

COMP3012/G53CMP: Lecture 4

Syntactic Analysis: Parser Generators

Henrik Nilsson

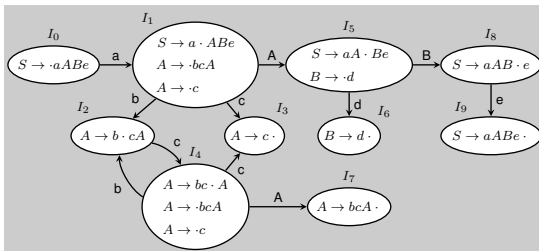
University of Nottingham, UK

COMP3012/G53CMP: Lecture 4 - p.1/32

Parser Generators (2)

An LR(0) DFA recognizing viable prefixes for

$S \rightarrow aABe$ $A \rightarrow bcA \mid c$ $B \rightarrow d$



COMP3012/G53CMP: Lecture 4 - p.1/32

Parser Generators (5)

- At a reduction, the terminals and non-terminals of the RHS of the production (the **handle**) are on the parse stack, associated with **semantic information** or **semantic values**; e.g., the corresponding AST fragments, expression values.
- Think of the RHS symbols as **variables** bound to the the semantic value resulting from a successful derivation for that symbol.
- Construction of AST, evaluation of expressions, etc. proceeds in **bottom-up** order.

COMP3012/G53CMP: Lecture 4 - p.1/32

This Lecture

- Parser generators (“compiler compilers”)
- The parser generator Happy
- A TXL parser written using Happy
- A TXL interpreter written using Happy

COMP3012/G53CMP: Lecture 4 - p.2/32

Parser Generators (3)

- Parser construction** is in many ways a very **mechanical** process. Why not write a program to do the hard work for us?
- A **Parser Generator** (or “compiler compiler”) takes a grammar as input and outputs a parser (a program) for that grammar.
- The input grammar is augmented with **“semantic actions”**: code fragments that get invoked when a derivation step is performed.
- The semantic actions typically construct an AST or interpret the program being parsed.

COMP3012/G53CMP: Lecture 4 - p.3/32

Parser Generators (6)

Some examples of parser generators:

- Yacc (“Yet Another Compiler Compiler”): A classic UNIX LALR parser generator for C. <http://dinosaur.compilertools.net/>
- Bison: GNU project parser generator, a free Yacc replacement, for C and C++.
- Happy: a parser generator for Haskell, similar to Yacc and Bison. <http://www.haskell.org/happy/>
- Cup: LALR parser generator for Java.

COMP3012/G53CMP: Lecture 4 - p.3/32

Parser Generators (1)

- Constructing parsers by hand can be very tedious and time consuming.
- This is true in particular for LR(*k*) and LALR parsers: constructing the corresponding DFAs is extremely laborious.
- E.g., this simple grammar (from the prev. lect.)

$$S \rightarrow aABe$$

$$A \rightarrow bcA \mid c$$

$$B \rightarrow d$$

gives rise to a 10 state LR(0) DFA!

COMP3012/G53CMP: Lecture 4 - p.3/32

Parser Generators (4)

Consider an LR shift-reduce parser:

- Some of the actions when parsing *abcde*:

State	Stack (γ)	Input (w)	Move
...
I_5	aA	de	Shift
I_6	aAd	e	Reduce by $B \rightarrow d$
I_8	aAB	e	Shift
I_9	$aABe$	ϵ	Reduce by $S \rightarrow aABe$
	S	ϵ	Done

- A **reduction** corresponds to a derivation step in the grammar (an LR parser performs a rightmost derivation in reverse).

COMP3012/G53CMP: Lecture 4 - p.3/32

Parser Generators (7)

- ANTLR: LL(*k*) (recursive descent) parser generator and other translator-oriented tools for Java, C#, C++. <http://www.antlr.org/>
- Many more compiler tools for Java here: <http://catalog.compilertools.net/java.html>
- And a general catalogue of compiler tools: <http://catalog.compilertools.net/>

COMP3012/G53CMP: Lecture 4 - p.3/32

Happy Parser for TXL (1)

We are going to develop a TXL parser using Happy. The TXL CFG:

```
TXLProgram → Exp
Exp        → AddExp
AddExp     → MulExp
           | AddExp + MulExp
           | AddExp - MulExp
```

Note: **Left-recursive!** (To impart associativity.)
LR parsers have no problems with left- or right-
recursion (except right recursion uses more stack).

