Compilers

2018A7PS0193P

February 12, 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:** The compiler searches for instances of identical expressions and analyzes whether it is worthwhile to replace them with a single variable holding the computer value.
- **Copy Propagation:** The compiler replaces the occurrences of targets of direct assignments with their value. For instance, if we had the code `y=x; z = 3+y;`, copy propagation would yield `z=3+x;`

- **Dead code elimination:** The compiler removes code which does not affect the program results, including unreachable code and unused variables.
- **Code Motion:** The compiler moves statements and expressions out of the body of a loop if they are loop-invariant, i.e., doing so does not affect program semantics.
- **Strength Reduction:** The compiler replaces expensive operations with equivalent but less expensive operations.
- **Constant Folding:** The compiler recognizes and evaluates constant expressions at compile time instead of runtime.

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

$$N_1 \to R_1$$

$$N_2 \to R_2$$

$$...$$

$$N_n \to R_n$$

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 an example regular definition for identifiers:

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

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

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

This too comes with its own challenges. Regular expressions often fail when identifying the appropriate token, and may pass the invalid tokens to the subsequent translation phases of the compiler (how?). They are only language specifications. Tokenization is a implementation problem.

Tokenization can be done via the following steps:

1. Construct regular expressions for lexemes of each token
2. Construct R matching all lexemes of tokens, so $R = R_1 + R_2 + ...$, in some well defined precedence order.
3. Consider the input stream to be $S = s_1 s_2 ... s_n$. For $i \in [1, n]$, verify whether $s_1 ... s_i \in L(R)$.
4. If $s_1 ... s_i \in L(R) \implies s_1 .. s_i \in L(R_x)$ for some x. We choose the smallest x to be the class of $s_1 .. s_i$.
5. Discard the tokenized input and go back to step 3.

The procedure gives preference to tokens specified earlier using regular expressions. If $s_1..s_i \in L(R)$ and $s_1...s_j \in L(R)$, we choose the longest prefix, in accordance with the principle of maximal munch.

To implement our regular definitions and recognize tokens, we use **transition diagrams**. They are shown diagrammatically in the same way as Finite Automata. Transitions can be labelled with a symbol, a group of symbols, or regular definitions. A few states may be **retracting states** that indicates that the lexeme does not include the symbol that can bring us to the accepting state.

Sometimes, we may want to push back extra characters into the token stream (think of `>` and `>=`, we may want to push back the extra character read if the token is `>`). We mark those states with a $*$, to show that we must push back extra characters.

Let us consider the example of hexadecimal and octal constant. The regular definition would be:

