A tiny GCC front end – Part 4

Jan 10, 2016 • Roger Ferrer Ibáñez • compilers, GCC • gcc, tiny

Now that we have a stream of tokens we can start performing syntactic analysis.

Syntactic correctness

The lexer is providing us a stream of tokens. But we still have to check if such sequence forms a valid program.

```
var i : int;
i := 1;
i := )i +1);
```

All the tokens in the program above are valid tokens, but the second assignment statement (line 3) is not an assignment statement because there is an incorrect token `)` after `:=`. On the other hand, the following program

```
if 3 then end
```

is syntactically valid but semantically incorrect because the condition expression does not have boolean type but integer type.

The syntactical analyzer will detect invalid token sequences but will not assess their semantic validity.

Recognizer

There is an extense and interesting bibliography about language recognition, but this is a blog not a compilers crash course so we will have to cut down the explanation: we will build a recursive descent recognizer based on the syntax description of part 1. If you are interested in parsing techniques an excellent (and a tad bit expensive too) reference is Parsing Techniques: A Practical Guide by Grune and Jacobs.

The strategy is very intuitive. Recognizing a rule like $\langle\text{declaration}\rangle \rightarrow$ **`var`** $\langle\text{identifier}\rangle$ **`:`** $\langle\text{type}\rangle$ **`;`** requires recognizing the elements in the right hand side of the rule. When an element in the right hand side is just a token (like **`var`**, **`:`** or **`;`**) then we only have to verify that the next token is of the expected kind at that point. When the element in the right hand side is another rule (like $\langle\text{identifier}\rangle$ or $\langle\text{type}\rangle$) we will just recursively recognize the rule. An element of the form w* is equivalent to a rule like ⟨rule⟩ → ϵ | w ⟨rule⟩ (where ϵ means an empty sequence of elements). An element of the form w+ is equivalent to a rule like ⟨rule⟩ → w | w ⟨rule⟩ .

Some rules are of the form ⟨rule⟩ → A | B | C. In such case we have to choose which A, B or C are going to recognize. We can break the tie peeking the current token. If each A, B or C start with different tokens, our problem is solved. If this is not the case, then we may have to peek some more tokens. Tiny is such a simple language that no more than one token will have to be peeked. When the alternative involves an empty sequence of tokens (like w* above) then we will consider that ϵ matches if the current token cannot start any of the alternatives.

Parser interface

In our implementation the syntactic and semantic analysis will be performed at the same time. Not that it has to be this way. For bigger languages splitting these two steps may be more appropiate. Tiny is so small that doing it is not worth. That said, in this post we will only focus on the syntactic recognition. For now, we only need to know that, upon recognizing a rule, a semantic value may be computed. What exactly this semantic value is, is not important now.

Since we will combine syntactic and semantic analysis we will call this step parsing. A `Parser` class will drive the parsing process.

```
struct Parser
{
  // ...
public:
  Parser (Lexer &lexer_) : lexer (lexer_) { }
  // ...
private:
  Lexer &lexer;
};
```

The parser will get tokens from the lexer, so we pass a reference to the lexer (recall that he lexer will synthesize tokens from the input file).

The main goal of our parser is parsing a program. So we will have a `parse_program` method.

```
struct Parser
{
  // ...
public:
  Parser (Lexer &lexer_) : lexer (lexer_) { }
  void parse_program();
  // ...
};
```

Ideally `parse_program` should return a semantic value, but at this moment we do not care.

Let's recall the syntax of ⟨program⟩

$$\langle\text{program}\rangle \rightarrow \langle\text{statement}\rangle^*$$

As said above $\langle\text{statement}\rangle^*$ is equivalent to $\langle\text{rule}\rangle \rightarrow \epsilon | \langle\text{statement}\rangle \langle\text{rule}\rangle$. We will call this rule $\langle\text{statement-seq}\rangle$. Like this.

$$\langle\text{statement-seq}\rangle \rightarrow \epsilon | \langle\text{statement}\rangle \langle\text{statement-seq}\rangle$$

Inside a $\langle\text{program}\rangle$ the $\langle\text{statement-seq}\rangle$ ends when the end-of-file is found. This suggests that we just have to keep parsing statements until we find an end-of-file and a possible implementation of `parse_program` does this.

```
void
Parser::parse_program ()
{
   parse_statement_seq();
}
```

and similarly, `parse_statement_seq`

```
void
Parser::parse_statement_seq ()
{
  // Parse statements until done and append to the current stmt list;
  while (lexer.peek()->get_id() != Tiny::END_OF_FILE)
    {
      parse_statement ();
    }
}
```

