Design of a Translator

Cabdifatah Mohamed

Introduction

A translator is a software tool that converts source code written in a high-level programming language into another form, typically an intermediate representation, machine code, or executable code. The design of a translator involves creating a set of components that together perform lexical analysis, syntax analysis, semantic analysis, and code generation. This report focuses on the design of a specific translator, primarily examining the structure of the grammar, the attributes associated with terminals and non-terminals, the actions performed on each production rule, and the auxiliary data structures and functions that support the translation process.

The translator in question is based on a subset of a simple programming language featuring basic constructs such as variable declarations, arithmetic expressions, and input/output operations.

(a) Augmentation of the Grammar

In the context of designing a translator, **grammar augmentation** refers to modifications made to the initial language grammar to facilitate analysis and translation. These modifications typically aim to resolve ambiguities, accommodate semantic checking, and prepare for code generation.

For the current language, the grammar defines the syntax for declarations, expressions, statements, and program structure. The original grammar, which is augmented for the translator design, might have been initially simple but requires enhancement to account for semantic attributes, scopes, and control structures.

Augmented Grammar Example

The basic grammar rules include constructs such as:

- `PROGRAM declarations statements END PROGRAM IDENTIFIER`: Defines a program with variable declarations and a sequence of statements.
- `declarations`: Handles variable declarations, especially the `INTEGER` type.
- `statements`: Defines various kinds of statements, such as `READ`, `PRINT`, and assignments.
- `expression`: Handles arithmetic expressions and terms.

For instance, the original grammar rules may look as follows:

```
program: PROGRAM declarations statements END PROGRAM IDENTIFIER;
declarations: declarations declaration | ;
declaration: INTEGER IDENTIFIER;
statements: statements statement | ;
statement: READ COMMA IDENTIFIER | PRINT COMMA STRING COMMA IDENTIFIER | IDENTIFIE
expression: expression PLUS term | expression MINUS term | term;
term: term TIMES factor | term DIVIDE factor | factor;
factor: NUM | IDENTIFIER | LPAREN expression RPAREN;
```

However, to enhance the grammar for practical use in a translator, additional modifications are made to support things like **symbol tables** and **temporary variable generation**. For example:

- Adding a `PROGRAM` rule to ensure the program is properly encapsulated and defined.
- Incorporating rules to manage **expressions** and terms more comprehensively, accommodating both **arithmetic operations** and **identifier resolution**.
- Managing **control flow** and **scoping** using additional constructs like the `DO-END DO` block.

Thus, the augmented grammar includes rules for handling more complex constructs, ensuring proper syntax handling, and accommodating the semantic operations like type checking.

(b) Attributes of Various Terminals and Non-terminals

Attributes are essential for associating semantic information with the syntactic structures of a program. These attributes can represent the types of variables, their values, offsets in memory, or code generation information. Terminals (tokens) and non-terminals in the grammar are assigned attributes that provide necessary information during the translation process.

Attributes of Terminals

- **NUM**: Represents integer literals. The attribute is an integer value (`intVal`) that holds the value of the literal.
- **IDENTIFIER**: Represents variable names. The attribute is a string (`strVal`) that holds the name of the identifier.
- **STRING**: Represents string literals. The attribute is a string (`strVal`) that contains the literal string.
- **ASSIGNOP**: Represents assignment operator (`=`). The attribute is not needed as it is purely syntactic.
- **COMMA, PLUS, MINUS, TIMES, DIVIDE, LPAREN, RPAREN**: These are operators or delimiters. They typically do not carry any semantic information, but the parser uses them to structure the program.

Attributes of Non-terminals

- **program**: This non-terminal represents the entire program. It may have a `name` attribute for the program's name.
- **declarations**: The non-terminal could include a list of symbol table entries (for the variables declared).
- **declaration**: This non-terminal represents a declaration of a variable and could have attributes like `type`, `size`, and `offset`.
- **statements**: This non-terminal represents a sequence of statements and might have a list of attributes for each statement.

- **expression**: The attribute may represent the type of the expression (e.g., integer, string) and its value if evaluated.

