Compiler Construction

Sibylle Schupp[1]

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

SoSe16

Lecture 3

TUHH STS Software Technology Systems

Outline

Example (work flow)

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

> bnfc -m CPP.cf
> make
> ./TestCPP mytest.cpp
mytest.cpp

Parse Successful!

[Abstract Syntax]

PDefs [DFun Type_int (Id "foo") [ADecl Type_double (Id "x"),
ADecl Type_int (Id "y")] [SReturn (EPlus (EId (Id "y"))
(EInt 9))]]

[Linearized tree]

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

Example: abstract syntax in C++

```
// Absyn.H
class SExp : public Stm {
public:
  Exp *exp_;
  SExp(const SExp &);
  SExp &operator=(const SExp &);
  SExp(Exp *p1);
  ~SExp();
  virtual void accept(Visitor *v);
  virtual SExp *clone() const;
  void swap(SExp &);
};
class SDecls : public Stm {
public:
  Type *type_;
  ListId *listid_;
  SDecls(const SDecls &);
  SDecls &operator=(const SDecls &);
  SDecls(Type *p1, ListId *p2);
  ~SDecls();
  virtual void accept(Visitor *v);
  virtual SDecls *clone() const;
  void swap(SDecls &);
};
```

Representing abstract syntax

- In C++ or Java, each category and each label is represented as a class
  - A category is represented as an abstract base class.
  - A label is represented as a derived class.
- In Haskell, there is one algebraic data type per category
  - An algebraic data type has constructors. Those are exactly the labels ("constructors") of a BNFC production.
  - Examples of constructors in CPP:
    `PDefs, SExp, SDecls, SInit`

Example: abstract syntax in Haskell

In Haskell, the abstract syntax of grammar is implemented using *algebraic data types*:

```
{- AbsCPP.hs
PDefs. Program ::= [Def] ;
..
SExp. Stm    ::= Exp ";" ;
SDecls. Stm  ::= Type [Id] ";" ;
SInit. Stm   ::= Type Id "=" Exp ";" ;
SReturn. Stm ::= "return" Exp ";" ;
SWhile. Stm  ::= "while" "(" Exp ")" Stm ;
-}

data Program =
   PDefs [Def]
  deriving (Eq,Ord,Show,Read)
data Stm =
   SExp Exp
 | SDecls Type [Id]
 | SInit Type Id Exp
 | SReturn Exp
 | SWhile Exp Stm
  deriving (Eq,Ord,Show,Read)
```

What is a parser?

Given a grammar for the subject language. The parser gets a sequence of tokens from the lexer.

- The main task of the parser is to check whether this sequence forms a sentence of the given grammar of the subject language.
- If yes, it returns the abstract syntax tree of the sequence.
- If no, it should provide a useful error message.
- Modern parsers are also expected to provide *error recovery.*

Several parsing strategies exist. One can:

- build the AST top-down or button-up;
- consider one token at a time or include lookahead;
- process the input from the start or from the end.

Context-free grammars

- Context-free grammars are the same as BNF grammars. They allow specifying matching parentheses
  - `{ ... }`
  - `begin ... end`
  - ...
- In contrast, regular expression are not expressive enough to specify most modern programming languages.
- In their most general form, CFGs are expensive to parse (cubic costs)
- Most programming languages are designed so that parsing takes linear time. Also, they require non-ambiguous grammars.

Agenda

- Parsing algorithms ("parsers")
- A parser generator is a program that takes a grammar and returns a parser. BNFC returns parser generators. It is therefore a parser generator generator.
- The first generator tools were Lex and YACC (Bell labs, 1970s).
  - C/Gnu version: Flex/Bison
  - Haskell: Alex/Happy
  - Java: JLex/Cup
- Parsers and lexer collaborate closely: the lexer breaks the user input into a sequence of tokens, which serve as input to the parser. We will first understand what a parser does, then study lexical analysis.

Outline

LR(k) parsing

LR(k) parsing denotes the parsing strategy that

- reads input left to right
- applies the rightmost derivation first
- has a lookahead of k tokens

The first widely used parser generator, YACC, generates LR parsers.

Example (rightmost derivation)

Provided the following statement grammar (assume Integer is the name of a token for integers)

SIf	.	Stm	::=	"if" "(" Exp ")" Stm ;
SWhile	.	Stm	::=	"while" "(" Exp ")" Stm ;
SExp	.	Stm	::=	Exp ";" ;
EInt	.	Exp	::=	Integer ;

Consider the input string

```
while (1) if (0) 6;
```

- Rewrite the rightmost non-terminal first.

