RedApple Compiler Infrastructure

Xiaofan Sun

Contents

Core Features

RedApple is a fast, usable, C Style Compiler Infrastructure, which not only works independently as a little compiler, but also can be an embedding compiler library.

RedApple supports the following features:

1. Global Functions without Prior Declaration
2. Common Type System
3. Simplified Pointer
4. Constant Fields
5. Control Flow Statements
6. Metadata and Reflection
7. User Macros

Maybe a demo source code will show these features intuitively:

```
int main() {
    // here we used the metadata to call
    // the 'print' function with its name
    FunctionCall("print", 5);
    return 0;
}


// user macro can make your grammar,
// and here we receved three parameters
defmacro for_n (p, size, code)  {
    for (int p = 1; p <= size; p = p+1)
        code;
}

void print(int k) {
    // call the user macro, i and k are two parameters,
    // the block '{}' is a parameter in its entirety.
    @for_n (i, k) {
        printf("hello-%d\n", i);
    }
}
```

Now we are going to discuss the design and how to make it.

Introduction

Most compilers adhere to the basic structural models which is known as Compiler Architecture. These models can be divided into five consecutive processes, known as Lexical Analysis, Syntactic Analysis, Semantic Analysis, and Target Code Generation.

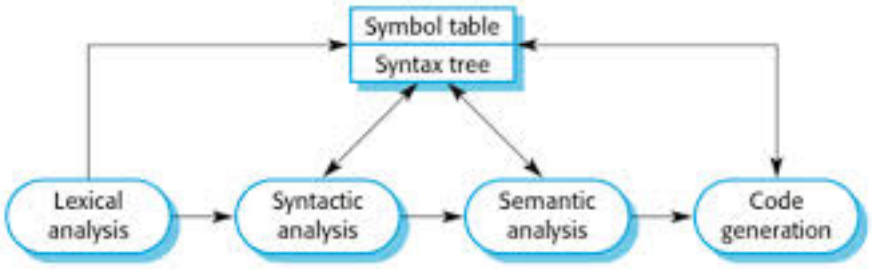

Figure 1: Basic Compiler Architecture

We used LLVM Infrastructure[1] to optimize our intermediate bitcode, and LLC[2] to generate the object code. The optimiztion and target code generation are included in LLC. So our working flow will be like the following Figure 2:

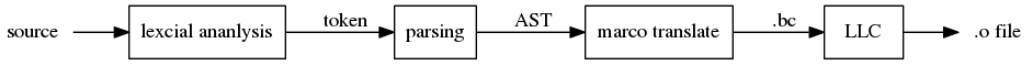

Figure 2: RedApple Compiler Workflow

The special point is that the Semantic Analysis was replaced by Macro Translation, which is used to extend the grammar more conveniently.

When the object files are all created, compiler will call the linker of system to build the executable program. This linker stage has been shown in Figure 3.

[1]LLVM - A collection of modular and reusable compiler and toolchain technologies. (from http://llvm.org/)
[2]LLC - LLVM Static Compiler (from http://llvm.org/docs/CommandGuide/llc.html)

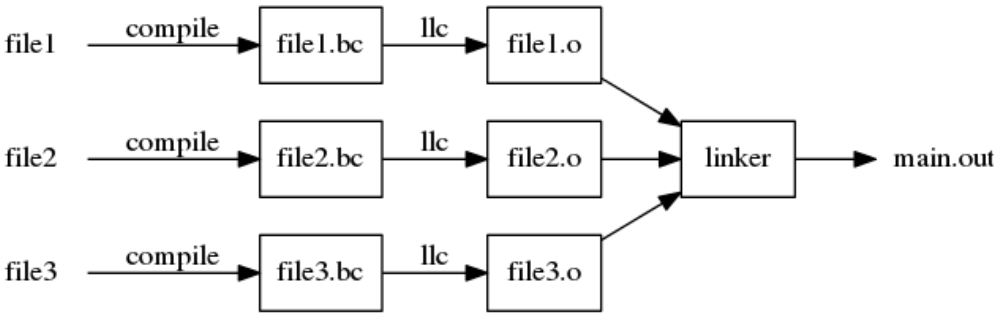

Figure 3: The linker stage

Major Modules

RedApple Compiler have five major modules:

