Compiler Principle

Prof. Dongming LU

Apr. 8th, 2024

Content

1. INTRODUCTION

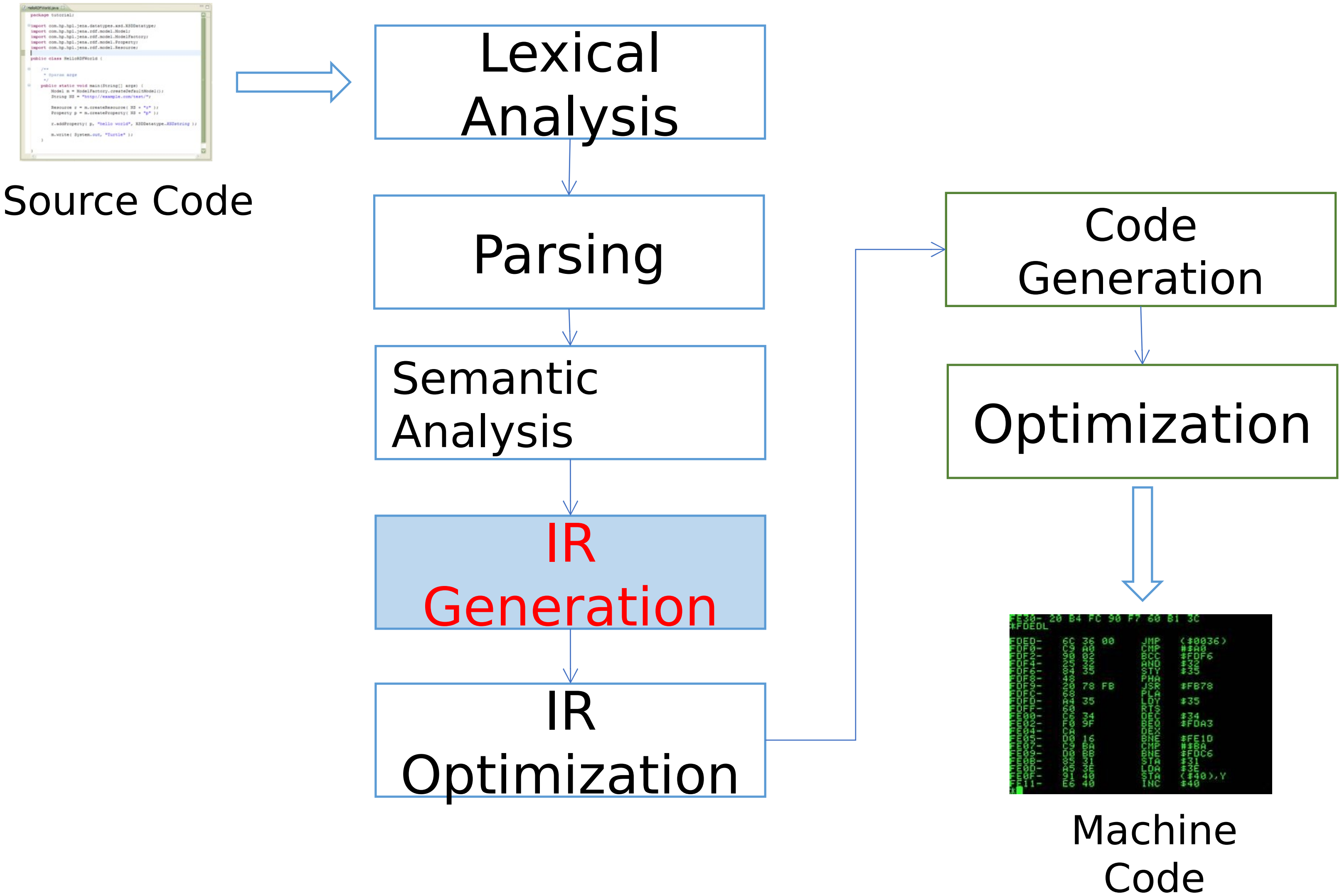
2. LEXICAL ANALYSIS
3. PARSING
4. ABSTRACT SYNTAX
5. SEMANTIC ANALYSIS

6. ACTIVATION RECORD
7. **TRANSLATING INTO INTERMEDIATE CODE**

8. OTHERS

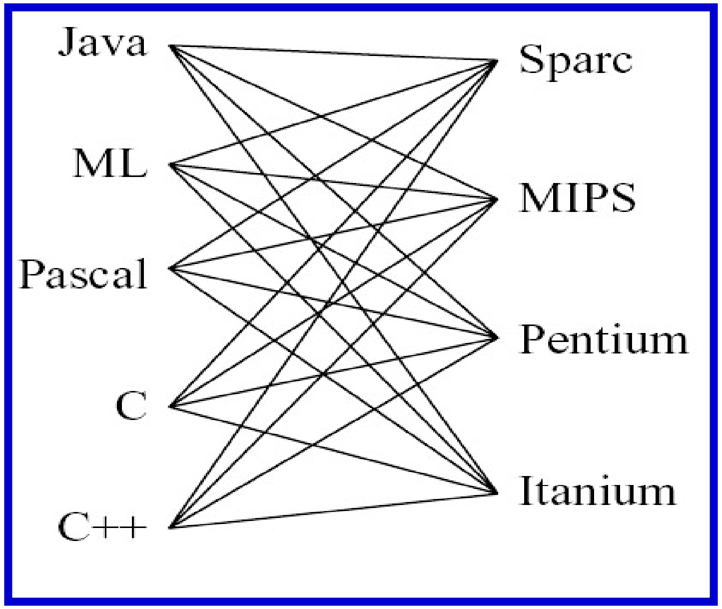
7 Translation to Intermediate Code

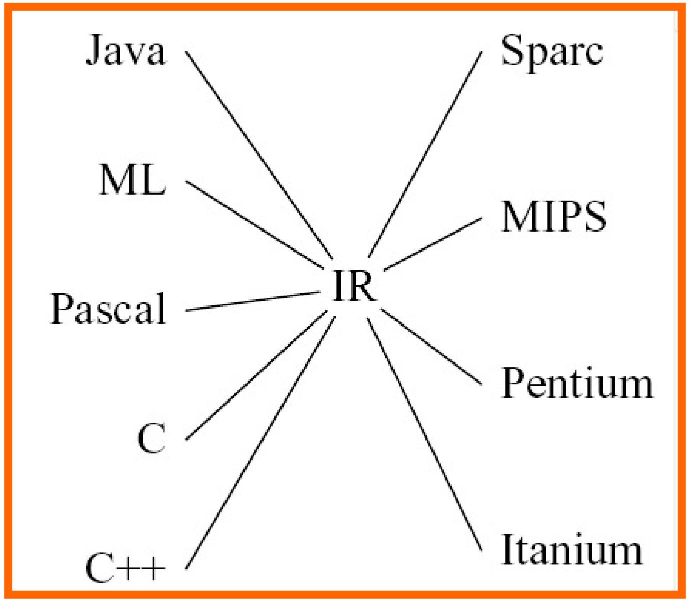
Where we are


Source Code
Lexical Analysis
Parsing
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Motivation

- **Translating directly to real machine code**
  - ✓ **hinders portability and modularity.**





Motivation

- **The front end**
  - ✓ **lexical analysis**
  - ✓ **Parsing**
  - ✓ **Semantic analysis**
  - ✓ **Translation to intermediate representation.**

- **The back end**
  - ✓ **IR optimization**
  - ✓ **Translation into machine language.**

What is Intermediate Code

- An intermediate representation (IR) is a kind of abstract machine language
  - ✓ Express the target-machine operations without committing to too much machine-specific detail.
  - ✓ Be independent of the details of the source language.

What is Intermediate Code

IR: Three-Address Code

- Basic instruction:

  x = y op z

- Given expression: 2*a+(b-3) , the corresponding three-address instructions are as follows:

  T1 = 2 * a
  T2 = b – 3
  T3 = T1 + T2

What is Intermediate Code

IR: Three-Address Code

- Four fields are necessary (a quadruple)
  - ✓ One for the operation
  - ✓ Three for the addresses

- One or more of the address fields is given a null or “empty” value

- The entire sequence of three-address instructions is implemented as
  - ✓ An array
  - ✓ Linked list.

