06 - Syntax Analysis

Dr. Robert Lowe

Division of Mathematics and Computer Science
Maryville College

Outline

1. Syntax Analysis

2. Example: L Programming Language

Outline

1 Syntax Analysis

2 Example: L Programming Language

Syntax Analyzer

- There are two main parts to syntax analysis:

Syntax Analyzer

- There are two main parts to syntax analysis:
    - **Lexing** - Process the micro-syntax of the language.

Syntax Analyzer

- There are two main parts to syntax analysis:
  - **Lexing** - Process the micro-syntax of the language.
  - **Syntax Analysis** - Process the context-free syntax of the language.

Syntax Analyzer

- There are two main parts to syntax analysis:
  - **Lexing** - Process the micro-syntax of the language.
  - **Syntax Analysis** - Process the context-free syntax of the language.
- The syntax analyzer can be created directly from the BNF specification of a language.

Lexical Analysis Abstraction

For now our lexer will consist of the following:

- A global variable `symbol`

Lexical Analysis Abstraction

For now our lexer will consist of the following:

- A global variable `symbol`
- **procedure** `next_symbol`

Lexical Analysis Abstraction

For now our lexer will consist of the following:

- A global variable `symbol`
- **procedure** `next_symbol`
    1. Place the next basic symbol in the global variable.

Lexical Analysis Abstraction

For now our lexer will consist of the following:

- A global variable `symbol`
- **procedure** `next_symbol`
    1. Place the next basic symbol in the global variable.
    2. Advance the input stream.

Lexical Analysis Abstraction

For now our lexer will consist of the following:

- A global variable `symbol`
- **procedure** `next_symbol`
  1. Place the next basic symbol in the global variable.
  2. Advance the input stream.
- **procedure** `mustbe(s)`

Lexical Analysis Abstraction

For now our lexer will consist of the following:

- A global variable `symbol`
- **procedure** `next_symbol`
    1. Place the next basic symbol in the global variable.
    2. Advance the input stream.
- **procedure** `mustbe(s)`
    1. if `s` is the `symbol`, call `next_symbol`

Lexical Analysis Abstraction

For now our lexer will consist of the following:

- A global variable `symbol`
- **procedure** `next_symbol`
  1. Place the next basic symbol in the global variable.
  2. Advance the input stream.
- **procedure** `mustbe(s)`
  1. if `s` is the `symbol`, call `next_symbol`
  2. Otherwise, report an error.

Lexical Analysis Abstraction

For now our lexer will consist of the following:

- A global variable `symbol`
- **procedure** `next_symbol`
  1. Place the next basic symbol in the global variable.
  2. Advance the input stream.
- **procedure** `mustbe(s)`
  1. if `s` is the `symbol`, call `next_symbol`
  2. Otherwise, report an error.
- **procedure** `have(s)`

Lexical Analysis Abstraction

For now our lexer will consist of the following:

- A global variable `symbol`
- **procedure** `next_symbol`
  1. Place the next basic symbol in the global variable.
  2. Advance the input stream.
- **procedure** `mustbe(s)`
  1. if `s` is the `symbol`, call `next_symbol`
  2. Otherwise, report an error.
- **procedure** `have(s)`
  1. if `s` is the `symbol`, call `next_symbol` and return `true`.

Lexical Analysis Abstraction

For now our lexer will consist of the following:

- A global variable `symbol`
- **procedure** `next_symbol`
  1. Place the next basic symbol in the global variable.
  2. Advance the input stream.
- **procedure** `mustbe(s)`
  1. if `s` is the `symbol`, call `next_symbol`
  2. Otherwise, report an error.
- **procedure** `have(s)`
  1. if `s` is the `symbol`, call `next_symbol` and return `true`.
  2. Otherwise, return `false`

Coding from BNF

- A production like this: `<  > ::= a<A>`

Coding from BNF

- A production like this: `< > ::= a<A>`
- Would be coded:

```
mustbe("a"); A();
```

Coding from BNF

- A production like this: `< > ::= a<A>`
- Would be coded:

```
mustbe(“a”); A();
```

- A production like this: `< > ::= a<A> | b<B>`

Coding from BNF

- A production like this: `< > ::= a<A>`
- Would be coded:
  `mustbe(“a”); A();`
- A production like this: `< > ::= a<A> | b<B>`
- Would be coded:
  `if have(“a”) then A() else { mustbe(“b”); B() }`

Coding from BNF

- A production like this: `< > ::= a<A>`
- Would be coded:
  `mustbe(“a”); A();`
- A production like this: `< > ::= a<A> | b<B>`
- Would be coded:
  `if have(“a”) then A() else { mustbe(“b”); B() }`
- A production like this: `< > ::= a<A> | b<B> | c <C>`

Coding from BNF

- A production like this: `< > ::= a<A>`
- Would be coded:
  `mustbe(“a”); A();`
- A production like this: `< > ::= a<A> | b<B>`
- Would be coded:
  `if have(“a”) then A() else { mustbe(“b”); B() }`
- A production like this: `< > ::= a<A> | b<B> | c <C>`
- Would be coded as:

```
if( have("a") ) { A(); }
else if( have("b") ) { B(); }
else { mustbe("c"); C(); }
```

Repetition Productions

`< > ::= <A> [b<A>]`⋆ (Where * means repeat “zero or more times”)

Repetition Productions

`< > ::= <A> [b<A>]`$\star$ (Where * means repeat “zero or more times”)

```
    do {
        A();
    } while(have("b"));
```

Outline

1 Syntax Analysis

2 Example: L Programming Language

The Grammar of L

```
⟨program⟩           ::=  <expression>
⟨expression⟩        ::=  <term> <expression-tail>
⟨expression-tail⟩   ::=  λ | ’+’ <term> <expression-tail>
⟨term⟩              ::=  <factor> <term-tail>
⟨term-tail⟩         ::=  λ | ’*’ <factor> <term-tail>
⟨factor⟩            ::=  <unit> | ’(’ <expression> ’)’
⟨unit⟩              ::=  ’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’ | ’8’ | ’9’
```

The Grammar of L

⟨*program*⟩ ::= <expression>

⟨*expression*⟩ ::= <term> <expression-tail>

⟨*expression-tail*⟩ ::= λ | ’+’ <term> <expression-tail>

⟨*term*⟩ ::= <factor> <term-tail>

⟨*term-tail*⟩ ::= λ | ’*’ <factor> <term-tail>

⟨*factor*⟩ ::= <unit> | ’(’ <expression> ’)’

⟨*unit*⟩ ::= ’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’ | ’8’ | ’9’

Activity: Let’s create a syntax analyzer for L!