CMSC 22600 **Compilers for Computer Languages** **Project 1**
Autumn 2016 **October 6, 2016**

SOOL parser and scanner

Due: October 21, 2016

1 Introduction

Your first assignment is to implement a scanner and a parser for SOOL, which will convert an input stream of characters into a parse tree. You will use the ML-ULex scanner generator and ML-Antlr parser generator (collectively the *ML Language Processing Tools*) for this assignment. These tools documented in the ML-LPT Manual, which is linked to on the course web site.

2 Requirements

We will seed a directory, called `proj1`, in your `phoenixforge` repository. This directory will contain five sub-directories:

`bin` — holds the `soolc.sh` script for running the compiler.

`common` — common utility code that will be used across the project

`driver` — main program for connecting phases and running the compiler

`parse-tree` — the representation of the parse tree

`parser` — the scanner and parser

as well as a `Makefile`.

Your task is to complete the `sool.lex` and `sool.grm` files that are located in the `parser` directory so as to implement the grammar of SOOL described below.

We recommend that you proceed by first writing the parser specification *without* actions and get that to pass the ML-Antlr compiler. Once you have done that step, you can write and test the lexer (the lexer depends on the token datatype that is generated by ML-Antlr). The last step will then be to add actions to the ML-Antlr specification that build the parse tree.

3 The SOOL scanner

You will use ML-ULex to implement the scanner for SOOL. Your scanner should handle the lexical properties of the language as described in Section 5.1.

4 The SOOL parser

ML-Antlr utilizes a parsing algorithm that integrates automatic error repair. Hence, your parser specification need not explicitly support error reporting. ML-Antlr does support declarations for improving error recovery, which you are welcome to include in your specification. The automatic error repair mechanisms require that semantic actions be free of significant side effects, because error repair may require executing a production's semantic action multiple times. All of the functions in the `ParseTree` structure are pure; thus, they may be freely used in semantic actions.

In order to support error reporting in the type-checker (to be implemented in Project 2), the parse tree must be annotated with position information. Therefore, each node in the parse tree is constructed with a *source span* that pairs the left and right source positions of the node. For example, you might have the following rule for matching a constant as an *AtomicExp* in your grammar:

```
AtomicExp : NUMBER
        => (PT.ExpMark{span = NUMBER_SPAN, tree = PT.IntExp NUMBER});
```

Source positions and spans of terminals are automatically provided by the scanner. Consult the ML-LPT manual for information for more information about source positions and accessing position information in semantic actions.[1]

5 The collected syntax of SOOL

5.1 Lexical issues

There are four classes of tokens in SOOL:

1. identifiers: `a`, `b`, `toString`, `y23`, `X`, `Foo`, *etc.*
2. delimiters and operators: **`(`**, **`)`**, **`=`**, **`<=`**, **`+`**, *etc.*
3. numbers: `0`, `42`, *etc.*
4. strings: `"hello world"`, `"some\ntext"`, *etc.*

Tokens can be separated by *whitespace* and/or *comments*.

Identifiers in SOOL can be any string of letters, digits, and underscores, beginning with a letter. Identifiers are case-sensitive (*e.g.*, `foo` is different from `Foo`). The following identifiers are reserved as keywords:

```
bool    class    else  extends   false     if
int   interface  meth    nil    override  return
self   string    true    var      void    while
```

The delimiters and operators of SOOL are the following:

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

[1] There is a link to the ML-LPT manual on the class website.

Numbers in SOOL are represented as sequences of decimal digits. Negative numbers are using the unary negation operator ("`-`"); the minus sign is not part of the literal token.

String literals are delimited by matching double quotes and can contain spaces (ASCII code 32) and any unescaped graphical character except "`\`" (ASCII code 92) or "`"`" (ASCII code 34). In addition, the following C-like escape sequences are permitted:

`\n`	—	newline (ASCII code 10)
`\r`	—	carriage return (ASCII code 13)
`\t`	—	horizontal tab (ASCII code 8)
`\\`	—	backslash
`\"`	—	quotation mark

A character in a string literal may also be specified by its numerical value using the escape sequence '`\`*ddd*,' where *ddd* is a sequence of three decimal digits. Strings in SOOL may contain any 8-bit value other than zero; thus '`\000`' is *not* a legal escape code.

Comments in SOOL are either *end-of-line*, which start with "`//`" and extend until the next newline character (or end-of file), or *block* comments, which start with "`/*`" and are terminated with a matching "`*/`." Note that block comments do not nest and that comments cannot start inside a string literal.

Whitespace is any non-empty sequence of spaces (ASCII code 32), horizontal or vertical tabs, form feeds, newlines, or carriage returns. Any other non-printable character is treated as an error.

