STS Software Technology Systems

Compiler Construction
Summer Term 24
April 18
Lübke/Schupp

Grammar

Deadline : 02.05.24 08:00 AM (2024-05-02T08:00:00+02:00)

How to complete a project successfully?

Follow the rules as described in the Lecture!

GitLab at TUHH[1] is used as the submission system in this course. Your GitLab credentials are the same as for StudIP. If you cannot log into GitLab with your StudIP login and password, contact the teaching assistant immediately. All your code must be pushed to your group repository provided to you by the teaching assistant. Remember, that each group member needs to demonstrate their contributions through code commits. You can update your solution until the deadline. The latest commit before the deadline is considered by default. If you want your late submission to be graded for penalized maximal number of possible points, the **whole** group must send an email to the teaching assistant **before** the interview day.

How to get additional information?

You are encouraged to discuss past and present project sheets with the teaching assistants. Either approach the teaching assistant during the exercise session, or visit us during the weekly office hours. We are also available through e-mail or on the StudIP forum. We try to reply as quickly as possible and in general, you should get a reply the next weekday, but we cannot guarantee this.

[1] https://collaborating.tuhh.de/

Grammar and Parser for C++

To achieve our goal of writing a compiler, we first need to understand how to specify the grammar of the source language. In this project, you will therefore demonstrate that skill by defining a grammar according to a given textual description of a subset of C++. This project is based on course materials taught by Aarne Ranta[2] modified for our purposes.

Tasks

Task 1 Navigate to the language specification below and gradually build a grammar for BNFC that corresponds to the textual description there. For a perfect score, the generated BNFC code should correctly parse all programs in the test suite, not contain any reduce/reduce conflicts, and not more than 5 shift/reduce conflicts.

Summary

The objective of this lab is to write a parser for a fragment of the C++ programming language. The parser should return an abstract syntax tree at success, and report an error with a line number at failure. The recommended implementation of the parser is via a BNF grammar processed by the BNF Converter (BNFC) tool. The approximate size of the parser source code should be around 100 rules. With BNFC, only the grammar only has to be written and submitted.

Method

Build the grammar gradually, so that you can parse the seven test files in the given order: as your first goal, parse the first test file. Then the second, and so on. When you have passed one level, just try to parse the next example with your current parser and examine the line at which it fails. In this way, you will find out what new grammar rules you need. After treating the six tests, continue working on the grammar to make your parser perfect.

You can use any code generator (Haskell, Java, C,...) from BNFC. It doesn't need to be the same language as you plan to use in later labs. However, if you use Haskell, you have the advantage of info files, which tell you exactly where the conflicts are. Assuming your grammar file is `CPP.cf` and you have called `bnfc` on it, you can create an info file from the generated Happy file:

```
happy -i ParCPP.y
```

[2] http://www.cse.chalmers.se/edu/year/2018/course/DAT151

The info file is now in `ParCPP.info`. Moreover, you can produce a debugging parser by

```
happy -da ParCPP.y
```

Running this parser shows a trace of the LR actions performed during parsing.

Language specification

It is suitable to explain and advisable to implement the parser top-down, starting with the largest units and proceeding to smaller ones. The specification differs in some places from the official C++ specification. To help you get going, we have marked with a bold **(n)** those rules that are needed to parse the n-th test file.

Programs

A program is a sequence of definitions. **(1)** A program may also contain comments and preprocessor directives, which are just ignored by the parser (see below). **(1)**

Definitions

a) A function definition has a type, a name, an argument list, and a body. **(1)** A function can optionally be prefixed by `inline`. Example:

```
inline int foo(double x, int y)
{
  return y + 9 ;
}
```

b) Many statements can be used as top-level definitions:

- typedef statements **(3)**
- variable declarations and initializations
- structure declarations

c) Finally, definitions for using qualified constants are allowed, **(2)** e.g.

```
using std::vector ;
```

Argument lists, declarations, and function bodies

An argument list is a comma-separated list of argument declarations. It is enclosed in parentheses `(` and `)`. **(1)** An argument declaration always has a type. This type is optionally followed by an identifier or an identifier and an initialization, and optionally preceded by the specifier `const`. **(4)** The following are examples of argument declarations.

```
int
int x
int x = 5
const int& x
```

