Final Report

Team 7 : Jae Eun Kim, Eun Do Lee

1. Introduction

During this semester, our team implemented a fully functioning compiler for the SnuPL/1 programming language. Compiling a source code can be divided into 5 phases : 1. Lexical Analysis (Scanning) 2. Syntactic Analysis (Parsing) 3. Semantic Analysis 4. Intermediate Representation 5. Code Generation. We implemented each phase in order. This report shall also discuss our compiler according to the 5 phases. It shall briefly explain how the compiler functions, provide a summary of our code, provide details on implementations that were particularly difficult or interesting in each phase, and finally show the test results of the entire compiler.

2-1. Lexical Analysis (Scanning)

Lexical analysis refers to the process of converting a character stream, which is the source code in our case written in SnuPL/1, to a stream of tokens. A token indicates the detected lexeme of SnuPL/1 in a specific position of the character stream. A lexeme represents the basic meaningful unit in SnuPL/1. Custom tokens were defined according to the SnuPL/1 EBNF and were used within the implementation of the scanner.

The scanner consists of two C++ classes: CToken, and CScanner. They are implemented in scanner.h and scanner.cpp. A CToken instance represents a token with all the necessary information such as the token's actual string value. A CScanner instance represents the scanner itself. It repeatedly gets characters from an input stream and turns them into a stream of tokens. The scanner also contains an enumeration of all the tokens of SnuPL/1.

2-1-1. Basic Implementation

The most important functions of CScanner are GetChar() and PeekChar(). The scanner uses the two functions to scan the source code one character by one until the entire code is a stream of tokens. GetChar() is used to obtain the next character from the character stream to decide whether the following character(s) should be ignored, and if not, whether the lexeme is complete and ready to be converted into a token, whether there is a lexical error, and to determine the type and value of the token of the lexeme of which the scanner is currently building on. If looking one character ahead is not enough, the scanner uses PeekChar() to look one more character ahead to decide. The following are examples of some of the token types and their implementation details.

2-1-2. Operators

Operators are the easiest lexemes to distinguish and thus were the easiest to implement. For example, if GetChar() returns '+', then the scanner will convert it into a token of type *tPlusMinus* and value '+'. If the operator composes of two characters, the scanner will need to use PeekChar(). Assume GetChar() returns '|'. Then the scanner calls PeekChar(), and if it returns '|', the lexeme '||' will be converted into a token of type *tOr*. However, if PeekChar() returns any other character, the scanner will report lexical error. For another example, assume GetChar() returns '>'. Then the scanner will call PeekChar(). Whatever character it returns, the type of the token will be *tRelOp*. However, if the return value of PeekChar() is '=', the token's value will be '>=', and if it is anything else, the token's value will be set to '>'.

2-1-2. Keywords vs Identifiers

Keywords are predefined set of strings that are reserved. Each keyword has a fixed, specific meaning. All identifiers should be different from any element in the keyword set. Therefore, it is crucial that

the scanner distinguishes keywords well, meaning that the scanner should never mistake a keyword as an identifier. In order to implement this safely, predefined pairs of keywords and their corresponding tokens are stored as a global variable. When a CScanner instance is initialized, the predefined pairs are stored in the instance's array called *Keywords.* When a sequence of alphanumeric characters and underscores that is not a part of a string constant is met, the sequence is scanned until it meets a non-alphanumeric and non-underscore character. Then, the scanner looks up the sequence in the array *Keywords*. If there is a match, the corresponding token is returned. If there isn't a match, then the sequence is treated as an identifier and a token with type as *tIdent* and value as the sequence is returned.

2-1-2. Comments

SnuPL/1 compiler must ignore white-spaces and comments. Ignoring white-spaces was pre-implemented. As for comments, the scanner looks one character ahead when it meets a slash ('/'), to see if another one comes after. When this is the case, the scanner flushes the stream until a newline character is met, which would be the end of the single line comment.

2-1-3. Character and String Constants

Before starting on how to scan character and string constants, it is essential to explain about escape sequences. Escape sequences should be well detected in character constants and string constants. Upon meeting a backslash ('\') during the scan of character and string constants, the scanner tries to match the next character with 'n', 't', '\', ''', '"', '0', or 'x'. For each character, a newline character, a tab, a backslash, a single quote, a double quote, and a null character is set as or appended to the token value, respectively. For 'x', the scanner tries to match the next two characters with hexdigits, and calculates the value denoted by the characters and appends it to the token value. If the character after the initial backslash is none of the above, or if the hexdigits are invalid, the scanner throws an error.

