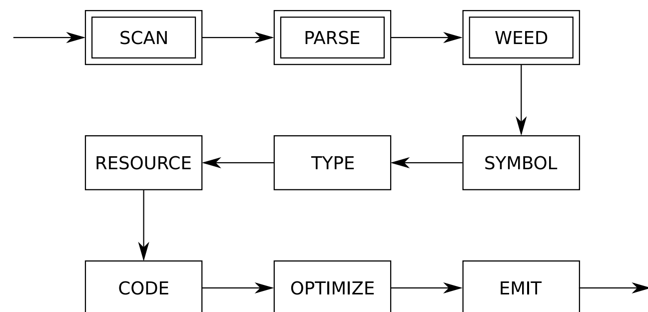
Frontend Wrapup

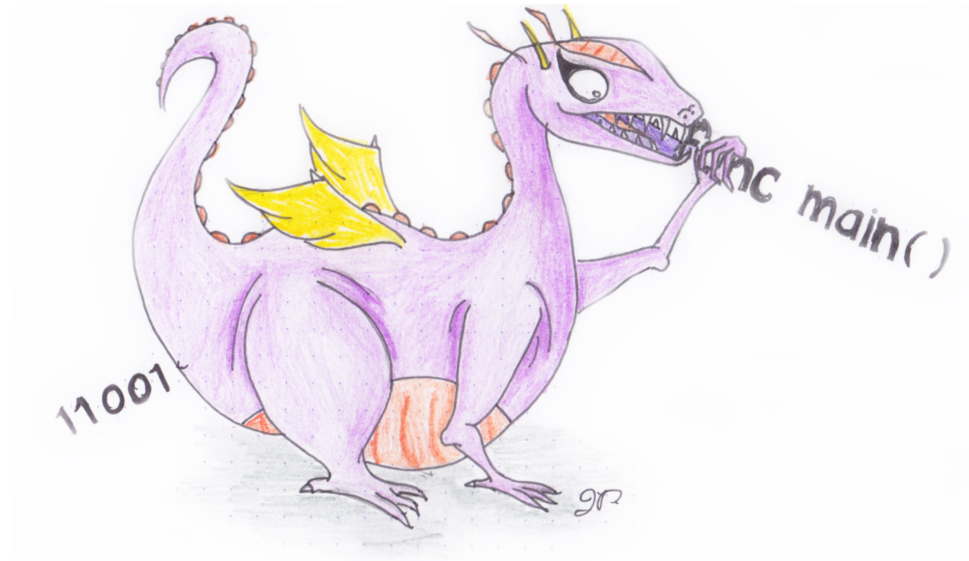
COMP 520: Compiler Design (4 credits)

Alexander Krolik

`alexander.krolik@mail.mcgill.ca`

MWF 13:30-14:30, MD 279





Announcements (Wednesday, January 23rd)

Milestones:

- Group project signup. 1 person per team please fill out `https://goo.gl/forms/ztYMHfcWJXjPA4A43` by the end of the week
- Assignment 1 due **tonight** on myCourses

Assignment 1:

- SableCC users, see email about running it on the Trottier machines
- be sure to write the run scripts!

Frequent questions:

- a string *is* an expression
- no type checking this assignment
- associativity will not be tested in this assignment
- you *should* fix shift/reduce conflicts

Goals in the design of JOOS:

- extract the object-oriented essence of Java;
- make the language small enough for course work, yet large enough to be interesting;
- provide a mechanism to link to existing Java code; and
- ensure that every JOOS program is a valid Java program, such that JOOS is a strict subset of Java.

Programming in JOOS:

- each JOOS program is a collection of classes;
- there are ordinary classes which are used to develop JOOS code; and
- there are external classes which are used to interface to Java libraries.

