Compiler Construction

Sibylle Schupp[1]

[1]Institute for Software Systems/Institut für Softwaresysteme
Hamburg University of Technology (TUHH)

SoSe16

Lecture 1

Outline

Welcome!

http://www.sts.tuhh.de

Compiler Construction is brought to you by the

Institute for Software Systems

Lecturer

- Prof. Sibylle Schupp

Teaching assistant

- Sven Mattsen

STS
Software Technology Systems
TUHH

When, where, & what

- Lectures: weekly
- Projects: throughout the term (4 deadlines)
- Course homepage
  - Stud.IP is a "learning platform"
  - Go to https://e-learning.tu-harburg.de
  - Login with your TUHH-username/password
  - Search for course "Compiler Construction"
- **All** relevant material accessible from Stud.IP
  - Lecture notes
  - Project description
  - News, announcements, ...
- The course language is English.

What is a compiler?

A compiler is a program that translates program code in some language to code in another language.

- We distinguish different languages
  - Source language (subject language):
    - High-level programming languages as well as low-level languages
  - Target language:
    - Traditionally, a compiler translates to machine code.
    - Compilers that target high-level programming languages are sometimes called *source-to-source compilers*.
  - Implementation language: independent from the other two languages
- Ex.:
  - Gnu GCC translates C programs to machine code, javac translates Java programs to JVM, mono translates C# programs to CLR, GHC translates Haskell programs to LLVM, C or native code.

Machine code

http://images.wisegeek.com/green-lit-numbers.jpg

Binary encoding

- All data can be encoded in bits
  - Integers:

0	0
1	1
2	10
3	11

  - Characters, e.g., ASCII 7-bit rep:

A	65 (dec)	1000001
B	66 (dec)	1000010
C	67 (dec)	1000011

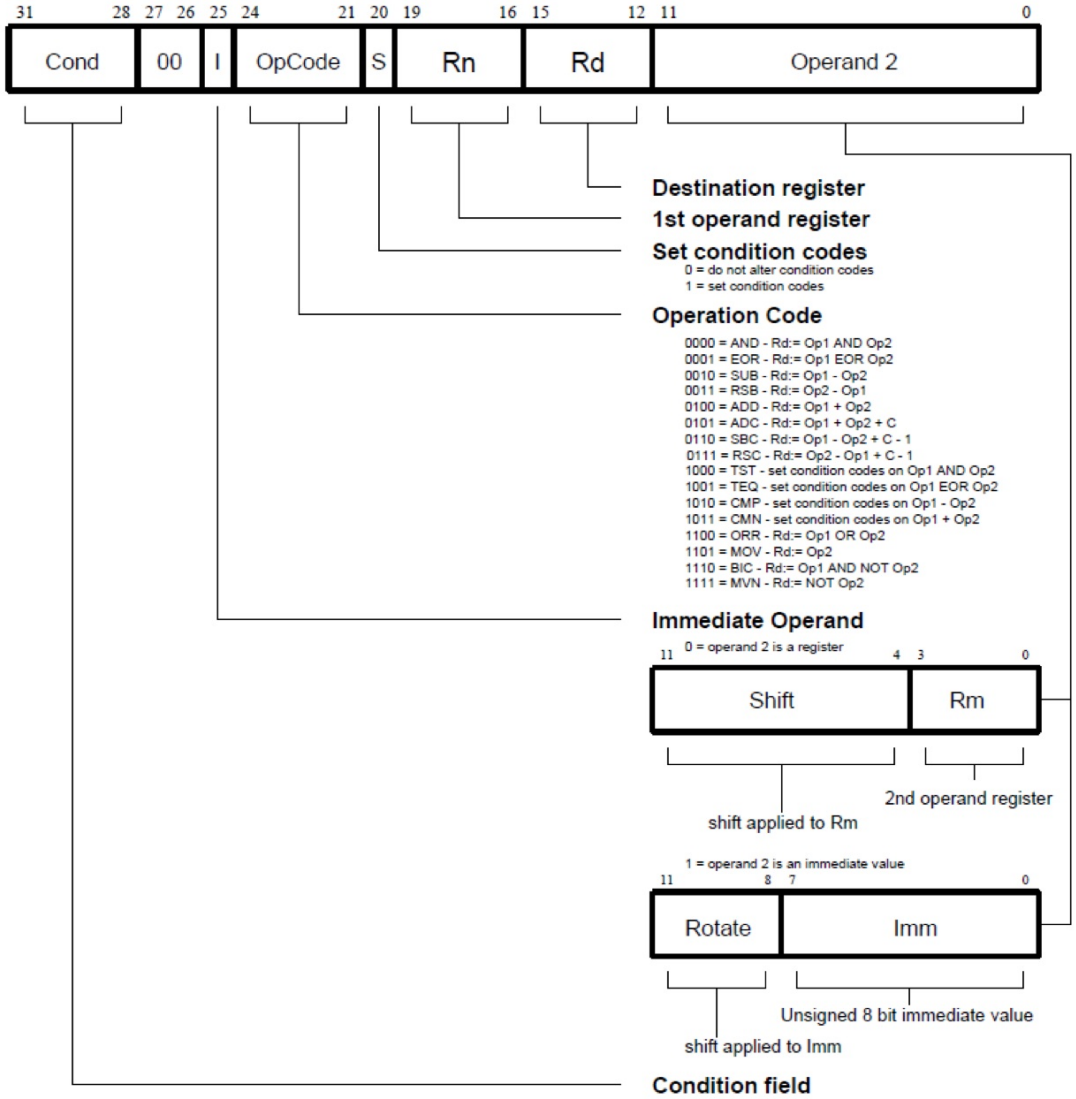
- Instructions can be encoded in bits as well.

