerDecl(LookaHead);

break;

case Token::BYTE:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::BYTE);

LookaHead = LocalPointerDecl(LookaHead);

break;

case Token::CARD:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::CARD);

LookaHead = LocalPointerDecl(LookaHead);

break;

case Token::INT:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::INT);

LookaHead = LocalPointerDecl(LookaHead);

break;

case Token::BOOL:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::BOOL);

LookaHead = LocalPointerDecl(LookaHead);

break;

case Token::RECORDTYPE:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::RECORDTYPE);

LookaHead = LocalPointerDecl(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit LocalDecls", --m\_Recursion);

return LHChild;

}

CLHead CParser::LocalPointerDecl(CLHead LookaHead)

{

//--------------------------------------------

// LocalPointerDecl ->LocalArrayDecl LocalPointerDecl\_1;

// LocalPointerDecl\_1 -> 'POINTER' LocalArrayDecl

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter LocalPointerDecl", ++m\_Recursion);

LHChild = LocalArrayDecl(LookaHead);

switch (LHChild.GetToken())

{

case Token::POINTER:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::POINTER);

LookaHead = LocalArrayDecl(LookaHead);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit LocalPointerDecl", --m\_Recursion);

return LHChild;

}

CLHead CParser::LocalArrayDecl(CLHead LookaHead)

{

//--------------------------------------------

// LocalArrayDecl ->IdentList LocalArrayDecl\_1;

// LocalArrayDecl\_1 -> 'ARRAY' IdentList

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter LocalArrayDecl", ++m\_Recursion);

LHChild = IdentList(LookaHead);

switch (LHChild.GetToken())

{

case Token::ARRAY:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::ARRAY);

LookaHead = IdentList(LookaHead);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit LocalArrayDecl", --m\_Recursion);

return LHChild;

}

//-------------------------------

// Compiler Constants

//-------------------------------

CLHead CParser::ConstList(CLHead LookaHead)

{

//--------------------------------------------

// CompConstList -> CompConst CompConstList\_1;

// CompConstList\_1 -> ',' CompConst CompConstList\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter ConstList", ++m\_Recursion);

LHChild = CompConst(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token(','):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(','));

LookaHead = CompConst(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit ConstList", --m\_Recursion);

return LHChild;

}

CLHead CParser::CompConst(CLHead LookaHead)

{

//--------------------------------------------

// CompConst -> BaseCompConst CompConst\_1;

// CompConst\_1 -> '+' BaseCompConst CompConst\_1

// -> '-' BaseCompConst CompConst\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter CompConst", ++m\_Recursion);

LHChild = BaseCompConst(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token('+'):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('+'));

LookaHead = BaseCompConst(LookaHead);

break;

case Token('-'):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('-'));

LookaHead = BaseCompConst(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit CompConst", --m\_Recursion);

return LHChild;

}

CLHead CParser::BaseCompConst(CLHead LookaHead)

{

//--------------------------------------------

// BaseCompConst -> 'NUMBER'

// -> '\*'

// -> '@' MemContentsType

// -> 'INTERUPT\_IDENT'

// -> 'FUNC\_IDENT'

// -> 'PROC\_IDENT'

// -> .

// ;

//--------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter BaseCompConst", ++m\_Recursion);

switch (LHChild.GetToken())

{

case Token::NUMBER:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::NUMBER);

break;

case Token('\*'):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('\*'));

break;

case Token('@'):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('@'));

LookaHead = MemContents(LookaHead);

break;

case Token::INTERRUPT\_IDENT:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::INTERRUPT\_IDENT);

break;

case Token::FUNC\_IDENT:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::FUNC\_IDENT);

break;

case Token::PROC\_IDENT:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::PROC\_IDENT);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit BaseCompConst", --m\_Recursion);

return LHChild;

}

//----------------------------------

// Variable References

// Memory References

//----------------------------------

CLHead CParser::ValueList(CLHead LookaHead)

{

//--------------------------------------------

// ValueList ->Value ValueList\_1;

// ValueList\_1 -> ',' Value ValueList\_1

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter ValueList", ++m\_Recursion);

LHChild = Value(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token(','):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(','));

LookaHead = Value(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit ValueList", --m\_Recursion);

return LHChild;

}

CLHead CParser::Value(CLHead LookaHead)

{

//--------------------------------------------

// Value ->MemContentsType Value\_1;

// Value\_1 -> 'NUMBER'

// -> '\*' // current memroy location

// -> '@' AddressOf

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Value", ++m\_Recursion);

LHChild = MemContentsType(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::NUMBER:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::NUMBER);

break;

case Token('\*'):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('\*'));

break;

case Token('@'):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('@'));

LookaHead = MemContents(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit Value", --m\_Recursion);

return LHChild;

}

CLHead CParser::AddressOf(CLHead LookaHead)

{

//--------------------------------------------

// AddressOf ->MemContentsType AddressOf\_1;

// AddressOf\_1 ->'PROC\_IDENT'//

// -> 'FUNC\_IDENT'

// -> 'INTERRUPT\_IDENT'

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter AddressOf", ++m\_Recursion);

LHChild = MemContentsType(LookaHead);

switch (LHChild.GetToken())

{

case Token::INTERRUPT\_IDENT:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::INTERRUPT\_IDENT);

break;

case Token::FUNC\_IDENT:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::FUNC\_IDENT);

break;

case Token::PROC\_IDENT:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::PROC\_IDENT);

break;

default:

Loop = false;

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit AddressOf", --m\_Recursion);

return LHChild;

}

CLHead CParser::MemContentsType(CLHead LookaHead)

{

//--------------------------------------------

// MemContentsType ->MemContents MemContentsType\_1;

// MemContentsType\_1 -> '(' ArithExpr ')' //array ref

// -> '^' // Pointer Dereference

// -> '.' MemContents MemContentsType\_1 //record ref

// -> .

// ;

//--------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter MemContentsType", ++m\_Recursion);

LHChild = MemContents(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token('('):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('('));

LookaHead = ArithExpr(LookaHead);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(')'));

Loop = false;

break;

case Token('^'):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('^'));

Loop = false;

break;

case Token('.'):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('.'));

LookaHead = MemContents(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit MemcontentsType", --m\_Recursion);

return LookaHead;

}

CLHead CParser::MemContents(CLHead LookaHead)

{

//--------------------------------------------

// MemContents -> 'IDENT'

// -> .

// ;

//--------------------------------------------

CAstNode\* pN = 0;

PrintLookahead(LogFile(), LookaHead, "Enter MemContents", ++m\_Recursion);

pN = new CAct65IDENT;

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::IDENT);

LookaHead.SetNode(pN);

PrintLookahead(LogFile(), LookaHead, "Exit MemCoontents", --m\_Recursion);

return LookaHead;

}

//-----------------------------------------------------------

// Inline Assembler Methods

//-----------------------------------------------------------

CLHead CParser::AsmStmt(CLHead LookaHead)

{

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter AsmStmt", ++m\_Recursion);

LookaHead = Section(LookaHead);

LookaHead = Processor\_1(LookaHead);

PrintLookahead(LogFile(), LookaHead, "Enter AsmStmt", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::Processor\_1(CLHead LookaHead)

{

//--------------------------------------------------

// Processor ->PROCESSOR ProcessorType Processor

// -> .

// ;

//--------------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Processor\_1", ++m\_Recursion);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::PROCESSOR:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::PROCESSOR);

LookaHead = ProcessorType(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Enter Processor\_1", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::ProcessorType(CLHead LookaHead)

{

//--------------------------------------------------

// ProcessorType -> R6502

// -> WD65C02

// -> WD65C816

// ;

//--------------------------------------------------

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter ProcessorType", ++m\_Recursion);

switch (LHChild.GetToken())

{

case Token::R6502:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::R6502);

break;

case Token::WD65C02:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::WD65C02);

break;

case Token::WD65C816:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::WD65C816);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Enter ProcessorType", ++m\_Recursion);

return LookaHead;

}

//--------------------------------------------------

// SECTION statment

//--------------------------------------------------

CLHead CParser::Section(CLHead LookaHead)

{

//--------------------------------------------------

// Section -> Org Section\_1;

// Section\_1 -> 'SECTION' SectionName Org Section\_1

// -> .

// ;

//--------------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Section", ++m\_Recursion);

LHChild = Org(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::SECTION:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::SECTION);

LookaHead = SectionName(LookaHead);

LookaHead = Org(LookaHead);

break;

default:

printf("Line%d Col:%d %s",

GetLexer()->GetLineNumber(),

GetLexer()->GetColunm(),

GetLexer()->GetLexBuffer()

);

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit Section", ++m\_Recursion);

return LHChild;

}

CLHead CParser::SectionName(CLHead LookaHead)

{

//--------------------------------------------------

// SectionName ->SectionDef SectionName\_1;

// SectionName\_1 -> 'IDENT' SectionDef

// -> .

// ;

//--------------------------------------------------

PrintLookahead(LogFile(), LookaHead, "Enter SectionName", ++m\_Recursion);

PrintLookahead(LogFile(), LookaHead, "Exit SectionName", ++m\_Recursion);

return CLHead();

}

CLHead CParser::SectionDef(CLHead LookaHead)

{

//--------------------------------------------------

// SectionDef -> '[' SectionAttributesList ']'

// -> .

// ;

//--------------------------------------------------

PrintLookahead(LogFile(), LookaHead, "Enter SectionDef", ++m\_Recursion);

PrintLookahead(LogFile(), LookaHead, "Exit SectionDef", ++m\_Recursion);

return CLHead();

}

CLHead CParser::SectionAttributesList(CLHead LookaHead)

{

//--------------------------------------------------

// SectionAttributesList ->SectionAtribute SectionAttributesList\_1;

// SectionAttributesList\_1 -> ',' SectionAtribute SectionAttributesList\_1

// -> .

// ;

//--------------------------------------------------

PrintLookahead(LogFile(), LookaHead, "Enter SectionAttributesList", ++m\_Recursion);

PrintLookahead(LogFile(), LookaHead, "Exit SectionAttributesList", ++m\_Recursion);

return CLHead();

}

CLHead CParser::SectionAtribute(CLHead LookaHead)

{

//--------------------------------------------------

// SectionAtribute -> START '=' AsmConstant

// -> SIZE '=' AsmConstant

// -> MODE '=' Modes

// -> .

