Project - Compiler

Report – Final

LEWONCZUK Louise – 2020-81806 and MASSARDIER Baptiste – 2020-82265

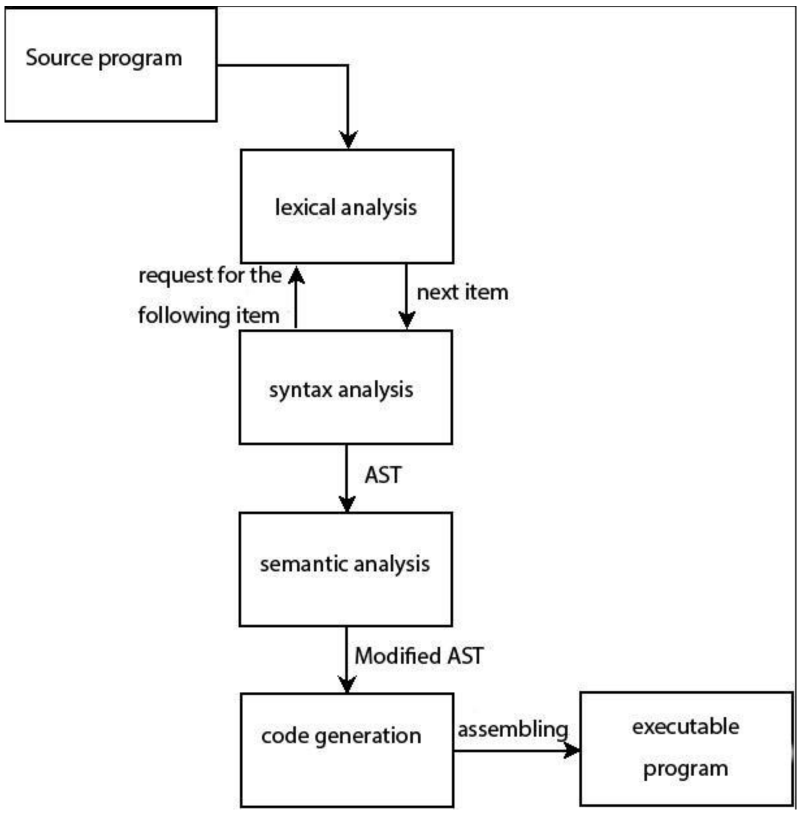
V2 – 2020/11/08

Summary

Introduction

During this project, we coded a compiler for SnuPL/2. The objective of this compiler is to transform a SnuPL/2 program to a x86 64 bits assembly program.

This compiler is divided into distinct phases that work sequentially as follow:



To translate the code, the compiler is subdivided into 4 parts.

The first part is the lexical analysis, it transforms the source program from text to small elementary pieces called tokens.

Then, the second part is the syntax analysis. This part is about the analysis of the tokens stream, to identify the syntax structure of the program.

The third part is the semantic analysis. It analyses the meaning of the program.

Then, the last part is code generation. This corresponds to part 4 and 5 of the project. So, it is the transformation of AST into intermediate code and then, using this intermediate code, generating the object code.

Phase 1: Lexical analysis

This is the first part of our compiler. It is about scanning every character and converting them into the appropriate sequence of tokens, with each token being a meaningful elementary piece of the programming language.

