Crafting a Compiler with C (IV)

資科系
林偉川

Syntax analysis for the demo compiler

- The task of syntax analysis is to structure the input into an AST (abstract syntax tree)
- The grammar shown followed can be done by 2 boolean read routines : parse_operator() and parse_expression() for non-terminal operator and expression

expression → digit | ‘(‘ expression operator expression ‘)’

operator → ‘+’ | ‘*’

digit → ‘0’ | ‘1’ | … | ‘9’   3,(5+8), (2*((3*4)+9))

2

Parser header file for demo compiler

```
typedef int Operator;
typedef struct _expression {
  char type; // ‘D’(terminal) or ‘P’(non-terminal)
  int value; // for ‘D’ (terminal)
  Operator oper; // for ‘P’
  struct _expression *left, *right; // for ‘P’
} Expression;
typedef Expression AST_node;
extern int parse_program(AST_node **);
```

3

Syntax analysis for the demo compiler

- Recursive descent parsing is using a set of routines descend recursively to construct the parse tree
- To differentiate the node type Expression, each node contains a type attribute, set with a characteristic value: ‘D’ for digit and ‘P’ for a parenthesized expression

  expression → digit | (expression operator expression)
- The type attribute tells us how to interpreter the fields in the rest of the node

4

Driver for demo compiler

```
#include "parse.h"
#include "backend.h"
#include "error.h"
int main(void) {
  AST_node *icode;
  if (!parse_program(&icode)) // return 0 ➔error!!
     error("no top-level expression");
  Process(icode);   //back-end process
  return 0;
}
```

5

Parser environment for demo compiler

```
#include "lex.h"
#include "error.h"
#include "parse.h"
static Expression *new_expression(void) {
  return (Expression *)malloc(sizeof(Expression));
}
static void free_expression(Expression *expr) {
  free((void *)expr);
}
```

6

Parser environment for demo compiler

```
static int parse_operator(Operator *oper_p);
static int parse_expression(Expression **expr_p);
int parse_program(AST_node **icode_p) {
  Expression *expr;
  get_next_token();
  if (parse_expression(&expr)) {
    if (Token.class != EOF) error(“garbage after eop”);
   *icode_p=expr; return 1; // succeed checking an expression
  }
  return 0;
}
```


Parsing routines for demo compiler

```
static int parse_expression(Expression **expr_p) {
 Expression *expr=*expr_p=new_expression();
 if (Token.class==DIGIT) {// expression → digit
   expr->type=‘D’; expr->value=Token.repr-’0’;
   get_next_token(); return 1;
  }
```

Parsing routines for demo compiler

```
 if (Token.class == '(') {//expression → (expression operator expression)
   expr->type='P'; get_next_token();
   if (!parse_expression(&expr->left)) error("missing
  expression");
   if (!parse_operator(&expr->oper))  error("missing
  operator");
   if (!parse_expression(&expr->right)) error("missing
  expression");
   if (Token.class != ')') error("missing )");
   get_next_token(); return 1;
 }
 free_expression(expr); return 0;
}
```

