Abstract Syntax Trees

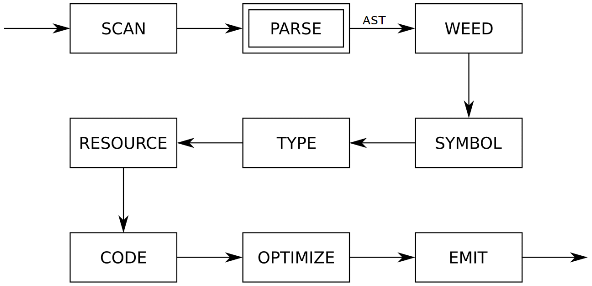
COMP 520: Compiler Design (4 credits)

Alexander Krolik

`alexander.krolik@mail.mcgill.ca`

MWF 10:30-11:30, TR 1100

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



Readings

Crafting a Compiler (recommended)

- Chapter 7

Announcements (Wednesday/Friday, January 22nd/24th)

Milestones

- Group signup form `https://forms.gle/HgeCthhH3dwD6WmG7`, fill this out over the next 2 weeks

Assignment 1

- Questions in a few minutes!
- **Due**: Friday, January 24th 11:59 PM

Midterm

- **Date**: Tuesday, February 25th from 6:00 - 7:30 PM in RPHYS 112

Background on Programming Languages - Expressions

An *expression* is a programming language construct which is associated with a *value*. We can define them recursively:

- Base cases
  - Literals: “string”, `true`, 1.0, . . .
  - Identifiers: `a`, `myVar`, . . .
- Recursive cases
  - Binary operations: <Expression> <Op> <Expression>
  - Unary operations: <Op> <Expression>
  - Parentheticals: (Expression)
  - Function calls

Note that in the above definitions, we do not specify any *type* information (e.g. `int`, `float`, etc.).

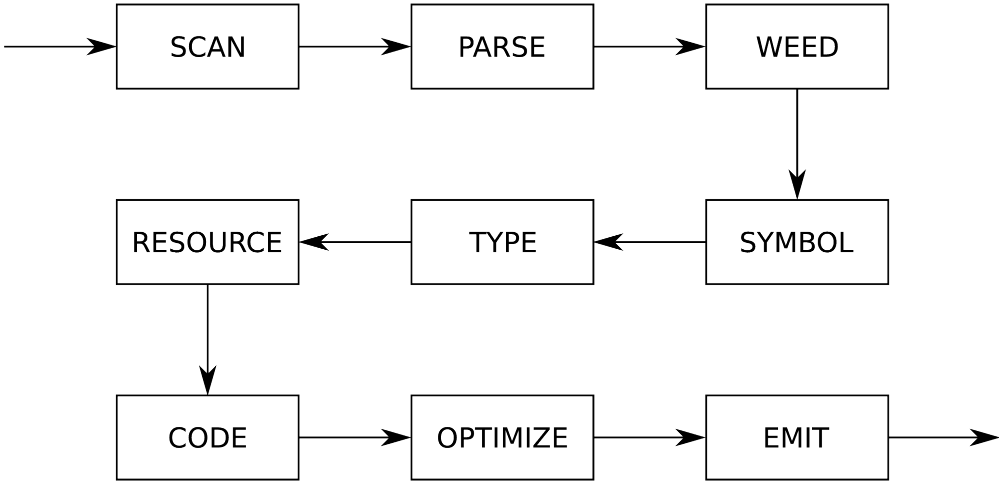
Background on Programming Languages - Statements

A *statement* is a programming language construct which gives structure to expressions and defines the flow of execution

- Control-flow constructs: if, while, for, ...
- Assignments
- Declarations (maybe)
- Expression statements (e.g. `foo();`)
- ...

Recap on Phases of the Compiler

A compiler is a **modular** pipeline of phases, with each phase handling different concerns.



The frontend of the compiler consists (informally) of the following phases and their responsibilities:

- **Scanning**: Verifying the source input characters and producing tokens;
- **Parsing**: Verifying the sequence of tokens and associating related tokens;
- **Symbol/Type**: Verifying the type correctness of expressions and their use in statements

Important: A grammar specifies the definition of “groupings” of non-terminals and terminals *without* types! Types are *semantic* information, and left to a later phase of the compiler.