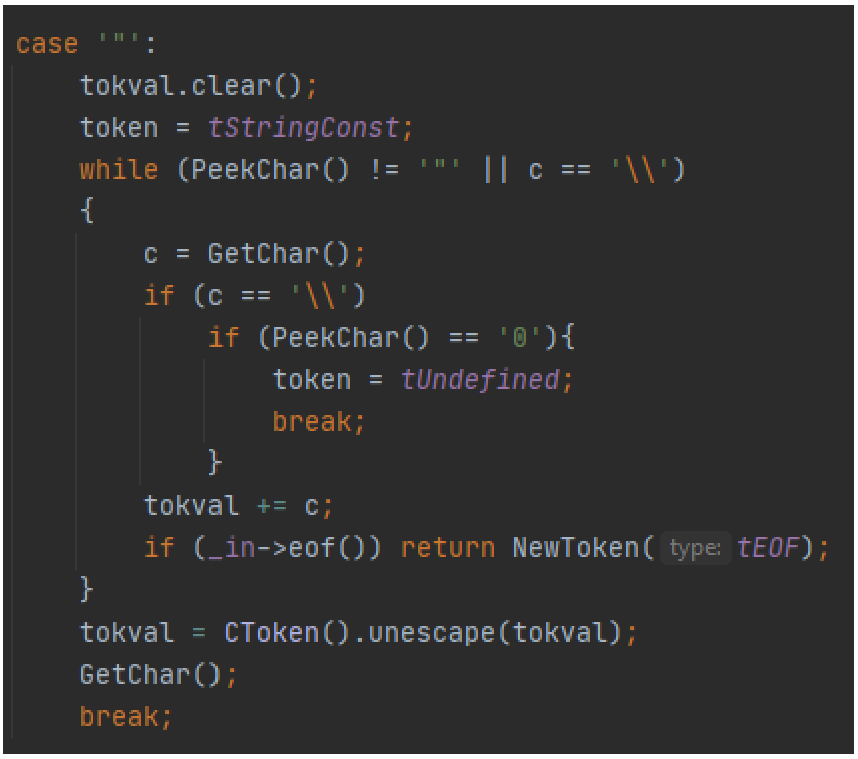
1.1. Implementation

We begin our work by looking at the description of languages SnuPL/-1 and SnuPL/2 to compare them to list tokens that need to be created or modified. We add a certain amount of token but more important, we create a list of keywords linked to the token tKeywords. This list will help us to identify any keyword use by the language SnuPL/2.

We then completed the code by adding a new keyword and modifying the already existing one if necessary.

String management

Being able to correctly parse string is a major challenge of this part. The way our compiler handles it is straightforward:



```
case '"':
    tokval.clear();
    token = tStringConst;
    while (PeekChar() != '"' || c == '\\')
    {
        c = GetChar();
        if (c == '\\')
            if (PeekChar() == '0'){
                token = tUndefined;
                break;
            }
        tokval += c;
        if (_in->eof()) return NewToken( type: tEOF);
    }
    tokval = CToken().unescape(tokval);
    GetChar();
    break;
```

When the program sees a quotation mark, which indicates the start of a string, he enters a while loop until either a quotation mark appears again, or an \0 is found. This is useful to detect string declaration errors in the input program.

Phase 2: Syntax analysis - Parser

In the second phase, we focus on the parser. This is a syntax analysis. The goal of this phase is to create an abstract syntax tree from the sequence of the tokens that we obtained as the output of the first phase, the scanner. This tree highlights the structure of the input text: each token is a terminal, each variable is a leaf node and each production rules of the EBNF define a subtree. Indeed, tokens are the smallest block we consider in our EBNF grammar. We use a top-down parser as the grammar is well-suited for this method. During our implementation, we firstly build a predictive parser that only consumes the tokens and does not build the abstract syntax tree and debug that before coding the creation of the abstract syntax tree.

2.1. First and Follow

In order to create an abstract syntax tree, we firstly followed the tokens in the order they appear. However, we need to be careful about what is the next token before creating the subtree concerning this token. To implement this correctly we had to calculate the First and Follow functions for each production rule of the EBNF.

Those functions also help us to factorize rather than implement it directly to the EBNF while checking and accounting for all the dependencies and thereby result in a far greater simplification. The first and follow statements allow accurate boundaries detection and determination for each ‘part’ (function, statement, or assignment). The ‘first’ easily determines the category depending on the first token we meet. The ‘follow’ helps us to determine the end of a given category.

```
// statSequence ::= [ statement { ";" statement } ].

// statement ::= assignment.

//

// FIRST(statSequence) = { tLetter }

// FOLLOW(statSequence) = { tDot }

//

// FIRST(statement) = { tLetter }

// FOLLOW(statement) = { tSemicolon, tDot }
```

2.2. Production rules detailed study

In parallel to the work done on the ‘first’ and “follow” statements, we decided to spend time on the EBNF to be able to identify the production rules that were called in only one other rule in order to be able to group and code them as one. This leads to major code simplification because there are less functions created, and therefore less function called.

