Compilation Principle
编 译 原 理

第17讲：中间代码(1)

张献伟

xianweiz.github.io

DCS290, 5/5/2022

Quiz Questions

- Q1: how do CFG, SDD and SDT relate to each other?

  CFG + attributes/symbol + rules/production → SDD → rules embedded into the production body (action) → SDT.

- Q2: is *C.c* an synthesized attribute?

  NO. It is an inherited attribute, depending on parent (A.*a*) and sibling (B.*b*)

Production	Semantic Rule
A → BC	C.*c* = B.*b* + A.*a*

- Q3: suppose *A.a* is synthesized, is it S-SDD or L-SDD?

  Neither.

  Not S-SDD: C.*c* is inherited; Not L-SDD: A.*a* is synthesized.

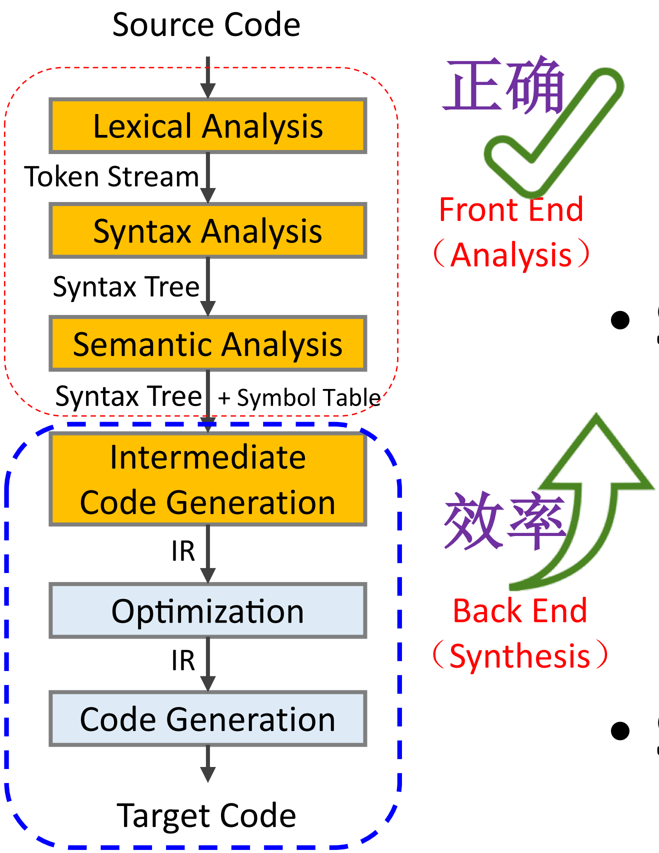
- Q4: for a L-SDD, how to convert it into SDT for LR parse?

  Add markers and empty rules to move all semantic actions to the end of production rule, just likewise S-SDD.

- Q5: briefly explain symbol table.

  A compiler data structure to track all symbols in semantic analysis phase, and holds info like name, type, value, scope, etc.

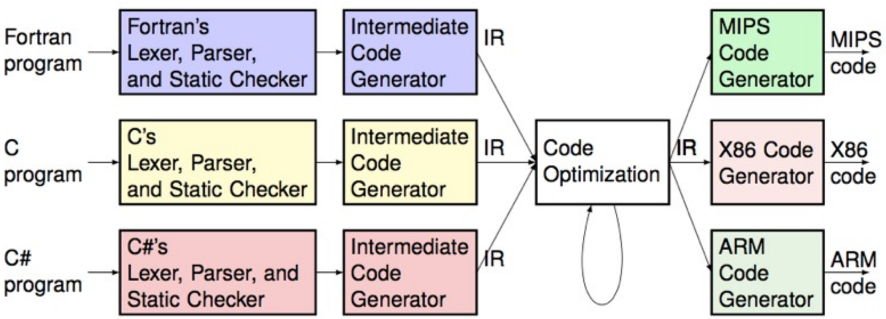
Compilation Phases[编译阶段]



