IR Generating

C++ Implementation

Outline

- 1.Introduction
- 2.Journal of testing
- 3.Conclusion

Outline

- **1.Introduction**
- 2.Journal of testing
- 3.Conclusion

Overview

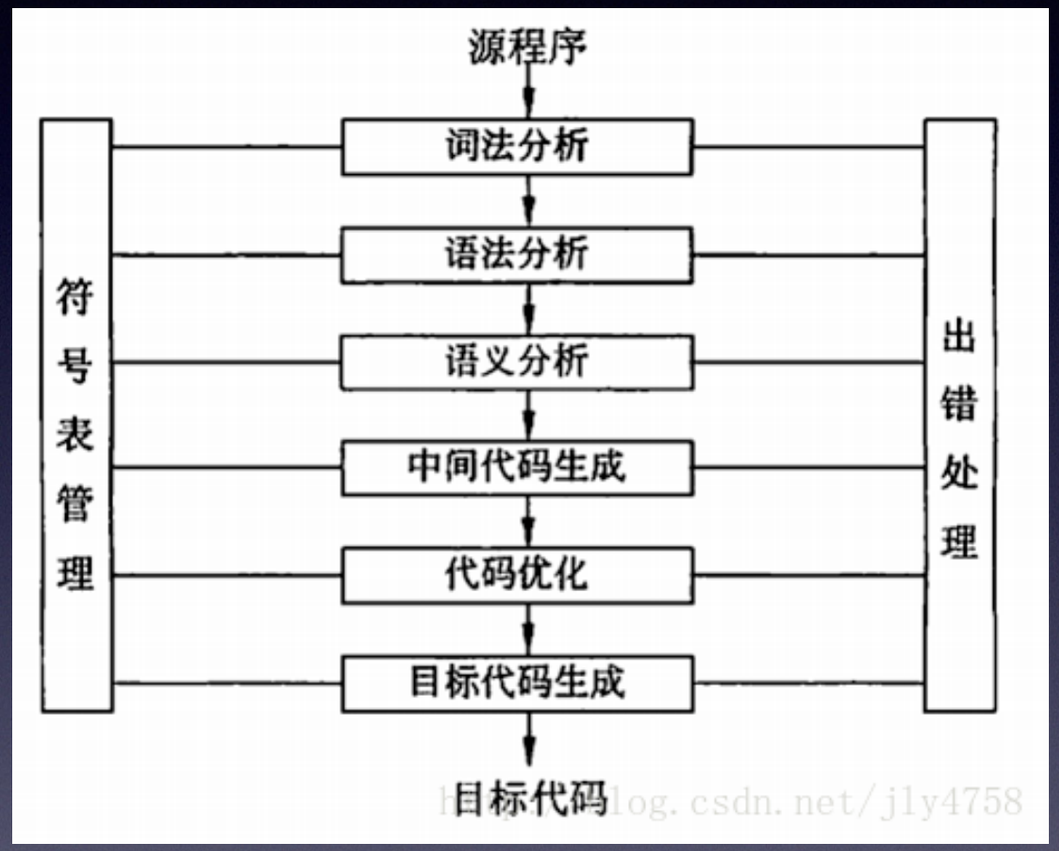
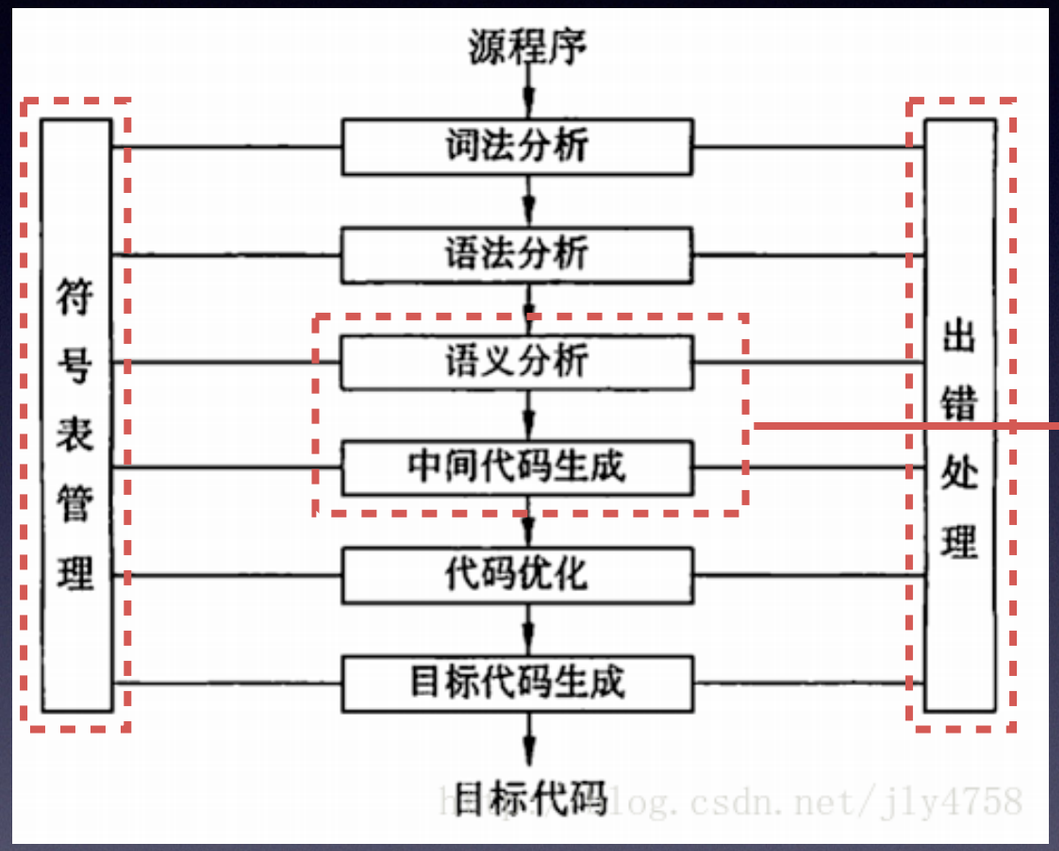


Figure: Processing Pipeline

Overview



We are here now!

Figure: Processing Pipeline

SDD(Syntax-Directed Definition)

- Associating **attributes** to grammar symbols, eg. E.code
- Associating **semantic rules** to productions, eg. T.code = ‘0’

	SDD
$E \rightarrow E + T$	$E.code = E.code \| \| T.code \| \| '+'$
$E \rightarrow E - T$	$E.code = E.code \| \| T.code \| \| '-'$
$E \rightarrow T$	$E.code = T.code$
$T \rightarrow 0$	$T.code = '0'$
$T \rightarrow 1$	$T.code = '1'$
$\ldots$	
$T \rightarrow 9$	$T.code = '9'$

Figure: An Example of SDD

SDD vs SDT scheme

- SDD: **associating** semantic rules with the productions.
- SDT scheme: **embeds** semantic actions with productions.
- SDD: easier to **read**; easy for **specification**.
- SDT scheme: easy for **implementation**.