An ordinary class consists of:

- protected fields;
- constructors; and
- public methods.

```
$ cat Cons.java
```

```
public class Cons {
      protected Object first;
      protected Cons rest;

      public Cons(Object f, Cons r) {
            super(); first = f; rest = r;
      }

      public void setFirst(Object newfirst) {
            first = newfirst;
      }

      public Object getFirst() {
            return first;
      }

      public Cons getRest() {
            return rest;
      }
```

```
    public boolean member(Object item) {
        if (first.equals(item))
            return true;
        else if (rest==null)
            return false;
        else
            return rest.member(item);
    }

    public String toString() {
        if (rest==null)
            return first.toString();
        else
            return first + " " + rest;
    }
}
```

The JOOS grammar calls for:

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

but that is not LALR(1).

However, the more general rule:

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

is LALR(1), so we can use a clever action:

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

Hacks like this only work sometimes.

Why does this work?

Begin by examining the first cast, where a cast requires an identifier:

```
$ bison --yacc --report=all joos.y
joos.y: warning: 1 shift/reduce conflict [-Wconflicts-sr]
```

Using flag `-report=all`, `bison` generates a file `y.output` with the underlying LALR parser states. In this file we can see the shift/reduce conflict details:

```
State 194 conflicts: 1 shift/reduce

[...]

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)
```

By generalizing our grammar, this generates the “equivalent” state:

```
State 139

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

            '='  shift, and go to state 192

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

Note that in this state we do not have the troublesome `castexpression` shift. This shift is moved to another state where it does not conflict:

```
State 194

    116 castexpression: '(' expression . ')' unaryexpressionnotminus
    122 primaryexpression: '(' expression . ')'

            ')'  shift, and go to state 232
```

Building LALR(1) lists:

```
formals : /* empty */
                  {$$ = NULL;}
        | neformals
                  {$$ = $1;}
;

neformals : formal
                  {$$ = $1;}
          | neformals ',' formal
                  {$$ = $3; $$->next = $1;}
;

formal : type tIDENTIFIER
                  {$$ = makeFORMAL($2,$1,NULL);}
;
```

The lists are naturally backwards.

Using backwards lists:

```
typedef struct FORMAL {
    int lineno;
    char *name;
    int offset; /* resource */
    struct TYPE *type;
    struct FORMAL *next;
} FORMAL;

void prettyFORMAL(FORMAL *f) {
    if (f!=NULL) {
        prettyFORMAL(f->next);
        if (f->next!=NULL) printf(", ");
        prettyTYPE(f->type);
        printf(" %s",f->name);
    }
}
```

What effect would a call stack size limit have?

LALR(1) and Bison are not enough when:

- our language is not context-free;
- our language is not LALR(1) (for now let's ignore the fact that Bison now also supports GLR); or
- an LALR(1) grammar is too big and complicated.

In these cases we can try using a more liberal grammar which accepts a slightly larger language.

A separate phase can then weed out the bad parse trees.

A weeding phase:

- enforces a specific syntactic or semantic rule of a program;
- that is hard (or impossible) to implement in the scanner or parser;
- often use contextual information; and
- traverse the AST (or lower level IR) and checks for a specific property.

Examples include:

- disallowing division by a constant zero;
- requiring a return statement by the end of a function;
- `break` or `continue` only allowed in switches and loops;
- ...

Example: disallowing division by constant 0:

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

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

We have doubled the size of our grammar.

This is not a very modular technique.

Instead, weed out division by constant 0:

```
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);
    }
}
```

A simple, modular traversal.

Requirements of JOOS programs:

- all local variable declarations must appear at the beginning of a statement sequence:

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

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

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

Also may not return from within a while-loop etc.

These are hard or impossible to express through an LALR(1) grammar.

Weeding bad local declarations:

```
int weedSTATEMENTlocals(STATEMENT *s,int localsallowed) {
    int onlylocalsfirst, onlylocalssecond;
    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:
                onlylocalsfirst = weedSTATEMENTlocals(
                      s->val.sequenceS.first,localsallowed);
                onlylocalssecond = weedSTATEMENTlocals(
                      s->val.sequenceS.second,onlylocalsfirst);
                return onlylocalsfirst && onlylocalssecond;
```

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

Weeding missing returns:

```
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;
        }
    }
```

Special topic #1: Scanner efficiency

Compiler efficiency is extremely important, but scanners operate on a character by character basis. In reality, scanning is one of the more time consuming elements of a (simple) compiler.

Recall: to produce a string of tokens, we match on *every* regular expression in the scanner.

Something quite simple we can do is:

- reduce the number of regular expressions;
- by observing that keywords are valid identifiers; and
- use a (fast) lookup mechanism to determine if it is a reserved word.

Special topic #2: Scanner error handling

Say in our language we require integers do not have a leading zero. The following assignment is invalid:

```
var a : int;
a = 011;
```

Using a standard `0|([1-9][0-9]*)` regular expression, this produces the token stream:

```
tVAR
tIDENT(a)
tCOLON
tINT
tSEMICOLON
tIDENT(a)
tASSIGN
tINTCONST(0)
tINTCONST(11)
tSEMICOLON
```

The first question to ask: is this a syntactic or a lexical error?

Syntactic error:

It might be tempting to automatically assume this is a lexical error, but what if the user intended to write:

```
var a : int;
a = 0 + 11;
```

This might not be a very useful computation, but it is *valid*. The new token stream:

```
tVAR
tIDENT(a)
tCOLON
tINT
tSEMICOLON
tIDENT(a)
tASSIGN
tINTCONST(0)
tPLUS              // this is new
tINTCONST(11)
tSEMICOLON
```

If we assume this is a syntactic error, the original program was simply missing the addition operator. This is easily handled by a correct grammar.

Lexical error:

On the other hand, if we assume it is unlikely they intended to use an operation with a zero, a lexical error may be more appropriate. But how to do so?

We define 2 regular expressions:

1. A valid regular expression: `0|([1-9][0-9]*)`
2. An invalid regular expression: `([0-9]*)`

For an invalid integer:

1. Valid regular expression matches on the leading zero only - this is of length 1
2. Invalid regular expression matches on the entire input number (length > 1)

Using the longest match principle we choose the invalid regular expression and throw an error.

For a valid integer:

1. Valid regular expression matches on the entire input n
2. Invalid regular expression matches on the entire input n

Using the first match principle we can choose the valid regular expression and produce an `tINTCONST(n)` token.

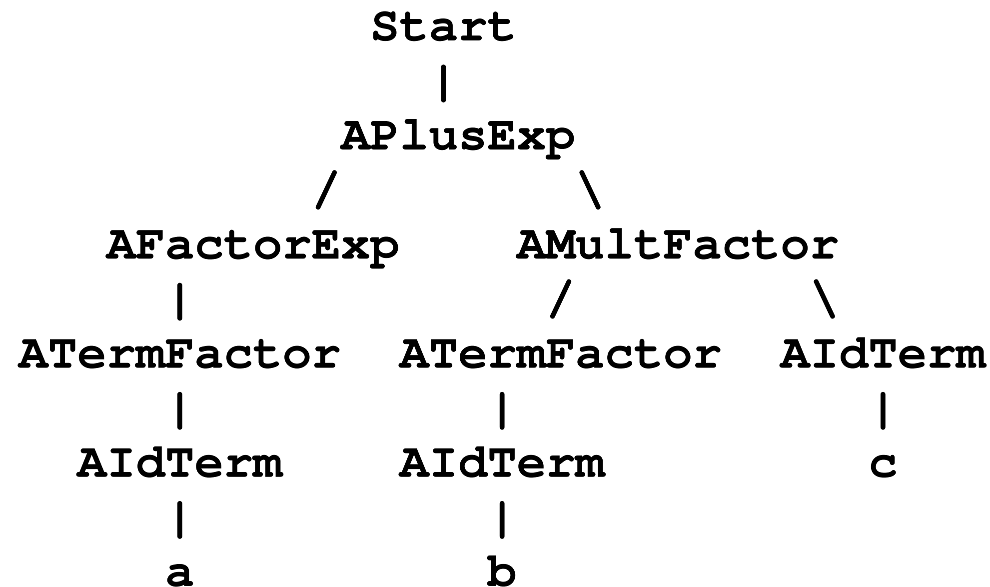
Review of SableCC CST to AST transformation:

```
Productions

        exp    = {plus}      exp plus factor
               | {minus}     exp minus factor
               | {factor}    factor;

        factor = {mult}      factor star term
               | {divd}      factor slash term
               | {term}      term;

        term   = {paren}     l_par exp r_par
               | {id}        id
               | {number}    number;
```

Concrete syntax trees:

Given some grammar, SableCC generates a parser that in turn builds a concrete syntax tree (CST) for an input program.

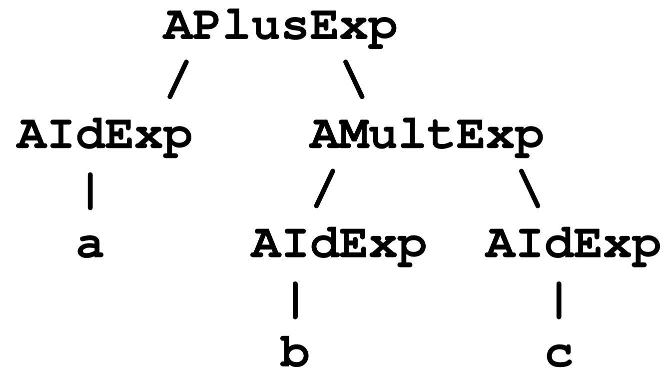
A parser built from the Tiny grammar creates the following CST for the program '`a+b*c`':



This CST has many unnecessary intermediate nodes.

Abstract syntax trees:

We only need an abstract syntax tree (AST) to operate on:



Recall that `bison` relies on user-written actions after grammar rules to construct an AST.

As an alternative, SableCC 3 actually allows the user to define an AST and the CST⟶AST transformations formally, and can then translate CSTs to ASTs automatically.

AST for the Tiny expression language:

```
Abstract Syntax Tree

        exp = {plus}          [l]:exp [r]:exp
            | {minus}         [l]:exp [r]:exp
            | {mult}          [l]:exp [r]:exp
            | {divd}          [l]:exp [r]:exp
            | {id}            id
            | {number}        number;
```

AST rules have the same syntax as rules in the `Production` section except for CST$\longrightarrow$AST transformations (obviously).

Extending Tiny productions with CST→AST transformations:

```
Productions
     cst_exp {-> exp} =
          {cst_plus}    cst_exp plus factor
                        {-> New exp.plus(cst_exp.exp,factor.exp)}  |
          {cst_minus}   cst_exp minus factor
                        {-> New exp.minus(cst_exp.exp,factor.exp)} |
          {factor}      factor {-> factor.exp};

     factor {-> exp} =
          {cst_mult}    factor star term
                        {-> New exp.mult(factor.exp,term.exp)} |
          {cst_divd}    factor slash term
                        {-> New exp.divd(factor.exp,term.exp)} |
          {term}        term {-> term.exp};

     term {-> exp} =
          {paren}       l_par cst_exp r_par {-> cst_exp.exp} |
          {cst_id}      id         {-> New exp.id(id)}       |
          {cst_number}  number     {-> New exp.number(number)};
```

A CST production alternative for a plus node:

```
cst_exp = {cst_plus} cst_exp plus factor
```

needs extending to include a CST⟶AST transformation:

```
cst_exp {-> exp} = {cst_plus} cst_exp plus factor
                       {-> New exp.plus(cst_exp.exp,factor.exp)}
```

- `cst_exp {-> exp}` on the LHS specifies that the CST node `cst_exp` should be transformed to the AST node `exp`.
- `{-> New exp.plus(cst_exp.exp, factor.exp)}` on the RHS specifies the action for constructing the AST node.
- `exp.plus` is the kind of `exp` AST node to create. `cst_exp.exp` refers to the transformed AST node `exp` of `cst_exp`, the first term on the RHS.

5 types of explicit RHS transformation (action):

1. Getting an existing node:

```
{paren}  l_par cst_exp r_par {-> cst_exp.exp}
```

2. Creating a new AST node:

```
{cst_id} id {-> New exp.id(id)}
```

3. List creation:

```
{block} l_brace stm* r_brace {-> New stm.block([stm])}
```

4. Elimination (but more like nullification):

```
{-> Null}
{-> New exp.id(Null)}
```

5. Empty (but more like deletion):

```
{-> }
```