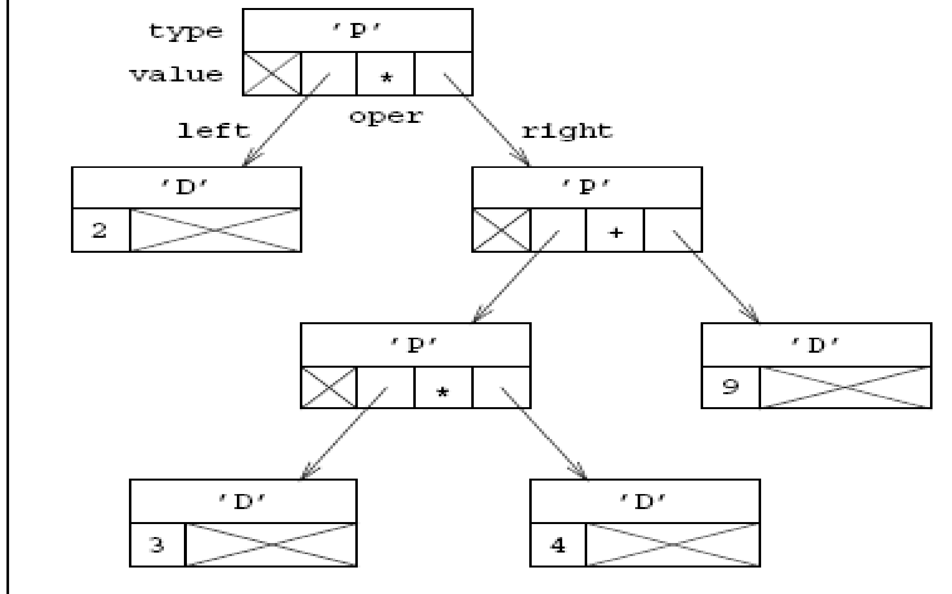
9

Parsing routines for demo compiler

```
static int parse_operator(Operator *oper) {
  if (Token.class == '+') {
    *oper='+'; get_next_token(); return 1;
  }
 if (Token.class == '*') {
    *oper='*'; get_next_token(); return 1;
 }
  return 0;
}
```

10

An AST for the expression (2*((3*4)+9))



Recursive Decent Parsing

- Apply the recursive parsing routine for each non-terminal production of CFG
- Within a parsing procedure, both non-terminals and terminals can be matched
- To match a non-terminal A, the corresponding procedure is named A

12

Recursive Decent Parsing

- To match a terminal t, a procedure match(t) is called and also call the scanner to get the next token
- The matched token is saved in a global variable named **current_token**
- If not matched, it is a syntax error and an error message is produced
- Some error correction or repair is done to restart the parser and continue compilation

13

Extended CFG for Micro

Start: <system goal>→ <program> SCANEOF

1.	<program>	→	**begin** <statement list> **end**
2.	<statement list>	→	<statement> {<statement>}
3.	<statement>	→	ID := <expression> ;
4.	<statement>	→	**read** (<id list>) ;
5.	<statement>	→	**write** (<expr list>) ;
6.	<id list>	→	ID {, ID}
7.	<expr list>	→	<expression> {, <expression>}
8.	<expression>	→	<primary> {<add op> <primary>}
9.	<primary>	→	(<expression>)
10.	<primary>	→	ID
11.	<primary>	→	INTLITERAL
12.	<add op>	→	PLUSOP
13.	<add op>	→	MINUSOP
14.	<system goal>	→	<program> SCANEOF

Figure 2.4 Extended CFG Defining Micro

Recursive parsing example (I)

<system goal>→ <program> SCANEOF

```
void system_goal(void) {  // rule 14
  program(); match(SCANEOF);
}
```

15

Recursive parsing example (II)

<program>→ BEGIN <statement list> END

```
void program(void) { // rule 1
 match(BEGIN); statement_list(); match(END);
}
```

16

Recursive parsing example (III)

<statement list> → <statement> {<statement>}

```
void statement_list(void) { // rule 2
  statement();
  while (true) {
     switch (next_token) {
     case ID:
     case READ:
     case WRITE:
        statement(); break;
      default: return;
     }
  }
}
```

17

Recursive parsing example (IV)

<statement> → ID:=<expression> | read(<id list>) | write(<expr list>)

```
void statement(void) { // rule 3,4,5
  token tok=next_token();
  switch (tok) {
     case ID: // rule 3
       match(ID); match(ASSIGNOP);
       expression(); match(SEMICOLON); break;
     case READ: // rule 4
       match(READ); match(LPAREN); id_list();
       match(RPAREN); match(SEMICOLON); break;
```

18

Recursive parsing example (IV)

```
    case WRITE: // rule 5
      match(READ); match(LPAREN); expr_list();
      match(RPAREN); match(SEMICOLON); break;
    default: syntax_error(tok); break;
    }
  }
}
```


Recursive parsing example (V)

<id list> → ID {, ID }

```
void id_list(void) { // rule 6
  match(ID);
  while (next_token() == COMMA) {
    match(COMMA); match(ID);
  }
}
```

Recursive parsing example (VI)

<expr list> → <expression> {, <expression> }

```
void expr_list(void) { // rule 7
  expression();
  while (next_token() == COMMA) {
    match(COMMA); expression();
  }
}
```


Recursive parsing example (VII)

<expression> → <primary> { <add op><primary> }

```
void expression(void) { // rule 8
  primary();
  for (t=next_token(); t == PLUSOP || t == MINUSOP;
                        t=next_token()) {
    add_op(); primary();
  }
}
```

Recursive parsing example (VIII)