Example

High-level language

```
read x ;  { input an integer }
if 0 < x then { don't compute if x
    fact:=1;
repeat
    fact:=fact*x;
    x:=x-1
until x=0;
write fact { output factorial of x
end
```

Three-address code

```
read x
t1=x>0
if_false t1 goto L1
fact=1
label   L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4= x= =0
if_false t4 goto  L2
write fact
label   L1
halt
```

7.1 Intermediate Representation Trees

The requirements for IR

- A good intermediate representation has **several qualities**
  - ✓ **Convenient** for the semantic analysis phase to produce.
  - ✓ **Convenient** to translate into all real machine language.
  - ✓ Each construct must have **a clear and simple** meaning, IR optimizing transformations can easily be specified and implemented.

```
/*tree.h*/

Typedef struct T_exp_ *T_exp;
Struct T_exp_ {enum {T_BINOP, T_MEM, T_TEMP, …,
T_CALL} kind;
                  union {struct {T_binop op; T_exp left,
right;} BINOP;
                                  …
                              } u; };

T_exp T_Binop(T_binOp, T_exp, T_exp);
T_exp T_Mem(T_exp);
T_exp T_Temp(Temp_temp);
T_exp T_Eseq(T_stm, T_exp);
T_exp T_Name(Temp_label);
T_exp T_const(int);
T_exp T_call(T_exp, T_expList);
…
```

Figure 7.2 The intermediate representation tree

The expressions (T_exp)

CONST(*i*)	**The integer constant *i***
NAME(*n*)	**The symbolic constant *n* (e.g. label)**
TEMP(*t*)	**Temporary *t.***
BINOP(o, e1, e2)	The application of binary operator *o* to operands *e*1, *e*2. The integer arithmetic operators are PLUS, MINUS, MUL, DIV; the integer bitwise logical operators are AND, OR, XOR; the integer logical shift operators are LSHIFT, RSHIFT; the integer arithmetic right-shift is ARSHIFT.
MEM(*e*)	The contents of *wordSize* bytes of memory starting at address *e.* When MEM is used as the left child of a MOVE, it means "store", but anywhere else it means "fetch“.
CALL(*f, l*)	A procedure call: the application of function *f* to argument list *l*.
ESEQ(*s, e*)	The statement *s* is evaluated for side effects, then *e* is evaluated for a result.

```
/*tree.h*/

Typedef struct T_stm_ *T_stm;
Struct T_stm_ {enum {T_SEQ, T_LABEL, T_JUMP, …,
T_EXP} kind;
                    union {struct {T_stm left, right;} SEQ;
                              …
                             } u; };

T_stm T_seq(T_stm left, T_stm right);
T_stm T_Label(Temp_label);
T_stm T_Jump(T_exp exp, Temp_labelList labels);
T_stm T_Cjump(T_relOp op, T_exp left, T_exp rght,
                                   Temp_label true, Temp_label
false);
T_stm T_move(T_exp, T_exp);
T_stm T_exp(T_exp);
…
```

Figure 7.2 The intermediate representation tree

The statements (T_stm)

MOVE(TEMP *t, e*)	Evaluate *e* and move it into temporary *t*.
MOVE(MEM(e_1) e_2)	Evaluate e_1, yielding address *a*. Then evaluate e_2, and store the result into *wordSize* bytes of memory starting at *a*.
EXP(*e*)	Evaluate *e* and discard the result.
JUMP(*e, labs*)	Transfer control (jump) to address e. The destination *e* may be a literal label, as in NAME(*lab*), or it may be an address calculated by any other kind of expression.
CJUMP(*o*, e_1, e_2, *t*, *f*)	Evaluate e_1, e_2 in that order, yielding values *a*, *b*. Then compare *a, b* using the relational operator *o*. If the result is true, jump to *t*; otherwise jump to *f*.
SEQ(s_1, s_2)	The statement s_1 followed by s_2.
LABEL(*n*)	Define the constant value of name *n* to be the current machine code address.

/*tree.h*/

Typedef struct T_expList_ *T_expList;
Struct T_expList_ {T_exp head; T_expList tail;};
T_expList T_ExpList(T_exp head, T_expList tail);

Typedef struct T_stmList_ *T_stmList;
Struct T_stmList_ {T_stm head; T_stmList tail;};
T_stmList T_StmList(T_stm head, T_stmList tail);

Typedef enum {T_plus, T_minus, T_mul, T_div, T_and, T_or,
T_lshift, T_rshift, T_arshift, T_xor} T_binOp;
Typedef enum {T_eq, T_ne, T_lt, T_gt, T_le, T_ge,
T_ult, T_ule, T_ugt, T_uge} T_relOp;

Figure 7.2 The intermediate representation tree

The end of Chapter 7(1)