Sequence 2.3 – Lexical Analysis

P. de Oliveira Castro S. Tardieu

Review of the Compiler's Front-end

- The first step to compile a program is to understand its structure (syntax) and meaning (semantics)
- The analysis is twofold:
  - *Syntactic analysis* parses the program into a abstract syntax tree (AST) by following grammar rules
  - *Semantic analysis* computes the program meaning

Syntactic analysis

- Syntactic analysis itself is composed of two steps:
  - The Lexer breaks the program into tokens or words
  - The Parser assembles the tokens into an AST by following Tiger's grammar rules

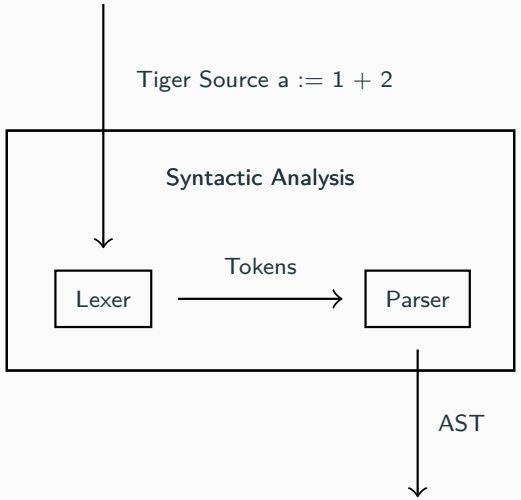

Figure 1: Syntactic Analysis

Tiger tokens

- In Tiger there are different kind of tokens.

Token	Examples
Signs or operators	`; ( ) + / = <`
Reserved words	`if then else let function`
String literals	`"hello world!\n"`
Integer literals	`42 -2754`
Identifiers	`my_variable print_int`
Comments	`/* Ignore this */`

Breaking the program into tokens ?

- A very simple lexer that break a sentence into words

```
std::string input = "hello world";
auto start = input.begin();
for(auto c = start;; c++) {
    if (*c == ' ' || c == input.end()) {
      emit_token(std::string(start, c));
      start=c+1;
    }
    if (c == input.end()) break;
}
```

- Such a simple approach does not scale to Tiger's complexity
- We require a systematic way to describe token's rules

Regular Expression

- A regular expression describes a language class.
- `[0-9]+` describes the language of positive numbers:
  - characters in the set $\{0, 1, \ldots, 9\}$ (`[0-9]`)
  - repeated 1 or more times (`+`)
- `[a-Z][a-Z0-9_]*` describes the language of identifiers:
  - first one letter (`[a-Z]`)
  - followed by a letter, number or underscore
  - repeated 0 or more times (`*`)

Regular Expression to DFA

- Every regular expression has an associated `Deterministic Finite Automaton` that recognises its language.

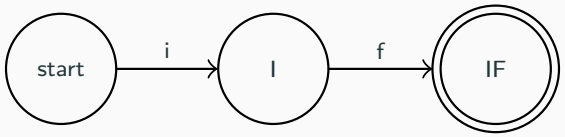

Figure 2: IF if

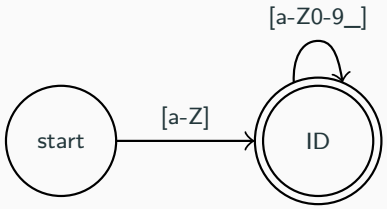

Figure 3: ID [a-Z][a-Z0-9_]*

- The full theory of Regular Expressions and Finite Automata is out of the scope of these lectures. Ressources for the curious student are in this week reading list.

Combining DFA

- Multiple DFA can be merged to produce a single DFA that does the full lexical analysis.

