Weeding

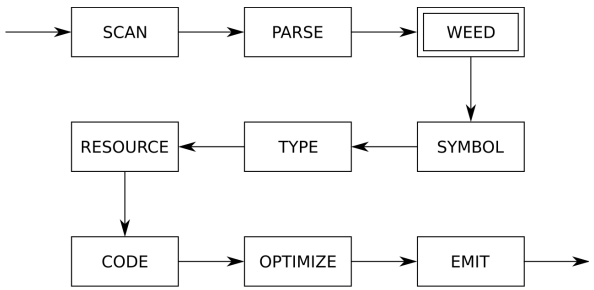
COMP 520: Compiler Design (4 credits)

Alexander Krolik

alexander.krolik@mail.mcgill.ca

MWF 10:30-11:30, TR 1100

http://www.cs.mcgill.ca/~cs520/2020/



Weeding

LALR(1) grammars (those used by `bison`) accept a subset of context-free languages.

Sometimes, programming languages require more power, such as when

- Our language is not context-free;
- Our language is not LALR(1) (for now let's ignore that `bison` now also supports GLR); or
- An LALR(1) grammar is too big and complicated.

Solution

In these cases we can use a parsing trick

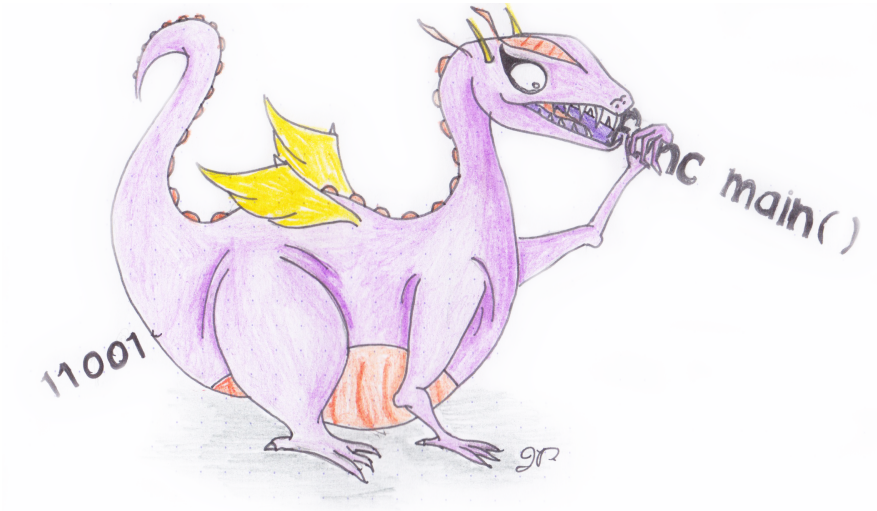
- Use a simple and more liberal grammar which accepts a slightly larger language; and
- A separate phase can then *weed* out the bad parse trees.

Weeding

Grammar expansion

LALR(1) restrictions

Not (easily) context-free



Division by Constant 0

If we wish to disallow division by a constant zero, we can always rewrite our grammar

```
exp : tIDENTIFIER
    | tINTCONST
    | exp '*' exp
    | exp '/' pos
    | exp '+' exp
    | exp '-' exp
    | '(' exp ')'
;

pos : tIDENTIFIER
    | tINTCONSTPOSITIVE
    | exp '*' exp
    | exp '/' pos
    | exp '+' exp
    | exp '-' exp
    | '(' pos ')'
;
```

While this is correct, we have doubled the size of our grammar – this is not a very modular technique.

