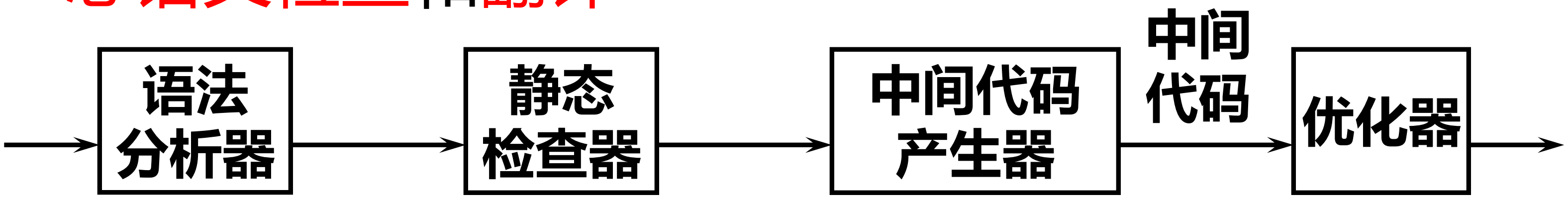
Chapter 7 语义分析和中间代码产生

Outlines

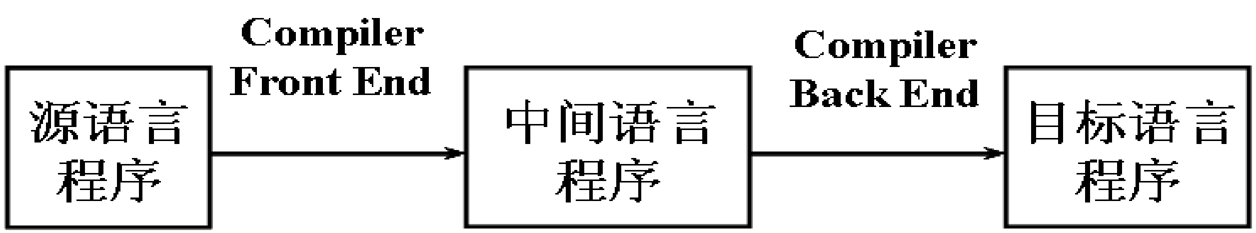
- 静态检查和中间语言简介
- 静态语义检查
- 中间语言形式
  - 后缀式
  - 图表示法
    - DAG
    - 抽象语法树
  - 三地址代码
    - 三元式
    - 四元式
    - 间接三元式

Static checking

- 词法分析和语法分析之后，编译程序的工作是进行静态语义检查和翻译



- 借助中间语言进行翻译
  - 便于进行与机器无关的代码优化工作
  - 使编译程序改变目标机更容易
  - 使编译程序的结构在逻辑上更为简单明确



Static checking

1）类型检查。

验证程序中执行的每个操作是否遵守语言的类型系统的过程，编译程序必须报告不符合类型系统的信息。

2）控制流检查。

控制流语句必须使控制转移到合法的地方。例如，在C语言中break语句使控制跳离包括该语句的最小while、for或switch语句。如果不存在包括它的这样的语句，则就报错

3）一致性检查。

在很多场合要求对象只能被定义一次。例如Pascal语言规定同一标识符在一个分程序中只能被说明一次，同一case语句的标号不能相同，枚举类型的元素不能重复出现等等

Static checking

4）相关名字检查。

有时，同一名字必须出现两次或多次。例如，Ada　语言程序中，循环或程序块可以有一个名字，出现在这些结构的开头和结尾，编译程序必须检查这两个地方用的名字是相同的。

5)名字的作用域分析

Intermediate language

- 常用的中间语言
  - 后缀式表示法
  - 图表示法
    - DAG
    - 抽象语法树
  - 三地址代码
    - 三元式
    - 四元式
    - 间接三元式

Postfix notation

- 后缀式表示法又称逆波兰表示法
  - Lukasiewicz发明的一种表示表达式的方法
  - 把运算量（操作数）写在前面，把算符写在后面
    - 如a+b写成ab+
- 表达式E的后缀式
  - 若E是一个变量或常量，则E的后缀式是E自身
  - 若E是E_1 op E_2形式的表达式，则E的后缀式为
  - E_1' E_2'op
    - op是任何二元操作符
    - E_1' 和E_2' 分别为E_1 和E_2的后缀式
  - 若E是(E_1)形式的表达式，则E_1的后缀式就是E的后缀式

Postfix notation

- abc+*等价a*(b+c)
- (a + b)*(c + d) → ab + cd +*

- 表达式 x+y≤z ∨ a > 0 ∧ (8+z) > 3 的逆波兰表示为

- 表达式¬A∨¬（C∨¬D）的逆波兰表示为

Postfix notation

- abc+*等价a*(b+c)
- (a + b)*(c + d) → ab + cd +*

- 表达式 x+y≤z ∨ a > 0 ∧ (8+z) > 3 的逆波兰表示为

  xy+z≤a0 > 8z+3 > ∧∨

- 表达式¬A∨¬（C∨¬D）的逆波兰表示为

  A¬CD¬∨¬∨

Example

- 表达式 a ∧ b ∨ c ∧ (b ∨ x=0 ∧ c) 的逆波兰表示为______________________________。

- 表达式 （A∨B）∧（C∨¬D∧E）的逆波兰表示为
。

Example

- 表达式 $a \wedge b \vee c \wedge (b \vee x=0 \wedge c)$ 的逆波兰表示为____________ab∧cbx0=c∧∨∧∨___。

- 表达式 （A∨B）∧（C∨¬D∧E）的逆波兰表示为________ABvCD¬E∧V∧________。

Postfix notation

- 逆波兰表示法不用括号
  - 只要知道每个算符的目数，对于后缀式，不论从哪一端进行扫描，都能对它进行唯一分解
- 后缀式的计算
  - 用一个栈实现
  - 一般的计算过程
    - 自左至右扫描后缀式
    - 每碰到运算量就把它推进栈
    - 每碰到k目运算符就把它作用于栈顶的k个项
    - 用运算结果代替这k个项

Postfix notation

- 把表达式翻译成后缀式的语义规则描述

产生式	语义规则
$E \rightarrow E_1 op E_2$	$E.code := E_1.code \mid\mid E_2.code \mid\mid op$
$E \rightarrow (E_1)$	$E.code := E_1.code$
$E \rightarrow id$	$E.code := id$