(We could do so for some cases, but it will explode the size of the grammar)

Assignment 1

Questions

- Who is using `flex+bison`? `SableCC`?
- Any questions about the tools?
- What stage is everyone at: scanner, tokens, parser?
- Any questions about the language?
- Any questions about the requirements?

Notes

- Use the assignment template (`https://github.com/comp520/Assignment-Template`)
- Make sure it runs using the scripts!
- Include in your README file, *all* resources that were consulted, or state “I worked alone”
- No AST building or typechecking in this assignment

ASTs

Internal Representations

Building ASTs

Bison

SableCC *(Optional)*

Pretty Printing

Compiler Architecture

- A compiler *pass* is a traversal of the program; and
- A compiler *phase* is a group of related passes.

One-pass compiler

A *one-pass* compiler scans the program only once - it is naturally single-phase. The following all happen at the same time

- Scanning
- Parsing
- Weeding
- Symbol table creation
- Type checking
- Resource allocation
- Code generation
- Optimization
- Emitting

Compiler Architecture

This is a terrible methodology!

- It ignores natural modularity;
- It gives unnatural scope rules; and
- It limits optimizations.

Historically

It used to be popular for early compilers since

- It's fast (if your machine is slow); and
- It's space efficient (if you only have 4K).

A modern *multi-pass* compiler uses 5–15 phases, some of which may have many individual passes: you should skim through the optimization section of '`man gcc`' some time!

Intermediate Representations

A multi-pass compiler needs an *intermediate representation* of the program between passes that may be updated/augmented along the pipeline. It should be

- An accurate representation of the original source program;
- Relatively compact;
- Easy (and quick) to traverse; and
- In optimizing compilers, easy and fruitful to analyze and improve.

In practice

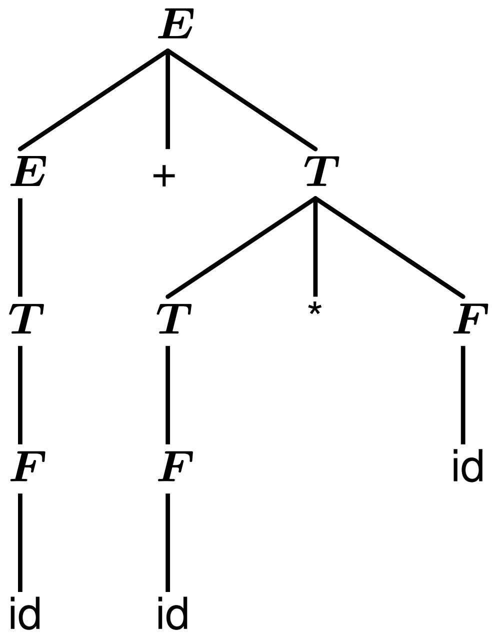
These are competing demands, so some intermediate representations are

- More suited to certain tasks than others; and
- More suited to certain languages than others.

In this class, we focus on tree representations.

Concrete Syntax Trees

A parse tree, also called a *concrete syntax tree* (CST), is a tree formed by following the exact CFG rules. Below is the corresponding CST for the expression `a+b*c`

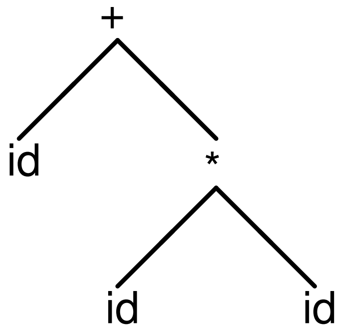


Note that this includes a lot of information that is not necessary to understand the original program

- Terms and factors were introduced for associativity and precedence; and
- Tokens $+$ and $\star$ correspond to the type of the $\boldsymbol{E}$ node.

