Abstract Syntax Trees

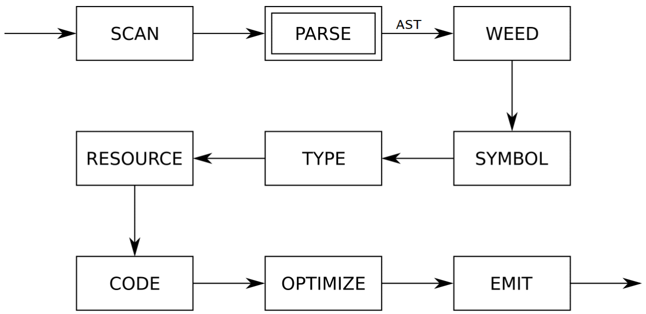
COMP 520: Compiler Design (4 credits)

Alexander Krolik

alexander.krolik@mail.mcgill.ca

MWF 9:30-10:30, TR 1080

http://www.cs.mcgill.ca/~cs520/2018/



Announcements (Wednesday, January 24th)

Milestones

- Group signup form `https://goo.gl/forms/L6Dq5CHLvbjNhT8w1`
- Office hours
  - Alex: Wednesdays 10:30-11:30
  - David: Thursdays 11:30-12:30

Assignment 1

- **Due**: Sunday, January 28th 11:59 PM

Midterm

- **Preferred**: Friday, March 16th, 1.5 hour "in class" midterm. *Thoughts?*
- **Otherwise**: Week of Monday, March 12th, 1.5 hour "evening" midterm.

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

- You **must** use the assignment template `https://github.com/comp520/Assignment-Template`
- You **must** make sure it runs using the scripts!
- No AST building or typechecking this assignment

Due: Sunday, January 28th 11:59 PM

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.

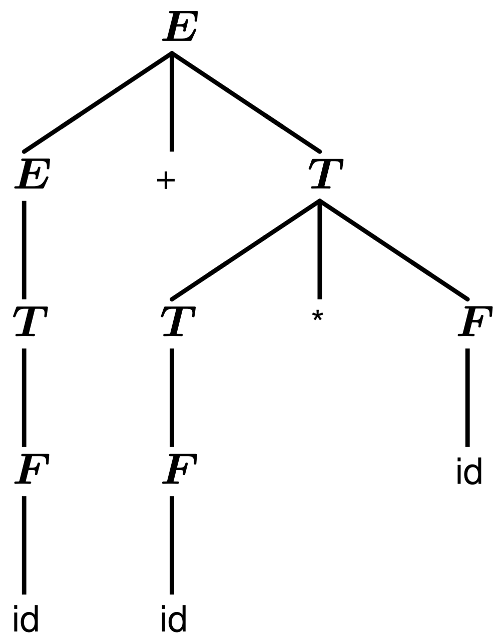
These are competing demands, so some intermediate representations are more suited to certain tasks than others. Some intermediate representations are also more suited to certain languages than others.

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`

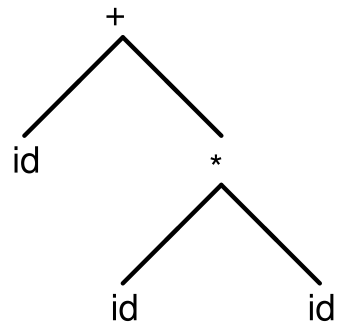


Note that this includes a lot of information that is not necessary to understand the original program

- 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



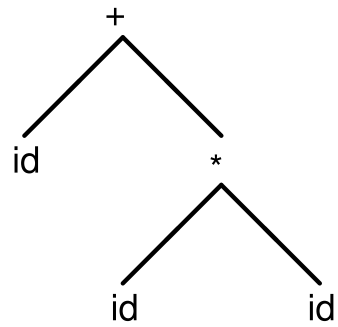
In an AST

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

This representation is thus *independent* of the syntax.

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

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.

Building IRs

Intuitively, as we recognize various parts of the source program, 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.

Semantic values

- Values associated with terminals and non-terminals;
  - **Terminals**: provided by the scanner (extra information other than the token type);
  - **Non-terminals**: created by the parser;

Building IRs - LR Parsers

When a bottom-up parser executes it

- Maintains a *syntactic stack* – the working stack of symbols; and
- Also maintains 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.

Using this mechanism, we can build an AST.

Constructing an AST with `flex/bison`

Begin by defining your AST structure in a `C` 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`

Node value

Node values are stored in a union. Depending on the kind of the node, 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); }
```

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 - Comarison

`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
```

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

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

Constructing an AST with SableCC

```
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$\longrightarrow$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$\longrightarrow$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

Pretty Printing

A recursive AST traversal that outputs the program in its “original”, “pretty” source form.

```
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

```c
#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))`

Why the extra parentheses?

Pretty Printing

The testing strategy for a parser that constructs an abstract syntax tree T from a program P usually involves a pretty printer

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 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.

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”.