- E.code表示E后缀形式
- op表示任意二元操作符
- “||”表示后缀形式的连接

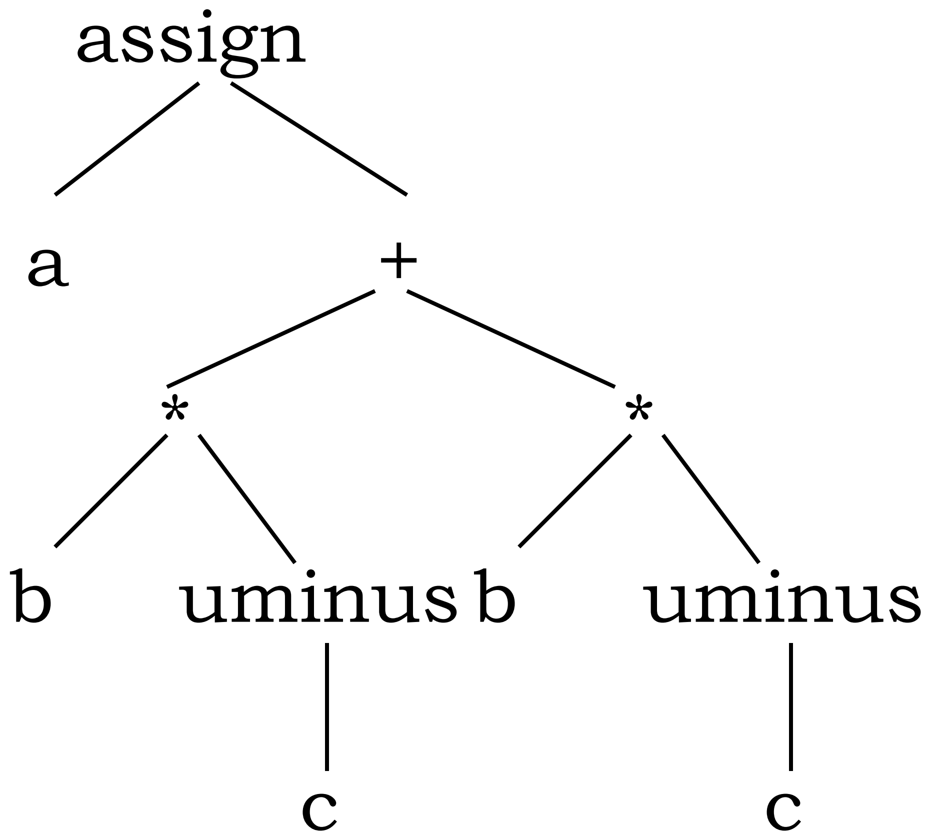
Graph

- 图表示法
  - DAG
  - 抽象语法树
    - 描述源程序的自然层次结构

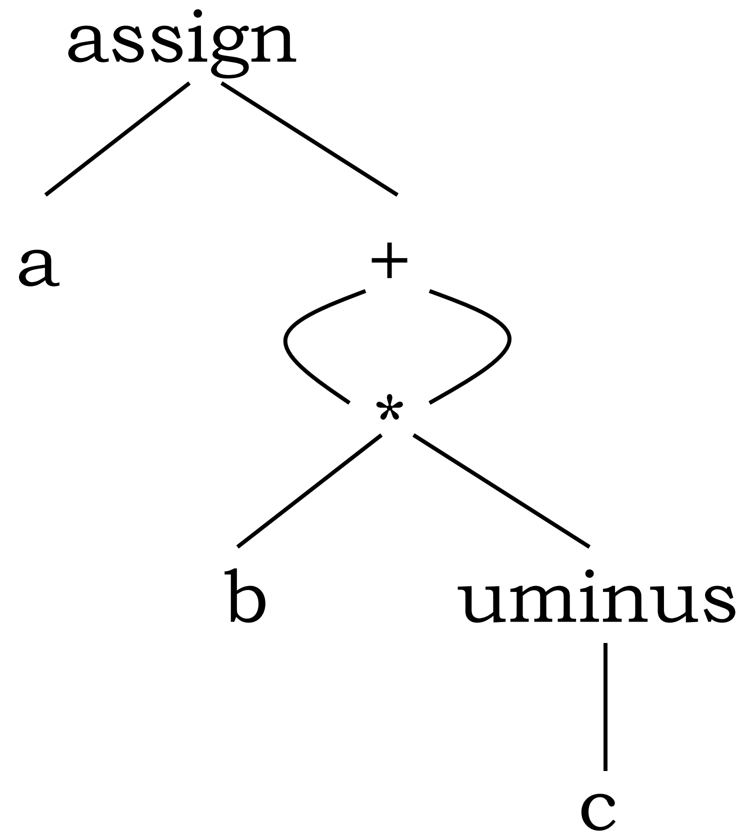
- 无循环有向图(Directed Acyclic Graph，DAG)
  - 对表达式中的每个子表达式，DAG中都有一个结点
  - 一个内部结点代表一个操作符，它的孩子代表操作数
  - 一个DAG中代表公共子表达式的结点具有多个父结点

DAG

a:=b*(-c)+b*(-c)



抽象语法树



DAG 更加紧凑

Abstract Syntax Tree

- mknode（op，left，right）
  - 建立一个运算符号结点
  - 标号是op
  - 两个域left和right分别指向左子树和右子树
- mkleaf（id，entry）
  - 建立一个标识符结点
  - 标号为id
  - 一个域eutry指向标识符在符号表中的入口
- mkleaf（num，ral）
  - 建立一个数结点
  - 标号为num
  - 一个域ral用于存放数的值

Abstract Syntax Tree

产生式	语义规则
$E \rightarrow E_1 + T$	$E.nptr := mknode(\text{'+'}, E_1.nptr, T.nptr)$
$E \rightarrow E_1 - T$	$E.nptr := mknode(\text{'-'}, E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr := T.nptr$
$T \rightarrow (E)$	$T.nptr := E.nptr$
$T \rightarrow id$	$T.nptr := mkleaf(id, id.entry)$
$T \rightarrow num$	$T.nptr := mkleaf(num, num.val)$

Three-address codes

- 一般形式
  - $x := y \; op \; z$
- 三地址代码包含三个地址
  - 两个用来表示操作数
  - 一个用来存放结果
- 表达式$x + y * z$翻译成的三地址语句序列是
  - $t_1 := y * z$
  - $t_2 := x + t_1$

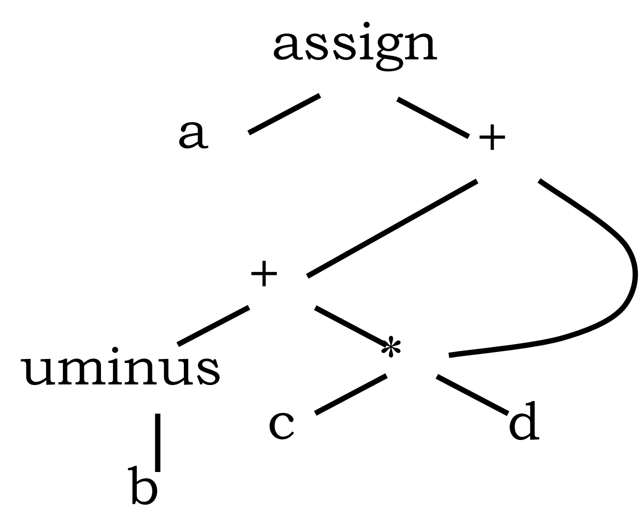
Three-address codes

- 三地址代码是语法树或DAG的一种线性表示

$$a := (-b + c*d) + c*d$$

语法树的代码	DAG的代码
$t_1 := -b$	$t_1 := -b$
$t_2 := c * d$	$t_2 := c * d$
$t_3 := t_1 + t_2$	$t_3 := t_1 + t_2$
$t_4 := c * d$	$t_4 := t_3 + t_2$
$t_5 := t_3 + t_4$	$a := t_4$
$a := t_5$	