We did have to precisely identify those functions to be precise in their implementation and to include all possible scenarios. We also had to be careful not to use this process too often in order not to over complicate the functions by multiplying the cases. Everything cannot be in the module function.

```
//
// type = basetype | type "[" [simpleexpr] "]".
// basetype = "boolean" | "char" | "integer" | "longint".
//
```

2.3 Creation of the abstract syntax tree

At this stage, every node within the abstract syntax tree includes the token of the element which has been processed, as well as all the elements part of this production rule. For example, for a ‘if’ statement: t, condition, ifBody, elseBody … To do so, we create a class for each and every node possible and assign to a node a given number of parameters which are all the parameters needed to perform the task described.

In this state we build a task and operation tree, and the next step will be to assign values to each leaf in this tree.

2.4 Error checking

Within the scanning phase, we can detect basic syntax errors such as structural code errors. For example, an open loop with no end, an improper line end (no semicolon), mismatched parentheses or brackets structure.

Phase 3: Semantic analysis

After having coded the scanner and the parser in previous parts of the project, which was able to scan the input and give as output a stream of token and transform tokens in an abstract syntax tree. In this part, we will implement the semantic analysis. The goal of this phase is to do a semantic analysis and evaluate constant expression. The output of this function is still an abstract syntax tree but now this abstract syntax tree is usable to generate an intermediate code generation in phase 4.

This signifies that most of the errors have been detected but also that the abstract syntax tree has been as simplified as possible.

3.1. Error checking

At this stage we check for type errors: improper condition (not a boolean statement), generic type error, missing parameters, too many parameters, …

In the previous phase, we already checked for syntax errors, so this stage is focused only on semantic errors which also appends to be more common than the previous ones. This is also where we detect variable allocation type errors like used before assignment or incorrect data.

3.2. Two steps in semantical analysis:

3.2.1 Category dependent type gathering

This part performs type inference. This is a list of methods which return the type of the node. We must implement GetType for: CAstBinaryOp, CAstUnaryOp, CAstSpecialOp, CAstArrayDesignator and CAstStringConstant. The output of this function is one type among the following list: integer, long int, boolean, NULL, pointer, constant. For a given node, the GetType function can return different types depending on the case. Indeed, for a CAstUnaryOp type, the function can return either integer for '+' and '-' or boolean for '!'.

3.2.2 Type checking

In this part, we check that the type we need to have for the node is consistent with our expectations and the rules. To do so, we check for each parameter of the node if it can or not be null, and if every parameter is of the correct type. For example, in the case of an if statement: t, condition -> boolean, ifBody -> empty or statement, elseBody -> empty or statement

3.3 Variable pre-evaluation

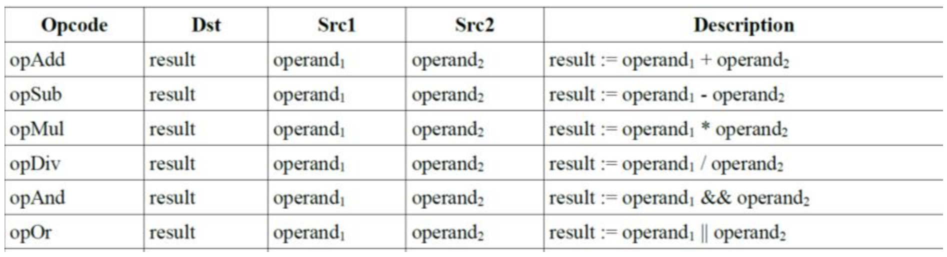
To simplify the next step, and when this can be done quickly, mathematically, and simply, a first evaluation of the constants and variables is done at this stage. This helps reduce the abstract syntax tree a fair bit. Indeed, rather than, for example, keeping subtrees corresponding to simple mathematical operations on numbers for the following phase, we simplify by performing the operation and replacing this subtree with a leaf containing the value of the operation. We can do the same for unary or binary operations and store the value of the operation (a number, a boolean, ...).

