INF400 2023-2024 Fall Semester

Burak Arslan

Homework I

As part of this homework, you are asked to implement a lexer and a parser for our own language, kiraz, as the first stage of the course project.

To ease the pains of starting out with a greenfield project, a `CMake`-based project that contains the needed boilerplate is provided.

- You will use `C++` to implement the compiler.
- You are supposed to work on Linux. Working on other platforms could (be made to) work but you will be on your own.
- Using the given starter pack is **mandatory**.

The entry point for our compiler is `kirazc`, the kiraz compiler.

As part of this homework, kirazc will either:

- Accept .ki files and print their parse trees (the `-f` option).
- Accept kiraz source code strings and print their parse trees (the `-s` option).

1 Error reporting

Note that lexers generated by `flex` simply copy unrecognized tokens to standard output. This is not acceptable – your lexer needs to recognize everything and deal with it.

Note that any sort of recovery is out of scope[1] of this project. When the lexer or the parser encounters a string that doesn't belong to the language set, it must report an error that includes the line number and character number of the offending code fragment and terminate. Some examples:

```
$ ./kirazc -s '('
** Parser Error at 1:1 at token: OP_LPAREN
$ ./kirazc -s '1#2'       # not in alphabet of kiraz
** Parser Error at 1:2 at token: REJECTED(#)
$ ./kirazc -s '1+2*3+4)'
** Parser Error at 1:8 at token: OP_RPAREN
$ ./kirazc -s '1+2*3+4'      # no error
Sub(Add(Integer(1),Mult(Integer(2),Integer(3))),Integer(4))
```

2 Language Spec

The language grammar mainly specified in the ders-03-II.pdf slide pack. Some additional notes:

- **Alphabet**: For the time being, 26 letters of the English alphabet (both upper and lower case), 10 arabic digits, underscore and following symbols: `"{}()+-/*<=>`. Anything else needs to be explicitly rejected by the lexical analyzer.
- **Identifiers**: They start with a letter or an underscore, and continue with a letter, digit or underscore.
- **Integer Literals**: At least one digit.
- **Strings Literals**: Anything between double quotes (`"`).
- **Keywords**:

`KW_IMPORT`	import
`KW_FUNC`	func
`KW_IF`	if
`KW_WHILE`	while
`KW_CLASS`	class

[1] fr. Hors sujet

The skeleton project comes with a base class named `Token` for The classes for terminal nodes are subclasses of the class. `Node` for AST nodes.

If you need information from the last token emitted by flex, you can use the global `curtoken` variable.

The classes for terminal nodes are subclasses of the `Token` class. However, as `Token` and `Node` are part of distinct class hierarchies, terminals (that are subclasses of the `Token` class) will have to be converted to a subclass of the `Node` class.

As an example, the `L_INTEGER(10, "4096")` token is the string representation of the integer 4096 in base 10. `L_INTEGER(16, "1000")` is the same integer in base 16. Both should be converted to an "integer literal node" in the parse tree that contain the value 4096.

> *Hint:* Some of the classes could be `PositiveInteger`, `NegativeInteger`, `OpAdd`, `OpMult`, etc. `OpAdd` and `OpMult` may have a common parent class named `OpBinary`.

3 Function Parser (Part I)

The next step in the compiler project is the partial implementation of the parser for the kiraz function definition AKA the `func` statement.

3.1 `func` Statement

A func statement is made of:

- The `func` keyword,
- One identifier that denotes the function name (`n`),
- One argument list wrapped by the `OP_LPAREN` and `OP_RPAREN` tokens (`a`),
- One type annotation denoting the return type of the function (`r`),
- One function scope (`s`).