Notice that argument declarations with multiple variables (`int x, y`) are not included. A declaration that occurs as a statement (as shown below), can also have multiple variables. But it must have at least one variable. The reference operator `&` is an optional part of the declaration, appearing right after the type. It can alternatively be defined as a type-forming constructor. A function body is either a list of statements enclosed in curly brackets { and } **(1)**, or an empty body consisting of a semicolon `;` **(6)**, for instance:

```
int foo(double x, int y) ;
```

Statements

Any expression followed by a semicolon `;` can be used as a statement. **(1)** Any variable declaration followed by a semicolon `;` can be used as a statement. **(2)** Variable declarations have the same form as argument declarations in functions, except that they can have more than one variable, cannot be empty, and allow nonrepetitive specifiers `const` and `constinit`, for example:

```
int x;
int x = 5, y, z = 3;
const int x, y = 0;
const int& x, y;
constinit int x = 0;
constinit const int x = 0;
```

Statements returning an expression **(1)**, for example

```
return i + 9 ;
```

While loops, with an expression in parentheses followed by a statement **(3)**, for example:

```
while (i < 10) ++i ;
```

Do-while loops, with an expression in parentheses after the loop body **(6)**, for example:

```
do ++i ; while (i < 10) ;
```

For loops, with a declaration and two expressions in parentheses followed by a statement. **(6)** For example:

```
for (int i = 0 ; i != 10 ; ++i) k = k + i ;
```

We do not require that any of the fields in parentheses may be empty.

Conditionals: if with an expression in parentheses followed by a statement and optionally by else and a statement. **(3)** Examples:

```
if (x > 0) return x ;

if (x > 0) return x ; else return y ;
```

Exception-handling: try followed by a statement, and by catch with an argument declaration in parentheses, which must not have an initialization, and a statement. **(7)** Examples:

```
try
  throw 42;
catch(int ex)
  fprintf(stderr, "error-%d\n", ex);

try
  maybe_crash();
catch(const int&)
  fprintf(stderr, "error!\n");
```

Blocks: any list of statement (including empty list) between curly brackets. **(3)** For instance,

```
{
    int i = 2 ;
    {
    }
    i++ ;
}
```

Type definitions: a type and a name for it. **(3)** Example:

```
typedef vector<string> Text ;
```

Structure definitions: a name, optionally followed by a colon (:) and a comma-separated list of types **(7)**, and a list of declarations. Example:

```
struct Student_info {
    string name;
    double final;
    vector<double> homework;
} ;

struct Extended_info : Student_info {
  string email;
} ;
```

Notice that the semicolon is obligatory in the end of a structure definition. Otherwise, semicolons are not used after curly brackets - but they are obligatory in all statements and definitions not ending with curly brackets.

Expressions

The following table gives the expressions and their precedence levels and associativity types.

Note. The table is not exactly the same as in the C++ standard.

level	expression forms	assoc	explanation
15	literal		atomic expressions **(1)**
15	`c::i`		qualified constants **(1)**
14	`e[i]`	left	indexing **(3)**
14	`e(e,...,e)`	left	function call **(3)**
14	`e.e`, `e->e`	left	structure projection **(3)**
14	`e++`, `e--`	left	in/decrement
13	`++e`, `--e`, `*e`, `!e`	right	in/decrement, dereference, negation **(6)**
13	`+e`, `-e`	right	unary plus/minus
12	`e*e`, `e/e`, `e%e`	left	multiplication, division, remainder **(3)**
11	`e+e`, `e-e`	left	addition, subtraction **(3)**
10	`e<<e`, `e>>e`	left	left and right shift / stream operators **(1)**
9	`e<=>e`	left	three-way comparison
9	`e<e`, `e>e`, `e>=e`, `e<=e`	left	comparison **(6)**
8	`e==e`, `e!=e`	left	(in)equality **(3)**
4	`e&&e`	left	conjunction **(6)**
3	`e||e`	left	disjunction **(6)**
2	`e=e`, `e+=e`, `e-=e`	right	assignment **(3)**
2	`e ? e : e`	right	conditional **(3)**
1	`throw e`	right	exception throwing **(7)**

Notice that it is syntactically permitted to increment and assign a value to any expression, not just variables (and other so-called lvalues).

Qualified constants and template instantiations

Qualified constants **(1)** are constant names separated by `::` as in `std::cout`. Names can be identifiers but also template instantiations, of the form

```
ident < typelist >
```

where a `typelist` is a comma-separated list of types. Thus possible qualified constants include

```
std::vector<t>::const_iterator
std::map<int, vector<string>>
```

Qualified constants without templates are needed in **(1)**.

