CSCI 355, Compiler Construction (Syntax for Final Project)

Hans de Nivelle

06.11.2020

1 Introduction

The final project of this course consists of implementing a compiler for the language below, which we call *C*& (pronounce 'C flat') for the time being. In order to make it digestible, we cut the final project into 3 parts:

1. Implement a parser, using **jflex** and **CUP**.
2. Implement a semantic analyzer.
3. Implement a translation function.

We will give you an implementation of the register/stack machine that was presented in the slides, so that you can test your translation function.

2 Terminal Symbols

Most of the terminal symbols already featured in earlier exercises, so you can reuse the **jflex** code that you have already. Note however that

- There will be character constants, which are formed with single quotes.
- There are couple of new reserved words:

```
if       then    else      while    do       return
pointer  array   function structdef constant
null
void char bool integer double
```

- **false** and **true** are Boolean constants. The tokenizer should recognize them, and return them as constants of type Bool.
- **null** is the null pointer, which is a constant of type **pointer**(**void**).

- There are more operators than in earlier tasks (you can just look them up in the grammar).
- The implementation of tree has changed. See Section 7 below.

In the description of the grammar, we use boldface for reserved words. This means that **integer** denotes the reserved word `integer`, while just integer denotes an integer constant. Terminal symbols start with a lowercase letter, non-terminal symbols with an uppercase letter in the description of the grammar.

3 Top Level Program

A program is a mixture of function definitions and struct definitions. The start symbol of the grammar is Prog.

Prog	→	Prog **function** identifier (Decllist) : Type Stat
	→	Prog **structdef** identifier = (Decllist);
	→	Prog **constant** identifier = bool;
	→	Prog **constant** identifier = char;
	→	Prog **constant** identifier = integer;
	→	Prog **constant** identifier = double;
	→	
Decllist	→	
	→	Decllist2
Decllist2	→	Decl
	→	Decllist2 , Decl

Constants play the same role as makros in *C*. Their purpose is for physical constants, and array bounds. We need both Decllist and Decllist2, in order to deal with semicolons between declarations properly.

4 Statements

We define statements. As in all modern programming languages, statements and variable declarations can be mixed. A variable exists until the end of the block in which it is declared. We don't allow declarations in the conditions of

ifs or **while**s, because these are a bit harder to implement.

$$\begin{array}{lcl}
\text{Stat} & \to & \textbf{if } \text{Expr } \textbf{then } \text{Stat} \\
& \to & \textbf{if } \text{Expr } \textbf{then } \text{Stat } \textbf{else } \text{Stat} \\
& \to & \textbf{while } \text{Expr } \textbf{do } \text{Stat} \\
& \to & \textbf{begin } \text{Statlist } \textbf{end} \\
& \to & \textbf{print } \text{Expr} \\
& \to & \textbf{print } \text{string} \\
& \to & \textbf{return } \text{Expr} \\
& \to & \text{Expr} \\
& \to & \text{Decl} \\
\\
\text{Statlist} & \to & \text{Stat} \\
& \to & \text{Statlist}; \text{Stat}
\end{array}$$

We need to say something about the delicate use of `;`. There are two interpretions: The first one, which is used by the C-family of languages, is to view `;` as a statement terminator. The second view, which is used by Pascal, is to view `;` as a statement connector. In that case, the last statement in a block, does not have a `;`. We follow the Pascal approach. We will also use **begin/end** instead of {}.

print string can be translated into a single instruction. In a real compiler, the string would have to be translated into an array of characters, but in this exercise, we will not do that. It was added to make output a bit more readable.

Constructing ASTs is straightforward. You have to invent names for the statement constructors. We recommend the following names: [if] (with two or three arguments), [while], [compound], [print], [return].

For expressions, we recommend [Expr](e). For Decl, there is the problem where to put the type. We recommend to represent id : t, as [decl](id), and put t in the type field. The lr field can be set to 'L'.