// ;

//--------------------------------------------------

CNumberStackItem\* pNSI;

CAccessModeStackItem\* pAMS;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter SectionAtribute", ++m\_Recursion);

switch (LHChild.GetToken())

{

case Token::START:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::START);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('='));

LookaHead = AsmConstant(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

if (pNSI->GetValue() == 0x0ff00)

{

printf("PieBug\n");

}

// fprintf(Act()->LogFile(), "Set Start Address to %04x\n", pNSI->GetValue());

GetCurrentSection()->SetStartAddress(pNSI->GetValue());

break;

case Token::SIZE:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::SIZE);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('='));

LookaHead = AsmConstant(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

GetCurrentSection()->SetSectionSize(pNSI->GetValue());

break;

case Token::MODE:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::MODE);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('='));

LookaHead = Modes(LookaHead);

pAMS = (CAccessModeStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::ACCESS\_MODE);

GetCurrentSection()->SetAccessMode(pAMS->GetAccessMode());

break;

case Token::ZEROPAGE:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::ZEROPAGE);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('='));

LookaHead = TrueFalse(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

if (pNSI->GetValue())

{

GetCurrentSection()->SetZeroPageFlag(CSection::AddressSize::ADDRESSSIZE\_ZEROPAGE);

}

else

{

GetCurrentSection()->SetZeroPageFlag(CSection::AddressSize::ADDRESSSIZE\_WORD);

}

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit SectionAtribute", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::Modes(CLHead LookaHead)

{

//--------------------------------------------------

// Modes ->READ\_ONLY

// ->READ\_WRITE

// ;

//--------------------------------------------------

CAccessModeStackItem\* pAMSI;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Modes", ++m\_Recursion);

pAMSI = new CAccessModeStackItem;

pAMSI->Create();

switch (LHChild.GetToken())

{

case Token::READ\_ONLY:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::READ\_ONLY);

pAMSI->SetAccessMode(CSection::Mode::MODE\_READ\_ONLY);

GetValueStack()->Push(pAMSI);

break;

case Token::READ\_WRTE:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::READ\_WRTE);

pAMSI->SetAccessMode(CSection::Mode::MODE\_READ\_WRITE);

GetValueStack()->Push(pAMSI);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit Modes", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::TrueFalse(CLHead LookaHead)

{

CNumberStackItem\* pNumber;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter TrueFalse", ++m\_Recursion);

pNumber = new CNumberStackItem;

pNumber->Create();

switch (LHChild.GetToken())

{

case Token::True:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::True);

pNumber->SetValue(1);

GetValueStack()->Push(pNumber);

break;

case Token::False:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::False);

pNumber->SetValue(0);

GetValueStack()->Push(pNumber);

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit TrueFalse", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::Org(CLHead LookaHead)

{

//-----------------------------------------

// Org ->DefineMemory Org1;

// Org1 -> 'ORG' 'NUMBER' DefineMemory Org1

// -> .

// ;

//-----------------------------------------

bool Loop = true;

CNumberStackItem\* pNSI;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Org", ++m\_Recursion);

LHChild = DefineMemory(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::ORG:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::ORG);

LookaHead = AsmConstant(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

GetCurrentSection()->SetLocationCounter(pNSI->GetValue());

LookaHead = DefineMemory(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit Org", ++m\_Recursion);

return LHChild;

}

CLHead CParser::DefineMemory(CLHead LookaHead)

{

//--------------------------------------------------

// DefineMemory -> DefineStorage DefineMemory1;

// DefineMemory1 -> DB AsmConstList DefineStorage DefineMemory1

// -> DW AsmConstList DefineStorage DefineMemory1

// -> DL AsmConstList DefineStorage DefineMemory1

// -> DAS STRING DefineStorage DefineMemory1

// -> DCS STRING DefineStorage DefineMemory1

// -> .

// ;

//--------------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

CDataSizeStackItem\* pDSI;

int i, l, c;

PrintLookahead(LogFile(), LookaHead, "Enter DefineMemory", ++m\_Recursion);

LHChild = DefineStorage(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::DB:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::DB);

pDSI = new CDataSizeStackItem;

pDSI->Create();

pDSI->SetSize(1); //Byte Size

GetValueStack()->Push(pDSI);

LookaHead = AsmConstList(LookaHead);

LookaHead = DefineStorage(LookaHead);

break;

case Token::DW:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::DW);

pDSI = new CDataSizeStackItem;

pDSI->Create();

pDSI->SetSize(2); //Word Size

GetValueStack()->Push(pDSI);

LookaHead = AsmConstList(LookaHead);

LookaHead = DefineStorage(LookaHead);

break;

case Token::DL:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::DL);

pDSI = new CDataSizeStackItem;

pDSI->Create();

pDSI->SetSize(4); //Long Size

GetValueStack()->Push(pDSI);

LookaHead = AsmConstList(LookaHead);

LookaHead = DefineStorage(LookaHead);

break;

case Token::DAS: //define action string

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::DAS);

l = strlen(GetLexer()->GetLexBuffer());

GetCurrentSection()->AddData(1, l);

for (i = 0; i < l; ++i)

{

c = GetLexer()->GetLexBuffer()[i];

GetCurrentSection()->AddData(1, c);

}

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::STRING);

break;

case Token::DCS: //define 'C' string

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::DCS);

l = strlen(GetLexer()->GetLexBuffer());

for (i = 0; i < l; ++i)

{

c = GetLexer()->GetLexBuffer()[i];

GetCurrentSection()->AddData(1, c);

}

GetCurrentSection()->AddData(1, 0);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::STRING);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LHChild, "Exit DefineMemory", ++m\_Recursion);

return LHChild;

}

CLHead CParser::DefineStorage(CLHead LookaHead)

{

//--------------------------------------------------

// DefineStorage -> Proceedure DefineStorage\_1;

// DefineStorage\_1 -> DS AsmConstant Proceedure DefineStorage\_1

// -> .

// ;

//--------------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

CNumberStackItem\* pNSI;

int BlockSize;

PrintLookahead(LogFile(), LookaHead, "Exit DefineStorage", ++m\_Recursion);

LHChild = Proceedure(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::DS:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::DS);

LookaHead = AsmConstant(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

BlockSize = pNSI->GetValue();

GetCurrentSection()->AllocateDataBlock(BlockSize);

LookaHead = Proceedure(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LHChild, "Exit DefineStorage", ++m\_Recursion);

return LHChild;

}

CLHead CParser::Proceedure(CLHead LookaHead)

{

//--------------------------------------------------

// Procedure -> Instruction Proceedure\_1;

// Procedure\_1 -> PROC Section EPROC Instruction Proceedure\_1

// -> .

// ;

//--------------------------------------------------

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter PRO|C", ++m\_Recursion);

LHChild = Instruction(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token::PROC:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::PROC);

LookaHead = Section(LookaHead);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::EPROC);

LookaHead = Instruction(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LHChild, "Exit PROC", --m\_Recursion);

return LHChild;

}

CLHead CParser::Instruction(CLHead LookaHead)

{

//--------------------------------------------------

// Instruction -> Labels Instruction\_1

// Instruction\_1 -> ADC AluAdrModes Labels Instruction\_1 //ALU Opcodes

// -> AND AluAdrModes Labels Instruction\_1

// ->EOR AluAdrModes Labels Instruction\_1

// ->ORA AluAdrModes Labels Instruction\_1

// ->CMP AluAdrModes Labels Instruction\_1

// ->SBC AluAdrModes Labels Instruction\_1

// ->LDA AluAdrModes Labels Instruction\_1

// ->STA StaAddressingModes Labels Instruction\_1

// ->ASL ShiftAddressingModes Labels Instruction\_1 //shift addressing modes

// -> ROL ShiftAddressingModes Labels Instruction\_1

// ->LSR ShiftAddressingModes Labels Instruction\_1

// ->ROR ShiftAddressingModes Labels Instruction\_1

// ->BCC RelAddressingMode Labels Instruction\_1 // Branch Op Codes

// -> BCS RelAddressingMode Labels Instruction\_1

// ->BEQ RelAddressingMode Labels Instruction\_1

// ->BMI RelAddressingMode Labels Instruction\_1

// ->BNE RelAddressingMode Labels Instruction\_1

// ->BPL RelAddressingMode Labels Instruction\_1

// ->BVC RelAddressingMode Labels Instruction\_1

// ->BVS RelAddressingMode Labels Instruction\_1

// ->BIT BitAddressModes Labels Instruction\_1 //BIT opcode

// -> BRK Labels Instruction\_1 //Implied Addressing Mode

// -> CLC Labels Instruction\_1

// ->CLD Labels Instruction\_1

// ->CLI Labels Instruction\_1

// ->CLV Labels Instruction\_1

// ->DEX Labels Instruction\_1

// ->DEY Labels Instruction\_1

// ->INX Labels Instruction\_1

// ->INY Labels Instruction\_1

// ->NOP Labels Instruction\_1

// ->PHA Labels Instruction\_1

// ->PLA Labels Instruction\_1

// ->PHP Labels Instruction\_1

// ->PLP Labels Instruction\_1

// ->RTI Labels Instruction\_1

// ->RTS Labels Instruction\_1

// ->SEC Labels Instruction\_1

// ->SED Labels Instruction\_1

// ->SEI Labels Instruction\_1

// ->TAX Labels Instruction\_1

// ->TAY Labels Instruction\_1

// ->TXA Labels Instruction\_1

// ->TYA Labels Instruction\_1

// ->TXS Labels Instruction\_1

// ->TSX Labels Instruction\_1

// ->INC IncAddressingMOdes Labels Instruction\_1 //Inc/Dec Addressing Modes

// -> DEC IncAddressingMOdes Labels Instruction\_1

// ->JMP JumpAddressingModes Labels Instruction\_1 // Jump

// -> JSR CallAddressingMode Labels Instruction\_1 // Jump To Subroutine

// -> CPX CPXAddressingMode Labels Instruction\_1

// ->CPY CPXAddressingMode Labels Instruction\_1

// ->LDX LdxAddressingMode Labels Instruction\_1

// ->LDY LdyAddressingMode Labels Instruction\_1

// ->STX StxAddressingMode Labels Instruction\_1

// ->STY StyAddressingMode Labels Instruction\_1

// -> .

// ;

//--------------------------------------------------

CInstruction\* pInstruct;

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Instruction", ++m\_Recursion);

LHChild = Labels(LookaHead);

while (Loop)

{

switch(LHChild.GetToken())

{

case Token::ADC: //ALU addressing

case Token::AND:

case Token::CMP:

case Token::EOR:

case Token::LDA:

case Token::ORA:

case Token::SBC:

PrepareInstruction(&pInstruct, LookaHead.GetToken());

GetValueStack()->Push(pInstruct);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), LookaHead.GetToken());

LookaHead = AluAdrModes(LookaHead);

GetValueStack()->Pop(CStackItem::ItemType::INSTRUCTION);

pInstruct->AddToCurrentSection();

LookaHead = Labels(LookaHead);

break;

case Token::STA: //store accumalator

PrepareInstruction(&pInstruct, Token::STA);

GetValueStack()->Push(pInstruct);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::STA);

LookaHead = StaAddressingModes(LookaHead);

GetValueStack()->Pop(CStackItem::ItemType::INSTRUCTION);

pInstruct->AddToCurrentSection();

LookaHead = Labels(LookaHead);

break;

case Token::ASL: //shift addressing modes

case Token::LSR:

case Token::ROL:

case Token::ROR:

PrepareInstruction(&pInstruct, LookaHead.GetToken());

GetValueStack()->Push(pInstruct);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), LookaHead.GetToken());

LookaHead = ShiftAddressingModes(LookaHead);

GetValueStack()->Pop(CStackItem::ItemType::INSTRUCTION);

pInstruct->AddToCurrentSection();

LookaHead = Labels(LookaHead);

break;

case Token::BCC: //relative addressing

case Token::BCS:

case Token::BEQ:

case Token::BMI:

case Token::BNE:

case Token::BPL:

case Token::BVC:

case Token::BVS:

