This repository Search Pull requests Issues Gist

gangliao / Tiger-Compiler Private

Unwatch 2 Unstar 1 Fork 0

Code Issues 0 Pull requests 2 Projects 1 Wiki Pulse Graphs Settings

implement a full compiler based on c++ 11

Edit

compiler parser codegenerator scanner Manage topics

209 commits 9 branches 1 release 2 contributors Apache-2.0

Branch: master New pull request Create new file Upload files Find file Clone or download

gangliao update docs Latest commit a9622f6 an hour ago

img	Add symbol table design	5 hours ago
phase1/part1	Add init parse table	21 days ago
src	Add exit	5 hours ago
testCases/test-phaseI	clang-format	19 hours ago
.gitignore	Add dump	7 days ago
.pre-commit-config.yaml	Add clang-format	21 days ago
.travis.yml	Add CI system (#5)	a month ago
LICENSE	Initial commit	a month ago
Makefile	Add readme	8 hours ago
Phase1-Testing and Output.pdf	updated test case 22 and 23 output.	18 hours ago
README.md	update docs	an hour ago

README.md

Tiger Compiler - Front End

How to Build

1. development environment

   Currently, this project repository is maintained on github privately and also been deployed on Travis CI. It supports both Ubuntu and Mac OS X.

2. build:

```
# cd project dir
cd Tiger-Compiler
# build scanner and parser
make
```

3. run:

   You can parsing test cases named `*.tiger` under `/testCases/test-phaseI` to generate IR code.

```
# naive mode
bin/parser <filename>
# verbose mode: "-d to implement a verbose mode"
bin/parser <filename> -d
```

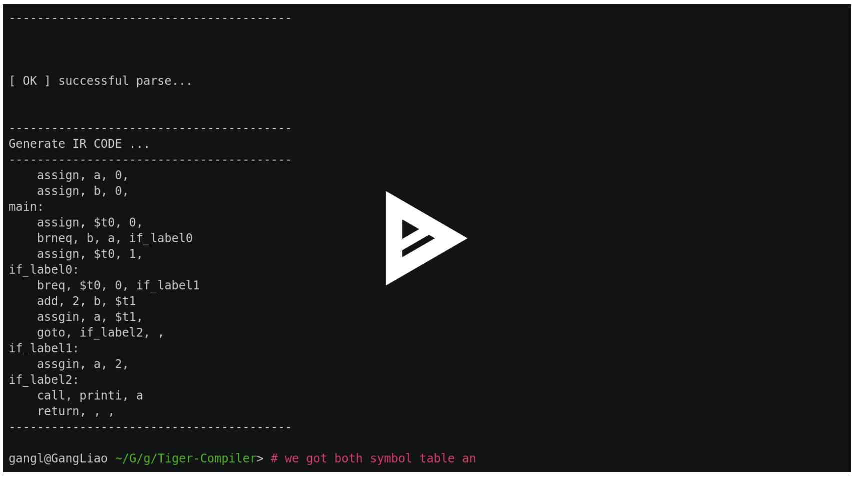
4. test:

In `/testCases/test-phaseI` directory, it includes a test script `test.sh` to execute all of 32 test cases and generate the corresponding ouput file `*.tiger.txt`.

You can simply issue the commands:

```
cd testCases/test-phaseI
sh ./test.sh
```

Demo



Desgin Internals

Design LL(1) Parse Table

1. Hand-modified Tiger grammars

   First, we need to rewrite the grammar given in the Tiger language specification to remove the ambiguity by enforcing operator precedences and left associativity. This part is done by hand. You can check out our modified grammar file in this repo.

2. Hand-written parse table

   Modifying the grammar obtained in step 1 to support LL(1) parsing. This could include removing left recursion and performing left factoring on the grammar obtained in step 1 above. Creating the LL(1) parser table for Tiger. This will drive the decision-making process for the parser. This part is also done by hand by using the theory of LL parsing and finding the first(), follow() sets that help you develop the parser table (please check out parser table file in this repo.)

3. Parser code

   After hand-written parser table is created, it should be hand-coded into our program. we create a data structure - hash table:

   ```
   std::map<SymbolTerminalPair, std::vector<int> > parseTable_;
   ```

   Here, class `SymbolTerminalPair` includes a pair members `(Entry entry, std::string name)`, `std::vector<int>` in `parseTable_` is the actual expansion grammar rules.