SDTScheme		*SDD*	
$E \to E + T$	$\{print'+'\}$	$E \to E + T$	$E.code = E.code \| \| T.code \| \| '+'$
$E \to E - T$	$\{print'-'\}$	$E \to E - T$	$E.code = E.code \| \| T.code \| \| '-'$
$E \to T$		$E \to T$	$E.code = T.code$
$T \to 0$	$\{print'0'\}$	$T \to 0$	$T.code =' 0'$
$T \to 1$	$\{print'1'\}$	$T \to 1$	$T.code =' 1'$
$\cdots$		$\cdots$	
$T \to 9$	$\{print'9'\}$	$T \to 9$	$T.code =' 9'$

Figure: SDT Scheme vs SDD

TAC(Three Address Code)

- The addresses of operator1, operator2 and target.
- A common type of Intermediate Representation(IR).

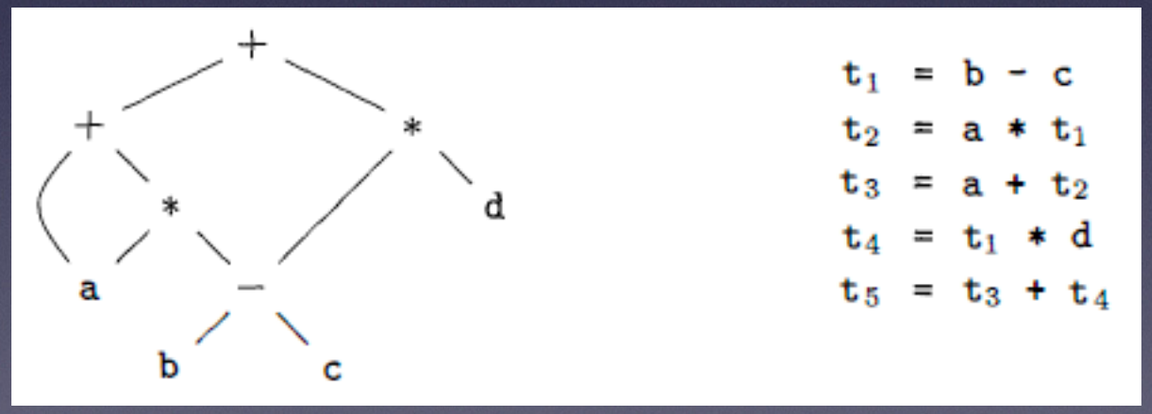


Figure: An Example of TAC

TAC(Three Address Code)

- Quadruple: (op, arg1, arg2, result)
- Triple: (op, arg1, arg2)

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

a) 三地址代码

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
	...			

b) 四元式

图 6-10　三地址代码及其四元式表示

Figure: An Example of Quadruple

Expression to IR

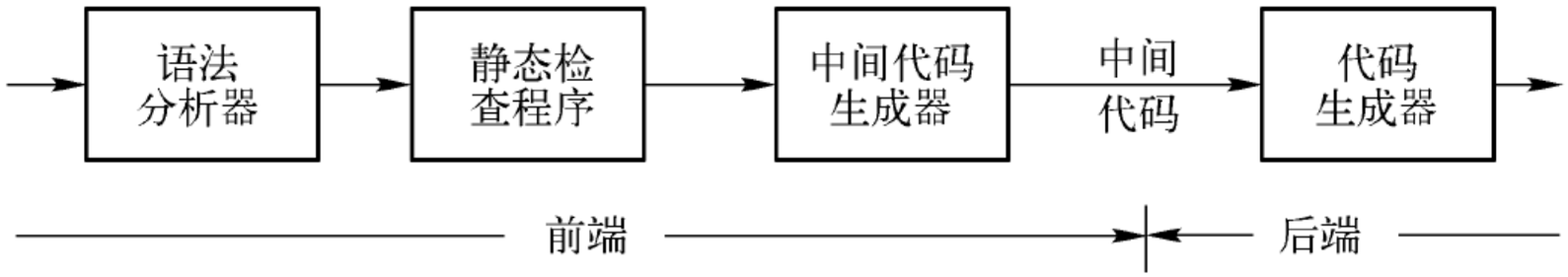


图 6-1　一个编译器前端的逻辑结构

Figure: The Structure of a Front-end Compiler

Expression to IR

x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)

Expressions

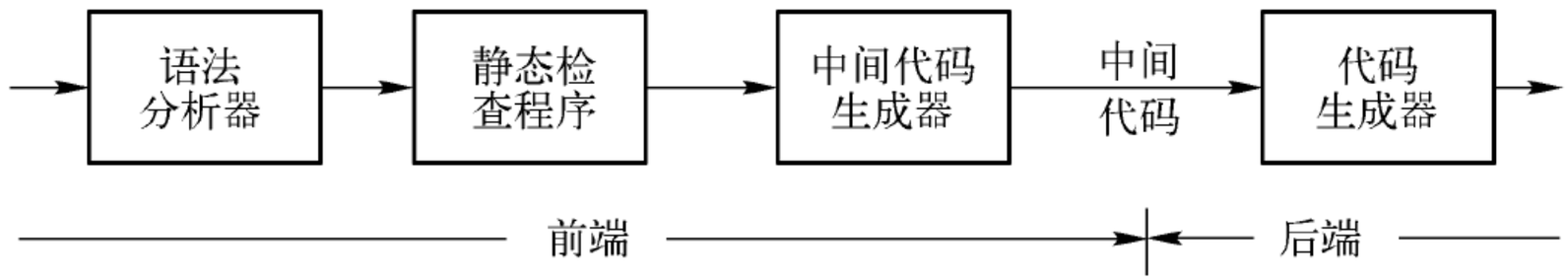


图 6-1　一个编译器前端的逻辑结构

Figure: The Structure of a Front-end Compiler

Expression to IR

x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)