  Ex.: ARM opcodes:

AND	0000
SUB	0010
ADD	0100

Example

http://stackoverflow.com/questions/11785973/converting-very-simple-arm-instructions-to-binary-hex



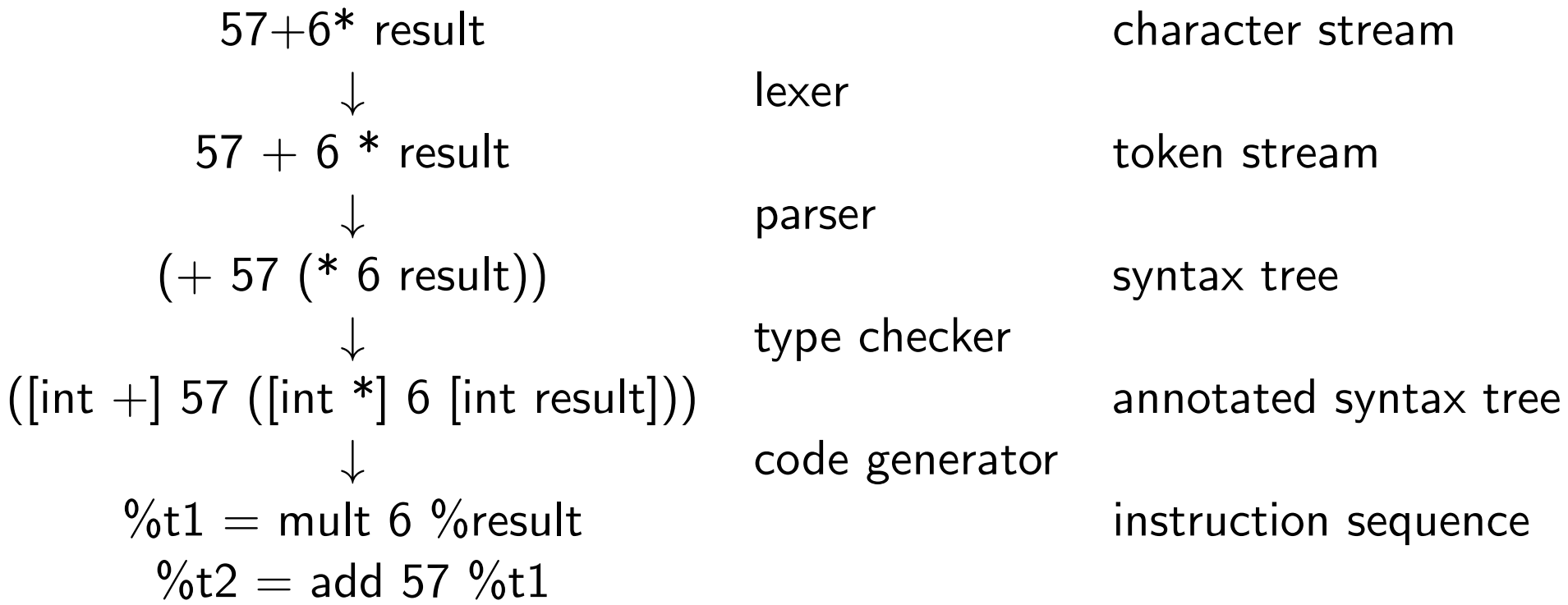
Syntax-directed translation

Compilation can be implemented in a systematic way by following the principle of syntax-directed translation.

Example (syntax-directed translation of a binary expression):

1. Analyze the expression: identify operator O, operand A1, operand A2
   - In general: analyze the input and break it down recursively in smaller parts
2. Compile the code for A1, compile the code for A2, compile the code for O
   - More generally: compile smaller parts, as defined by the syntactic analysis

Compilation phases



Phases that analyze the subject program are part of the *front end* (lexer, parser, type checker), phases that synthesize code are part of the *back end* (code generator) of a compiler.

Compilation errors

Each compilation phase in the front end identifies specific errors in the subject program:

- Lexer: illegal character. Ex.: ?x
- Parser: ill-formed "sentences", e.g., arithmetic expressions.
  Ex: 4 * 5 6
- Type checker: 3.0 + "8"

In-class exercise

IPL, Ex.1.2

Optimizing compiler

The classical compiler pipeline is a simplification.

- All industrial-strength compilers include optimzation passes.
- Optimization takes place at many different levels
  - Computing at compile time (address calculation, constant expressions)
  - Improving loop nests
  - Packing data structures
  - Rearranging computations
  - Eliminating code
  - ...
- Many optimizations require their own kind of analysis and synthesis.

History & future

- Traditionally, compiler construction was part of artificial intelligence.
