A Final Report of the SnuPL/1 Compiler Implementation

2013-11431 Hyunjin Jeong

Compilers

Professor: Bernhard Egger

06/10/2016

Index

1. Introduction

A compiler is a program that converts source code in some programming languages into another language. A transformed language is usually having form of binary file. In our project, a language of source code is 'SnuPL/1' and the transformed language is 'assembly language', so we can translate assembly files into machine code using 'gcc' then the result files can be executed by computer.

SnuPL language is an educational language for students generating compiler. The last '1' means its version. In this version, a language specification includes 'array' data structure, also there are changes from SnuPL/0.

Building of SnuPL/1 compiler has 5 phases: 1. Scanning, 2. Parsing, 3. Semantic Analysis, 4. Intermediate Code Generation, 5. Code Generation. The first part, scanning, accepts an input file written in SnuPL/1 language and converts it into tokens. In second phase, the parser accepts tokens from scanner, then makes an Abstract Syntax Tree(AST). In third phase, the type checker takes AST and checks semantic-related errors like type mismatching, using variable before declaration. The intermediate code generation makes a Three-Address Code(TAC) from AST. TAC is an intermediate code, and it looks like assembly code. In the final phase, our compiler takes TAC and converts it into assembly code with AT&T syntax.

2. Scanning

In this phase, I will write which token types I defined, and how I implemented generating tokens especially hard cases like string, character.

2.1. Token types

The basic tokens are tCharacter, tString, tIdent, tNumber, tTermOp, tFactOp, tRelOp, tAssign, tSemicolon, tColon, tDot, tComma, tLBrak, tRBrak, tLSBrak, tRSBrak, tEMark. They are language symbols or basic types. The reserved keywords are tModule, tBegin, tEnd, tTrue, tFalse, tBoolean, tChar, tInteger, tIf, tThen, tElse, tWhile, tDo, tReturn, tVar, tProcedure, tFunction. These keywords like "module", "begin" are used by program. The whole token types and its value can be seen in the table below.

Token type	Value
tCharacter	Character (ex: 'a')
tString	String (ex: "hello")
tIdent	Identifier
tNumber	Number (1, 234, etc.)
tTermOp	+, -. &&
tFactOp	*, /, ||
tRelOp	=, #, <, <=, >, >=
tAssign	:=
tSemicolon	;
tColon	:
tDot	.
tComma	,
tLBrak	(
tRBrak	)
tLSBrak	[
tRSBrak	]
tEMark	!
tModule	Reserved keyword: module
tBegin	Reserved keyword: begin
tEnd	Reserved keyword: end
tTrue	Reserved keyword: true
tFalse	Reserved keyword: false
tBoolean	Reserved keyword: boolean
tChar	Reserved keyword: char
tInteger	Reserved keyword: integer
tIf	Reserved keyword: if
tThen	Reserved keyword: then
tElse	Reserved keyword: else
tWhile	Reserved keyword: while
tDo	Reserved keyword: do
tReturn	Reserved keyword: return
tVar	Reserved keyword: var
tProcedure	Reserved keyword: procedure
tFunction	Reserved keyword: function

2.2 Generating tokens

Most token types are easy to generate. A scanner just compares one character, and generate token if matched. For example, 'tAssign' token is symbol ":=". A scanner first compares input character with ':', then compare next input character with '='. If they are all matched, then a scanner consumes two input characters and generates 'tAssign' token.

The hard cases are 'tCharacter' and 'tString'. The character type is started with ' , and ended with '. The character should have length 1. Therefore, I gave 'tUndefined' token (which means error) with length 2 or more character. But there are exception cases. The escaped characters have length 2. In SnuPL/1, there are 6 escaped characters: ₩n, ₩t, ₩₩, ₩', ₩", ₩0. So 'tCharacter' allows length 2 with cases '₩n', '₩t', '₩₩', '₩"', '₩"', '₩0'. The string type is started with ", and ended with ". It's easier than character. 'tString' has no length limit, so it consumes all characters until second double quote is came. In both tokens, I gave tUndefined token if the input file ended with not closing single quote or double quote.