更加简洁



Three-address codes

- 赋值语句
  - x := y op z， x := op y， x := y
- 无条件转移
  - goto L
- 条件转移
  - if x relop y goto L
- 过程调用
  - param x和call p , n
- 过程返回
  - return y
- 索引赋值
  - x := y[i]和x[i] := y
- 地址和指针赋值
  - x := &y， x := *y和*x := y

Quadruples

- 一个带有四个域的记录结构
  - 这四个域分别称为op, arg1, arg2及result

- 例： a:=b*-c+b*-c 的四元式

Quadruples

- 一个带有四个域的记录结构
  - 这四个域分别称为op, arg1, arg2及result

- 例： a:=b*-c+b*-c 的四元式

	op	arg1	arg2	result
(0)	uminus	c		T_1
(1)	*	b	T_1	T_2
(2)	uminus	c		T_3
(3)	*	b	T_3	T_4
(4)	+	T_2	T_4	T_5
(5)	:=	T_5		a

Triples

- 通过计算临时变量值的语句的位置来引用这个临时变量

- 例： a:=b*-c+b*-c 的三元式
  - 三个域：op、arg1和arg2

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

Triples

- 三元式中的多目运算符用若干相继的三元式表示
  - 例如，x[i]:=y

	op	arg1	arg2
(0)	[ ] =	x	i
(1)	assign	(0)	y

  - 又如， x:=y[i]

	op	arg1	arg2
(0)	= [ ]	y	i
(1)	assign	x	(0)

Indirect triples

- 三元式表+间接码表
  - 用于中间代码表示
  - 间接码表
    - 一张指示器表
    - 按运算的先后次序列出有关三元式在三元式表中的位置
  - 优点
    - 便于优化
    - 节省空间

Indirect triples

- 例如，语句
  - X:=(A+B)*C;
  - Y:=D↑(A+B)

的间接三元式表示

间接代码

(1)
(2)
(3)
(1)
(4)
(5)

三元式表

	OP	ARG1	ARG2
(1)	+	A	B
(2)	*	(1)	C
(3)	=	X	(2)
(4)	↑	D	(1)
(5)	=	Y	(4)

Example

例：

$$A + B * (C - D) + E / (C - D)^N$$

分别用逆波兰、三元式、四元式表示。

Example

A + B * (C - D) + E / (C - D) ^N

逆波兰： A B C D - * + E C D – N ^ / +

四元式：

(1) (- C D T1)

(2) (* B T1 T2)

(3) (+ A T2 T3)

(4) (- C D T4)

(5) (^ T4 N T5)

(6) (/ E T5 T6)

(7) (+ T3 T6 T7)

Example

A + B * (C - D) + E / (C - D) ^N

三元式:

(1)	(-	C	D	)
(2)	(*	B	(1)	)
(3)	(+	A	(2)	)
(4)	(-	C	D	)
(5)	(^	(4)	N	)
(6)	(/	E	(5)	)
(7)	(+	(3)	(6)	)

Example

表达式 -a+b*c+d+(e*f) / d*e，如果优先级由高到低依次为-、+、*、/，且均为左结合，则其后缀式为________________________________。

如果优先级由高到低依次为-、+、*、$（乘幂），且均为右结合，则表达式2+3-2+2*2*1$2$3-3-2+1的后缀式为________________________________。

如果某表达式的后缀式为ab+cd+*，则其中缀形式的表达式为________________________________。

Example

表达式 -a+b*c+d+(e*f)/d*e，如果优先级由高到低依次为-、+、*、/，且均为左结合，则其后缀式为＿＿a-b+cd+ef*+*de*/＿＿。

如果优先级由高到低依次为-、+、*、$（乘幂），且均为右结合，则表达式2+3-2+2*2*1$2$3-3-2+1的后缀式为＿＿232-2++21**2332--1+$$＿＿。

如果某表达式的后缀式为ab+cd+*，则其中缀形式的表达式为＿＿(a+b) * (c+d)＿＿。

Example

- 给出下面表达式的后缀式（逆波兰表示）
  - a*(-b+c)
  - if(x+y)*z=0 then s:=(a+b)*c else s:=a*b*c
    - 用￥表示if-then-else 运算

Example

- 给出下面表达式的后缀式（逆波兰表示）
  - a*(-b+c)
  - if(x+y)*z=0 then s:=(a+b)*c else s:=a*b*c
    - 用￥表示if-then-else 运算

解：

(1) ab-c+*

(2) xy+z*0=sab+c*:=sab*c*:=￥

第3题

- 请将表达式-(a+b)*(c+d)-(a+b)分别表示成三元式、间接三元式和四元式序列

第3题

- 请将表达式-(a+b)*(c+d)-(a+b)分别表示成三元式、间接三元式和四元式序列

解：

三元式

(1)	(+	a,	b)
(2)	(+	c,	d)
(3)	(*	(1),	(2))
(4)	(-	(3),	/)
(5)	(+	a,	b)
(6)	(-	(4),	(5))

间接三元式

间接三元式序列

(1)	(+	a,	b)
(2)	(+	c,	d)
(3)	(*	(1),	(2))
(4)	(-	(3),	/)
(5)	(-	(4),	(1))

间接码表

(1)
(2)
(3)
(4)
(1)
(5)