Abstract Syntax Trees

An *abstract syntax tree* (AST), is a much more convenient tree form that represents a more abstract grammar. The same `a+b*c` expression can be represented as



In an AST

- Only important terminals are kept; and
- Intermediate non-terminals used for parsing are removed.

This representation is thus *independent* of the syntax and *independent* of the grammar.

Constructing an AST

Designing the right AST nodes is important for later phases of the compiler as they will extensively use the AST. The set of AST nodes should

- Represent all distinct programming language constructs; and
- Be minimal, avoiding excess intermediate nodes (e.g. terms and factors).

A concise AST will have ~1 node type for each type of programming language construct.

Example

In MiniLang the main construct types are declarations, statements, and expressions. The AST would therefore include

- `[Program]`
- `[Declaration]`
- `Statement`
- `Expression`

Constructing an AST

Language constructs may be several variants (e.g. mathematical expressions `+`, `-`, `*`, `/`, unary `-`)

Solutions

1. Inefficient: Create a separate expression node for each kind
   - AddExpression;
   - MinusExpression;
   - TimesExpression;
   - DivideExpression;
   - UnaryMinusExpression;

   Problem: Large duplication of near identical code (traversal and AST)

2. Preferred: Create a parametrized expression
   - Expression(op);

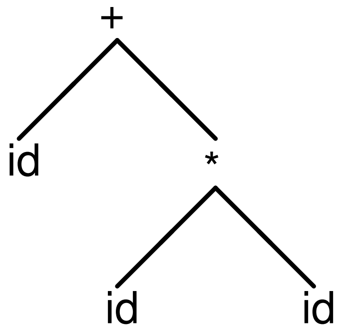
   Slight problem: There may be special cases which have distinct functionality

Why not use different nodes for each kind? Repetitive traversal code and structures

Why not use a single node for all constructs? Lack of type information to constrain methods

Intermediate Language

Alternatively, instead of constructing the tree a compiler can generate code for an internal compiler-specific grammar, also known as an *intermediate language*.



Early multi-pass compilers wrote their IL to disk between passes. For the above tree, the string `+(id,*(id,id))` would be written to a file and read back in for the next pass.

It may also be useful to write an IL out for debugging purposes.

Examples of Intermediate Languages

- Java bytecode
- C, for certain high-level language compilers
- Jimple, a 3-address representation of Java bytecode specific to Soot, created by Raja Vallee-Rai at McGill
- Simple, the precursor to Jimple, created for McCAT by Prof. Hendren and her students
- Gimple, the IL based on Simple that `gcc` uses
- LLVM-IR

In this course, you will generally use an AST as your IR without the need for an explicit IL.

Note: somewhat confusingly, both industry and academia use the terms IR and IL interchangeably.

ASTs

Internal Representations

Building ASTs

Bison

SableCC *(Optional)*

Pretty Printing

Building IRs

Intuitively, as we recognize parts of the source program during parsing, we assemble them into an IR.

- Requires extending the parser; and
- Executing *semantic actions* during the process.

Semantic actions

- Arbitrary actions executed during the parser execution.

In other words, each time we recognize part of the source program (i.e. apply a reduction), the semantic action creates a new tree for this portion of the program.

Building IRs

Each time a semantic action is applied, it produces a *semantic value*.

Semantic values

Values associated with terminals and non-terminals;

- **Terminals**: provided by the scanner (base case);
- **Non-terminals**: created by the parser;

Each semantic value is thus the root of a subtree in the AST!

Once the entire AST is formed (reduced to the root)

- Tokens form the leaves of the tree; and
- Variables form the internal nodes

Note: Not all non-terminals have distinct node types, this is an **A**ST after all!

Building IRs - LR Parsers

When a bottom-up parser executes it maintains a

- *Syntactic stack* – the working stack of symbols; and a
- *Semantic stack* – the values associated with each grammar symbol on the syntactic stack.

We use the semantic stack to recursively build the AST, executing semantic actions on *reduction*.

In your code

A reduction using rule $A \rightarrow \gamma$ executes a semantic action that

- Synthesizes symbols in γ; and
- Produces a new node representing A

In other words, each time we apply a reduction, the semantic action merges subtrees into a new rooted tree. Using this mechanism, we can build an AST.