Returning to the main part, at the start of a new lexeme, if GetChar() returns a single quotation, the lexeme must be a character constant. In other words, the token to be converted into is of type *tCharConst*. The scanner then calls PeekChar(). If PeekChar() returns a backslash, the character is an escape sequence, and its identity is judged as above. If there is no error, GetChar() consumes the entire escape sequence. If PeekChar() returns a single quotation, the character is empty, which is invalid. The scanner will throw an error. If PeekChar() returns any other character, then GetChar() simply consumes the character. After that, the scanner calls PeekChar() again and it must return a single quotation to end the lexeme. Otherwise the compiler throws an error.

Scanning a string constant is like continuously scanning a character in loops. At the start of a lexeme, if GetChar() returns a double quotation, the lexeme must be a string constant. The token to be converted into is of type *tStringConst*. The scanner then enters a while loop. At the beginning of the loop, PeekChar() returns a value and if it is a valid character, the character is added to the string constant, in other words it is added to the token value. GetChar() consumes the character, and the loop is started over again. The loop successfully terminates when PeekChar() returns a double quotation, indicating the end of the string. If it returns an invalid character, an error is thrown. Invalid characters are an unescaped newline character(strings cannot have multiple lines) and a null character(strings cannot have a null character in the middle). Of course, escape sequences apply in the same way as character constants.

2-2. Syntactic Analysis (Parsing)

The next step was implementing a parser, which constructs an abstract syntax tree from the stream of tokens produced by the scanner according to its grammatical structure. The grammar is specified by productions. Each non-terminal in a production can be modeled as a function. The relationship between non-terminals and terminals, which is essentially the production, can be implemented by function calls.

2-2-1. Basic Function Implementation

The following is the list of functions we implemented for the parser, which are implemented in parser.h and parser.cpp. The last four functions are for creating nodes of terminals rather than non-terminals. More on nodes would be explained later.

```
CAstModule*       module(void);

void              constDeclaration(CAstScope *s);
void              constDeclSequence(CAstScope *s);
void              constDecl(CAstScope *s);

void              varDeclaration(CAstScope *s);
void              varDeclSequence(CAstScope *s, bool formalParam);
void              varDecl(CAstScope *s, bool formalParam);

CAstProcedure*    subroutineDecl(CAstScope *s);
CAstStatement*      statSequence(CAstScope *s);
CAstStatement*      statement(CAstScope *s);

CAstStatReturn*     returnStatement(CAstScope *s);
CAstStatWhile*      whileStatement(CAstScope *s);
CAstStatIf*         ifStatement(CAstScope *s);
CAstFunctionCall*   subroutineCall(CAstScope *s, CToken *t, const CSymbol *sym);
CAstStatAssign*     assignment(CAstScope *s, CToken *t, const CSymbol *sym);

CAstExpression*     expression(CAstScope *s);
CAstExpression*     simpleexpr(CAstScope *s);
CAstExpression*     term(CAstScope *s);
CAstExpression*     factor(CAstScope *s);

CAstDesignator*     qualident(CAstScope *s, CToken *t, const CSymbol *sym);
CAstType*           type(CAstScope *s, const bool formalParam);
const CType*        arraytype(CAstScope *s, const CType *t, const bool formalParam);

CAstConstant*       boolean(void);
CAstConstant*       number(void);
CAstStringConstant* stringConst(CAstScope *s);
CAstConstant*       characterConst(void);
```

Implementing the function itself is, in principle, quite simple. We implement it according to the right hand side of the non-terminal's production. When a non-terminal is requested, the parser calls the corresponding function. When a terminal is requested, it consumes the token and checks if the type matches the one the production requested. The following is an example from SnuPl/1. There is a production 'assignment = qualident ":=" expression'. 'assignment', 'qualident' and 'expression' are non-terminals whereas ":=" is a terminal. The function 'assignment' first calls the function 'qualident'. If 'qualident' does not throw any error, 'assignment' consumes the next token and checks if it is ":=", the assignment token. If it is not, it throws an error. Then 'assignment' calls the 'expression' function. If 'expression' throws no error, 'assignment' is complete.