Expressions

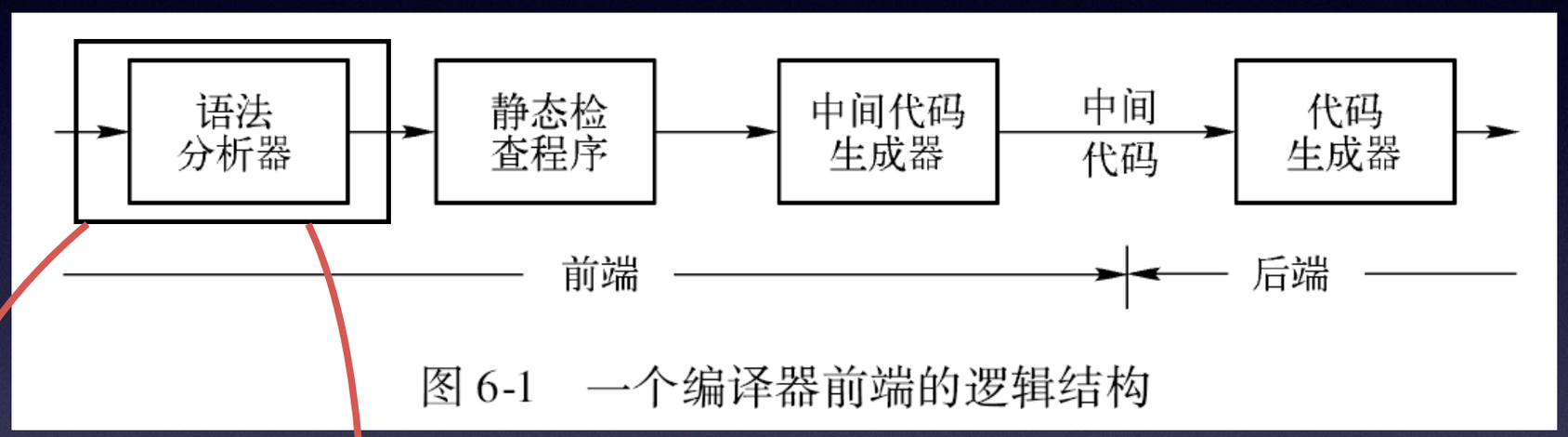
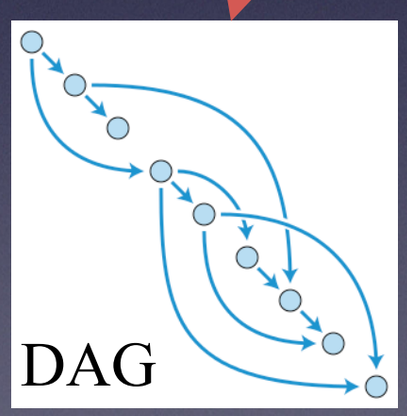
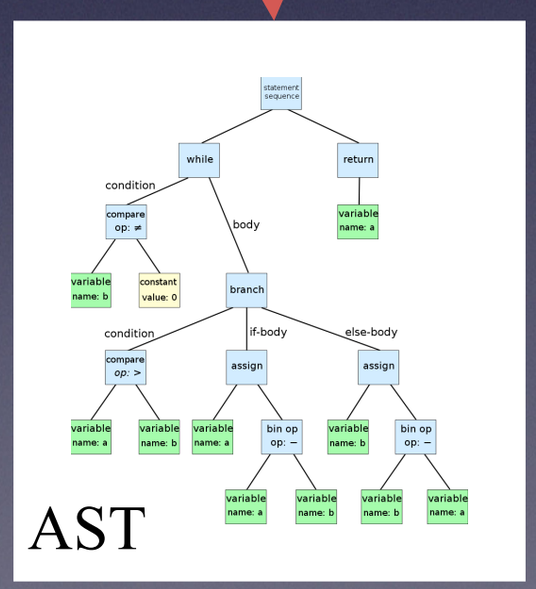


图 6-1　一个编译器前端的逻辑结构





Expression to IR

x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)

Expressions

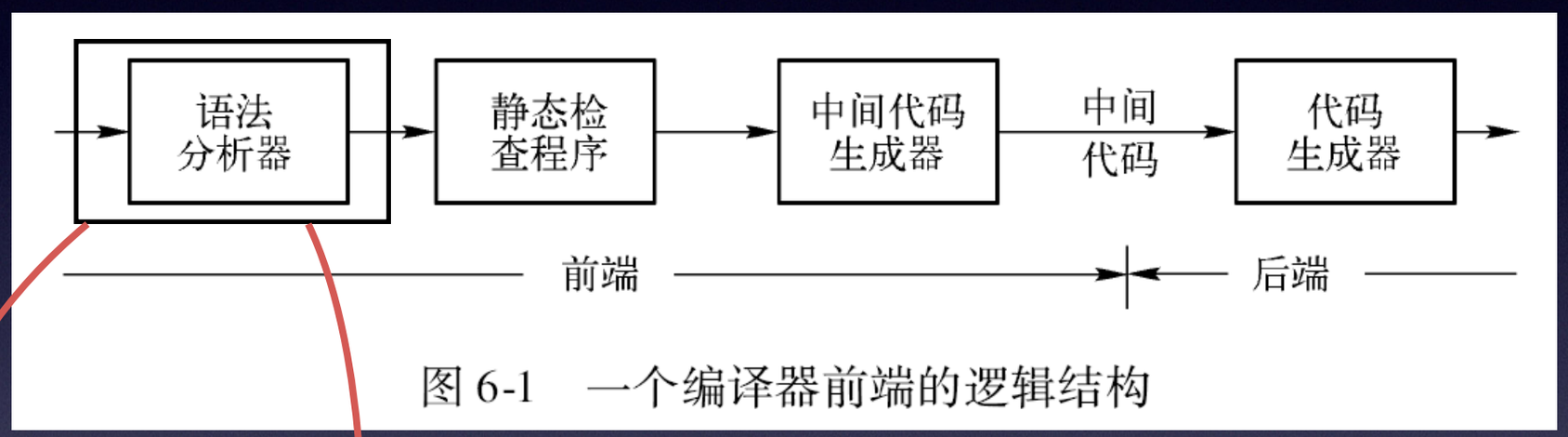
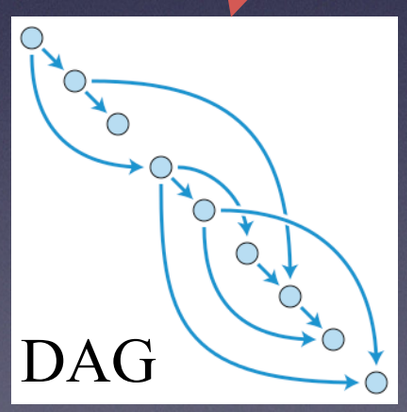
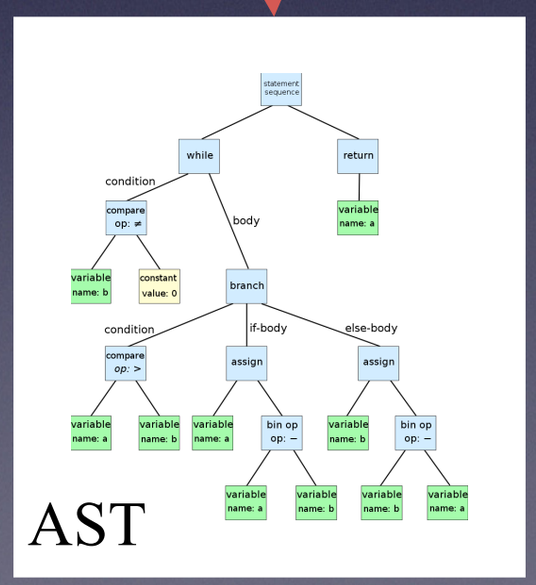


图 6-1　一个编译器前端的逻辑结构





How to convert AST to IR like TAC?