Example (cont'd)

Ex.:

```
while (1) if (0) 6;
```

For this input string, rightmost derivation yields the following sequence of derivations:

$\rightarrow$ "while" "(" Exp ")" Stm
$\rightarrow$ "while" "(" Exp ")" "if" "(" Exp ")" Stm
$\rightarrow$ "while" "(" Exp ")" "if" "(" Exp ")" Exp ";"
$\rightarrow$ "while" "(" Exp ")" "if" "(" Exp ")" 6 ";"
$\rightarrow$ "while" "(" Exp ")" "if" "(" 0 ")" 6";"
$\rightarrow$ "while" "(" 1 ")" "if" "(" 0 ")" 6";"

Shift-reduce parsing

LR(k) postpones the decision about which rule to apply until it has seen the entire right-hand side of a production. Internally, it uses a stack.

Based on the stack and the next k tokens it performs one of the following actions

- Shift: push the next k tokens onto the stack
- Reduce: assume the stack has the form $\ldots X\ Y\ Z$ and a production exists $N ::= XYZ$. Then pop elements X, Y, Z and push N onto the stack.

When no input is left and the stack contains the start symbol only, parsing has succeeded. Alternatively, one can introduce an action "accept" and terminate parsing when "accept" is met.

Example (LR(1))

Provided the following expression grammar

1.	Exp ::=	Exp "+" Exp1
2.	Exp ::=	Exp1
3.	Exp1 ::=	Exp1 "*" Integer ;
4.	Exp1 ::=	Integer ;

Input string: 1 + 2 * 3

Stack	Input	Action
	1 + 2 * 3	shift

- Initially, the stack is empty and the first action is a shift action.

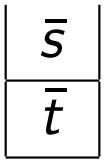
Example (cont'd)

Input string: 1 + 2 * 3

Stack	Input	Action
	1 + 2 * 3	shift
1	+ 2 * 3	reduce 4
Exp1	+ 2 * 3	reduce 2
Exp	+ 2 * 3	shift
Exp +	2 * 3	shift
Exp + 2	* 3	reduce 4
Exp + Exp1	* 3	shift
Exp + Exp1 *	3	shift
Exp + Exp1 * 3		reduce 3
Exp + Exp1		reduce 1
Exp		accept

How to automatically determine when to shift and when to reduce?

Action selection

Assume a stack of the form  where $\bar{s}$ is called the *prefix*. Then the reduction

$$N \to \bar{s}$$

is applicable. But, how to split the stack in prefix and rest?

Idea

- Represent the stack as a deterministic finite automaton (DFA), where prefixes denote states, and the terminals and non-terminals are labels of the transitions.
- Change: do not just stack input symbols, but also store the state. The stack has the form

  $$s_0 X_1 s_1 X_2 \ldots$$

  where s_i state number, X_i one or more grammar symbols.

LR(1) parser tables

The DFA is represented by a table:

- Columns are the terminals and non-terminals of the grammar.
- Rows are *parser states*.
- Cells are actions
  - si: shift onto the stack (input and state i)
  - rn: reduce by rule n
  - a: accept
  - gk: go to state k
  - –: error

Example (parser table)

State	+	*	$	L_int	Integer	Exp	Exp1
0				s3	g4	g6	g7
1				s3	g4		g5
2				s3			
3	r0	r0	r0				
4	r4	r4	r4				
5		s8	a				
6	s9		a				
7	r2	s8	r2				
8				s3	g11		
9				s3	g4		g10
10	r1	s8	r1				
11	r3	r3	r3				

- r0 is a new rule for literals: Integer $\rightarrow$ L_int, $ marks the end of the input. Starting in state 0, the next action is indexed by the next token and operates on the current stack.

Example (parsing)

Recall: the stack contains states and grammar symbols.

Stack	Input	Action
0	1 + 2 * 3$	s3
0 L_int 3	+ 2 * 3$	r0
0 Integer	+ 2 * 3$	g4
0 Integer 4	+ 2 * 3$	r4
0 E1	+ 2 * 3$	g7
0 E1 7	+ 2 * 3$	r2
0 E	+ 2 * 3$	g6
0 E 6	+ 2 * 3$	s9
0 E 6 + 9	2 * 3$	s3
0 E 6 + 9 L_int 3	* 3$	r0
0 E 6 + 9 Integer	* 3$	g4
0 E 6 + 9 Integer 4	* 3$	r4

