Chapter 3: Morphological Analysis

Lesson 4
Also known as *lexical analyser* or *scanner*.

**Purpose**: translation of the source code into a sequence of syntactic units.

**Procedure**:
- Given the grammar of the whole language, determine a subset of the rules which altogether form a regular sub-language (usually, identifiers, reserved words, separators, symbols, constants, comments, etc.)
- The syntactic units identified by the morphological analyser will be considered terminal symbols in the grammar used by the syntactic analyser.

**Other tasks**:
- Identification of morphological errors
- Deletion of blank-spaces
- Deletion of comments
- We could say that it does a lower-level analysis than the parser.
- The syntactic units must represent regular languages.

---

**Morphological Analyser: purpose**

Identifying the scope of the lexical analysis in the grammar of the language

- The first task: analyse the context-independent grammar, and extract the subsets of rules which describe regular languages for the syntactic units.
- A compromise solution will be taken:
  - Between the required effort and the advantages obtained.
  - Maintaining all the time the theoretical correctness.

- Example, the grammar of a version of the ASPLE language, in which we have highlighted the following elements:
  - *Reserved words*
  - *Numeric constants*
  - *Identifiers*
  - *Boolean constants*
Morphological Analyser: purpose

Identifying the scope of the lexical analysis in the grammar of the language

1: <program> ::= begin <dcl train> ; <stm train> end
2: <dcl train> ::= <declaration> | ; <dcl train>
3: <stm train> ::= <statement> | ; <stm train>
4: <declaration> ::= <mode> <idlist>
5: <mode> ::= bool | int | ref <mode>
6: <idlist> ::= <id> | <id>, <idlist>
7: <statement> ::= <asgt stm> | <cond stm> | <loop stm> | <transput stm> | <case stm>
8: <asgt stm> ::= <id> := <exp>
9: <cond stm> ::= if <exp> then <stm train> fi | if <exp> then <stm train> else <stm train> fi
10: <loop stm> ::= while <exp> do <stm train> end | repeat <stm train> until <exp>
11: <transput stm> ::= input <id> | output <exp>
12: <exp> ::= <factor> | <exp> + <factor> | <exp> - <factor> | - <exp>
13: <factor> ::= <primary> | <factor> * <primary> | <factor> / <primary>
14: <primary> ::= <id> | <constant> | ( <exp> ) | { <compare> }
15: <compare> ::= <exp> = <exp> | <exp> < <exp> | <exp> > <exp>
Morphological Analyser: purpose

Identifying the scope of the lexical analysis in the grammar of the language

• It is easy to check that there exists a regular grammar for each of the syntactic units:

<reserved word> ::= begin | i | end | bool | int | ref | = |
| cali | := | if | then | fi | else |
| while | do | repeat | until | input |
| output | + | - | * | ( | ) | = | <= | > |
| case | esac | := | procedure

<Boolean constant> ::= true | false
<int constant> ::= 0 | ··· | 9 | 0 <int constant> | ··· |
| 9 <int constant>

/id> ::= A | ··· | Z | a | ··· | Z |
| A <rest id> | ··· | Z <rest id> |
| a <rest id> | ··· | z <rest id>

<rest id> ::= A | ··· | Z | a | ··· | Z | 0 | ··· | 9 |
| 0 <rest id> | ··· | 9 <rest id> |
| A <rest id> | ··· | Z <rest id> |
| a <rest id> | ··· | z <rest id>
The following are derivation trees of the identifier “var9”, according to the grammars before. Left: Context-Independent Grammar; Right: Regular Grammar.

For the numeric constant 9390,
From the previous grammar, it is easy to find a complete regular grammar for every syntactic unit.

In order to do that, the following steps will be followed:

1. Design a new symbol which will represent any syntactic unit, e.g. \(<SU>\).
2. Add rules having
   1. At the left-hand side, the new symbol.
   2. All the right-hand sides from the rules that generate syntactic units in the previous grammar.
3. Keep, from that grammar, the rules of the non-terminal symbols that appear in those right-hand sides.
4. Remove unused rules.
**Morphological analyser: purpose**

*Identifying the scope of the lexical analysis in the grammar of the language*

- It can be seen, with examples, comparing the derivation trees, that anything that can be generated with the previous rules, can be generated from `<SU>`.  
  - Example with the identifier `var9`

```
<id>
    <rest id>
        <rest id>
            var9
```

**Morphological analyser: purpose**

*Identifying the scope of the lexical analysis in the grammar of the language*

- Example with the numeric constant `9390`

```
<int constant>
    <int constant>
        <int constant>
            9
            3
            9
            0
```

```
<int constant>
    <int constant>
    <int constant>
    <int constant>
    9
    3
    9
    0
```
Concerning where to stop,

- There are other rules in the grammar which are also regular grammars, such as:

  \[
  \begin{align*}
  7 & : \quad <\text{mode}> & ::= & \text{bool} \\
  8 & : \quad | \quad \text{int} \\
  9 & : \quad | \quad \text{ref} <\text{mode}> \\
  \end{align*}
  \]

- The idea is to find a compromise between effort and benefits.

