Report for Compiler Course Project

Qinglin Li
Shanghai Jiaotong University
lostleaf@icloud.com

Abstract

This report describes the compiler course project. The design of Abstract syntax tree and immediate representative along with some optimization are included in this report.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: compiler

Keywords Compiler, Abstract Syntax Tree, Immediate Representative, Code Optimization, Register Allocation, Constant Folding

1. Introduction

This project aims at building a compiler for a subset of C language. It removed float numbers, some confusing grammars and most library functions in C language. And, of course, the compiler translate C code to MIPS code with ANTLR4 parser generate tool.

2. Abstract Syntax Tree

The Abstract Syntax Tree(AST for short) is generated while parsing and the whole process is contained in *C.g4* file under *parser* directory.
The AST is similar with Parsing Tree, but removes useless information. Every node in Parsing Tree is corresponded to a node in AST.
And since addition, multiplication, and other binary operator expressions are similar, a generic type is used here. And it brings much benefit when generate Immediate Representative.
The inheritance of AST is shown below:

- Node
  - Program
  - Declaration
  - ...
  - Stmt(Correspond to Statement)
    - CompStmt(Correspond to Compound-Statement)
    - ...
  - Expression
    - AssExpr(Correspond to Assignment-Statement)
    - BinExpr$<$ ExprType $>$(generic type)
      - AddExpr:BinExpr$<$ MulExpr $>$
      - MulExpr:BinExpr$<$ CastExpr $>$
      - ...

3. Semantic Checking

The semantic checking producure is called after AST generating. Semantic checking mainly check the following items:

1. Type
2. Left value
3. Declaration and use before declaraed
4. Other items including breaks, returns, etc.

3.1 Type

Types all have upcase class names in case of mixing up with Java type names.
The inheritance:

- TYPE
  - CHAR
  - FUNCTION
  - INT
  - VOID
  - STRING
  - NAME
  - POINTER
    - ARRAY
  - RECORD
    - UNION
    - STRUCT

CHAR, **INT**, and **VOID** are singleton classes.
Type checking mainly happened in expressions and some statements.
In expressions, if the oprands and operators doesn't match, a error would be reported. For instance, if a structure is multiplied by an integer, a "type not match" error would be reported.
And some statements require special types. For instance, the condition of **if** or **while** statements must be integer.

3.2 Left Value

Most left values checking happens in assignments. And some operator such as "&"(get address) and "++"(self increment)

3.3 Declaration

The check about declaration and use before declarated is based on symbol table. If a variable cannot be found in the symbol table while used, a "variable not declaraded" error would be reported. The uniqueness would also be checked.

3.4 Returns and Breaks

Returns and breaks are checked by some counters.

3.5 Other items

The details not metioned can be found in *Semantic.java* under *semantic* directory

4. Immediate Representative(IR)

4.1 Temp

- Addr: an interface all temps need to inplement
  - Temp: regiseters
  - Reference: correspond to memory space to support pointers.
  - Label: labels in MIPS
  - IntConstant: integers
  - AddrList: to initialize structures or arrays

4.2 Quad

- Binop: binary expressions
- Branch: branch with condition
- Call: function call
- Enter: enter a function, save registers
- Goto: branch with no condition
- IfFalse: bnez
- IfTrue: beqz
- LabelQuad: label, packaging of Label in temps
- Leave: leave a function, load registers
- Move: assignment, **move** in MIPS
- Return: assign the register for returen value
- Unaryop: unary operator expressions including **-a**, **~a** and **!a**

5. Code Optimization

5.1 Combine Assigntment and IfTrue/IfFalse

Such a sequence of instructions always appear:

```
slt $t3, $t1, $t2
bnez $t3, LABEL
```

It can be combined:

```
bge $t1, $t2, LABEL
```

So, if the result of a comparison expression like **slt**(*Binop with* <) is used by a following **beqz**(*IfTrue*) or **bnez**(*IfFalse*), the two codes can be combine to one *Branch*—- **bge**(*Branch with 'greater or equal'*) in this case.

5.2 Constant Folding

Constant folding is an easy but useful optimization. The sequence shown below can be calculated while compiling:

```
li $t1, 1
li $t2, 2
add $t3, $t1, $t2
li $t4, 3
mul $t5, $t3, $t4
```

This is MIPS code sequence for

$$t5 = (1 + 2) * 3$$

If **t5** is replaced with the constant 9, both registers and instructions are reduced.
This happens while generating IR, if a *Binop* has both constant operands, the target would be replaced with a constant of the calculated result.
Constant Folding reduced about 170,000 instructions for queens testcase.

5.3 Register Allocation

Regiseter Allocation is the most powerful optimimation, and the linear scan algorithm is used here.

6. Other work

6.1 Variable-length array

Since the length of an array can either be an constant or an temp(register).
If the length of the array is an constant, the space would be directly allocate in the stack, else the **malloc** function would be called.

6.2 Translate wikipedia

A. Full inheritance of AST

- Node
- Program
- InitDeclarator
- PlainDeclarator
- Parameters
- Initializer
- InitDeclarators
- Declaration
- Arguments
- FunctionDefinition
- Declarators
- TypeName
- PlainDeclaration
- Stmt
  - SelStmt
  - ExprStmt
  - CompStmt
  - IterStmt
    - ForStmt
    - WhileStmt
  - JumpStmt
    - ReturnStmt
    - BreakStmt
    - ContinueStmt
- Expression
  - Expr
  - AssExpr
  - PriExpr
  - BinExpr
    - ShiftExpr
    - AddExpr
    - AndExpr
    - EquExpr

- LogOrExpr
- InOrExpr
- RelExpr
- LogAndExpr
- MulExpr
- ExOrExpr

- Constant
  - CharConst
  - IntConst
- Postfix
  - SelfDecPostfix
  - ArrPostfix
  - PtrAttrPostfix
  - FunPostfix
  - ValAttrPostfix
  - SelfIncPostfix
- Id
- StringExpr
- CastExpr
- ConstExpr
- PostExpr
- UnaryExpr

- Declarator
  - FunDeclarator
  - ArrDeclarator
- TypedefName
- TypeSpecifier
  - VoidType
  - RecordType
  - IntType
  - NameType
  - CharType

Acknowledgments

Many thanks to TAs

Reference

Compilers: Principles, Techniques and Tools
Xiao Jia's and other TAs' materials