Report on

Compiler for the while and if-else statement in C language

Submitted in partial fulfillment of the requirements for ***Semester VI***

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Tushar Raj	**PES1201700221**
Kaustubh Jha	**PES1201700040**
Sulabh Mittal	**PES1201700264**

Under the guidance of

C. O. Prakash
Assistant Professor
PES University, Bengaluru

January – May 2020

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

TABLE OF CONTENTS

Introduction

Our mini-compiler is built for a subset of the PhP language (i.e. only while and if-else statements). We have used tools such as yacc, lex and Python scripts to build the complete compiler. An example of what our compiler produces -

Sample input -

```php
<?php
    $s = 0;
    $i = 0;
    while($i < 100){
        $s = $s + $i;
        $i = $i + 1;
    }
?>
```

Our compiler's output -

```
     push  0
     pop   $s
     push  0
     pop   $i
L000:
     push  $i
     push  100
     compLT
     jz    L001
     push  $s
     push  $i
     add
     pop   $s
     push  $i
     push  1
     add
     pop   $i
     jmp   L000
L001:
```

Architecture of Language

Our compiler supports the following language features -

- We handle variables which are of integer type only.
- All types of arithmetic and logical expressions are handled.
- While and if-else statements are also handled.

Literature Survey

- [Yacc Documentation]
- [Lex Documentation]

Context Free Grammar

```
program: START function END
        ;

function: function stmt
          | /* NULL */
          ;
stmt:   ';'
        | expr ';'
        | PRINT expr ';'
        | VARIABLE '=' expr ';'
        | WHILE '(' expr ')' stmt
        | IF '(' expr ')' stmt %prec IFX
        | IF '(' expr ')' stmt ELSE stmt
        | '{' stmt_list '}'
        ;
stmt_list: stmt
        | stmt_list stmt
        ;
expr:   INTEGER
        | VARIABLE
        | '-' expr %prec UMINUS
        | expr '+' expr
        | expr '-' expr
        | expr '*' expr
```

```
    | expr '/' expr          { $$ = opr('/', 2, $1, $3); }
    | expr '<' expr          { $$ = opr('<', 2, $1, $3); }
    | expr '>' expr          { $$ = opr('>', 2, $1, $3); }
    | expr GE expr           { $$ = opr(GE, 2, $1, $3); }
    | expr LE expr           { $$ = opr(LE, 2, $1, $3); }
    | expr NE expr           { $$ = opr(NE, 2, $1, $3); }
    | expr EQ expr           { $$ = opr(EQ, 2, $1, $3); }
    | '(' expr ')'           { $$ = $2; }
    ;
```

Design Strategy

- **Symbol Table Creation -**
  - There are usually multiple scopes in a program.
  - Scope is decided on how many open curly braces we see before we store the variable in the symbol table.
  - Whenever we see a variable declared or initialized with a constant value, we store it in the symbol table.
- **Abstract Syntax Tree -**
  - The abstract syntax tree is generated as we parse the program.
  - A tree node is created based on the type of tokens parsed.
  - We handle three basic types of nodes, that is - constant, identifier and operator.
  - All these nodes are built from bottom up to form the abstract syntax tree.
- **Intermediate Code Generation -**
  - To generate intermediate code, we also make use of the parse tree indirectly.
  - We have written quadruple form of code to a file based on the syntax we are currently parsing in the program.
  - We make use of multiple stacks for this process.
- **Code Optimization -**
  - We have performed reduction in number of live registers and constant folding optimization.
  - We have performed the optimization in C and Python.
  - We analyze the program line by line and use string manipulation and stack to perform these optimizations.
- **Error Handling - strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator) -**
  - The scanner doesn't crash when it comes across unknown symbols.
  - The parser doesn't stop parsing on encountering error and prints a syntax error at the corresponding line number.

    - Semantic analyzer produces an error on uninitialized variables, undeclared variables and re-declaration of variables.
    - The code generator expects error free code to be passed to it.
- **Target Code Generation**
    - Target code is generated using a simple load-use-store mechanism.
    - This is done by looking at quadruple address code line by line.

<u>Implementation Details</u> (Tools and Data Structures Used in order to implement the following):

- **Symbol Table Creation -**
    - We use lex, yacc and custom code to generate the symbol table.
    - We hold the type, value and name of variables in an array of structures.
    - These array of structures are different for different scopes.
    - The symbol table therefore has an array of scopes which store an array of variables.
    - We also use a stack to maintain the current scope.
- **Abstract Syntax Tree (internal representation) -**
    - The abstract syntax tree basically consists of only three types of nodes that we have defined -

```c
/*denotes the type of node in abstract sysntax tree*/
typedef enum { typeCon, typeId, typeOpr } nodeEnum;

/* constants */
typedef struct {                    /*int constant node*/
        int ivalue;                 /* value of int constant */
} conNodeType;

/* identifiers */
typedef struct {               /*identifier node*/
    int i;                     /* index to symbol table array */
} idNodeType;

/* operators */
typedef struct {
    int oper;                  /* operator */
    int nops;                  /* number of operands */
    struct nodeTypeTag *op[1]; /* operands, extended at runtime */
} oprNodeType;

typedef struct nodeTypeTag {
    nodeEnum type;             /* type of node */
    union {
        conNodeType con;       /* constants */
        idNodeType id;         /* identifiers */
```

```
        oprNodeType opr;        /* internal node with an operators */
    };
} nodeType;
```

- These nodes are built bottom up using yacc.

- **Intermediate Code Generation -**
    - We maintain a stack of all the operators and identifiers we parse. There's also a stack for maintaining the labels.
    - When a production symbol is completely parsed, we pop from the stack and print it to a file according to the symbol we parsed.
    - We make use of the labels stack when dealing with the while loop and switch construct. Appropriately consuming the stack to print labels for loop and case statements.
    - Also an arithmetic code generation function, which generates suitable code for any required operator.
- **Code Optimization -**
    - We use Python and basic string manipulation to convert three address code into optimized three address code.
    - For constant folding we just inspect every statement and use raw string manipulation.
    - For dead code removal, we use Python sets to find out variables not being used in the code.
- **Assembly Code Generation -**
    - We use Python and basic string manipulation to convert optimized three address code into target assembly code.
    - We use a hash table just to maintain the required condition variable to be loaded for the while loop.
- **Error Handling -**
    - In the parser, we use yacc's built-in error handling mechanism.
    - In the semantic analyzer we use the symbol table to catch any errors.
- **Instructions for the usage of the compiler -**
    - Write your PhP code in Sample.php.
    - Then run "bash compiler.sh" on your

Results and Shortcomings

- Our compiler is a very minimal and basic compiler, and handles programs which purely perform mathematical computations.
- Error printing of our compiler is not exhaustive and too simple to handle complicated errors.
- Assembly code outputted will be correct for any type of program that our grammar parses. However, the cost of the program is high due to a simple assembly generation algorithm.

Conclusions

- It's very easy to type a command to compile a program, but writing and understanding all phases of a compiler is challenging.
- Powerful tools like Lex and Yacc can be used in order to replicate or build a compiler.
- Working on this project has helped us grasp the internals and all the phases of a compiler.

Further Enhancements

- Handling more data types.
- Handling arrays, pointers etc.
- Function calls and argument parsing.
- More efficient assembly code generator.

References/Bibliography

- If-else - https://www.isi.edu/~pedro/Teaching/CSCI565-Spring10/Lectures/IntermCodeGen.part2.6p.pdf
- Assembly Code Generation - https://web.cs.ucdavis.edu/~pandey/Teaching/ECS142/Lects/final.codegen.pdf
- Course notes.