Attribute Propagation

In a syntax-directed translation scheme, attributes are propagated through the parse tree. For instance:

- In `expression` → `expression PLUS term`, the type of the expression and term would be checked to ensure type compatibility (e.g., both being integers).
- In `statement` → `IDENTIFIER ASSIGNOP expression`, the `IDENTIFIER` would be linked to the evaluated value of the `expression`.

(c) Actions on Every Production Rule

The **actions** associated with each production rule specify the operations to be performed during the parsing phase. These actions include constructing parse tree nodes, updating the symbol table, type checking, and generating intermediate code.

Example Production Rules and Actions

- **Program Rule**:

  ```
  program: PROGRAM declarations statements END PROGRAM IDENTIFIER;
  ```

  Action: Create a new program entry in the symbol table, record the program's name, and link the declarations and statements.

- **Declarations Rule**:

  ```
  declarations: declarations declaration;
  ```

  Action: Insert the variable declared into the symbol table with its type and other attributes (e.g., memory offset).

- **Statement Rule** (Assignment):

  ```
  statement: IDENTIFIER ASSIGNOP expression;
  ```

  Action: Look up the `IDENTIFIER` in the symbol table, ensure the `expression` type matches the variable type, and generate code for the assignment.

- **Expression Rule** (Addition):

  ```
  expression: expression PLUS term;
  ```

  Action: Evaluate the left and right operands (`expression` and `term`), perform type checking, and generate code for the addition operation.

- **Expression Rule** (Parentheses):

  ```
  factor: LPAREN expression RPAREN;
  ```

  Action: Recursively process the inner `expression` and return the result.

The actions associated with these rules may also interact with auxiliary data structures like the **symbol table**, where variables are stored, and the **quadruple** or **three-address code** for generating intermediate code.

(d) Auxiliary Data Structures and Functions

The design of the translator requires several auxiliary data structures to manage symbol tables, intermediate code generation, and backpatching.

Symbol Table

The **symbol table** stores information about identifiers, including their types, sizes, offsets, and scope. This table is critical for type checking, variable resolution, and memory allocation during code generation.

Data Structure:

```
typedef struct symbol_table_record {
    char* name;            // Variable or function name
    char** type;           // Data type of the variable
    char* initial_value;   // Initial value if any
    int size;              // Size of the variable
    int offset;            // Memory offset
    int array_ind;         // Array indicator (if the variable is an array)
};

typedef struct symbol_table {
    symbol_table_record* list;   // List of symbol table records
    int size;                    // Size of the table
    int curr_size;               // Current number of entries
    int curr_offset;             // Current memory offset
    char* name;                  // Name of the symbol table (scope)
};
```

Quadruple Representation

The translator generates **three-address code** (TAC), often represented in quadruple form. This code serves as an intermediate representation for the compiler to optimize before generating final machine code.

Quadruple Structure:

```
typedef struct fields_quad {
    char* arg1;             // First operand
    char* arg2;             // Second operand
    char* res;              // Result of the operation
    quad_data_types op;     // Operation type (e.g., addition, multiplication)
};

typedef struct quadArray {
```

```
    fields_quad* quad_Table; // Array of quadruples
};
```

Functions for Auxiliary Operations

- **Insert/Lookup Functions**: These functions help insert new records into the symbol table and search for existing ones.

  ```
  void insert(symbol_table* this, symbol_table_record* x);
  symbol_table_record* lookup(symbol_table* this, const char* name);
  ```

- **Temporary Variable Generation**: These functions help create temporary variables needed during the intermediate code generation process.

  ```
  symbol_table_record* gentemp(symbol_table* this, enum date_types temp);
  ```

- **Backpatching**: This function is essential for handling jump addresses in the intermediate code.

  ```
  void backpatch(lnode* list, int address);
  ```

Conclusion

The design of the translator involves a systematic approach to handle the syntax and semantics of a simple programming language. By augmenting the grammar, assigning appropriate attributes to terminals and non-terminals, defining actions for each production rule, and utilizing auxiliary data structures, the translator is capable of converting the high-level source code into an intermediate representation. This process lays the groundwork for further optimization and eventual code generation, forming the backbone of the compiler design.