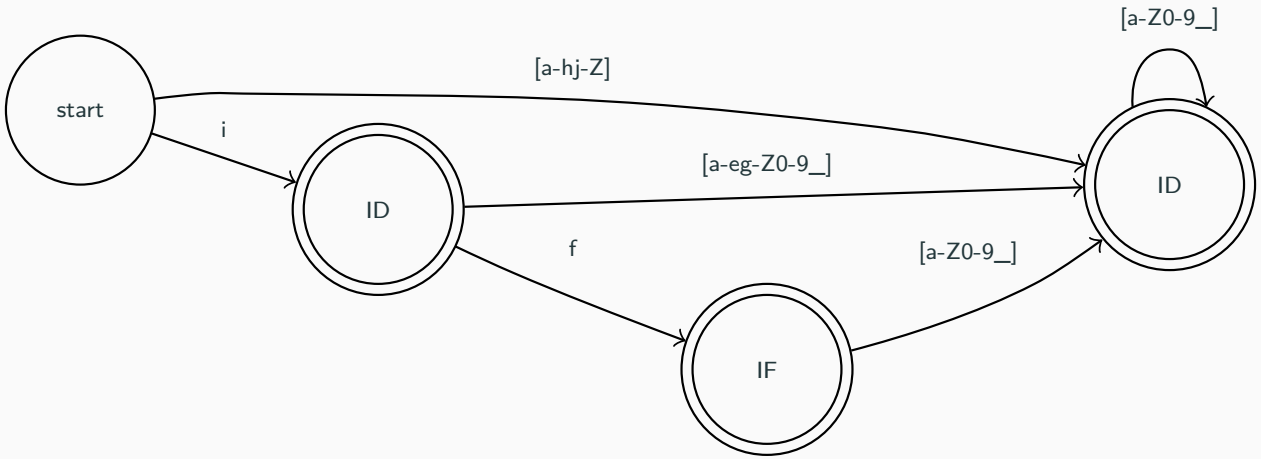

Figure 4: Merging IF and ID DFAs

Why use DFA ?

- Why use finite automata ?
  - Automata decides the category of a token or rejects it
  - Fast word recognition: the decision is done in $O(N)$ with N the length of the input
  - Compact rules representation

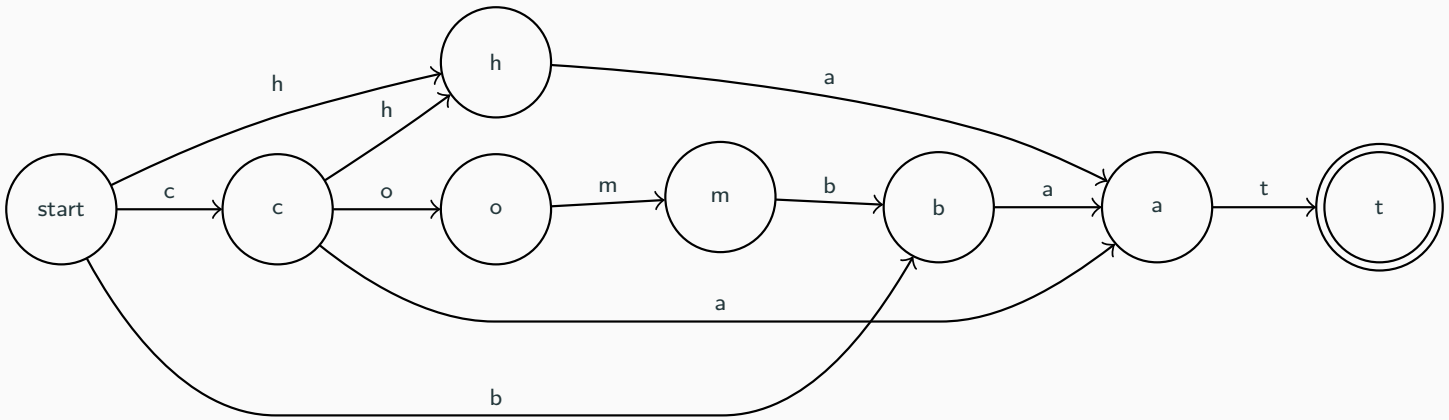

Figure 5: DFA for words in language {combat, chat, hat, cat, bat}

Flex

- *Flex* is a lexer generator
  - From a set of regular expressions and extra rules . . .
  - . . . *Flex* produces a DFA

Internal Token representation (see parser/tiger_parser.hh)

- Constant tokens such as `else` or `;` represented with,
  - a token integer code such as TOK_ELSE (280) or TOK_SEMICOLON (260)
  - a source location (useful to report localized errors)
- Variable tokens such as 42 or `my_variable` represented with,
  - a code such TOK_INT (295) or TOK_ID (293)
  - a source location
  - the variable content: in this case an `int` or a `std::string`
- Tokens are produced with calls to helper functions,

```
yy::tiger_parser::make_INT(42, loc);
```

Flex rules

- A Flex rule has two parts:
  1. a regular expression
  2. an action that usually produces a token

```
";" {
  return yy::tiger_parser::make_SEMICOLON(loc);
}
[a-zA-Z][_0-9a-zA-Z]* {
  return yy::tiger_parser::make_ID(Symbol(yytext), loc);
}
```

- Flex has helper variables and functions, for instance `yytext` contains the text matched by the regular expression

Flex sub-automata

- Sometimes it is useful to have different regular expression rules for different scenarios. For example, inside a comment usual rules do not apply: all text is ignored.
- Flex has support for sub-automata states which change the current set of rules.

Example of sub-automata

- The default state is called `INITIAL`
- To change states one calls `BEGIN(STATE)`
- Particular rules of a `STATE` must be declared inside a `<STATE>{ }` block.

```
"/*"      {comment_depth = 1; BEGIN(COMMENT);}
<COMMENT>{
    "/*" {comment_depth++;}
    "*/" {comment_depth--;
          if (comment_depth == 0) BEGIN(INITIAL);}
    <<EOF>> utils::error (loc, "unterminated comment");
    . {}
}
```