$$hex \rightarrow 0|1|2|...|9|A|B|C|...F$$

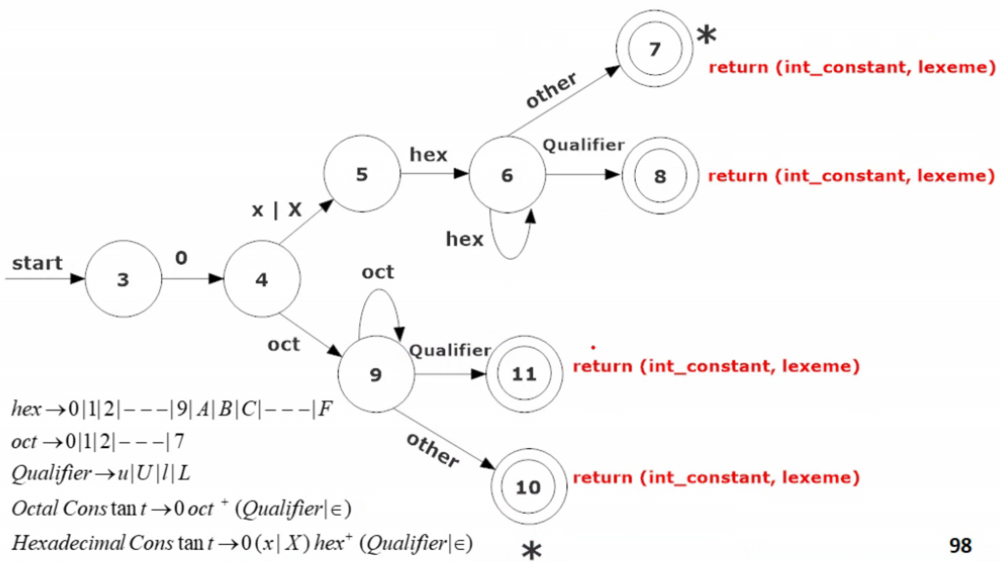
$$oct \rightarrow 0|1|2|...|7$$

$$Qualifier \rightarrow u|U|l|L$$

$$OctalConstant \rightarrow 0oct^{+}(Qualifier|\epsilon)$$

$$HexadecimalConstant \rightarrow 0(x|X)hex^{+}(Qualifier|\epsilon)$$

The transition diagram for the given regular definition is given in Fig 1.



start
3
0
4
x | X
5
hex
6
hex
other
7
*
return (int_constant, lexeme)
Qualifier
8
return (int_constant, lexeme)
oct
9
oct
Qualifier
11
return (int_constant, lexeme)
other
10
*
return (int_constant, lexeme)
hex→0|1|2|−−−|9|A|B|C|−−−|F
oct→0|1|2|−−−|7
Qualifier→u|U|l|L
Octal Constant→0 oct⁺ (Qualifier|∈)
Hexadecimal Constant→0(x|X) hex⁺ (Qualifier|∈)
98

Figure 1: Transition diagram for the Hex and Octal constants

The retracting states are not given. If we get a "bad character", we should report it as a lexical error.

Let us consider another example of a generalized expression for unsigned numbers. The regular definition is:

$$Digit \rightarrow 0|1|2|...|9$$

$$Digits \rightarrow Digit^{+}$$

$$Fraction \rightarrow' .'Digits|\epsilon$$

$$Exponent \rightarrow (E(+|-|\epsilon)Digits)|\epsilon$$

$$Number \rightarrow Digits \cdot Fraction \cdot Exponent$$

The resulting transition diagram from this is in Fig 2.

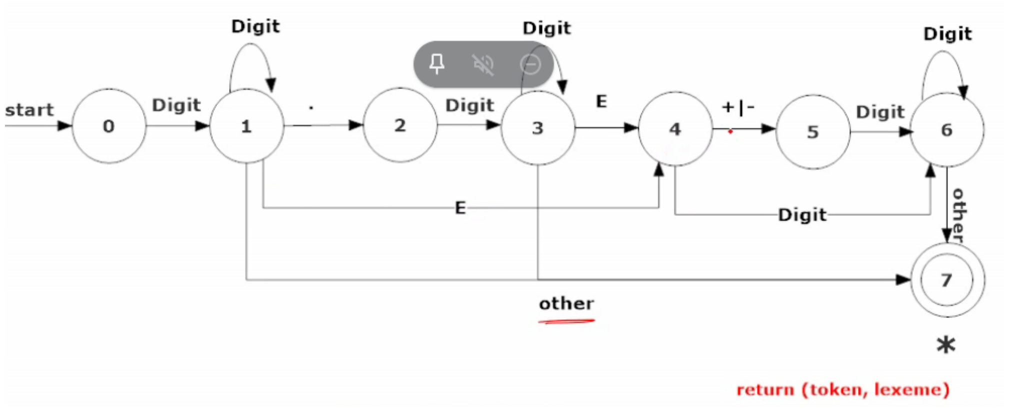


Figure 2: Transition Diagram for Unsigned Numbers

The issue here is this often introduces a lot of complexity. Instead, we may add split this into multiple transition diagrams, to improve implementation complexity. This also empirically better speeds. These multiple transition diagrams can be appropriately combined to generate a lexical analyzer.

The matching process should always start with some transition diagram. If failure occurs in one transition diagram, we retract the forward pointer to the start state, and activate the next transition diagram. If there is failure in all the transition diagrams, we throw a lexical error.

4 Syntax Analysis

4.1 Grammar Rules

Regular definitions are not enough for syntax analysis, since languages are generally not regular. Instead, we use Context Free Grammars to model our language, given by $G = \{T, N, P, S\}$, where

- T is the set of tokens (or terminals)
- N is the set of non-terminals
- P is the set of production rules
- S is the start symbol.

