Chapter 2 A Simple Compiler

Outlines

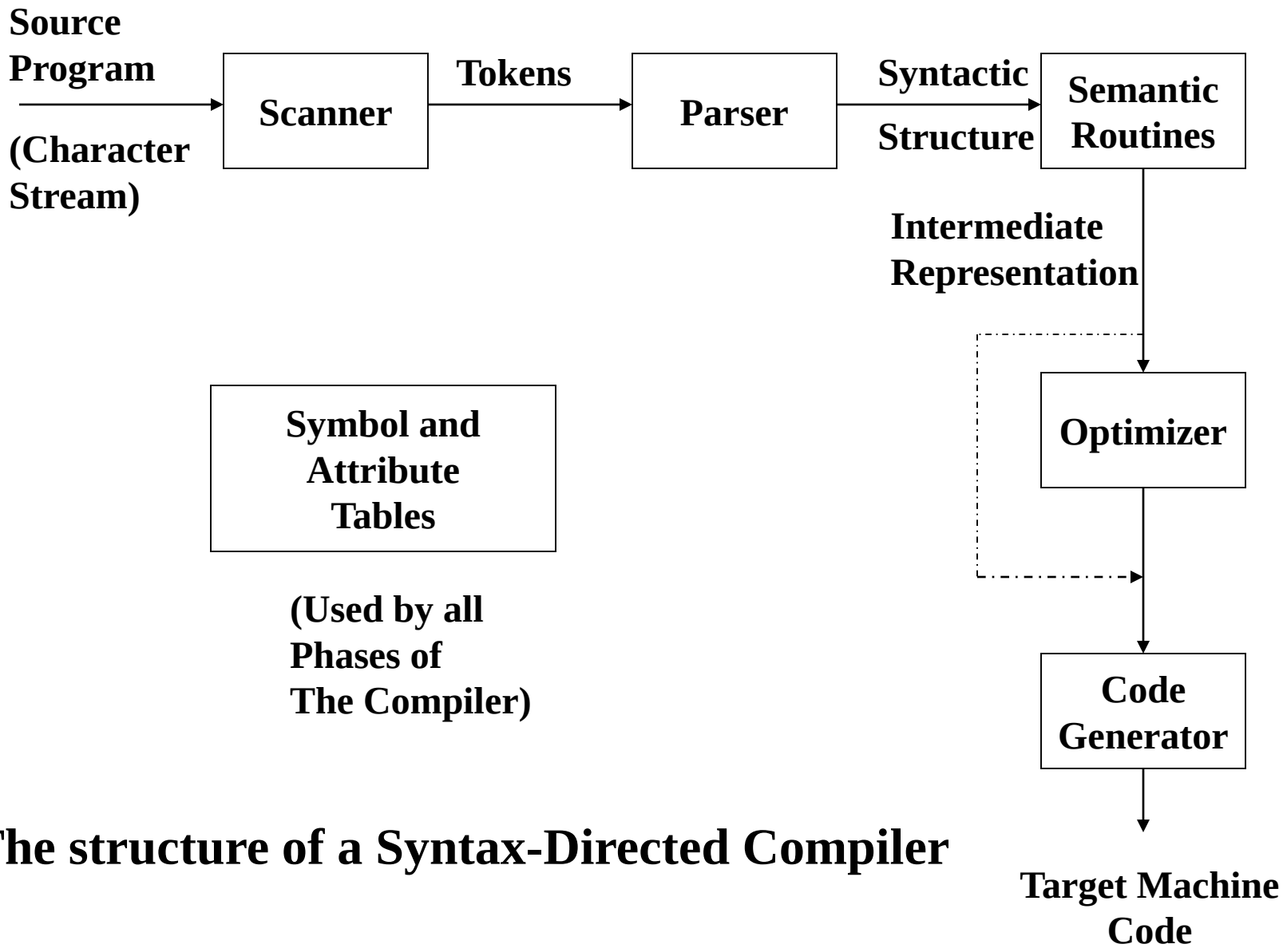
- 2.1 The Structure of a Micro Compiler
- 2.2 A Micro Scanner
- 2.3 The Syntax of Micro
- 2.4 Recursive Descent Parsing
- 2.5 Translating Micro

Micro

- Micro: a very simple language
  - Only integers
  - No declarations
  - Variables consist of A..Z, 0..9, and at most 32 characters long.
  - Comments begin with -- and end with end-of-line.
  - Three kinds of stmts:
    - assignments, e.g., a := b + c
    - read(list of ids), e.g., read(a, b)
    - write(list of exps), e.g., write(a+b)
  - **Begin**, **end**, **read**, and **write** are reserved words.
  - Tokens may not extend to the following line.