   To build a parse table, we can simply insert all next terminals with their grammer rules with action symbol into hash table `parseTable_`.

   As a simple example, consider the following:

   ```
   /// insert items into parse table
   void Parser::addToParseTable(const int nonterm,
                                const std::vector<int>& terminals,
                                const std::vector<int>& expand_rule) {
   for (auto& term : terminals) {
       SymbolTerminalPair stp(nonterm, term);
       parseTable_[stp] = expand_rule;
   ```

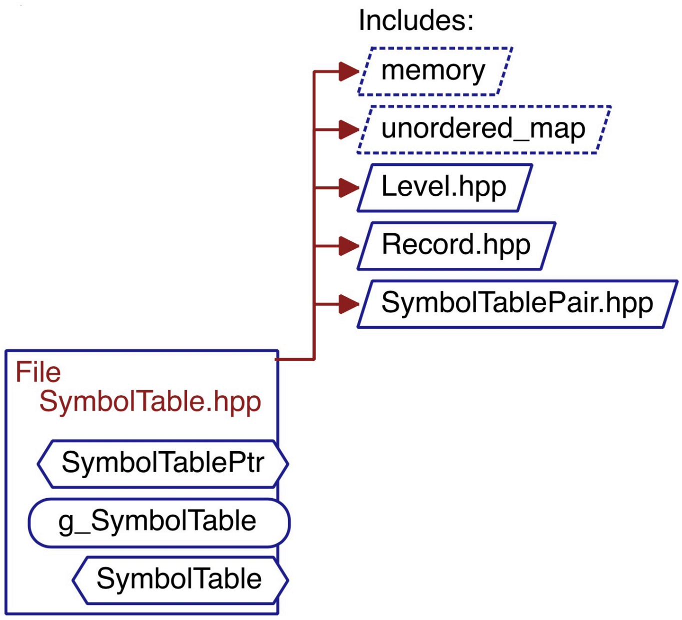
```
}

// # tiger-program
// 1: <tiger-program> -> let <declaration-segment> in <stat-seq> end
addToParseTable(Symbol::Nonterminal::TIGER_PROGRAM,         // NOLINT
                {Symbol::Terminal::LET},                    // NOLINT
                {Symbol::Action::InitializeScope,           // NOLINT
                 Symbol::Terminal::LET,                     // NOLINT
                 Symbol::Nonterminal::DECLARATION_SEGMENT,  // NOLINT
                 Symbol::Terminal::IN,                      // NOLINT
                 Symbol::Action::MakeMainLabel,             // NOLINT
                 Symbol::Nonterminal::STAT_SEQ,             // NOLINT
                 Symbol::Terminal::END,                     // NOLINT
                 Symbol::Action::FinalizeScope});           // NOLINT
...
```

The next terminals are inside `std::vector<int> &terminals`, their grammar rules with action symbol are part of third parameter `std::vector<int> &expand_rule` in `addToParseTable`.

In general, combining `addToParseTable` and hand-written parse table, we can embed parse table into program before it starts parsing.

Symbol Table



Since `let` statements can be nested as per the grammar, Scoping-sensitive data structure is required to be stored in the different level symbol tables. For convenience and simplicity, we create a global data structure `g_SymbolTable` : `int`, which is the current scoping level and `SymbolTablePtr` is a c++11 shared ptr which refers to the corresponding symbol table.

```
/// global symbol table <level, symbol table>
std::unordered_map<int, SymbolTablePtr> g_SymbolTable;

/// initialize Scop
inline void initScoping() {
    ++currentLevel;
    SymbolTablePtr st = std::make_shared<SymbolTable>(currentLevel);
    g_SymbolTable[currentLevel] = st;
}

/// finalize Scope
inline void finalizeScoping() {
    g_SymbolTable[currentLevel]->dump();
    g_SymbolTable.erase(currentLevel);
    --currentLevel;
}
```

Each `let` statement opens a new scope which ends at the corresponding end of the `let` statement. When a new scope is opened, new symbol table based on incremental level will be initlized. Since current Tiger grammar rules only support `int` and `float`, we only embed `int`, `float` and the related standard functions like `printi`, `flush`, `exit`, `not` into symbol table.

When you execute `bin/parser <filename> -d`, the symbol table will be generated on your screen.