PrepareInstruction(&pInstruct, LookaHead.GetToken());

GetValueStack()->Push(pInstruct);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), LookaHead.GetToken());

LookaHead = RelAddressingMode(LookaHead);

GetValueStack()->Pop(CStackItem::ItemType::INSTRUCTION);

pInstruct->AddToCurrentSection();

LookaHead = Labels(LookaHead);

break;

case Token::BIT:

PrepareInstruction(&pInstruct, LookaHead.GetToken());

GetValueStack()->Push(pInstruct);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::BIT);

LookaHead = BitAddressModes(LookaHead);

GetValueStack()->Pop(CStackItem::ItemType::INSTRUCTION);

pInstruct->AddToCurrentSection();

LookaHead = Labels(LookaHead);

break;

case Token::BRK: //implicit addressing

case Token::CLC:

case Token::CLD:

case Token::CLI:

case Token::CLV:

case Token::DEX:

case Token::DEY:

case Token::INX:

case Token::INY:

case Token::NOP:

case Token::PHA:

case Token::PLA:

case Token::PHP:

case Token::PLP:

case Token::RTI:

case Token::RTS:

case Token::SEC:

case Token::SED:

case Token::SEI:

case Token::TAX:

case Token::TAY:

case Token::TXA:

case Token::TYA:

case Token::TXS:

case Token::TSX:

PrepareInstruction(&pInstruct, LookaHead.GetToken());

pInstruct->AddToCurrentSection();

LookaHead.m\_Token = Expect(LookaHead.GetToken(), LookaHead.GetToken());

LookaHead = Labels(LookaHead);

break;

case Token::CPX: //compare index registers

case Token::CPY:

PrepareInstruction(&pInstruct, LookaHead.GetToken());

GetValueStack()->Push(pInstruct);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), LookaHead.GetToken());

LookaHead = CPX\_CPY\_AddressingMode(LookaHead);

GetValueStack()->Pop(CStackItem::ItemType::INSTRUCTION);

pInstruct->AddToCurrentSection();

LookaHead = Labels(LookaHead);

break;

case Token::DEC: //inc/dec

case Token::INC:

PrepareInstruction(&pInstruct, LookaHead.GetToken());

GetValueStack()->Push(pInstruct);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), LookaHead.GetToken());

LookaHead = IncAddressingMOdes(LookaHead);

GetValueStack()->Pop(CStackItem::ItemType::INSTRUCTION);

pInstruct->AddToCurrentSection();

LookaHead = Labels(LookaHead);

break;

case Token::JMP: //jump addressing modes

PrepareInstruction(&pInstruct, LookaHead.GetToken());

GetValueStack()->Push(pInstruct);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::JMP);

LookaHead = JumpAddressingModes(LookaHead);

GetValueStack()->Pop(CStackItem::ItemType::INSTRUCTION);

pInstruct->AddToCurrentSection();

LookaHead = Labels(LookaHead);

break;

case Token::JSR: //jsr addressing modes

PrepareInstruction(&pInstruct, LookaHead.GetToken());

GetValueStack()->Push(pInstruct);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::JSR);

LookaHead = CallAddressingMode(LookaHead);

GetValueStack()->Pop(CStackItem::ItemType::INSTRUCTION);

pInstruct->AddToCurrentSection();

LookaHead = Labels(LookaHead);

break;

case Token::LDX: //load index register

PrepareInstruction(&pInstruct, LookaHead.GetToken());

GetValueStack()->Push(pInstruct);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::LDX);

LookaHead = LdxAddressingMode(LookaHead);

GetValueStack()->Pop(CStackItem::ItemType::INSTRUCTION);

pInstruct->AddToCurrentSection();

LookaHead = Labels(LookaHead);

break;

case Token::LDY:

PrepareInstruction(&pInstruct, LookaHead.GetToken());

GetValueStack()->Push(pInstruct);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::LDY);

LookaHead = LdyAddressingMode(LookaHead);

GetValueStack()->Pop(CStackItem::ItemType::INSTRUCTION);

pInstruct->AddToCurrentSection();

LookaHead = Labels(LookaHead);

break;

case Token::STX: //store index registers

PrepareInstruction(&pInstruct, LookaHead.GetToken());

GetValueStack()->Push(pInstruct);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::STX);

LookaHead = StxAddressingMode(LookaHead);

GetValueStack()->Pop(CStackItem::ItemType::INSTRUCTION);

pInstruct->AddToCurrentSection();

LookaHead = Labels(LookaHead);

break;

case Token::STY:

PrepareInstruction(&pInstruct, LookaHead.GetToken());

GetValueStack()->Push(pInstruct);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::STY);

LookaHead = StyAddressingMode(LookaHead);

GetValueStack()->Pop(CStackItem::ItemType::INSTRUCTION);

pInstruct->AddToCurrentSection();

LookaHead = Labels(LookaHead);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LHChild, "Exit Instruction", ++m\_Recursion);

return LHChild;

}

CLHead CParser::Labels(CLHead LookaHead)

{

//--------------------------------------------------

// Labels -> IDENT LocalGlobal

// -> .

// ;

//--------------------------------------------------

CSymbol\* pSym;

CStackSymbolItem\* pSI;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

int Address = 0;

PrintLookahead(LogFile(), LookaHead, "Enter Labels", ++m\_Recursion);

switch (LHChild.GetToken())

{

case Token::IDENT:

if (GetLexer()->GetLexSymbol() == NULL)

{

pSym = new CSymbol();

pSym->Create();

pSym->SetName(GetLexer()->GetLexBuffer());

pSym->SetResolved();

Address = GetCurrentSection()->GetLocationCounter();

pSym->SetAddress(Address);

GetLexer()->GetSymTab()->AddSymbol(pSym);

pSym->SetSection(GetCurrentSection());

}

else

{

pSym = (CSymbol\*)GetLexer()->GetLexSymbol();

Address = GetCurrentSection()->GetLocationCounter();

pSym->SetAddress(Address);

pSym->SetResolved();

pSym->SetSection(GetCurrentSection());

pSym->BackFillUnresolved();

}

pSI = new CStackSymbolItem;

pSI->Create(pSym);

GetValueStack()->Push(pSI);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::IDENT);

LookaHead = LocalGlobal(LookaHead);

GetValueStack()->Pop(CStackItem::ItemType::SYMBOL);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit Labels", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::LocalGlobal(CLHead LookaHead)

{

//--------------------------------------------------

// LocalGlobal -> ':' //Local

// -> . //Global

// ;

//--------------------------------------------------

CStackSymbolItem\* pSSI = 0;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter LocalGlobal", ++m\_Recursion);

if (GetValueStack()->GetHead())

{

pSSI = (CStackSymbolItem\*)GetValueStack()->Look(0, CStackItem::ItemType::SYMBOL);

if (!pSSI)

ThrownException.SetXCeptType(Exception::ExceptionType::INTERNAL\_ERROR);

}

else

{

ThrownException.SetXCeptType(Exception::ExceptionType::INTERNAL\_ERROR);

}

switch (LHChild.GetToken())

{

case Token(':'):

pSSI->GetSymbol()->SetToken(Token::LOCAL\_LABEL);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(':'));

break;

default:

pSSI->GetSymbol()->SetToken(Token::GLOBAL\_LABLE);

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit LocalGlobal", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::AluAdrModes(CLHead LookaHead)

{

//--------------------------------------------------

// AluAdrModes -> '#' ConstUpperLower

// -> '(' Indirect

// -> AsmConstant OptIndexReg

// ;

//--------------------------------------------------

CInstruction\* pInst;

CNumberStackItem\* pIntValue;

int Address;

CRegisterStackItem\* pRSI;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter AluAdrModes", ++m\_Recursion);

pInst = (CInstruction\*)GetValueStack()->Look(0, CStackItem::ItemType::INSTRUCTION);

switch (LHChild.GetToken())

{

case Token('#'):

LookaHead = Immediate(LookaHead, pInst);

break;

case Token('('):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('('));

LookaHead = Indirect(LookaHead);

pInst->SetByteCount(2);

break;

default: //absolute, Zero Page and Indexed

LookaHead = AsmConstant(LookaHead);

pIntValue = (CNumberStackItem\*)GetValueStack()->Pop(

CStackItem::ItemType::INTVALUE

);

Address = pIntValue->GetValue();

LookaHead = OptIndexReg(LookaHead);

pRSI = (CRegisterStackItem\*)GetValueStack()->Pop(

CStackItem::ItemType::REGTYPE

);

switch (pRSI->GetRegType())

{

case CRegisterStackItem::RegType::NONE:

if (Address < 0x100 && pInst->IsResolved())

PageZero(pInst, Address, AdrModeType::ZERO\_PAGE\_ADR);

else

Absolute(pInst, Address, AdrModeType::ABSOLUTE\_ADR);

break;

case CRegisterStackItem::RegType::X:

if (Address < 0x100 && pInst->IsResolved())

PageZero(pInst, Address, AdrModeType::ZERO\_PAGE\_X\_ADR);

else

Absolute(pInst, Address, AdrModeType::ABSOLUTE\_X\_ADR);

break;

case CRegisterStackItem::RegType::Y:

Absolute(pInst, Address, AdrModeType::ABSOLUTE\_Y\_ADR);

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit AluAdrModes", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::Indirect(CLHead LookaHead)

{

//--------------------------------------------------

// Indirect -> AsmConstant Indirect\_1;

// Indirect\_1 -> ')' ',' YREG

// -> ',' XREG ')'

// -> .