There is a special case, comment. This language has a line comment. The words after // are handled with comment, and they should not be consumed with token. The picture below is my comment implementation. I used while loop to check, but now I think it's better to use recursive to implement handling comment.

```
362   if(c == '/') {
363     while(_in->peek() == '/') { // consume consecutive comments.
364       while(_in->peek() != '\n') GetChar(); // consume one line.
365       while(IsWhite(_in->peek())) GetChar(); // consume white spaces.
366
367       RecordStreamPosition();
368       c = GetChar();
369       if(c != '/') {
370         if(c == EOF) return NewToken(tEOF);
371         break;
372       }
373     }
374   }
```

3. Parsing

This phase was hardest phase in this project. In this phase, we parse tokens and convert them into Abstract Syntax Tree. Concretely, a parser consumes tokens and makes nodes of AST. I will write how I implemented parsing each part functions of EBNF. I will not write how I implemented

whole functions (it's already written in previous report). I will write some ideas about parsing implementation and problems I suffered, and the solution of the problems.

3.1 LL(2)

Our parser uses LL(1) basically. The LL(k) means leftmost derivation, left-to-right parsing, k-lookahead. LL(1) means a parser peeks 1 token ahead such as if next token is 'tIf', then the parser goes to 'ifStatement'. However, in SnuPL/1 syntax, some parts such as statement cannot be parsed well using LL(1). Therefore, I used LL(2) scheme there.

3.2 FIRST

In this phase, the FIRST() of non-terminals are used frequently. In 'statement', FIRST(statement) is {tIf, tWhile, tReturn, tIdent}. In 'expression', FIRST(expression) is {+, -, FIRST(factor)}, and FIRST(factor) = {tIdent, tNumber, tTrue, tFalse, tCharacter, tString, tLBrak, tEMark}, therefore FIRST(expression) is {+, -, tIdent, tNumber, tTrue, tFalse, tCharacter, tString, tLBrak, tEMark}. If two non-terminals have same FIRST, then I used LL(2) scheme above.

3.3 functionDecl

functionDecl is a part of subroutineDecl. The problem was declaring parameters. Because the function needs its type to define CSymProc, but the position of function type is behind the parameter declaration. Therefore I cannot handle it exactly same with procedureDecl. In procedureDecl, I just called 'varDeclParam' function to declare parameters. So I didn't use functions to declare parameter, but I did it directly in functionDecl. Below is how a parser declares parameters in function.

- There are two vectors to save parameter information: params, vars & vars_loop. 'params' is used to save parameters, and 'vars' and 'vars_loop' are used to save parameters temporarily in loop.

- Each 'varDecl' is each loop, so 'varDeclSequence' consists of loops. In each loop, idents are saving temporarily in 'vars' or 'vars_loop'. After end of each loop, parameters are saved into 'params' with index. Index is used to remember current position of parameter.

- After end of all loops, parameters in 'params' are added into symbol table.

3.4 Getting types

I used 'type' function to get types. First, it consumes one token, which must be 'basetype', then it sets types by using type manager's GetType() functions(Boolean: GetBool, character: GetChar, integer: GetInt).

After that, if next token is 'tLSBrak', it means the type is array. Then a parser consumes array elements, and set array type by using type manager's GetArray() function. If isParam is true, then arrays are addressed by using type manager's GetPointer() function.

3.5 value of constants

SnuPL/1 has three constants: integer, Boolean, character. They are saved in 'CAstConstant' node with its value. The value of integer constants is their value as-is. For example, some integer with value '1234' are saved into CAstConstant with value '1234'. The character constants are saved with their ASCII values. For example, some character 'a' are saved into value '97'. The value of Boolean constants is conventional. 'true' has value 1, 'false' has value 0.

3.6 Type checking in parser

I did some type checking related implementation in this project. My parser checks duplicate parameters, variables, procedures or functions declaration. Also my parser checks usage of undefined variables, parameters, procedures or functions. In constant, the range of integer value was also checked.