Example (parsing), cont'd

Stack	Input	Action
0 E 6 + 9 Integer	* 3$	g4
0 E 6 + 9 Integer 4	* 3$	r4
0 E 6 + 9 E1	* 3$	g10
0 E 6 + 9 E1 10	* 3$	s8
0 E 6 + 9 E1 10 * 8	3$	s3
0 E 6 + 9 E1 10 * 8 L_int 3	$	r0
0 E 6 + 9 E1 10 * 8 Integer	$	g11
0 E 6 + 9 E1 10 * 8 Integer 11	$	r3
0 E 6 + 9 E1	$	g10
0 E 6 + 9 E1 10	$	r1
0 E	$	g6
0 E 6	$	
a		

STS Software Technology Systems

LALR(1)

- An LR(1) *parser state* is a set of LR(1) *item sets*.
  - An LR(1) *item set* consists of a grammar production and a "." that marks the current position of the parser. Ex:

```
Stm ::= ''if'' ''('' . Exp '')'' Stm
```

    denotes the state where the parser has read "if" and "(".
  - Parser tables for LR(1) are large: #prefixes x # symbols
- A *lookahead LR(1)* (short: LALR(1)) parser uses a table that is constructed in the same way as the parser table of LR(1) but merges any two states whose item sets to the left of "." are identical.
  - The table of the previous example is a LALR table.
  - Ex.: State 7 combines the two item sets

    Exp → Exp1 .

    Exp1 → Exp1 . "*" Integer
  - Most programming languages use LALR(1) grammars.

Expressivity

$$\text{LR}(0) < \text{LALR}(1) < \text{LR}(1) < \text{LR}(2) < \ldots$$

- A grammar is in a certain class if its parsing table contains no more than one action per cell ("no conflict").
- LR(1) is strictly more expressive than LALR(1): an LALR table may contain conflicts that do not exist in the LR table.

Example (conflicts)

Consider the following grammar:

SIf	.	Stm	::=	"if" "(" Exp ")" Stm ;
SIfElse	.	Stm	::=	"if" "(" Exp ")" Stm "else" Stm;

The *dangling else* problem is a classical example of a conflict. It arises when if-statements are nested. Consider the following input and the parsing position ".":

```
if (x>0) if (y < 8) return y ; . else return x ;
```

Both actions are possible

- shift: else return x ;
- reduce: if (y < 8) return y ;

Conflicts

Generally, two kinds of conflicts can arise in LR and LALR parsing:

- shift-reduce conflicts
- reduce-reduce conflicts

Any conflict implies that some programs in the grammar cannot be parsed. In this case, only two options exist:

- Resolve the conflict
- Switch to a more powerful parser

Conflict resolution: grammar rewrite

The following grammar contains a reduce-reduce conflict: an `Ident` can be reduced in two ways:

Var.	Exp	::=	Ident ;
Cons.	Exp	::=	Ident ;

In this case, one can rewrite the grammar:

- Remove one of the two productions. Have the type checker distinguish between constants and variables.

Rewrites are not always possible, yet.

In-class exercise

Consider the following fragment of the C++ grammar:

SExp.	Stm	::=	Exp ;
SDecl.	Stm	::=	Decl ;
DTyp.	Decl	::=	Typ ;
EId.	Exp	::=	Ident ;
TId.	Typ	::=	Ident ;

In-class exercise

Consider the following fragment of the C++ grammar:

SExp.	Stm	::=	Exp ;
SDecl.	Stm	::=	Decl ;
DTyp.	Decl	::=	Typ ;
EId.	Exp	::=	Ident ;
TId.	Typ	::=	Ident ;

Conflict resolution: priorities

- Shift-reduce conflicts are not so easy to eliminate.
  - Most parsers resolve the conflict by giving the shift action priority over the reduce action.
  - The dangling else is therefore resolved by shifting the inner `if`.
- Reduce-reduce conflicts can often be avoided. If not, most parser generators take the rule listed first.
- In both cases, the grammar is no longer faithfully implemented.

Outline

LL(k) parsing

The alternative to LR parsing is LL(k) parsing, that is, *left-to-right, leftmost derivation* with lookahead k.

- Parsers that are written by hand, are often LL parsers.
- Manual LL parsing follows the technique of *recursive descent parsing*.

Note

$$\text{LL}(k) < \text{LR}(k)$$

Example (leftmost derivation)

Recall the following statement grammar

SIf	.	Stm	::=	"if" "(" Exp ")" Stm ;
SWhile	.	Stm	::=	"while" "(" Exp ")" Stm ;
SExp	.	Stm	::=	Exp ";" ;
EInt	.	Exp	::=	Integer ;

Consider the input string

```
while (1) if (0) 6;
```

- Leftmost derivation rewrites the leftmost non-terminal first.

