Compilers

2018A7PS0193P

January 29, 2021

1 Process of Translation

The process of compilers converting from source code (the high level language) to target code (the machine language) is known as translation. It consists of many steps, which are described below.

1.1 Lexical Analysis

Lexical analysis follows the following steps:

1. Identify the valid set of characters in the language

2. Break the sequence of characters into appropriate words or tokens (keywords, numbers, operators, etc.)

3. Find out whether these tokens are valid or not.

The key goal of the lexical analyzer is to break a sentence into a series of words/tokens. These breaks are generally done via certain separators. The tokens are recognized via some rules encoded into a Finite State Machine.

During the lexical phase, we may experience the following lexical errors:

- Occurrence of illegal characters

- Exceeding the length of identifier

The output of the lexical analyzer will be a sequence of tokens and their "type", which signifies whether it is a identifier, operator, etc.

1.2   Syntax Analysis

The syntax analysis takes the sequence of tokens as an input, and generates a parse tree. In case the syntax is not correct according to the grammar rules, it flags a syntactical error. This is modelled using Context Free Grammars that will be recognized using PDAs or Table Driven processes.

1.3   Semantic Analysis

Semantic analysis takes the parse tree as an input, and outputs a disambiguated parse tree. It performs the following:

- Check Semantics

- Error reporting (types, etc.)

- Disambiguate overloaded operators (meaning of operators depends on operands)

- Type coercion (type casting)

- Uniqueness checking (redeclaration of variables)

As such, the disambiguated parse tree gives us an unambiguous representation of the parse tree.

The phases mentioned till now comprise the **front end** of the compiler, where the source code is handled. After this, the compiler works on generating the target code.

1.4   Code optimization

This is an optional phase that modifies the programs to run faster and consume less resources like memory, registers, etc. However, it will not change the representation of the program.

Some examples of machine independent code optimization done is:

- Common sub-expression elimination

- Copy Propagation

- Dead code elimination

- Code Motion

- Strength Reduction

- Constant Folding

TODO: See what these do and examples of each.

1.5   Code Generation

This is the process of mapping from source level abstractions (identifiers, values, etc) to target machine abstractions (registors, memory, etc.). This is a two step process - initially, intermediate code gets generated from the disambiguated parse tree, which is used to generate the final machine code.

During code generation, we have to do the following:

- Map identifiers to locations (memory or registers)

- Map source code operators to opcodes or sequences of opcodes.

- Transform conditionals and iterations to a test/jump or compare instructions

- We use layout parameter passing protocols - the locations for parameters, return values, etc.

1.6   Post Translation Optimizations

Unlike in the code optimization phase where we perform machine independent code optimizations, this does machine-dependent code optimizations. This is an optional phase as well, where we may remove unneeded operations or rearrange to prevent hazards. It is a flexible phase, and may occur at any time in the back-end of the compiler.

1.7   Symbol Table

The symbol table contains information required about the source program identifiers during compilation, including:

- Category of variable

- Data type

- Quantity stored in structure

- Scope information

- Address in Memory

The symbol table must be present in every phase of the compiler, and is used in all the phases to get information about the identifiers.

2 Advantages and Disadvantages of Compilers

The advantages of compilers are:

- Highly modular in nature

- It is retargetable. This means that if there is a single language and multiple machines, then we can use the same front end. If there are many languages

- Source code and machine independent optimizations are possible.

The limitations of the compiler are:

- Design of programming languages has a huge effect on the performance of compilers.

- Lots of work is repeatable. For S languages and M machines, $S \cdot M$ compilers are needed. This is known as the $S * M$ problem of compilers.

The $S * M$ problem is generally solved by introducing some common intermediate language, called the **Universal Intermediate Language Generator**. Some common machine independent intermediate code generation techniques are:

- Postfix Notation

- Three Address code

- Syntax tree

- Directed Acyclic Graph

3 Lexical Analysis

3.1 Functions of the Lexical Analyzer

The lexical analyzer performs the following functions:

- Take high level language as input and output a sequence of tokens

- It generally cleans the code, by stripping off blanks, tabs newlines and comments.