Here's a breakdown:

$$\texttt{func}\ \underbrace{\texttt{f}}_{\text{name}}\ \texttt{(}\underbrace{\overbrace{\texttt{a1: A}}^{\text{IwTA}}\texttt{, a2: A}}_{\text{argument list}}\texttt{)}\ \text{r.}\ \underbrace{\texttt{: R}}_{\text{type annot.}}\ \underbrace{\texttt{\{ let a : A = a1 + a2; ...\}}}_{\text{function scope}}$$

Here's the resulting AST[2]:

```
Func(
  n=Id(f),
  a=FuncArgs([
    Arg(n=Id(a1), t=Id(A)),
    Arg(n=Id(a2), t=Id(A))
  ]),
  r=Id(R),
  s=NodeList([
    Let(n=Id(a), t=Id(A), i=OP_PLUS(l=Id(a1), r=Id(a2)))   ]) )
```

3.1.1 Function Argument List

A function argument list is made of zero or more **identifiers with type annotations**, delimited by the comma operator. A function argument list is always wrapped by the `OP_LPAREN` and `OP_RPAREN` tokens.

3.1.2 Identifier with Type Annotation

It's an identifier followed by a type annotation which is made of:

- A colon operator,
- An identifier that denotes the type name.

3.1.3 Function Scope

A function scope is actually a **regular statement list** wrapped by `OP_LBRACE` and `OP_LBRACE` tokens.

The regular statement list is made of regular statements that are delimited by the `OP_SCOLON` token.

[2] Assuming no statements follow the first statement in the function scope

3.2 Regular Statement

For the scope of this homework, a regular statement is either one of the following:[3]

- The `let` statement (§3.3),
- The `func` statement (§3.1),
- The assignment statement(§3.4),
- The arithmetic statement
- Bare integer literals or identifiers.

3.3 `let` Statement

The kiraz let statement is made of:

- The `let` keyword,
- One identifier that denotes the variable name (`n`),
- Either:
  - The assignment operator followed by a literal (`i`). This is called an **initial value**.
  - A type annotation (`t`) followed by an optional statement as the initial value (`i`).

Valid examples:

- `let a = 1;`
- `let a :  Int64;`
- `let a :  Int64 = 1;`
- `let a :  Int64 = -(b + c) * 4;`

Please note that:

- A variable declaration that is missing both the type annotation and the inital value is to be rejected by the parser.
- A variable declaration that is missing the type annotation must only have a literal value on the right hand size of the assignment. Regular statements are to be rejected.

Invalid examples:

[3] This is the heart of the programming language. Be prepared for this list to grow in the coming assignments

- `let a;`
- `let a = b;`
- `let a = b + c;`
- `let a = (2 * 3);`

3.4 The Assignment Statement

An assignment statement is made of:

- One identifier,
- One `OP_ASSIGN` token,
- One regular statement.

3.5 The Arithmetic Statement

You may need to implement a `SignedNode` class to contain any arithmetic statement.

Here is a compound example within a `let` statement:

```
let a : Int64 = -(b + c) * 4;
```

And here is its parse tree:

```
Let(
  n=Id(a),
  t=Id(Int64),
  i=SignedNode(
    OP_MINUS,
    OP_MULT(
      l=OP_PLUS(l=Id(b), r=Id(c)),
      r=Int(10, 4)
    )
  )
)
```

3.6 Examples

Valid examples for the statements in the function scope:

- `let a: Int64 = -(a + b * 4) * (1 / 2 + 3);`
- `class B { let a : Int64; func init() : Void { a = 5; } }`

The AST for the first example would be:

```
Let(
    n=Id(a),
    t=Id(Int64),
    i=SignedNode(
        OP_MINUS,
        OP_MULT(
            l=OP_PLUS(
                l=Id(a), r=OP_MULT(l=Id(b), r=Int(10, 4))
            ),
            r=OP_PLUS(
                l=OP_DIVF(l=Int(10, 1), r=Int(10, 2)),
                r=Int(10, 3)
            )
        )
    )
)
```

Your Submission

- You are expected to turn in a `<student_id>_hw1.tar.gz` file that contains the modified skeleton project. Submit only the source code. More specifically, don't submit your local build directory.
- Remember that your code is expected to compile **without any warnings**. Compiler warnings are supposed to improve the quality of your code so don't mess with the CMake files.
- You may add new test cases if you think it's going to make your life easier. You are not supposed to touch the existing ones though.