Phase 4: Intermediate-Code Generation

This part is done after having created the abstract syntax tree in the precedent phase thanks to the scanning and the parsing done before. In this phase, the goal is to generate an intermediate code. In order to do that, we have implemented the ToTac function of the class composing the abstract syntax tree.

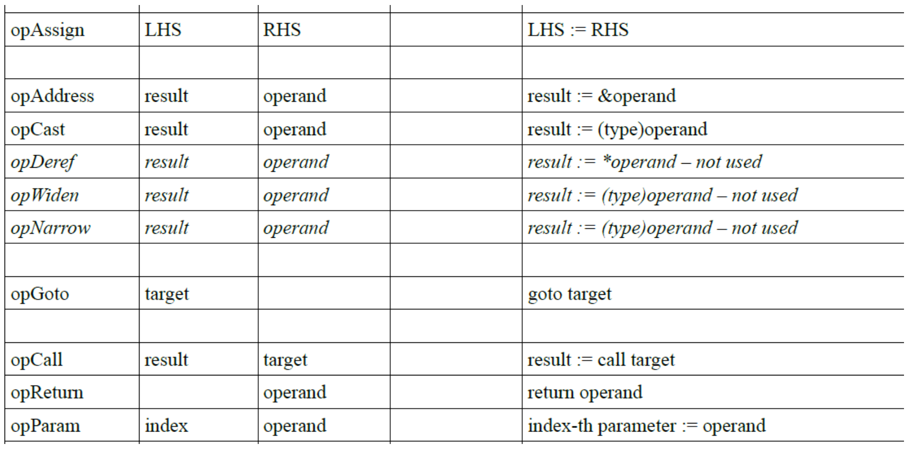
4.1 Conversion of the AST into the Intermediate-Code Generation

The abstract syntax tree is converted into an intermediate representation in three-address code form.



Opcode	Dst	Src1	Src2	Description
opAdd	result	operand$_1$	operand$_2$	result := operand$_1$ + operand$_2$
opSub	result	operand$_1$	operand$_2$	result := operand$_1$ - operand$_2$
opMul	result	operand$_1$	operand$_2$	result := operand$_1$ * operand$_2$
opDiv	result	operand$_1$	operand$_2$	result := operand$_1$ / operand$_2$
opAnd	result	operand$_1$	operand$_2$	result := operand$_1$ && operand$_2$
opOr	result	operand$_1$	operand$_2$	result := operand$_1$ || operand$_2$

Those three-address can be NULL for some OpCode. The OpCode represents the type of operation that will be computed by the computer after the final code génération. We have different mathematical operations but also the assignment, the call, the return, and a lot of different functions.



opAssign	LHS	RHS		LHS := RHS
opAddress	result	operand		result := &operand
opCast	result	operand		result := (type)operand
opDeref	*result*	*operand*		*result := *operand – not used*
opWiden	*result*	*operand*		*result := (type)operand – not used*
opNarrow	*result*	*operand*		*result := (type)operand – not used*
opGoto	target			goto target
opCall	result	target		result := call target
opReturn		operand		return operand
opParam	index	operand		index-th parameter := operand

Moreover, one of the most important OpCode is the opGoto which is a short-circuit code to go to the give target. This is the operation that allows us to jump to a given target for a given condition.

example:

```
CTacAddr* CAstStatement::ToTac(CCodeBlock *cb, CTacLabel *next)
{
    // generate code for statement (assignment, if-else, etc.)


    // jump to next
    cb->AddInstr(new CTacInstr(opGoto, next));
    return NULL;
}
```

4.2 principle of the three-address code:

The goal is to create a list of instructions. Those instructions are composed by the OpCode and three parameters such as this instruction: *AddInstr(new CTacInstr(op, dest, GetLeft()->ToTac(cb), GetRight()->ToTac(cb)))*

- OpCode
- Destination
- Two operands or values

To implement this, you need to generate a lot of labels and use lots of GoTo.

4.3 Presentation of the different class

The different classes of the abstract syntax tree nodes are: Scope, Statement and Expression. We will also developpe the translation of array access. It also handles with expressions, but it has some issues like calling predefined functions, calculating, ...