This is fine but if you check the syntax of tiny, you will see that the condition of finalization of a $\langle\text{statement-seq}\rangle$ is not always the end of file. Sometimes can be **end** (in the then or else part of an if statement, int the body for statement and in the body of a while statement) and sometimes is **else** (in the then part of an if statement). So this means that `parse_statement_seq` can be reused if we parameterize the finalization condition. Something like this.

```
void
Parser::parse_statement_seq (bool (Parser::*done) ())
{
  // Parse statements until done and append to the current stmt list;
  while (!(this->*done) ())
    {
      parse_statement ();
    }
}
```

And now we rewrite `parse_program` like.

```
bool
Parser::done_end_of_file ()
{
  const_TokenPtr t = lexer.peek_token ();
  return (t->get_id () == Tiny::END_OF_FILE);
}

void
Parser::parse_program ()
{
  parse_statement_seq (&Parser::done_end_of_file);
}
```

Now we can proceed to parse a statement. Let's recall the syntax of a statement.

$$\langle\text{statement}\rangle \rightarrow \langle\text{declaration}\rangle \mid \langle\text{assignment}\rangle \mid \langle\text{if}\rangle \mid \langle\text{while}\rangle \mid \langle\text{for}\rangle \mid \langle\text{read}\rangle \mid \langle\text{write}\rangle$$

Now we have one of those alternatives. Fortunately tiny is so simple that is easy to tell by just peeking the current token which kind of statement it can be.

```
void
Parser::parse_statement ()
{
  const_TokenPtr t = lexer.peek_token ();
  switch (t->get_id ())
    {
    case Tiny::VAR:
      parse_variable_declaration ();
      break;
    case Tiny::IF:
      parse_if_statement ();
      break;
    case Tiny::WHILE:
      parse_while_statement ();
      break;
    case Tiny::FOR:
      parse_for_statement ();
      break;
    case Tiny::READ:
      parse_read_statement ();
      break;
    case Tiny::WRITE:
      parse_write_statement ();
      break;
    case Tiny::IDENTIFIER:
      parse_assignment_statement ();
      break;
    default:
      unexpected_token (t);
      skip_after_semicolon ();
      break;
    }
}
```

We peek the current token and we check which statement it can initiate. If no statement can be initiated given the current token, the we call a diagnostic function with the unexpected token. We do some minimal error recovery by skiping all tokens until a semicolon is found.

```
void
Parser::unexpected_token (const_TokenPtr t)
{
  error_at (t->get_locus (), "unexpected %s\n", t->get_token_description ());
}

void
Parser::skip_after_semicolon ()
{
  const_TokenPtr t = lexer.peek_token ();

  while (t->get_id () != Tiny::END_OF_FILE && t->get_id () != Tiny::SEMICOLON)
    {
      lexer.skip_token ();
      t = lexer.peek_token ();
    }

  if (t->get_id () == Tiny::SEMICOLON)
    lexer.skip_token ();
}
```

`error_at` is a function that tells GCC to emit a diagnostic in the given location we just complain of an unexpected token. For instance the following erroneous program.

```
3;
```

will emit the following diagnostic.

```
$ gcctiny -c foo.tiny
foo.tiny:1:1: error: unexpected integer literal

 3;
 ^
```

If the front end has signaled any error, once it finishes, GCC will stop and return a non-zero error code. So no assembler is emitted at all for erroneous inputs.

A user-friendly front end, though, should attempt to continue in order to diagnose more errors to the user. A front end that stops at the first error may be OK but then forces the user to repeatedly invoke the compiler to discover new errors. It seems, thus, sensible to try to diagnose as much as possible each invocation of the compiler (some compilers have a configurable error limit to avoid spending more time diagnosing errors than doing useful work). This implies that after an error has been diagnosed the front end has to recover from it. To do this the front end will have to use some error recovery strategy.

The strategy that we will use for tiny is rather simple and it is commonly known as *panic mode*. When an un expected token appears, the parser attempts to advance the input to some sensible position. Here we skip after a semicolon in the hope that a correct statement will start there. Note that error recovery is always a *best effort*. Until the compiler is able to read the mind of the programmer, it can only guess where the real error happened. It is not unlikely that a cascade of errors is generated because the parsing restarts in the wrong place. It is not the case of tiny but some programming languages are noticeably hard when it comes to diagnosing syntactic errors.