四元式

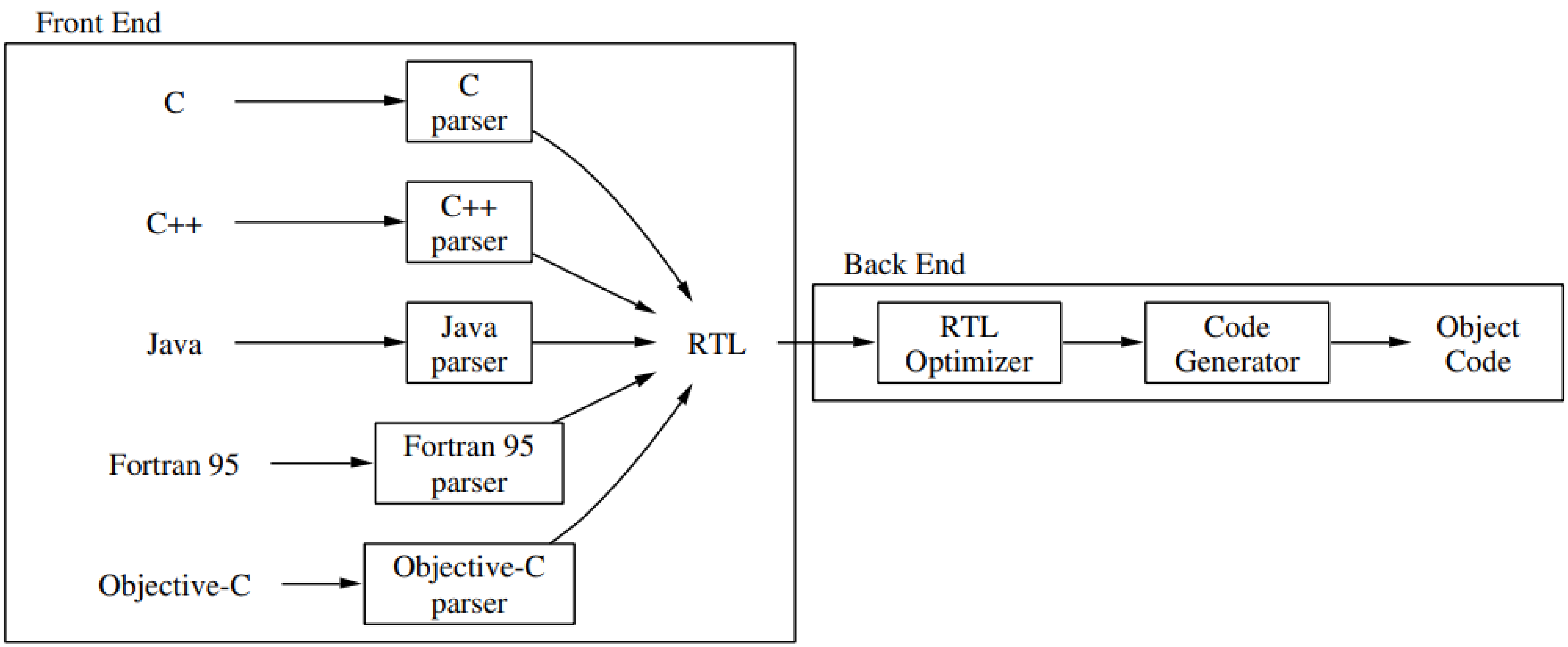
(1)	(+,	a,	b,	t1)
(2)	(+,	c,	d,	t2)
(3)	(*,	t1,	t2,	t3)
(4)	(-,	t3,	/,	t4)
(5)	(+,	a,	b,	t5)
(6)	(-,	t4,	t5,	t6)

history

- Richard Stallman
  - 1971 MIT AI Lab
  - 1985 GNU Manifesto
  - 1985 FSF
  - 1987 GCC
  - 1991 Linux
- Chris Lattner
  - 2003 UIUC llvm
  - 2005 Apple Clang