However, just creating the CFG is not the end of our problems - we must choose a method of derivation (left vs right), choose the right non-terminal to expand, choose the right production rule and maintain some precedence order.

When we replace the leftmost non-terminal in any sentential form of the grammar, it is called **leftmost derivation**. Choosing the rightmost non-terminal makes it **rightmost derivation**. If a grammar generates more than one leftmost/rightmost derivation of a string, it is said to be **ambiguous**.

The end goal of our syntax analysis is to create the **parse tree** - where the root is a start symbol, internal nodes are non-terminals and leaf nodes are labelled by tokens. An ambiguous grammar can have more than one parse tree.

Unfortunately, no well defined algorithm exists to remove ambiguity in a grammar. However, we can manually convert any ambiguous grammar into an unambiguous grammar. We can do this by enforcing associativity and precedence rules. This is pretty well covered in PPL and TOC.

In programming languages, ambiguity can come in many forms. For instance, consider the sentence `if A then if B then C1 else C2`. This could be:

```
if A then
{
  if B then
    C1
  else
    C2
}
```

or:

```
if A then
{
  if B then
    C1
}
else
  C2
```

Since there is ambiguity in its meaning, we must resolve it. One way to do this is to relate an `if` with the closest unrelated `else` (the second interpretation). Another way is to match each `else` with the closest previous `then`(the first interpretation). The unambiguous grammar would then be:

$$stmt \rightarrow matchedstmt | unmatchedstmt$$

$$matchedstmt \rightarrow \text{if } exp \text{ then } matchedstmt \text{ else } matchedstmt | others$$

$$unmatchedstmt \rightarrow \text{if } exp \text{ then } stmt | \text{if } exp \text{ then } matchedstmt \text{ else } unmatchedstmt$$

4.2 Parsing

Resolving ambiguity is the problem of the language specification, not of the syntax analysis. Parsing is the job of implementation. We can do this in two ways:

- **Top down parsing:** Construction of the parse tree starting from the root node (start symbol) and proceeds towards the leaves (terminals).
- **Bottom up parsing:** Construction of the parse tree starts from the terminal nodes and builds up towards the root node(start symbol).

4.2.1 Top Down Parsing

In top down parsing, we repeat the following two steps:

1. At a node labelled with non-terminal A, select one of the productions of A and construct the children nodes.
2. Find the next node at which the subtree is to be constructed.

To choose the production rule, as in step 1, backtracking must be done. We try every production rule that we can apply and see which one works. The parser must be intelligent enough to choose the production rule to apply based on the token being pointed to by the pointer. If we could determine the first character of the string produced when a production rule is applied, we could compare it to the current token and choose the production rule correctly. For this, we use a concept called the First Set.

The **First set**, denoted by `FIRST(X)` for a grammar symbol X is the set of tokens that begin the strings derivable form X. If there is a production rule:

$$A \rightarrow \alpha$$

then First(α) is the set of tokens that appear as the first token in strings generated from α.

Consider the following example:

$$S \rightarrow ABCDE$$

$$A \rightarrow a|\epsilon$$

$$B \rightarrow b|\epsilon$$

$$C \rightarrow c$$

$$D \rightarrow d|\epsilon$$

$$E \rightarrow e|\epsilon$$

Then, the first sets are as following:

- First(S) $= \{a, b, c\}$
- First(A) $= \{a, \epsilon\}$
- First(B) $= \{b, \epsilon\}$
- First(C) $= \{c\}$
- First(D) $= \{d, \epsilon\}$
- First(E) $= \{e, \epsilon\}$

Let us look at our first parser - the **recursive descent parser**. RDP is a top down method of syntax analysis in which a set of recursive procedures are executed to parse the stream of tokens. A procedure is associated with each non-terminal of the grammar. The procedure generally implements one of the production rules of the grammar.

In each procedure, we perform a match operation on hitting any token on the RHS of the grammar with the current token in the input that needs to be parsed. For example, if we have the string aba and the rule $S \rightarrow abB$, it will match the tokens a in both the RHS and the string, and move on to the next token. If there is no match, we throw a syntax error.

