Phase 1: Lexical Analysis and Parsing

Compiler Construction
Radboud University

January 17, 2019

1 Preliminaries

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

1.1 Grammar

Programs in SPL are defined by the grammar SPL:

```
SPL       = Decl+
Decl      = VarDecl
          | FunDecl
VarDecl   = ('var' | Type) id '=' Exp ';'
FunDecl   = id '(' [ FArgs ] ')' [ '::' FunType ] '{' VarDecl* Stmt+ '}'
RetType   = Type
          | 'Void'
FunType   = [ FTypes ] '→' RetType
FTypes    = Type [ FTypes ]
Type      = BasicType
          | '(' Type ',' Type ')'
          | '[' Type ']'
          | id
BasicType = 'Int'
          | 'Bool'
          | 'Char'
FArgs     = [ FArgs ',' ] id
Stmt      = 'if' '(' Exp ')' '{' Stmt* '}' [ 'else' '{' Stmt* '}' ]
          | 'while' '(' Exp ')' '{' Stmt* '}'
          | id Field '=' Exp ';'
          | FunCall ';'
          | 'return' [ Exp ] ';'
Exp       = id Field
          | Exp Op2 Exp
          | Op1 Exp
          | int
          | char
          | '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**, **read** and **isEmpty**. The functions **print** and **isEmpty** have a single argument of type t and [t] respectively. The binary operators +, −, ∗, /, and %, have type t×t→t. For the type **Int** all these operators are defined. For **Char** at least + and − are defined. The comparison operators ==, <, >, <=, >=, and != have type t×t→**Bool**. The binary logical operators && and || have type **Bool**×**Bool**→**Bool**. The cons-operator, :, has type t×[t]→[t]. The unary negation operator, !, has type **Bool**→**Bool** and the unary minus operator, −, has type **Int**→**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]→**Bool**. The polymorphic **print** function has type t→**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.

1.2 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, character and Boolean function arguments are plain values. They behave like local values 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.

2 SPL Example Code

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.
*/
facR ( n ) :: Int → Int {
    if ( n < 2 ) {
        return 1;
    } else {
        return n * facR ( n - 1 );
    }
}

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

// A main function to check the results
// It takes no arguments, so the type looks like this:
main ( ) :: -> Void {
    var n = 0;
    var facN = 1;
    var 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)
facL ( n ) :: Int -> Int {
    return product (fromTo ( 1, n ));
}

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

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

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

// Absolute value in one line
abs ( n ) :: Int → Int { if (n < 0) { return −n; } else { return n ; } }

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

// swap the elements in a tuple
swap ( tuple ) :: (a, a) → (a, a) {
    var tmp = tuple.fst;
    tuple.fst = tuple.snd;
    tuple.snd = tmp;
    return tuple;
}

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

// square the odd numbers in a list and remove the even numbers
squareOddNumbers ( list ) :: [Int] → [Int] {
    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;
}
```

3 Assignment

The point 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 the compiler. We strongly recommend to work together in teams of two.

The first assignment consists of two parts: a parser and 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. The pretty printed program should be a valid `SPL` program.

3.1 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.

It is *not* allowed to use a parser generator tool to implement your compiler. The purpose of this exercise is that you learn how scanners and parsers work. You are free to use support libraries provided by the language in which you write your compiler, 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.

3.2 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.

3.3 Deliverables

You have to give a brief presentation about your parser. The date has been announced in the lecture and on Brightspace. In this presentation you tell the other students and us how your parser is constructed and demonstrate your parser and pretty printer. We all know the language, so you should mention things specific to your parser, like implementation language used, changes to the grammar, and other interesting points.

Don't forget that your final report must have a section about your parser. You should make notes now that your memory is still fresh. As a starting point briefly describe the common problems of parsing and how you solve them. Describe which features your parser has and how you implemented them. Do you have a separate lexer? How do lexer and parser communicate? How detailed are the error messages? Do you have error recovery and if so how does it work?