<add op> → PLUSOP | MINUS

```
void add_op(void) { // rule 12,13
  token tok=next_token();
  if (tok == PLUSOP || tok == MINUSOP) match(tok);
  else syntax_error(tok);
}
```

23

Recursive parsing example (IX)

<primary> → (<expression>) | ID | INTLITERAL

```
void primary(void) { // rule 9,10,11
  token tok=next_token();
  switch (tok) {
  case LPAREN:
    match(LPAREN); expression(); match(RPAREN); break;
  case ID: match(ID); break;
  case INTLITERAL: match(INTLITERAL); break;
  default: syntax_error(tok); break;
  }
}
```

24

Homework

- Combine scanner and RDP to trace the example as followed:

```
begin
read (c1,c3);
c2:=c1+(5-c3);
write (c2);
end$
```

25

Translating Micro

- Decide what machine code should be generated
- The format of the machine code ➔ assembly code, object module …
- Three-address assembly code is selected as the machine instruction such as:
  OP a, b, c
- OP is the op-code, a and b is the operands of the operator, and c is the location where the result is stored

26

Translating Micro

- The target code is for a simple virtual machine and could be used to drive an interpreter
- The target code is like the quadruples ➔ a common used intermediate representation
- Use temporarily allocated storage locations, known as temporaries, to hold intermediate results of a computation ➔ internal variable Temp&N, N is the index of the temporary, starting at 1

27

Action symbols

- A translation is done by semantic routines called by the parser
- Action symbols can be added to a grammar to specify when semantic processing should take place
- Action symbol denoted by #name in the examples that can be placed anywhere in the RHS of a production (<statement> → <ident>:=<expression> #assign)
- Action symbol #add corresponds to a semantic routine named add()

28

Action symbols

- When parsing procedures are created, calls to semantic routines or in-line code segments to do semantic processing are inserted in the positions designated by action symbols
- If a grammar containing action symbols is given as input to a parser generator, the generator must include appropriate information in the tables it produces to trigger calls to semantic routine at corresponding times during the parsing process

29

Action symbols

- Action symbols are not actually part of the syntax the CFG specifies
- It serves to comment a CFG, indicating when semantic actions need to execute
- When a CFG is used as the input to a parser generator, the action symbols tell the generator when the corresponding semantic routines must be called

30

Semantic information

- Design of semantic routines is the specification of the data on which they operate and the information they produce ➔ semantic record
- The semantic record for a terminal contains the token's token_buffer or some value derived from it. Semantic record is produced by a semantic routine called after parser successfully matches a token against an expected terminal symbol

31

Semantic information

- The semantic record for a non-terminal is created by a semantic routine that has access to information about any of the symbols on the RHS of the production
- If some of these symbols are non-terminal, their corresponding semantic records come from semantic routine calls specified within their own production (add() is called by semantic routine)

  <expression> → <primary> + <primary> #add

32

Semantic records for grammar symbol

<expression> → <primary> + <primary> #add

- A semantic record will generated for each of the <primary>'s in the RHS of the production
- These semantic records record data about each of the operands
- When add() is called, it must be given these semantic records as parameters. It uses them to generate the appropriate code, then produces a new semantic record corresponding to <expression> that records necessary information about the expression just processed

33

Semantic records for grammar symbol

- Records containing the semantic information for non-terminal symbols may be returned as result parameters by corresponding parsing routines. When a table-driven parser is used, an explicit semantic stack is necessary to store semantic records between semantic routine calls
- All other symbols need no associated semantic information and have null semantic records
- Can add fields to a semantic record if we decide extra data is needed
- In deciding on semantic records, we are deciding what parameters a semantic routine will have

34

Semantic records for grammar symbol

```
#define MAXIDLEN 33
typedef char[MAXIDLEN] string;
typedef struct operator {
  enum op { PLUS, MINUS } operator;
} op_rec;
enum expr { IDEXPR, LITERALEXPR, TEMPEXPR };
typedef struct expression {
  enum expr kind;
  union {
    string name; // for IDEXPR, TEMPEXPR
    int val;  // for LITERALEXPR
   } ;
} expr_rec;
```

