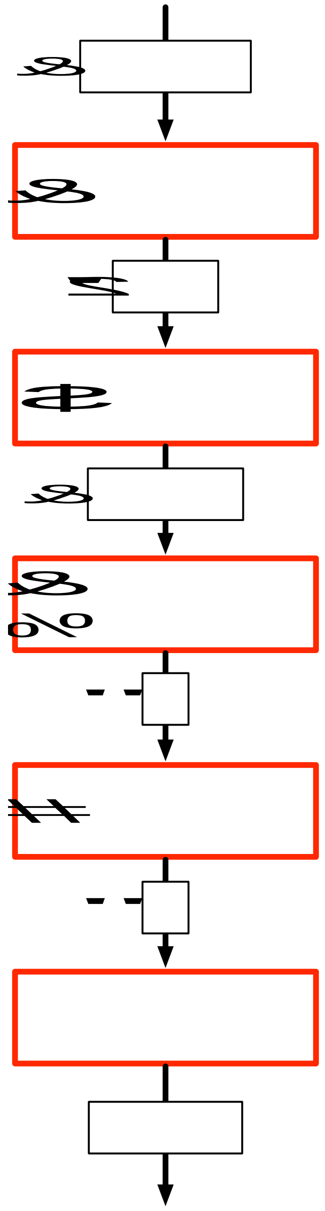
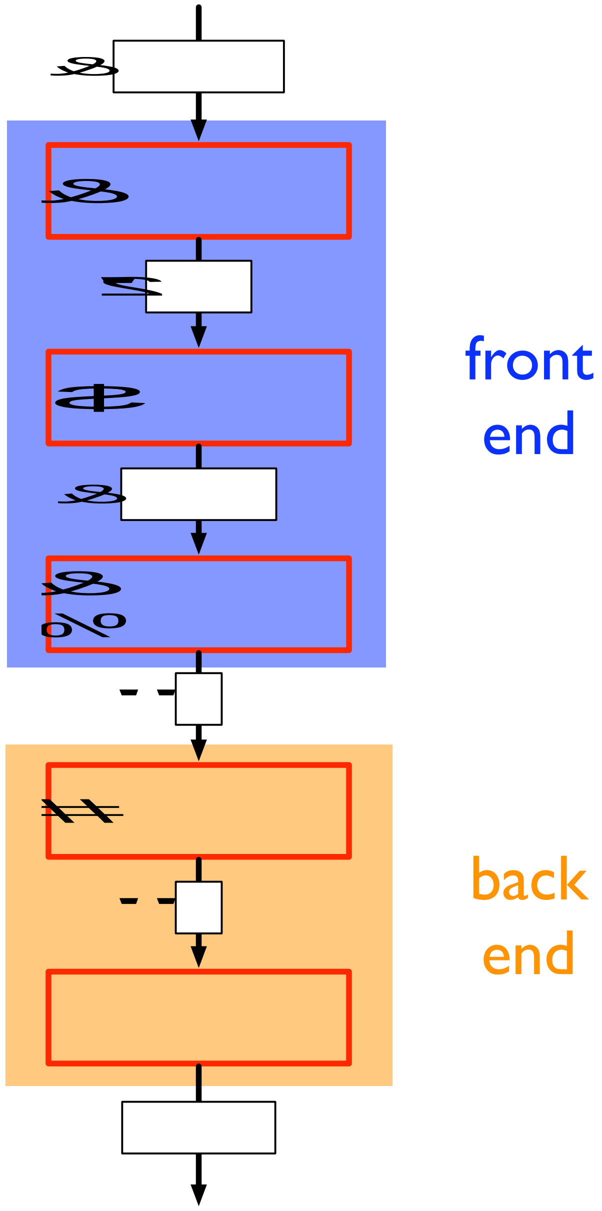
Last time

- What are compilers?
- Phases of a compiler



Extra: Front-end vs. Back-end

- Scanner + Parser + Semantic actions + (high level) optimizations called the *front-end* of a compiler
- IR-level optimizations and code generation (instruction selection, scheduling, register allocation) called the *back-end* of a compiler
- Can build multiple front-ends for a particular back-end
  - e.g., gcc & g++, or many front-ends which generate CIL
- Can build multiple back-ends for a particular front-end
  - e.g., gcc allows targeting different architectures