- Keeps track of the line numbers for associated error messages

The lexical analyzer is modelled using regular expressions. As such, it's implementation is done with a DFA. An example of one rule is $L \cdot (L + D)^*$, where L refers to a letter and D refers to a digit.

Definition 3.1. *A token is a string of characters which logically belong together, e.g. keywords, number, identifiers, etc.*

Definition 3.2. *A pattern is the set of strings for which the same token is produced.*

Definition 3.3. *A lexeme is a sequence of characters matched by a pattern to form the corresponding token.*

Now that we understand the definitions, we can see what the lexical analyzer actually does - it transforms strings to the token and passes the lexeme as it's corresponding attribute. For instance, the integer 43 would become `<num,43>`.

3.2 Working of the Lexical Analyzer

The lexical analyzer reads the character one by one from the source code into the lexeme. When it reaches a separator, it assigns a token to the lexeme based on certain rules, and continues to read the characters once more.

However, reading the lexemes character by character is slow, and involves many IO operations. This is done from a buffer instead of directly from the file. Moreover, the prefix of a lexeme is often not enough to determine the token - think of the lexemes `=` and `==`. We instead use a lookahead pointer to determine the appropriate token for a lexeme, and then push back the characters that we do not need in the current lexeme.

3.3 Symbol Table and the Lexical Analyzer

The lexical analyzer also interfaces with the symbol table. When the lexical analyzer discovers a lexeme constituting an identifier, it enters that lexeme to the symbol table. Sometimes, information regarding the token of a particular lexeme may also be store in the symbol table. As such, the symbol table must implement the following operations:

1. `insert(s,t)` : Save lexeme `s` and token `t` and return pointer.

2. `lookup(s)` : return the index of entry for lexeme `s` or '0' if `s` is not found.

To make the symbol table space efficient, we save lexemes in some separate memory, and instead store pointers to the lexemes in the symbol table.

The rule for identifying an identifier and a keyword is generally the same. To be able to tokenize the identifiers and keywords separately, we initialize the symbol table with the list of keywords, say, by calling `insert("if",keyword)`.

3.4 Challenges in Development of Lexical Analyzer

- **Free vs Fixed Lexemes** : A language could specify that lexemes must be in a free or a fixed format. For instance, in a free format, code could look like this.

  ```
  flag = flag
  * 6;
  ```

  But in the case of fixed format, this must be entirely in one line. An example of a fixed format language is Python, while a free format language is C.

- **Whitespaces** : How do we deal with whitespaces? Some languages ignore whitespaces until a separator is reached (or interpret contextually), while some languages consider the whitespaces as separators themselves. The former is much more complicated to implement than the latter.

- **Maximal Munch** : The principle of maximal munch directs the lexical analyzer to consume as much available input as possible while creating a construct. This allows us to deal with lexemes like `iff`, and correctly assign it as a identifier rather than the keyword `if`.

3.5 Techniques for specifying tokens

Definition 3.4. *Consider R_i is a regular expression and N_i is a unique name, then a regular definition is a series of definitions of the following form*

$$N_1 \rightarrow R_1$$

$$N_2 \rightarrow R_2$$

$$...$$

$$N_n \rightarrow R_n$$

where each R_i is a regular expression over $\sum \bigcup \{N_1, N_2, ...N_n\}$.

Hence, by assigning a special name N_i to the regular expression R_i, we are in effect defining macros, that remove redundancy in later parts.

The following is a regular definition for identifiers:

$$\text{Alphabet} \rightarrow A|B|C|...Z|a|b|c|...|z$$

$$\text{Digit} \rightarrow 0|1|2|...|9$$

$$\text{Identifier} \rightarrow \text{Alphabet}(\text{Alphabet}|\text{Digit})^*$$