35

Extended CFG for Micro

Start: <system goal>→ <program> SCANEOF

1.	<program>	→	**begin** <statement list> **end**
2.	<statement list>	→	<statement> {<statement>}
3.	<statement>	→	ID := <expression> ;
4.	<statement>	→	**read** (<id list>) ;
5.	<statement>	→	**write** (<expr list>) ;
6.	<id list>	→	ID {, ID}
7.	<expr list>	→	<expression> {, <expression>}
8.	<expression>	→	<primary> {<add op> <primary>}
9.	<primary>	→	(<expression>)
10.	<primary>	→	ID
11.	<primary>	→	INTLITERAL
12.	<add op>	→	PLUSOP
13.	<add op>	→	MINUSOP
14.	<system goal>	→	<program> SCANEOF

Figure 2.4 Extended CFG Defining Micro

Grammar with 9 action symbols

<program>→ #start begin <statement list> end
<statement list> → <statement> {<statement>}
<statement> → <ident>:=<expression> #assign
<statement> →read(<id list>);
<statement> → write(<expr list>);
<id list> → <ident> #read_id {, <ident> #read_id }
<expr list> → <expression> #write_id {, <expression> #write_id }
<expression> → <primary> { <add op><primary> #gen_infix }
<primary> → (<expression>)
<primary> → <ident>
<primary> → INTLITERAL #process_literal
<add op> → PLUSOP #process_op
<add op> → MINUS #process_op
<ident> → ID #process_id
<system goal>→ <program> SCANEOF #finish

37

Action symbols explanation

<ident> → ID #process_id

- This production is useful because ID appeared in several different context in the grammar, and need to call process_id() immediately after the parser matches any occurrence of an ID (to access the characters in the token_buffer and build an appropriate semantic record). Substitute <ident> for ID everywhere in the grammar is a simple way to ensure that process_id() is always called

38

Action symbols explanation

- generate() will take 4 string arguments corresponding to the operation code, 2 operands, and the result field.
- extract() will take a semantic record and return a string corresponding to the semantic information it contains. The string may be an identifier, an op code, a literal, etc. The extracted information is fed to generate() to create a complete instruction
- Allow the assembler to allocate storage for variables. The only information about an identifier is whether it is already in the symbol table, so the compiler will know if it must generate an instruction that will cause space allocation (Declare A, Integer)

39

Action symbols explanation

- The specification of the proposed symbol tables are followed:

```
extern int lookup(string s); //Is s in the symbol table?
Extern void enter(string s); // put s into symbol table
void check_id(string s) {
  if (!lookup(s)) {
     enter(s); generate(“Declare”, s, ”Integer”, ””); }
}
```

- check_id() will declare a variable by entering it in the symbol table and then generating an assembler directive to reserve space for it.

40

Action symbols explanation

- Need a routine to allocate temporaries. The names should be Temp&1, Temp&2 , etc.

```
char *get_temp(void) {
  static int max_temp=0;
  static char tempname[MAXIDLEN];
  max_temp++;
  sprintf(tempname, ”Temp%d”, max_temp);
  check_id(tempname); return tempname;
}
```

41

Action symbols explanation

- Other auxiliary routines necessary to define the semantic routines corresponding to grammar's action symbols as shown followed:

```
void start(void) { /* semantic initialization */}
void finish(void) {
  generate(“Halt”, ””, ””,””);
}
void assign(expr_rec target, expr_rec source) {
  generate(“Store”, extract(source), target.name, “”);
}
```

42

Action symbols explanation

- Other processing identifier is shown followed:

```
op_rec process_op(void) {
  op_rec o;
  if (current_token == PLUSOP) o.operator=PLUS;
  else o.operator=MINUS;
  return o;
}
expr_rec gen_infix(expr_rec e1, op_rec op, expr_rec e2)
{
  expr_rec e_r; e_r.kind=TEMPEXPR;
 strcpy(e_r.name, get_temp());
 generate(extract(op), extract(e1), extract(e2), e_r.name);
 }
```

43

Action symbols explanation

- Other processing read/write instruction is shown followed:

```
void read_id(expr_rec in_var) {
  generate("Read", in_var.name, "Integer", "");
}
expr_rec process_id(void) {
  expr_rec t;
  check_id(token_buffer); t.kind=IDEXPR;
  strcpy(t.name, token_buffer); return t;
}
void write_expr(expr_rec out_expr) {
  generate("Write", extract(out_expr), "Integer", "");
}
```

44

Action symbols explanation

- Other processing expression is shown as followed:

```
expr_rec process_literal(void) {
  expr_rec t;  t.kind=LITERALEXPR;
  (void) sscanf(token_buffer, "%d", &t.val);
  return t;
}
```