// ;

//--------------------------------------------------

int Address = 0;

int OpCodeInc = 0;

CNumberStackItem\* pNSI;

CInstruction\* pInst;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Indirect", ++m\_Recursion);

LHChild = AsmConstant(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

Address = pNSI->GetValue();

pInst = (CInstruction\*)GetValueStack()->Look(0, CStackItem::ItemType::INSTRUCTION);

pInst->SetOperandW(Address);

switch (LHChild.GetToken())

{

case Token(')'): //indirect Y xxx (vv),Y

CheckZeroPageAddress(Address);

OpCodeInc = GetLexer()->

FindKeyword(pInst->GetOpCodeToken())->

FindInc(AdrModeType::INDIRECT\_Y\_ADR);

pInst->AddToOpCode(OpCodeInc);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(')'));

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(','));

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::YREG);

break;

case Token(','): //indirect X xxx (vv,X)

CheckZeroPageAddress(Address);

OpCodeInc = GetLexer()->

FindKeyword(pInst->GetOpCodeToken())->

FindInc(AdrModeType::INDIRECT\_X\_ADR);

pInst->AddToOpCode(OpCodeInc);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(','));

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::XREG);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(')'));

break;

default:

break;

}

PrintLookahead(LogFile(), LHChild, "Exit Indirect", ++m\_Recursion);

return LHChild;

}

CLHead CParser::StaAddressingModes(CLHead LookaHead)

{

//--------------------------------------------------

// StaAddressingModes -> AsmConstant OptIndexReg

// -> '(' Indirect

// ;

//--------------------------------------------------

int Address = 0;

CNumberStackItem\* pNSI;

CInstruction\* pInst = 0;

CRegisterStackItem\* pRSI;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter StaAddressingModes", ++m\_Recursion);

pInst = (CInstruction\*)GetValueStack()->Look(0, CStackItem::ItemType::INSTRUCTION);

switch (LHChild.GetToken())

{

case Token('('):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('('));

LookaHead = Indirect(LookaHead);

pInst->SetByteCount(2);

break;

default:

LookaHead = AsmConstant(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

Address = pNSI->GetValue();

LookaHead = OptIndexReg(LookaHead);

pRSI = (CRegisterStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::REGTYPE);

switch (pRSI->GetRegType())

{

case CRegisterStackItem::RegType::NONE:

if (Address < 0x100 && pInst->IsResolved())

PageZero(pInst, Address, AdrModeType::ZERO\_PAGE\_ADR);

else

Absolute(pInst, Address, AdrModeType::ABSOLUTE\_ADR);

break;

case CRegisterStackItem::RegType::X:

if (Address < 0x100 && pInst->IsResolved())

PageZero(pInst, Address, AdrModeType::ZERO\_PAGE\_X\_ADR);

else

Absolute(pInst, Address, AdrModeType::ABSOLUTE\_X\_ADR);

break;

case CRegisterStackItem::RegType::Y:

Absolute(pInst, Address, AdrModeType::ABSOLUTE\_Y\_ADR);

break;

}

}

PrintLookahead(LogFile(), LookaHead, "Exit StaAddressingModes", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::ShiftAddressingModes(CLHead LookaHead)

{

//--------------------------------------------------

// ShiftAddressingModes -> AREG

// -> AsmConstant OptXReg

// ;

//--------------------------------------------------

int Address = 0;

int OpCodeInc;

CInstruction\* pInst;

CNumberStackItem\* pNSI;

CRegisterStackItem\* pRSI;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter ShiftAddressingModes", ++m\_Recursion);

pInst = (CInstruction\*)GetValueStack()->Look(0, CStackItem::ItemType::INSTRUCTION);

switch (LHChild.GetToken())

{

case Token::AREG:

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::AREG);

pInst->SetByteCount(1);

OpCodeInc = GetLexer()->

FindKeyword(pInst->GetOpCodeToken())->

FindInc(AdrModeType::ACCUMULATOR);

pInst->AddToOpCode(OpCodeInc);

break;

default:

LookaHead = AsmConstant(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

Address = pNSI->GetValue();

LookaHead = OptXReg(LookaHead);

pRSI = (CRegisterStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::REGTYPE);

switch (pRSI->GetRegType())

{

case CRegisterStackItem::RegType::NONE:

if (Address < 0x100 && pInst->IsResolved())

PageZero(pInst, Address, AdrModeType::ZERO\_PAGE\_ADR);

else

Absolute(pInst, Address, AdrModeType::ABSOLUTE\_ADR);

break;

case CRegisterStackItem::RegType::X:

if (Address < 0x100 && pInst->IsResolved())

PageZero(pInst, Address, AdrModeType::ZERO\_PAGE\_X\_ADR);

else

Absolute(pInst, Address, AdrModeType::ABSOLUTE\_X\_ADR);

break;

}

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit ShiftAddressingModes", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::RelAddressingMode(CLHead LookaHead)

{

//--------------------------------------------------

// RelAddressingMode -> AsmConstant;

//--------------------------------------------------

int Address = 0;

unsigned DestAddress = 0;

CNumberStackItem\* pNSI;

CInstruction\* pInst;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter RelAddressingMode", ++m\_Recursion);

pInst = (CInstruction\*)GetValueStack()->Look(0, CStackItem::ItemType::INSTRUCTION);

LHChild = AsmConstant(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

if (pInst->IsUnResolved())

{

Address = 0;

}

else

{

DestAddress = pNSI->GetValue();

Address = GetCurrentSection()->GetLocationCounter() + 2;

Address = DestAddress - Address;

}

if (Address > 127 || Address < -128)

{

//-----------------------------------

// Address has exceeded it's boundry

// throw an exception.

//-----------------------------------

printf("Rel Address Boo-boo\n");

}

else

{

pInst->SetLowByte(Address);

}

PrintLookahead(LogFile(), LHChild, "Exit RelAddressingMode", ++m\_Recursion);

return LHChild;

}

CLHead CParser::BitAddressModes(CLHead LookaHead)

{

//--------------------------------------------------

// BitAddressModes -> AsmConstant;

//--------------------------------------------------

int Address = 0;

CNumberStackItem\* pNSI;

CInstruction\* pInst;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter BitAddressModes", ++m\_Recursion);

pInst = (CInstruction\*)GetValueStack()->Look(0, CStackItem::ItemType::INSTRUCTION);

LookaHead = AsmConstant(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

if (pInst->IsUnResolved())

Address = 0;

else

Address = pNSI->GetValue();

if (Address < 0x100 && pInst->IsResolved())

{

//---------------------

// Zero Page Absolute

//---------------------

PageZero(pInst, Address, AdrModeType::ZERO\_PAGE\_ADR);

}

else

{

//---------------------

// Absolute

//---------------------

Absolute(pInst, Address, AdrModeType::ABSOLUTE\_ADR);

}

PrintLookahead(LogFile(), LookaHead, "Exit BitAddressModes", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::IncAddressingMOdes(CLHead LookaHead)

{

//--------------------------------------------------

// IncAddressingMOdes -> AsmConstant OptXReg;

//--------------------------------------------------

int Address = 0;

CNumberStackItem\* pNSI;

CInstruction\* pInst;

CRegisterStackItem\* pRSI;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter IncAddressingMOdes", ++m\_Recursion);

pInst = (CInstruction\*)GetValueStack()->Look(0, CStackItem::ItemType::INSTRUCTION);

LHChild = AsmConstant(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

Address = pNSI->GetValue();

LookaHead = OptXReg(LookaHead);

pRSI = (CRegisterStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::REGTYPE);

switch (pRSI->GetRegType())

{

case CRegisterStackItem::RegType::X:

if (Address < 0x100 && pInst->IsResolved())

PageZero(pInst, Address, AdrModeType::ZERO\_PAGE\_X\_ADR);

else

Absolute(pInst, Address, AdrModeType::ABSOLUTE\_X\_ADR);

break;

case CRegisterStackItem::RegType::NONE:

if (Address < 0x100 && pInst->IsResolved())

PageZero(pInst, Address, AdrModeType::ZERO\_PAGE\_ADR);

else

Absolute(pInst, Address, AdrModeType::ABSOLUTE\_ADR);

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit IncAddressingMOdes", ++m\_Recursion);

return LHChild;

}

CLHead CParser::JumpAddressingModes(CLHead LookaHead)

{

//--------------------------------------------------

// JumpAddressingModes -> AsmConstant

// -> '(' AsmConstant ')'

// ;

//--------------------------------------------------

int Address = 0;

int OpCodeInc;

CNumberStackItem\* pNSI;

CInstruction\* pInst;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter JumpAddressingModes", ++m\_Recursion);

pInst = (CInstruction\*)GetValueStack()->Look(0, CStackItem::ItemType::INSTRUCTION);

switch (LHChild.GetToken())

{

case Token('('):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('('));

LookaHead = AsmConstant(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

Address = pNSI->GetValue();

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(')'));

OpCodeInc = GetLexer()->

FindKeyword(pInst->GetOpCodeToken())->FindInc(

AdrModeType::INDIRECT\_ADR

);

pInst->AddToOpCode(OpCodeInc);

pInst->SetOperandW(Address);

pInst->SetByteCount(3);

break;

default:

LookaHead = AsmConstant(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

Address = pNSI->GetValue();

Absolute(pInst, Address, AdrModeType::ABSOLUTE\_ADR);

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit JumpAddressingModes", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::CallAddressingMode(CLHead LookaHead)

{

//--------------------------------------------------

// CallAddressingMode -> AsmConstant;

//--------------------------------------------------

int Address = 0;

CNumberStackItem\* pNSI;

CInstruction\* pInst;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter CallAddressingMode", ++m\_Recursion);

pInst = (CInstruction\*)GetValueStack()->Look(0, CStackItem::ItemType::INSTRUCTION);

LookaHead = AsmConstant(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

Address = pNSI->GetValue();

Absolute(pInst, Address, AdrModeType::ABSOLUTE\_ADR);

PrintLookahead(LogFile(), LookaHead, "Exit CallAddressingMode", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::LdxAddressingMode(CLHead LookaHead)

{

//--------------------------------------------------

// LdxAddressingMode -> AsmConstant OptYReg

// -> '#' AsmConstant

// ;

//--------------------------------------------------

int Address = 0;

CNumberStackItem\* pNSI;

CInstruction\* pInst;

CRegisterStackItem\* pRSI;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter LdxAddressingMode", ++m\_Recursion);

pInst = (CInstruction\*)GetValueStack()->Look(0, CStackItem::ItemType::INSTRUCTION);

switch (LHChild.GetToken())

{

case Token('#'):

LookaHead = Immediate(LookaHead, pInst);

break;

default:

LookaHead = AsmConstant(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

Address = pNSI->GetValue();

LookaHead = OptYReg(LookaHead);

pRSI = (CRegisterStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::REGTYPE);

switch (pRSI->GetRegType())

{

case CRegisterStackItem::RegType::Y:

if (Address < 0x100 && pInst->IsResolved())

PageZero(pInst, Address, AdrModeType::ZERO\_PAGE\_Y\_ADR);

else

Absolute(pInst, Address, AdrModeType::ABSOLUTE\_Y\_ADR);

break;

case CRegisterStackItem::RegType::NONE:

Address = pNSI->GetValue();

if (Address < 0x100 && pInst->IsResolved())

PageZero(pInst, Address, AdrModeType::ZERO\_PAGE\_ADR);

else

Absolute(pInst, Address, AdrModeType::ABSOLUTE\_ADR);

break;

}

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit LdxAddressingMode", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::CPX\_CPY\_AddressingMode(CLHead LookaHead)

{

//--------------------------------------------------

// CPX\_CPY\_AddressingMode -> AsmConstant

// -> '#' AsmConstant

// ;

//--------------------------------------------------

int Address = 0;

CNumberStackItem\* pNSI;

CInstruction\* pInst;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter CPX\_CPY\_AddressingMode", ++m\_Recursion);

pInst = (CInstruction\*)GetValueStack()->Look(0, CStackItem::ItemType::INSTRUCTION);

switch (LHChild.GetToken())

{

case Token('#'):

LookaHead = Immediate(LookaHead, pInst);

break;

default:

LookaHead = AsmConstant(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

Address = pNSI->GetValue();

if (Address < 0x100 && pInst->IsResolved())

PageZero(pInst, Address, AdrModeType::ZERO\_PAGE\_ADR);

else

Absolute(pInst, Address, AdrModeType::ABSOLUTE\_ADR);

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit CPX\_CPY\_AddressingMode", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::StxAddressingMode(CLHead LookaHead)

{

//--------------------------------------------------

// StxAddressingMode -> AsmConstant OptYReg;

//--------------------------------------------------

int Address = 0;