5 Declarations

A declaration has form

$$\text{Decl} \quad \to \quad \text{Identifier} : \text{Type}$$

The rules for Type follow below, we first give some explanations. We decided to remove unsigned, because Java doesn't support it, All primitive types are represented by their corresponding Java types, but as `Object`, i.e. `java.lang.Boolean`, `java.lang.Character`, `java.lang.Integer`, and `java.lang.Double`. Arrays and structs are represented by simple, flat layout in memory. The grammar rules for Type are:

$$\begin{array}{lll} \text{Type} & \rightarrow & \textbf{void} \mid \textbf{bool} \mid \textbf{char} \mid \textbf{integer} \mid \textbf{double} \\ & \rightarrow & \textbf{pointer}(\text{ Type }) \\ & \rightarrow & \textbf{array}(\text{ integer, Type }) \\ & \rightarrow & \textbf{array}(\text{ identifier, Type }) \\ & \rightarrow & \text{identifier} \end{array}$$

We use intentional type equivalence for structs, and extensional equivalence for all other types. You don't need to worry about this, because it is already implemented by `equals( )` in class `type.Type`.

Types of form **array**(identifier, Type) are allowed only when the identifier is defined as a constant. The size of an array must be always known at compile time.

6 Expressions

Expressions are like *C* expressions, but a bit simplified. We decided to distinguish between variables and function calls without arguments. This means that `id` and `id( )` are now distinguished. In earlier exercises, `id( )` was not allowed.

We omitted the modifying assignment operators `+=, -=, *=, /=, %=` . If you want, you may add them back. We kept the increasement and decreasement operators, because they are characteristic for *C*. As usual, we index expressions, in order to get priorities:

$$
\begin{array}{lcl}
\text{Expr} & \to & \text{Expr2} = \text{Expr2} \;\mid\; \text{Expr2} \\[1em]
\text{Expr2} & \to & \text{Expr3} \;?\; \text{Expr3} : \text{Expr3} \;\mid\; \text{Expr3} \\[1em]
\text{Expr3} & \to & \text{Expr3} \;||\; \text{Expr4} \;\mid\; \text{Expr4} \\[1em]
\text{Expr4} & \to & \text{Expr4} \;\&\&\; \text{Expr5} \;\mid\; \text{Expr5} \\[1em]
\text{Expr5} & \to & !\, \text{Expr5} \;\mid\; \text{Expr6} \\[1em]
\text{Expr6} & \to & \text{Expr7} \;==\; \text{Expr7} \;\mid\; \text{Expr7} \;!=\; \text{Expr7} \\
 & \to & \text{Expr7} < \text{Expr7} \;\mid\; \text{Expr7} > \text{Expr7} \\
 & \to & \text{Expr7} <= \text{Expr7} \;\mid\; \text{Expr7} >= \text{Expr7} \\
 & \to & \text{Expr7} \\[1em]
\text{Expr7} & \to & \text{Expr7} + \text{Expr8} \;\mid\; \text{Expr7} - \text{Expr8} \;\mid\; \text{Expr8} \\[1em]
\text{Expr8} & \to & \text{Expr8} \;\star\; \text{Expr9} \;\mid\; \text{Expr8} \;/\; \text{Expr9} \;\mid\; \text{Expr8} \;\%\; \text{Expr9} \\
 & \to & \text{Expr9} \\[1em]
\text{Expr9} & \to & -\text{Expr9} \;\mid\; \text{Expr10} \\[1em]
\text{Expr10} & \to & \star\,\text{Expr10} \;\mid\; \&\,\text{Expr10} \;\mid\; \text{Expr11} \\[1em]
\text{Expr11} & \to & ++\,\text{Expr11} \;\mid\; --\,\text{Expr11} \;\mid\; \text{Expr12} \\[1em]
\text{Expr12} & \to & \text{Expr12}\,++ \;\mid\; \text{Expr12}\,-- \\
 & \to & \text{Exp12}\,.\ \text{identifier} \;\mid\; \text{Expr12} \to \text{identifier} \\
 & \to & \text{Expr12}\ [\ \text{Expr}\ ] \\
 & \to & \text{identifier} \\
 & \to & \text{identifier}\ (\ \text{Exprlist}\ ) \\
 & \to & \text{bool} \;\mid\; \text{char} \;\mid\; \text{integer} \;\mid\; \text{double} \;\mid\; \text{pointer} \\
 & \to & (\ \text{Expr}\ )
\end{array}
$$