COMP3012/GS3CMP: Lecture 4 – p.1632

Happy Parser for TXL (4)

Haskell datatypes for AST:

```
data BinOp = Plus | Minus | Times | Divide

data Exp = LitInt Int
         | Var Id
         | BinOpApp BinOp Exp Exp
         | Let Id Exp Exp
```

COMP3012/GS3CMP: Lecture 4 – p.1632

Happy Parser for TXL (7)

- The code fragment between curly braces is a Haskell **pattern** that is matched against the actual tokens returned by the parsing function.
- If this pattern contains the special variable `$$`, then the corresponding part of the matched token becomes the semantic value. Examples: `T_Int $$`, `T_Id $$`
- Otherwise the entire token becomes the semantic value. Examples: `T_Plus`, `T_In`
- The semantic values of different terminal symbols may thus have different types.

COMP3012/GS3CMP: Lecture 4 – p.1632

Happy Parser for TXL (2)

The TXL CFG continued:

```
MulExp → PrimExp
       | MulExp * PrimExp
       | MulExp / PrimExp

PrimExp → IntegerLiteral
       | Identifier
       | ( Exp )
       | let Identifier = Exp in Exp
```

COMP3012/GS3CMP: Lecture 4 – p.1632

Happy Parser for TXL (5)

A simple Happy parser specification:

```
{ Module Header }
%name ParserFunctionName
%tokentype { TokenTypeName }

%token
Specification of Terminal Symbols
%%
Grammar productions with semantic actions

{ Further Haskell Code }
```

COMP3012/GS3CMP: Lecture 4 – p.1632

Happy Parser for TXL (5)

The grammar productions are written in BNF, with an additional semantic action defining the semantic value for each production:

```
add_exp
: mul_exp { $1 }
| add_exp '+' mul_exp { BinOpApp Plus $1 $3 }
| add_exp '-' mul_exp { BinOpApp Minus $1 $3 }

mul_exp
: prim_exp { $1 }
| mul_exp '*' prim_exp { BinOpApp Times $1 $3 }
| mul_exp '/' prim_exp { BinOpApp Divide $1 $3 }
```

COMP3012/GS3CMP: Lecture 4 – p.1732

Happy Parser for TXL (3)

Haskell datatype for tokens:

```
data Token = T_Int Int
           | T_Id Id
           | T_Plus
           | T_Minus
           | T_Times
           | T_Divide
           | T_LeftPar
           | T_RightPar
           | T_Equal
           | T_Let
           | T_In
```

COMP3012/GS3CMP: Lecture 4 – p.1632

Happy Parser for TXL (6)

The terminal symbol specification specifies terminals to be used in productions and relates them to Haskell constructors for the tokens:

```
%token
int      { T_Int $$ }
ident    { T_Id $$ }
'+'      { T_Plus }
'-'      { T_Minus }
...
'='      { T_Equal }
let      { T_Let }
in       { T_In }
```

COMP3012/GS3CMP: Lecture 4 – p.1632

Happy Parser for TXL (6)

It is also possible to add type annotations:

```
add_exp :: { Exp }
add_exp
: mul_exp { $1 }
| add_exp '+' mul_exp { BinOpApp Plus $1 $3 }
| add_exp '-' mul_exp { BinOpApp Minus $1 $3 }
```

Most useful when semantic values are of different types.

See `HappyTXL.y` for the complete example.

COMP3012/GS3CMP: Lecture 4 – p.1632

Shift/Red. and Red./Red. Conflicts (1)

Context-free grammars are often initially ambiguous. Consider the grammar fragment:

```
Cmd → ...
     | if Exp then Cmd
     | if Exp then Cmd else Cmd
```

According to this grammar, a program fragment

```
if e1 then if e2 then c1 else c2
```

can be parsed in two ways, with very different meanings (the “dangling else” problem):

```
if e1 then (if e2 then c1) else c2
if e1 then (if e2 then c1 else c2)
```

COMP9012/GS3CMP: Lecture 4 – p.19/32

Shift/Red. and Red./Red. Conflicts (4)

- Shift/reduce conflicts often resolved by opting for shifting:
 - Typically the default option (e.g. Yacc, Bison, Happy)
 - Usually gives the desired result; e.g., resolves the dangling else problem in a natural way.
- Reduce/reduce conflicts are worse as no reason to pick one production over another: grammar has to be manually disambiguated.

COMP9012/GS3CMP: Lecture 4 – p.20/32

A TXL Interpreter (2)

One way:

- Each semantic action returns a **function** of type

```
Env -> Int
```

where (for example)

```
Type Env = Id -> Int
```

- The semantic action for evaluating a composite expression passes on the environment. E.g. semantic action for +:

```
| exp '+' exp
{ \env -> $1 env + $3 env }
```

COMP9012/GS3CMP: Lecture 4 – p.25/32

Shift/Red. and Red./Red. Conflicts (2)

In LR-parsing, ambiguous grammars lead to **shift/reduce** and **reduce/reduce** conflicts:

- shift/reduce: some states have mixed complete and incomplete items:

```
A → a ·
A → a · b
```

Should parser shift or reduce?