For example, issue the command `./bin/parser testCases/test-phaseI/test1.tiger -d`:

The symbol table is shown as follows:

```
[ RUN ] parsing code...

----------------------------------------
Table: Variables
Name: $t0
----------------------------------------
Scope: 0
Type: int
Dimension: 0
Parameters: -
Parameter types: -
Parameter dimensions: -
Return type: -


----------------------------------------
Table: Variables
Name: $t1
----------------------------------------
Scope: 0
Type: int
Dimension: 0
Parameters: -
Parameter types: -
Parameter dimensions: -
Return type: -

...

----------------------------------------
Table: Functions
Name: printi
----------------------------------------
Scope: 0
Type: -
Dimension: -
Parameters: [i]
Parameter types: [int]
Parameter dimensions: [0]
Return type: -
----------------------------------------

[ OK ] successful parse...
```

Semantic Checking

In our implementation, semantic checks are also performed. It leverages action symbols, semantic records on the stack and symbol table.

There are several cases in Tiger where type checking must occur:

```
1. Agreement between binary operands.
2. Agreement between funtion return values and the funtion's return type.
3. Agreement between funtion calls and the funtion's parameters.
4. Redeclaration of same variable.
5. Error nous comparison operator.
6. Test for printi with float value.
7. Test for inbuilt function ‘exit’ with wrong parameters.
8. Multiple let-in-end test.
10. For loop expression with float parameter.
```

We already added these negative test cases in directory `/testCases/test-phaseI`, please refer to our Phase1-Testing and Output to find more details.

As a simple example, consider `test32.tiger` in directory `/testCases/test-phaseI`, Its tiger code is as follows:

```
/* test for loop expression type as float, this should generate error */
let
        type ArrayInt = array [100] of int; /* Declare ArrayInt as a new type */
        var X, Y : ArrayInt := 10; /* Declare vars X and Y as arrays with initialization */
        var a, i : int := 0;
        var b : float := 10.0;
in
        for i := a to b do /* Error: b is float */
                sum := sum + X[i] * Y [i];
        enddo;
        printi(sum); /* library call to printi to print the dot product */
end
```

After issuing the command `./bin/parser testCases/test-phaseI/test32.tiger -d`:

```
[ RUN ] parsing code...

let type id = array [ intlit ] of int ; var id , id : id := intlit ; var id , id : int := intlit ; var id : float :=
loatlit ; in for id := id to id do

Error: for statement begin or end value is not int type!
```

Intermediate Code

1. new labels

To generate intermediate code, we need helper functions like `new_temp()`, `new_loop_label()` and `new_if_label()` to generate unique labels in IR code.

1. action fuctions

   Program starts parsing tiger code, when action symbols are detected, corresponding functions to do semantic checking and generate IR code generation are triggered.

   Here is some action functions:

```
/// parse action like TYPES, VARIABLES, FUNCTIONS declaration
void parseAction(int expr, std::vector<TokenPair> &tempBuffer);

/// parse for statement action
void parseForAction(std::vector<TokenPair> &blockBuffer);

/// parse for statement end action
void parseForActionEnd(std::vector<TokenPair> &blockBuffer);

/// parse function action: function name (x:int) : return-type
void parseFuncAction(std::vector<TokenPair> &tempBuffer);

/// parse if statement action
void parseIfAction(std::vector<TokenPair> &tempBuffer);

/// parse return statement action
void parseReturnAction(std::vector<TokenPair> &tempBuffer);

/// parse while statement action
void parseWhileAction(std::vector<TokenPair> &tempBuffer);

...
```

2. evaluate expression

   The toughest part is to generate IR code for expression or expression assignment. Because it could includes `+, -, *, /, &, |` and `(, )`.

   For instance, how to generate IR code for `a := (b + 2) / 5 * a`? We use the postfix expression to generate IR code: http://faculty.cs.niu.edu/~hutchins/csci241/eval.htm

```
1. convert infix expression to postfix expression
2. evaluate postfix expression to semantic checking and IR code generation
```

```
/**
* @brief parse expression from infix to postfix expression.
*
* @note  posifix expression is convenient way to do semantic
*        check and generate IR.
*/
std::vector<TokenPair> cvt2PostExpr(std::vector<TokenPair> &tempBuffer,
                                    size_t index);

/// generate IR and symbol table elements from postfix expression
TokenPair evaPostfix(std::vector<TokenPair> &expr);
```

Finally, we can generate the code as follows:

```
    add, 2, b, $t0
    div, 5, $t0, $t1
    mult, a, $t1, $t2
    assgin, a, $t2,
```