Exprlist is a possibly empty list of Exprs:

$$
\begin{array}{lcl}
\text{Exprlist} & \to & \\
 & \to & \text{Exprlist2} \\[1em]
\text{Exprlist2} & \to & \text{Expr} \\
 & \to & \text{Exprlist}, \text{Expr}
\end{array}
$$

When constructing the AST, use the following replacements:

Expression	AST
E_1 ? E_2 : E_3	$??(E_1, E_2, E_3)$
E_1 && E_2	$??(E_1, E_2, \mathbf{false})$
E_1 || E_2	$??(E_1, \mathbf{true}, E_2)$
$!E$	$??(E, \mathbf{false}, \mathbf{true})$
$E.f$	$\mathrm{select}_f(E)$
$E \to f$	$\mathrm{select}_f(\star E)$
$E_1[\ E_2\ ]$	$\star(+(E_1, E_2))$
E ++	[xpp](E)
E −−	[xmm](E)
++ E	[ppx](E)
−− E	[mmx](E)

The other operators just remain themselves. The purpose of the [ ] is to make it impossible to directly use the names in the program.

7 Abstract Syntax Tree

In exercise 3, we used a class `tree.Tree` for representing abstract syntax trees. In this final project, we use `ast.Tree`, which is similar, but a bit different. The differences are:

- We distinguish between id() (it will be `ast.Apply( "id" )`) and id (which will be `ast.Identifier( "id" )`).
- Constants cannot have subtrees any more.
- We added `ast.Select( "f", t )` for $t.f$

8 Hints

- `null` should be tokenized as POINTERCONST with attribute `ast.Pointer(0)`. It is the only pointer constant.
- `false` and `true` should be tokenized as BOOLCONST with attribute `ast.Bool(false/true)`. This means that they are Boolean constants.
- We need different tokens for integer, one for the word `integer`, and one for integer constants. The same applies to all primitive types. We use `INTEGERCONST` for integer constants, and INTEGER for the word `INTEGER`. INTEGERCONST has an attribute of type `ast.Integer`, while INTEGER has no attribute.
- For the names of the operators, I use the Python names as much a possible. This means that `*` is called `MUL`, for example. The exception is `&`, which

would be AND in Python. I call it `AMPERSAND`, because its function has nothing to do with logical and in our language. The tokens `++`, `--` are called PLUSPLUS and MINUSMINUS.

- It is useful to allow `#` as additional end of file symbol. This makes it easier to terminate input from a terminal, and you can put comments after `#` in a file. Unfortunately, there is an irritating bug in CUP: You need to give two `#`s in order to terminate the input.
- You have to run **CUP** with option `-expect 1`. This means that we expect one shift/reduce conflict. We expect this conflict because of the dangling else problem.

You are allowed to make small changes in the grammar that don't change the language, or make small extensions. If you are in doubt, contact us.

9 Examples

```
structdef list = ( d: double, next: pointer( list ));
structdef complex = ( re: double, im: double );
constant pi = 3.141592653589793;

function length( p : pointer( list )) : integer
begin
   len : integer; len = 0;
      /* We don't have syntax for immediate initialization */
      /* of form integer len = 0;                           */

   while p != null do
   begin
      ++ len;
      p = p -> next
   end;
   return len
end

function fact( n : integer ) : double
begin
   res : integer;
   res = 1.0;
   while n != 0 do
   begin
      res = res * n;
      -- n
   end;
   return res
```

```
end

function sum( c1 : complex, c2 : complex ) : complex
begin
   res : complex;
   res. re = c1. re + c2. re;
   res. im = c1. im + c2. im;
   return res
end

function sinegrad( x : double ) : double
begin
   x = x * pi / 180;
   n : integer; n = 1;
   sum = 0;
   while sum + term != sum do
   begin
      sum = sum + term;
      term = term * x * x / ( n + 1 ) / ( n + 2 );
      n = n + 2
   end;
   return sum
end
```