COMP9012/GS3CMP: Lecture 4 – p.20/32

Precedence and Associativity

Happy (like e.g. Yacc and Bison) allows operator precedence and associativity to be explicitly specified to disambiguate a grammar:

```
%left '+' '-'
%left '*' '/'
exp : exp '+' exp { BinOpApp Plus $1 $3 }
    | exp '-' exp { BinOpApp Minus $1 $3 }
    | exp '*' exp { BinOpApp Times $1 $3 }
    | exp '/' exp { BinOpApp Divide $1 $3 }
    ...
```

See HappyTXL2.y for further details.

COMP9012/GS3CMP: Lecture 4 – p.23/32

A TXL Interpreter (3)

- The semantic action for a variable looks up the variable value in the environment:

```
| ident { \env -> env $1 }
```
- The semantic action for `let` extends the argument environment and evaluates the body in the extended environment:

```
| let ident '=' exp in exp
{\env -> let v = $4 env
  in $6 (\i -> if i == $2
    then v
    else env i)}
```

COMP9012/GS3CMP: Lecture 4 – p.26/32

Shift/Red. and Red./Red. Conflicts (3)

- reduce/reduce: some states have more than one complete item:

```
A → a ·
B → a ·
```

Reduce, but by which production?

COMP9012/GS3CMP: Lecture 4 – p.21/32

A TXL Interpreter (1)

The semantic actions do not have to construct an AST. An alternative is to **interpret** the code being parsed. Basic idea:

```
exp :: { Int }
exp
  : exp '+' exp { $1 + $3 }
  | exp '-' exp { $1 - $3 }
  ...
```

But TXL has a `let`-construct ...

What about TXL VARIABLES??? E.g.:

```
let x = 3 in x + x, semantic value of x?
```

COMP9012/GS3CMP: Lecture 4 – p.27/32

A TXL Interpreter (4)

- The semantic action for literal just returns the value of the literal, ignoring the environment:

```
| int { \_ -> $1 }
```
- A program gets evaluated by applying the overall result function to the empty environment:

```
(\_ -> error "undefined variable")
```

See HappyTXLInterpreter.y for further details.

COMP9012/GS3CMP: Lecture 4 – p.27/32

A TXL Interpreter (5)

Another way: Attribute Grammars

- Each non-terminal associated with a set of **semantic attributes** (values).
- Set of **equations** for each production defines the attributes.
- Attributes can be:
 - **Inherited**: defined in terms of attributes of parents or siblings in derivation tree.
 - **Synthesized**: defined in terms of attributes of children in derivation tree.

COMP3012/GS3CMP: Lecture 4 – p.29/32

A TXL Interpreter (6)

- We need two attributes:

```
%attribute value { Int }
%attribute env { Env }
```

The first one is the **default attribute**: assumed unless other attribute name explicitly stated, and gives overall result.

- Semantic equations for composite expressions:

```
| exp '+' exp
{ $$      = $1 + $3; - synthesized
  $1.env  = $$ .env; - inherited
  $3.env  = $$ .env } - inherited
```

COMP3012/GS3CMP: Lecture 4 – p.29/32

A TXL Interpreter (7)

- Semantic equation for literals:

```
| int { $$ = $1 }
```

- Semantic equation for identifiers:

```
| ident { $$ = $$ .env $1 }
```

- Semantic equations for let:

```
| let ident '=' exp in exp
{ $$      = $6;
  $4.env  = $$ .env;
  $6.env  = extEnv $2 $4 $$ .env }
```

where `extEnv` is an auxiliary function for extending an environment with a new binding.

COMP3012/GS3CMP: Lecture 4 – p.30/32

A TXL Interpreter (8)

- Semantic equations for program:

```
| txl_program : exp
{ $$ = $1; $1.env = emptyEnv }
```

where

```
emptyEnv :: Env
emptyEnv =
  \_ -> error "undefined variable"
```

See `HappyTXLInterpreterAG.y` for further details.

COMP3012/GS3CMP: Lecture 4 – p.31/32

Standard vs. Attribute Grammar

Note that a standard Happy grammar can be seen as a very simple attribute grammar:

- A single attribute: the semantic value
- A single (implicit) equation: the semantic action
- Only synthesised attributes

COMP3012/GS3CMP: Lecture 4 – p.32/32