ASTs

Internal Representations

Building ASTs

Bison

SableCC *(Optional)*

Pretty Printing

Constructing an AST with `flex/bison`

Begin defining your AST structure in a header file `tree.h`. Each node type is defined in a `struct`

```
typedef struct EXP EXP;
struct EXP {
      ExpressionKind kind;
      union {
            char *identifier;
            int intLiteral;
            struct { EXP *lhs; EXP *rhs; } binary;
      } val;
};
```

Node kind

For nodes with more than one kind (i.e. expressions), we define an enumeration `ExpressionKind`

```
typedef enum {
      k_expressionKindIdentifier,
      k_expressionKindIntLiteral,
      k_expressionKindAddition,
      k_expressionKindSubtraction,
      k_expressionKindMultiplication,
      k_expressionKindDivision
} ExpressionKind;
```

Node value

Node values are stored in a union. Depending on the node kind, a different part of the union is used.

Constructing an AST with `flex/bison`

Next, define constructors for each node type in `tree.c`

```
EXP *makeEXP_intLiteral(int intLiteral)
{
      EXP *e = malloc(sizeof(EXP));
      e->kind = k_expressionKindIntLiteral;
      e->val.intLiteral = intLiteral;
      return e;
}
```

The corresponding declaration goes in `tree.h`

```
EXP *makeEXP_intLiteral(int intLiteral);
```

Constructing an AST with `flex/bison`

Finally, we can extend `bison` to include the tree-building actions in `tiny.y`.

Semantic values

For each type of semantic value, add an entry to `bison`'s union directive

```
%union {
      int int_val;
      char *string_val;
      struct EXP *exp;
}
```

For each token type that has an associated value, extend the token directive with the association. For non-terminals, add `%type` directives

```
%type <exp> program exp
%token <int_val> tINTVAL
%token <string_val> tIDENTIFIER
```

Semantic actions

```
exp : tINTVAL     { $$ = makeEXP_intLiteral($1); }
    | exp '+' exp { $$ = makeEXP_plus($1, $3); }
```

Using an AST

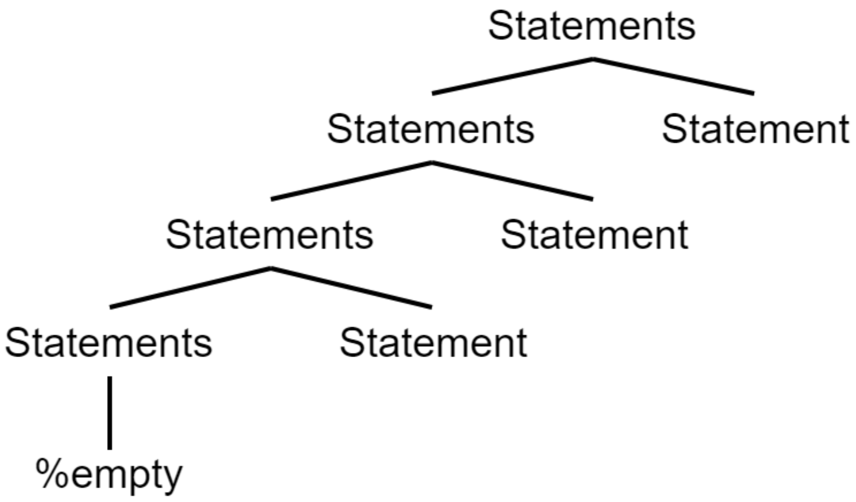
Traversing an AST is done using a set of mutually recursive functions, each switching on the kind and performing the appropriate action.

```
/* pretty.h */
void prettyEXP(EXP *e);

/* pretty.c */
void prettyEXP(EXP *e)
{
      switch (e->kind) {
            case k_expressionKindIdentifier:
                  printf("%s", e->val.identifier);
                  break;
            case k_expressionKindIntLiteral:
                  printf("%i", e->val.intLiteral);
                  break;
            case k_expressionKindAddition:
                  printf("(");
                  prettyEXP(e->val.binary.lhs);
                  printf("+");
                  prettyEXP(e->val.binary.rhs);
                  printf(")");
                  break;
            [...]
      }
}
```

LALR(1) Lists

LALR grammars typically build lists using left-recursion, largely for efficiency. Consider the following example for lists of expressions

```
statements : %empty { $$ = NULL; }
           | statements statement { $$ = $2; $$->next = $1; }
;

statement : tIDENT '=' exp ';' { $$ = makeSTATEMENT_assign($1, $3); }
;
```



The lists are naturally backwards!