- Lexical analyzer and Parser (generated by Flex and Bison)
  Which is used to analysis the structrue of input files.
- Node Models
  The composition of AST(Abstract Syntax Tree)
- Pass Manager
  Manage to run multi-pass translation
- Marco Translator
  Same as Semantic Analysis
- Low-Level Code Generator
  Unified abstraction layer for LLVM and other Code Generator

The call graph shows their relationships between each other:

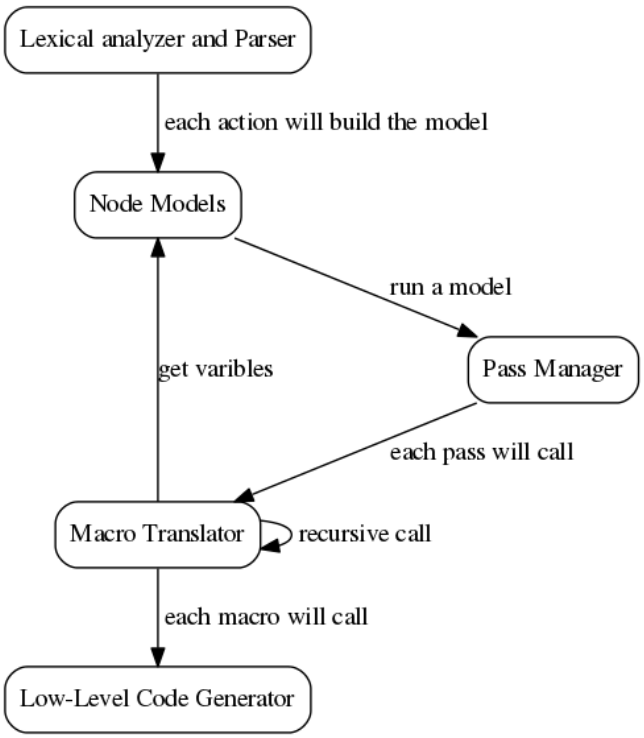

Figure 4: Call graph

Lexical Analysis and Parsing

To make a simple lexical analyzer, we used Flex[3], same as Lex, an unix utility for generating lexical analyzer that perform pattern-matching on text. It reads the Lex configuration file and generate C/C++ code which can be compile to a lexical analyzer library and has approximately the same high performance as hand-written analyzer.

A basic Lex configuration file has three parts which are splited by %%:

```
/* C headers and definitions */

%%

 /* pattern-matching rules of tokens */

%%

/* user's C/C++ code */
```

Another core utility is Bison[4], a parser generator employing LALR(1) parser tables and fully backward compatible with Yacc. It has the configuration file same as Lex and has been designed working together with it. Next, we are going to explain the first two steps and how they are designed.

Here is a basic calculator made by Bison, which parses and calculates the line of expression:

```
/* parser.y */
/* Location tracking calculator.  */

%{
  #include <cmath>
  int yylex (void);
  void yyerror (char const *);
%}

/* Bison declarations.  */
%define api.value.type {int}
%token NUM

%left '-' '+'
%left '*' '/'
%precedence NEG
%right '^'
```

[3]Flex - The Fast Lexical Analyzer (from http://flex.sourceforge.net/)

[4]Bison - GNU Bison is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employing LALR(1) parser tables.(from http://www.gnu.org/software/bison/)

```
%% /* The grammar follows.  */

input:
  %empty
| input line
;

line:
  '\n'
| exp '\n' { printf ("%d\n", $1); }
;

exp:
  NUM           { $$ = $1; }
| exp '+' exp   { $$ = $1 + $3; }
| exp '-' exp   { $$ = $1 - $3; }
| exp '*' exp   { $$ = $1 * $3; }
| exp '/' exp
    {
      if ($3)
        $$ = $1 / $3;
      else
        {
          $$ = 1;
          fprintf (stderr, "%d.%d-%d.%d: division by zero",
                   @3.first_line, @3.first_column,
                   @3.last_line, @3.last_column);
        }
    }
| '-' exp %prec NEG     { $$ = -$2; }
| exp '^' exp           { $$ = pow ($1, $3); }
| '(' exp ')'           { $$ = $2; }

%%

int
main (void)
{
  return yyparse ();
}
```