(1)**Evaluates** the meaning at each node

(2)**Emits** the meaning as semantic rules

SDD	
$E \to E + T$	$E.code = E.code \| \| T.code \| \| '+'$
$E \to E - T$	$E.code = E.code \| \| T.code \| \| '-'$
$E \to T$	$E.code = T.code$
$T \to 0$	$T.code = '0'$
$T \to 1$	$T.code = '1'$
$\cdots$	
$T \to 9$	$T.code = '9'$

SDD

x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)

Expressions

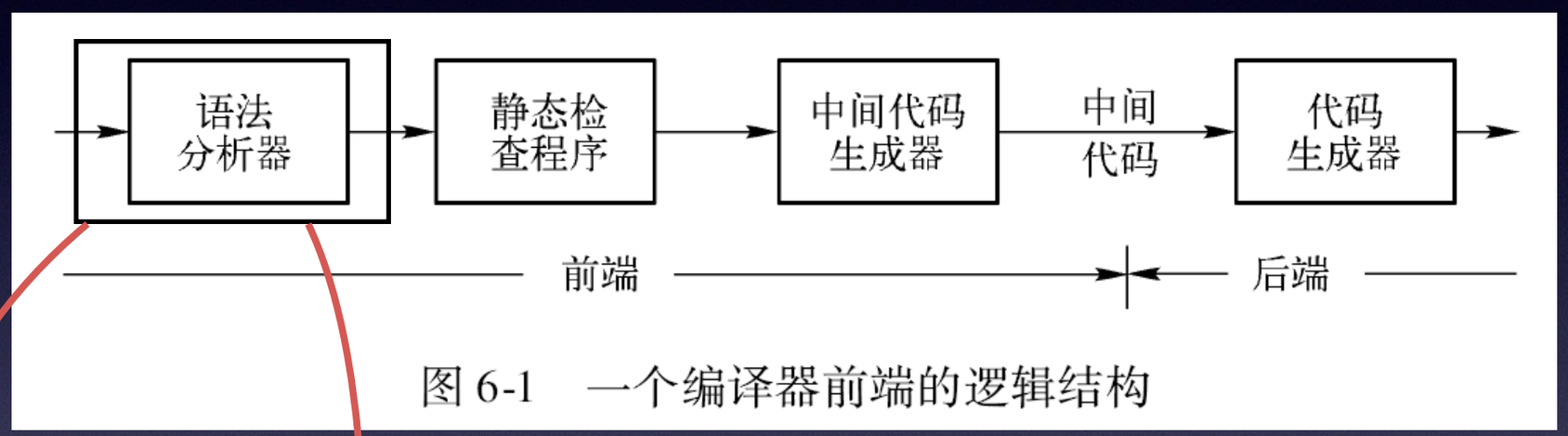
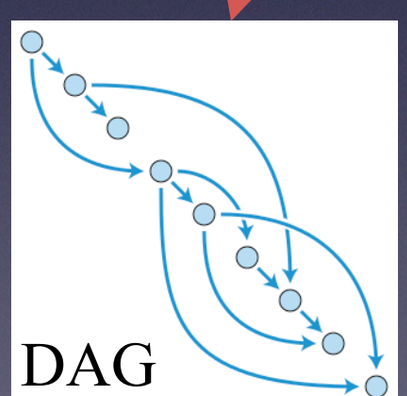
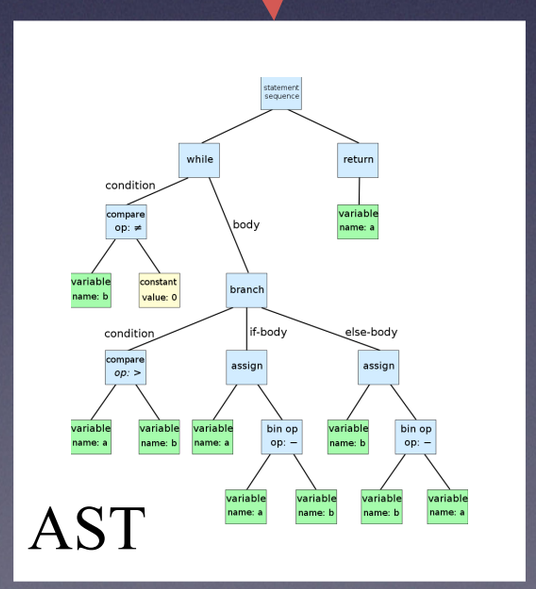


图 6-1　一个编译器前端的逻辑结构





(1)**Evaluates** the meaning at each node

(2)**Emits** the meaning as semantic rules

SDD	
$E \to E + T$	$E.code = E.code \| \| T.code \| \| '+'$
$E \to E - T$	$E.code = E.code \| \| T.code \| \| '-'$
$E \to T$	$E.code = T.code$
$T \to 0$	$T.code = '0'$
$T \to 1$	$T.code = '1'$
$\cdots$	
$T \to 9$	$T.code = '9'$

SDD

x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)