Parsing statements

Ok, now we can parse a program and its statement sequence. Let's see how we parse each individual statement.

A variable declaration statement has the following form.

$$\langle\text{declaration}\rangle \rightarrow \textbf{var}\ \langle\text{identifier}\rangle\ \textbf{:}\ \langle\text{type}\rangle\ \textbf{;}$$

So a straightforward implementation of a parser of this statement is the one below.

```
void
Parser::parse_variable_declaration ()
{
  if (!skip_token (Tiny::VAR))
    {
      skip_after_semicolon ();
      return;
    }

  const_TokenPtr identifier = expect_token (Tiny::IDENTIFIER);
  if (identifier == NULL)
    {
      skip_after_semicolon ();
      return;
    }

  if (!skip_token (Tiny::COLON))
    {
      skip_after_semicolon ();
      return;
    }

  if (!parse_type ())
     return;

  skip_token (Tiny::SEMICOLON);
}
```

Here we use a function `skip_token` that given a token id, checks if the current token has that same id. If it has, it just skips it and returns true. Otherwise diagnoses an error and returns false. When skip_token fails (i.e. returns false) we immediately go to panic mode and give up parsing the current statement. As you can see this code quickly becomes tedious and repetitive. No wonder there exist tools, like ANTLR by Terence Parr, that automate the code generation of recursive descent recognizers.

Function `skip_token` simply forwards to `expect_token`.

```
bool
Parser::skip_token (Tiny::TokenId token_id)
{
  return expect_token (token_id) != const_TokenPtr ();
}
```

Function `expect_token` checks the current token. If its id is the same as the one we expect, it skips and returns it, otherwise it diagnoses an error and returns an empty pointer (i.e. a null pointer).

```
const_TokenPtr
Parser::expect_token (Tiny::TokenId token_id)
{
  const_TokenPtr t = lexer.peek_token ();
  if (t->get_id () == token_id)
    {
      lexer.skip_token ();
      return t;
    }
  else
    {
      error_at (t->get_locus (), "expecting %s but %s found\n",
                get_token_description (token_id), t->get_token_description ());
      return const_TokenPtr ();
    }
}
```

When parsing a variable declaration we invoke a `parse_type` function, that parses the rule $\langle\text{type}\rangle$.

$$\langle\text{type}\rangle \rightarrow \texttt{int} \mid \texttt{float}$$

Its associated parsing function is rather obvious too.

```
bool
Parser::parse_type ()
{
  const_TokenPtr t = lexer.peek_token ();

  switch (t->get_id ())
    {
    case Tiny::INT:
      lexer.skip_token ();
      return true;
    case Tiny::FLOAT:
      lexer.skip_token ();
      return true;
    default:
      unexpected_token (t);
      return false;
    }
}
```

Note that we return a boolean because we want the caller know if the parsing of the type succeeded.

Another interesting statement is the if-statement. Let's recall its syntax definition.

$$\begin{array}{l} \langle \text{if} \rangle \rightarrow \textbf{if} \ \langle \text{expression} \rangle \ \textbf{then} \ \langle \text{statement} \rangle^* \ \textbf{end} \\ \quad | \ \textbf{if} \ \langle \text{expression} \rangle \ \textbf{then} \ \langle \text{statement} \rangle^* \ \textbf{else} \ \langle \text{statement} \rangle^* \ \textbf{end} \end{array}$$

As shown, deriving a parse function for the rule $\langle \text{if} \rangle$ is not obvious because the two forms share a lot of elements. It may help to split the rule $\langle \text{if} \rangle$ in two rules follows.

$$\begin{array}{l} \langle \text{if} \rangle \rightarrow \langle \text{if-then} \rangle \ \textbf{end} \\ \quad | \ \langle \text{if-then} \rangle \ \textbf{else} \ \langle \text{statement} \rangle^* \ \textbf{end} \\ \langle \text{if-then} \rangle \rightarrow \textbf{if} \ \langle \text{expression} \rangle \ \textbf{then} \ \langle \text{statement} \rangle^* \end{array}$$