LALR(1) Lists

Processing backwards lists requires head recursion to start with the first element

```
struct STATEMENT {
      StatementKind kind;
      union {
            struct { char *identifier; EXP *value; } assignment;
      } val;
      STATEMENT *next;
};

void traverseSTATEMENT(STATEMENT *s) {
      if (s == NULL) {
            return;
      }

      traverseSTATEMENT(s->next);
      /* TODO: ... */
}
```

What effect would a call stack size limit have?

Extending the AST

As mentioned before, a modern compiler uses 5–15 phases. Each phases of the compiler may contribute additional information to the IR.

- **Scanner**: line numbers;
- **Symbol tables**: meaning of identifiers;
- **Type checking**: types of expressions; and
- **Code generation**: assembler code.

Extending the AST - Manual Line Numbers

If using manual line number incrementing, adding line numbers to AST nodes is simple.

1. Introduce a global `lineno` variable in the `main.c` file

```
int lineno;
int main(){
      lineno = 1; /* input starts at line 1 */
      yyparse();
      return 0;
}
```

2. increment `lineno` in the scanner

```
%{
      extern int lineno;      /* declared in main.c */
%}

%%
[ \t]+                        /* no longer ignore \n */
\n     lineno++;              /* increment for every \n */
```

Extending the AST - Manual Line Numbers

3. Add a `lineno` field to the AST nodes

```
struct EXP {
      int lineno;
      [...]
};
```

4. Set `lineno` in the node constructors

```
EXP *makeEXP_intLiteral(int intLiteral)
{
      EXP *e = malloc(sizeof(EXP));
      e->lineno = lineno;
      e->kind = k_expressionKindIntLiteral;
      e->val.intLiteral = intLiteral;
      return e;
}
```

Extending the AST - Automatic Line Numbers

1. Turn on line numbers in `flex` and add the user action

```
%{
      #define YY_USER_ACTION yylloc.first_line = yylloc.last_line = yylineno;
%}
%option yylineno
```

2. Turn on line numbers in `bison`

```
%locations
```

3. Add a `lineno` field to the AST nodes

```
struct EXP {
      int lineno;
      [...]
};
```

Extending the AST - Automatic Line Numbers

4. Extend each constructor to take an `int lineno` parameter

```
EXP *makeEXP_intLiteral(int intLiteral, int lineno)
{
      EXP *e = malloc(sizeof(EXP));
      e->lineno = lineno;
      e->kind = k_expressionKindIntLiteral;
      e->val.intLiteral = intLiteral;
      return e;
}
```

5. For each semantic action, call the constructor with the appropriate line number

```
exp : tINTVAL { $$ = makeEXP_intLiteral($1, @1.first_line); }
```

Accessing the token location is done using `@<token position>.<attribute>`

Extending the AST - Comparison

`https://github.com/comp520/Examples/tree/master/flex%2Bbison/linenumbers`

Given the example program `3 + 4`, we expect the expression node to be located on line 1.

Manual

```
(3[1]+[2]4[1])
```

Automatic

```
(3[1]+[1]4[1])
```

What happened?