- One of the fields in computer science that shaped the discipline:
  - Fundamental data structures of computer science
  - Theoretical foundations of computer science
  - Focus
    - 1940s-1960s: code generation
    - 1960s-mid 1970s: parsing
    - late 1970s-today: typing, programming paradigms, optimization
  - Current topics
    - Compilation for DSLs (domain-specific languages)
    - Multicore
    - Optimiziation for resources (energy consumption, . . .)
    - Software-engineering topics (correctness, program restructuring, . . .)

Project-based course

This course is entirely project-based. No exam.

- Your task is a compiler project:
  - Divided in a sequence of subprojects
  - Defined along major compilation phases
    1. Grammar and parser
    2. Type checker
    3. Code generator
    4. Paradigm-specific extension (your choice)
- **All work takes place during the lecture period**.
- Clear focus on writing software

Project organization

- You work in teams of 3 students.
  - Form a group and register in StudIP by Tuesday at the latest.
  - If you can't form a group of 3, send email to the course assistant.
- You organize the tasks mostly independently. You are also responsible for setting up meetings in your group.
  - In the exercise session, you can ask for help and feedback.
- You upload your artifacts electronically through StudIP.
  - Each artifact must contain the name of the author. Multiple authors are permitted. Indicate if there is a main author.
- You meet with the course assistant roughly bi-weekly.
  - The meeting is mandatory.
  - Those meeting are **the only mandatory part** of the course.
  - They take place during the exercise session.

