Compilers, kevät 2016

Mini-PL -tulkki

Projektin raportti

Noora Imberg
013449315

3.4.2016

Rakenne

Tulkissa on eroteltu omiin pakkauksiinsa parsinta (compiler.parsing), tulkkaus (compiler.interpreter) ja abstrakti syntaksipuu (compiler.ast.*). Edelleen parsinta-pakkauksessa on erikseen luokat Parserille ja Skannerille sekä luokka jossa määritellään Token.

Skanneri muuntaa annetun tiedoston tokeneiksi. Skanneri lukee tiedostosta aina seuraavan merkin ja päättelee sen perusteella, mitkä tokenit saattavat tulla kyseeseen. Suuri osa tokeneista on yhden merkin pituisia, joten ne tunnistetaan jo tässä. Joissakin tilanteissa on luettava vielä seuraavakin merkki, jotta saadaan eroteltua esimerkiksi ':' ja ':='. Muuttujat ja avainsanat erotetaan toisistaan vertaamalla luettua sanaa avainsanat sisältävään listaan. Skanneri muodostaa tokeneita, joilla on tieto niiden tyypistä, sisällöstä ja sijainnista lähdekooditiedostossa.

Parseri pyytää skannerilta aina viimeistä sen lukemaa tokenia. Parserin rakenne perustuu myöhemmin tässä tiedostossa esitettyyn kontekstittomaan kielioppiin. Tokeneista muodostetaan abstrakti syntaksipuu. Abstrakti syntaksipuu tukee Visitor-suunnittelumallin mukaista läpikäyntiä StatementVisitor ja ASTVisitor-rajapintojen kautta. Abstaktin syntaksipuun rakenne on esitelty myöhemmin tässä dokumentissa.

Testaus

Ohjelmaa on testattu lähinnä antamalla sille erilaisia valideja ja epävalideja mini-pl-ohjelmia ja tutkimalla sen toimintaa erilaisissa tilanteissa. Yksikkötestejä ei ole kirjoitettu ajan puutteen takia.

Puutteet ja parannusehdotukset

Ohjelma vaatisi lisää testausta, jotta sen voisi sanoa toimivan luotettavasti kaikissa tilanteissa. Yksikkötestit auttaisivat tässä merkittävästi. Lisäksi ohjelman virheidenkäsittelyä voisi parantaa merkittävästi. Joissakin tilanteissa voisi olla hyödyllistä, jos esimerkiksi puuttuva puolipiste ei tuhoaisi kaikkea, vaan ohjelma osaisi etsiä sen jälkeen seuraavan kelvollisen rakenteen ja jatkaa siitä. Useamman rivin kommentteja /* kommentti */ ei ole toteutettu.

Käyttöohjeet

Ohjelma vaatii Javan version 8. Ohjelman suoritus:
 java -jar compiler.jar tulkattavanTiedostonNimi

Mini-PL-kieli

Tokenit

Mini-Pl-kielestä löytyvät tokenit on esitetty taulukossa. Muuttujat erotaan avainsanoista tarkastamalla löytyykö tietty merkkijono ennalta määriteltyjen avainsanojen listasta.

muuttujat ja avainsanat	[a-zA-Z][0-9a-zA-Z_]*
integer-literaali	[0-9][0-9]*
string-literaali	"([^"\\]|\\[\\"])*"
operaattorit	+ | - | * | / | < | = | & | ! | (|) | ; |: | := | ..