Also I implemented some 'GetType()' function in ast.cpp. 'GetType()' functions of 'CAstBinaryOp', 'CAstUnaryOp', 'CAstSpecialOp', 'CAstArrayDesignator', and 'CAstStringConstant' are implemented in parser phase.

4. Semantic Analysis

In this phase, I chose integer constant range check scheme, and did type checking in parser.cpp and ast.cpp. I will write my integer range check scheme and type checking in parser.cpp. Also I will write important type checking and GetType() function in ast.cpp.

4.1 Integer constant range check scheme

I chose 'SIMPLE' scheme. I checked a type of simpleexpr. If the first term of the simpleexpr is CASTConstant and 'integer', sign-folding is implemented. Otherwise, new node 'CAstUnaryOp' is made with its sign.

4.2 Type checking in parser.cpp

My parser checks module/subroutine identifier mismatch, duplicate subroutine/variable declaration, scalar return type check for functions, use before declaration, subroutine calls requires valid symbol, array dimensions provided for array declarations, array dimension constants bigger than 0, integer range is less equal than 2^31 in parser.cpp.

When the parser consumes first module/subroutine identifier, it saves in a token. When the parser consumes second module/subroutine identifier, the parser compares names between first identifier and second identifier. If they are not same, then the parser sets error.

When new ident is consumed in variable declaration, the parser checks if symbol table has a same ident already. If current scope is module, the parser checks global symbol table. If current scope is procedure/module, the parser checks local symbol table. If symbol table has same ident with input, the parser sets error.

The type() function takes parameter 'bool isFunction'.. This Boolean value is true when the function calls type(). If isFunction is true and type is not scalar, then the parser sets error.

When the parser wants to use symbols, the parser checks symbol tables. If both local symbol table and global symbol table don't have symbol, then the parser sets error.

When subroutine calls require symbol, the parser checks symbol table. If symbol tables have symbol but the type is not 'CSymproc', then the parser sets error.

When arrays are declared, the type() function consumes 'tNumber'. If consumed token is not 'tNumber', then the parser sets error. However, arrays in parameter are allowed open arrays. The type() function takes parameter 'bool isParam'. So if 'isParam' is true, then open arrays can be handled. The type() function consumes 'tNumber', which means it only consumes positive constants. (negative sign is added in simpleexpr). Therefore, only positive array dimension constants are allowed.

4.3 Type Check of CAstStatAssign

First the parser checks its left-hand side(LHS) and right-hand side(RHS). Then the parser checks LHS's GetType() is array. If then, the parser sets error because array assignments are not allowed. Then the parser checks LHS and RHS have same type using Match. If not, the parser sets error.

4.4 GetType of CAstBinaryOp

In '+', '-', '*', '/'cases, it returns integer when both LHS and RHS are integer. In '&&', '||'cases, it returns Boolean when both LHS and RHS are Boolean. In '=', '#' cases, it returns Boolean when both LHS and RHS have same type (integer, character, Boolean). In '>', '>=', '<', '<='cases, it returns Boolean when both LHS and RHS have same type (integer, character). Otherwise, it returns null.

4.5 GetType of CAstUnaryOp

In '+'and '-' cases, it returns integer when expression type is integer. In '!'cases, it returns Boolean when expression type is Boolean. Otherwise, it returns null.

4.6 TypeCheck of CAstFunctionCall

First the parser checks arguments number. If the number of arguments is more or less than parameters, then the parser sets error. Then the parser checks each argument expression. Then the parser checks argument types are same with they are defined. If not, then the parser sets error. If all checks finishes successfully, then TypeCheck() returns true.

4.7 TypeCheck and GetType of CAstArrayDesignator

In TypeCheck, First the parser checks symbol type is pointer or array. If not, the parser sets error. Then the parser checks dimension number. If the dimension number is less than index size, then the parser sets error. Opposite case is okay. Then the parser checks each dimension expression and its type. The type of expression should be integer, so if the type is not integer then the parser sets error.