Prerequisites

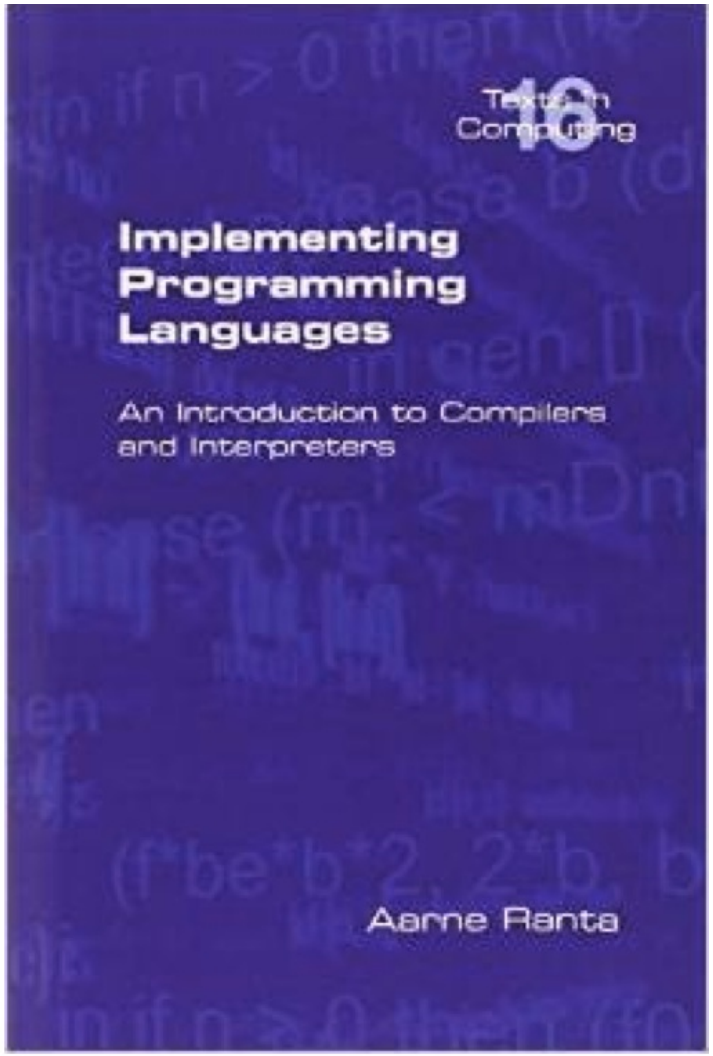
Coding-intensive course. Upper-level course for computer science majors.

- You should have:
  - Knowledge of at least one higher programming language
  - Project/programming experience beyond homework assignments, e.g., through programming labs or small software projects
  - Knowledge of the fundamental data structures of computer science (stack, list, tables, tree, graphs)
  - All students in the IIW/CS program meet the prerequisites; others have to catch up.
- Further, you must have the **time** to take the course
  - Five deadlines during the term. Must work regularily during the term
  - Must be able to attend the mandatory meetings
  - No exam. No peak performance required
- No prior experience with compiler construction expected
  - We take a systematic approach and learn step by step how to write a compiler.

Implementation language

- You can choose among different programming languages (see later today for more details)
  - Haskell, C++ , C#, OCaml, Java, ...
  - Good opportunity to learn a new language. You may use the time before the first project is out.
  - Could also use the time to brush up your previous knowledge.
  - You may make your own suggestion for the later subprojects.
- **Note**: we provide support only for Haskell and C++ .

Textbook


Texts in Computing 16
Implementing Programming Languages
An Introduction to Compilers and Interpreters
Aarne Ranta

Lecture

- Lecture covers chapters 1-4 and 6-7 of [IPL]
  - Project parts aligned with lecture topics
- Lecture notes available in the evening before class
  - Figures and slides from [IPL] unless said otherwise
- Questions during the lecture are welcome!

Grades

Percentage	Grade
51-55	4.0
56-60	3.7
61-65	3.3
66-70	3.0
71-75	2.7
76-80	2.3
81-85	2.0
86-90	1.7
91-95	1.3
96-100	1.0

Grading

In the course of the term, each group accumulates points.

- With each subproject, a group earns a number of points.
  - Details on the project sheets
  - Might earn bonus points (percentage) for various code quality parameters
- Mandatory meetings:
  - The entire group meets with the teaching assistant
  - 3 meetings in the first half, 3 meetings in the second half; you need to arrange for those meetings
  - Demo a snapshot of your work, be prepared to answer questions
  - If group performs inhomogeneously, individual weights are assigned to the points earned in the respective subproject.
    - Ex.: Assume a group size of 3. If all members roughly did the same amount of work, everyone gets all points earned. If someone did less than 1/3 of the work, we deduct points.