The Structure of a Micro compiler

- One-pass type, no explicit intermediate representations used
  - See P. 9, Fig. 1.3
- The interface
  - Parser is the main routine.
  - Parser calls scanner to get the next token.
  - Parser calls semantic routines at appropriate times.
  - Semantic routines produce output in assembly language.
  - A simple symbol table is used by the semantic routines.



The structure of a Syntax-Directed Compiler

A Micro Scanner

- The Micro Scanner will be a function of no arguments that returns ***token*** values
  - There are 14 tokens.

```
typedef enum token_types {
        BEGIN, END, READ, WRITE, ID, INTLITERAL,
        LPAREN, RPAREN, SEMICOLON, COMMA, ASSIGNOP,
        PLUOP, MINUSOP, SCANEOF
} token;

Extern token scanner(void);
```

A Micro Scanner (Cont'd)

- The scanner returns the longest string that constitutes a token, e.g., in

  abcdef

  ab, abc, abcdef are all valid tokens.

  The scanner will return the

  longest one (i.e., abcdef).

```
#include <stdio.h>
#include <ctype.h>

int in_char, c;

while ((in_char = getchar()) != EOF) {
    if (isspace(in_char))
        continue;     /* do nothing */
    else if (isalpha(in_char)) {
        /*
         * ID ::= LETTER | ID LETTER
         *               | ID DIGIT
         *               | ID UNDERSCORE
         */

        for (c = getchar(); isalnum(c) || c == '_';
                    c = getchar())
            ;
        ungetc(c, stdin);

        return ID;
    } else if (isdigit(in_char)) {
        /*
         * INTLITERAL ::= DIGIT |
         *                INTLITERAL DIGIT
         */
        while (isdigit((c = getchar())))
            ;
        ungetc(c, stdin);

        return INTLITERAL;
    } else
        lexical_error(in_char);
}
```

Figure 2.1 Scanner Loop to Recognize Identifiers and Integer Literals

Continue: skip one iteration

getchar() , isspace(), isalpha(), isalnum(), isdigital()

ungetc(): push back one character

```
#include <stdio.h>
#include <ctype.h>

int in_char, c;

while ((in_char = getchar()) != EOF) {
    if (isspace(in_char))
        /* do nothing */
        continue;
    else if (isalpha(in_char))
        /* code to recognize identifiers goes here */
    else if (isdigit(in_char))
        /* code to recognize int literals goes here */
    else if (in_char == '(')
        return LPAREN;
    else if (in_char == ')')
        return RPAREN;
    else if (in_char == ';')
        return SEMICOLON;
    else if (in_char == ',')
        return COMMA;
    else if (in_char == '+')
        return PLUSOP;
    else if (in_char == ':') {
        /* looking for ":=" */
        c = getchar();
        if (c == '=')
            return ASSIGNOP;
        else {
            ungetc(c, stdin);
            lexical_error(in_char);
        }
    } else if (in_char == '-') {
        /* looking for --, comment start */
        c = getchar();
        if (c == '-') {
            while ((in_char = getchar()) != '\n');
        } else {
            ungetc(c, stdin);
            return MINUSOP;
        }
    } else
        lexical_error(in_char);
}
```

Figure 2.2 Scanner Loop with New Code to Recognize Operators, Comments, and Delimiters

A Micro Scanner (Cont'd)

- How to handle RESERVED words?
  - Reserved words are similar to identifiers.
- Two approaches:
  - Use a separate table of reserved words
  - Put all reserved words into symbol table initially.

