Compiler Construction Plovdiv, Bulgaria 2006

Markus Löberbauer
Johannes Kepler Universität
Institut for System Software
Altenbergerstrasse 69, Linz 4040, Austria

Project

For every part of this exercise we will offer skeleton files on the course homepage. The skeleton files give you a structure for your compiler and contain TODO comments at the positions where you have work to do, describing what you have to do.

1 Scanner (Lexical Analyzer)

Start the work on your Z#-Compiler by implementing a lexical analyzer for the language Z#. Please study the language specification carefully. Think about terminal symbols and terminal classes for the given grammar. Check out the comment style, character constants und keywords.

Every call of Scanner.next() must return a terminal symbol (Token). At the end of the file every call of Scanner.next() results in an end of file (eof) token. Keywords, identifiers, (positive) integer numbers, character constants and operators must be recognized. Whitespaces like blanks, tabs, line breaks and comments must be skipped.

The following errors should be reported:

- Invalid character.
- Invalid character constant (e.g.: line break in character constant).

- Invalid escape sequences.
- Too big number literals (valid range: 0 to 2147483647, negative numbers are handled from the parser).

2 Parser (Syntax Analyzer)

Implement a recursive descend syntax analyzer for Z#. Use the skeleton files from the course homepage. Implement a method for every production in the grammar.

The error reporting should cover:

- Unexpected tokens (e.g.: 'Method must start either with void or a type name', 'Number or character constant expected').
- Try to make descriptive error messages.

3 Symbol Table

Download and complete the skeleton files for the symbol table and extend your parser such that a complete symbol table gets built up.

Additional error handling in this part:

- Missing Main method (every program needs an entry point).
- Type expected (e.g.: for variable or formal parameter declarations)
- Class type expected (in new statement).
- Too many arguments (a method must not have more than 256 parameters).
- Too many local variables (a method must not have more than 256 local variables).
- Name declared twice (a name must be unique per scope).
- Name not found.
- Field not found.

4 Code Generation

To complete the compiler we have to add code generation. While generating code we are able to detect more errors:

- Usage of an undefined symbol.
- Usage of an incompatible symbol.
- Main method must not have a return type (void expected).
- Main method must not have parameters.
- Integer expected (e.g.: on array creation and access; mathematic expressions).
- Integer variable expected (e.g.: on increment and decrement operator).
- Void methods must not be left with an return value (on return statement).
- Return value must be assignable to return type of the method (on return statement).
- Return value expected (on non void methods).
- Only int and char values can be read/written (on read/write).
- Not a method (on method calls, if the given item is not a method).
- Parameter type mismatch (if the actual parameter can not be assigned to the formal parameter).
- Incompatible types (on relations, e.g.: 1 ¡ 'r').
- Only == and != allowed on reference types (e.g.: myCar ¡ yourCar).
- Procedure (void function) called as function.
- Object expected (e.g.: myCar.speed is ok, but 123.speed is not).
- Array expected (e.g.: myCar[2]).