Kontekstiton kielioppi

Projektin kuvauksessa esitetty kielioppi ei ole LL(1)-muotoinen, joten sitä oli muokattava, jotta saatiin toteutettua rekursiivisesti etenevä jäsentäjä. Muokattu kielioppi on seuraavanlainen:

```
 <prog>   ::=  <stmts>
 <stmts>  ::=  <stmt> ";" <stmts-tail>
<stmts-tail> ::= <stmts>
             | ε
 <stmt>    ::=  "var" <var_ident> ":" <type> <opt-assign>
             |   <var_ident> ":=" <expr>
             |   "for" <var_ident> "in" <expr> ".." <expr> "do" <stmts> "end" "for"
             |   "read" <var_ident>
             |   "print" <expr>
             |   "assert" "(" <expr> ")"
<opt-assign> ::=  ":=" <expr>
             |  ε
 <expr>   ::=  <opnd> <expr-tail>
            |    <unary_op> <opnd>
<expr-tail> ::= <op> <opnd>
```

```
              | ε
<opnd>   ::=  <int>
            |   <string>
            |   <var_ident>
            |   "(" expr ")"
<type>   ::=  "int" | "string" | "bool"
<var_ident> ::= <ident>

<reserved keyword> ::=
          "var" | "for" | "end" | "in" | "do" | "read" |
          "print" | "int" | "string" | "bool" | "assert"
```

Abstrakti syntaksipuu

Abstraktin syntaksipuun muodostavat luokat on jaoteltu seuraavasti:

Expressions
- BinaryExpression
- UnaryExpression

Operands
- ExpressionOperand
- IntegerLiteralOperand
- StringLiteralOperand
- VariableIdentifierOperand

Statements
- AssertStatement
- AssignStatement
- BlockStatement
- ForStatement
- PrintStatement
- ReadStatement
- VariableDeclarationStatement

Virheiden käsittely

Skannerin havaitsemia virheitä ovat tilanteet, joissa ei muodostu mitään olemassa olevaa tokenia. Kelvolliset tokenit on esitetty aiemmin tässä dokumentissa kohdassa Tokenit. Virheellisistä tokeneista heitetään virheilmoitus. Lisäksi skanneri havaitsee tiedoston

loppumisen, joka ei varsinaisesti ole virhe, mutta poikkeuksellinen tilanne joka tapauksessa. Skanneri lisää loppumisen havaitessaan tiedoston lopetustokenin.

Parseri havaitsee virheet, joissa tokeneista ei muodostu mitään tunnettua stamenttia eli tilanteet, joissa saadaan jokin token väärässä paikassa. Ohjelma heittää virheilmoituksen havaitessaan tällaisen virheen.

Semanttisessa analyysissa havaittavia virheitä ovat väärät tyypit operaatioissa.

Kaikista edellä mainituista virheistä näytetään käyttäjälle virheilmoitus, ja ohjelmaa ei suoriteta.

Tulkkaamisen yhteydessä havaittava virhe on esimerkiksi nollalla jakaminen, jota ei eksplisiittisesti käsitellä, mutta josta aiheutuu Javan normaalin poikkeuksen heittäminen.

Liitteet

Tehtävänanto:

Compilers **Project 2016:** ***Mini-PL interpreter*** **(19.2.2016)**

Implement an *interpreter* for the Mini-PL programming language. The language analyzer must correctly recognize and process all valid (and invalid) Mini-PL programs. It should report syntactic errors, and then continue analyzing the rest of the source program. It must also construct an AST and make necessary passes over this program representation. The semantic analysis part binds names to their declarations, and checks semantic constraints, e.g., expressions types and their correct use. If the given program was found free from errors, the interpreter part will immediately execute it.

Implementation requirements and grading criteria

The assignment is done as individual work. When building your interpreter, you are expected to properly use and apply the compiler techniques taught and discussed in the lectures and exercises. The Mini-PL analyzer is to be written purely in a *general-purpose programming language*. C# is to be used as the implementation language, by default (if you have problems, please consult the teaching assistant). **Language-processing tools (language recognizer generators, regex libraries, translator frameworks, etc.) are not allowed**. Note that you can of course use the basic general data structures of the implementation language - such as strings and string builders, lists/arrays, and associative tables (dictionaries, maps). You must yourself make sure that your system can be run on the development tools available at the CS department.
The emphasis on one part of the grading is the quality of the implementation: especially its overall architecture, clarity, and modularity. Pay attention to programming style and

commenting. Grading of the code will consider (undocumented) bugs, level of completion, and its overall success (solves the problem correctly). Try to separate the general (and potentially reusable) parts of the system (text handling and buffering, and other utilities) from the source-language dependent issues.