2-2-2. FIRST and FOLLOW

This process was not done without careful planning. In many cases, it was necessary to figure out the FIRST and the FOLLOW of the non-terminal before implementing the function. For example, some non-terminals were optional. Then the simple thing would be to move on if the next token was not in the FIRST of such non-terminal. In another case, a non-terminal had multiple productions. An example is 'statement = assignment | subroutineCall | ifStatement | whileStatement | returnStatement'. We solved the dilemma of which one to call by comparing the next token and the FIRST sets of the non-terminals on the right hand side. In the SnuPL/1 language, we didn't have to left factor any productions. FIRST can also be useful when spotting errors.

FOLLOW was useful when implementing non-terminals that ended with the form '{α}', where α denotes some grammar. Such cases were statSequence, constDeclSequence, and varDeclSequence. We implemented them by loops, and broke the loop when next token was in the FOLLOW set. Simply stopping when an unexpected token appears would be unwise because the token may be an error.

2-2-3. Implementing LL(2)

One of the most difficult sequences to implement was varDeclSequence because it requires LL(2). Its production is 'varDeclSequence = varDecl { ";" varDecl }'. When the parser has gone through varDecl

and the next token is “)”, it is clear that varDeclSequence is finished. The problem is when the next token is “;”. It is impossible to tell whether it is a separator between the last varDecl and the next varDecl or it is the last token of varDeclaration. The parser has to look at one more token ahead. If it is an identifier, the parser has to call another varDecl. If it is one of the elements of FOLLOW(varDeclaration) = {tBegin, tProcedure, tFunction}, varDeclSequence is over. The semicolon was the last token of varDeclaration. Implementing this neatly was not easy. The scanner can peek only one token. In order to know the next token of the next token, the parser has to consume the semicolon even though it might belong to varDeclaration. That is why our ‘varDeclaration’ function does not consume tSemicolon at the end. It is done during the varDeclSequence stage, and the parser decides what to do next then. To minimize complicated condition statements, we separated varDeclSequence into two functions, ‘varDecl’ and ‘varDeclSequenceAux’. ‘varDeclSequenceAux’ is called when the non-terminal varDeclSequence is requested. ‘varDecl’ is the exact implementation of the non-terminal varDecl according to its production. It consumes tokens and calls functions accordingly, and adds symbols to symbol tables. ‘varDeclSequenceAux’ is the function that decides what to do next after each varDecl. To make things simple, it accepts a boolean parameter ‘formalParam’. ‘varDeclSequenceAux’ has a loop whose condition is that the next token is tSemicolon. If it is, it first consumes the semicolon. Then it checks if ‘formalParam’ is negative and the next token is not an identifier. If both are true, the loop is terminated and so is varDeclSequence. Otherwise ‘varDecl’ is called and varDeclSequence continues.

2-2-4. Eliminating Left Recursion

It is crucial that there is no left recursion when building a predictive parser. However, the ‘type’ non-terminal had a left recursion, so we had to fix it. Originally, the production of type was ‘type = basetype | type “[“ [simpleexpr] “]”. We split it into two productions to eliminate left recursion :

type = basetype arraytype

arraytype = “[“ expression “]” arraytype

We made a separate function named ‘arraytype’ to effeciently implement the ‘type’ function.

2-2-5. Nodes and Scope

Certain functions create and return nodes of our parsing tree. Return types that start with letters ‘CAst’ are nodes.

Of these nodes, CAstModule and CAstProcedure are special because they have their own scope and symbol table. CAstModule’s scope is global, while CAstProcedure’s is local. Scope is passed down from caller to callee via parameter CastScope *s. Whenever a symbol table is created, it is initialized so that predefined function symbols are included. That way, functions of the same name cannot be declared by the user.

2-2-6. Symbols

Symbols must be created according to their type and inserted into the correct symbol table. For most symbols, this was not too difficult. Type could be set using the type manager, and local symbols simply get inserted in s→GetSymbolTable(). Global symbols, notably CSymProc, automatically get inserted to the global symbol table. Constant symbols(CSymConst), variable symbols(CSymbol) and parameter symbols(CSymParam) were the difficult ones because they were created and inserted in sequences. The following are the details of their implementations.