- Having in mind:
  - It is possible to decide whether a language is regular or not (TALF I)
  - After a more-or-less costly analysis, we can determine, for each programming language, which is its regular sub-language.
  - Most of the programming languages will have identifiers, reserved words (including symbols), and constants, and the shape of these usually does not vary much from language to language. In general, these will always be regular.

In general, the reach of the morphological analyser will always be the same:

- Identifiers
- Constants
- Reserved words

This seems a good objective, given that

- They are usually very similar for all programming language. This makes it easy to formalise it with grammars and automata.
- We can, with small effort, implement, in the morphological analyser, a reasonable portion of the analysis.
• Bearing in mind that the lexical analyser will treat information from a text file, it is common to make it perform the following tasks as well:
  • Excess blank-space removal:
    • " " : white-space
    • "\t" : tabs
    • "\n" : carriage-returns
  • Comments removal:
    • Comments usually represent information which is useful for the developers to maintain the source code.
    • They do not affect the executable program and, thus, they should be removed in the initial steps by the compiler.

• ASPLE just provides a few data types, but there are others which are also very common:
  • Real numbers, e.g. 3.45, .44, -5., 3.45E2, .44E-2, -5.E123
    <real>::=<fixed point>
    | <fixed_point><exponent>
    <integer>::=<int constant>
    | -<int constant>
    <fixed_point>::=<integer>.<int_constant>
    | .<int_constant>
    | <integer>.
    <exponent>::=E<integer>
  • Character and strings, e.g. "", "hello world", ‘a’, ..., ‘z’
    <literal>::="" | "<string>"
    <character>::='<symbol>'
    <string> ::= <symbol> | <symbol><string>
The compiler can delegate other semantic tasks to the morphological analyser:
- Storing the information about the identifiers in the symbols table.
- Calculating the numeric values (in binary code) for each numeric constant.
- Etc.

These tasks vary according to:
- The objectives of the translators / interpreters.
- The division of tasks between the different components in the translator / interpreter.

These actions are sometimes expressed by inserting actions between the symbols in the rules. For instance:

\[
\text{id} ::= \text{action}_{a_0} \text{letter} \text{action}_{a_1} \text{action}_{a_2} \text{rest id}\text{action}_{a_3}
\]

\[
\text{rest id} ::= \text{letter} \text{action}_{a_1} \text{action}_{a_2} \text{rest id}
\]

\[
| \text{digit} \text{action}_{a_1} \text{action}_{a_2} \text{rest id}
\]

\[
| \lambda
\]

where
- \text{action}_{a_0} might be: initialise a counter
- \text{action}_{a_1} add 1 to the counter
- \text{action}_{a_2} copy the character which has just been recognised inside a buffer.
- \text{action}_{a_3} add to the buffer an end-of-string mark. Check that the number of characters is not higher than the maximum length allowed. If this happens, notify the error. Otherwise, insert the identifier in the symbols table and return a pointer to the element inside the table.
Morphological analyser: Semantic actions

Semantic actions

- Other example:

\[
\text{<integer>::= action}_{g0} \text{ <int constant>}
\]

| \text{action}_{g0} \text{-<int constant> action}_{g2} \\
\text{<int constant>::= <digit> action}_{g1}
| \text{<digit> action}_{g1} \text{ <int constant>}

- where
  - \text{action}_{g0} can be the initialisation of an integer variable
    \[
    \text{value} \leftarrow 0
    \]
  - \text{action}_{g1} performs the calculation of the value of the number which has been read until now
    \[
    \text{value} \leftarrow (10 \text{value}) + \text{value(digit)}
    \]
  - \text{action}_{g2} changes the sign of the value calculated:
    \[
    \text{value} \leftarrow -\text{value}
    \]

Construction of the morphological analyser

Choices

- We shall mention the following possibilities for building a compiler:
  - “Ad hoc” program development.
  - Design of a deterministic finite automata (DFA), and use of the DFA simulator for executing it. This design can be done:
    - From the regular grammar, which has been obtained from the context-independent grammar.
    - From the regular expressions of the language.
The implementation of a lexical analyser can be attempted as the development of a general program:

- The specification of requisites would consist of the following steps:
  - Describing the format and content of the input source (in the case of an ASPLE compiler, the type of the input source).
  - Describing the process: the identification of the list of syntactic units, the expected behaviour with the blank-spaces and comments, possible morphological errors, etc.
  - Describing the output, e.g. the sequence of syntactic units associated to the input source if this was correct; and the errors messages in other case.