From this definition it is clear that we have to parse first an **if**, followed by an expression, followed by a **then** and followed by a statement sequence. In this case the statement sequence will finish when we encounter an **end** or an **else** token. If we find an **end** we are done parsing the if statement. If we find an **else**, it means that we still have to parse a statement sequence (this time the sequence finishes only if we encounter an **end**) and then an **end** token.

```
void
Parser::parse_if_statement ()
{
  if (!skip_token (Tiny::IF))
    {
      skip_after_end ();
      return;
    }

  parse_expression ();

  skip_token (Tiny::THEN);

  parse_statement_seq (&Parser::done_end_or_else);

  const_TokenPtr tok = lexer.peek_token ();
  if (tok->get_id () == Tiny::ELSE)
    {
      // Consume 'else'
      skip_token (Tiny::ELSE);

      parse_statement_seq (&Parser::done_end);
      // Consume 'end'
      skip_token (Tiny::END);
    }
  else if (tok->get_id () == Tiny::END)
    {
      // Consume 'end'
      skip_token (Tiny::END);
    }
  else
    {
      unexpected_token (tok);
      skip_after_end ();
    }
}
```

Function `skip_after_end` is similar to `skip_after_semicolon` but with an **end** token. Note that these `skip_`*x* functions must protect themselves from an unexpected end of file.

```
void
Parser::skip_after_end ()
{
  const_TokenPtr t = lexer.peek_token ();

  while (t->get_id () != Tiny::END_OF_FILE && t->get_id () != Tiny::END)
    {
      lexer.skip_token ();
      t = lexer.peek_token ();
    }

  if (t->get_id () == Tiny::END)
    lexer.skip_token ();
}
```

Remaining statements are parsed likewise and they do not bear special complexity except for a pervasive rule appearing in several of the statements: expression. This rule is so special that has its own parsing technique.

Parsing expressions

Parsing expressions is complex because the sublanguage of expressions must be flexible enough to express lots of different kinds of computations. Expressions can be understood as being formed by two kinds of elements: *operators* that most of the time correspond with some punctuation (or keywords like **or**, **and** and **not**) and *operands* that correspond to other expressions (usually a subset of the expression sublanguage). Operators have an *arity*, which means the number of operands they operate, and a "*fixity*" which defines the position of the operator respect its operands in the syntax. Arity of most operators is either *unary*, a single operand, or *binary*, two operands (some languages have *ternary* operators like the conditional operator though they may need to include extra operators). When it comes to "fixity" operators can be *prefix*, the operands appear after the operator, or *postfix*, the operands appear before the operator. For binary operators an extra *fixity* is possible called *infix*: the operator appears between the two operands.

Some programming languages have only prefix operators (in some form the LISP family works this way) This simplifies a lot the syntactic analysis as all unary expressions are of the form $\langle\text{op}\rangle$ $\langle\text{operand1}\rangle$ and all binary expressions of the form $\langle\text{op}\rangle$ $\langle\text{operand1}\rangle$ $\langle\text{operand2}\rangle$. Some notations (like the Reverse Polish notation) only use postfix operators, this has the same advantages as using only prefix operators.

While using prefix or postfix notation may be OK, most programming languages, including tiny, choose to use a notation closer, though not exactly the same, to the mathematical notation of arithmetic where most operators are infix. Infix notation introduces an additional problem though: it is ambiguous unless we define some *operator priority* and *associativity*. Operator priority, following the rules of basic arithmetic, is what tells us that a * b + c is equivalent to (a*b) + c and not a * (b + c). Associativity is what tells us that a + b + c is (a + b) + c and not a + (b + c). Associativity is most of the time left-to-right, like in the case of a + b + c, but it can be right-to-left like in exponentiation. Tiny does not not have exponentiation so all binary operators will associate left-to-right. In addition, some operators will be unary like -x or +x or **`not`** x. Parentheses **`(`** and **`)`** can be used to change the priority of operands if needed.

Let's recall first the definition of expressions in tiny.

$$\langle\text{expression}\rangle \rightarrow \langle\text{primary}\rangle \mid \langle\text{unary-op}\rangle \langle\text{expression}\rangle \mid \langle\text{expression}\rangle \langle\text{binary-op}\rangle \langle\text{expression}\rangle$$

This definition is not very useful because it does not define the priority of the operators. We defined, though, the priority of the operators in a table.