Deliverables

You will work on a compiler for (a realistic subset) of the C++ language:

		out	due	#p	#bp	LOC (Haskell)
0.	Installation	W1	W3	5	-	
1.	Parser	W3	W6	20	-	100 rules
2.	Type checker	W6	W8	25	5	300-800
3.	Code generator	W8	W10	25	5	300-800
4.	Paradigm-specific choice	W10	W13	25	10	

- You can earn points (p) and bonus points (bp). The table shows the maximal possible number. For the deadlines: see project descriptions.
- Deliverable 0: you need to install the tool chain (see below), preferably this week.
- Deliverable 4: you will extend your compiler in a paradigm-specific way. The particular task you will define yourself.

What if . . . ?

- The group missed the deadline for
  - deliverable 0? Then you simply lost the points.
  - deliverable 1-4 or if you want to better your earnings? You may submit late (or resubmit) by a later deadline, but you will miss points; the maximal number of possible points will be reduced with each deadline.

Del.	No. 1	No. 2	No. 3	No. 4	July 23
Nr.1	20	15	10	5	-
Nr.2	-	25	20	15	10
Nr.3	-	-	25	20	15
Nr.4	-	-	-	25	20

- July 23 is the **last** safety net. Note that one cannot pass the course if one postpones the entire course work to that date.
- For the final grade, the last submission counts (even in case your performance worsened).

What if . . . ? (part 2)

- An individual student missed the group meeting with the teaching assistant
  - with a proper execuse (as accepted by the examination office): a new meeting will be arranged
  - without an excuse: the points of the subproject do not get counted
- If you fail the entire course, please be aware of the fact that the course is not offered in the winter.
  - Next possibility: next summer
  - **No exception possible**

Course load

Depending on your FSPO, the course counts either 4 or 6 ECTS. The course load is accordingly:

Old FSPO	Current FPSO
Lectures	
Deliverable 0,1,2	
Deliverable 3 **or** 4 3+1 mandatory meetings	Deliverable 3 **and** 4 3+3 mandatory meetings

For the old FSPO, 100% is correspondingly adjusted.

Rules: no cell phones

. . . and even more rules: no pictures

Source: randomwire.com

Most material is electronically available. For everything else, we keep the right.

Outline

BNFC Converter (BNFC)

- BNFC is a compiler compiler generator.
  - BNFC is based on *labeled BNF grammars* .
  - Rules are of the form

    Label . Category ::= Production;
- From the labeled grammar, BNFC *generates* automatically the most important tools for writing compilers:
  - Lexer, parser, abstract syntax tree, tests.
  - Those tools are generated for a variety of implementation languages: C, C++, C#, Haskell, OCaml, Java

Example (BNFC)

Here is a first taste of a labeled BNF grammar

```
EAdd. Exp  ::= Exp  "+" Exp1 ;
ESub. Exp  ::= Exp  "-" Exp1 ;
EMul. Exp1 ::= Exp1 "*" Exp2 ;
EDiv. Exp1 ::= Exp1 "/" Exp2 ;
EInt. Exp2 ::= Integer ;

coercions Exp 2 ;
```

(Later more on *coercion* and the digit suffixes.)

Using BNFC

BNFC is invoked from the command line using `bncf`.

```
12> bnfc -m Calc.cf

8 rules accepted

Use Alex 3.0 to compile LexCalc.x.
ParCalc.y Tested with Happy 1.15
writing new file AbsCalc.hs
writing new file LexCalc.x
writing new file ParCalc.y
writing new file TestCalc.hs
writing new file DocCalc.txt
writing new file SkelCalc.hs
writing new file PrintCalc.hs
writing new file ErrM.hs
writing new file Makefile
```

- `bnfc` generates different files for the different compiler components.
- Default language for the compiler is Haskell.

Generating C++ compiler components

BNFC can also generate compiler components in C++ .

```
> bnfc -m --cpp Calc.cf

8 rules accepted

writing new file Absyn.H
writing new file Absyn.C
writing new file Calc.l
writing new file Calc.y
writing new file Parser.H
writing new file Skeleton.H
writing new file Skeleton.C
writing new file Printer.H
writing new file Printer.C
writing new file Test.C
writing file Makefile (saving old file as Makefile.bak)
```

- One can control the implementation language by setting the appropriate flag.

Makefiles

https://www.gnu.org/software/make/

- A *Makefile* is a common part of an automated build process for software. It contains instructions for compiling and linking a program.
- A Makefile specifies dependencies between different program parts and allows for parameterization via variables.
- The Makefile is input to the `make` utility that invokes those instructions.
- `make` was introduced in 1976 for the Unix operating system. His creator, Stuart Feldman, won the ACM Software System Award for it in 2003.
- Further development exists, but `make` is still in use.