a) Constant Symbols

Constant symbols are created and stored at function ‘constDecl’. Production of constDecl is ‘constDecl = varDecl “=” expression’. The ‘constDecl’ function does not call the ‘varDecl’ function because it must create constant symbols, not variable symbols. We simply implemented constDecl’s own version of varDecl. Production of varDecl is ‘varDecl = ident { “,” ident } “:” type’. Until the parser consumes a colon, it keeps consuming identifier tokens and storing them in a vector. After consuming a colon, it calls the ‘type’ function and stores the returning node. Then varDecl is over, and the parser consumes tRelOp(=) and calls ‘expression’. The returning node of ‘expression’ is stored and the node has an Evaluate function which returns the value of the expression. After that, the parser brings s→ GetSymbolTable(), where it will store the

symbols. Until the vector is empty, the parser pops an identifier token, creates a symbol of type CSymConstant with the value of the token as its name, returning value of GetType() of the type node as its type, and its data initialized as the returning value of the Evaluate function of the expression node. However, for this phase, we initialized the data to 1. The data initializing was done with the help of CDataInitializer. After the symbol is created, the parser inserts it into the symbol table and shows an error message if the same symbol is already in the same table. This way we can prevent re-defined variables and constants.

b) Variable Symbols and Parameter Symbols

Creating and inserting variable and parameter symbols are actually implemented at the same location, the 'varDecl' function because the process is the basically the same. The difference is the type of the symbols and the fact that parameter symbols must be included to their corresponding procedure symbol. As mentioned above, the 'varDecl' function is the implementation of the production of the non-terminal varDecl. Therefore the overall procedure is similar to the 'constDecl' function without consuming tRelOp(=) and calling 'expression'. We will only mention what is different. One major difference is that the 'varDecl' function uses a boolean parameter 'formalParam'. If 'formalParam' is false, variable symbols are being requested, and the parser creates symbols of the type CSymbol and stores them at s→GetSymbolTable(). If 'formalParam' is true, parameter symbols are being requested, so the parser first creates CAstProcedure *cap = dynamic_cast<CAstProcedure*>(s) and brings symbol table cap → GetSymbolTable(). The parser then creates symbols of the type CSymParam, stores them at the symbol table, and adds them to the parameters of the corresponding procedure symbol obtained by cap→GetSymbol(). Unlike constant symbols, variable and parameter symbols do not require data initializing.

2-2-7. Array Type

There is one more function that accepts the boolean parameter 'formalParam', the 'type' function. Normally open arrays are not allowed, but when 'formalParam' is true, the parser accepts open arrays because they are allowed in parameter definitions.

2-3. Semantic Analysis

The third phase of compiling a source code is semantic analysis. The basic requirement was to add support for type checking, type inference, and expression evaluation to the parser built in the second phase, and to treat symbolic constants as previously evaluated constants. In addition, support for parsing minimum possible integer value in 32-bit system was implemented in the form of 'relaxed' parsing of unary negation operation '-'.

2-3-1. Basic Implementation

As mentioned above, newly implemented parser needed to support type checking, evaluation, and substitution of symbolic constants with appropriate constants. This section will provide explanations for the corresponding implementations. Implementations were mostly completed without altering the structure of the code from the second phase, with some exceptions where additional arguments had to be passed to some functions, or the return type of some functions needed to be changed.

2-3-2. Type Checking

Type checking is implemented in member functions of the abstract syntax tree nodes. The function is named TypeCheck. All type checking functions must satisfy the following property. For each node that type checks, the TypeCheck function returns true if and only if there is no type error in the entire subtree whose root is that node. Each TypeCheck function accepts as arguments a pointer to a CToken instance and a pointer to a string for error reporting purposes. When type error is detected, the token of the abstract syntax tree node that is performing the type check and a relevant error message is written to the address that is pointed by the arguments.

Type checking is invoked on the root node of the abstract syntax tree, the CAstModule instance, after parsing is completed. By the property of the type checking function explained above, this should ensure that all nodes in the abstract syntax tree contain correct types.

Before inferring the type of a node or evaluating the value of a node, type checking must be preceded. Otherwise the inferred type or evaluated value is not guaranteed to be accurate. Sometimes inferring types of children nodes is required to type check some node. In such a case children nodes are always type checked before the inference occurs. Evaluation is required in the parsing stage for acquiring correct array lengths for array declarations. This too requires early type checking before the global type check that is going to occur after parsing is completed.

2-3-3. Type Inference

Type inference is implemented in member functions of the abstract syntax tree nodes. The function is named GetType. Type inference of a node presumes that the subtree that has the node as root has been type checked, and no incorrect type is contained in all the descendant nodes. This is achieved by always ensuring that TypeCheck is called before GetType is called. TypeCheck ensures that all descendant nodes contain correct types.

2-3-4. Implicit Type Conversion from Arrays to Pointers

In order to pass an array as an argument of a subroutine in SnuPL/1, the caller passes a pointer to the array instead of the array itself. To resolve this issue, an array is converted into a pointer type when it becomes an argument of a subroutine.