Operators	**Priority**
(unary)`+` (unary)`-`	Highest priority
`*` `/` `%`	
(binary)`+` (binary)`-`	
`==` `!=` `<` `<=` `>` `>=`	
`not`, `and`, `or`	Lowest priority

By following the table of priorities above, it is possible to derive the following syntax. The lower the level, the higher the priority of the operand.

$$\begin{aligned}
\langle\text{expression}\rangle &\rightarrow \langle\text{sixth-level}\rangle \\
\langle\text{sixth-level}\rangle &\rightarrow \textbf{not}\ \langle\text{sixth-level}\rangle \\
&\mid \langle\text{sixth-level}\rangle\ \textbf{and}\ \langle\text{fifth-level}\rangle \\
&\mid \langle\text{sixth-level}\rangle\ \textbf{or}\ \langle\text{fifth-level}\rangle \\
&\mid \langle\text{fifth-level}\rangle \\
\langle\text{fifth-level}\rangle &\rightarrow \langle\text{fifth-level}\rangle\ \mathbf{<}\ \langle\text{third-level}\rangle \\
&\mid \langle\text{fifth-level}\rangle\ \mathbf{<=}\ \langle\text{fourth-level}\rangle
\end{aligned}$$

| ⟨fifth-level⟩ **>** ⟨fourth-level⟩
| ⟨fifth-level⟩ **>=** ⟨fourth-level⟩
| ⟨fifth-level⟩ **==** ⟨fourth-level⟩
| ⟨fifth-level⟩ **!=** ⟨fourth-level⟩
| ⟨fourth-level⟩

⟨fourth-level⟩ → ⟨fourth-level⟩ **+** ⟨third-level⟩
| ⟨fourth-level⟩ **-** ⟨third-level⟩
| ⟨third-level⟩

⟨third-level⟩ → ⟨third-level⟩ ***** ⟨second-level⟩
| ⟨third-level⟩ **/** ⟨second-level⟩
| ⟨third-level⟩ **%** ⟨second-level⟩
| ⟨second-level⟩

⟨second-level⟩ → **+** ⟨second-level⟩
| **-** ⟨second-level⟩
| ⟨first-level⟩

⟨first-level⟩ → ⟨primary⟩

By restricting lower priority expressions in the right hand side of an expression (but allowing lower or equal priority expressions in the left hand side) we automatically force a left-to-right association. This is why a + b + c cannot be parsed as a + (b + c) because it would mean that in the right hand side of the first + directly appears another + operand, which is not possible because it has the same priority and we explicitly disallowed that in the syntax above.

Unfortunately we cannot apply our algorithm because some of the rules are *left-recursive*. A left-recursive rule is of the form ⟨rule⟩ → ⟨rule⟩ X. This means that our algorithm to parse the rule would need first to parse the rule but without having consumed any token from the input. So it would lead use to an infinite recursion. It is, indeed, possible to rewrite the rule so it is not left-recursive. For instance, ⟨third-level⟩ (and similarly the other left-recursive rules) can be rewritten as

⟨third-level⟩ → ⟨second-level⟩ ***** ⟨third-level⟩
| ⟨second-level⟩ **/** ⟨third-level⟩
| ⟨second-level⟩ **%** ⟨third-level⟩
| ⟨second-level⟩

but unfortunately this would change the association of the expressions: now they would be associated right-to-left. Most tiny operators will behave associatively (because the mathematical properties of the operations) so it would not make much difference in terms of evaluation but the integer division operator is not associative. Consider

```
write 100/10/2;
```

If we evaluate (100/10)/2 the result is 5. If we evaluate 100/(10/2) the result is 20. Since the semantics of the language call for left-to-right association the result in tiny must be 5.

Clearly we need another strategy: priority parsing.

The notion of priority appears more or less naturally in the syntax of expressions. Can we use it to get a more or less sensible algorithm? The answer is yes, it is called a Pratt parser and it is suprisingly simple yet powerful.

Pratt parser for expressions

A Pratt parser defines the concept of *binding power* as some sort of priority number: the higher the binding power the more priority the operand has. This parser associates three extra values to the tokens of expressions: a *left binding power*, a *null denotation* function and a *left denotation* function.