GCC IR

- 早期GCC架构
  - 函数>RTL
  - GCC 3.0 函数>语法树>RTL

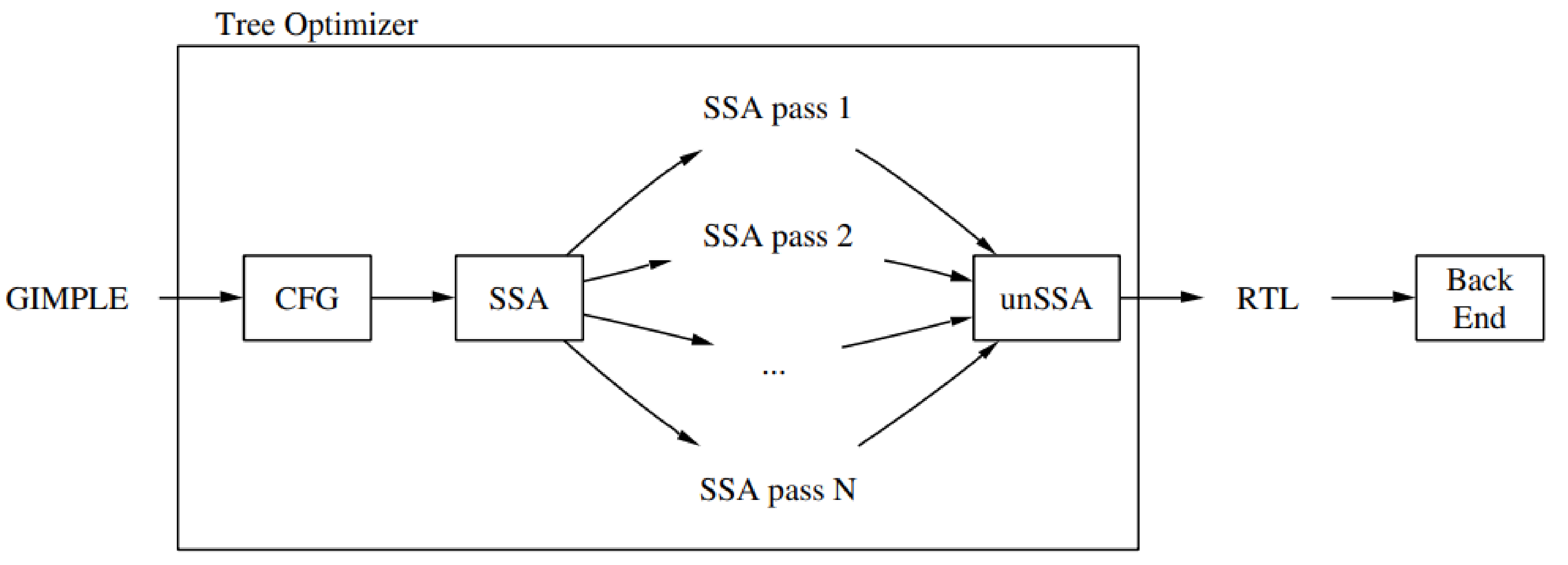


GCC IR

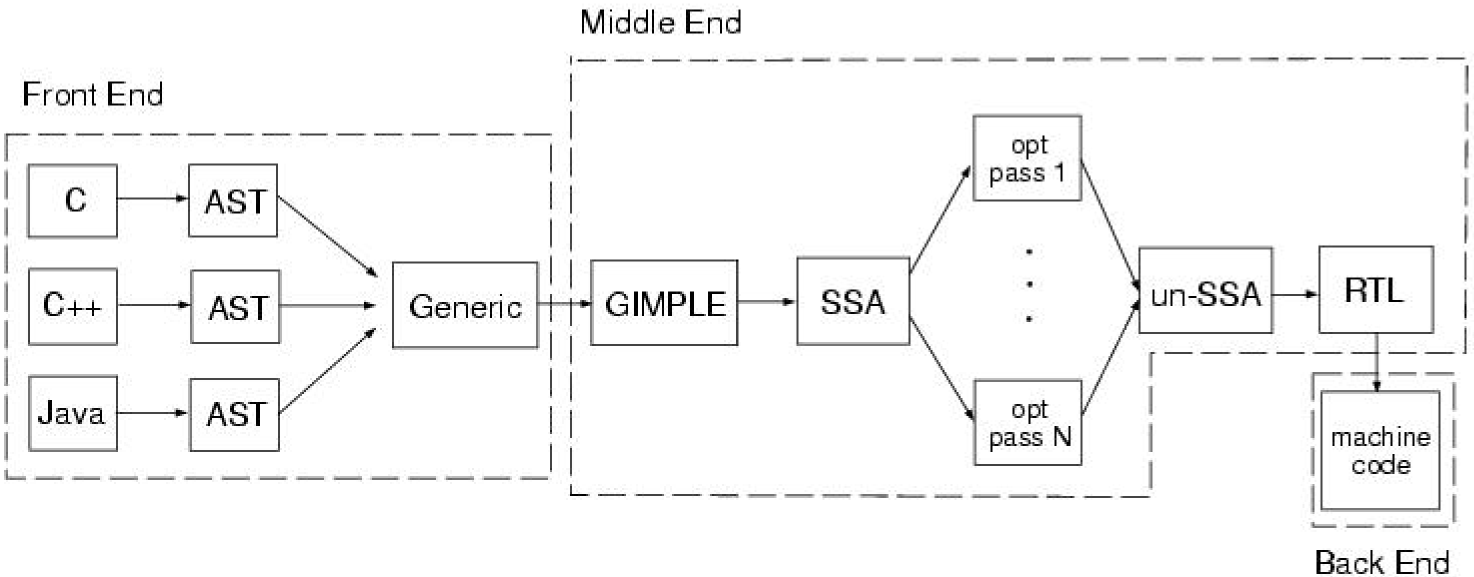
- 源语言>语法树>RTL>目标代码
- 语法树
  - 语言相关
  - 有副作用
  - 结构复杂，一个语句可能有多个基本块
- RTL
  - 可以进行一些机器相关优化（寄存器分配、窥孔优化）
  - 无数据结构（无数组、结构体等）
  - 过早引入栈
- 添加一种IR

GCC IR

- Tree SSA



GCC IR

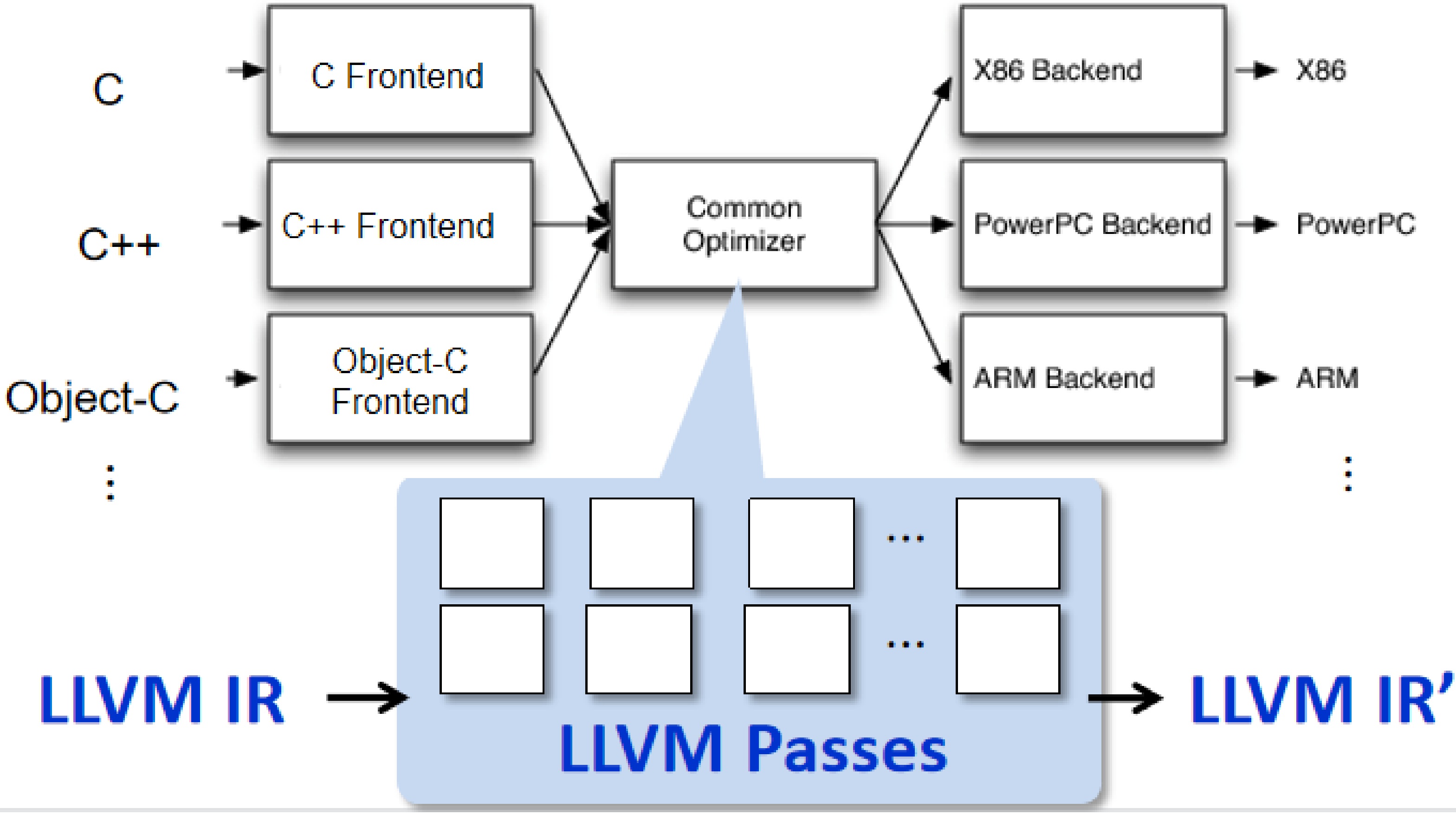


Gimple：主要的中间语言，基于McCat中的Simple

GCC IR

- Gimple
  - 三地址代码
  - 有各种类型
  - 有条件语句
  - 有无限循环语句
  - 有无条件跳转语句
  - 有try catch

LLVM IR


C
C++
Object-C
C Frontend
C++ Frontend
Object-C Frontend
Common Optimizer
X86 Backend
PowerPC Backend
ARM Backend
X86
PowerPC
ARM
LLVM IR
LLVM Passes
LLVM IR'

LLVM IR

- LLVM IR
  - 语言无关
  - 寄存器机器
    - 无限寄存器
    - 三地址代码
    - 31个指令
  - SSA
  - 基本块组成
  - 带类型
  - 控制流

LLVM IR At a Glance

C program language	*LLVM IR*
• Scope: *file, function*	*module, function*
• Type: *bool, char, int, struct{int, char}*	*i1, i8, i32, {i32, i8}*
• A statement with multiple expressions	A sequence of instructions each of which is in a form of "x = y *op* z".
• Data-flow: a sequence of reads/writes on variables	1. load the values of memory addresses (variables) to registers; 2. compute the values in registers; 3. store the values of registers to memory addresses * each register must be assigned exactly once (SSA)
• Control-flow in a function: if, for, while, do while, switch-case,…	A set of basic blocks each of which ends with a conditional jump (or return)