Function *AddArg* of class CAstFunctionCall accepts *arg* of type CAstExpression* as parameter and adds it to the vector of the corresponding subroutine's arguments. If the type of *arg* is not an array, it simply pushes it into the vector. If, however, its type is an array, *AddArg* pushes CAstSpecialOp(*arg*→GetToken(), opAddress, *arg*, NULL) instead, which is a pointer to *arg*. Thus, the pointer's base type is *arg*'s type. To briefly explain the mechanism behind this, according to the parameters of CAstSpecialOp, its operand is *arg* and its operator is opAddress. If its operator is opAddress, the GetType() function of CAstSpecialOp returns a pointer type with the operand's type as its base type.

2-3-5. Value Evaluation

Evaluation is implemented in member functions of the abstract syntax tree nodes that inherit CAstExpression class. The function is named Evaluate. Evaluation of a node presumes that the subtree that has the node as root is TypeChecked. Thus, a node's TypeCheck must be called before calling Evaluate. Evaluation returns an instance of CDataInitializer. CDataInitializer is CdataInitInteger, CDataInitBoolean, CDataInitChar, or CDataInitString. The instance contains appropriate data. Evaluation of a parent node is inherently dependent on the evaluation of its children nodes. For example, if we are to evaluate a binary integer summation, we would need the values of the left-hand-side and right-hand-side to compute the sum. The evaluation is implemented so that the sum becomes a new value instance, to avoid having to make the strange decision of whether to update left-hand-side CDataInitializer or right-hand-side CDataInitializer and delete the other one. Instead, each evaluation frees CDataInitializers from its children's evaluation and returns a new instance. The exception is that the CDataInitializers contained in the symbol table should not be freed. So, the evaluation of CAstDesignator just returns a new CDataInitializer with the same internal value of its symbol's CdataInitializer.

2-3-6. Symbolic Constants

In the abstract syntax tree, symbolic constants need to be treated as a constant node. That is, we should be able to replace symbolic constants with its assigned values at compile time. Otherwise little justification exists for having such feature as part of the language. For symbolic constants of scalar types, the symbolic constants are parsed as CAstConstant, with the value that is stored in the symbol representing the symbolic constant. For symbolic constants of string type, the symbolic constants are parsed as a

CAstDesignator that, instead of pointing to the symbol of the symbolic constant itself, points to the global string variable that was added to the symbol table when parsing the declarations of the symbolic constants.

2-3-7. Relaxed Folding of Unary Negation

We chose to implement the relaxed folding of unary negation. For the implementation, the parser's functions were modified to accept an extra boolean parameter, fold. The modified functions are term and factor, which are called from simpleexpr, where the unary operation is first met. If the unary operation of opNeg is recognized, then simpleexpr looks ahead 1 token. If the peeked token is tNumber or tLParens, then fold is set to true, and term is called with fold and passes the value of fold to factor. If factor sees a number, then the factor parses a CAstConstant that contains the negative of the CAstConstant it gets from calling number. If factor sees tLParens, then the factor parses a CAstUnaryOp between the opNeg and the parenthesized expression. If simpleexpr does not see a number or a tLParens, then simpleexpr returns a unary operation between opNeg and whatever is to be parsed by calling term.

2-4. Intermediate Representation

The fourth phase is about converting the abstract syntax tree into three-address codes. The abstract syntax tree is two-dimensional. Converting it into three-address code form means transforming it into a linear code and making it one step closer to assembly code.

2-4-1. Basic Implementation

The ToTac functions in the AST nodes are responsible for generating three-address codes. The ToTac function returns CTacAddr*, which represents a variable in the three-address code. The function uses both synthesized and inherited attributes, which are the returned values of ToTac of the nodes corresponding to the attributes. With the appropriate operator and attributes, ToTac creates three-address codes that are relevant to each AST node and adds them to the code block in its argument. Three-address codes may be simple arithmetic operations, or branches that calls for a change in the program counter (PC). One such instruction is of operation type opGoto. Instead of attributes, the instruction is created with a label. The instruction tells the program to jump to the location in the code where the label is. Others are conditional branches that compare values of src1 and src2, and branch to a destination label if the comparison result matches that of the operation type. For example, three-address code of type opEqual would jump to its destination label if src1 and src2 are equal. The following are some specific examples of ToTac functions that we implemented.

2-4-2. Statement Sequence

Most statement nodes' ToTac functions are as simple as explained above. After creating and adding an instruction with their corresponding operator and attributes, they add an instruction with the operator opGoto and the label "next" that tells the program to jump to the next statement in the statement sequence. There were, however, slightly more complicated statements to implement because they required more labels and condition checks.