Semantic actions are executed when a rule is applied (reduction). An expression grammar can only reduce `3 + 4` if it knows the next token - in this case, the newline.

```
makeEXPintconst
makeEXPintconst
lineno++
makeEXPplus
```

ASTs

Internal Representations

Building ASTs

Bison

SableCC *(Optional)*

Pretty Printing

Constructing an AST with SableCC

SableCC 2 automatically generates a CST for your grammar, with nodes for terminals and non-terminals. Consider the grammar for the TinyLang language

Scanner

```
Package tiny;
Helpers
    tab   = 9;
    cr    = 13;
    lf    = 10;
    digit = [’0’..’9’];
    lowercase = [’a’..’z’];
    uppercase = [’A’..’Z’];
    letter  = lowercase | uppercase;
    idletter = letter | ’_’;
    idchar  = letter | ’_’ | digit;
Tokens
    eol   = cr | lf | cr lf;
    blank = ’ ’ | tab;
    star  = ’*’;
    slash = ’/’;
    plus  = ’+’;
    minus = ’-’;
```

Constructing an AST with SableCC

```
    l_par = ’(’;
    r_par = ’)’;
    number  = ’0’| [digit-’0’] digit*;
    id    = idletter idchar*;
Ignored Tokens
    blank, eol;
```

Parser

```
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;
```

Constructing an AST with SableCC

SableCC generates subclasses of '`Node`' for terminals, non-terminals and production alternatives

- **Classes for terminals**: '`T`' followed by (capitalized) terminal name

  `TEol, TBlank, ..., TNumber, TId`

- **Classes for non-terminals**: '`P`' followed by (capitalized) non-terminal name

  `PExp, PFactor, PTerm`

- **Classes for alternatives**: '`A`' followed by (capitalized) alternative name and (capitalized) non-terminal name

  `APlusExp (extends PExp), ..., ANumberTerm (extends PTerm)`

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

[...]
```

SableCC Directory Structure

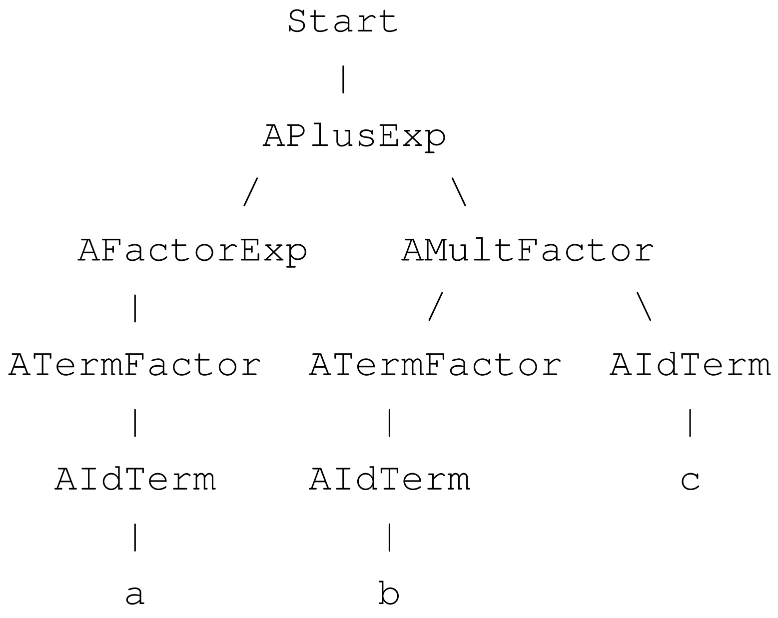
SableCC populates an entire directory structure

```
tiny/
    |--analysis/  Analysis.java
    |             AnalysisAdapter.java
    |             DepthFirstAdapter.java
    |             ReversedDepthFirstAdapter.java
    |
    |--lexer/     Lexer.java lexer.dat
    |             LexerException.java
    |
    |--node/      Node.java TEol.java ... TId.java
    |             PExp.java PFactor.java PTerm.java
    |             APlusExp.java ...
    |             AMultFactor.java ...
    |             AParenTerm.java ...
    |
    |--parser/    parser.dat Parser.java
    |             ParserException.java ...
    |
    |-- custom code directories, e.g. symbol, type, ...
```

SableCC - 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`’