CNumberStackItem\* pNSI = 0;

CInstruction\* pInst;

CRegisterStackItem\* pRSI = 0;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter StxAddressingMode", ++m\_Recursion);

pInst = (CInstruction\*)GetValueStack()->Look(0, CStackItem::ItemType::INSTRUCTION);

LHChild = AsmConstant(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

Address = pNSI->GetValue();

LookaHead = OptYReg(LookaHead);

pRSI = (CRegisterStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::REGTYPE);

switch (pRSI->GetRegType())

{

case CRegisterStackItem::RegType::Y:

if (Address >= 0x100)

{

fprintf(stderr, "Warning:STX Zero page: Truncated %04X\n",

Address & 0x0ffff

);

}

PageZero(pInst, Address, AdrModeType::ZERO\_PAGE\_Y\_ADR);

break;

case CRegisterStackItem::RegType::NONE:

if (Address < 0x100 && pInst->IsResolved())

PageZero(pInst, Address, AdrModeType::ZERO\_PAGE\_ADR);

else

Absolute(pInst, Address, AdrModeType::ABSOLUTE\_ADR);

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit StxAddressingMode", ++m\_Recursion);

return LHChild;

}

CLHead CParser::LdyAddressingMode(CLHead LookaHead)

{

//--------------------------------------------------

// LdyAddressingMode -> AsmConstant OptXReg

// -> '#' AsmConstant

// ;

//--------------------------------------------------

int Address = 0;

CNumberStackItem\* pNSI;

CInstruction\* pInst;

CRegisterStackItem\* pRSI;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter LdyAddressingMode", ++m\_Recursion);

pInst = (CInstruction\*)GetValueStack()->Look(0, CStackItem::ItemType::INSTRUCTION);

switch (LHChild.GetToken())

{

case Token('#'):

LookaHead = Immediate(LookaHead, pInst);

break;

default:

LookaHead = AsmConstant(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

Address = pNSI->GetValue();

LookaHead = OptXReg(LookaHead);

pRSI = (CRegisterStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::REGTYPE);

switch (pRSI->GetRegType())

{

case CRegisterStackItem::RegType::X:

if (Address < 0x100 && pInst->IsResolved())

PageZero(pInst, Address, AdrModeType::ZERO\_PAGE\_X\_ADR);

else

Absolute(pInst, Address, AdrModeType::ABSOLUTE\_X\_ADR);

break;

case CRegisterStackItem::RegType::NONE:

if (Address < 0x100 && pInst->IsResolved())

PageZero(pInst, Address, AdrModeType::ZERO\_PAGE\_ADR);

else

Absolute(pInst, Address, AdrModeType::ABSOLUTE\_ADR);

break;

}

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit LdyAddressingMode", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::StyAddressingMode(CLHead LookaHead)

{

//--------------------------------------------------

// StyAddressingMode -> AsmConstant OptXReg;

//--------------------------------------------------

int Address = 0;

CNumberStackItem\* pNSI;

CInstruction\* pInst;

CRegisterStackItem\* pRSI;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter StyAddressingMode", ++m\_Recursion);

pInst = (CInstruction\*)GetValueStack()->Look(0, CStackItem::ItemType::INSTRUCTION);

LHChild = AsmConstant(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

Address = pNSI->GetValue();

LookaHead = OptXReg(LookaHead);

pRSI = (CRegisterStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::REGTYPE);

switch (pRSI->GetRegType())

{

case CRegisterStackItem::RegType::X:

if (Address >= 0x100)

{

fprintf(stderr, "Warning:STY Zero page: Truncated %04X\n",

Address & 0x0ffff

);

}

PageZero(pInst, Address, AdrModeType::ZERO\_PAGE\_X\_ADR);

break;

case CRegisterStackItem::RegType::NONE:

if (Address < 0x100 && pInst->IsResolved())

PageZero(pInst, Address, AdrModeType::ZERO\_PAGE\_ADR);

else

Absolute(pInst, Address, AdrModeType::ABSOLUTE\_ADR);

break;

}

PrintLookahead(LogFile(), LHChild, "Exit StyAddressingMode", ++m\_Recursion);

return LHChild;

}

CLHead CParser::OptIndexReg(CLHead LookaHead)

{

//--------------------------------------------------

// OptIndexReg -> ',' OptIndexReg\_1

// -> .

// ;

//--------------------------------------------------

CRegisterStackItem\* pRSI = 0;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter OptIndexReg", ++m\_Recursion);

switch (LHChild.GetToken())

{

case Token(','):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(','));

LookaHead = OptIndexReg\_1(LookaHead);

break;

default:

pRSI = new CRegisterStackItem;

pRSI->Create();

pRSI->SetRegType(CRegisterStackItem::RegType::NONE);

GetValueStack()->Push(pRSI);

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit OptIndexReg", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::OptIndexReg\_1(CLHead LookaHead)

{

//--------------------------------------------------

// OptIndexReg\_1 -> XREG

// -> YREG

// ;

//--------------------------------------------------

CRegisterStackItem\* pRSI;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter OptIndexReg\_1", ++m\_Recursion);

pRSI = new CRegisterStackItem;

pRSI->Create();

switch (LHChild.GetToken())

{

case Token::XREG:

pRSI->SetRegType(CRegisterStackItem::RegType::X);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::XREG);

GetValueStack()->Push(pRSI);

break;

case Token::YREG:

pRSI->SetRegType(CRegisterStackItem::RegType::Y);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::YREG);

GetValueStack()->Push(pRSI);

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit OptIndexReg\_1", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::OptXReg(CLHead LookaHead)

{

//--------------------------------------------------

// OptXReg -> ',' XREG

// -> .

// ;

//--------------------------------------------------

CRegisterStackItem\* pRSI;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter OptXReg", ++m\_Recursion);

pRSI = new CRegisterStackItem;

pRSI->Create();

GetValueStack()->Push(pRSI);

switch (LHChild.GetToken())

{

case Token(','):

pRSI->SetRegType(CRegisterStackItem::RegType::X);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(','));

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::XREG);

break;

default:

pRSI->SetRegType(CRegisterStackItem::RegType::NONE);

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit OptXReg", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::OptYReg(CLHead LookaHead)

{

//--------------------------------------------------

// OptYReg -> ',' YREG

// -> .

// ;

//--------------------------------------------------

CRegisterStackItem\* pRSI;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter OptYReg", ++m\_Recursion);

pRSI = new CRegisterStackItem;

pRSI->Create();

GetValueStack()->Push(pRSI);

switch (LHChild.GetToken())

{

case Token(','):

pRSI->SetRegType(CRegisterStackItem::RegType::Y);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(','));

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::YREG);

break;

default:

pRSI->SetRegType(CRegisterStackItem::RegType::NONE);

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit OptYReg", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::AsmConstList(CLHead LookaHead)

{

//--------------------------------------------------

// AsmConstList ->STRING

// ->AsmConstList\_1

//--------------------------------------------------

int l, i;

int c;

int ObjectSize;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

CDataSizeStackItem\* pDSSI;

PrintLookahead(LogFile(), LookaHead, "Enter AsmConstList", ++m\_Recursion);

switch (LHChild.GetToken())

{

case Token::STRING:

pDSSI = (CDataSizeStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::DATA\_SIZE);

ObjectSize = pDSSI->GetSize();

l = strlen(GetLexer()->GetLexBuffer());

for (i = 0; i < l; ++i)

{

c = GetLexer()->GetLexBuffer()[i];

GetCurrentSection()->AddData(ObjectSize, c);

}

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::STRING);

break;

default:

LookaHead = AsmConstList\_1(LookaHead);

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit AsmConstList", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::AsmConstList\_1(CLHead LookaHead)

{

//---------------------------------------------------

// AsmConstList\_1 -> AsmConstant AsmConstList\_2

// AsmConstList\_2 -> ',' AsmConstant AsmConstList\_2

// -> .

// ;

//---------------------------------------------------

bool Loop = true;

CDataSizeStackItem\* pDSSI;

CNumberStackItem\* pNSI;

int ObjectSize;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter AsmConstList\_1", ++m\_Recursion);

pDSSI = (CDataSizeStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::DATA\_SIZE);

ObjectSize = pDSSI->GetSize();

LHChild = AsmConstant(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

GetCurrentSection()->AddData(ObjectSize, pNSI->GetValue());

while (Loop)

{

switch (LHChild.GetToken())

{

case Token(','):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token(','));

LookaHead = AsmConstant(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

GetCurrentSection()->AddData(ObjectSize, pNSI->GetValue());

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LHChild, "Exit AsmConstList\_1", ++m\_Recursion);

return LHChild;

}

CLHead CParser::AsmConstant(CLHead LookaHead)

{

//--------------------------------------------------

// AsmCpmstamt -> '>' AsmConstAddSub

// -> '<' AsmConstAddSub

// -> AsmConstAddSub

// ;

//--------------------------------------------------

CNumberStackItem\* pNSI = 0;

int v;

bool Loop = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter AsmConstant", ++m\_Recursion);

LHChild = AsmConstAddSub(LookaHead);

while (Loop)

{

switch (LHChild.GetToken())

{

case Token('>'):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('>'));

LookaHead = AsmConstAddSub(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Look(0, CStackItem::ItemType::INTVALUE);

//Lower half of word

v = pNSI->GetValue();

v &= 0x0ff;

pNSI->SetValue(v);

break;

case Token('<'):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('<'));

LookaHead = AsmConstAddSub(LookaHead);

pNSI = (CNumberStackItem\*)GetValueStack()->Look(0, CStackItem::ItemType::INTVALUE);

//Upper half of word

v = pNSI->GetValue();

v &= 0x0ff00;

v >>= 8;

pNSI->SetValue(v);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LHChild, "Exit AsmConstant", ++m\_Recursion);

return LHChild;

}

CLHead CParser::AsmConstAddSub(CLHead LookaHead)

{

//--------------------------------------------------

// AsmConstAddSub -> BaseAsmConstant AsmConstAddSub\_1;

// AsmConstAddSub\_1-> '+' BaseAsmConstant AsmConstAddSub\_1

// -> '-' BaseAsmConstant AsmConstAddSub\_1

// -> .