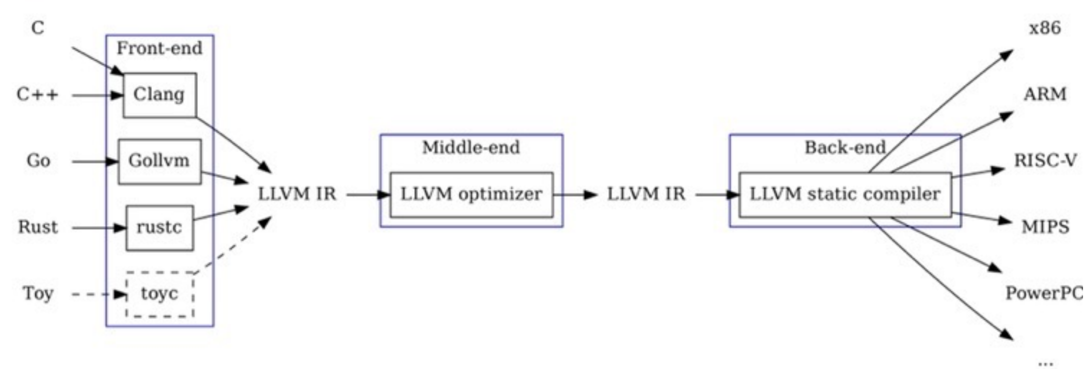
- **Lexical**: source code → tokens
  - RE, NFA, DFA, …
  - Is the program lexically well-formed?
    - E.g., x#y = 1
- **Syntax**: tokens → AST or parse tree
  - CFG, LL(1), LALR(1), …
  - Is the input program syntactically well-formed?
    - E.g., for(i = 1)
- **Semantic**: AST → AST + symbol table
  - SDD, SDT, typing, scoping, …
  - Does the input program has a well-defined meaning?
    - E.g., int x; y = x(1)

Modern Compilers[现代编译器]

- Compilation flow[编译流程]
  - First, translate the source program to some form of **intermediate representation** (IR, 中间表示)
  - Then convert from there into machine code[机器代码]
- IR provides advantages[IR的优势]
  - Increased abstraction, cleaner separation, and retargeting, etc



m languages
n machines
O(*mn*) vs. O(*m*+*n*)



Different IRs for Different Stages

- Modern compilers use different IRs at different stages

- **High-Level** IR: close to high-level language[接近语言]
  - Examples: Abstract Syntax Tree, Parse Tree
  - Language dependent (a high-level IR for each language)
  - Purpose: semantic analysis of program
- **Low-Level** IR: close to assembly[接近汇编]
  - Examples: Three address code[三地址码], Static Single Assignment[静态单赋值]
  - Essentially an instruction set[指令集] for an abstract machine
  - Language and machine independent (one common IR)
  - Purpose: compiler optimizations to make code efficient
    - All optimizations written in this IR is automatically applicable to all languages and machines

SUN YAT-SEN UNIVERSITY 中山大學

Different IRs for Different Stages (cont.)

- **Machine-Level** IR[机器层级]
  - Examples: x86 IR, ARM IR, MIPS IR, RISC-V IR, …
  - Actual instructions for a concrete machine ISA
  - Machine dependent (a machine-level IR for each ISA)
  - Purpose: code generation / CPU register allocation
    - (Optional) Machine-level optimizations (e.g. strength reduction: x / 2 → x » 1)
- Possible to have one IR (AST) — some compilers do
  - Generate machine code from AST after semantic analysis[AST到机器代码，无真正意义上的IR]
  - Makes sense if compilation time is the primary concern (e.g. JIT)
    - Skip the IR generation step

- So **why have multiple IRs?**

Why Multiple IRs?