However, it will be easier to follow a systematic approach with linear grammars or regular expressions.

---

The regular grammar that we have obtained for generating the syntactic units, which is equivalent to the context-independent grammar for that sub-language, is right-linear.

There exist algorithms for:
- Obtaining a left-linear grammar equivalent to a right-linear grammar.
- Obtaining the finite automata (not necessarily deterministic) equivalent to that grammar.
Regular grammar

Right-linear grammar

\[ \text{SU} ::= \begin{align*}
\text{begin} & | \text{;} & \text{end} & | \text{bool} & | \text{int} & | \text{ref} & | \\
\text{call} & | \text{:=} & \text{if} & \text{then} & \text{fi} & \text{else} & \\
\text{while} & | \text{do} & \text{repeat} & \text{until} & \text{input} & \\
\text{output} & | \text{+} & \text{-} & \text{*} & \text{=} & \text{:=} & \text{<=} & \text{>=} & > \\
\text{case} & | \text{esac} & \text{:=} & \text{procedure} & \\
\text{true} & | \text{false} & \\
0 & | \cdots & 9 & | 0 \ < \text{int constant} & | \cdots & \\
9 & \ < \text{int constant} & \\
A & | \cdots & Z & | a & | \cdots & z & \\
A & \ < \text{id} & | \cdots & Z & \ < \text{id} & \\
a & \ < \text{id} & | \cdots & z & \ < \text{id} & \\
\end{align*} \]

\[ \begin{align*}
\text{int constant} & ::= 0 & | \cdots & 9 & | 0 \ < \text{int constant} & | \cdots & \\
& | 9 \ < \text{int constant} & \\
\text{rest id} & ::= A & | \cdots & Z & | a & | \cdots & z & | 0 & | \cdots & 9 & \\
& | 0 \ < \text{rest id} & | \cdots & 9 \ < \text{rest id} & \\
& | A \ < \text{rest id} & \cdots & Z \ < \text{rest id} & \\
& | a \ < \text{rest id} \cdots & z \ < \text{rest id} & \\
\end{align*} \]
Regular grammar

Modification of the graph

1. Exchange initial and final nodes.
2. Reverse all arrows.

[Diagram of a graph with nodes labeled as A,...,Z, a,...,z, 0,...,9, SU, and int constant, with arrows indicating the connections between nodes.]

Regular grammar

Left-linear grammar obtained from the graph

<SU> ::= begin | end | bool | int | ref | call | if | then | fi | else | while | do | repeat | until | input | output | procedure | true | false | 0 | ··· | 9 | <int constant> 0 | ···

| <int constant> 9
| A | ··· | Z | a | ··· | z
| <rest id> A | ··· | <rest id> Z
| <rest id> a | ··· | <rest id> z
| <rest id> 0 | ··· | <rest id> 9

<int constant> ::= 0 | ··· | 9 | <int constant> 0 | ···

<int constant> 9
| <rest id> ::= A | ··· | Z | a | ··· | z
| <rest id> 0 | ··· | <rest id> 9
| <rest id> A | ··· | <rest id> Z
| <rest id> a | ··· | <rest id> z
| <rest id> a | ··· | <rest id> z
Try some examples to see that both grammars can generate the same set of words.

Example with `var9`

Example with the constant `9390`
Compilers

Obtaining the non-deterministic finite automata

• It is sometimes easier to define a set of regular expressions rather than a linear grammar.
• It is possible to:
  • Obtain the regular expression of a language.
  • Obtain, automatically, a non-deterministic finite automata (NFA) with $\lambda$-transitions, which recognises the language described by the regular expression.
  • Obtain, automatically, an equivalent DFA for that language.
Construction using regular expressions

Representation of the source code with regular expressions

- The following regular expressions correspond to syntactic units in ASPLE:
  - Identifiers:
    \[[A-Za-z][A-Za-z][0-9]*\]
    - where
      - \[[A-Za-z]\] is an abbreviation for (A+...+Z+a+...+z)
      - \[0-9\] is an abbreviation for (0+1+2+3+4+5+6+7+8+9)
  - Integers:
    \((\lambda + + -) [0-9].[0-9]\)
Construction using regular expressions

Representation of the source with regular expressions

Compilers

Compilers
Construction using regular expressions

Representation of the source with regular expressions

Construction of the morphological analyser

Simulation of finite automata

- Possibilities:
  - "Ad hoc" program development
  - Use of available software tools
Construction of the morphological analyser

"Ad hoc" simulation of finite automata

- The development of finite automata can be addressed as any other software project.

Finite Automata Simulation with available software tools

- However, there exist software tools which automate the construction of these programs.
- In this course we shall use lex/flex.

Bibliography

[Hop] “Introducción a la teoría de autómatas, lenguajes y computación” Hopcroft, J.; Motwani, R.; Ullman, J.