Expressions

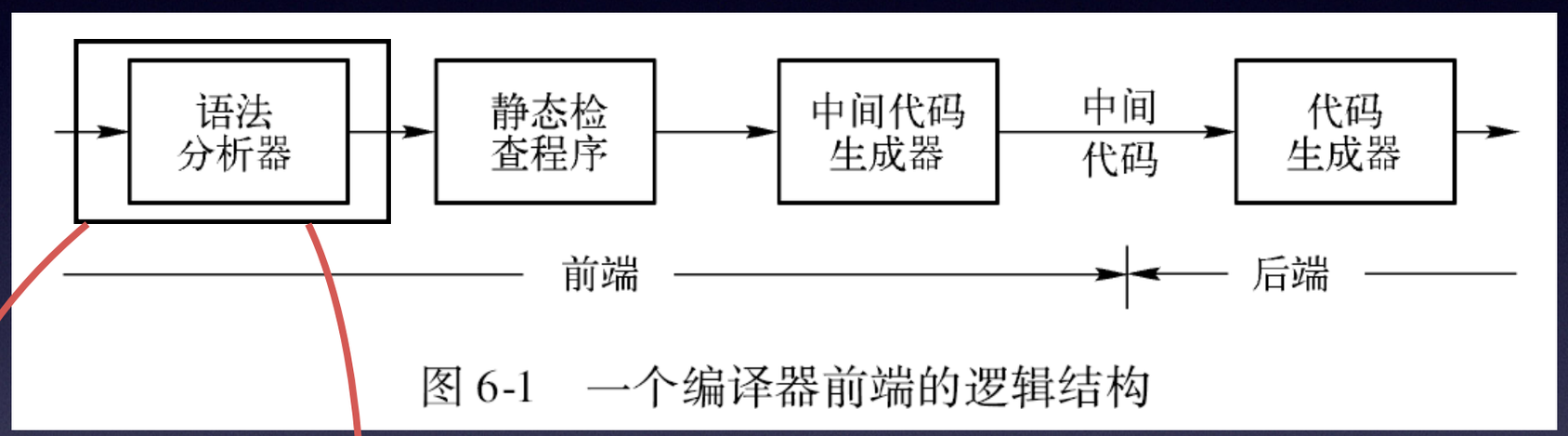
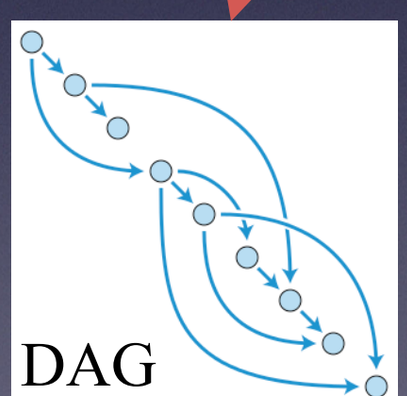
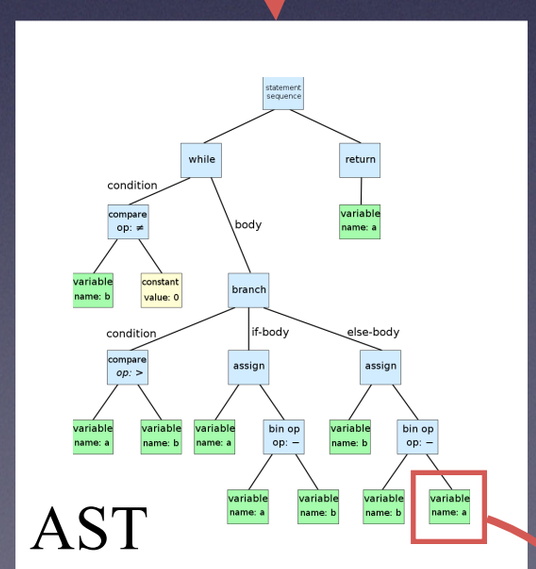


图 6-1　一个编译器前端的逻辑结构





Attributes

(1)**Evaluates** the meaning at each node

(2)**Emits** the meaning as semantic rules

SDD	
$E \to E + T$	$E.code = E.code \| \| T.code \| \| '+'$
$E \to E - T$	$E.code = E.code \| \| T.code \| \| '-'$
$E \to T$	$E.code = T.code$
$T \to 0$	$T.code = '0'$
$T \to 1$	$T.code = '1'$
$\cdots$	
$T \to 9$	$T.code = '9'$

SDD

x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)

Expressions

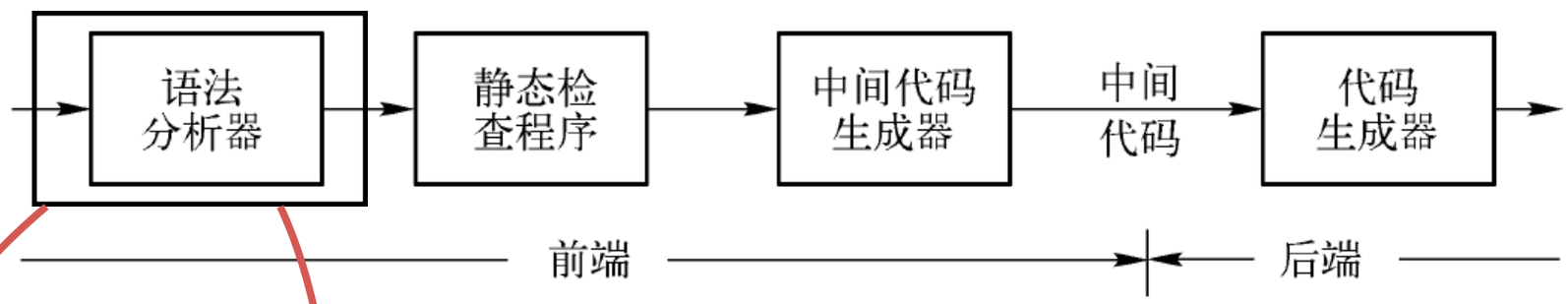
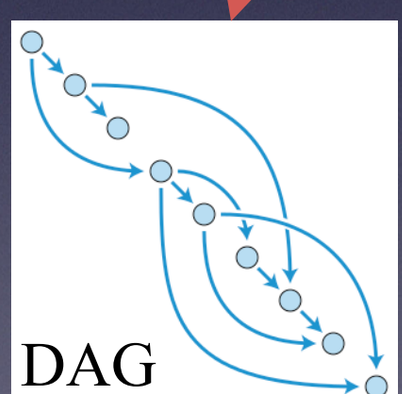
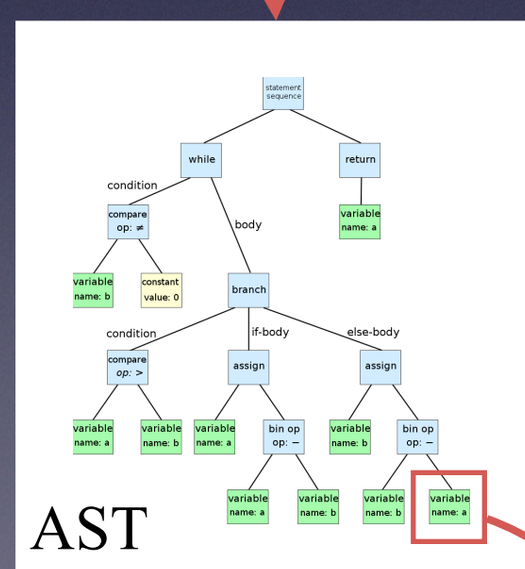


图 6-1　一个编译器前端的逻辑结构

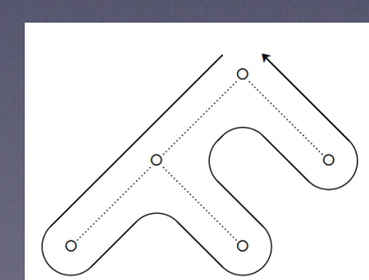




Attributes

Attaching **semantic rules** to productions

$$E \to E - T \quad E.code = E.code \| T.code \| '-'$$



(1)**Evaluates** the meaning at each node

(2)**Emits** the meaning as semantic rules

SDD	
$E \to E + T$	$E.code = E.code \| \| T.code \| \| '+'$
$E \to E - T$	$E.code = E.code \| \| T.code \| \| '-'$
$E \to T$	$E.code = T.code$
$T \to 0$	$T.code = '0'$
$T \to 1$	$T.code = '1'$
$\cdots$	
$T \to 9$	$T.code = '9'$