// ;

//--------------------------------------------------

bool Loop = true;

CNumberStackItem\* pNSI\_1;

CNumberStackItem\* pNSI\_2;

int Value;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter AsmConstAddSub", ++m\_Recursion);

LHChild = BaseAsmConstant(LookaHead);

pNSI\_1 = (CNumberStackItem\*)GetValueStack()->Look(0, CStackItem::ItemType::INTVALUE);

// AsmConstAddSub\_1

while (Loop)

{

switch (LHChild.GetToken())

{

case Token('+'):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('+'));

LookaHead = BaseAsmConstant(LookaHead);

pNSI\_2 = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

Value = pNSI\_1->GetValue();

Value += pNSI\_2->GetValue();

pNSI\_1->SetValue(Value);

break;

case Token('-'):

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('-'));

LookaHead = BaseAsmConstant(LookaHead);

pNSI\_2 = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

Value = pNSI\_1->GetValue();

Value -= pNSI\_2->GetValue();

pNSI\_1->SetValue(Value);

break;

default:

Loop = false;

break;

}

}

PrintLookahead(LogFile(), LHChild, "Exit AsmConstAddSub", ++m\_Recursion);

return LHChild;

}

CLHead CParser::BaseAsmConstant(CLHead LookaHead)

{

//--------------------------------------------------

// BaseAsmConstant -> LOCAL\_LABLE

// -> GLOBAL\_LABLE

// -> IDENT

// -> NUMBER

// -> CHAR\_CONSTANT

// -> '\*' //current location counter

// ;

//--------------------------------------------------

CNumberStackItem\* pNum;

CSymbol\* pSym;

CWhereSymbolIsUsed\* pSymUsed;

CInstruction\* pInstruction;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter BaseAsmConstant", ++m\_Recursion);

switch (LHChild.GetToken())

{

case Token::LOCAL\_LABEL:

pNum = new CNumberStackItem;

pNum->Create();

pNum->SetValue(((CSymbol\*)GetLexer()->GetLexSymbol())->GetAddress());

pNum->SetSymbol(((CSymbol\*)GetLexer()->GetLexSymbol()));

GetValueStack()->Push(pNum);

pSymUsed = new CWhereSymbolIsUsed;

pSymUsed->Create();

pSymUsed->SetAddress(GetCurrentSection()->GetLocationCounter());

GetLexer()->GetLexSymbol()->Add((CBin\*)pSymUsed);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::LOCAL\_LABEL);

break;

case Token::GLOBAL\_LABLE:

pNum = new CNumberStackItem;

pNum->Create();

pNum->SetValue(((CSymbol\*)GetLexer()->GetLexSymbol())->GetAddress());

pNum->SetSymbol(((CSymbol\*)GetLexer()->GetLexSymbol()));

GetValueStack()->Push(pNum);

pSymUsed = new CWhereSymbolIsUsed;

pSymUsed->Create();

pSymUsed->SetAddress(GetCurrentSection()->GetLocationCounter());

GetLexer()->GetLexSymbol()->Add((CBin\*)pSymUsed);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::GLOBAL\_LABLE);

break;

case Token::IDENT:

//-----------------------------------------

// This is complicated...

// If this token is returned, it means that

// this symbol has not yet been defined.

// That means it will not be possible to

// resolve what the address in the operand

// needs to be. If the lex symbol is NULL,

// then this is the first time it has been

// used. That means it needs to be

// created.

//-----------------------------------------

pSymUsed = new CWhereSymbolIsUsed;

pSymUsed->Create();

pSymUsed->SetSection(GetCurrentSection());

pNum = new CNumberStackItem;

pNum->Create();

pInstruction = (CInstruction\*)GetValueStack()->Look(0, CStackItem::ItemType::INSTRUCTION);

if (pInstruction)

{

//--------------------------------

// Kind of klugy, but if this is

// an instruction, the data is

// one byte beyound the location

// counter

//--------------------------------

pSymUsed->SetAddress(GetCurrentSection()->GetLocationCounter() + 1);

pInstruction->SetUnResolved();

if (pInstruction->IsRelative())

pSymUsed->SetUnResType(CWhereSymbolIsUsed::UnResolvedType::RELATIVE\_REFERENCE);

else

pSymUsed->SetUnResType(CWhereSymbolIsUsed::UnResolvedType::ABSOLUTE\_REFERENCE);

}

else

{

//--------------------------------

// Kind of klugy, but if this is

// a data declaration, then the

// location is the same as the

// location counter

//--------------------------------

pSymUsed->SetAddress(GetCurrentSection()->GetLocationCounter());

}

if (GetLexer()->GetLexSymbol())

{

pNum->SetSymbol((CSymbol\*)GetLexer()->GetLexSymbol());

GetLexer()->GetLexSymbol()->Add((CBin\*)pSymUsed);

}

else

{

//-----------------------------------

// No symbol exists, so create on

//-----------------------------------

pSym = new CSymbol;

pSym->Create();

pSym->SetName(GetLexer()->GetLexBuffer());

pSym->SetToken(Token::IDENT);

pSym->SetUnResolved();

pSym->Add((CBin\*)pSymUsed);

GetLexer()->GetSymTab()->AddSymbol(pSym);

pNum->SetSymbol(pSym);

}

GetValueStack()->Push(pNum);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::IDENT);

break;

case Token::NUMBER:

pNum = new CNumberStackItem;

pNum->Create();

pNum->SetValue(GetLexer()->GetNumber());

GetValueStack()->Push(pNum);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::NUMBER);

break;

case Token::CHAR\_CONSTANT:

pNum = new CNumberStackItem;

pNum->Create();

pNum->SetValue(GetLexer()->GetNumber());

GetValueStack()->Push(pNum);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token::CHAR\_CONSTANT);

break;

case Token('\*'):

pNum = new CNumberStackItem;

pNum->Create();

pNum->SetValue(GetCurrentSection()->GetLocationCounter());

GetValueStack()->Push(pNum);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('\*'));

break;

default:

break;

}

PrintLookahead(LogFile(), LookaHead, "Exit BaseAsmConstant", ++m\_Recursion);

return LookaHead;

}

CLHead CParser::Immediate(CLHead LookaHead, CInstruction\* pInst)

{

CNumberStackItem\* pIntValue;

int OpCodeInc;

int Address;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

PrintLookahead(LogFile(), LookaHead, "Enter Immediate", ++m\_Recursion);

LookaHead.m\_Token = Expect(LookaHead.GetToken(), Token('#'));

LookaHead = AsmConstant(LookaHead);

pIntValue = (CNumberStackItem\*)GetValueStack()->Pop(CStackItem::ItemType::INTVALUE);

OpCodeInc = GetLexer()->

FindKeyword(pInst->GetOpCodeToken())->

FindInc(AdrModeType::IMMEDIATE\_ADR);

pInst->AddToOpCode(OpCodeInc);

Address = pIntValue->GetValue();

pInst->SetLowByte(Address);

pInst->SetByteCount(2);

PrintLookahead(LogFile(), LookaHead, "Exit Immediate", ++m\_Recursion);

return LookaHead;

}

void CParser::PageZero(

CInstruction\* pInst,

int Address,

AdrModeType ModeType

)

{

int OpCodeInc;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

pInst->SetByteCount(2);

OpCodeInc = GetLexer()->

FindKeyword(pInst->GetOpCodeToken())->

FindInc(ModeType);

pInst->SetLowByte(Address);

pInst->AddToOpCode(OpCodeInc);

}

void CParser::Absolute(

CInstruction\* pInst,

int Address,

AdrModeType ModeType

)

{

int OpCodeInc;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

pInst->SetByteCount(3);

OpCodeInc = GetLexer()->

FindKeyword(pInst->GetOpCodeToken())->

FindInc(ModeType);

pInst->SetOperandW(Address);

pInst->AddToOpCode(OpCodeInc);

}

bool CParser::CheckZeroPageAddress(int A)

{

bool rV = true;

CAstNode\* pN= 0;

CLHead LHChild, LHNext;

if (A > 255||A<0)

{

ThrownException.SetXCeptType(Exception::ExceptionType::VALUE\_EXCEEDS\_RANGE);

sprintf\_s(

ThrownException.GetErrorString(),

ThrownException.GetMaxStringLen(),

"Zero Page Address:$%x Must be <= $ff",

A

);

throw(ThrownException);

}

return rV;

}

void CParser::PrintLookahead(

FILE\* pLog,

CLHead token,

const char\* pS,

int RecursionLevel,

int bump

)

{

char\* pLookaheadToken;

int TokenValue;

char\* pLexBuff;

if (pLog)

{

if (bump)

m\_Bump += bump;

pLexBuff = GetLexer()->GetLexBuffer();

switch (token.GetToken())

{

case Token::IDENT:

pLookaheadToken = GetLexer()->GetLexSymbol()->GetName();

fprintf(LogFile(), " %5d.%d::%s Lookahead = %s LexBuffer \'%s\'\n",

RecursionLevel,

m\_Bump,

pS,

pLookaheadToken,

pLexBuff

);

break;

case Token::NUMBER:

TokenValue = GetLexer()->GetNumber();;

fprintf(LogFile(), " %5d.%d::%s TokenValue: = %d LexBuffer \'%s\'\n",

RecursionLevel,

m\_Bump,

pS,

TokenValue,

pLexBuff

);

break;

default:

pLookaheadToken = (char\*)GetLexer()->GetKeyWords()->LookupToName(token.m\_Token);

if (token.GetNode() == 0)

{

fprintf(LogFile(), " %5d.%d::%s Lookahead CLHead: %s LexBuffer \'%s\' \n",

RecursionLevel,

m\_Bump,

pS,

pLookaheadToken,

pLexBuff

);

}

else

{

fprintf(LogFile(), " %5d.%d::%s TOKEN:%s LexBuffer:\'%s\' AST node:%s\n",

RecursionLevel,

m\_Bump,

pS,

pLookaheadToken,

pLexBuff,

token.m\_pNode->GetNodeName() ? token.m\_pNode->GetNodeName() : "Name is NULL"

);

}

break;

}

}

if (token.GetNode())

token.GetNode()->Print(LogFile(), 0,(char\*) "");

}

const char\* CParser::PHASE\_LUT::LookupPhaseName(PHASE phase)

{

return nullptr;

}

CParser::PHASE CParser::PHASE\_LUT::LookupPhaseToken(const char\* pName)

{

return PHASE();

}

---------- pch.cpp -------------

#include "pch.h"

Exception ThrownException;

const char\* Exception::GetExceptionTypeString(ExceptionType xType)

{

const char\* pName = 0;

int i;

bool Loop = true;

for (i = 0; Loop && ExceptionTypesLUT[i].m\_Name; ++i)

{

if (ExceptionTypesLUT[i].m\_Type == xType)

{

pName = ExceptionTypesLUT[i].m\_Name;

Loop = false;

}

}

return pName;

}

const char\* Exception::ExcepTypeToString::FindString(ExceptionType Type)

{

const char\* pString = 0;

int i;

bool Loop = true;

for (i = 0; Loop && (ExceptionTypesLUT[i].m\_Name != 0 ); ++i)

{

if (ExceptionTypesLUT[i].m\_Type == Type)

{

pString = ExceptionTypesLUT[i].m\_Name;

Loop = false;

}

}

return pString;

}

const char\* Exception::ExceptionSubTypeStrings::FindSubTypeString(ExceptionSubType Type)

{

const char\* pString = 0;

int i;

bool Loop = true;

for (i = 0; Loop && (XcepSubTypesLUT[i].m\_Name != 0); ++i)

{

if (XcepSubTypesLUT[i].m\_Type == Type)

{

pString = XcepSubTypesLUT[i].m\_Name;

Loop = false;

}

}

return pString;

}

---------- RegisterStackItem.cpp -------------

#include "pch.h"

CRegisterStackItem::CRegisterStackItem()

{

m\_RegType = RegType::NONE;

}

CRegisterStackItem::~CRegisterStackItem()

{

}

bool CRegisterStackItem::Create()

{

CStackItem::Create(CStackItem::ItemType::REGTYPE);

return true;