Example (cont'd)

Ex.:

```
while (1) if (0) 6;
```

For this input string, leftmost derivation yields the following derivation sequence:

Statement
- → "while" "(" Exp ")" Stm
- → "while" "(" 1 ")" Stm
- → "while" "(" 1 ")" "if" "(" Exp ")" Stm
- → "while" "(" 1 ")" "if" "(" 0 ")" Stm
- → "while" "(" 1 ")" "if" "(" 0 ")" Exp ";"
- → "while" "(" 1 ")" "if" "(" 0 ")" Exp ";"
- → "while" "(" 1 ")" "if" "(" 0 ")" 6 ";"

Recursive-descent parsing

In LL(k) parsing, the AST is constructed top-down.

- In a *recursive-descent parser* there exists one function for each category, which inspects the next k tokens and recursively continues. Ex. (k=1):

```
Stm pStm() :
   if (next = "if")
         ignore ("if")
         ignore ("(")
         Exp e := pExp()
         ignore (")")
         Stm s := pStm()
         return SIf (e, s)
   if (next = "while") ...
   if (next is integer ) ...
Exp pExp() :
   if (next is integer k) return SExp k
```

- Alternatively, one can construct table-driven LL parsers.

Example (conflict)

The *dangling else*, caused by a nested if statement, causes for LL languages a conflict as well

SIf	.	Stm	::=	"if" "(" Exp ")" Stm ;
SIfElse	.	Stm	::=	"if" "(" Exp ")" Stm "else" Stm ;

Now consider an if-statement, not necessarily nested:

```
if (x>0) if (y < 8) return y ; else return x ;
```

- When the parser probes the first if, two rules are possible: `SIf` and `SIfElse`. Conflict!

Left factoring

Left refactoring can resolve certain conflicts. In left refactoring the shared part of the conflicting rule is factored out and the rules "split off" only afterwards.

SIE	.	Stm	::=	"if" "(" Exp ")" Stm Rest ;
RElse	.	Rest	::=	"else" Stm ;
REmp	.	Rest	::= ;	

```
// if (x>0) if (y < 8) return y ; else return x ;

Stm pStm() :
    if (next = "if")
            ...
            return SIE (e, s)
Stm pRest() :
    if (next = ``else'')
            ignore (``else'')
            s := pStm()
            return s
    if (next = "'')
          return pStm(epsilon())
```

Left recursion

A rule is *left recursive* if the non-terminal on the left-hand side of a grammar is the same as the *first* item on the right-hand side. Ex.:

Exp	::=	Exp "+" Integer ;
Exp	::=	Integer ;

- A LL(1) parser loops on the grammar above.
- Implicit left recursions complicates matters.

A	::=	B
B	::=	A

Right recursion

A *left recursive* rule can be written using right recursion.

Exp	::=	Integer Rest ;
Rest	::=	"+" Integer Rest ;
Rest	::=	;

- The new category Rest uses right recursion only.
- The resulting language is the same.
- Note that the AST now contains categories that are not part of the original grammar. If one wants to keep the original abstract syntax, the tree needs to be rewritten to the original form.

Error recovery

Two major error recovery strategies: error correction and suffix-grammar based recovery.

- Error correction
  - Modify the input or the internal parser state
  - Example: error production for common user errors
  - Potential problem: if the correction does not fix the actual error, additional error messages are triggered that confuse the user
- Suffix grammars
  - Discard input until some "recovery point", e.g., a delimiter; continue parsing after that.
  - Potential problem: might miss other errors; produces partial syntax tree only

Outline

Syntactic analysis with bison

http://www.gnu.org/software/bison/manual/bison.html

The `bison` program takes a CFG in a file with ending `.y` and returns a `C` source file. The grammar file consists of four sections:

```
%{
C Declarations
%}
Bison Declarations
%%
Grammar Rules
%%
Additional C Code
```

- Bison keywords `%token, %type`
- Bison types: `yylval`
- Bison functions: `yyparse()`, `yyerror()`

Bison grammars

http://www.gnu.org/software/bison/manual/bison.html

The sections, again:

- Prologue: declaration of variables and function; macros.
- Bison declarations: names for tokens, definition of precedence
- Grammar rules: rules of the form `lhs : rhs ;`, including alternatives (`|`), interspersed with C code surrounded by parentheses. The C code contains *actions*.
- Epilogue: C code, also copied verbatim. Often contains the definition of `yylex` and `yyerror`.

Example (bison prologue)

The Prologue sections are enclosed in %{ and %}

```
%{
  #include <stdio.h>
  extern int yylex (void);
  void yyerror (char const *);
%}
```

- The code is copied verbatim into the output.
- Typically contains the declaration of `yylex` and `yyerror`.