SDD

x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)

Expressions

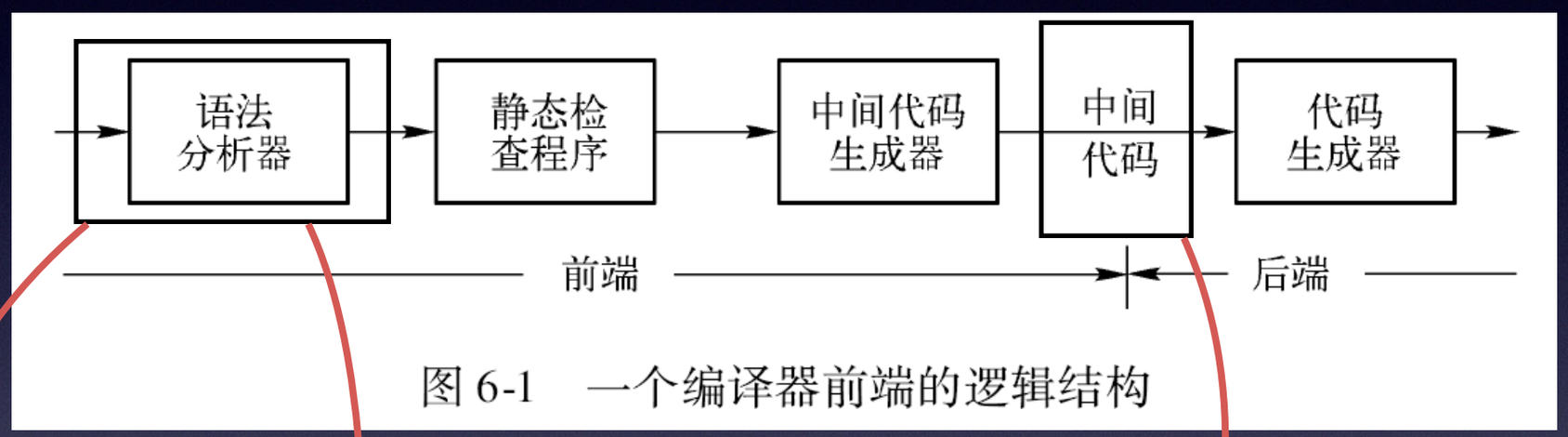
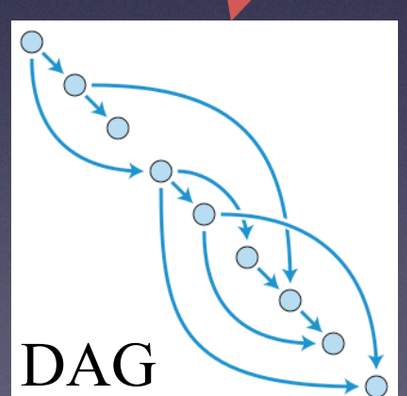
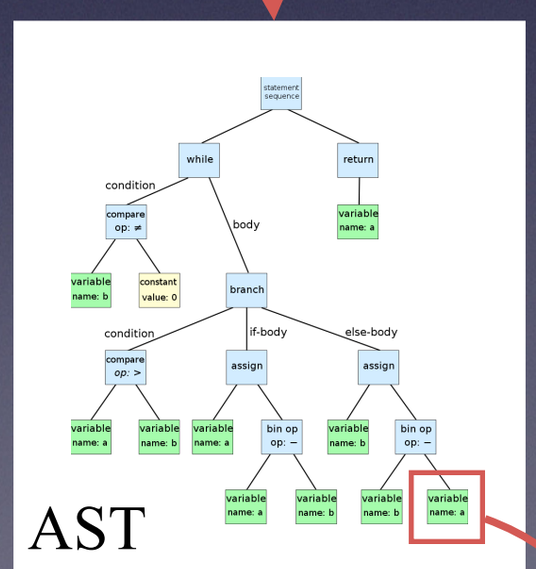


图 6-1　一个编译器前端的逻辑结构





convert to

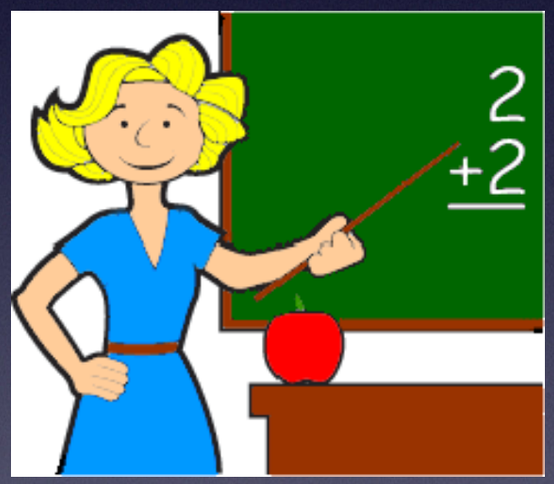
```
t1 := b * b
t2 := 4 * a
t3 := t2 * c
t4 := t1 - t3
t5 := sqrt(t4)
t6 := 0 - b
t7 := t5 + t6
t8 := 2 * a
t9 := t7 / t8
x := t9
```

TAC

Attributes

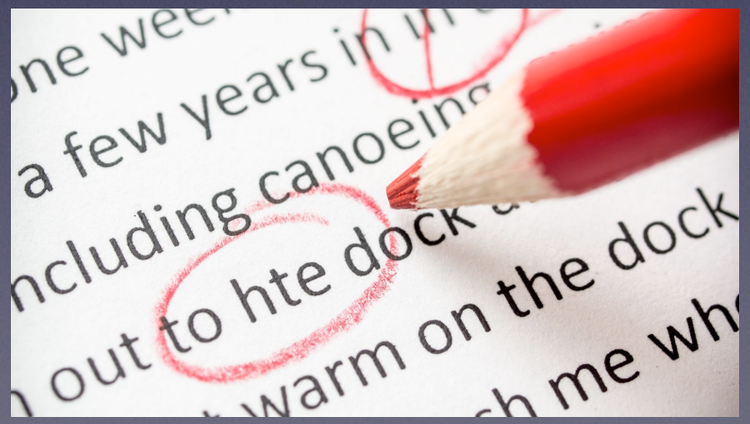
Outline

- 1.Introduction
- **2.Journal of testing**
- 3.Conclusion



Input Grammar

- S -> program id (id_lists); compound_stmt .
- id_lists -> id | id_lists , id
- compound_stmt -> begin optional_stmts end
- optional_stmts -> stmts | ε
- stmts -> stmt | stmts; stmt
- stmt -> id := expr | compound_stmt | if bool then stmt | if bool then stmt else stmt | while bool do stmt | repeat stmts until bool
- bool -> expr < expr
- expr -> expr + term | expr - term | term
- term -> term * factor | term / factor | factor
- factor -> id | num



Dismiss Shift-Reduce Conflicts

- **stmt => if bool then stmt | if bool then stmt else stmt**
- As meeting ‘else’, we apply ‘**shift**’ in our program.
- “else” is always matched against the **adjacent** “if”.