For BNCF, Makefiles can be treated as black boxes. One only needs to invoke them.

Invoking the BNCF Makefile

A Makefile is invoked from the command line using `make`.

The rules for the BNCF Makefile

- . . . for Haskell invoke the programs `ghc`, `alex`, and `happy`.
- . . . for C++ invoke the programs `g++`, `flex`, `bison`.

Those programs need to be installed beforehand.

Compiling Haskell compiler components

```
> make -f Makefile
happy -gca ParCalc.y
alex -g LexCalc.x
ghc --make TestCalc.hs -o TestCalc
[1 of 7] Compiling ErrM             ( ErrM.hs, ErrM.o )
[2 of 7] Compiling AbsCalc          ( AbsCalc.hs, AbsCalc.o )
[3 of 7] Compiling PrintCalc        ( PrintCalc.hs, PrintCalc.o )
[4 of 7] Compiling SkelCalc         ( SkelCalc.hs, SkelCalc.o )
[5 of 7] Compiling LexCalc          ( LexCalc.hs, LexCalc.o )
[6 of 7] Compiling ParCalc          ( ParCalc.hs, ParCalc.o )
[7 of 7] Compiling Main             ( TestCalc.hs, TestCalc.o )
Linking TestCalc ...
```

Compiling C++ compiler components

```
> make
g++ -g -c Absyn.C
flex -oLexer.C Calc.l
g++ -g -c Lexer.C
Calc.l:42:68: warning: control reaches end of non-void function
[-Wreturn-type]
int initialize_lexer(FILE *inp) { yyrestart(inp);
                                  BEGIN YYINITIAL; }
                                                    ^


1 warning generated.
bison Calc.y -o Parser.C
g++ -g -c Parser.C
g++ -g -c Printer.C
g++ -g -c Test.C
Linking TestCalc...
g++ -g *.o -o TestCalc
```

Generating and compiling Ocaml compiler components

```
> bnfc -m --ocaml Calc.cf

8 rules accepted

writing new file AbsCalc.ml
writing new file LexCalc.mll
writing new file ParCalc.mly
writing new file SkelCalc.ml
writing new file PrintCalc.ml
writing new file ShowCalc.ml
writing new file TestCalc.ml
writing new file BNFC_Util.ml
writing file Makefile (saving old file as Makefile.bak)
l1> make
ocamlyacc ParCalc.mly
ocamllex LexCalc.mll
19 states, 558 transitions, table size 2346 bytes
ocamlc -o TestCalc BNFC_Util.ml AbsCalc.ml SkelCalc.ml
    ShowCalc.ml PrintCalc.ml ParCalc.mli ParCalc.ml
    LexCalc.ml TestCalc.ml
```

. . . Java compiler components, anyone?

```
 bnfc -m --java Calc.cf

8 rules accepted

   (Tested with JLex 1.2.6.)
   (Parser created for category Exp)
   (Tested with CUP 0.10k)
writing new file Calc/Absyn/Exp.java
writing new file Calc/Absyn/EAdd.java
writing new file Calc/Absyn/ESub.java
writing new file Calc/Absyn/EMul.java
writing new file Calc/Absyn/EDiv.java
writing new file Calc/Absyn/EInt.java
writing new file Calc/PrettyPrinter.java
writing new file Calc/VisitSkel.java
writing new file Calc/ComposVisitor.java
writing new file Calc/AbstractVisitor.java
writing new file Calc/FoldVisitor.java
writing new file Calc/AllVisitor.java
writing new file Calc/Test.java
writing new file Calc/Yylex
writing new file Calc/Calc.cup
writing new file Makefile
```

. . . Java (2)

```
> bnfc -m --java Calc.cf
> make
javac -sourcepath . Calc/Absyn/Exp.java Calc/Absyn/EAdd.java Calc↩
    /Absyn/ESub.java Calc/Absyn/EMul.java Calc/Absyn/EDiv.java ↩
    Calc/Absyn/EInt.java
java  JLex.Main Calc/Yylex
Error: Could not find or load main class JLex.Main
make: *** [Calc/Yylex.java] Error 1
```