The MICRO compiler: a simple example

High level structure

- Single pass compiler: no intermediate representation
  - Scanner tokenizes input stream, but is called by parser on demand
  - As parser recognizes constructs, it invokes semantic routines
  - Semantic routines build symbol table on-the-fly and directly generate code for a 3-address virtual machine

The Micro language

- Tokens are defined by *regular expressions*
    - Tokens: BEGIN, END, READ, WRITE, ID, LITERAL, LPAREN, RPAREN, SEMICOLON, COMMA, ASSIGN_OP, PLUS_OP, MINUS_OP, SCANEOF
- Implicit identifier declaration (no need to predeclare variables): ID = [A-Z][A-Z0-9]*
- Literals (numbers): LITERAL = [0-9]+
- Comments (not passed on as tokens): --(Not(\n))*\n
- Program:
    - BEGIN {statements} END

The Micro language

- One data type—all IDs are integers
- Statement:

  ID := EXPR

  - Expressions are simple arithmetic expressions which can contain identifiers
  - Note: no unary minus
- Input/output

  READ(ID, ID, ...)

  WRITE(EXPR, EXPR, ...)

Scanner

- What the scanner can identify corresponds to what the finite automaton for a regular expression can accept
- Identifies the next token in the input stream
  - Read a token (process finite automaton until accept state found)
  - Identify its type (determine which accept state the FA is in)
  - Return type and "value" (*e.g.*, type = LITERAL, value = 5)

Recognizing tokens

- Skip spaces
- If the first non-space character is:
    - letter: read until non-alphanumeric. Check for reserved words ("begin," "end"). Return reserved word or (ID and variable name)
    - digit: read until non-digit. Return LITERAL and number
    - ( ) ; , +: return single character
    - : : next must be =. Return ASSIGN_OP
    - - : if next is also - skip to end of line, otherwise return MINUS_OP
- "unget" the next character that had to be read to find end of IDs, reserved words, literals and minus ops.

Parsers and Grammars

- Language syntax is usually specified with *context-free grammars* (CFGs)
- Backus-Naur form (BNF) is the standard notation
- Written as a set of rewrite rules:
  - Non-terminal ::= (set of terminals and non-terminals)
  - Terminals are the set of tokens
  - Each rule tells how to compose a non-terminal from other non-terminals and terminals

Micro grammar

program	::=	BEGIN statement_list END
statement_list	::=	statement; | statement; statement_list
statement	::=	ID := expression | READ(id_list) | WRITE(expr_list)
id_list	::=	ID | ID, id_list
expr_list	::=	expression | expression, expr_list
expression	::=	primary | primary add_op expression
primary	::=	(expression) | ID | LITERAL
add_op	::=	PLUSOP | MINUSOP
system_goal	::=	program SCANEOF

Relating the CFG to a program

- CFGs can produce a program by applying a sequence of productions
- How to produce BEGIN id := id + id; END
    - Rewrite by starting with the goal production and replacing non-terminals with the rule's RHS

Relating the CFG to a program

- CFGs can produce a program by applying a sequence of productions
- How to produce BEGIN id := id + id; END
    - Rewrite by starting with the goal production and replacing non-terminals with the rule's RHS

program SCANEOF

replace program

Relating the CFG to a program

- CFGs can produce a program by applying a sequence of productions
- How to produce BEGIN id := id + id; END
    - Rewrite by starting with the goal production and replacing non-terminals with the rule's RHS

BEGIN statement_list END

replace statement_list

Relating the CFG to a program

- CFGs can produce a program by applying a sequence of productions
- How to produce BEGIN id := id + id; END
  - Rewrite by starting with the goal production and replacing non-terminals with the rule's RHS

BEGIN statement; END

replace statement

Relating the CFG to a program

- CFGs can produce a program by applying a sequence of productions
- How to produce BEGIN id := id + id; END
    - Rewrite by starting with the goal production and replacing non-terminals with the rule's RHS

BEGIN ID := expression; END

replace expression

Relating the CFG to a program

- CFGs can produce a program by applying a sequence of productions
- How to produce BEGIN id := id + id; END
  - Rewrite by starting with the goal production and replacing non-terminals with the rule's RHS

BEGIN ID := primary add_op expression; END

replace 1st primary

Relating the CFG to a program

- CFGs can produce a program by applying a sequence of productions
- How to produce BEGIN id := id + id; END
  - Rewrite by starting with the goal production and replacing non-terminals with the rule's RHS