45

Recursive parsing example (VII)

```
<expression> → <primary> { <add op><primary> }
void expression(void) { // rule 8
  primary();
  for (t=next_token(); t == PLUSOP || t == MINUSOP;
                            t=next_token()) {
    add_op(); primary();
  }
}
```

46

Parsing procedure including semantic processing

- One example of a production is shown followed:

```
void expression(expr_rec *result) {
  expr_rec left, right;  op_rec op;
  primary(&left);
  while (next_token()==PLUSOP ||
           next_token()==MINUSOP) {
     add_op(&op); primary(&right);
     left=gen_infix(left, op, right);
  }
  *result=left;
}
```

<expression> → <primary> { <add op><primary> #gen_infix }

47

Code generation for an expression

begin A:=BB-314+A; end SCANEOF

The code generation results are shown as followed:

```
Declare A, Integer
Declare BB, Integer
Declare Temp&1, Integer
Sub BB, 314, Temp&1
Declare Temp&2, Integer
Add Temp&1, A, Temp&2
Store Temp&2, A
Halt
```

48

Code generation procedure

begin A:=BB-314+A; end SCANEOF

1. Call system_goal(); begin A:=BB-314+A; end
2. Call program(); begin A:=BB-314+A; end
3. Semantic action: start();
4. Match(begin); A:=BB-314+A; end SCANEOF
5. Call statement_list();
6. Call statement();
7. Call ident();
8. Match(ID);

49

Code generation procedure

9. Semantic action: process_id(); ➔Declare A, Integer :=BB-314+A; end SCANEOF
10. Match(ASSIGNOP);
11. Call expression(); BB-314+A; end SCANEOF
12. Call primary();
13. Call ident();
14. Match(ID);
15. Semantic action: process_id() ➔ Declare BB, Integer -314+A; end SCANEOF

50

Code generation procedure

16. Call add_op();
17. Match(MINUSOP);
18. Semantic action: process_op(); 314+A; end SCANEOF
19. Call primary();
20. Match(INTLITERAL);
21. Semantic action: process_literal(); ➔ Declare Temp&1, Integer +A; end SCANEOF
22. Semantic action: gen_infix (); ➔ Sub BB, 314, Temp&1
23. Call add_op();

51

Code generation procedure

24. Match(PLUSOP);
25. Semantic action: process_op(); A; end SCANEOF
26. Call Primary();
27. Call ident();
28. Match(ID);
29. Semantic action: process_id();