}

---------- Section.cpp -------------

#include "pch.h"

CSection::CSection():CBin(BinType::SECTION)

{

m\_StartAddress = 0;

m\_Size = 0;

m\_LocationCounter = 0;

m\_pSectionData = 0;

m\_AccessMode = Mode::MODE\_READ\_ONLY;

m\_ZeroPageAddressSize = AddressSize::ADDRESSSIZE\_WORD;

}

CSection::~CSection()

{

if (m\_pSectionData) delete[] m\_pSectionData;

}

bool CSection::Create()

{

bool rV = true;

CBin::Create();

return rV;

}

void CSection::SetStartAddress(unsigned Start)

{

if ((Start + m\_Size) <= 0x010000u)

{

m\_LocationCounter = Start;

m\_StartAddress = Start;

if (m\_Size > 0)

{

unsigned i;

m\_pSectionData = new char[m\_Size];

for (i = 0; i < m\_Size; ++i)

m\_pSectionData[i] = 0;

}

}

else

{

ThrownException.SetXCeptType(Exception::ExceptionType::SECTION\_ADDRES\_RANGE\_EXCEEDED);

ThrownException.SetSymbol(this);

ThrownException.SetGotToken(Token::NUMBER);

sprintf\_s(

ThrownException.GetErrorString(),

ThrownException.GetMaxStringLen(),

"Section:%s Start Address:$%04x Size:$%04x Over By:$%04x\n",

GetName(),

Start,

m\_Size,

Start + m\_Size - 0x10000

);

throw(ThrownException);

}

}

void CSection::SetSectionSize(unsigned Size)

{

if (m\_StartAddress + Size <= 0x10000)

{

m\_Size = Size;

if (m\_Size > 0)

{

unsigned i;

m\_pSectionData = new char[m\_Size];

for (i = 0; i < m\_Size; ++i)

m\_pSectionData[i] = 0;

}

}

else

{

ThrownException.SetXCeptType(Exception::ExceptionType::SECTION\_ADDRES\_RANGE\_EXCEEDED);

ThrownException.SetSymbol(this);

ThrownException.SetGotToken(Token::NUMBER);

sprintf\_s(

ThrownException.GetErrorString(),

ThrownException.GetMaxStringLen(),

"Section:%s Start Address:$%04x Size:$%04x Over By:$%04x\n",

GetName(),

m\_StartAddress,

Size,

m\_StartAddress + Size - 0x10000

);

throw(ThrownException);

}

}

void CSection::SetLocationCounter(unsigned short NewAddress)

{

if ((NewAddress < (m\_StartAddress + m\_Size)) &&

(NewAddress >= m\_StartAddress)

)

{

m\_LocationCounter = NewAddress;

}

else

{

ThrownException.SetXCeptType(Exception::ExceptionType::SECTION\_ADDRES\_RANGE\_EXCEEDED);

ThrownException.SetSymbol(this);

ThrownException.SetGotToken(Token::NUMBER);

sprintf\_s(

ThrownException.GetErrorString(),

ThrownException.GetMaxStringLen(),

"Section:%s Start Address:$%04x Size:$%04x Over By:$%04x\n",

GetName(),

0,

m\_Size,

NewAddress

);

throw(ThrownException);

}

}

unsigned short CSection::AddData(unsigned ObjectSize, int Value)

{

char\* b;

short\* s;

long\* l;

Info();

switch (ObjectSize)

{

case 1:

b = (char\*)&m\_pSectionData[m\_LocationCounter - m\_StartAddress];

m\_LocationCounter += 1;

\*b = char(Value);

break;

case 2:

s = (short\*)&m\_pSectionData[m\_LocationCounter - m\_StartAddress];

m\_LocationCounter += 2;

\*s = short(Value);

break;

case 4:

l = (long\*)&m\_pSectionData[m\_LocationCounter - m\_StartAddress];

m\_LocationCounter += 4;

\*l = long(Value);

break;

}

return m\_LocationCounter;

}

void CSection::AddDataAt(

unsigned Address,

unsigned ObjectSize,

int Value

)

{

char\* b;

short\* s;

long\* l;

Info();

switch (ObjectSize)

{

case 1:

b = (char\*)&m\_pSectionData[Address - m\_StartAddress];

\*b = char(Value) + \*b;

break;

case 2:

s = (short\*)&m\_pSectionData[Address - m\_StartAddress];

\*s = short(Value) + \*s;

break;

case 4:

l = (long\*)&m\_pSectionData[Address - m\_StartAddress];

\*l = long(Value) + \*l;

break;

}

}

unsigned short CSection::AllocateDataBlock(unsigned short size)

{

m\_LocationCounter += size;

return m\_LocationCounter;

}

void CSection::Print(FILE\* pOut, const char\* s)

{

fprintf(pOut, "Section:%s:Size:$%04X MODE:%s\n",

GetName(),

m\_Size,

int(m\_AccessMode) ? "Read/Write" : "Read Only"

);

CActionApp::Dump(

pOut,

m\_pSectionData,

GetStartAddress(),

GetSectionSize()

);

}

void CSection::Info()

{

// fprintf(

// Act()->LogFile(),

// "%s Loc Cntr:%04x\n",

// GetName(),

// );

}

---------- Stack.cpp -------------

#include "pch.h"

CStack::CStack()

{

m\_pHead = 0;

m\_ItemCount = 0;

m\_pName = 0;

}

CStack::~CStack()

{

CStackItem\* pSI;

// fprintf(

// stderr,

// "Stack Size = %d\n", m\_ItemCount

// );

while (m\_pHead)

{

pSI = m\_pHead->GetNext();

delete m\_pHead;

m\_pHead = pSI;

}

}

bool CStack::Create(const char\* pName)

{

m\_pName = pName;

return true;

}

void CStack::Push(CStackItem\* pItem)

{

if (pItem)

{

++m\_ItemCount;

pItem->SetNext(GetHead());

SetHead(pItem);

if (m\_ItemCount > 100)

printf("Is Stack Overflowing? %d\n", m\_ItemCount);

Print(Act()->LogFile(), "PUSH", 4, pItem);

}

}

CStackItem\* CStack::Pop(CStackItem::ItemType Type)

{

CStackItem\* pRV = 0;

if (m\_ItemCount)

{

--m\_ItemCount;

if (GetHead())

{

if (IsTopOfType(Type))

{

pRV = GetHead();

SetHead(GetHead()->GetNext());

Print(Act()->LogFile(), "POP", 4, pRV);

}

else

{

sprintf\_s(

m\_sExceptionString,

MAX\_STRING\_LEN,

"Got Stack Item Type:%s\nWanted Stack Item Type:%s\n",

GetHead()->LookupItemName(GetHead()->GetStackItemType()),

GetHead()->LookupItemName(Type)

);

StackException(

m\_sExceptionString,

Exception::ExceptionSubType::STACK\_ITEM\_MISMATCH

);

}

}

else

{

StackException(

"Stack Emptyh:Count wrong",

Exception::ExceptionSubType::STACK\_UNEXPECTED\_NULL

);

}

}

else

{

StackException(

"Cannot Pop an Empty Stack",

Exception::ExceptionSubType::STACK\_EMPTY

);

}

return pRV;

}

CStackItem\* CStack::Look(int Depth, CStackItem::ItemType Type)

{

int i = 0;

CStackItem\* pRV = 0;;

bool Loop = true;

if (GetHead())

{

for (i = 0, pRV = GetHead(); i < Depth && Loop; ++i)

{

if (pRV == 0)

{

Loop = false;

}

else

{

pRV = pRV->GetNext();

}

}

if (!pRV->Verify(Type))

{

ThrownException.SetXCeptType(Exception::ExceptionType::INTERNAL\_ERROR);

sprintf\_s(

ThrownException.GetErrorString(),

ThrownException.GetMaxStringLen(),

"Look:%s Stack:Look:Got %s Expected %s--Line %d Col %d",

GetStackName(),

CStackItem::LookupItemName(pRV->GetStackItemType()),

CStackItem::LookupItemName(Type),

Act()->GetParser()->GetLexer()->GetLineNumber(),

Act()->GetParser()->GetLexer()->GetColunm()

);

throw(ThrownException);

}

}

return pRV;

}

bool CStack::IsTopOfType(CStackItem::ItemType Type)

{

bool rV = false;

if (GetHead())

{

if (GetHead()->GetStackItemType() == Type)

rV = true;

}

else

{

fprintf(stderr,

"Stack %s Is Empty Line:%d\n",

GetStackName(),

Act()->GetParser()->GetLexer()->GetLineNumber()

);

}

return rV;

}

void CStack::Print(

FILE\* pLog,

const char\* pOperation,

int Indent,

CStackItem\* pRV

)

{

char\* IndentString = new char[256];

const char\* pItemTypeName;

Act()->CreateIndentString(IndentString, 256, Indent);

if (pLog)

{

if (pRV)

{

pItemTypeName =pRV->LookupItemName(pRV->GetStackItemType());

}

else

pItemTypeName = "";

fprintf(pLog, "\n%s------------------\n", IndentString);

fprintf(pLog,

"%sStack:%s:%s:TYPE:%s Line:%d\n",

IndentString,

GetStackName(),

pOperation,

pItemTypeName,

Act()->GetParser()->GetLexer()->GetLineNumber()

);

fprintf(pLog, "%sItemCount:%d\n", IndentString,GetItemCount());

fprintf(pLog, "%s------------------\n\n", IndentString);

}

delete[]IndentString;

}

void CStack::StackException(const char\* pDesc, Exception::ExceptionSubType SubType)

{

ThrownException.SetXCeptType(Exception::ExceptionType::STACK);

sprintf\_s(

ThrownException.GetErrorString(),

ThrownException.GetMaxStringLen(),

"Stack:%s::%s\n--Line: %d Col: %d",

GetStackName(),

pDesc,

Act()->GetParser()->GetLexer()->GetLineNumber(),

Act()->GetParser()->GetLexer()->GetColunm()

);

throw(ThrownException);

}

---------- StackItem.cpp -------------

#include "pch.h"

CStackItem::CStackItem()

{

m\_pNext = 0;

m\_Type = ItemType::NONE;

}

CStackItem::~CStackItem()

{

}

bool CStackItem::Create(ItemType Type)

{

bool rV = true;

m\_Type = Type;

return rV;

}

bool CStackItem::Verify(CStackItem::ItemType Type)

{

bool rV = false;

if (Type == GetStackItemType())

rV = true;

return rV;

}

const char\* CStackItem::LookupItemName(CStackItem::ItemType Type)

{

return CStackItem::ItemTypeLUT::FindItemName( Type);

}

const char\* CStackItem::ItemTypeLUT::FindItemName(CStackItem::ItemType Type)

{

const char\* pName = 0;

int i;

bool Loop = true;

for (i = 0;Loop && TypeString[i].m\_pName; ++i)

{

if (Type == TypeString[i].m\_Type)

{

pName = TypeString[i].m\_pName;

Loop = false;

}

}

return pName;

}

---------- StackNodeItem.cpp -------------

#include "pch.h"

CStackNodeItem::CStackNodeItem()

{

m\_pAstNode = 0;