A Micro Scanner (Cont'd)

- Provision for saving the characters of a token as they are scanned
  - token_buffer, buffer_char(), clear_buffer(), check_reserved()
- Handle end of file
  - feof(stdin)

Complete Scanner Function for Micro

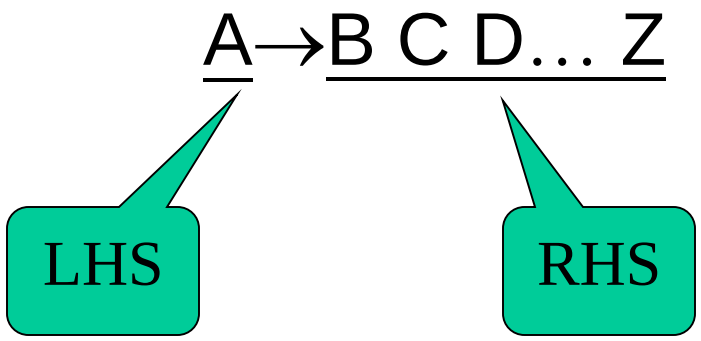
```
#include <stdio.h>
/* character classification macros */
#include <ctype.h>

extern char token_buffer[];

token scanner(void)
{
    int in_char, c;

    clear_buffer();
    if (feof(stdin))
        return SCANEOF;

    while ((in_char = getchar()) != EOF) {
        if (isspace(in_char))
            continue;    /* do nothing */
        else if (isalpha(in_char)) {
            /*
             * ID ::= LETTER | ID LETTER
             *               | ID DIGIT
             *               | ID UNDERSCORE
             */
            buffer_char(in_char);
            for (c = getchar(); isalnum(c) || c == '_';
                    c = getchar())
                buffer_char(c);
```

```
            ungetc(c, stdin);
            return check_reserved();
        } else if (isdigit(in_char)) {
            /*
             * INTLITERAL ::= DIGIT |
             *                INTLITERAL DIGIT
             */
            buffer_char(in_char);
            for (c = getchar(); isdigit(c);
                    c = getchar())
                buffer_char(c);
            ungetc(c, stdin);
            return INTLITERAL;
        } else if (in_char == '(')
            return LPAREN;
        else if (in_char == ')')
            return RPAREN;
        else if (in_char == ';')
            return SEMICOLON;
        else if (in_char == ',')
            return COMMA;
        else if (in_char == '+')
            return PLUSOP;
        else if (in_char == ':') {
            /* looking for ":=" */
            c = getchar();
            if (c == '=')
                return ASSIGNOP;
            else {
                ungetc(c, stdin);
                lexical_error(in_char);
            }
        } else if (in_char == '-') {
            /* is it --, comment start */
            c = getchar();
            if (c == '-') {
                do
                    in_char = getchar();
                while (in_char != '\n');
            } else {
                ungetc(c, stdin);
                return MINUSOP;
            }
        } else
            lexical_error(in_char);
    }
}
```

Figure 2.3 Complete Scanner Function for Micro

The Syntax of Micro

- Micro's syntax is defined by a context-free grammar (CFG)
  - CFG is also called BNF (Backus-Naur Form) grammar
- CFG consists of a set of production rules,



LHS must be a single **nonterminal**

RHS consists 0 or more **terminals** or **nonterminals**

The Syntax of Micro (Cont'd)

- Two kinds of symbols
  - Nonterminals
    - Delimited by < and >
    - Represent syntactic structures
  - Terminals
    - Represent tokens
- E.g.

  <program> → **begin** <statement list> **end**
- Start or goal symbol
- λ : empty or null string

The Syntax of Micro (Cont'd)

- E.g.

<statement list> $\rightarrow$ <statement><statement tail>

<statement tail> $\rightarrow \lambda$

<statement tail> $\rightarrow$ <statement><statement tail>

The Syntax of Micro (Cont'd)

- Extended BNF: some abbreviations

1. optional: [ ] 0 or 1

   <stmt> → if <exp> then <stmt>

   <stmt> → if <exp> then <stmt> else <stmt>

   can be written as

   <stmt> → if <exp> then <stmt> [else <stmt>]

2. repetition: { } 0 or more

   <stmt list> → <stmt> <tail>

   <tail> → λ

   <tail> → <stmt> <tail>

   can be written as

   <stmt list> → <stmt> { <stmt> }

The Syntax of Micro (Cont'd)

- Extended BNF: some abbreviations

  3. alternative: | or

     <stmt> → <assign>

     <stmt> → <if stmt>

     can be written as

     <stmt> → <assign> | <if stmt>

- Extended BNF == BNF
  - Either can be transformed to the other.
  - Extended BNF is more compact and readable

The Syntax of Micro (Cont'd)