Parsing an expression requires a *right binding power*. A top level expression will use the lowest priority possible. Then the parser starts by peeking the current token t_1 and skipping it. Then it invokes the null denotation function of t_1. If this token cannot appear at this point then its null denotation function will diagnose an error and the parsing will end at this point. Otherwise the null denotation function will do something (that may include advancing the token stream, more on this later). Once we are back from the null denotation, the parser checks if the current right binding power is lower or than that of the current token (call it t_2, but note that it may not be the next one after t_1). If it is not, parsing ends here. Otherwise the parser skips the token and the left denotation function is invoked on t_2. The left denotation function (will do something, including advancing the current token, more on this later). Once we are back from the left denotation we will check again if the current token has a higher left binding power than the current right binding power and proceed likewise.

Ok, I tried, but the explanation above is rather dense. Behold the stunning simplicity of this parser at its core.

```
// This is a Pratt parser
bool
Parser::parse_expression (int right_binding_power)
{
  const_TokenPtr current_token = lexer.peek_token ();
  lexer.skip_token ();

  if (!null_denotation (current_token))
     return false;

  while (right_binding_power < left_binding_power (lexer.peek_token ()))
    {
      current_token = lexer.peek_token();
      lexer.skip_token ();

      if (!left_denotation (current_token))
         return false;
    }
  return true;
}

bool
Parser::parse_expression ()
{
  return parse_expression(LBP_LOWEST);
}
```

Intuitively the idea is that while we encounter tokens of higher priority than the priority of the expression we need to parse them first, otherwise if we find a lower priority token we stop parsing. This only makes sense if we recursively invoke `parse_expression`, that we will.

First let's see the null denotations. They represent the action that we have to do when we find a token at the *beginning* of an expression.

```
bool
Parser::null_denotation (const_TokenPtr tok)
{
  switch (tok->get_id ())
    {
    case Tiny::IDENTIFIER:
    case Tiny::INTEGER_LITERAL:
    case Tiny::REAL_LITERAL:
    case Tiny::STRING_LITERAL:
      return true;
    case Tiny::LEFT_PAREN:
      {
        if (!parse_expression ())
           return false;
        tok = lexer.peek_token ();
        return skip_token(Tiny::RIGHT_PAREN);
      }
    case Tiny::PLUS:
      {
        if (!parse_expression (LBP_UNARY_PLUS))
           return false;
        return true;
      }
    case Tiny::MINUS:
      {
        if (!parse_expression (LBP_UNARY_MINUS))
           return false
        return true;
      }
    case Tiny::NOT:
      {
        if (!parse_expression (LBP_LOGICAL_NOT))
           return false;
        return true;
      }
36     default:
37       unexpected_token (tok);
38       return false;
39     }
40 }
```

There is little to do now for identifiers, real, integer and string literals. So they trivially return true (lines 6 to 10).

If the current token is **(** (line 11) it means that we have to parse a whole expression. So we do by recursively invoking `parse_expression` (with the lowest priority possible, as if it were a top-level expression). When we return from parse_expression we have to make sure that the current token is **)** (line 16).

If the current token is +, - or not (lines 18, 24, 30) it means that this is a unary operator. We will invoke parse_expression recursively with the appropiate priority for each operand (`LBP_UNARY_PLUS`, `LBP_UNARY_NEG`, `LBP_LOGICAL_NOT`, more on this later).

It may not be obvious now, but `tok`, is not the current token in the input stream but the previous one since `parse_expression` already skipped `tok` before calling `null_denotation`.

The left denotation will be called for each token that can appear in an *infix* position. In tiny they will just be operators but sometimes other punctuation may appear.

```
bool
Parser::left_denotation (const_TokenPtr tok)
{
  BinaryHandler binary_handler = get_binary_handler (tok->get_id ());
  if (binary_handler == NULL)
    {
      unexpected_token (tok);
      return false;
    }

  return (this->*binary_handler) (tok);
}
```

Rather than making a relatively large switch (like we did in `null_denotation`), here we call a function that given a token will return us a pointer to the member function that implements the left denotation for token `tok`. We could have taken the same approach in the `null_denotation` function but given that there are much less unary operators it looked like unnecesary.