Division by Constant 0

Instead, we use the default grammar and weed out division by constant 0 using a simple modular traversal. Return `1` if an illegal division occurs, `0` otherwise.

```
int zerodivEXP(EXP *e) {
   switch (e->kind) {
      case idK:
      case intconstK:
         return 0;
      case timesK:
         return zerodivEXP(e->val.timesE.left) ||
                zerodivEXP(e->val.timesE.right);
      case divK:
         if (e->val.divE.right->kind == intconstK &&
             e->val.divE.right->val.intconstE == 0) return 1;

         return zerodivEXP(e->val.divE.left) ||
                zerodivEXP(e->val.divE.right);
      case plusK:
         return zerodivEXP(e->val.plusE.left) ||
                zerodivEXP(e->val.plusE.right);
      case minusK:
         return zerodivEXP(e->val.minusE.left) ||
                zerodivEXP(e->val.minusE.right);
   }
}
```

Weeding Local Declarations

In JOOS, all local variable declarations must appear at the beginning of a statement sequence

```
int i;
int j;
i = 17;
int b;  /* illegal */
b = i;
```

This can easily be solved in the grammar: each statement sequence is a list of declarations followed by a list of statements.

It can also be solved in the weeder, after accepting declarations as statements.

Weeding Local Declarations

Return `1` if declarations may follow, `0` otherwise.

```
int weedSTATEMENTlocals(STATEMENT *s, int localsallowed) {
   if (s == NULL) return 0;
   switch (s->kind) {
      case skipK:
         return 0;
      case localK:
         if (!localsallowed) {
            reportError("illegally placed local declaration", s->lineno);
         }
         return 1;
      case expK:
         return 0;
      case returnK:
         return 0;
      case sequenceK:
         int onlylocalsfirst = weedSTATEMENTlocals(
            s->val.sequenceS.first,localsallowed);
         int onlylocalssecond = weedSTATEMENTlocals(
            s->val.sequenceS.second,onlylocalsfirst);
         return onlylocalsfirst && onlylocalssecond;
```

Weeding Local Declarations

```
        case ifK:
           weedSTATEMENTlocals(s->val.ifS.body,0);
           return 0;
        case ifelseK:
           weedSTATEMENTlocals(s->val.ifelseS.thenpart,0);
           weedSTATEMENTlocals(s->val.ifelseS.elsepart,0);
           return 0;
        case whileK:
           weedSTATEMENTlocals(s->val.whileS.body,0);
           return 0;
        case blockK:
           weedSTATEMENTlocals(s->val.blockS.body,1);
           return 0;
        case superconsK:
           return 1;
    }
}
```

Break Statements

Break statements are typically only allowed within loops (or switch statements).

```
while (a) {
      break; /* Allowed */
}

break; /* Error: Break outside loop context */
```

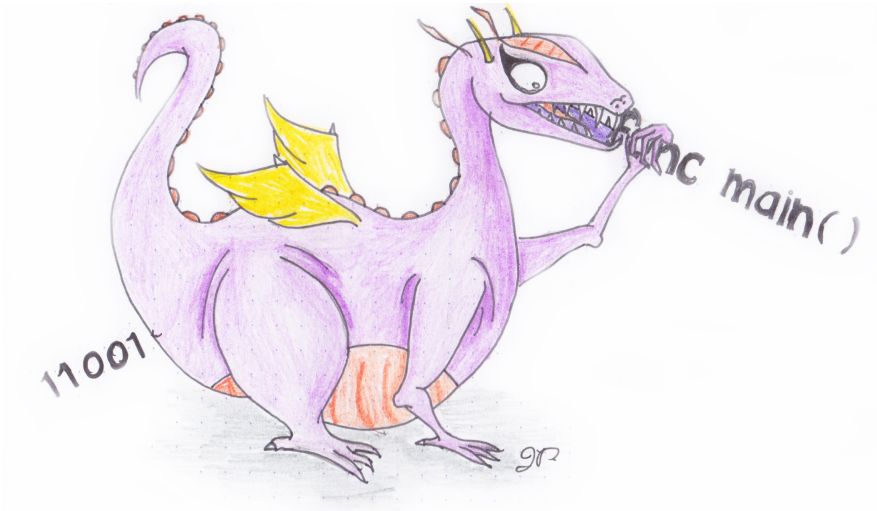
Is this context-free? LALR(1)? Simple and compact LALR(1)?