STS Software Technology Systems TUHH

Example (bison declarations)

```
%union
{
  int int_;
  char char_;
  double double_;
  char* string_;
  Exp exp_;
}

%token _ERROR_
%token _SYMB_0     /*    +    */
%token _SYMB_1     /*    -    */

%type <exp_> Exp
%type <exp_> Exp1
```

- The %union type defines the set of possible data types for semantic values. The %type declaration specifies the value type for each non-terminal.
- The %token declaration introduces symbols that can be used within yylex.

Example (bison grammar)

http://www.gnu.org/software/bison/manual/bison.html#Language-and-Grammar

```
Exp : Exp _SYMB_0 Exp1 { $$ = make_EAdd($1, $3);
                         YY_RESULT_Exp_= $$; }
  | Exp _SYMB_1 Exp1   {  $$ = make_ESub($1, $3);
                         YY_RESULT_Exp_= $$; }
  | Exp1 { $$ = $1; YY_RESULT_Exp_= $$; }
;
```

- The grammar section contains rules and actions.
- Actions are C code enclosed in parentheses.
- $$ is a *pseudo variable* for the semantic value of the result. $1, $2 refer to the components of the rule.

Invoking bison

bison expects as input a grammar file with extension .y.

- If invoked standalone, it returns two source files with extension .tab.c and .tab.h.
- bison generates a parser, yyparse(), which calls a routine yylex().
- Yet, bison does not deal with tokenization itself. The client must provide an implementation of the lexer.
- If flex is used, the C output of flex and bison must be compiled together.

```
// outside bnfc
> bison -d foo.y
> flex foo.l
> gcc foo.tab.c lex.yy.c  -o foo
```

Collaboration of bison and flex

bison and flex mutually depend on each other. In the definition section of foo.l include the header file of the parser.

```
%option noyywrap
%{
#include <stdio.h>
#include "tutorialCalc.tab.h"
void yyerror(char const* c);
int yyparse();
%}

%%

%%
void yyerror(char const* str) { printf("ERROR"); }
int main(void) {
    yyparse();
}
```

Collaboration of bison and flex (2)

In the prologue section of foo.y, include extern declarations that the lexer provides.

```
%{
  #include <stdio.h>
  #include <math.h>
  extern int yylex ();
  extern void yyerror (char const *);
%}

%token NUM STR

%left '+' '-'
%left '*'

%%
..
```

Example (grammar)

The example grammar (without precedences) contains conflicts.

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

> bnfc -m conflictCalc.cf
> bnfc -m -c conflictCalc.cf
> happy ParconflictCalc.y
shift/reduce conflicts:  16
> bison -d conflictCalc.y
conflictCalc.y: conflicts: 16 shift/reduce
```

Debugging a grammar

Parsing conflicts are a property of the grammar, not the parser generator. For debugging purposes, one can therefore look at the "information" file of any parsing tool.

- The `happy` parser generates a text file with ending `.info` that contains debugging information.

```
> happy -i conflictCal.y
```

- In `bison`, the `report` flag generates a file with ending `.output`, which one can inspect.

```
> bison --report=state -d conflictCalc.y
```

Limits of context-free grammars

Not every language is context-free. Consider the copy language, which contains *aca*, *abcab*, but not *acba*:

$$\{wcw \mid w \in (a|b)^*\}$$

The following grammar does *not* guarantee the copy effect

S	::=	W c W
W	::=	"a" W | "b" W
W	::=	

- The language contains wrongfully, e.g., the sentence *acba*.
- One can prove that no CFG exists for the copy language.

Example (copy language)

An instance of the copy language in language processing is the requirement that every variable is declared before used.

$$P \quad ::= \quad \text{Var} \ldots \text{Var} \ldots$$

- The previous language can be parsed.
- But a parser cannot check the "declare-before-use" requirement.
- A later phase has to carry out the check: the type checker.

Another example (non-context-free language)

The language L,

$$L = \{a^n b^m c^n d^m \mid n, m \geq 1\}$$

is not context-free.

- L captures the requirement that the number of formal and actual parameters must agree. (a^n, b^m denote the formal parameter lists, c^n, d^m the actual parameter lists.)
- In contrast, the language $L' = \{a^n b^m c^m d^n \mid n, m \geq 1\}$ is context-free.

Summary

- LR parsing, LL parsing
- Parsing conflicts: shift-reduce, reduce-reduce; conflict resolution
- Bison; collaboration lexer/parser

References

- IPL, Ch. 3.5-3.9