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

- The **derivation** of

  begin ID:= ID + (INTLITERAL – ID); end

<program>	
begin <statement list> **end**	(Apply rule 1)
begin <statement> {<statement>} **end**	(Apply rule 2)
begin <statement> **end**	(Choose 0 repetitions)
begin ID := <expression> ; **end**	(Apply rule 3)
begin ID := <primary> {<add op> <primary>} ; **end**	(Apply rule 8)
begin ID := <primary> <add op> <primary> ; **end**	(Choose 1 repetition)
begin ID := <primary> + <primary> ; **end**	(Apply rule 12)
begin ID := ID + <primary> ; **end**	(Apply rule 10)
begin ID := ID + (<expression>) ; **end**	(Apply rule 9)
begin ID := ID + (<primary> {<add op> <primary>}) ; **end**	(Apply rule 8)
begin ID := ID + (<primary> <add op> <primary>) ; **end**	(Choose 1 repetition)
begin ID := ID + (<primary> – <primary>) ; **end**	(Apply rule 13)
begin ID := ID + (INTLITERAL – <primary>) ; **end**	(Apply rule 11)
begin ID := ID + (INTLITERAL – ID) ; **end**	(Apply rule 10)

The Syntax of Micro (Cont'd)

- A CFG defines a *language*, which is a set of sequences of **tokens**
- Syntax errors & semantic errors

  A:='X'+True;

The Syntax of Micro (Cont'd)

- Associativity

  A-B-C

- Operator precedence

  A+B*C

- A grammar fragment defines such a precedence relationship

<expression> → <factor> {<add op> <factor>}
<factor> → <primary> {<mult op> <primary>}
<primary> → (<expression>)
<primary> → ID
<primary> → INTLITERAL