The ToTac function for the CAstStatIf node creates two labels, ifTrue and ifFalse. The condition is evaluated and the program jumps to ifTrue or ifFalse according to the condition. ToTac must generate code for both cases. It first adds the label ifTrue to the code block. Then it calls ToTac of the if body. In doing so, because it needs to tell the program to come back after executing the body, it makes a new label ifNext and passes it as a parameter to ToTac of the if body. Then it adds ifNext to the code block. After that it adds an instruction telling the program to jump to "next". Likewise, ToTac adds the label ifFalse to the code block and calls ToTac of the else body. It makes a new label elseNext, passes it as a parameter to ToTac of the else body, and adds it to the code block.

The ToTac function for the CAstStatWhile node creates two labels, conditionLabel and bodyLabel. ToTac adds conditionLabel to the code block and evaluates the condition. If the condition is true, the program jumps to bodyLabel, and if the condition is false, the program jumps to "next". The body of the while statement is implemented in a similar way to the if body or the else body of the if statement. A new

label bodyNext is created, and passed as a parameter to ToTac of the body. After calling ToTac of the body, the label is added to the code block. After that, the instruction with operator opGoto and label conditionLabel is added to the code block, telling the program to jump back to the condition and evaluate it again.

2-4-3. Operations

We will only explain binary operations' ToTac functions because unary and special operations' ToTac functions are similar to them and much simpler.

The three-address code is generated differently depending on whether the operator is a relational operator or not. If the operator is not a relational operator, that is, if it is '+', '-', '*', or '/', then the implementation is much easier. We created a temporary variable dst of CIntType in the code block. The attributes are the return values of ToTac of the left and right operands. Then we added to the code block an instruction with arguments : (GetOperation() (this will return the operator), dst, attribute1, attribute2). Thus, a three-address code will be generated that states that dst will have the result of the operation with attribute1 and attribute2 as operands.

Now the case of the relational operators remains. We had to be careful because boolean expressions may be control flows. To implement this efficiently, another ToTac function that accepts the true and false labels as additional arguments is required. We will explain this function, ToTac(CCodeBlock *cb, CTacLabel *ltrue, CTacLabel *lfalse) first. First, consider the case when the operator is not opAnd or opOr. Attributes are the return values of ToTac of the left and right operands. Then we added two instructions to the code block, one for the true condition and the other for the false condition. Next, we implemented for the operator opAnd. This operator is special because if the left operand is false, the entire result is false, so the right operand does not have to be evaluated. An additional label is necessary. We created a new label called localLabel. If the left operand is false, the program jumps to lfalse. Otherwise it stays at the localLabel. Then the program evaluates the right operand and jumps to ltrue or lfalse accordingly. opOr is just the opposite case of opAnd, so the implementation is quite the same. If the left operand is true, the entire result is true, so the right operand does not have to be evaluated. If the left operand is true, the program jumps to ltrue. Otherwise it stays at localLabel, then the program evaluates the right operand and jumps to ltrue or lfalse accordingly.

We return back to ToTac(CCodeBlock *cb). ToTac(CCodeBlock *cb) creates the labels ltrue, lfalse and calls ToTac(CCodeBlock *cb, CTacLabel *ltrue, CTacLabel *lfalse) explained above. Then it creates and adds a temporary variable dst to the code block, but this time of type CBoolType. Then it creates a CAstDesignator node with dst's symbol. After that, two boolean CAstConstant nodes are created, one with true value and the other with false value. Two CAstStatAssign nodes are created with the designator node and one with the true constant node, the other with the false constant node. Then each CAstStatAssign node's ToTac function is called. dst is the return value of ToTac(*cb), like the integer operation case.

2-5. Code Generation

The fifth and last phase of this project converts the intermediate representation(three-address codes) into an x86 assembly code. The assembly code generated from this phase can be compiled into an executable file by an assembler. Our implementation for this phase can be found in backend.h and backend.cpp.

The output of this phase is largely consisted of two parts, code and data. Code is basically the sequence of instructions. Data are either global or local. Local data are data only active in their scope such as parameters, local variables, and temporaries and are stored on the stack. Global data are allocated statically.