- Why multiple IRs?
  - Better to have an appropriate IR for the task at hand[针对性]
    - Semantic analysis much easier with AST
    - Compiler optimizations much easier with low-level IR
    - Register allocation only possible with machine-level IR
  - Easier to add a new front-end (language) or back-end (ISA)[易于扩展]
    - Front-end: a new AST → low-level IR converter
    - Back-end: a new low-level IR → machine IR converter
    - Low-level IR acts as a bridge between multiple front-ends and back-ends, such that they can be reused
- If one IR (AST), and adding a new front-end ...
  - Reimplement all compiler optimizations for new AST
  - A new AST → machine code converter for each ISA
  - Same goes for adding a new back-end

Three-Address Code[三地址码]

- High-level assembly where each operation has **at most three** operands. Generic form is X = Y op Z[最多3个操作数]
  - where X, Y, Z can be variables, constants, or compiler-generated temporaries holding intermediate values
- Characteristics[特性]
  - Assembly code for an 'abstract machine'
  - Long expressions are converted to multiple instructions
  - Control flow statements are converted to jumps[控制流->跳转]
  - Machine independent
    - Operations are generic (not tailored to any specific machine)
    - Function calls represented as generic call nodes
    - Uses symbolic names rather than register names (actual locations of symbols are yet to be determined)
- Design goal: for easier machine-independent optimization

Three-Address Code Example

- For example, x * y + x * y is translated to

  t1 = x * y ; t1, t2, t3 are temporary variables

  t2 = x * y

  t3 = t1 + t2

  - Can be generated through a depth-first traversal of AST
  - Internal nodes in AST are translated to temporary variables
- Notice: repetition of x * y[重复]
  - Can be later eliminated through a compiler optimization called common subexpression elimination (CSE):[通用子表达式消除]

    t1 = x * y

    t3 = t1 + t1
  - Using 3-address code rather than AST makes it:
    - Easier to spot opportunities (just find matching RHSs)
    - Easier to manipulate IR (AST is much more cumbersome)

Three-Address Statements

- Assignment statement[二元赋值]

  x = y op z

  where op is an arithmetic or logical operation (binary operation)

- Assignment statement[一元赋值]

  x = op y

  where op is an unary operation such as -, not, shift

- Copy statement[拷贝]

  x = y

- Unconditional jump statement[无条件跳转]

  goto L

  where L is label

Three-Address Statements (cont.)

- Conditional jump statement[条件跳转]

  if (x relop y) goto L

  where relop is a relational operator such as $=, \neq, >, <$

- Procedural call statement[过程调用]

  param x_1, ..., param x_n, call F_y, n

  As an example, foo(x_1, x_2, x_3) is translated to

  param x_1

  param x_2

  param x_3

  call foo, 3

- Procedural call return statement[过程调用返回]

  return y

  where y is the return value (if applicable)

中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ

Three-Address Statements (cont.)

- Indexed assignment statement[索引]

  x = y[i]

  or

  y[i] = x

  where x is a scalar variable and y is an array variable

- Address and pointer operation statement[地址和指针]

  x = & y ; a pointer x is set to address of y

  y = * x ; y is set to the value of location

  ; pointed to by pointer x

  *y = x ; location pointed to by y is assigned x

Example

```
i = 1
do {
   a[i] = x * 5;
   i ++;
} while (i <= 10);
```

Source program

$$
\begin{aligned}
& \quad i = 1 \\
& L: t_1 = x * 5 \\
& \quad \boxed{\begin{aligned} & t_2 = \&a \\ & t_3 = \mathrm{sizeof(int)} \\ & t_4 = t_3 * i \\ & t_5 = t_2 + t_4 \end{aligned}} \quad a[i] \\
& \quad *t_5 = t_1 \\
& \quad i = i + 1 \\
& \quad \text{if } i <= 10 \text{ goto } L
\end{aligned}
$$

Three-address code

Implementation of TAC

- 3 possible ways (and more)
  - quadruples[四元式]
  - triples[三元式]
  - indirect triples[间接三元式]
- Trade-offs between, space, speed, ease of manipulation
- Using **quadruples**[四元式]

  op arg1, arg2, result

  - There are four(4) fields at maximum
  - arg1 and arg2 are optional, depending on the *op*
  - Examples:
    - x = a + b => + a, b, x
    - x = - y => - y, , x
    - goto L => goto , , L