- Remember: `bnfc` depends on certain tools, here JLex and Cup.
- You might have to inform the Java interpreter `java` about their location. Set the `CLASSPATH` variable:
  `export CLASSPATH=./<your-path-to-JLex>`

Example (demo BNCF)

The example program `Calc.cf` ships with a test program, `TestCalc`. Its input is of I/O type.

```
> echo "2 * 3 + 4" | ./TestCalc

Parse Successful!

[Abstract Syntax]

EAdd (EMul (EInt 2) (EInt 3)) (EInt 4)

[Linearized tree]

2 * 3 + 4
> cat input
4 * 5 + 6
> ./TestCalc < input
```

Observe the compilation *pipeline*:

- Parsing, abstract syntax tree (AST), linearization
- The AST is the output of a parser and the input to the linearizer (which returns a flattened tree).

Outline

What is a grammar?

A grammar is a set of rules that define a language. The rules define:

- all permitted symbols;
- all permitted combinations of symbols, including {required, possible, illegal} orders of symbols;

In this course, two kinds of languages are relevant:

- (General-purpose) programming languages
- "Domain-specific languages" (short DSL)

It is quite common to consider protocols or specialized computations as specialized, domain-specific languages.

Example (grammar of arithmetic expressions)

Consider the language of "arithmetic expressions." Its grammar defines all legal *sentences* in this language in a compact way.

expr	$\rightarrow$	*expr op expr*
expr	$\rightarrow$	(*expr*)
expr	$\rightarrow$	- *expr*
expr	$\rightarrow$	int
op	$\rightarrow$	+
op	$\rightarrow$	*
op	$\rightarrow$	/
op	$\rightarrow$	-

Notation

- Technically, a grammar is a set of *productions*.
  - Ex.: "$expr \rightarrow expr\ op\ expr$" is a production.
- A production consists of terminals (a.k.a tokens) and non-terminals.
  - Terminals are basic symbols. They never occur on the left-hand side.
  - Non-terminals occur at least once on the left-hand side.
  - Terminals and non-terminals use different notation, e.g., different fonts or quotes.
- By default, the start production is the first production.
- Productions with the same left-hand side can be combined using | ("alternative")
  - $expr \rightarrow expr\ op\ expr$ | ($expr$) | -exp | integer
  - $op \rightarrow +$ | * | - | /

Derivations

A production can be considered a *rewrite rule*, where (the non-terminal on) the left-hand side is replaced by the right-hand side.

- The right-hand side is said to be *derived* from the left hand side.
  - Ex.: *expr op expr* is derived from *expr*.
- A sequence of rewrites is called a *derivation*.
- A derivation starts with the start production. It stops when all non-terminals are replaced by terminals. The resulting string is called a *sentence*.
- Given a grammar *G* with start production S. Then L(G), the language generated from *G*, is the set of all possible sentences.

Example (derivation)

Given the grammar G of the language of arithmetic expressions. Here is an example of a derivation in G:

$$\textit{expr} \rightarrow_{\textit{rewrites}} \textit{-expr} \rightarrow \textit{-(expr)} \rightarrow \textit{-(expr + expr)}$$
$$\rightarrow \textit{-(int + expr)} \rightarrow \textit{-(int + int)}$$

- Thus, -(int + int) is a sentence in L(G).
- Counterexample: "- (int +)" is not a sentence: it cannot be derived.
- Another counterexample: -(3 + 4)

Derivation strategies

- From a given non-terminal, more than one derivation sequence might be possible.
  - For a single derivation step, one needs to decide which non-terminal should be replaced and by what alternative (if the production contains alternatives).
- A derivation *strategy* denotes a systematic application of rewrites.
- The two most common strategies are leftmost and rightmost derivations.
  - In a leftmost derivation strategy, each derivation step rewrites the leftmost non-terminal.
  - In a rightmost derivation strategy, each derivation step rewrites the rightmost non-terminal.

Summary

- Source language, target language, implementation language
- Compilation phases; front end, back end; compilation errors; optimizing compilers
- bnfc
- Makefile and make
- Grammar; legal sentences, L(G), production, derivation (strategy)

References

- IPL, Ch. 1