Weeding

Grammar expansion

LALR(1) restrictions

Not (easily) context-free



JOOS Cast Expressions

The JOOS grammar calls for casts calls for a rule

```
castexpression : '(' identifier ')' unaryexpressionnotminus
```

But this is not LALR(1)!

```
State 194

    88 assignment: tIDENTIFIER . '=' expression
   116 castexpression: '(' tIDENTIFIER . ')' unaryexpressionnotminus
   118 postfixexpression: tIDENTIFIER .  [tINSTANCEOF, tEQ, tLEQ, tGEQ, tNEQ, tAND, tOR,
                                          ')', '<', '>', '+', '-', '*', '/', '%']
   127 receiver: tIDENTIFIER .  ['.']

    ')'  shift, and go to state 232
    '='  shift, and go to state 192

    ')'        [reduce using rule 118 (postfixexpression)]
    '.'        reduce using rule 127 (receiver)
    $default   reduce using rule 118 (postfixexpression)
```

When the next token is `)`, we can either reduce the identifier to an expression to form a parenthetical expression, or shift to form the cast.

JOOS Cast Expressions

Instead, we can use a more general rule which avoids the shift-reduce conflict

```
castexpression : '(' expression ')' unaryexpressionnotminus
```

This rule is LALR(1), so `bison` can generate the corresponding parser. However, it will also accept invalid cast expressions like `(4)x`.

Solution

To fix the grammar, we use a clever rule and action pair

```
castexpression : '(' expression ')' unaryexpressionnotminus {
   if ($2->kind != idK) yyerror("identifier expected");
   $$ = makeEXPcast($2->val.idE.name,$4);
}
;
```

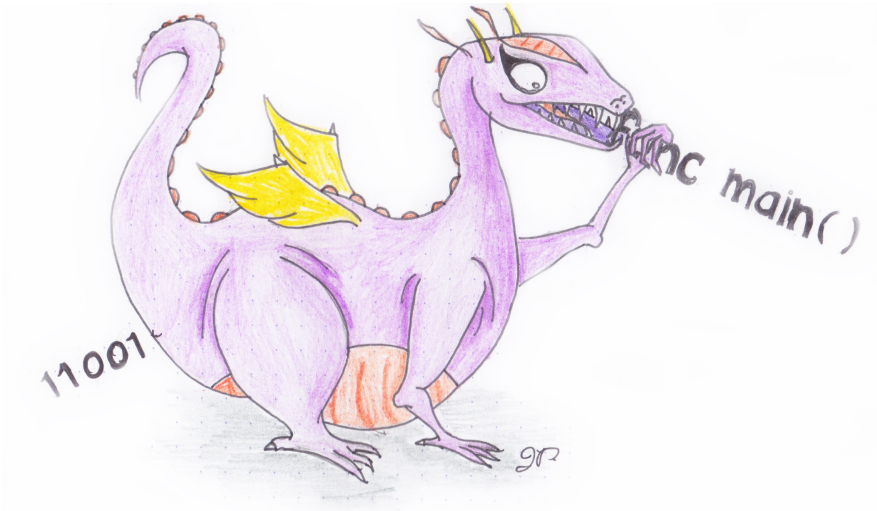
Hacks like this only work sometimes.

Weeding

Grammar expansion

LALR(1) restrictions

Not (easily) context-free



Weeding Return Statements

In JOOS, every branch through the body of a non-`void` method must terminate with a return statement

```
/* illegal */
public boolean foo(Object x, Object y) {
   if (x.equals(y))
      return true;
}
```

This is hard or impossible to express through an LALR(1) grammar.

Weeding Return Statements

Return `1` if a path returns, `0` otherwise

```
int weedSTATEMENTreturns(STATEMENT *s) {
   if (s != NULL) return 0;
   switch (s->kind) {
      case skipK:
         return 0;
      case localK:
         return 0;
      case expK:
         return 0;
      case returnK:
         return 1;
      case sequenceK:
         return weedSTATEMENTreturns(s->val.sequenceS.second);
      case ifK:
         return 0;
      case ifelseK:
         return weedSTATEMENTreturns(s->val.ifelseS.thenpart) &&
                weedSTATEMENTreturns(s->val.ifelseS.elsepart);
      case whileK:
         return 0;
      case blockK:
         return weedSTATEMENTreturns(s->val.blockS.body);
      case superconsK:
         return 0;
   }
}
```