T-603-THYD Compilers
Fall 2020 Project Part 2: Parsing and AST

Objective

The objective of this part of the project is to get a hands-on experience in writing a parser, both manually and by using compiler-construction software tools.

Logistics

You can work on the project in a group of two. The project is due on **Sunday, October 11th, 2020 at 23:59**.

You are provided with a skeleton C++ program. You should change the following files, which you then submit via Canvas: *hparser.h*, *hparser.cpp*, and *tpparser.yy*.

Description

You are to write two functionally equivalent parsers for a simplified version of the programming language *Pascal*. The first parser is a top-down recursive-decent parser written by hand, and the second is a bottom-up parser written in Bison.

The parser creates an abstract-syntax tree (AST) during parsing. The code for the AST is already provided (in *ast.h*). If successful in parsing the input, the parser returns an AST; however, if a syntax error is detected the parser outputs an error message and quits.

Following are language-dependent things to note:

- A grammar for the language is provided in an appendix in an *Extended Backus Naur Form (EBNF)*. Note that you might have to rewrite the grammar to be better suited for each parser, i.e. to be a LL(1) or LARL(1) grammar, respectively. In particular, *Bison* does not support grammar constructs in *EBNF*, only plain BNF.
- During parsing you must ensure that variables, procedures, and functions are declared before they are used. Use the provided symbol table for this (*symbol_table.h*).
- Parameters to functions and procedures are passed by value. A function returns a value by assigning the value to return to the identifier naming the function (e.g., in a function named *Pi*, the value to return would be set by the statement *Pi := 3.1415926).* Such assignment can though only be done within the corresponding function.
- There are four built-in procedures provided: *read*, *write*, and *writeln*: *read* inputs a value from the keyboard, *readln* inputs a value from the keyboard and moves the cursor to the next line on the console, *write* outputs a value to the console, and *writeln* outputs a value to the console followed by a new-line. In our simplified versions those functions can only take a single argument.

- The lexical analyser returns *integer* literals, *Boolean* literals (*True* and *False*), *real* literals and *string* literals. String literals can, however, only be used as arguments to *write*/*writeln* statements.
- One-dimensional arrays are provided and when they are declared their indexing range is specified as *n .. m*, where *0 < n < m <= 100*. Note that (in our simplified version) entire arrays cannot be used in expressions, passed as arguments, or returned by functions; however, individual array elements can do so because they are simple variable types.
- **You do not have to worry about type-checking for now, it is done during the semantic-analysis phase (in the next part of the assignment)**. For your information ---and to satisfy your curiosity--- the type checking will involve checking for appropriate types being used in expressions, that expression results are being assigned to appropriately typed variables, or used in the right context (e.g., return a Boolean or an integer value where appropriate). For example, in *if* and *while* statements, the conditional expression must resolve to a Boolean value, otherwise it results in a type error. Also, one would ensure that the number and type of arguments and formal parameters matches for function/procedure call, and that expressions used as array indices resolve to an integer value.

Appendix - Grammar

<program> ::= **program** *<identifier>* **;** *<main block>* **.**

<main block> ::= *<variable declarations>* *<callable declarations>* *<statements>*

<variable declarations> ::= *<empty>*
| **var** *<variable declaration>* **;** { *<variable declaration>* **;** }
<variable declaration> ::= *<identifier>* { **,** *<identifier>* } **:** *<type>*
<type> ::= *<simple type>* | *<array type>*
<array type> ::= **array [** *<index range>* **] of** *<simple type>*
<index range> ::= *<integer constant>* **..** *<integer constant>*
<simple type> ::= **integer** | **boolean** | **real**

<callable declarations> ::= { *<procedure declaration>* | *<function declaration>* }
<procedure declaration> ::= **procedure** *<procedure identifier>* *<opt params>* **;** *<block>* **;**
<function declaration> ::= **function** *<function identifier>* *<opt params>* **:** *<simple type>* **;** *<block>* **;**
<opt params> ::= *<empty>* | **(** *<parameter list>* { **;** *<parameter list>* } **)**
<parameter list> ::= *<identifier>* { **,** *<identifier>* } **:** *<simple type>*
<block> ::= *<variable declarations>* *<statements>*

<statements> ::= *<compound statement>*
<compound statement> ::= **begin** *<statement>* { **;** *<statement>* } **end**
<statement> ::= *<simple statement>* | *<structured statement>*
<simple statement> ::= *<assignment statement>*
| *<procedure statement>*
| *<read statement>*
| *<write statement>*
| *<empty statement>*
<empty statement> ::= *<empty>*
<assignment statement> ::= *<variable lvalue>* **:=** *<expression>*
| *<function identifier>* **:=** *<expression>*
<procedure statement> ::= *<procedure identifier>* *<optional arguments>*
<read statement> ::= **read (** *<input variable>* **)** | **readln (** *<input variable>* **)**
<input variable> ::= *<variable lvalue>*
<write statement> ::= **write (** *<output value>* **)** | **writeln (** *<output value>* **)**
<output value> ::= *<string constant>* | *<expression>*

<structured statement> ::= *<compound statement>*
| *<if statement>*
| *<while statement>*
<if statement> ::= **if** *<expression>* **then** *<statement>*
| **if** *<expression>* **then** *<statement>* **else** *<statement>*
<while statement> ::= **while** *<expression>* **do** *<statement>*

<expression> ::= *<simple expression>* { *<rel op>* *<simple expression>* }
<simple expression> ::= *<term>* { *<add op>* *<term>* }
<term> ::= *<complemented factor>* { *<mult op>* *<complemented factor>* }
<complemented factor> ::= *<signed factor>* | **not** *<signed factor>*
<signed factor> ::= *<factor>* | *<sign>* *<factor>*
<factor> ::= *<variable rvalue>* | *<constant>* | **(** *<expression>* **)** | *<function call>*

<function call> ::= *<function identifier>* *<optional arguments>*
<optional arguments> ::= *<empty>* | **(** *<argument>* { **,** *<argument>* } **)**
<argument> ::= *<expression>*

<rel op> ::= **=** | **<>** | **<** | **<=** | **>=** | **>**
<add op> ::= **+** | **-** | **or**

<mult op> ::= ***** | **/** | **div** | **and**
<sign> ::= **+** | **-**

<variable lvalue> ::= *<variable>*
<variable rvalue> ::= *<variable>*
<variable> ::= *<entire variable>* | *<indexed variable>*
<indexed variable> ::= *<array variable>* **[** *<expression>* **]**
<array variable> ::= *<entire variable>*
<entire variable> ::= *<variable identifier>*

<variable identifier> ::= *<identifier>*
<procedure identifier> ::= *<identifier>*
<function identifier> ::= *<identifier>*

<constant> ::= *<integer constant>* | *<real constant>*| *<boolean constant>*
<integer constant> ::= ***<integer>***
<real constant> ::= ***<real>***
<boolean constant> ::= **true** | **false**
<string constant> ::= **<string>**

<empty> ::=