P1: Lexical Analysis and Parsing

Compiler Construction
2013-2014

Preliminaries

We describe the SPL, for Simple Programming Language grammar and give several examples. SPL is a simple strict and first order language with polymorphic functions. It is a simplified version of C extended with lists and 2-tuples that are common in all modern functional programming languages. The only basic types are integers, **Int**, and Booleans, **Bool**.

Grammar

A program in SPL is defined by the grammar SPL:

```
SPL     = Decl+
Decl    = VarDecl
        | FunDecl
VarDecl = Type id '=' Exp ';'
FunDecl = RetType id '(' [ FArgs ] ')' '{' VarDecl* Stmt+ '}'
RetType = Type
        | 'Void'
Type    = 'Int'
        | 'Bool'
        | '(' Type ',' Type ')'
        | '[' Type ']'
        | id
FArgs   = [ FArgs ',' ] Type id
Stmt    = '{' Stmt* '}'
        | 'if' '(' Exp ')' Stmt [ 'else' Stmt ]
        | 'while' '(' Exp ')' Stmt
        | id Field '=' Exp ';'
        | FunCall ';'
        | 'return' [ Exp ] ';'
Exp     = id Field
        | Exp Op2 Exp
        | Op1 Exp
        | int
        | 'False' | 'True'
        | '(' Exp ')'
        | FunCall
        | '[]'
        | '(' Exp ',' Exp ')'
Field   = [ Field ( '.' 'hd' | '.' 'tl' | '.' 'fst' | '.' 'snd' ) ]
FunCall = id '(' [ ActArgs ] ')'
ActArgs = Exp [ ',' ActArgs ]
Op2     = '+' | '-' | '*' | '/' | '%'
        | '==' | '<' | '>' | '<=' | '>=' | '!='
        | '&&' | '||'
        | ':'
Op1     = '!' | '-'
int     = [ '-' ] digit+
id      = alpha ( '_' | alphaNum)*
```

The language SPL has the predefined functions **print** and **isEmpty**. These functions have a single argument. The binary operators $+$, $-$, $*$, $/$, and %, have type **Int**$\times$**Int**$\rightarrow$**Int**. The comparison operators ==, <, >, <=, >=, and !=, have type **Int**$\times$**Int**$\rightarrow$**Bool**. The binary logical operators && and || have type **Bool**$\times$**Bool**$\rightarrow$**Bool**. The cons-operator, :, has type t$\times$[t]$\rightarrow$[t]. The unary negation operator, !, has type **Bool**$\rightarrow$**Bool** and the unary minus operator, $-$, has type **Int**$\rightarrow$**Int**. All operators have to obey the usual binding powers and directions.

The function **isEmpty** yields **True** iff the given list is empty. Its type is [t]$\rightarrow$**Bool**. The polymorphic **print** function has type t$\rightarrow$**Void**. It is able to print any value.

Comments on a single line start with // and end with the first newline character. Comments on multiple lines are delimited by /* and */. It is not demanded that comments can be nested.

Semantics

SPL is a strict language. This implies that function arguments are always evaluated. Since there are side effects the evaluation order of function arguments and the arguments of operators can influence the result of a program. You are free to make any choice that fits your implementation well. You should document and motivate your choice in exercise 3.

Integer and boolean function arguments are plain values. The argument behaves like a local value inside the function. List and tuple arguments behave like arguments in Java or by-reference arguments in C++. Such an argument is a reference to the list or tuple object.

The syntax fragments '(' Exp ',' Exp ')' and Exp ':' Exp to create tuples or list nodes always make new objects. Compared to Java the keyword new is missing or implicit.