Example

simple.c

```
1  #include <stdio.h>
2  int x, y ;
3
4  int main() {
5    int t ;
6    scanf("%d %d",&x,&y);
7    t = x - y ;
8    if (t > 0)
9     printf("x > y") ;
10   return 0 ;
11 }
```

```
$ clang -S -emit-llvm simple.c
```

simple.ll (simplified)

```
…
2   6 @x = common global i32 0, align 4
    7 @y = common global i32 0, align 4

4  11 define i32 @main() #0 {
   12 entry:
…
5  14 %t = alloca i32, align 4
…
6  16 %call = call i32 (i8*, ...)*
        @__isoc99_scanf(…i32* @x,i32* @y)

7  17 %0 = load i32* @x, align 4
   18 %1 = load i32* @y, align 4
   19 %sub = sub nsw i32 %0 %1
   20 store i32 %sub, i32* %t, align 4

8  21 %2 = load i32* %t, align 4
   22 %cmp = icmp sgt i32 %2, 0
   23 br i1 %cmp, label %if.then,
                  label %if.end

9  24 if.then:
   25   %call1 = call i32 … @printf(…
   26   br label %if.end

10 27 if.end:
   28   ret i32 0
```

Contents

- LLVM IR Instruction
  - architecture, static single assignment
- Data representation
  - types, constants, registers, variables
  - load/store instructions, cast instructions
  - computational instructions
- Control representation
  - control flow (basic block)
  - control instructions
- How to instrument LLVM IR

* *LLVM Language Reference Manual* http://llvm.org/docs/LangRef.html
* *Mapping High-Level Constructs to LLVM IR*
http://llvm.lyngvig.org/Articles/Mapping-High-Level-Constructs-to-LLVM-IR

LLVM IR Architecture

- RISC-like instruction set
  - Only 31 op-codes (types of instructions) exist
  - Most instructions (e.g. computational instructions) are in three-address form: one or two operands, and one result
- Load/store architecture
  - Memory can be accessed via load/store instruction
  - Computational instructions operate on registers
- Infinite and typed *virtual registers*
  - It is possible to declare a new register any point
    (the backend maps virtual registers to physical ones).
  - A register is declared with a primitive type (boolean, int, float, pointer)

Static Single Assignment (1/2)

- In SSA, each variable is assigned exactly once, and every variable is defined before its uses.
- Conversion
  - For each definition, create a new version of the target variable (left-hand side) and replace the target variable with the new variable.
  - For each use, replace the original referred variable with the versioned variable reaching the use point.

```
x = y + x ;
y = x + y ;
if (y > 0)
  x = y ;
else
  x = y + 1 ;
```

```
x1 = y0 + x0 ;
y1 = x1 + y0 ;
if (y1 > 0)
  x2 = y1 ;
else
  x3 = y1 + 1 ;
```

Static Single Assignment (2/2)

- Use ϕ function if two versions of a variable are reaching one use point at a joining basic block
  - $\phi(x_1, x_2)$ returns a either x_1 or x_2 depending on which block was executed

```
x = y + x ;
y = x + y ;
if (y > 0)
  x = y ;
else
  x = y + 1 ;
y = x - y ;
```

```
x1 = y0 + x0 ;
y1 = x1 + y0 ;
if (y1 > 0)
  x2 = y1 ;
else
  x3 = y1 + 1 ;
x4 = ϕ(x2, x3);
y2 = x4 - y1 ;
```

Data Representations

- Primitive types
- Constants
- Registers (virtual registers)
- Variables
  - local variables, heap variables, global variables
- Load and store instructions
- Aggregated types

Primitive Types

- Language independent primitive types with predefined sizes
  - void: **`void`**
  - bool: **`i1`**
  - integers: **`i[N]`** where N is 1 to 2^{23}-1
    e.g. **`i8, i16, i32, i1942652`**
  - floating-point types:
    `half` (16-bit floating point value)
    `float` (32-bit floating point value)
    `double` (64-bit floating point value)

- Pointer type is a form of **`<type>*`** (e.g. `i32*`, `(i32*)*`)

Constants

- Boolean (`i1`): **`true`** and **`false`**
- Integer: standard integers including negative numbers
- Floating point: decimal notation, exponential notation, or hexadecimal notation (IEEE754 Std.)
- Pointer: **`null`** is treated as a special value

Registers

- Identifier syntax
  - Named registers: `[%][a-zA-Z$._][a-zA-Z$._0-9]*`
  - Unnamed registers: `[%][0-9][0-9]*`
- A register has a function-level scope.
  - Two registers in different functions may have the same identifier
- A register is assigned for a particular type and a value at its first (and the only) definition

Variables

- In LLVM, all addressable objects (“lvalues”) are explicitly allocated.
- Global variables
  - Each variable has a global scope symbol that points to the memory address of the object
  - Variable identifier: `[@][a-zA-Z$._][a-zA-Z$._0-9]*`
- Local variables
  - The `alloca` instruction allocates memory in the stack frame.
  - Deallocated automatically if the function returns.
- Heap variables
  - The `malloc` function call allocates memory on the heap.
  - The `free` function call frees the memory allocated by `malloc`.