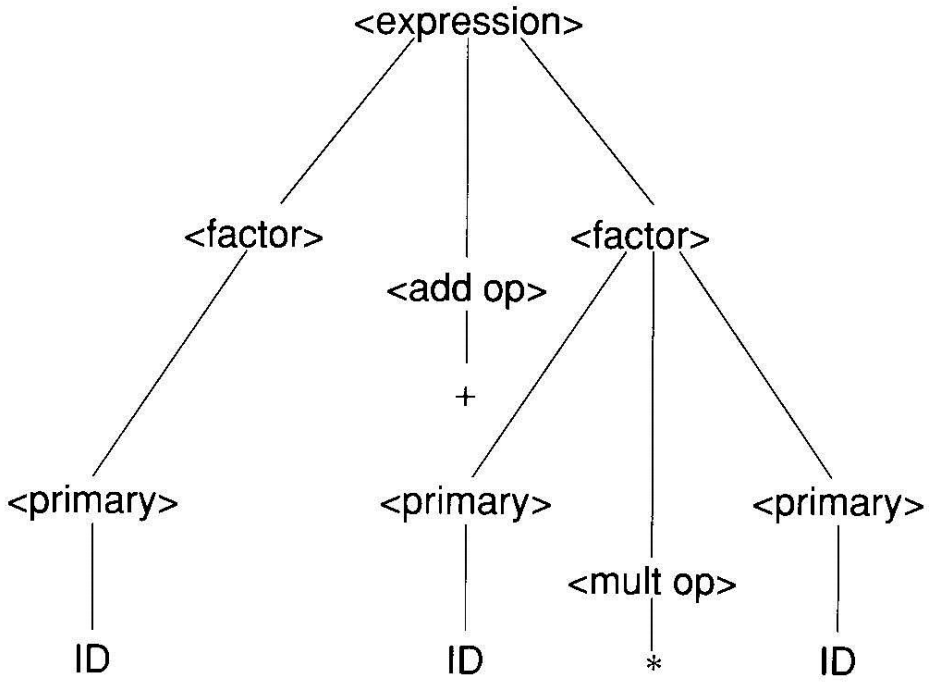


Figure 2.5 Derivation Tree for A+B*C

- With parentheses, the desired grouping can be forced

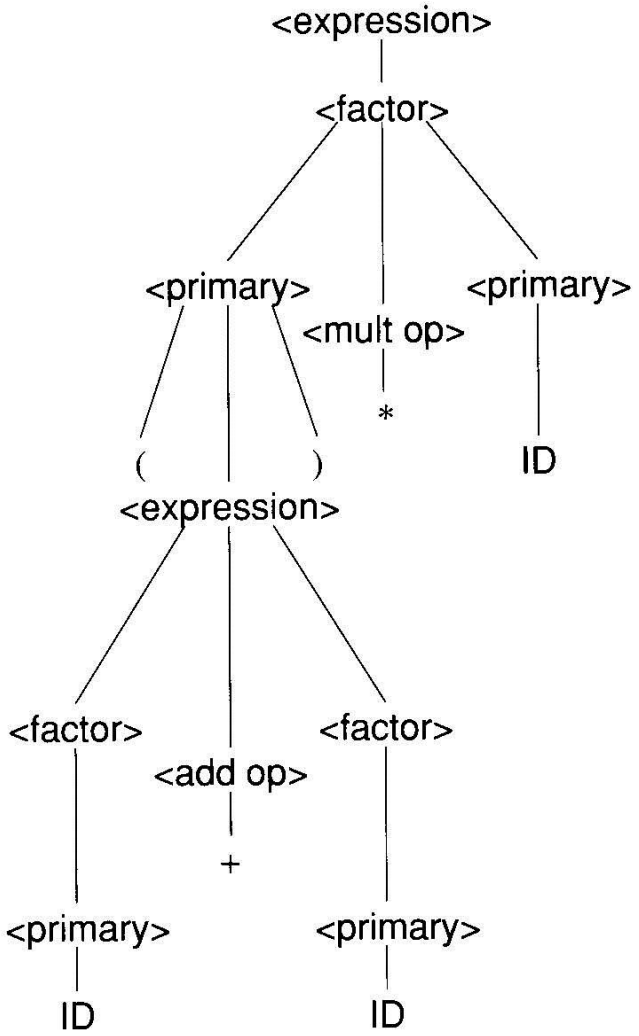


Figure 2.6 Derivation Tree for (A+B)*C

Recursive Descent Parsing

- There are many parsing techniques.
  - Recursive descent is one of the *simplest* parsing techniques
- Basic idea
  - Each nonterminal has a *parsing procedure*
  - For symbol on the RHS : a sequence of matching
    - To Match a nonterminal A
      - Call the parsing procedure of A
    - To match a terminal symbol t
      - Call match(t)
        - » match(t) calls the scanner to get the next token. If it is, everything is correct. If it is not t, we have found a syntax error.

Recursive Descent Parsing

- If a nonterminal has several productions, choose an appropriate one based on the next input token.
- Parser is started by invoking system_goal().

```
void system_goal(void)
{
     /* <system goal> ::= <program> SCANEOF */
     program();
     match(SCANEOF);
}

void program(void)
{
     /* <program> ::= BEGIN <statement list> END */
     match(BEGIN);
     statement_list();
     match(END);
}
```

```
void statement_list(void)
{
    /*
     * <statement list> ::= <statement>
     *                      { <statement> }
     */
    statement();
    while (TRUE)  {
        switch (next_token()) {
        case ID:
        case READ:
        case WRITE:
            statement();
            break;
        default:
            return;
        }
    }
}
```

處理{<statement>}不出現的情形

- **next_token()** : a function that returns the next token. It does not call scanner(void).

```
void statement(void)
{
    token tok = next_token();

    switch (tok) {
    case ID:
        /* <statement> ::= ID := <expression> ; */
        match(ID); match(ASSIGNOP);
        expression(); match(SEMICOLON);
        break;

    case READ:
        /* <statement> ::= READ ( <id list> ) ; */
        match(READ); match(LPAREN);
        id_list(); match(RPAREN);
        match(SEMICOLON);
        break;

    case WRITE:
        /* <statement> ::= WRITE ( <expr list> ) ; */
        match(WRITE); match(LPAREN);
        expr_list(); match(RPAREN);
        match(SEMICOLON);
        break;

    default:
        syntax_error(tok);
        break;
    }
}
```

<statement>
必出現一次

```
void id_list(void)
{
    /* <id list> ::= ID { , ID } */
    match(ID);

    while (next_token() == COMMA) {
        match(COMMA);
        match(ID);
    }
}

void expression(void)
{
    token t;

    /*
     * <expression> ::= <primary>
     *                  { <add op> <primary> }
     */
    primary();
    for (t = next_token(); t == PLUSOP || t == MINUSOP;
                  t = next_token()) {
        add_op();
        primary();
    }
}
```

```
void expr_list(void)
{
 /* <expr list> ::= <expression> { , <expression> } */
    expression();

    while (next_token() == COMMA) {
        match(COMMA);
        expression();
    }
}

void add_op(void)
{
    token tok = next_token();

    /* <addop> ::= PLUSOP | MINUSOP */
    if (tok == PLUSOP || tok == MINUSOP)
        match(tok);
    else
        syntax_error(tok);
}
```

```
void primary(void)
{
    token tok = next_token();

    switch (tok) {
    case LPAREN:
        /* <primary> ::= ( <expression> ) */
        match(LPAREN); expression();
        match(RPAREN);
        break;

    case ID:
        /* <primary> ::= ID */
        match(ID);
        break;

    case INTLITERAL:
        /* <primary> ::= INTLITERAL */
        match(INTLITERAL);
        break;

    default:
        syntax_error(tok);
        break;
    }
}
```