```
            Start
              |
           APlusExp
          /        \
   AFactorExp    AMultFactor
     |              /        \
ATermFactor  ATermFactor  AIdTerm
     |          |            |
  AIdTerm    AIdTerm         c
     |          |
     a          b
```

This CST has many unnecessary intermediate nodes. Can you identify them?

SableCC - Abstract Syntax Trees

We only need an abstract syntax tree (AST) to maintain the same useful information for further analyses and processing

```
         APlusExp
         /     \
    AIdExp    AMultExp
      |        /       \
      a     AIdExp  AIdExp
              |        |
              b        c
```

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.

Constructing an AST with SableCC

For the TinyLang expression language, the AST definition is as follows

```
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 productions, except that their elements define the abstract structure. We remove all unnecessary tokens and intermediate non-terminals.

Constructing an AST with SableCC

Using the AST definition, we augment each production in the grammar with a 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)};
```

Constructing an AST with SableCC

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.

Constructing an AST with SableCC

There are 5 types of explicit RHS transformations (actions)

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)

```
{-> }
```

Constructing an AST with SableCC

Writing down straightforward, non-abstracting CST→AST transformations can be tedious. For example, consider the following production of optional and list elements

```
prod = elm1 elm2* elm3+ elm4?;
```

An equivalent AST construction would be

```
prod{-> prod} = elm1 elm2* elm3+ elm4?
                {-> New prod.prod(
                          elm1.elm1,
                          [elm2.elm2],
                          [elm3.elm3],
                          elm4.elm4)
                };
```

SableCC 3 Documentation

- `http://www.natpryce.com/articles/000531.html`
- `http://sablecc.sourceforge.net/documentation/cst-to-ast.html`

Announcements (Monday, January 27th)

Milestones

- Group signup form `https://forms.gle/HgeCthhH3dwD6WmG7`, fill this out over the next week
- How’s everyone doing?

Assignments

- Assignment 1 will be graded soon! Solution programs will be available on myCourses
- Assignment 2 out today! **Due**: Friday, February 7th 11:59 PM

Midterm

- **Date**: Tuesday, February 25th from 6:00 - 7:30 PM in RPHYS 112
  - Conflicts with: COMP 361D2, COMP 362, MATH 315

ASTs

Internal Representations

Building ASTs

Bison

SableCC *(Optional)*

Pretty Printing

Pretty Printing

Pretty printing is a compiler function that outputs the parsed program in its “original”, “pretty” source form (i.e. in the *original* source language)

The recursive form of ASTs allows us to easily construct recursive traversals as shown below.

```
void prettyEXP(EXP *e)
{
      switch (e->kind) {
            case k_expressionKindIdentifier:
                  printf("%s", e->val.identifier);
                  break;
            case k_expressionKindIntLiteral:
                  printf("%i", e->val.intLiteral);
                  break;
            case k_expressionKindAddition:
                  printf("(");
                  prettyEXP(e->val.binary.lhs);
                  printf("+");
                  prettyEXP(e->val.binary.rhs);
                  printf(")");
                  break;
            [...]
}
```

Pretty Printing

Given a parsed AST, invoking the pretty printer starts at the root node.

```
#include "tree.h"
#include "pretty.h"

void yyparse();

EXP *root;

int main()
{
      yyparse();
      prettyEXP(root);
      return 0;
}
```

Pretty printing the expression `a*(b-17) + 5/c` in TinyLang will output

```
((a*(b-17))+(5/c))
```

Question: Why the extra parentheses?

Pretty Printing

If *parse*(P) constructs T and *pretty*(T) reconstructs the text of P, then

$$\textit{pretty}(\textit{parse}(P)) \approx P$$

Even better, we have a stronger relation which says that

$$\textit{pretty}(\textit{parse}(\textit{pretty}(\textit{parse}(P)))) \equiv \textit{pretty}(\textit{parse}(P))$$

Of course, this is a necessary but not sufficient condition for parser correctness.

The testing strategy for a parser that constructs an abstract syntax tree T from a program P usually involves a pretty printer, but requires other tests.

Important observations

- Pretty printers do not output an identical program to the input (whitespace ignored, etc.); and
- Pretty printers should make some effort to be “pretty”.