Dismiss Shift-Reduce Conflicts

- Refer to the implementation of “Free Pascal”
- Website: https://www.freepascal.org/

SDD Construction

sdd.txt

```
"S' -> S"
        S.next = newlabel();
        S'.code = S.code || label(S.next);


"S -> program id;  compound_stmt ."
        compound_stmt.next = newlabel();
        S.code = compound_stmt.code || label(compound_stmt.next);


"S -> program ( id_lists ) ; compound_stmt ."
        compound_stmt.next = newlabel();
        S.code = compound_stmt.code || label(compound_stmt.next);


"id_lists -> id"
        id_lists.code = '';


"id_lists -> id_lists1, id"
        id_lists.code = '';


"compound_stmt -> begin optional_stmts end"
        optional_stmts.next = newlabel();
        compound_stmt.code = optional_stmts.code || label(optional_stmts.next);


"optional_stmts -> stmts"
        stmts.next = newlabel();
```

“production” + rules

Figure: SDD, referring to the Dragon Book

SDD Construction

sdd.txt

```
"S' -> S"
        S.next = newlabel();
        S'.code = S.code || label(S.next);


"S -> program id;  compound_stmt ."
        compound_stmt.next = newlabel();
        S.code = compound_stmt.code || label(compound_stmt.next);
```

```
"compound_stmt -> begin optional_stmts end"
        optional_stmts.next = newlabel();
        compound_stmt.code = optional_stmts.code || label(optional_stmts.next);
```

```
        id_lists.code = '';


"id_lists -> id_lists1, id"
        id_lists.code = '';
```

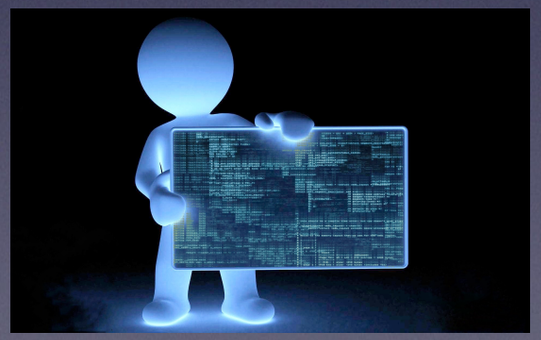
```
"compound_stmt -> begin optional_stmts end"
        optional_stmts.next = newlabel();
        compound_stmt.code = optional_stmts.code || label(optional_stmts.next);
```

```
"optional_stmts -> stmts"
        stmts.next = newlabel();
```

Figure: SDD, referring to the Dragon Book

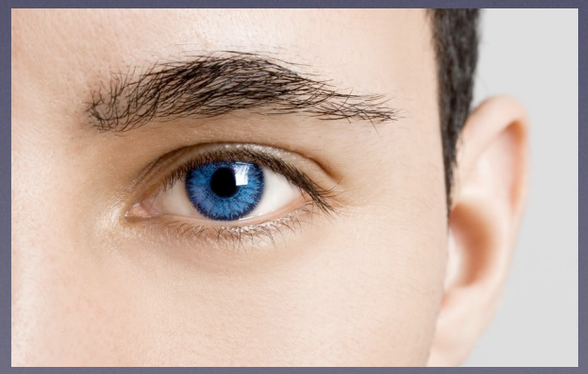
Test Program

```
program (a,b);
  begin
    x:=19;
    if x<y then y:=x
    else begin
      repeat
        y:=y-1
      until y<x;
      z:=z*7
      end
  end.
```



Basic Mechanism

- Generating TAC **as SLR(1) parsing**, bottom-up.
  - Calculating all the **synthesized attributes in one-pass**.
- **"Looking back"**: similar to back patching.
  - Calculating **inherited attributes**, such as "true"/"false".



- Generating output files in build/ folder, using Shell scripts.

```
ChenMac:pascal-compiler wasdns$ ls
LICENSE                 docs                    run_ir_demo.sh
README                  ir_tests                run_lexer.py
ROADMAP                 lexer_tests             run_lexer_ply.py
build_ir                parser_tests            run_parser_demo.sh
cleanup.sh              ply_frontend            src
ChenMac:pascal-compiler wasdns$ cd build_ir/
ChenMac:build_ir wasdns$ ls
action_and_goto.txt     input.txt               slr.txt
error.txt               ir                      top.txt
first_and_follow.txt    ir_code.txt
grammar.txt             output.txt
ChenMac:build_ir wasdns$
```

- Generating output files in build/ folder, using Shell scripts.

```
ChenMac:pascal-compiler wasdns$ ls
LICENSE                 docs                    run_ir_demo.sh
README                  ir_tests                run_lexer.py
ROADMAP                 lexer_tests             run_lexer_ply.py
build_ir                parser_tests            run_parser_demo.sh
cleanup.sh              ply_frontend            src
ChenMac:pascal-compiler wasdns$ cd build_ir/
ChenMac:build_ir wasdns$ ls
action_and_goto.txt     input.txt               slr.txt
error.txt               ir                      top.txt
first_and_follow.txt    ir_code.txt
grammar.txt             output.txt
ChenMac:build_ir wasdns$
```

What's does this script do?

1. Creating build_ir/ folder;
2. Running Lexer and generating **output files** in build/;
3. **Compiling** IR program, which is based on our previous program;
4. Reading and parsing the grammar and the output files;
5. Generating TAC in build_ir/ folder.

- Generating output files in build/ folder, using Shell scripts.

```
ChenMac:pascal-compiler wasdns$ ls
LICENSE                 docs                    run_ir_demo.sh
README                  ir_tests                run_lexer.py
ROADMAP                 lexer_tests             run_lexer_ply.py
build_ir                parser_tests            run_parser_demo.sh
cleanup.sh              ply_frontend            src
ChenMac:pascal-compiler wasdns$ cd build_ir/
ChenMac:build_ir wasdns$ ls
action_and_goto.txt     input.txt               slr.txt
error.txt               ir                      top.txt
first_and_follow.txt    ir_code.txt
grammar.txt             output.txt
ChenMac:build_ir wasdns$
```

Results of running our procedure

1. first_and_follow.txt: Print the first set and follow set
2. action_and_goto.txt: Print the action and goto table
3. output.txt: Print the runtime stack and applied actions
4. error.txt: Print runtime error log, while continuing analyzing.