Documentation

Write a report on the assignment, as a document in PDF format. The title page of the document must show appropriate identifications: the name of the student, the name of the course, the name of the project, and the valid date and time of delivery.
Describe the overall architecture of your language processor with, e.g., UML diagrams. Explain your diagrams. Clearly describe yout testing, and the design of test data. Tell about possible shortcomings of your program (if well documented they might be partly forgiven). Give instructions how to build and run your interpreter. The report must include the following parts

1. The Mini-PL token patterns as *regular expressions* or, alternatively, as *regular definitions*.
2. A *modified context-free grammar* suitable for recursive-descent parsing (eliminating any LL(1) violations); modifications must not affect the language that is accepted.
3. Specify *abstract syntax trees* (AST), i.e., the internal representation for Mini-PL programs; you can use UML diagrams or alternatively give a syntax-based definition of the abstract syntax.
4. *Error handling* approach and solutions used in your Mini-PL implementation (in its scanner, parser, semantic analyzer, and interpreter).

For completeness, include the original project definition and the Mini-PL specification as appendices of your document; you can refer to them when explaining your solutions.

Delivery of the work

The final delivery is due at 23 o'clock (11 p.m.) on **Sunday 3rd of April**, 2016. After the appointed deadline, the maximum points to be gained for a delivered work diminishes linearly, decreasing two (2) points per each hour late.
The work should be returned to the exercise assistant via e-mail, in a zip form. This zip (included in the e-mail message) should contain all relevant files, within a directory that is named according to your unique department user name. The deliverable zip file must contain (at least) the following subfolders.

```
<username>
  ./doc
  ./src
```

When naming your project (.zip) and document (.pdf) files, always include your CS user name and the packaging date. These constitute nice unique names that help to identify the files later. Names would be then something like:

```
project zip: username_proj_2016_20_3.zip
document:  username_doc_2016_20_3.pdf
```

More detailed instructions and the requirements for the assignment are given in the exercise group. If you have questions about the folder structure and the ways of delivery, or in case

you have questions about the whole project or its requirements, please contact the teaching assistant (Jiri Hamberg).

Kielen määrittely:

Syntax and semantics of Mini-PL (9.2.2016)

Mini-PL is a simple programming language designed for pedagogic purposes. The language is purposely small and is not actually meant for any real programming. Mini-PL contains few statements, arithmetic expressions, and some IO primitives. The language uses static typing and has three built-in types representing primitive values: **int**, **string**, and **bool**. The BNF-style syntax of Mini-PL is given below, and the following paragraphs informally describe the semantics of the language.

Mini-PL uses a single global scope for all different kinds of names. All variables must be declared before use, and each identifier may be declared once only. If not explicitly initialized, variables are assigned an appropriate default value.

The Mini-PL **read** statement can read either an integer value or a single *word* (string) from the input stream. Both types of items are whitespace-limited (by blanks, newlines, etc). Likewise, the **print** statement can write out either integers or string values. A Mini-PL program uses default input and output channels defined by its environment. Additionally, Mini-PL includes an **assert** statement that can be used to verify assertions (assumptions) about the state of the program. An **assert** statement takes a **bool**argument. If an assertion fails (the argument is *false*) the system prints out a diagnostic message.
The arithmetic operator symbols '+', '-', '*','/' represent the following functions:

```
 "+" : (int, int) -> int          // integer addition
  "-" : (int, int) -> int          // integer subtraction
  "*" : (int, int) -> int          // integer multiplication
  "/" : (int, int) -> int          // integer division
```

The operator '+' *also* represents string concatenation (i.e., this operator symbol is overloaded):

```
 "+" : (string, string) -> string   // string concatenation
```

The operators '&' and '!' represent logical operations:

```
 "&" : (bool, bool) -> bool        // logical and
  "!" : (bool) -> bool             // logical not
```