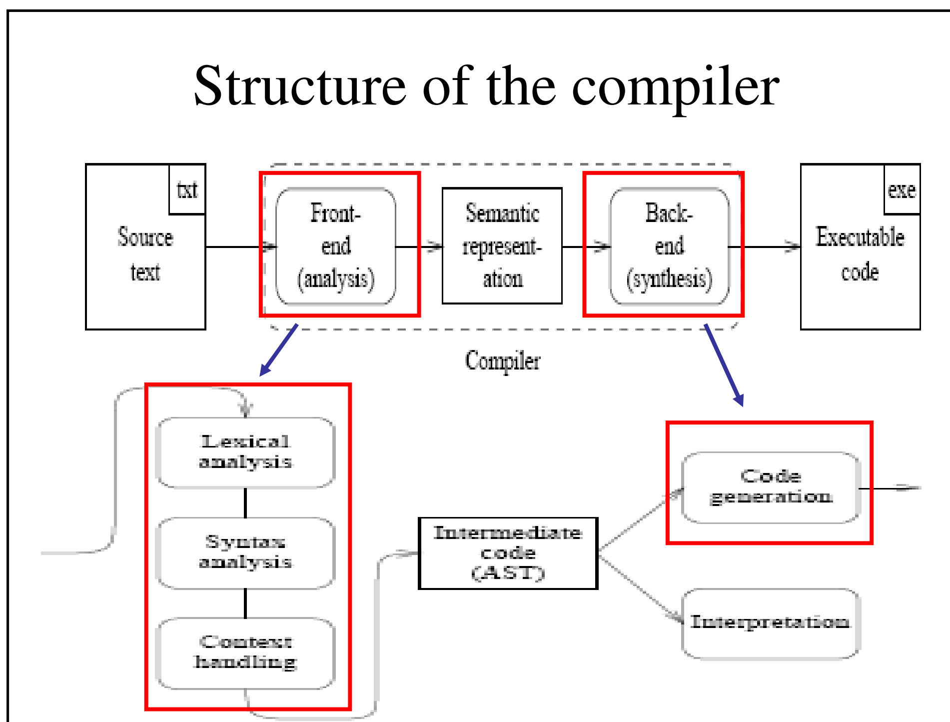
52

Code generation procedure

30. Semantic action: gen_infix(); ➔Declare Temp&2, Integer
    Add Temp&1, A, Temp&2   ; end SCANEOF
31. Semantic action: assign() ➔ Store Temp&2, A
32. Match(SEMICOLON);   end SCANEOF
33. Match(END);   SCANEOF
34. Match(SCANEOF);
35. Semantic action: finish(); ➔ Halt

53

Structure of the compiler

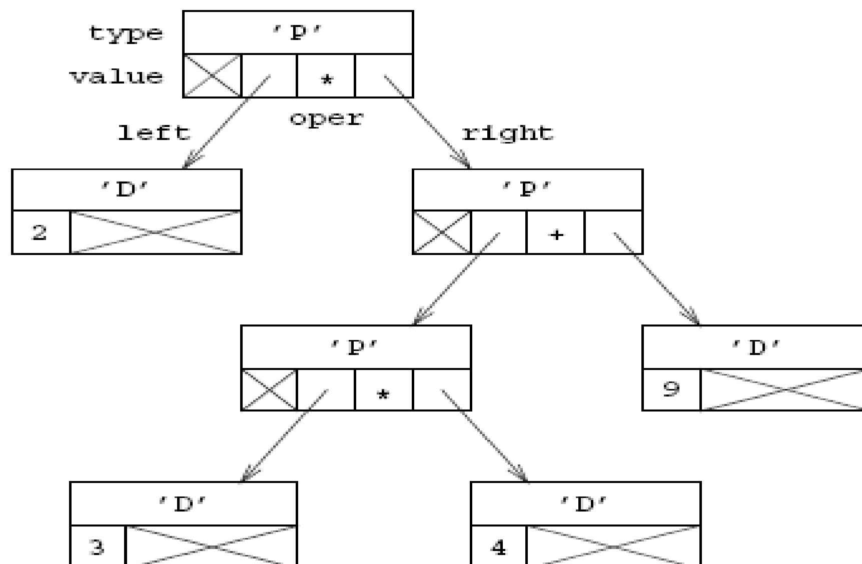


Driver for demo compiler

```
#include "parse.h"
#include "backend.h"
#include "error.h"
int main(void) {
  AST_node *icode;
  if (!parse_program(&icode)) // return 0 →error!!
     error("no top-level expression");
  Process(icode);   //back-end process
  return 0;
}
```

55

An AST for the expression (2*((3*4)+9))



Code generation back-end for demo compiler

```
void Process(AST_node *icode) {
 code_gen_expression(icode); printf("PRINT\n");
}
```

(2*((3*4)+9))

```
PUSH 2
PUSH 3
PUSH 4
MULT
PUSH 9
ADD
MULT
PRINT
```

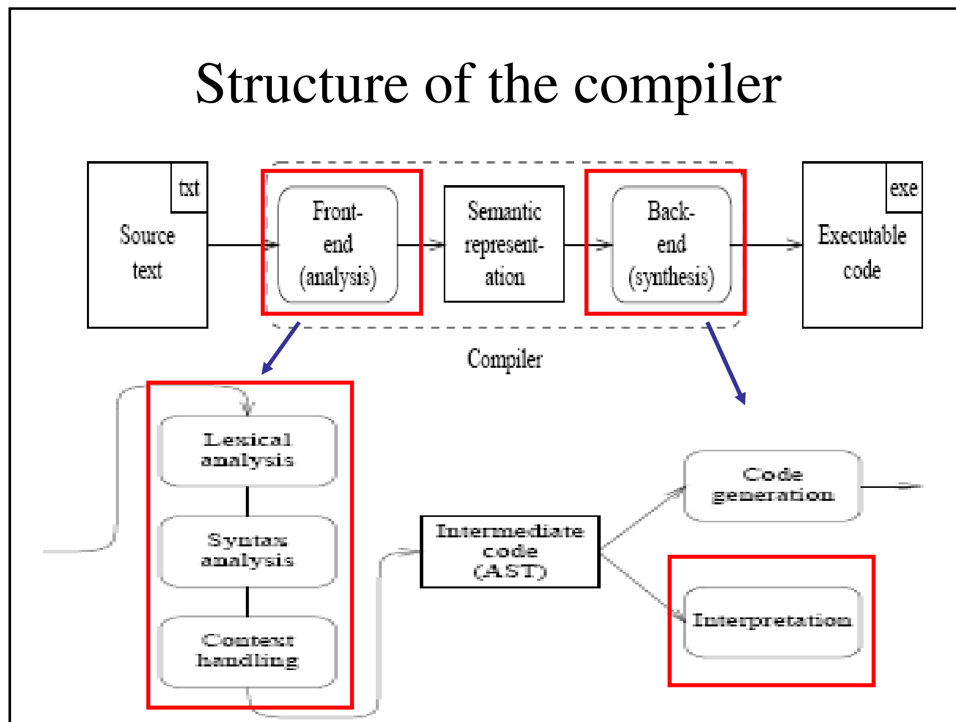
PUSH n	Pushes the integer n onto the stack
MULT	Replace the topmost two elements by their product
ADD	Replace the topmost two elements by their sum
PRINT	Pops the top element and print its value

