Project (part 2): Lexical and Syntactic Analysis
Due Date: April 22, 2011: 11:59 PM.

1 Overview

This course requires you to write a compiler that takes a program written in a language, Java^{--}, and constructs an equivalent program written in assembly instructions for MIPS architecture family.

Java^{--} is a subset of the Java programming language that inherits types, methods, expressions, and the notion of single inheritance from Java. However, it does not include more complex concepts such as concurrency, exception handling, packaging mechanism and interfaces. Further, it includes only a subset of Java's operators and expression building mechanisms. Finally, we have removed arrays from the language, since their implementation is very similar to that of classes. The primary objective in designing this project has been to select those features that will facilitate your understanding of how specific language features can be implemented in a compiler.

This document first describes the lexical structure of the tokens of Java^{--}, followed by its syntactic structure. The semantics of the various constructs will be described in a separate document.

2 Lexical Structure

Java^{--} derives much of its lexical structure from Java. The different tokens of the language are described below:

2.1 Comments and White Space

Spaces, end of line, and comments may occur anywhere in a program except within a basic symbol. At least one space, end of line or comment must occur between any two adjacent identifiers or constants.

Java^{--} supports single-line comments that begin with "//" and terminate at the end of the line.

2.2 Reserved words

Java^{--} reserves the following identifiers:

```
boolean     char     class      continue     else
extends     false    if         input        output   int
new         null     return     super        string   this
true        while    void
```

2.3 Constants

The language supports the following set of constants:

1. `Integers`: Integer constants denote any number of digits between 0 and 9.

2. `Char`: Char constants begin and end with a single quote (') and may contain any ascii characters. In addition, the following (non)printable characters must be escaped using the following convention:

Character	Escape
Newline	`\n`
Tabulator	`\t`
Backspace	`\b`
Form feed	`\f`
\	`\\`
"	`\"`

3. Strings: Strings begin and end with a double quote (") and may contain any sequence of characters including newlines.

2.4 Identifiers

An identifier is a sequence of letters, digits and the underscore character "_". It must start with a letter, and cannot be one of the reserved words.

2.5 Operators

Java^{--} defines the following operators:

```
,     .     ;     (     )     {     }     +
-     *     /     =     &&    ||    !     >
<     ==    !=    >=    <=
```

3 Syntax Description

We use the EBNF notation to describe the syntax of Java^{--}. A few notes about the notation:

- Nonterminals begin with uppercase letters.
- Terminals that are grammar symbols ('[' for instance) or other non-identifier symbols (';', '{', '*', '+', etc.) are enclosed in single quotes ' '.
- Keywords (such as "class" and "boolean") are represented directly by their corresponding string representations.
- Certain tokens such as identifier, integer and string constants are represented respectively by `id`, `integer_constant` and `string_constant` tokens. Your lexical analyzer will return them as tokens.

3.1 Operator precedences and associativity

The rules of composition specify operator *precedences*. The operators (from lowest precedence to the highest) are:

```
||
&&
==  !=
<   >       <=     >=
+   -
*   /
!   Unary -  Unary +
```

All operators other than !, `Unary -` and `Unary +` are left associative. Operators !, Unary - and Unary + are right associative.

3.2 EBNF representation

The following describes the syntactic structure of Java^{--}. Program is the start symbol for the grammar.

```
Program ::=  ClassDeclaration*
ClassDeclaration ::=  class id [Extends] ClassBody
Extends ::=  extends ClassType
ClassBody ::=  '{' ClassBodyDeclaration* '}'
ClassBodyDeclaration ::=  ClassMemberDeclaration
ClassMemberDeclaration ::=  FieldDeclaration
        | MethodDeclaration
Type ::=  PrimitiveType
        | ReferenceType
PrimitiveType ::=  int
        | boolean
        | char
        | string
ReferenceType ::=  ClassType
ClassType ::=  SimpleName
Name ::=  SimpleName
        | QualifiedName
SimpleName ::= id
QualifiedName ::=  Name '.' id
FieldDeclaration ::=  Type VariableDeclarators ';'
VariableDeclarators ::=  VariableDeclarator (',' VariableDeclarator)*
VariableDeclarator ::=  id
MethodDeclaration ::=  MethodHeader MethodBody
MethodHeader ::=  Type MethodDeclarator
        |  void MethodDeclarator
MethodDeclarator ::=  id '(' [FormalParameterList] ')'
FormalParameterList ::=  FormalParameter (',' FormalParameter)*
FormalParameter ::=  Type id
MethodBody ::=  '{' LocalVariableDeclarationStatement* Statement* '}'
        | ';'
LocalVariableDeclarationStatement ::=  Type VariableDeclarators ';'
Statement ::=  IfThenStatement
        | IfThenElseStatement
```

```
            | WhileStatement
            | SimpleBlock
            | EmptyStatement
            | ExpressionStatement
            | ContinueStatement
            | ReturnStatement
            | IOStatement
    SimpleBlock ::=  '{' Statement* '}'
    EmptyStatement ::=  ';'
    ExpressionStatement ::=  StatementExpression ';'
    StatementExpression ::=  Assignment
            | MethodInvocation
            | ClassInstanceCreationExpression
    IfThenStatement ::=  if  '(' Expression ')' Statement
    IfThenElseStatement ::=  if '(' Expression ')' Statement
                else  Statement
    WhileStatement ::= while '(' Expression ')' Statement
    ContinueStatement ::=  continue ';'
    ReturnStatement ::=  return  [Expression]  ';'
    IOStatement ::= (input | output) Expression ';'
    Primary ::=   Literal
            | this
            | '(' Expression ')'
            | ClassInstanceCreationExpression
            | FieldAccess
            | MethodInvocation
    ClassInstanceCreationExpression ::= new ClassType '('[ArgumentList] ')'
    ArgumentList ::= Expression (',' Expression)*
    FieldAccess ::=  Primary '.' id
            | super '.' id
    MethodInvocation ::=  Name '(' [ArgumentList] ')'
            | Primary '.' id  '(' [ArgumentList] ')'
            | super '.' id '(' [ArgumentList]')'
    PrimitiveExpression ::=  Primary
            | Name
    Expression ::=  Expression '*' Expression
            | Expression '/' Expression
            | Expression '+' Expression
            | Expression '-' Expression
            | Expression '&&' Expression
            | Expression '||' Expression
            | Expression '==' Expression
            | Expression '!=' Expression
            | Expression '<' Expression
            | Expression '>' Expression
            | Expression '<=' Expression
            | Expression '>=' Expression
```

```
            | Assignment
            | '-' Expression
            | '+' Expression
            | '!' Expression
            | PrimitiveExpression
    Assignment ::=  LeftHandSide '=' Expression
    LeftHandSide ::=  Name
            | FieldAccess
    Literal ::=  integer_constant
            | string_constant
            | null
            | true
            | false
```

4 Project Description

For this phase of the project, you will implement a lexical analyzer and a parser for Java^{--}. Use `Lex` to construct the lexical analyzer, and `Yacc` to construct the syntactic analyzer. The output of your compiler will mainly involve reporting any lexical or parsing errors. Your lexical analyzer must detect the following set of errors:

1. unterminated character strings
2. unterminated comments, and
3. illegal characters.

Your parser should also recognize any syntactic errors. It should handle errors by reporting the error along with the line number where the error occurred and then exit. In other words, your compiler will only report the first error in the source program.

The grammar provided above can be used with Yacc with minor modifications. It has been presented in an expanded form to make it clearer. Feel free to modify or optimize it, as long as it generates the same language.