BEGIN ID := ID add_op expression; END

replace add_op

Relating the CFG to a program

- CFGs can produce a program by applying a sequence of productions
- How to produce BEGIN id := id + id; END
    - Rewrite by starting with the goal production and replacing non-terminals with the rule's RHS

BEGIN ID := ID + expression; END

replace expression

Relating the CFG to a program

- CFGs can produce a program by applying a sequence of productions
- How to produce BEGIN id := id + id; END
    - Rewrite by starting with the goal production and replacing non-terminals with the rule's RHS

BEGIN ID := ID + primary; END

replace primary

Relating the CFG to a program

- CFGs can produce a program by applying a sequence of productions
- How to produce BEGIN id := id + id; END
    - Rewrite by starting with the goal production and replacing non-terminals with the rule's RHS

BEGIN ID := ID + ID; END

How do we go in reverse?

- How do we parse a program given a CFG?
- Start at goal term, rewrite productions from left to right
  - If it is a terminal, make sure we match input token
    - Otherwise, there is a *syntax error*
  - If it is a non-terminal
    - If there is a single choice for a production, pick it
    - If there are multiple choices for a production, choose the production that matches the next token(s) in the stream
      - e.g., when parsing statement, could use production for ID, READ or WRITE
    - Note that this means we have to *look ahead* in the stream to match tokens!

Question: how much lookahead?

program	::=	BEGIN statement_list END
statement_list	::=	statement; | statement; statement_list
statement	::=	ID := expression | READ(id_list) | WRITE(expr_list)
id_list	::=	ID | ID, id_list
expr_list	::=	expression | expression, expr_list
expression	::=	primary | primary add_op expression
primary	::=	(expression) | ID | LITERAL
add_op	::=	PLUSOP | MINUSOP
system_goal	::=	program SCANEOF

Recursive descent parsing

- Idea: parse using a set of mutually recursive functions
  - One function per non-terminal
  - Each function attempts to match any terminals in its production
  - If a rule produces non-terminals, call the appropriate function

```
statement() {
   token = peek_at_match();
   switch(token) {
   case ID:
      match(ID); //consume ID
      match(ASSIGN); //consume :=
      expression(); //process non-terminal
      break;
   case READ:
      match(READ); //consume READ
      match(LPAREN); //match (
      id_list(); //process non-terminal
      match(RPAREN); //match )
      break;
   case WRITE:
      match(WRITE);
      match(LPAREN); //match (
      expr_list(); //process non-terminal
      match(RPAREN); //match )
      break;
   }
   match(SEMICOLON);
}
```

statement ::= ID := expression; | READ(id_list) | WRITE(expr_list)

Recursive descent parsing (II)

- How do we parse id_list ::= ID id_list
  - Basic idea:

```
id_list() {
   match(ID); //consume ID
   if (peek_at_match() == COMMA) {
      match(COMMA)
      id_list();
   }
}
```

  - This is equivalent to the following loop (tail recursion)

```
id_list() {
   match(ID); //consume ID
   while (peek_at_match() == COMMA) {
      match(COMMA)
      match(ID);
   }
}
```

  - Note: in both cases, if peek_at_match() isn't COMMA, we don't consume the next token!

General rules

- One function per non-terminal
- Non-terminals with multiple choices (like statement) use `case` or `if` statements to distinguish
  - Conditional based on *first set* of the non-terminal, the terminals that can distinguish between productions
- When non-terminal encountered, call appropriate function
  - Functions are *mutually recursive*
  - Some rules (like id_list) can be implemented with loops

Semantic processing

- Want to generate code for a 3-address machine:
    - `OP A, B, C` performs `A` op `B` → `C`
- Temporary variables may be created to convert more complex expressions into three-address code
    - Naming scheme: Temp&1, Temp&2, etc.

$$\texttt{D = A + B * C} \longrightarrow$$

```
MULT C, B, Temp&1
ADD A, Temp&1, Temp&2
STORE &Temp2, D
```

Semantic action routines

- To produce code, we call routines during parsing to generate three-address code.
- These *action routines* do one of two things:
  - Collect information about passed symbols for use by other semantic action routines. This information is stored in semantic records.
  - Generate code using information from semantic records. and the current parse procedure
  - Note: for this to work correctly, we must parse expressions according to order of operations (*i.e.*, must parse a * expression before a + expression)