Figure 2.7 Remaining Parsing Procedures for Micro

Translating Micro

- Target language: 3-addr code (quadruple)
  - OP A, B, C
- Note that we did not worry about registers at this time.
  - temporaries: Sometimes we need to hold temporary values.
    - E.g. A+B+C

      ADD A,B,TEMP&1

      ADD TEMP&1,C,TEMP&2

Translating Micro (Cont'd)

- Action Symbols
  - The bulk of a translation is done by semantic routine
  - Action symbols can be added to a grammar to specify when semantic processing should take place
    - Be placed anywhere in the RHS of a production
  - translated into procedure call in the parsing procedures
    - #add corresponds to a semantic routine named add()
  - No impact on the languages recognized by a parser driven by a CFG

```
<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_expr
                        {, <expression> #write_expr}
<expression>       → <primary>
                        {<add op> <primary> #gen_infix}
<primary>          → ( <expression> )
<primary>          → <ident>
<primary>          → INTLITERAL #process_literal
<add op>           → PLUSOP #process_op
<add op>           → MINUSOP #process_op
<ident>            → ID #process_id
<system goal>      → <program> SCANEOF #finish
```

Figure 2.9 Grammar for Micro with Action Symbols

Semantic Information

- ***Semantic routines*** need certain information to do their work.
  - These information is stored in ***semantic records***.
  - Each kind of grammar symbol has a semantic record

```
#define MAXIDLEN        33
typedef char string[MAXIDLEN];

typedef struct operator {     /* for operators */
    enum op { PLUS, MINUS } operator;
} op_rec;

/* expression types */
enum expr { IDEXPR, LITERALEXPR, TEMPEXPR };

/* for <primary> and <expression> */
typedef struct expression {
    enum expr kind;
    union {
        string name;      /* for IDEXPR, TEMPEXPR */
        int val;          /* for LITERALEXPR */
    };
} expr_rec;
```

Figure 2.8 Semantic Records for Micro Grammar Symbols

```
void expression(void)
{
    token t;

    /*
     * <expression> ::= <primary>
     *                  { <add op> <primary> }
     */
    primary();
    for (t = next_token(); t == PLUSOP || t == MINUSOP;
                  t = next_token()) {
        add_op();
        primary();
    }
}
```

Old parsing procedure P. 37

```
void expression(expr_rec *result)
{
    expr_rec left_operand, right_operand;
    op_rec op;

    primary(& left_operand);
    while (next_token() == PLUSOP ||
           next_token() == MINUSOP) {
        add_op(& op);
        primary(& right_operand);
        left_operand = gen_infix(left_operand, op,
                                 right_operand);
    }
    *result = left_operand;
}
```

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

New parsing procedure which involved semantic routines

Figure 2.11 A Parsing Procedure Including Semantic Processing

Semantic Information (Cont'd)

- Subroutines for symbol table and temporaries

```
/* Is s in the symbol table? */
extern int lookup(string s);

/* Put s unconditionally into symbol table. */
extern void enter(string s);

void check_id(string s)
{
    if (! lookup(s)) {
        enter(s);
        generate("Declare", s, "Integer", "");
    }
}
```

```
char *get_temp(void)
{
    /* max temporary allocated so far */
    static int max_temp = 0;
    static char tempname[MAXIDLEN];

    max_temp++;
    sprintf(tempname, "Temp&%d", max_temp);
    check_id(tempname);
    return tempname;
}
```

Semantic Information (Cont'd)