Types

Types are either qualified constants (including plain identifiers **(1)**, e.g., `string`), type references **(4)**, e.g, `int &`, and built-in types **(1)**, of which we include the following:

int
bool
char
double
void

Literals

We include integer literals **(1)**, e.g., `0`, `00`, `20`, `151`, floating point literals **(3)**, e.g., `01.10`, `3.14`, `10.0e10`, `0.00e-00`, `1.1e-80`, character literals **(6)**, e.g., `'0'`, `'a'`, `'"'`, `"'"`, and double-quoted string literals **(1)**, e.g., `"abc"`, `"This,\n is \"insane\"!"`, `"LaTeX's \"\\newcommand\""`, e.g. , and . A string literal may consist of many concatenated strings and may be divided over lines in this way **(3)**:

```
"hello-" "my-little-"
"world"
```

Identifiers

An identifier is a letter followed by a list of letters, digits, and underscores. **(1)**

Comments

There are three kinds of comments.

- anything between tokens `/*` and `*/` **(1)**
- anything from token `//` to the end of a line or the file **(1)**
- anything from token `#` to the end of a line or the file (preprocessor directive)

Test programs

Most of these programs (a – f) are original code from the web page of the book Accelerated C++ (A. Koenig & B. Moo, Addison-Wesley, 2000).

a) `hello.cc`: "Hello, world!"

b) `greet.cc`: ask the user's name and say hello to them

c) `med.cc`: compute the grade of a student

d) `grade.cc`: a smarter way to compute the grade

e) `palin.cc`: test if a string is a palindrome

f) `grammar.cc`: random-generate English sentences

g) `exception.cc`: division

Once again: build the grammar gradually so that you can parse these files in this order.

Success criteria

Your grammar must pass the test suite. The test suite contains the example programs, as well as a number of programs which your parser must reject. The maximum number of 5 shift/reduce conflicts is permitted. The minimum of 2 is almost unavoidable (the "dangling else", and template instantiation vs. less-than operator). But reduce/reduce conflicts are forbidden.

Your parser must pass the testsuite available on INGInious[3] accessible from the TUHH network or through the TUHH VPN. You can log in with you usual TUHH user account (the one you also use for StudIP).

After you log in, if you haven't done so already, click the *Course List* and register for the *Compiler Construction* course. When you're registered, click *My Courses* in the side bar, select *Compiler Construction*, and then the *Grammar* task. Use the *Choose File* button to select your grammar file and then click *Submit*. Please, be patient and do not resubmit the same solution to the same problem.

When all tests are run you'll see a red or green box (depending on whether your submission already passes all tests). In this box, there are two sections, *debug info* and *test output*. Most of the time, you can ignore the *debug info*, but please always include it when you encounter problems with INGInious and ask for help. The *test output* will tell you how many tests your submission already passed, and gives hints on how to pass any remaining tests.

Submission

Submit your solution via GitLab[4]. If you have any problems getting the test program to run, or if you think that there is an error in the test suite, contact the teaching assistant (please include debug info if shown).

Grading

You can earn up to 30 points in this project phase.

3 points are awarded for readability of your grammar file. This includes but is not limited to: sensible naming of identifiers and rules, the logical structure of the file content, as well as the use of comments. In essence: make sure someone can understand your grammar without any further explanations.

[3] https://inginious.sts.tuhh.de
[4] https://collaborating.tuhh.de

The remaining 27 points are awarded for your grammar / parser implementation, split evenly between definitions, statements, and expressions (9 points each).

C++ features

Assuming you know some C but not C++, here is a summary of the extra features of C++ involved in this lab.

- qualified names: `s::n`, where `s` is a name space or a class.
- using directives: license to use an unqualified name from a name space.
- IO streams: `cout` for output, `cin` for input. Output is produced by the left shift operator, input is read by the right shift. Example from the second program:

```
// read the name
std::string name;       // define 'name'
std::cin >> name;       // read into 'name'

// write a greeting
std::cout << "Hello, " << name  << "!" << std::endl;
```

- templates, e.g. generic types: `vector<int>` is a vector of integers.
- arguments passed by reference (`&`), with the possibility of forbidding modification (`const`), e.g. (from the program `grammar.cc`)

```
gen_aux(const Grammar& g, const string& word, vector
    <string>& ret)
```

- qualifier `constinit` requiring constant initialization of the variable.