We find that all the tokens are undefined, as a matter of fact, which will be defined in the configuration file of Lex.

```
/* scanner.l */

%{
```

```
#include "parser.hpp"
#include <cstdlib>

#define SAVE_TOKEN     yylval = maketoken(yytext, yyleng)
extern "C" int yywrap() { return 1; }
int maketoken(const char* data, int len);
%}

%%

[0-9]+                  SAVE_TOKEN; return NUM;

"("                     return '(';
")"                     return ')';
"+"                     return '+';
"-"                     return '-';
"*"                     return '*';
"/"                     return '/';
"^"                     return '^';

%%

int maketoken(const char* data, int len) {
    char* str = new char[len+1];
    strncpy(str, data, len);
    str[len] = 0;
    int num = atoi(str);
    delete str;
    return num;
}
```

Each token defined by the 'parser.y' file will be a number which the pattern-matching returns in the 'scanner.l' (A char also can be known as a number).

The Difficulties in Lexical Analysis

One of the difficulties is analysing the comment such as `/* */`. We need to enumerate all the possiblities in the Regex. The classic solution looks like:

```
%%
"/*"([^\*]|(\*)*[^\*/])*(\*)*"*/" ; /* the comment likes this kind */
 /* => */
"/*"   (  [^\*]   |   (\*)* [^\*/]   )*   (\*)*    "*/" ;
```

Returning anything means that lexical analyzer will discard comment statements.

Some other important regexs have been listed below:

```
%{
#define SAVE_TOKEN     yylval.str = maketoken(yytext, yyleng)
#define SAVE_STRING    yylval.str = maketoken(yytext, yyleng)
#define SAVE_STRING_NC yylval.str = maketoken(yytext, yyleng)
%}

%%

/* some pattern-matching regexs */

[a-zA-Z_][a-zA-Z0-9_]*  SAVE_TOKEN; return ID;

-?[0-9]*\.[0-9]*        SAVE_TOKEN; return DOUBLE;
-?[0-9]+                SAVE_TOKEN; return INTEGER;
0x[0-9A-Fa-f]+          SAVE_TOKEN; return INTEGER;

\"(\\.|[^\\"])*\"       SAVE_STRING; return STRING;
@\"(\\.|[^\\"])*\"      SAVE_STRING_NC; return STRING;
\'(\\.|.)\'             SAVE_STRING; return CHAR;
```

The Generator of Common Nodes Tree

Abstract Class, which is the most usage of the functions to building the AST, can derive types of each kind of leaf nodes. Those nodes may represent one kind of statements in our grammar, a simple expression, or just a number.

For the consideration about that our grammar should have extensibility, we won't arrange the statement node, and all kinds of the node types will be a basic constant or a data type.

We just defined some basic nodes for identity, types and constant fields:

- IDNode - which will contain an id-name
- TypeNode - type strings like "int[]"
- IntNode - just a integer
- StringNode - raw string
- FloatNode - a float point number

These nodes enable to arrange to a list liking the code in lisp.

Here is the nodes tree of a main function:

```
(function main () (
    ...
    (return 0)
))
```

Some other useful statements will be listed below:

```
(int x 10)  ; define varibals

(* (+ a b) (- 10 3)) ; expression

(for (int i 0) (< i 10) (++ i) ; for statement
    ...
)

(if (() ; if-else statement
    (< i j)
    (++ i))
    (return i))
```

The model layer can be generated by its constructive functions which is named by 'Create'. Here is an example of if-statement:

```
if_state : IF '(' expr ')' statement
          { $$ = Node::make_list(3, IDNode::Create("if"), $3, $5); }

         | IF '(' expr ')' statement ELSE statement
          { $$ = Node::make_list(4, IDNode::Create("if"), $3, $5, $7); }
         ;
```

The function `Node::make_list`, which is common in use, can create a list by passing a variable number of parameters.

Figure 5 shows the inheritance relationship between the models.

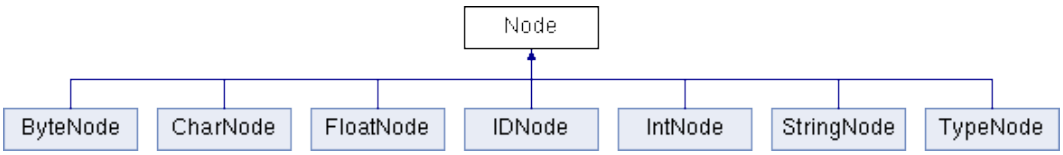

Figure 5: Inheritance relationship diagram

Traverse of The models

When the models have been created, the AST was confirmed. We can crawl the each node one by on or visit the node we need on its own.