Load and Store Instructions

- Load

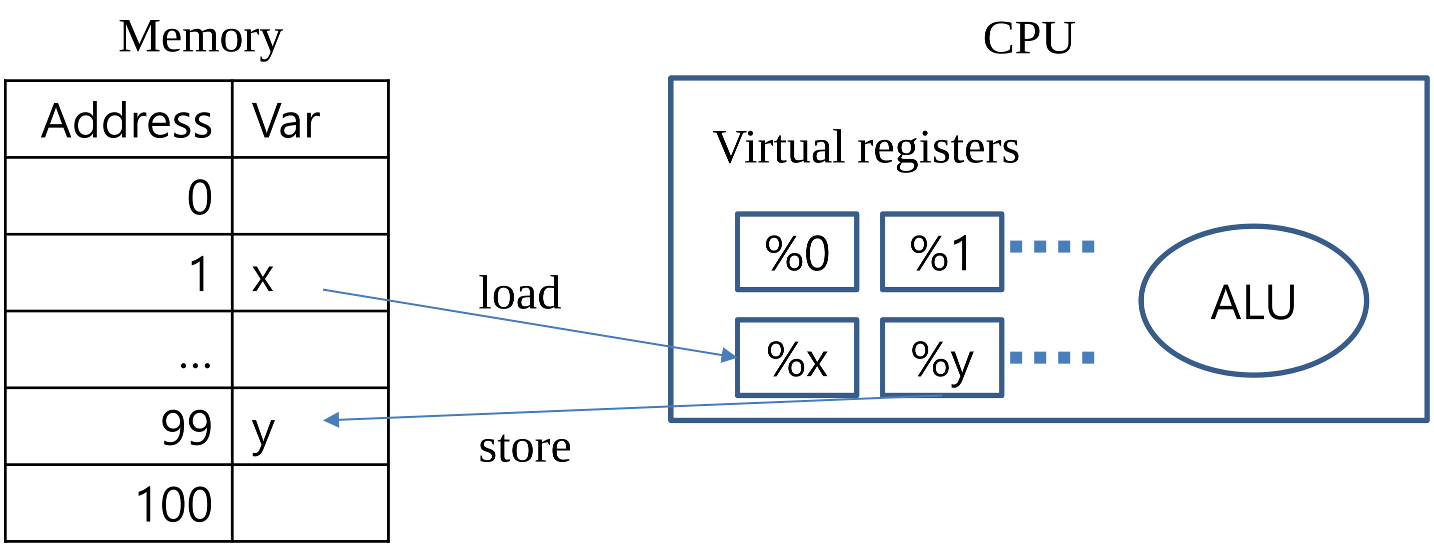
```
<result>=load <type>* <ptr>
```

  - result: the target register
  - type: the type of the data (a pointer type)
  - ptr: the register that has the address of the data

- Store

```
store <type> <value>,<type>* <ptr>
```

  - type: the type of the value
  - value: either a constant or a register that holds the value
  - ptr: the register that has the address where the data should be stored



Variable Example

```c
#include <stdlib.h>

int g = 0 ;

int main() {
 int t = 0;
 int * p;
 p=malloc(sizeof(int));
 free(p);
}
```

```llvm
@g = global i32 0, align 4
…
define i32 @main() #0 {
…
%t = alloca i32, align 4
store i32 0, i32* %t, align 4
%p = alloca i32*, align 8
%call = call noalias i8*
 @malloc(i64 4) #2
%0 = bitcast i8* %call to i32*
store i32* %0, i32** %p,
align 8
%1 = load i32** %p, align 8
…
```

Aggregate Types and Function Type

- Array: **`[<# of elements> x <type>]`**
  - Single dimensional array ex: `[40 x i32]`,`[4 x i8]`
  - Multi dimensional array ex: `[3 x [4 x i8]]`,`[12 x [10 x float]]`

- Structure: **`type {<a list of types>}`**
  - E.g. `type{ i32, i32, i32 }`,`type{ i8, i32 }`

- Function: **`<return type> (a list of parameter types)`**
  - E.g. `i32 (i32)`, `float (i16, i32*)*`

Getelementptr Instruction

- A memory in an aggregate type variable can be accessed by **`load`**/**`store`** instruction and **`getelementptr`** instruction that obtains the pointer to the element.

- Syntax:

```
<res> = getelementptr <pty>* <ptrval>{,<t> <idx>}*
```

  - res: the target register
  - pty: the register that defines the aggregate type
  - ptrval: the register that points to the data variable
  - t: the type of index
  - idx: the index value

Aggregate Type Example 1

```c
struct pair {
    int first;
    int second;
};

int main() {
  int arr[10];
  struct pair a;

  a.first = arr[1];
  …
```

```llvm
%struct.pair = type{ i32, i32 }

define i32 @main() {
entry:
  %arr = alloca [10 x i32]
  %a = alloca %struct.pair

  %arrayidx = getelementptr
     [10 x 32]* %arr,i32 0,i64 1

  %0 = load i32* %arrayidx

  %first = getelementptr
    %struct.pair* %a,i32 0,i32 0

  %store i32 %0, i32* %first
```

Aggregate Type Example 2

```c
struct RT {
  char A;
  int B[10][20];
  char C;
};
struct ST {
  int X;
  double Y;
  struct RT Z;
};

int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}
```

```llvm
%struct.RT = type { i8, [10 x [20 x i32]], i8 }
%struct.ST = type { i32, double, %struct.RT }

define i32* @foo(%struct.ST* %s) nounwind uwtable readnone optsize ssp {
entry:
  %arrayidx = getelementptr inbounds %struct.ST* %s, i64 1, i32 2, i32 1, i64 5, i64 13
  ret i32* %arrayidx
}
```

Integer Conversion (1/2)

- Truncate
  - Syntax: **`<res> = trunc <iN1> <value> to <iN2>`** where **`iN1`** and **`iN2`** are of integer type, and **`N1`** > **`N2`**
  - Examples
    - `%X = trunc i32 257 to i8 ;%X becomes i8:1`
    - `%Y = trunc i32 123 to i1 ;%Y becomes i1:true`
    - `%Z = trunc i32 122 to i1 ;%Z becomes i1:false`

Integer Conversion (2/2)

- Zero extension
  - **`<res> = zext <iN1> <value> to <iN2>`** where **`iN1`** and **`iN2`** are of integer type, and **`N1`** < **`N2`**
  - Fill the remaining bits with zero
  - Examples
    - `%X = zext i32 257 to i64 ;%X becomes i64:257`
    - `%Y = zext i1 true to i32 ;%Y becomes i32:1`
- Sign extension
  - **`<res> = sext <iN1> <value> to <iN2>`** where **`iN1`** and **`iN2`** are of integer type, and **`N1`** < **`N2`**
  - Fill the remaining bits with the sign bit (the highest order bit) of `value`
  - Examples
    - `%X = sext i8 -1 to i16 ;%X becomes i16:65535`
    - `%Y = sext i1 true to i32 ;%Y becomes` $i32{:}2^{32}-1$

Other Conversions

- Float-to-float
  - `fptrunc .. to`, `fpext .. to`
- Float-to-integer (vice versa)
  - `fptoui .. to`, `tptosi .. to`, `uitofp .. to`, `sitofp .. to`
- Pointer-to-integer
  - `ptrtoint .. to`, `inttoptr .. to`
- Bitcast
  - `<res> = bitcast <t1> <value> to <t2>`
    where `t1` and `t2` should be different types and have the same size

Computational Instructions

- Binary operations:
  - Add: `add`, `sub` , `fsub`
  - Multiplication: `mul` , `fmul`
  - Division: `udiv` , `sdiv` , `fdiv`
  - Remainder: `urem` , `srem` , `frem`

- Bitwise binary operations
  - shift operations: `shl` , `lshl` , `ashr`
  - logical operations: `and` , `or` , `xor`

Add Instruction

- `<res> = add [nuw][nsw] <iN> <op1>, <op2>`
  - nuw (no unsigned wrap): if unsigned overflow occurs, the result value becomes a poison value (undefined)
    - E.g: `add nuw i8 255, i8 1`
  - nsw (no signed wrap): if signed overflow occurs, the result value becomes a poison value
    - E.g. `add nsw i8 127, i8 1`

Control Representation

- The LLVM front-end constructs the control flow graph (CFG) of every function explicitly in LLVM IR
  - A function has a set of basic blocks each of which is a sequence of instructions
  - A function has exactly one entry basic block
  - Every basic block is ended with exactly one *terminator* instruction which explicitly specifies its successor basic blocks if there exist.
    - Terminator instructions: branches (conditional, unconditional), return, unwind, invoke
- Due to its simple control flow structure, it is convenient to analyze, transform the target program in LLVM IR