5.2 Grammar

The collected syntax of SOOL given below using an extended-BNF format. Literal symbols, such as keywords and punctuation, are written in a **`bold fixed-width font`**, other terminal symbols are written in roman font, and non-terminal symbols are written in *italic font*. We use the following terminal symbols in the grammar:

Terminal	Lexical class	Description
tyid	identifier	class or interface name
varid	identifier	local-variable, field, or method name
num	number	integer literal
str	string	string literal

Note that tyid and varid have the same lexical specification (*i.e.*, the are both identifiers). The collected syntax of SOOL is presented in Figures 1 (declarations) and 2 (statements and expressions). We use parentheses for grouping, when necessary, and superscript "*" for zero-or-more items, superscript "$^+$" for one-or-more items, and superscript "^{opt}" for zero-or-one items.

5.3 Syntactic conventions

The syntax of expressions in Figure 2 is ambiguous, but we resolve the ambiguities by specifying the precedence and associativity of operators.[2] Expression forms are grouped into seven precedence levels from lowest to highest as follows:

[2] You will also need to do some factoring of the grammar to make it satisfy the LL(k) requirements of ML-Antlr.

Prog
::= *TopDcl*$^{+}$

TopDcl
::= **class** tyid **(** *Params*^{opt} **)** (**extends** tyid **(** *Args*^{opt} **)**^{opt} { *VarDcl** *MethDcl** }
| **interface** tyid (**extends** tyid)^{opt} { *MethSpc** }

Params
::= varid **:** *Type* (**,** varid **:** *Type*)*

Type
::= *TypeCon*
| *TypeCon* **?**

TypeCon
::= tyid
| **int**
| **bool**
| **string**

VarDcl
::= **var** varid **:** *Type* **=** *Exp*

MethDcl
::= **override**^{opt} **meth** varid **(** *Params*^{opt} **) ->** *ReturnType Block*

MethSpc
::= **meth** varid **(** (*Type* (**,** *Type*)*)^{opt} **) ->** *ReturnType*

ReturnType
::= *Type*
| **void**

Figure 1: SOOL grammar: declarations

```
Block
    ::=   { Stm* }

 Stm
    ::=   var varid := Exp ;
      |   while Exp Block
      |   if Exp Block (else if Exp Block)* (else Block)opt
      |   return Expopt ;
      |   Exp := Exp ;
      |   ReqMember ( Argsopt ) ;

ReqMember
    ::=   Exp . varid
      |   Exp ! varid

Args
    ::=   Exp (, Exp)*

 Exp
    ::=   Exp || Exp
      |   Exp && Exp
      |   Exp < Exp
      |   Exp <= Exp
      |   Exp == Exp
      |   Exp != Exp
      |   Exp @ Exp
      |   Exp + Exp
      |   Exp - Exp
      |   Exp * Exp
      |   Exp / Exp
      |   - Exp
      |   tyid ( Argsopt )
      |   ReqMember ( ( Argsopt ) )opt
      |   Exp ? varid ( ( Argsopt ) )opt
      |   Exp !
      |   self
      |   varid
      |   num
      |   str
      |   true
      |   false
      |   nil TypeCon
      |   ( Exp )
```

Figure 2: SOOL grammar: statements and expressions

- conditional operators: `||` and `&&`
- relational operators: `==`, `!=`, `<`, and `<=`
- string-concatenation operator: `@`
- addition operators: `+` and `-`
- multiplication operators: `*` and `/`
- unary negation: `-`
- field/method selection: `.`, `?`, and `!`

And all binary operators are left associative, but unary negation is right associative.

The type checker will restrict the left-hand-side of an assignment to be a specification of either a local variable (varid) or an expression that denotes an object's member variable. If you wish, you can enforce this restriction (or a weaker form) in your grammar. The more restrictive syntactic rules for the left-hand-side of an assignment (*LHSExp*) are as follows:

LHSExp
::= varid
| *LHSObj* (*Selector Member*)* *Selector* varid

LHSObj
::= varid
| **self**
| tyid **(** *Args*^{opt} **)**

Selector
::= **.**
| **!**

Member
::= varid (**(** *Args*^{opt} **)**)^{opt}

This approach leads to a more complicated grammar, but rules out certain nonsensical assignments, such as "`(1+2):=3;`", earlier in the compilation process. If you follow this approach, you should also apply it to the syntax of method-call statements and handle assignments and calls with an integrated set of productions.

6 Submission

We will collect the projects at **11pm** on **Friday October 21** from the SVN repositories, so make sure that you have committed your final version before then.

Important note: You are expected to submit code that **compiles** and that is well documented. Remember that points for project code are assigned 30% for coding style (documentation, choice of variable names, and program structure), and 70% for correctness. Code that does not compile will **not** receive any points for correctness.

7 Document history

October 16, 2016 Added discussion about an alternative syntax for the left-hand-side of assignments.

October 13, 2016 Fix bug in grammar; the rule for *VarDcl* was missing the initialization expression.

October 13, 2016 Fix bug in grammar; the rule for `return` statements was missing a terminating semicolon.

October 9, 2016 Add type constructor to `nil` expression to simplify Project 2.

October 7, 2016 Added some missing details about string literals.

October 6, 2016 Original version.