`getChild()` and `getNext()` are used to traverse the AST, a left-child right-sibling binary tree.

Here is an example abour for-statement in C style:

```
for (i = 0; i < j; ++i)
    return i;
```

in lisp style:

```
(for (           ; for-else statement
        (= i 0) ; init
        (< i j) ; condition
        (++ i)) ; action
    (return i)) ; body
```

The AST:

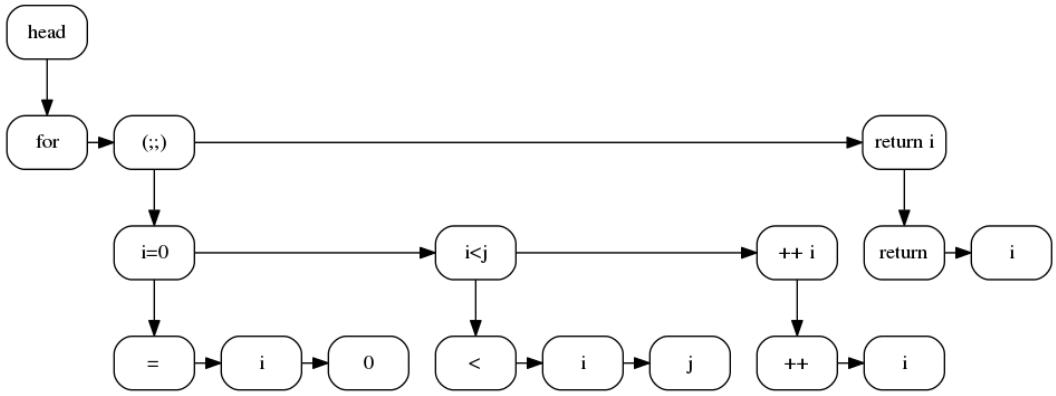

Figure 6: A left-child right-sibling binary tree

Macro Translation

Macro Translation is an important stage for RedApple Compiler replacing the Semantic Analysis stage and having an IR generator. It can run the multi-pass macro scanner to get the definitions of functions and types.

It's a recursive process to run the multi-pass macro scanner, that figure 7 has shown.

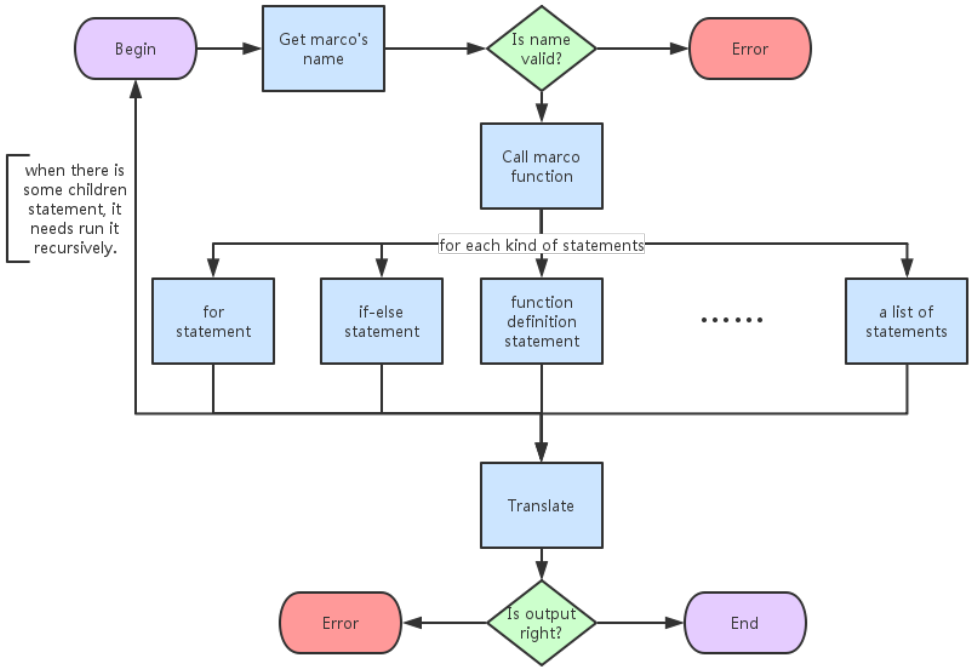

Figure 7: Flow Diagram of Macro Translation

All marco functions need register before used, since this design let the module have a expansibility for user's new marco.