Label, Return, and Unconditional Branch

- A label is located at the start of a basic block
  - Each basic block is addressed as the start label
  - A label `x` is referenced as register `%x` whose type is label
  - The label of the entry block of a function is "`entry`"
- Return **`ret <type> <value> | ret void`**
- Unconditional branch **`br label <dest>`**
  - At the end of a basic block, this instruction makes a transition to the basic block starting with label **`<dest>`**
  - E.g: `br label %entry`

Conditional Branch

- <res> = icmp <cmp> <ty> <op1>, <op2>
  - Returns either true or false (i1) based on comparison of two variables (op1 and op2) of the same type (ty)
  - cmp: comparison option

    eq (equal), ne (not equal), ugt (unsigned greater than),
    uge (unsigned greater or equal), ult (unsigned less than),
    ule (unsigned less or equal), sgt (signed greater than),
    sge (signed greater or equal), slt (signed less than), sle (signed less or equal)
- br i1 <cond>, label <thenbb>, label <elsebb>
  - Causes the current execution to transfer to the basic block <thenbb> if the value of <cond> is true; to the basic block <elsebb> otherwise.

- Example:

```
if (x > y)
  return 1 ;
return 0 ;
```

```
%0 = load i32* %x
%1 = load i32* %y
%cmp = icmp sgt i32 %0, %1
br i1 %cmp, label %if.then, label %if.end
if.then:
…
```

Switch

- **`switch <iN> <value>, label <defaultdest> [<iN> <val>, label <dest> …]`**
  - Transfer control flow to one of many possible destinations
  - If the value is found (`val`), control flow is transferred to the corresponding destination (`dest`); or to the default destination (`defaultdest`)
  - Examples:

```
switch(x) {
    case 1:
        break ;
    case 2:
        break ;
    default:
       break ;
}
```

```
%0 = load i32* %x
switch i32 %0, label %sw.default [
  i32 1, label %sw.bb
  i32 2, label %sw.bb1]
sw.bb:
  br label %sw.epilog
sw.bb1:
  br label %sw.epilog
sw.default:
  br label %sw.epilog
sw.epilog:
  …
```

PHI (Φ) instruction

- `<res> = phi <t> [ <val_0>, <label_0>], [ <val_1>, <label_1>], …`
  - Return a value `val_i` of type `t` such that the basic block executed right before the current one is of `label_i`

- Example

```
y = (x > 0) ? x : 0 ;
```

```
%0 = load i32* %x
%c = icmp sgt i32 %0 0
br i1 %c, label %c.t, %c.f

c.t:
 %1 = load i32* %x
 br label %c.end

c.f:
 br label %c.end

c.end:
 %cond = phi i32 [%1, %c.t], [0, %c.f]
 store i32 %cond, i32* %y
```

Function Call

- `<res> = call <t> [<fnty>*] <fnptrval>(<fn args>)`
  - `t`: the type of the call return value
  - `fnty`: the signature of the pointer to the target function (optional)
  - `fnptrval`: an LLVM value containing a pointer to a target function
  - `fn args`: argument list whose types match the function signature

- Examples:

```c
printf("%d", abs(x));
```

```llvm
@.str = [3 x i8] c"%d\00"

%0 = load i32* %x
%call  = call i32 @abs(i32 %0)

%call1 = call i32 (i8*, ...)*
    @printf(i8*
     getelementptr ([3 x i8]* @.str,
      i32 0, i32 0),
    i32 %call)
```

Unaddressed Issues

- Many options/attributes of instructions
- Vector data type (SIMD style)
- Exception handling
- Object-oriented programming specific features
- Concurrency issues
  - Memory model, synchronization, atomic instructions

* *http://llvm.org/docs/LangRef.html*

参考资料

- gcc与clang/llvm比较
  - https://www.alibabacloud.com/blog/gcc-vs--clangllvm-an-in-depth-comparison-of-cc%2B%2B-compilers_595309
  - https://stackoverflow.com/questions/40799696/how-is-gcc-ir-different-from-llvm-ir
  - https://blog.csdn.net/m0_37477061/article/details/85993447
- Architecture of llvm
  - http://www.aosabook.org/en/llvm.html

语法制导的翻译方案

- Syntax-Directed Translation scheme SDT
- 产生式中嵌入了程序片段

中缀表达式转后缀表达式

$$E \rightarrow TR$$
$$R \rightarrow opTR_1 | \varepsilon$$
$$T \rightarrow num$$

$$E \rightarrow TR$$
$$R \rightarrow op\ T$$
$$R_1 \qquad \{\text{print(op.str)}\}$$
$$|\varepsilon$$
$$T \rightarrow num \qquad \{\text{print(num.val)}\}$$

说明语句

- 语法
  - $D \rightarrow T\ id\ ;\ D \mid \varepsilon$
  - $T \rightarrow BC$
  - $B \rightarrow int|float$
  - $C \rightarrow \varepsilon|[num]C$
- 语义
  - 变量名
  - 变量类型 type
  - 变量地址 offset / 变量大小 width

说明语句

- 类型

$T \rightarrow B$	{t=B.type; w=B.width;}
$\quad\quad$ C	{T.type=C.type; T.width=C.width;}
$B \rightarrow int$	{B.type=int; B.width=4;}
$B \rightarrow float$	{B.type=float; B.width=8;}
$C \rightarrow \varepsilon$	{C.type=t; C.width=w;}
$C \rightarrow [num]C_1$	{C.type=array(num.val, C1.type); C.width=num.val * C1.width;}

说明语句

- 变量

$P \rightarrow$ {offset=0;}

D

$D \rightarrow T\ id;$ {enter(id.name, T.type, offset);

offset += T.width;}

D_1

$D \rightarrow \varepsilon$

算术表达式

- 语法
  - $S \rightarrow id = E$
  - $E \rightarrow E_1 + E_2$
  - $E \rightarrow E_1 * E_2$
  - $E \rightarrow -E_1$
  - $E \rightarrow (E_1)$
  - $E \rightarrow id$
- 三地址代码
  - addr1 = addr2 + addr 3
  - get(var)
  - new temp()

算术表达式

增量翻译

$S \rightarrow id = E$	{gen(get(id.name)=E.addr);}
$E \rightarrow E_1 + E_2$	{E.addr = new temp(); gen(E.addr=E1.addr+E2.addr);}
$E \rightarrow E_1 * E_2$	{E.addr = new temp(); gen(E.addr=E1.addr*E2.addr);}
$E \rightarrow -E_1$	{E.addr = new temp(); gen(E.addr= minus E1.addr);}
$E \rightarrow (E_1)$	{E.addr = E1.addr;}
$E \rightarrow id$	{E.addr = get(id.name);}

其他内容

- 布尔表达式
- 控制流
- 类型检查
- 等等……