The operators '=' and b '<' are overloaded to represent the comparisons between two values of the same type T (**int**, **string**, or **bool**):

```
 "=" : (T, T) -> bool             // equality comparison
  "<" : (T, T) -> bool             // less-than comparison
```

A **for** statement iterates over the consequent values from a specified integer range. The expressions specifying the beginning and end of the range are evaluated once only (at the beginning of the **for** statement). The **for** control variable behaves like a constant inside the loop: it cannot be assigned another value (before exiting the **for** statement). A control

variable needs to be declared before its use in the **for** statement (in the global scope). Note that loop control variables are *not* declared inside **for** statements.

Context-free grammar for Mini-PL

The syntax definition is given in so-called *Extended Backus-Naur* form (EBNF). In the following Mini-PL grammar, the notation X* means 0, 1, or more repetitions of the item X. The '|' operator is used to define alternative constructs. Parentheses may be used to group together a sequence of related symbols. Brackets ("[" "]") may be used to enclose optional parts (i.e., zero or one occurrence). Reserved keywords are marked bold (as "**var**"). Operators, separators, and other single or multiple character tokens are enclosed within quotes (as: "**..**"). Note that nested expressions are always fully parenthesized to specify the execution order of operations.

```
<prog>   ::=  <stmts>
 <stmts>  ::=  <stmt> ";" ( <stmt> ";" )*
 <stmt>   ::=  "var" <var_ident> ":" <type> [ ":=" <expr> ]
         |   <var_ident> ":=" <expr>
         |   "for" <var_ident> "in" <expr> ".." <expr> "do"
               <stmts> "end" "for"
         |   "read" <var_ident>
         |   "print" <expr>
         |   "assert" "(" <expr> ")"

 <expr>   ::=  <opnd> <op> <opnd>
         |   [ <unary_op> ] <opnd>

 <opnd>   ::=  <int>
         |   <string>
         |   <var_ident>
         |   "(" expr ")"

 <type>   ::=  "int" | "string" | "bool"
 <var_ident> ::= <ident>

 <reserved keyword> ::=
            "var" | "for" | "end" | "in" | "do" | "read" |
            "print" | "int" | "string" | "bool" | "assert"
```

Lexical elements

In the syntax definition the symbol *<ident>* stands for an identifier (name). An identifier is a sequence of letters, digits, and underscores, starting with a letter. Uppercase letters are distinguished from lowercase.

In the syntax definition the symbol <*int*> stands for an integer constant (literal). An integer constant is a sequence of decimal digits. The symbol <*string*> stands for a string literal. String literals follow the C-style convention: any special characters, such as the quote character (") or backslash (\), are represented using escape characters (e.g.: \").

A limited set of operators include (only!) the ones listed below.

'+' | '-' | '*' | '/' | '<' | '=' | '&' | '!'

In the syntax definition the symbol <*op*> stands for a binary operator symbol. There is one unary operator symbol (<*unary_op*>): '!', meaning the logical *not* operation. The operator symbol '&' stands for the logical *and* operation. Note that in Mini-PL, '=' is the*equal* operator - not assignment.

The predefined type names (e.g.,"**int**") are reserved keywords, so they cannot be used as (arbitrary) identifiers. In a Mini-PL program, a comment may appear between any two tokens. There are two forms of comments: one starts with "/*", ends with "*/", can extend over multiple lines, and may be nested. The other comment alternative begins with "//" and goes only to the end of the line.

Sample programs

```
var X : int := 4 + (6 * 2);
    print X;
```

```
var nTimes : int := 0;
    print "How many times?";
    read nTimes;
    var x : int;
    for x in 0..nTimes-1 do
        print x;
        print " : Hello, World!\n";
    end for;
    assert (x = nTimes);
```

```
    print "Give a number";
    var n : int;
    read n;
    var v : int := 1;
    var i : int;
    for i in 1..n do
        v := v * i;
    end for;
    print "The result is: ";
    print v;
```