- Semantic routines

```
void start(void)
{
    /* Semantic initializations, none needed. */
}


void finish(void)
{
    /* Generate code to finish program. */
    generate("Halt", "", "", "");
}


void assign(expr_rec target, expr_rec source)
{
    /* Generate code for assignment. */
    generate("Store", extract(source),
             target.name, "");
}
```

```
op_rec process_op(void)
{
    /* Produce operator descriptor. */
    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_rec;
    /* An expr_rec with temp variant set. */
    e_rec.kind = TEMPEXPR;

    /*
     * Generate code for infix operation.
     * Get result temp and set up semantic record
     * for result.
     */
    strcpy(erec.name, get_temp());
    generate(extract(op), extract(e1),
             extract(e2), erec.name);
    return erec;
}
```

```
void read_id(expr_rec in_var)
{
    /* Generate code for read. */
    generate("Read", in_var.name,
             "Integer", "");
}

expr_rec process_id(void)
{
    expr_rec t;

    /*
     * Declare ID and build a
     * corresponding semantic record.
     */
    check_id(token_buffer);
    t.kind = IDEXPR;
    strcpy(t.name, token_buffer);
    return t;
}
```

```
expr_rec process_literal(void)
{
    expr_rec t;

    /*
     * Convert literal to a numeric representation
     * and build semantic record.
     */
    t.kind = LITERALEXPR;
    (void) sscanf(token_buffer, "%d", & t.val);
    return t;
}


void write_expr(expr_rec out_expr)
{
    generate("Write", extract(out_expr),
             "Integer", "");
}
```

Figure 2.10 Action Routines for Micro

Step	Parser Action	Remaining Input	Generated Code
(1)	Call `system_goal()`	**begin** A:=BB–314+A ; **end** SCANEOF	
(2)	Call `program()`	**begin** A:=BB–314+A ; **end** SCANEOF	
(3)	Semantic Action: `start()`	**begin** A:=BB–314+A ; **end** SCANEOF	
(4)	`match(BEGIN)`	**begin** A:=BB–314+A ; **end** SCANEOF	
(5)	Call `statement_list()`	A:=BB–314+A ; **end** SCANEOF	
(6)	Call `statement()`	A:=BB–314+A ; **end** SCANEOF	
(7)	Call `ident()`	A:=BB–314+A ; **end** SCANEOF	
(8)	`match(ID)`	A:=BB–314+A ; **end** SCANEOF	
(9)	Semantic Action: `process_id()`	:=BB–314+A ; **end** SCANEOF	Declare A,Integer
(10)	`match(ASSIGNOP)`	:=BB–314+A ; **end** SCANEOF	
(11)	Call `expression()`	BB–314+A ; **end** SCANEOF	
(12)	Call `primary()`	BB–314+A ; **end** SCANEOF	
(13)	Call `ident()`	BB–314+A ; **end** SCANEOF	
(14)	`match(ID)`	BB–314+A ; **end** SCANEOF	
(15)	Semantic Action: `process_id()`	–314+A ; **end** SCANEOF	Declare BB,Integer
(16)	Call `add_op()`	–314+A ; **end** SCANEOF	
(17)	`match(MINUSOP)`	–314+A ; **end** SCANEOF	
(18)	Semantic Action: `process_op()`	314+A ; **end** SCANEOF	
(19)	Call `primary()`	314+A ; **end** SCANEOF	
(20)	`match(INTLITERAL)`	314+A ; **end** SCANEOF	
(21)	Semantic Action: `process_literal()`	+A ; **end** SCANEOF	
(22)	Semantic Action: `gen_infix()`	+A ; **end** SCANEOF	Declare Temp&1,Integer Sub BB,314,Temp&1
(23)	Call `add_op()`	+A ; **end** SCANEOF	

(24)	`match(PLUSOP)`	+A ; **end** SCANEOF	
(25)	Semantic Action: `process_op()`	A ; **end** SCANEOF	
(26)	Call `primary()`	A ; **end** SCANEOF	
(27)	Call `ident()`	A ; **end** SCANEOF	
(28)	`match(ID)`	A ; **end** SCANEOF	
(29)	Semantic Action: `process_id()`	; **end** SCANEOF	
(30)	Semantic Action: `gen_infix()`	; **end** SCANEOF	Declare Temp&2,Integer Add Temp&1,A,Temp&2
(31)	Semantic Action: `assign()`	; **end** SCANEOF	Store Temp&2,A
(32)	`match(SEMICOLON)`	; **end** SCANEOF	
(33)	`match(END)`	**end** SCANEOF	
(34)	`match(SCANEOF)`	SCANEOF	
(35)	Semantic Action: `finish()`		Halt