Order Problem of the Macro Scanning

It's hardly to make a one pass compiler whick grammar looks like Java. We can't do all thing in one pass.

Think about this example:

```
void run(int p) {
    hard(p-1);
    ...
}

void hard(int p) {
    if (p==0) return;
    run(p-1);
    ...
}
```

The two functions, `run` and `hard`, called each other and you can't scan any one firstly.
So our idea is to scan all the global names first, and collect the information about them. Their names will be stored in the Symbol Table, the bottom of the symbol table in stack fashion.

Before it, when there is a struct defined beside the function:

```
complex Add(complex x, complex y) {
    complex ans = new complex();
    ans.real = x.real + y.real;
    ans.imag = x.imag + y.imag;
    return ans;
}


struct complex {
    double real;
    double imag;
}
```

So, before we scan the functions, all types should be stored in Symbol Table.

Another pass need to preproccess is User Marco Pass, which is used to find the Marco defined by user and to perform macro substitution on AST nodes.

Here is the passes definition code in this project, which can be divided into four major passes. Each of the passes has the different kinds of marco-functions.

1. User Marco Pass
2. Scanning Types Pass

3. Scanning Functions Pass
4. Main Pass

```
extern const FuncReg macro_funcs[];
extern const FuncReg macro_classes[];
extern const FuncReg macro_prescan[];
extern const FuncReg macro_pretype[];
extern const FuncReg macro_defmacro[];

void PassManager::LoadDefaultLists() {
    list<Pass*> prescan = {
        new Pass(macro_defmacro),
        new Pass(macro_prescan),
        new Pass(macro_pretype) };
    list<Pass*> main = {
        new Pass(macro_funcs, macro_classes) };
    NewPassList("prescan", prescan);
    NewPassList("main", main);
}
```

`LoadDefaultLists` has defined two lists, one is the prescan list, the other is the main list.

User Marcos and the Replacement

Definition of User Marcos is the feature of RedApple Compiler Infrastructure, which let user make their own grammar.

```
void print(int k) {
    @for_n (i, k) {  // call user macro
        printf("hello-%d\n", i);
    }
}

defmacro for_n (p, size, code)  {  // define user macro
    for (int p = 1; p <= size; p = p+1)
        code;
}

int main() {
    print(5);
    return 0;
}
```

This code will be translate to the code below, each name in the macro definition will be replaced:

```
void print(int k) {
    for (int i = 1; i <= k; i = i+1) {
        printf("hello-%d\n", i);
    }
}

int main() {
    print(5);
    return 0;
}
```

Each user macro call should begin with a `@`, following the macro name, we pass arguments to it in the `()`, but the block `{}`, will be recognized as an entirety.

In one user macro call, we can pass lots of arguments until it's not `id`, `(` or `{`;

Here is our bnfs for the user macro grammar:

```
<macro_def_args> = [id:id] {{ return newIDNode(id.val); }}
                 | <macro_def_args:args> "," [id:id]
                 {{ addBrother(args, newIDNode(id.val));  return args; }}
                 ;

<marco_def> = "defmacro" [id:id] "(" <macro_def_args:args> ")" <block:b>
               {{return makeList(newIDNode("defmacro"), newIDNode(id.val), args, b);}}
            ;

<macro_call> = "@" [id:id] {{ return newIDNode(id.val); }}
             | <macro_call:m> "(" <macro_call_args:args> ")"
             {{ addBrother(m, args); return m; }}
             | <macro_call:m> <block:b> {{ addBrother(m, getList(b)); return m; }}
             | <macro_call:m> [id:id] <block:b>
             {{ addBrother(m, newIDNode(id.val)); addBrother(m, getList(b)); return m; }}
             ;
```

Meta Model of RedApple Grammar

We support simplied C style grammar, like functions and structs. An obvious difference is that RedApple has bound meta data for reflection.

Each kind of the meta data has a class inherited from `MetaModel`, which can store all of the information about it.

When we have generated the code, those information will be written by lists in the `meta.bc`. All of the meta data which has a common constructor function in C++ style, will be the initialization of the program.