By using X-Macros again we define our binary handlers for further consumption.

```
struct Lexer {
  // ...
private:
  typedef bool (Parser::*BinaryHandler) (const_TokenPtr);
  BinaryHandler get_binary_handler (TokenId id);

#define BINARY_HANDLER_LIST                                                  \
  BINARY_HANDLER (plus, PLUS)                                           \
  BINARY_HANDLER (minus, MINUS)                                         \
  BINARY_HANDLER (mult, ASTERISK)                                       \
  BINARY_HANDLER (div, SLASH)                                           \
  BINARY_HANDLER (mod, PERCENT)                                         \
                                                                        \
  BINARY_HANDLER (equal, EQUAL)                                         \
  BINARY_HANDLER (different, DIFFERENT)                                 \
  BINARY_HANDLER (lower_than, LOWER)                                    \
  BINARY_HANDLER (lower_equal, LOWER_OR_EQUAL)                          \
  BINARY_HANDLER (greater_than, GREATER)                                \
  BINARY_HANDLER (greater_equal, GREATER_OR_EQUAL)                      \
                                                                        \
  BINARY_HANDLER (logical_and, AND)                                     \
  BINARY_HANDLER (logical_or, OR)

#define BINARY_HANDLER(name, _)                                         \
  bool binary_##name (const_TokenPtr tok);
  BINARY_HANDLER_LIST
#undef BINARY_HANDLER
  // ...
};
```

Function get_binary handler is implemented using `BINARY_HANDLER_LIST`.

```
Parser::BinaryHandler
Parser::get_binary_handler (TokenId id)
{
  switch (id)
    {
#define BINARY_HANDLER(name, token_id)                                  \
  case Tiny::token_id:                                                  \
    return &Parser::binary_##name;
      BINARY_HANDLER_LIST
#undef BINARY_HANDLER
    default:
      return NULL;
    }
}
```

Now we can provide implementations of the binary operators. At this point all of them will look the same, so let's consider only the binary addition.

```
bool
Parser::binary_plus (const_TokenPtr tok)
{
  if (!parse_expression (LBP_PLUS))
     return false;
  return true;
}
```

Finally we are only missing to define the left binding power of our tokens: recall that the higher is this number, the higher is the priority. This numbers fulfill the priority defined in the table above.

```
enum binding_powers
{
  // Highest priority
  LBP_HIGHEST = 100,

  LBP_UNARY_PLUS = 50,  // Used only when the null denotation is +
  LBP_UNARY_MINUS = LBP_UNARY_PLUS, // Used only when the null denotation is -

  LBP_MUL = 40,
  LBP_DIV = LBP_MUL,
  LBP_MOD = LBP_MUL,

  LBP_PLUS = 30,
  LBP_MINUS = LBP_PLUS,

  LBP_EQUAL = 20,
  LBP_DIFFERENT = LBP_EQUAL,
  LBP_LOWER_THAN = LBP_EQUAL,
  LBP_LOWER_EQUAL = LBP_EQUAL,
  LBP_GREATER_THAN = LBP_EQUAL,
  LBP_GREATER_EQUAL = LBP_EQUAL,

  LBP_LOGICAL_AND = 10,
  LBP_LOGICAL_OR = LBP_LOGICAL_AND,
  LBP_LOGICAL_NOT = LBP_LOGICAL_AND,

  // Lowest priority
  LBP_LOWEST = 0,
};


int
Parser::left_binding_power (const_TokenPtr token)
{
  switch (token->get_id ())
    {
    //
    case Tiny::ASTERISK:
      return LBP_MUL;
    case Tiny::SLASH:
      return LBP_DIV;
    case Tiny::PERCENT:
      return LBP_MOD;
    //
    case Tiny::PLUS:
      return LBP_PLUS;
    case Tiny::MINUS:
      return LBP_MINUS;
    //
    case Tiny::EQUAL:
      return LBP_EQUAL;
    case Tiny::DIFFERENT:
      return LBP_DIFFERENT;
    case Tiny::GREATER:
      return LBP_GREATER_THAN;
    case Tiny::GREATER_OR_EQUAL:
      return LBP_GREATER_EQUAL;
    case Tiny::LOWER:
      return LBP_LOWER_THAN;
    case Tiny::LOWER_OR_EQUAL:
      return LBP_LOWER_EQUAL;
    //
    case Tiny::OR:
      return LBP_LOGICAL_OR;
    case Tiny::AND:
      return LBP_LOGICAL_AND;
    case Tiny::NOT:
      return LBP_LOGICAL_NOT;
    // Anything that cannot appear in an infix position
    // is given the lowest priority
    default:
      return LBP_LOWEST;
    }
}
```

Wrap-up

Phew. This has been long. But now we are in a position to recognize the syntax of tiny. In the next chapter we will assess the semantic validity of the input.

That's all for today.

« A tiny GCC front end – Part 3 A tiny GCC front end – Part 5 »