2-5-1. Operand

In order to correctly emit corresponding assembly code of a three address code, a correct representation of the operands of the three address code is necessary. Essentially, constant operands are represented with their value prefixed with '$', and variable operands are represented with the combination of their base register and offsets to follow the required assembly code convention. For example, an operand that has base register %ebp and offset of 12 will be represented as 12(%ebp). The base registers and offsets are calculated and saved when a scope's stack offsets are computed.

One special case that needs care is the operands of type CTacReference. These reference variables hold the address of its data in the address indicated by the base register and offset. Hence, the data needs to be loaded into memory. In order to achieve this our implementation emits code that moves the data into register %edi. Then, the operand is represented as “(%edi)”. So if the operand has base register of %ebp and offset of 12, code “movl 12(%ebp), %edi” will be emitted, and the operand will be represented as (%edi) to be used in other emission functions.

2-5-2. Code Emission

Code is emitted per scope. EmitCode collects the module’s subscopes and calls the function EmitScope with each scope as its parameter in iteration, ordering each scope to emit its code.

EmitScope is mostly about entering (prologue) and exiting (epilogue) the procedure or function corresponding to the scope, which shall be discussed in detail in section 2-5-3. This section will focus on emitting the code of the function body. EmitScope prints “# function body” and calls the function EmitCodeBlock. EmitCodeBlock simply keeps emitting each instruction in the code block in order until the end. The function EmitInstruction is responsible for emitting a single instruction, and its implementation is of main interest in this section.

There are two EmitInstruction functions : EmitInstruction(string mnemonic, string args, string comment) and EmitInstruction(CTacInstr *i). The former prints the instruction in the assembly code in the order of mnemonic, args, and comment. mnemonic denotes the instruction type, and examples are “addl” and “subl”. args are the registers used to execute the instruction.

The latter EmitInstruction’s actions will be explained according to the instruction *i’s operation. It is useful to examine the case when the instruction’s operation is a binary operation because there are many other cases with similar or simpler implementations, so it is helpful for general understanding. Binary operations include additions(opAdd), subtractions(opSub), multiplications(opMul), divisions(opDiv), and(opAnd) operations, and or(opOr) operations. There are two sources and one destination. The first source is obtained by i→GetSource(1), the second by i→GetSource(2). The destination is obtained by i→GetDest(). Then the first source is loaded to register “%eax”, and the second to register “%ebx” with the Load function. Let mnm be a short notation for mnemonic, the instruction type. EmitInstruction(mnm, “%ebx, %eax”) is called. After that, the destination value is stored in the memory by the Store function.

The implementation of unary operations is very similar because apart from the fact that there is only one source, so GetSource(1) is called only once, everything else is the same. More careful observation reveals how similar implementing an unconditional branch is to implementing a binary operation, as there are two sources and a comparing operation between the two. However, the destination value will not be stored on the stack.

The last to look at are function-call related operations. Not only are they intriguing implementations, they help the understanding of function calling and are related to section 2-5-3. The operation for calling functions is opCall. To start a function, the name of the function is important, because the program uses the name to jump to the function’s code. Therefore instruction “call <function name>” must be printed. EmitInstruction calls GetSource(1), which returns the function name, and GetDest(), which returns the return value of the function. After printing “call <function name>”, it prints the instruction ordering to add paramSize = (# of parameters)*4 to %esp if paramSize is not zero. This means allocating space in the stack for parameters, and is implemented by calling EmitInstruction(“call”, Imm(paramSize) + “,%esp”). If the return value type is not null, the destination value is stored in the memory by the Store function.

The operation opReturn exits the function. It calls GetSource(1), which is the optional return value of the function, and if it not NULL, it is loaded to the register “%eax” by the Load function. Then the instruction telling the program to jump to the corresponding exit label is printed, which is implemented by calling EmitInstruction(“jmp”, Label(“exit”)).

2-5-3. Data Emission

The function EmitGlobalData is responsible for emitting all the global data in the current scope, as the name suggests. In the assembly code, the data are listed in a standard format. The function’s job is to print the data in this format. Constant symbols in the scope are ignored. The name and type of each symbol is printed. If the symbol is an array, its dimension, the number of elements in each dimension, and its innermost

type are also printed. If the symbol is a string constant, the string is printed. Then EmitGlobalData with each subscope of the current scope as its parameter is called because our compiler supports static local variables.

The function EmitLocalData emits the variables for the current scope unless the current scope is global. It initializes the meta data of all arrays in the current scope on the stack. It is called after a subroutine has been called and the prologue instructions are printed.