57

Code generation back-end for demo compiler

```
#include "parser.h"
#include "backend.h"
static void code_gen_expression(Expression *expr) {
 switch (expr->type) {
 case 'D' : printf("push %d\n",expr->value); break;
 case 'P' :
  code_gen_expression(expr->left);
  code_gen_expression(expr->right);
   switch (oper) {
   case '+' : printf("ADD\n"); break;
   case '*' : printf("MULT\n"); break;
   }
  break;
}}
```

58



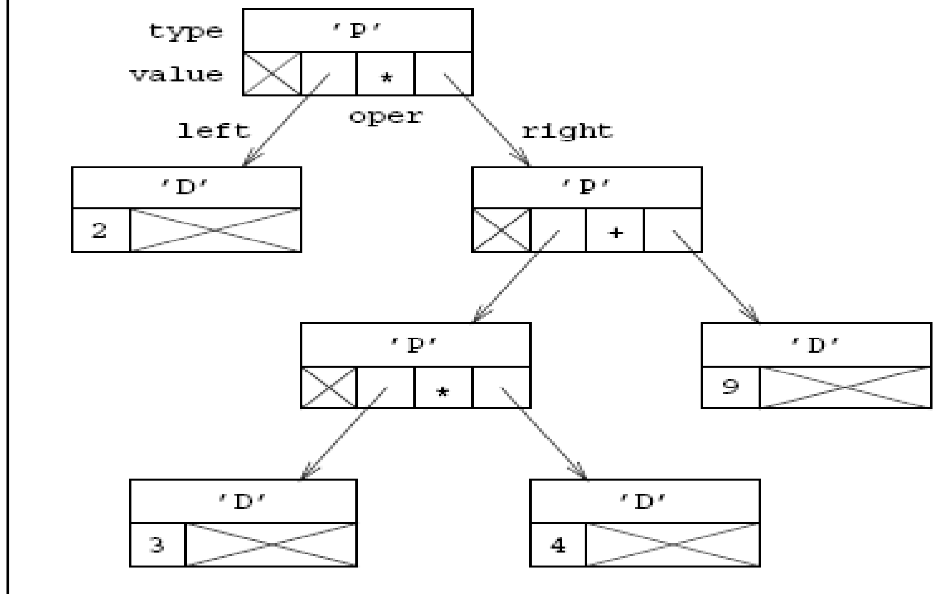
Code generation back-end for demo compiler

```
void Process(AST_node *icode) {
  printf("%d\n", Interpreter_expression(icode));
}
```

(2*((3*4)+9))

60

An AST for the expression (2*((3*4)+9))



Code generation back-end for demo compiler

```
#include "parser.h"
#include "backend.h"
static int Interpreter_expression(Expression *expr) {
 switch (expr->type) {
 case 'D' :  return expr->value;
 case 'P' : {
   int e_left=Interpreter_expression(expr->left);
   int e_right=Interpreter_expression(expr->right);
    switch (expr->oper) {
    case '+' : return e_left+e_right;
    case '*' : return e_left*e_right;
    }}
   break;
}}
```

(2*((3*4)+9))