This approach, of course, has it's own limitations. Consider a grammar with two productions:

$$X \rightarrow \gamma_1$$

$$X \rightarrow \gamma_2$$

Suppose $\text{First}(\gamma_1) \cap \text{First}(\gamma_2) \neq \phi$, and that a is the common terminal symbol. In cases like this, we need to perform backtracking to choose the right production rule. However, RDP does not support backtracking right now. To support it, all productions should be tried in some order. Failure for some productions implies we need to try remaining productions. We would report an error only when there are no other rules.

However, a recursive descent parser may loop forever on productions of the form:

$$A \rightarrow A\alpha$$

This is known as left recursion. We can remove it from the grammar by rewriting the rule as:

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' | \epsilon$$

Left recursion can be hidden as well, where expanding production rules in some order could lead to left recursion. This is called hidden left recursion, and we must handle it in the same way as we did left recursion.

Left factoring is the process of removing the common left factor that appears in two productions of the same non-terminal. An example of a rule to remove is:

$$A \rightarrow \alpha\beta_1 | \alpha\beta_2$$

We can remove the left factoring and get:

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \alpha\beta_1 | \beta_2$$

By doing this, we can reduce the amount of backtracking done by the parser.

Consider the following grammar:

$$A \rightarrow aBb$$

$$B \to c|\epsilon$$

And suppose we want to parse an input string "ab". How would the parser know to use ϵ for B? To do this we use the **follow set**.

Follow(X) for a non terminal X is the set of symbols that might follow the derivation of X in an input stream. The follow of the start symbol S is \$ - the follow set can never be ϵ. The steps to compute the follow set is:

- Always include \$ in Follow(S).
- If there is a production rule $A \to \alpha B \beta$, where B is a non terminal, then Follow(B) = First(β).
- If there is a production rule $A \to \alpha B \beta$, where B is a non terminal, and First(β) contains ϵ, then everything in Follow(A) is in Follow(B).
- If there is a production rule $A \to \alpha B$, then Follow(B) = Follow(A).

Now that we have talked about all the issues that can face RDP, let us look at a new parser - **predictive parsing**. This is a non recursive top down parsing method, which recognizes LL(1) languages. The first L means that we scan the input stream from left to right, and the second L means that we do leftmost derivation. The predictive parser makes use of a parse table and a stack to process the input token stream.

The parse table is a two dimensional array $M[X, a]$ where X is a non terminal and a is a terminal. This parse table tells the parse table which production rule to use when we have a non-terminal X and the lookahead pointer points to a. This parse table is generated by finding the First and Follow set of every non-terminal in the grammar.

To construct the parse table, we do the following steps for each production rule $A \to \alpha$:

1. For each terminal a in First(α), $M[A, a] = A\alpha$
2. If ϵ is in First (α), $M[A, b] = A\alpha$ for each terminal b in Follow(A).
3. If ϵ is in First(α), and \$ is in Follow(A), $M[A, \$] = A \to \alpha$.

Now we can finally see the algorithm for predictive parsing. Consider that \$ is a special token at the bottom of the stack and also terminates the input string. Assume that initially, the predictive parser has X symbol on top of the stack and a is the current input symbol. Then, the predictive parser does the following:

- If $X = a = \$$, then stop.

- If $X = a \neq \$$, then pop X and increment the lookahead pointer.
- If X is a non terminal, then
  - If $M[X, a] = X \rightarrow PQR$, then pop X and push R, Q, P.
  - Else, throw an error

Predictive parsing needs a LL(1) Grammar. A grammar is LL(1) if the constructed parse tree has no multiple entries. If it does have multiple entries of same production in the same cell, it cannot be LL(1), and cannot be parsed using predictive parsing.

Another way to tell if a grammar is LL(1) is to consider rules of the type:

$$A\alpha_1|\alpha_2|\cdots|\alpha_n$$

All rules of this type must be such that $\bigcap_{i=1}^{n} First(\alpha_i) = \phi$ and rules of the type:

$$A \rightarrow \alpha|\epsilon$$

must be such that $First(\alpha) \cap Follow(A) = \phi$.