Formal Languages and Compilers

ACSE: Introduction

Alessandro Barenghi & Michele Scandale

January 8, 2020

ACSE: Advanced Compiler System for Education

Simple compiler frontend:

- accepts a C-like source language
- emits a RISC-like intermediate code

The lab test *usually* requires to:

- add tokens
- add grammar rules to recognize new constructs
- write semantic actions to generate code for the new constructs

ACSE: Advanced Compiler System for Education

Toolchain

The toolchain is composed of three programs:

- **acse** compiler frontend (from LANCE to assembly)
- **asm** assembler (from assembly to machine code)
- **mace** machine interpreter

In this course we will modify only the ACSE compiler:

- the source is located in `acse` directory
- the code is simple and well documented
- there are a **lot** of helper functions to perform common operations

`LANCE`: LANguage for Compiler Education

`LANCE` is the source language recognized by `ACSE`:

- very small subset of C99
- standard set of arithmetic/logic/relational operators
- reduced set of control flow statements (**`while`**, **`do-while`**, **`if`**)
- only one scalar type (**`int`**)
- only one aggregate type (array of **`int`**s)
- no functions

Very limited support to I/O operations:

reading **`read(var)`** stores into **`var`** an integer read from standard input

writing **`write(var)`** writes **`var`** to standard output

LANCE: LANguage for Compiler Education

Input format

A LANCE source file is composed by two sections:

- variable declarations
- program body as a **;** separated list of statements

```
int x, y, z = 42;
int arr[10];
int i;

read(x);
read(y);

i=0;
while (i < 10) {
  arr[i] = (y - x) * z;
  i = i + 1;
}
z = arr[9];
write(z);
```

Intermediate Representation

`LANCE` code is first translated into a RISC-like intermediate language having:

- a few essential computing instructions (e.g. `ADD`, `SUB`)
- memory instructions (e.g. `LOAD`, `STORE`)
- branches (e.g. `BEQ`, `BT`)
- special I/O instructions (e.g. `READ`, `WRITE`)

Two addressing modes are supported:

direct data inside the register

indirect data at the memory location pointed by register

Data storage:

- unbounded registers
- unbounded memory locations

Intermediate Representation

Instruction Formats

Type	Operands	Example
Ternary	1 destination and 2 source registers	`ADD R3 R1 R2`
Binary	1 destination and 1 source register and 1 immediate operand	`ADD R3 R1 #4`
Unary	1 destination and 1 address operand	`LOAD R1 L0`
Jump	1 address operand	`BEQ L0`

Intermediate Representation

Operands & Addressing modes

Operand type	Syntax	Notes
Register direct	`Rn`	The `n`-th register
Register indirect	`(Rn)`	Data at the address contained in the `n`-th register
Symbolic address	`Ln`	The address identified by the `n`-th label
Immediate	`#n`	The scalar integer constant `n`

Intermediate Representation

Addressing modes example

This should be known from bachelor courses on Computer Architectures.

`ADD R3 R1 R2`

Before

Register file		Memory	
$R1$	1	$0xd$	0
$R2$	2	$0xe$	0
$R3$	0	$0xf$	0

↓

After

Register file		Memory	
$R1$	1	$0xd$	0
$R2$	2	$0xe$	0
$R3$	3	$0xf$	0

`ADD R3 R1 (R2)`

Before

Register file		Memory	
$R1$	1	$0xd$	2
$R2$	$0xd$	$0xe$	0
$R3$	0	$0xf$	0

↓

After

Register file		Memory	
$R1$	1	$0xd$	2
$R2$	$0xd$	$0xe$	0
$R3$	3	$0xf$	0

Intermediate Representation

Register notes

There are two special registers:

- zero `R0` contains the constant value `0`: it cannot be written
- status word (a.k.a. PSW) implicitly read/written by some instructions

The *zero* register is useful to perform constant value materialization:

```
ADD R1 R0 #10
```

The status register contains four single-bit flags, that are exploited mainly by conditional jump instructions:

- N negative
- Z zero
- O overflow
- C carry

```
SUBI R1 R1 #1
BNE L0
```

ACSE

The core elements of ACSE compiler are:

- scanner: listed in `Acse.lex`
- parser: listed in `Acse.y`
- codegen: code generation functions listed in `axe_gencode.h`

ACSE is a syntax directed translator:

- no explicit AST is built
- the code generation is performed while parsing
- once an instruction is emitted you cannot go back

ACSE

The compiler context is an instance of **t_program_infos**:

```
typedef struct t_program_infos {
  t_list *variables;
  t_list *instructions;
  t_list *data;
  t_axe_label_manager *lmanager;
  t_symbol_table *sy_table;
  int current_register;
} t_program_infos;
```

All the information about variables, symbols, virtual registers and the generated code are saved in this structure.