2-5-4. Calling Convention

As mentioned in section 2-5-1, EmitScope shows what procedures the compiler takes when a subroutine is called and when it is exited. It computes the stack offsets of the parameters, local variables and temporaries of the subroutine, calculates the stack size and allocates it, then prints a stream of instructions called prologue. After the stream of instructions of the function body is printed, it removes the local data from the stack and prints a stream of instructions called epilogue. Then the subroutine has come to an end.

EmitScope calls the function ComputeStackOffsets, which calculates the stack offsets of local data and returns the required stack size. For ComputeStackOffsets' parameters int *param_ofs* and int *local_ofs*, we assigned 8 and -12 respectively because there are two 4byte values above the parameters, which are the return address and the saved ebp, and there are three callee-saved regs(ebx, esi, edi) below the local variables and temporaries. The returned stack size is the size of the stack occupied by the local variables and the temporaries.

Calculating the stack offset of each parameter was easy because each parameter occupies 4 bytes, no matter the data size of the parameter. For each parameter p, its stack offset is param_ofs+4*(p→GetIndex()). This way, the parameters are passed on the stack in reverse order. Param i+1 is below Param i.

Local variables and temporaries are more difficult to assign stack offsets. This compiler follows 4-byte alignment. Thus the allocation of stack space depends on the size of the data. We first defined a variable called localOffsetCount, which is initialized to local_ofs. One thing to be careful is that unlike param_ofs, local_ofs is an offset that is already occupied, so the local variables and temporaries must start from an offset smaller than it(that is, above that stack point). Let dSize denote the data size of the local variable or the temporary to be passed on the stack. If dSize is less than or equal to 1, or (dSize%4)==(localOffsetCount %4), data is just pushed to the next available space on the stack. No forced alignment is necessary. dSize is simply subtracted from localOffsetCount. If neither is the case, the stack needs to be aligned when the data is pushed. This is implemented by changing the value of localOffsetCount to ((localOffsetCount – dSize -4)/4)*4. Finally the stack offset of the variable is set to localOffsetCount. If a local symbol turns out to be a constant, it is not pushed on to the stack.

All the parameters', local variables', and temporaries' base register is set to "%ebp". The returning value is calculated by subtracting 12 from -localOffsetCount if it is a multiple of 4, and from -((localOffsetCount – 4)/4)*4 if it is not. This is a straightforward implementation of calculating the space on the stack occupied by the local variables and temporaries.

The following are part of the prologue and part of the epilogue worth discussing.

```
# prologue
pushl   %ebp
movl    %esp, %ebp
pushl   %ebx                    # save callee saved registers
pushl   %esi
pushl   %edi
subl    $92, %esp               # make room for locals
```

```
# epilogue
addl    $92, %esp               # remove locals
popl    %edi
popl    %esi
popl    %ebx
popl    %ebp
ret
```

The first line of the prologue is about pushing the current %ebp value on the stack to save it. Then the %ebp value is set to the %esp value, formally announcing the start of a new activation frame. After the new base pointer, the three callee-saved registers, %ebx, %esi and %edi are pushed on to the stack. Then the space for the local variables and temporaries is provided by subtracting the return value of ComputeStackOffset from %esp.

The epilogue kind of looks like the prologue turned upside-down. The stack size(space occupied by local variables and temporaries) is added to the %esp to remove the local data. Callee-saved registers are popped in reverse order in which they were pushed, the saved %ebp is popped, and finally the program returns to the caller. The activation frame is discharged.

3. Test Results

To test our compiler, we first converted all the provided test codes written in SnuPL/1 into assembly codes with our compiler. Then with gcc we converted the assembly codes into object files, linked them with the array and I/O routines and generated executable files. We executed each program to check the results. At first, we ran into a problem because some codes ran into segmentation faults. After debugging with gdb, we realized that parameters of some subroutines were not pushed on to the stack properly because the parser assigned indices of parameters of the same subroutine but of different types separately. As a result, there were overlapping indices. We fixed the parser so that it assigns a parameter's index according to the order of its declaration, independent of its type.

Some programs that take integer inputs, such as the code for calculating the sum of integers from 1 to the input or the one for calculating the factorial of the input, run into segmentation faults when the input is very large. This is also true for the reference binary code, not just our compiler. We concluded that this error was due to the fact that the compiler does not take care of stack overflows.

To sum up, the test code executables all produced the desired results.

4. Conclusion

Our team successfully implemented a fully functioning compiler for the SnuPl/1 programming language. We went through all the provided tests and spent lots of time studying the principles and considering the corner cases. Therefore, we trust that our compiler is reliable.