2 , right

Left, left, 3

4

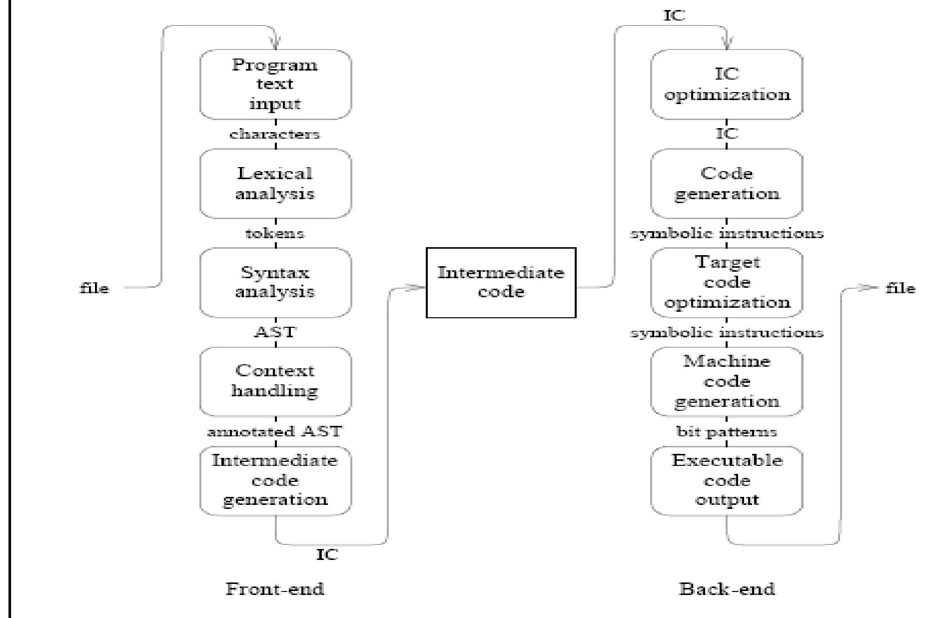
* → 12

9

+ → 21

* → 42

Structure of a compiler



Structure explanation

- The program text input module finds the program text file, reads it and turns it into a stream of characters, allowing for different kinds of newlines, escape codes, etc. This function may cooperate with the lexical analyzer
- The lexical analysis module isolates tokens in the input stream and determines their class and representation. It may do some limited interpretation on some of the tokens (identifier or keywords)

64

Structure explanation

- The syntax analysis module converts the stream of tokens in an AST. Some syntax analyzers consist of 2 modules. The first one reads the token stream and calls a function from the second module for each syntax construct it recognizes; the second function construct the nodes of AST and link them (can replace the AST generation module to obtain a different AST from the same syntax analyzer or can replace the syntax analyzer to obtain the same type of AST from a different language)

65

Structure explanation

- The context handling module collects context information from various places in the program, and annotates nodes with the result (connect goto statement to their label, relating type information from declarations to expressions, deciding the routine calls are local or remote). These annotations are used for performing context checks or are passed on to subsequent modules to aid in code generation

66

Structure explanation

- The intermediate code generation module translates language-specific constructs in the AST into more general constructs; these general construct constitute the intermediate code (IC). Deciding what is a language-specific and what a more general construct is up to the compiler designer. The level of the intermediate code is that it should be reasonably to generate machine code from it for various machines

67

Structure explanation

- The intermediate code optimization module performs preprocessing on the intermediate code, with the intention of improving the effectiveness of the code generation module (in-lining which chosen calls to some routines are replaced by the bodies of those routines, while at the same time substituting the parameters)

68

Structure explanation

- Code generation module rewrites the AST into a linear list of target machine instructions in symbolic form. To this end, it selects instructions for segments of the AST, allocates registers to hold data and arranges the instructions in proper order

69

Structure explanation

- Target code optimization module considers the list of symbolic machine instructions and to optimize it by replacing sequences of machine instructions by faster or shorter sequences. It uses target-machine-specific properties.

70

Structure explanation

- The precise boundaries between intermediate code optimization, code generation, and target code optimization are floating: if code generation is good, little target code optimization may be needed

71

Structure explanation

- Machine code generation module converts the symbolic machine instructions into the corresponding bit pattern. It determines machine address of program code and data and produces tables of constants and relocation tables
- Executable code output module combines the encoded machine instructions, the constant tables, the relocation tables, and the headers, trailers, and other material required by the OS into an executable code file

72