In GetType, the parser first checks a type. If the type is pointer, then we need inside array. If the type is array, then it's good. Then the parser gets inner types of array according to index size.

After the end of for loop, then the parser gets the type we wanted and returns it.

5. Intermediate Code Generation

In this phase, we made Three-Address Code(TAC). In specific, we implemented ToTac() function of ast.cpp. In this report, I will write about toTac of Boolean expressions, and how I implemented difficult parts such as array.

5.1 ToTac() of Boolean expressions

In SnuPL/1, the totac() of boolean expressions is handled specially. Unlike other expressions, the Boolean expressions are not translated to Three-Address Code directly, but they are translated into TAC using 'goto' expression. For example, 'a && b' are translated like below. The point is that the expression '&&' is gone.

```
If a = 1 then goto andtrue
goto lfalse
andtrue:
if b = 1 then goto ltrue
ltrue:
t0 := 1
goto end
lfalse:
t0 := 0
end:
```

5.2 CAstBinaryOp

In operator case '<', '<=', '>', '>=', '#', '=': First it creates labels 'ltrue', 'lfalse', 'end', and 'dummy'. Dummy label is useless but it is used to synchronize label number with the reference

implementation. Then it calls totac() of LHS and RHS and their results are saved into 'left' and 'right'. After that, it adds an instruction with the operator, 'ltrue', 'left', and 'right'. Then it adds jump instruction to 'false' label, and it sets label 'ltrue'. Then it creates a temporary variable and assigns '1' to 'temp' in true case, or assigns '0'to 'temp' in false case. Finally, it returns 'temp'.

In operator case '&&': It creates lables 'ltrue', 'lfalse', 'end', and 'andtrue'. Then it calls totac() of LHS, and sets label 'andtrue'. Then it calls totac() of RHS and sets label 'ltrue'. Then it creates a temporary variable and assigns '1'to 'temp' if true or assigns '0'to temp if false. Finally it returns 'temp'. In operator case '||', the label 'andtrue' is changed to 'orfalse' but the process is similar to '&&' case.

In operator case '+', '-', '/', '*': This case is simple. First it calls totac() of LHS and RHS, then creates temporary variable 'temp'. Then it adds new instruction with the operator, 'temp', and the results of totac(). Finally it returns 'temp'.

5.3 CAstSpecialOp

First it calls totac() of the operand and saves the result to 'src'. Then it sets return type to the pointer of 'src' type. The sub-array types are converted into their basetype. I don't know why but the reference implementation did it same way. Finally, it creates 'temp' and adds an address instruction of 'src' to 'temp'. It returns 'temp'.

5.4 CAstFunctionCall

For all arguments, this function calls totac() of the argument and add them to parameter in reverse order. After that, if the type is NULL (it means procedure/module), it adds a call instruction with no destination and returns NULL. Otherwise, it adds a call instruction with 'temp' and returns 'temp'.

5.5 CAstArrayDesignator

There are two cases: pointer to array, array. In array case, it first creates temp that address of array is saved in it. Then it gets first array index expression by calling its totac(). Then it gets the dimension number. Then it loops according to dimension number. In for loop, it first sets parameters for DIM(A: array, i: i-th dimension). Then it calls DIM and saves the result to temporary variable. Then it multiplies by previous array index expression which saved in 'res_tmp'. Then if an index exists, it calls totac() of an index expression to get new index expression and it adds new

array index expression and saves the result to ‘res_tmp’. Otherwise, it just adds ‘0’. After the for loop, it multiplies the result to the array element size (4 if integer, 1 if Boolean/character). Then it takes parameter for DOFS(A: array), and calls DOFS(A). Then it adds element offset to the data offset, and the address of the array. Finally, it returns ‘CTacReference’ of the result value. The pointer to array case has same procedure with array case, but it doesn’t create temporary variables to use the address of the array.

6. Code Generation

The final phase is making assembly codes from Three-Address Codes. Our target file was backend.cpp. I will write how I implemented functions.

