Rust$^-$: A Simple Rust Programming Language

Programming Assignment 2

Syntactic and Semantic Definitions

Due Date: 1:20PM,Tuesday, May 22, 2018

Your assignment is to write an LALR(1) parser for the *Rust*$^-$ language. You will have to write the grammar and create a parser using **yacc**. Furthermore, you will do some simple checking of semantic correctness. Code generation will be performed in the third phase of the project.

1 Assignment

You first need to write your symbol table, which should be able to perform the following tasks:

- Push a symbol table when entering a scope and pop it when exiting the scope.
- Insert entries for variables, constants, and procedure declarations.
- Lookup entries in the symbol table.

You then must create an LALR(1) grammar using **yacc**. You need to write the grammar following the syntactic and semantic definitions in the following sections. Once the LALR(1) grammar is defined, you can then execute **yacc** to produce a C program called "**y.tab.c**", which contains the parsing function **yyparse()**. You must supply a main function to invoke **yyparse()**. The parsing function **yyparse()** calls **yylex()**. You will have to revise your scanner function **yylex()**.

1.1 What to Submit

You should submit the following items:

- revised version of your **lex** scanner
- a file describing what changes you have to make to your scanner
- your **yacc** parser
  Note: comments must be added to describe statements in your program
- Makefile
- test programs

1.2 Implementation Notes

Since **yyparse()** wants tokens to be returned back to it from the scanner. You should modify the definitions of **token**, **tokenInteger**, **tokenString**. For example, the definition of **token** should be revised to:

```
#define token(t) {LIST; printf("<\%s>\n","t"); return(t);}
```

2 Syntactic Definitions

2.1 Program Units

The two program units are the *program* and *functions*.

2.1.1 Program

A program has the form:

> <zero or more variable and constant declarations>
> one or more function declaration

where the item in the < > pair is optional. Every *Rust*$^{-}$ program has at least one function, i.e. the `main` function: `fn main() { }`.

2.2 Constant and Variable Declarations

There are two types of constants and variables in a program:

- global constants and variables
  declared inside the program
- local constants and variables
  declared inside functions

Data Types and Declarations

The predefined data types are **string**, **int**, **bool**, and **float**.

2.2.1 Constants

A constant declaration has the form:

> **let** *identifier*<: *type*> **=** *constant_exp*

where *type* is optional with one of the predefined data types listed above. When the type attribute is omitted, the type of the declared constant must be inferred based on the constant expression on the right-hand side. Note that constants are immutable. In other words, constants cannot be reassigned or this code would cause an error.

For example,

```
let s = "Hey There";
let i = -25;
let f:float = 3.14;
let b:bool = true;
```

2.2.2 Variables

A variable declaration has the form:

let mut *identifier*<: *type*> <**=** *constant_exp*>

where *type* is one of the predefined data types. When both the type attribute declaration, i.e : *type* and initialization are omitted from variable declarations, the default data type is **int**. For example,

```
let mut s = "Hey There";
let mut i;
let mut d:float;
let mut b:bool = true;
```

Arrays

Arrays declaration has the form:

let mut *identifier* [*type* , *constant_exp*]

For example,

```
let mut a[int, 10];         // an array of 10 integer elements
let mut b[bool, 5];         // an array of 5 boolean elements
let mut f[float, 100];      // an array of 100 float-point elements
```

2.2.3 Functions

Function declaration has the following form:

fn *identifier* (<formal arguments>) <−> *type* >
{
<zero or more variable and constant declarations>
<one or more statements>
}

where −> *type* is optional and *type* can be one of the predefined types. The formal arguments are declared in the following form:

identifier : *type* <, *identifier* : *type*, ... , *identifier* : *type*>

Parentheses are required even when no arguments are declared. No functions may be declared inside a function. For example,

```
// variables
let mut c;
let a = 5;

// function declaration
fn add(a:int, b:int) -> int
{
  return a+b;
}
```

```
// main function
fn main ( )
{
  c = add(a, 10);
  print c;
}
```

Note that functions with no retuen type are usually called as procedures and can not be used in expressions.

2.3 Statements

There are several distinct types of statements in *Rust*$^{-}$.

2.3.1 simple

The simple statement has the form:

identifier = *expression*

or

identifier[*integer_expression*] = *expression*

or

print *expression* or **println** *expression*

or

read *identifier*

or

return or **return** *expression*

expressions
Arithmetic expressions are written in infix notation, using the following operators with the precedence:

(1) − (unary)
(2) `* /`
(3) `+ -`
(4) `< <= == => > !=`
(5) `!`
(6) `&&`
(7) `||`

Note that the − token can be either the binary subtraction operator, or the unary negation operator. Associativity is the left. Parentheses may be used to group subexpressions to dictate a different precedence. Valid components of an expression include literal constants, variable names, function invocations, and array reference of the form

A [integer_expression]

function invocation
A function invocation has the following form:

identifier (<comma-separated expressions>)

2.3.2 block

A block is a collection of statements enclosed by { and }.The simple statement has the form:

{
<zero or more variable and constant declarations>
<one or more statements>
}

2.3.3 conditional

The conditional statement may appear in two forms:

if (*boolean_expr*)
block
else
block

or

if (*boolean_expr*)
block

2.3.4 loop

The loop statement has the form:

while (*boolean_expr*)
block

2.3.5 function invocation

A function is a procedure that has no return value. It has the following form:

identifier (<comma-separated expressions>)

3 Semantic Definition

The semantics of the constructs are the same as the corresponding Pascal and C constructs, with the following exceptions and notes:

- The parameter passing mechanism for procedures in call-by-value.
- Scope rules are similar to C.
- The identifier after the **end** of program or procedure declaration must be the same identifiers as the name given at the beginning of the declaration.
- Types of the left-hand-side identifier and the right-hand-side expression of every assignment must be matched.
- Any declaration in the interface block must have a matching procedure declaration. Furthermore, the types of formal parameters must match the types of the actual parameters.

4 *yacc* Template (yacctemplate.y)

```
%{
#define Trace(t)        if (Opt_P) printf(t)
int Opt_P = 1;
%}

/* tokens */
%token SEMICOLON

%%
program:        identifier semi
                {
                Trace("Reducing to program\n");
                }
                ;

semi:           SEMICOLON
                {
                Trace("Reducing to semi\n");
                }
                ;
%%
#include "lex.yy.c"

yyerror(msg)
char *msg;
{
    fprintf(stderr, "%s\n", msg);
}

main()
{
    yyparse();
}
```