- Generating output files in build/ folder, using Shell scripts.

```
ChenMac:pascal-compiler wasdns$ ls
LICENSE                 docs                    run_ir_demo.sh
README                  ir_tests                run_lexer.py
ROADMAP                 lexer_tests             run_lexer_ply.py
build_ir                parser_tests            run_parser_demo.sh
cleanup.sh              ply_frontend            src
ChenMac:pascal-compiler wasdns$ cd build_ir/
ChenMac:build_ir wasdns$ ls
action_and_goto.txt     input.txt               slr.txt
error.txt               ir                      top.txt
first_and_follow.txt    ir_code.txt
grammar.txt             output.txt
ChenMac:build_ir wasdns$
```

Results of running our procedure

1. first_and_follow.txt: Print the first set and follow set
2. action_and_goto.txt: Print the action and goto table
3. output.txt: Print the runtime stack and applied actions
4. error.txt: Print runtime error log, while continuing analyzing.
5. **ir_code.txt: The IR of input program, in terms of TAC.**

```
program (a,b);
  begin
    x:=19;
    if x<y then y:=x
    else begin
      repeat
        y:=y-1
      until y<x;
      z:=z*7
      end
end.
```

ir_code.txt

```
0: x = 19
1: if x < y goto 3
2: goto 5
3: y = x
4: goto 11
5: t1 = y - 1
6: y = t1
7: if y < x goto 9
8: goto 5
9: t2 = z * 7
10: z = t2
11:
```

Figure: Output IR, in terms of TAC

```
program (a,b);
  begin
    x:=19;
    if x<y then y:=x
    else begin
      repeat
        y:=y-1
      until y<x;
      z:=z*7
      end
end.
```

ir_code.txt

```
0:  x = 19
1:  if x < y goto 3
2:  goto 5
3:  y = x
4:  goto 11
5:  t1 = y - 1
6:  y = t1
7:  if y < x goto 9
8:  goto 5
9:  t2 = z * 7
10: z = t2
11:
```

Figure: Output IR, in terms of TAC

ir.cpp UNREGISTERED

```cpp
    cout<<(int)code[i].size()<< " "<<endl;

    fclose(stdout);
}

int main() {
    char gramarFile[] = "grammar.txt";
    char outputFile[] = "output.txt";
    char actionAndGotoFile[] = "action_and_goto.txt";
    char DFAFile[] = "slr.txt";
    char errorFile[] = "error.txt";
    char inputFile[] = "input.txt";
    char topFile[] = "top.txt";
    char codeFile[] = "ir_code.txt";


    init();
    getGrammar(gramarFile);
    getCanonical();

    calFirst();
    printFirst();
    calFollow();
    printFollow();

    printDFA(DFAFile);
    setActionAndGoto();
    printActionAndGoto(actionAndGotoFile);
    readInput(inputFile);
    readTop(topFile);
    solve(outputFile, errorFile);
    print(codeFile);
    return 0;
}
```

Line 1, Column 1 Spaces: 4 C++

Figure: Overview of Main Procedure

ir.cpp UNREGISTERED

```cpp
    cout<<(int)code[i].size()<< " "<<endl;

    fclose(stdout);
}

int main() {
    char gramarFile[] = "grammar.txt";
    char outputFile[] = "output.txt";
    char actionAndGotoFile[] = "action_and_goto.txt";
    char DFAFile[] = "slr.txt";
    char errorFile[] = "error.txt";
    char inputFile[] = "input.txt";
    char topFile[] = "top.txt";
    char codeFile[] = "ir_code.txt";


    init();
    getGrammar(gramarFile);
    getCanonical();

    calFirst();
    printFirst();
    calFollow();
    printFollow();

    printDFA(DFAFile);
    setActionAndGoto();
    printActionAndGoto(actionAndGotoFile);
    readInput(inputFile);
    readTop(topFile);
    solve(outputFile, errorFile);
    print(codeFile);
    return 0;
}
```

Total: ~800 LoC

Line 1, Column 1 Spaces: 4 C++

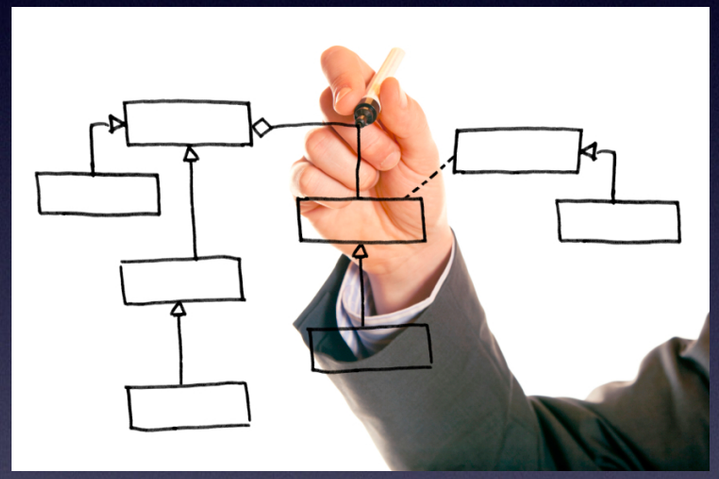
Figure: Overview of Main Procedure

Outline

- 1.Introduction
- 2.Journal of testing
- **3.Conclusion**

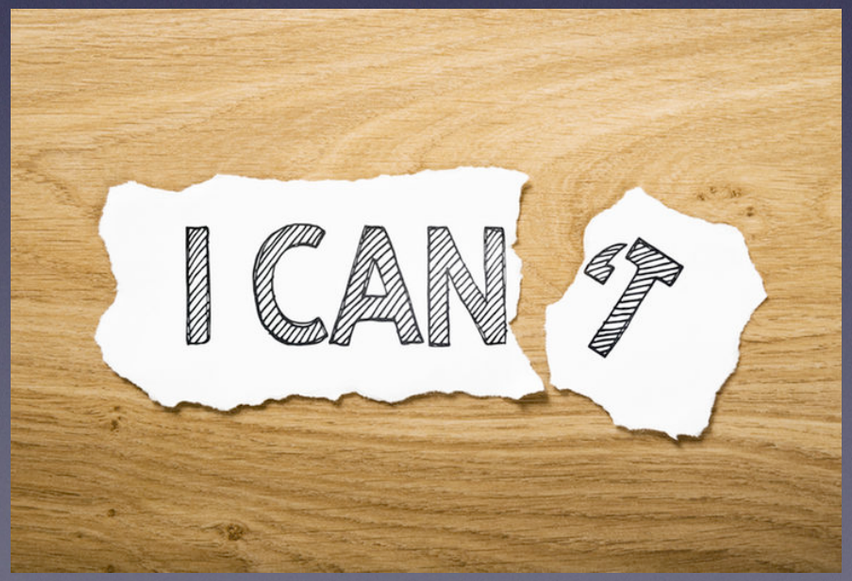
Conclusion

- Introduction of IR generating;
- An Example for Testing Our Program.



Experiences

- Learning lots of **Designing Principles** of a compiler.
- Appreciating our teacher, teammates and this subject.
- Memorable experience.



Thank you!

Group Members: 吴媛媛, 林诗尧, 陈翔