1 Example

This example illustrates a number of constructs in SPL as well as the two allowed styles of comments.

```
/*
    Three ways to implement the factorial function in SPL.
    First the recursive version.
*/
Int facR ( Int n )
{
    if ( n < 2 )
        return 1;
    else
        return n * facR ( n - 1 );
}

// The iterative version of the factorial function
Int facI ( Int n )
{
    Int r = 1;
    while ( n > 1 )
    {
        r = r * n;
        n = n - 1;
    }
    return r;
}

// A main function to check the results
Void main ()
{
    Int n = 0;
    Int facN = 1;
    Bool ok = True;
    while ( n < 20 )
    {
        facN = facR ( n );
        if ( facN != facI ( n ) || facN != facL ( n ))
        {
            print ( n : facN : facI ( n ) : facL ( n ) : [] );
            ok = False;
        }
        n = n + 1;
    }
    print ( ok );
}

// A list based factorial function
// Defined here to show that functions can be given in any order (unlike C)
Int facL ( Int n )
{
    return product (fromTo ( 1, n ));
}

// Computes the product of a list of integers
Int product ( [ Int ] list )
{
    if ( isEmpty ( list ))
        return 1;
    else
        return list.hd * product ( list.tl );
}

// Generates a list of integers from the first to the last argument
[Int] fromTo (Int from, Int to)
{
    if ( from <= to )
        return from : fromTo ( from + 1, to );
    else
        return [];
}

// Make a reversed copy of any list
[t] reverse ( [t] list )
{
    [t] accu = [];
    while ( ! isEmpty ( list ))
    {
        accu = list.hd : accu ;
        list = list.tl;
    }
    return accu ;
}

// Absolute value, in a strange layout
Int abs (Int n) { if (n < 0) return -n; else return n ; }

// make a copy of a tuple with swapped elements
(b, a) swapCopy ( (a, b) pair ) {
    return (pair.snd, pair.fst);
}

// swap the elements in a tuple
(a, a) swap ( (a, a) tuple )
{
    a tmp = tuple.fst;
    tuple.fst = tuple.snd;
    tuple.snd = tmp;
    return tuple;
```

```
}

// list append
[t] append ( [t] l1, [t] l2 )
{
    if ( isEmpty ( l1 ))
        return l2;
    else
    {
        l1.tl = append ( l1.tl, l2 );
        return l1;
    }
}

// square the odd numbers is a list and remove the even numbers
[Int] squareOddNumbers ([Int] list)
{
    while (! isEmpty (list) && list.hd % 2 == 0)
        list = list.tl;
    if ( ! isEmpty (list))
    {
        list.hd = list.hd * list.hd;
        list.tl = squareOddNumbers(list.tl);
    }
    return list;
}
```

2 Assignment

An important part of this course is to make your own implementation of SPL. We will split this in four steps. The current exercise is the first step of this compiler. It is allowed to construct the compiler with a partner. We strongly recommend to work together.

The assignment consists of two parts: 1) a parser and 2) a pretty-printer for SPL. The input of the parser and pretty-printer chain is a plain-text `.spl` file, containing a single SPL program. The output is the pretty-printed abstract syntax tree in concrete syntax. Hence, the pretty printed program should be a valid SPL program.

2.1 Parser

Implement a parser for the SPL concrete syntax. Think carefully about the design of the abstract syntax, since this is the foundation of your compiler. A proper design will make your life easier in the upcoming assignments.

You are free in the choice of implementation language for your compiler. You can either use one of the imperative/object oriented languages C++ or Java, or one of the functional languages Clean Haskell, F# or Scala. If you prefer to use an other language for the implementation you should first come to talk with us. Student experience over the last years shows that it is much easier and faster to use a functional language, even if you did not use them for a while.

It is *not* allowed to use a parser generator tool to implement your compiler. The purpose of this exercise is that you learn to understand how a scanner and parser work. You are free to use support libraries provided by the language if they do not solve the essential scanning and parsing problems for you. This implies that you have to solve problems like left recursion, overlapping rules and ambiguity by transforming the grammar.

2.2 Pretty-printer

Implement a pretty-printer for the SPL abstract syntax tree. Printing the parsed program should produce an equivalent program that is accepted by your parser. Differences between the original file and the produced file typically concern layout, comments, and parentheses.

2.3 Deliverable

On Tuesday March 11 at 15:30 you have to give a very brief presentation. In this presentation you have to tell us how your parser is constructed (implementation language used, changes to the grammar, and other interesting points of your program) and demonstrate your parser and pretty printer to us and the other students. For this demonstration you have to prepare at least 10 test programs in SPL. In you presentation you have to show only the most interesting or challenging example. You can use the program above as starting point.