ACSE: Variables

A LANCE variable is represented by the **IDENTIFIER** token:

- the semantic value of the token is a C-string that represent the name of the variable

```
%union {
  ...
  char *svalue;
  ...
}

%token <svalue> IDENTIFIER
```

- the lexical value is put in the token semantic value by the lexer

```
ID        [a-zA-Z_][a-zA-Z0-9_]*
%%
...
{ID}  { yylval.svalue=strdup(yytext); return IDENTIFIER; }
...
```

ACSE: Variables

Internally information on variables is stored in a list of `t_axe_variables`:

```
typedef struct t_axe_variable {
  int type;
  int isArray;
  int arraySize;
  int init_val;
  char *ID;
  t_axe_label *labelID;
} t_axe_variable;
```

To retrieve such information you can use:

```
t_axe_variable *getVariable(t_program_infos *, char *)
```

Variables have a **label** (symbolic address) representing their memory location. To simplify the code generation phase, scalar variables are identified also by an unique **virtual register**:

- a scalar variable is assumed to be always stored in the same register, the compiler will manage loads and stores operations when necessary (e.g. *register allocation*).

ACSE: Variables

Example

Let's analyze this simple program:

```
int a;
write(a);
```

The intermediate representation generated is (see `output.cfg`):

```
WRITE R1 0
HALT
```

- `R1` is the virtual register that the compiler assigned to `a`
- in the intermediate representation there are no explicit operation to *load/store* the variable *from/to* memory

ACSE: Variables
Example

The declaration rules involved are:

```
var_declarations : var_declarations var_declaration
                 |
                 ;

var_declaration : TYPE declaration_list SEMI
                  { set_new_variables(program, $1, $2); }
                ;

declaration_list  : declaration_list COMMA declaration
                    { $$ = addElement($1, $3, -1); }
                  | declaration
                    { $$ = addElement(NULL, $1, -1); }
                  ;

declaration : IDENTIFIER ASSIGN NUMBER
              {
                $$ = alloc_declaration($1, 0, 0, $3);
                if ($$ == NULL) notifyError(AXE_OUT_OF_MEMORY);
              }
            | IDENTIFIER
              {
                $$ = alloc_declaration($1, 0, 0, 0);
                if ($$ == NULL) notifyError(AXE_OUT_OF_MEMORY);
              }
            | ...
            ;
```

ACSE: Variables

Example

The other involved rules are:

```
write_statement : WRITE LPAR exp RPAR
                  {
                    int location;
                    if ($3.expression_type == IMMEDIATE)
                      location = gen_load_immediate(program, $3.value);
                    else
                      location = $3.value;
                    gen_write_instruction(program, location);
                  }
                ;

exp : NUMBER { $$ = create_expression($1, IMMEDIATE); }
    | IDENTIFIER
      {
        int location = get_symbol_location(program, $1, 0);
        $$ = create_expression(location, REGISTER);
        free($1);
      }
    | ...
    ;
```

`get_symbol_location` returns the register that the compiler assigned to the variable.

ACSE: Variables

Another example

Let's analyze another simple program:

```
int a[10];
write(a[1]);
```

The intermediate representation generated is (see `output.cfg`):

```
MOVA R1 L0
ADDI R1 R1 #1
ADD R2 R0 (R1)
WRITE R2 0
HALT
```

- `L0` is the symbolic address of the array `a`
- the address is copied through `MOVA` into `R1`
- `R1` is incremented by one: this represents the address of the element 1 of the array
- the value is `a[1]` is loaded from memory in `R2` through an `ADD` with indirect addressing mode

ACSE: Variables

Another example

The declaration rule for an array is:

```
declaration : IDENTIFIER LSQUARE NUMBER RSQUARE
               {
                 $$ = alloc_declaration($1, 1, $3, 0);
                 if ($$ == NULL) notifyError(AXE_OUT_OF_MEMORY);
               }
             | ...
             ;
```

The expression rule for an array element is:

```
exp : IDENTIFIER LSQUARE exp RSQUARE
       {
         int reg = loadArrayElement(program, $1, $3);
         $$ = create_expression(reg, REGISTER);
         free($1);
       }
     | ...
     ;
```

`loadArrayElement` is an helper function that implements the loading from an array at a given index expression.

ACSE: Variables

Another example

Here is the code of two array helper functions (see **axe_array.h**):

```
int loadArrayElement(t_program_infos *program, char *ID,
                     t_axe_expression index) {
  int addr = loadArrayAddress(program, ID, index);
  int reg = getNewRegister(program);

  gen_add_instruction(program, reg, REG_0, addr, CG_INDIRECT_SOURCE);
  return reg;
}

int loadArrayAddress(t_program_infos *program, char *ID,
                     t_axe_expression index) {
  t_axe_label *label = getLabelFromVariableID(program, ID);
  int reg = getNewRegister(program);
  gen_mova_instruction(program, reg, label, 0);

  if (index.expression_type == IMMEDIATE && index.value != 0)
    gen_addi_instruction (program, reg, reg, index.value);
  else if (index.expression_type == REGISTER)
    gen_add_instruction(program, reg, reg, index.value, CG_DIRECT_ALL);

  return reg;
}
```

Class task

Let's extend the ACSE compiler to add the support to macro definition C-like:

- we want to implement simple macros (no functions) that define integer constants
- a macro name can be used wherever is legal to have an integer constant
- a macro cannot be modified by an assigment

```
define FOO 42
define BAR 56

int a;
read(a);
write(a * FOO - BAR);
```

Hints:

- think about the behavior of macros in C