}

CStackNodeItem::~CStackNodeItem()

{

}

bool CStackNodeItem::Create(CAstNode\* pNode)

{

bool rV = true;

CStackItem::Create(CStackItem::ItemType::NODE);

SetNode(pNode);

return rV;

}

---------- StackSectionItem.cpp -------------

#include "pch.h"

CStackSectionItem::CStackSectionItem()

{

}

CStackSectionItem::~CStackSectionItem()

{

}

bool CStackSectionItem::Create(CSection\* pSect)

{

bool rV = true;

CStackItem::Create(CStackItem::ItemType::SECTION);

SetSection(pSect);

return true;

}

---------- StackSymbolItem.cpp -------------

#include "pch.h"

CStackSymbolItem::CStackSymbolItem()

{

m\_pSymbol = 0;

}

CStackSymbolItem::~CStackSymbolItem()

{

}

bool CStackSymbolItem::Create(CSymbol\* pSym)

{

bool rV = true;

CStackItem::Create(CStackItem::ItemType::SYMBOL);

SetSymbol(pSym);

return rV;

}

---------- Symbol.cpp -------------

#include "pch.h"

bool CSymbol::Compare(const char\* name, int scope)

{

bool rV = false;

if (strcmp(GetName(), name) == 0)

{

if (scope > 0)

{

if (scope == GetScope())

rV = true;

}

else

rV = true;

}

return rV;

}

void CSymbol::BackFillUnresolved()

{

CWhereSymbolIsUsed\* pWSIU;

int Address;

int URLocation; //unresolved location

Address = GetAddress();

// fprintf(

// As65App.LogFile(),

// "Back Fill %s @ 0x%4x\n",

// GetName(),

// Address

// );

pWSIU = (CWhereSymbolIsUsed\*)GetHead();

while (pWSIU)

{

URLocation = pWSIU->GetAddress();

// fprintf(

// As65App.LogFile(),

// "Unresolved: @0x%04x\n",

// URLocatioon

// );

if (pWSIU->GetUnResType() == CWhereSymbolIsUsed::UnResolvedType::ABSOLUTE\_REFERENCE)

{

//------------------------------------------

// Get the location of an unresolved value,

// and then put the data defined by this

// symbol in there.

//------------------------------------------

pWSIU->GetSection()->AddDataAt(

pWSIU->GetAddress(), //address of data

2, // size of object

Address // object value

);

}

else

{

//------------------------------------------

// for relative address, subtract the

// location of the unresoved symbol from

// the value of this object.

//------------------------------------------

unsigned WhereSymIsUsedAddress = pWSIU->GetAddress();

unsigned RelAddress = Address - WhereSymIsUsedAddress;

RelAddress--;

pWSIU->GetSection()->AddDataAt(

WhereSymIsUsedAddress,

1,

RelAddress

);

}

pWSIU = (CWhereSymbolIsUsed\*)pWSIU->GetNext();

}

SetResolved();

}

void CSymbol::AddAtHead(CObjType\* pObjType)

{

if (GetHead())

{

GetTcHead()->SetPrev(pObjType);

pObjType->SetNext(GetTcTail());

SetTcHead(pObjType);

}

else

{

SetTcTail(pObjType);

SetTcHead(pObjType);

}

}

void CSymbol::AddAtTail(CObjType\* pObjType)

{

if (GetHead())

{

GetTcTail()->SetNext(pObjType);

pObjType->SetPrev(GetTcTail());

SetTcTail(pObjType);

}

else

{

SetTcTail(pObjType);

SetTcHead(pObjType);

}

}

void CSymbol::Delete(CObjType\* pObjType)

{

Unlink(pObjType);

delete pObjType;

}

void CSymbol::Unlink(CObjType\* pObjType)

{

if (GetTcHead() == pObjType)

{

SetTcHead(pObjType->GetNext());

if (GetTcHead())

GetTcHead()->SetPrev(0);

}

else if (GetTcTail() == pObjType)

{

SetTcTail(pObjType->GetPrev());

if (GetTcTail())

GetTcTail()->SetNext(0);

}

else

{

pObjType->GetNext()->SetPrev(pObjType->GetPrev());

pObjType->GetPrev()->SetNext(pObjType->GetNext());

}

}

void CSymbol::Print(FILE\* pOut, const char\* s)

{

fprintf(pOut, "%s:Address=%08lx Value=%04x Scope=%d\n",

GetName(),

GetAddress(),

GetValue(),

GetScope()

);

}

---------- SymTab.cpp -------------

#include "pch.h"

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Implementation of Hash Table Member

// Functions

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//-------------------------------------------

// Constructor

//

// parameter:

// size......number of Bins in the table

//

// Now, don't ask me why, but I heard that

// it is best to use a prime number for the

// number of bins. I suspect the reason why

// is that we are doing a MOD opertion in

// the hashing function to make sure the

// index ends up in the table.

//

// And if you are like me, you can't

// remember what the prime number are.

// So, here is a list form 0 to 200.

// 2, 3, 5, 7, 11, 13, 17, 19, 23, 29,

// 31, 37, 41, 43, 47, 53, 59, 61, 67

// 71, 73, 79, 83, 89, 97, 101, 103, 107,

// 109, 113, 127, 131, 137, 139, 149,

// 151, 157, 163, 167, 173, 179, 181,

// 191, 193, 197, 199

//--------------------------------------------

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Destructor

//

// destroy HashTab and all of its components

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

CSymTab::~CSymTab()

{

CBucket\* pBucket;

int i = 0;

for (i = 0; i < m\_tSize; ++i)

{

pBucket = m\_ppTab[i];

delete pBucket;

}

if (m\_ppTab) delete[] m\_ppTab;

}

bool CSymTab::Create(int TableDepth)

{

m\_ppTab = new CBucket \* [TableDepth];

m\_tSize = TableDepth;

for (int i = 0; i < m\_tSize; ++i)

m\_ppTab[i] = 0;

m\_nElements = 0;

return true;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// FindSym

//

// parameter:

// name....name of object to find

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

CBin\* CSymTab::FindSymbol(const char\* name, int scope)

{

CBin\* pRV = NULL;

int Index = Hash(name); //get index from hash of name

if(m\_ppTab[Index] != 0)

pRV = m\_ppTab[Index]->Find(name, scope);

return pRV;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// AddSym

//

// parameter:

// pSym....pointer to Bin to add to table

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void CSymTab::AddSymbol(CBin\* pSym)

{

int Index = Hash(pSym->GetName()); //generate index

if (m\_ppTab[Index] == NULL) //there is NO bucket here

{

m\_ppTab[Index] = new CBucket;

m\_ppTab[Index]->Create();

}

if (m\_ppTab[Index]->GetHead() == NULL) //nobody home

{

m\_ppTab[Index]->SetHead(pSym);

m\_ppTab[Index]->SetTail(pSym);

}

else

{

CBucket\* pB = m\_ppTab[Index];

pB->GetTail()->SetNext(pSym);

pSym->SetPrev(pB->GetTail());

pB->SetTail(pSym);

}

++m\_nElements; //increment number of symbols

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// DelSym

//

// parameter:

// pSym.....pointer to symbol to remove

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void CSymTab::DelSymbol(CBin\* pSym)

{

int HashValue = Hash(pSym->GetName());

m\_ppTab[HashValue]->Delete(pSym);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Hash

//

// parameter

// name.....pointer to name to Hash

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int CSymTab::Hash(const char\* name)

{

unsigned HashVal = 0;

unsigned TopBits;

unsigned char\* n;

n = (unsigned char\*)name;

while (\*n)

{

HashVal = (HashVal << 12) + \*n;

if ((TopBits = HashVal & 0xff000000) != 0)

HashVal = ((TopBits >> 24) ^ HashVal) & 0x00ffffff;

++n; //next char please...

}

return HashVal % m\_tSize; //returns index in table

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// PrinTable

//

// paramter:

// pOut.....output file stream

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void CSymTab::PrintTable(FILE\* pOut)

{

char\* s = new char[512];

CBin\* pB;

int i;

int maxStringLen = 0;

int l;

//------------------------------

// Scan table to find longest

// name

//------------------------------

for (i = 0; i < m\_tSize; ++i)

{

if (m\_ppTab[i] == NULL)

{

pB = NULL;

}

else

pB = m\_ppTab[i]->GetHead();

while (pB)

{

l = (int)strlen(pB->GetName());

if (l > maxStringLen)

maxStringLen = l;

pB = pB->GetNext();

}

}

for (i = 0; i < m\_tSize; ++i)

{

if (m\_ppTab[i] == NULL)

{

pB = NULL;

}

else

pB = m\_ppTab[i]->GetHead();

while (pB)

{

pB->Print(pOut);

pB = pB->GetNext();

}

}

}

bool CSymTab::FindAll(CBin::BinType Type, int NumberOfObject, CBin\*\* ppObjects)

{

bool rV = true;

int ItemCount = 0;

CBucket\* pBucket;

CBin\* pBin;

int i;

for (i = 0; i < m\_tSize; ++i)

{

pBucket = this->GetTable()[i];

if (pBucket)

{

pBin = pBucket->GetHead();

while (pBin)

{

if (pBin->GetType() == Type)

{

if (ItemCount < NumberOfObject)

{

ppObjects[ItemCount] = pBin;

ItemCount++;

}

else

{

++ItemCount;

}

}

pBin = pBin->GetNext();

}

}

}

if (ItemCount > NumberOfObject - 1)

rV = false;

return rV;

}

---------- UnResolvedSymbol.cpp -------------

#include "pch.h"

CUnResolvedSymbol::CUnResolvedSymbol():CBin(CBin::BinType::UNRESOLVED)

{

m\_Address = 0;

m\_pSection = 0;

m\_UnResRefType = CWhereSymbolIsUsed::UnResolvedType::ABSOLUTE\_REFERENCE;

}

CUnResolvedSymbol::~CUnResolvedSymbol()

{

}

bool CUnResolvedSymbol::Create()

{

return CBin::Create();

}

void CUnResolvedSymbol::SetAddress(unsigned a)

{

m\_Address = a;

fprintf(

Act()->LogFile(),

"Set Unresolved Symbol Location : %04x\n",

a

);

}

---------- Value.cpp -------------

#include "pch.h"

CValue::CValue()

{

m\_pSym = 0;

}

CValue::~CValue()

{

}

bool CValue::Create(CBin\* pSym)

{

bool rV = true;

SetSymbol(pSym);

return rV;

}

---------- WhereSymbolIsUsed.cpp -------------

#include "pch.h"

CWhereSymbolIsUsed::CWhereSymbolIsUsed() :CBin(CBin::BinType::WHERE\_USED)

{

m\_Address = 0;

m\_pSection = 0;

m\_UnResRefType = UnResolvedType::ABSOLUTE\_REFERENCE;

}

CWhereSymbolIsUsed::~CWhereSymbolIsUsed()

{

}

bool CWhereSymbolIsUsed::Create()

{

return CBin::Create();

}

void CWhereSymbolIsUsed::SetAddress(int a)

{

m\_Address = a;

// fprintf(

// Act()->LogFile(),

// "Set Unresolved Symbol Location : %04x\n",

// a

// );

}