4.3.1: statement

For statements, it is interpreted by at least one three-address code. Some statements are harder to implement than others. For example, for the ifStatement you need to check the condition to execute either the if body or the else body. Moreover, both bodies can be NULL.

4.3.2: expression

For expressions, it corresponds to a value. However, you may need to create different cases before returning the value. Indeed, this value can be of a lot of different types.

4.3.3: Translation of Array access

This part is mostly composed of one function so we will explain this function:

CTacAddr* CAstArrayDesignator::ToTac(CCodeBlock *cb)

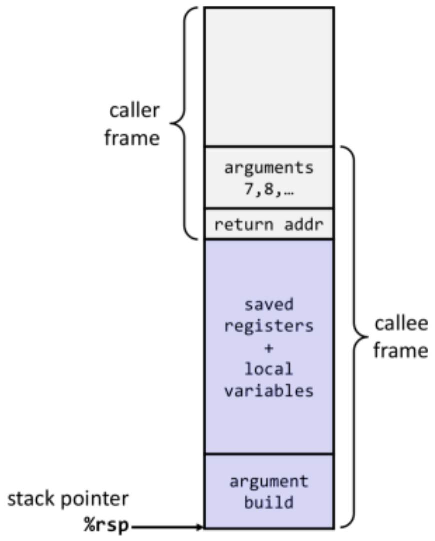
This function is the most complex one. It deals with an array of designators. It computes the pointer of the data and references it to the pointer calculated by the function. The different steps are the following. Firstly, if needed, we link a pure array to a pointer. After that we use the formula of the dragon book as follows: A[i1][i2]...[ik] = (base) + ((... (i1 n2+ i2)n3+ i3) ... nk+ ik) *w

The base corresponds to the element located at &A + DOFS(A) ; nk= DIM(&A, k + 1) for k in range (1, n-1) ; w is the size of the basetype. To do that, we create some temporary variables to store the different designators contained in the array.

At last, thanks to CTacReference, we reference the array location.

Phase 5: Code Generation

This is the last phase of the compiler. This phase is about converting the IR code into x86_64 assembly code with AT&T syntax. The output code is decomposed in 4 different sections: header, code, data, and footer. The header and footer are just here to delimit the code. However, the text section contains main and subroutines of the program. The data section contains all variables used by the program.



5.1. Implementation

The code firstly call the function Emit(CModule *m), which call 4 fonctions for each phase: EmitHeader(), EmitCode(), EmitData(), EmitFooter().

Both EmitHeader() and EmitFooter() simply emit the name of the program and static text.

EmitCode() however, firstly calls every external function used. Then, it calls EmitScope() for each scope.

This function firstly computes the size of locals to organize the stack. However, we did not manage to get this working correctly. This means that our compiler is not able to stock values on the stack.

We still tried to implement the rest of the function. Firstly, it sets up the function prologue by storing saved registers, adjusting the stack pointer to make room for PAF, save parameters to stack, set argument build & local and initialize local arrays. It then emits the code body by using the function EmitCodeBlock, which Emit Instruction for each CTacInstruction.

Finally, the function EmitData() is called and emits global data section.

Conclusion

This project brought us knowledge on many different aspects. The most important contribution was on the management of complex code.

Indeed, the code for the compiler is complex and contains a lot of different modules. The code needs special care about how to be sure that all modules will work fine together. Moreover, we must also pay special attention to error handling, because debugging a code of this size is very difficult.

The organization is also a challenge. Indeed, being able to work two persons on the same time on a huge code require a lot of coordination. On this side we have improved too. We know each other more and especially our strength and weakness so we can split the work more efficiently. We also learn to use GitLab more efficiently.

this project also allowed us to apply the notions seen in class in a very concrete way. this is especially important for a subject like this, which is both complex and very abstract without this project.

However, we unfortunately did not manage to finish the last phase of the project, both due to a lack of time in the last period, but also due to a lack of knowledge of x86 assembler.

Moreover, the last phase reveals all the small errors accumulated since the beginning of the project. We have corrected some of them, but others require a very deep investigation and are therefore very hard to correct. This is due to a lack of testing during the previous phases, which led us to the accumulation of errors without realizing it. So, we understood that code of this complexity requires a lot of testing throughout the programming process.