Operator Precedence

- Operator precedence can be specified in the CFG
  - CFG can determine the order in which expressions are parsed
- For example:

expr	::=	factor {+ factor}
factor	::=	primary {* primary}
primary	::=	(expr) | ID | LITERAL

- Because +-expressions are composed of *-expressions, we will finish dealing with the * production before we finish with the + production

Example

- Annotations are inserted into grammar, specifying when semantic routines should be called

```
statement ::= ID = expr #assign
expr ::= term + term #addop
term ::= ID #id | LITERAL #num
```

- Consider A = B + 2;
  - num() and id() create semantic records containing ID names and number values
  - addop() generates code for the expression, using information from the num() and id() records, and creates a temporary variable
  - assign() generates code for the assignment using the temporary variable generated by addop()

Calling semantic routines

```
statement() {
   match(ID); //consume ID
   match(ASSIGN); //consume :=
   expr(); //process non-terminal
}

expr() {
   term(); //process non-terminal
   match(PLUS); //consume +
   term(); //process non-terminal
}

term() {
   token = peek_at_match();
   switch(token) {
   case ID:
      match(ID);
      break;
   case LIT;
      match(LIT);
      break;
   }
}
```

```
statement() {
   match(ID); //consume ID
   match(ASSIGN); //consume :=
   expr(); //process non-terminal
   assign();
}
expr() {
   term(); //process non-terminal
   match(PLUS); //consume +
   term(); //process non-terminal
   addop();
}
term() {
   token = peek_at_match();
   switch(token) {
   case ID:
      match(ID);
      id();
      break;
   case LIT;
      match(LIT);
      num();
      break;
   }
}
```

Next time

- Scanners
  - How to specify the tokens for a language
  - How to construct a scanner
  - How to use a scanner generator

Backup slides

Annotated Micro Grammar (fig. 2.9)

```
Program          ::= #start BEGIN Statement-list END
Statement-list   ::= Statement {Statement}
Statement        ::= ID := Expression;   #assign |
                      READ ( Id-list ) ; |
                      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 )    |
                      Ident             |
                      INTLITERAL  #process_literal
Ident            ::= ID #process_id
Add-op           ::= PLUSOP  #process_op |
                      MINUSOP #process_op
System-goal      ::= Program SCANEOF  #finish
```

Annotated Micro Grammar

Program ::= #start **BEGIN** Statement-list **END**

Semantic routines in Chap. 2 print information about what the parser has recognized.

At #start, nothing has been recognized, so this takes no action. End of parse is recognized by the final production:

System-goal ::= Program **SCANEOF** **#**finish

In a production compiler, the #start routine might set up program initialization code (i.e. initialization of heap storage and static storage, initialization of static values, etc.)

Annotated Micro Grammar

Statement-list ::= Statement {Statement}

No semantic actions are associated with this statement because the necessary semantic actions associated with statements are done when a statement is recognized.

Annotated Micro Grammar

```
Statement    ::= ID := Expression;  #assign |
                 READ ( Id-list ) ; |
                 WRITE ( Expr-list ) ;
Expr-list    ::= Expression  #write_expr {, Expression #write_expr }
Expression   ::= Primary { Add-op Primary #gen_infix}
Primary      ::= ( Expression )    |
                  Ident             |
                  INTLITERAL  #process_literal
```

Different semantic actions used when the parser finds an expression. In **Expr-list**, it is handled with write_expr, whereas **Primary** we choose to do nothing – but could express a different semantic action if there were a reason to do so.

We know that different productions, or rules of the grammar, are reached in different ways, and can tailor semantic actions (and the grammar) appropriately.

Annotated Micro Grammar

```
Statement      ::= Ident := Expression;  #assign |
                     READ ( Id-list ) ; |
                     WRITE ( Expr-list ) ;
Id-list        ::=  Ident #read_id {, Ident #read_id }
Ident          ::= ID #process_id
```

Note that in the grammar of Fig. 2.4, there is no Ident nonterminal. By adding a nonterminal Ident a placeholder is created to take semantic actions as the nonterminal is processed.   The programs look syntactically the same, but the additional productions allow the semantics to be richer.

Semantic actions create a semantic record for the ID and thereby create something for **read_id** to work with.