6.1 EmitScope

First it sets target scope to current scope. Then to get stack offset size, it calls ‘ComputeStackoffset’ function, and computed size is saved into ‘size’ variable. Then it emits function prologue. The process is conventional. It pushes %ebp first, then moves %esp to %esp. Then it pushes %ebx, %esi, and %edi in order. Finally, it subtracts computed stack offset size from %esp for local variable rooms.

Then it initializes local stack area. It clears ‘%eax’. When the size is below than 20, it moves %eax to %esp + 0, 4, 8, … The added value is up to computed size / 4. Otherwise, if the size is greater equal than 20, it clears direction flags using ‘cld' function, then it moves $(computed size / 4) to %ecx, then %esp to %edi. Finally, it repeats the process of case of the size below 20 using ‘rep stosl'.

After that, it emits local arrays using ‘EmitLocalData’ function. Then it emits all instructions using ‘EmitInstruction’ function. Finally, it emits function prologue. It adds the compute stack size, then pops registers, then return.

6.2 EmitLocalData

This function emits local arrays. For all symbols of scope, it finds local array symbols. For each array, it first emits dimension number of array. Then for each array dimensions, it emits the element number of dimension.

6.3 Emitinstruction

This function emits each instruction according to the operators: opAdd, opSub, opMul, opDiv, opNeg, opAssign, opAddress, opGoto, opEqual, opNotEqual, opLessThan, opLessEqual, opBiggerEqual, opBiggerThan, opCall, opReturn, opParam, opLabel.

The opAdd, opSub, opMul cases have same process. It loads first source to %eax, second source to %ebx. Then it emits instruction with operator, finally it stores destination to %eax. The opDiv case has similar, but it emits instruction 'cdq' additionally. 'cdq' extends the sign bit of %eax into the %edx register.

In case opNeg, it loads first source to %eax, emits instruction 'negl', then stores destination into %eax. In case opAssign, it justs load first source to %eax and stores destination into %eax. In case opAddress, it finds source address using 'Operand' function. Then computed address is saved into %eax using 'leal'. Then it stores destination into %eax. In case opGoto, it just add jump instruction.

In case opEqual, opNotEqual, opLessThan, opLessEqual, opBiggerEqual, opBiggerThan, it loads two sources into %eax and %ebx, then emits comparing instruction with %eax and %ebx. Then it jumps destination labels.

In case opParam, it loads source to %eax and pushes %eax. In case opReturn, it loads source if exists. Then it jumps to 'exit' label. In case opCall, it first takes parameter number, then it calls destination function name. If parameter number > 0, it emits adding parameter instruction. If destination exists, it emits instruction storing destination into %eax.

6.4 Operand

There are three cases in operand function. In CTacReference case (which means array or pointer), it takes base register of array then moves into %edi. Then the operand name will be "(%edi)". In CTacConst case, the operand name is "$value". In other case, it takes base register and the operand name is "offset(%ebp)". If the symbol is global symbol, the operand name becomes symbol name.

6.5 OperandSize

It takes parameter 'CTac *t'. If t is constant, then it just returns 4. If t is array, then it returns a size of base type. If t is pointer to array, then it returns a size of base type of base type. Otherwise, it returns symbol size.

6.6 ComputeStackOffsets

It takes parameters 'params_ofs' and 'local_ofs'. They are 8 and -12 each. In EmitScope, we pushed %ebp, so params_ofs is 8. Also we pushed three registers %ebx, %esi, and %edi. Therefore local_ofs is -12.

For all symbols, it finds CSymParam or CSymLocal. Then it computes current offset for each symbol. The parameter size is always 4, and the local size is different with its type. After computing current offset, it sets symbol base register to %ebp and sets symbol offset to computed current offset value. The key of this function is 4-byte align. Other parts are straight-forward to implement. In local symbol case, if symbol has type 'Integer' or 'Pointer' or 'Array', then I align stack offset with 4-byte.

After all, it aligns stack offset to 4-byte again. Finally, it prints stack frame information for assembly file.