Using Triples[三元式]

- **Triple**: quadruple without the result field
  - Result field is implicitly index of instruction
  - Result referred to by index of instructions computing it
  - Example: a = b * (-c) + b * (-c)

	Quadruples				Triples		
	op	arg1	arg2	result	op	arg1	arg2
(0)	-	c		t1	-	c	
(1)	*	b	t1	t2	*	b	(0)
(2)	-	c		t3	-	c	
(3)	*	b	t3	t4	*	b	(2)
(4)	+	t2	t4	t5	+	(1)	(3)
(5)	=	t5		a	=	a	(4)

More About Triples

- What if LHS of assignment is not a var but an expression?
  - Array location (e.g. x[i] = y)
  - Pointer location (e.g. *(x+i) = y)
  - Struct field location (e.g. x.i = y)
- Compute memory address of LHS location beforehand
- Example: triples for array assignment statement

    x[i] = y

  - is translated to

```
(0): [] x i          // Compute address of x[i] location
(1): = (0) y         // Assign y to that location
```

  - Complex LHS may require more triples to compute address

Using Indirect Triples[间接三元式]

- Problem with triples
  - Compiler optimizations often involve moving instructions
  - Hard to move instructions because numbering will change, even for instructions not involved in optimization
  - See below CSE performed on the second (-c) * b:

	Quadruples				Triples		
	op	arg1	arg2	result	op	arg1	arg2
(0)	-	c		t1	-	c	
(1)	*	b	t1	t2	*	b	(0)
~~(2)~~	~~-~~	~~c~~		~~t3~~	~~-~~	~~c~~	
~~(3)~~	~~*~~	~~b~~	~~t3~~	~~t4~~	~~*~~	~~b~~	~~(2)~~
~~(4)~~(2)	+	t2	~~t4~~ t2	t5	+	(1)	~~(3)~~(1)
~~(5)~~(3)	=	t5		a	=	a	(4)

Using Indirect Triples[间接三元式]

- Problem with triples
  - Compiler optimizations often involve moving instructions
  - Hard to move instructions because numbering will change, even for instructions not involved in optimization
  - See below CSE performed on the second (-c) * b:

	Quadruples				Triples		
	op	arg1	arg2	result	op	arg1	arg2
(0)	-	c		t1	-	c	
(1)	*	b	t1	t2	*	b	(0)
(2)	+	t2	t2	t5	+	(1)	(1)
(3)	=	t5		a	=	a	(4)

Instruction (3) refers to (4) which is no longer there.

Using Indirect Triples (cont.)

- Triples are stored in a triple 'database'
- IR is a listing of pointers to triples in database
  - Can reorder listing without changing numbering in database
- Pointer indirection overhead but allows easy code motion

	Listing
	(ptr to triple database)
(0)	(0)
(1)	(1)
(2)	(2)
(3)	(3)
(4)	(4)
(5)	(5)

	Database		
	op	arg1	arg2
(0)	-	c	
(1)	*	b	(0)
(2)	-	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

After CSE Optimization

- After CSE, empty entries in database can be reused
  - Code in triple database becomes non-contiguous over time
  - That's fine since the listing is the code, not the database

	Listing
	(ptr to triple database)
(0)	(0)
(1)	(1)
(2)	(4)
(3)	(5)

	Database		
	op	arg1	arg2
(0)	-	c	
(1)	*	b	(0)
(2)	empty		
(3)	empty		
(4)	+	(1)	(1)
(5)	=	a	(4)

Single Static Assignment[静态单赋值]

- Every variable is assigned to exactly once statically[仅一次]
  - Give variable a different version name on every assignment
    - e.g. $x \rightarrow x_1, x_2, \ldots, x_5$ for each static assignment of x
  - Now value of each variable guaranteed not to change
  - On a control flow merge, ϕ-function combines two versions
    - e.g. $x_5 = \phi(x_3, x_4)$: means x_5 is either x_3 or x_4