Cecko

Assignment 2 – Parsing

bison

- **bison**
  - 1985, part of the GNU project
    - based on **yacc** (yet another compiler-compiler, 1975)
  - compatible with **flex**
- LALR(1) parser generator

```
From: RMS@MIT-OZ@mit-eddie.UUCP (Richard Stallman)
Subject: new UNIX implementation
Message-ID: <771@mit-eddie.UUCP>
Date: Tue, 27-Sep-83 13:35:59 EDT
Date-Received: Thu, 29-Sep-83 07:38:11 EDT

Free Unix!

Starting this Thanksgiving I am going to write a complete
Unix-compatible software system called GNU (for Gnu's Not Unix), and
give it away free to everyone who can use it.  Contributions of time,
money, programs and equipment are greatly needed.

To begin with, GNU will be a kernel plus all the utilities needed to
write and run C programs: editor, shell, C compiler, linker,
assembler, and a few other things.  After this we will add a text
formatter, a YACC, an Empire game, a spreadsheet, and hundreds of
other things.  We hope to supply, eventually, everything useful that
normally comes with a Unix system, and anything else useful, including
```

- **bison**
  - 1985, part of the GNU project
    - based on **yacc** (yet another compiler-compiler, 1975)
  - compatible with **flex**
- LALR(1) parser generator
- Input
  - caparser.y
    - Configuration (do not touch!)
    - Declarations of terminals (do not touch!)
      - Terminal names start with uppercase
    - Declarations of semantic types for non-terminals (not used in Assignment 2)
      - Non-terminal names start with lowercase
    - The grammar
    - C/C++ code fragments (not used in Assignment 2)
- Output (in modern C++ mode)
  - caparser.hpp
    - Declarations of **make_T** functions for terminals
    - Declaration of the parser class
  - caparser.cpp
    - Implementation of the parser class
    - Essentially an interpreter of LALR(1) Action/Goto tables
    - Contains user-defined C/C++ code fragments

- Input file syntax
  - The language supports both /*...*/ and //... comments
  - The first section is line-oriented

```
// switches and definitions
%code requires
{
   // this code is emitted to the .hpp file
}
%code
{
   // this code is emitted to the beginning of the .cpp file
}
// declarations of terminals and their semantic types
// the strings are used in syntax-error messages generated by the parser
%token       EOF    0  "end of file"
%token       LBRA      "["
%token<int>  INTLIT    "integer literal"
// declarations of semantic types of non-terminals (not used in Asgn. 2)
%type<casem::ArgList>  argument_list

%%

// grammar rules

%%

// this code is emitted to the end of the .cpp file
```

- Rule syntax
  - All rules for a non-terminal usually concentrated in one place
    - Right-hand-side parts separated by |
      - This is not a regular expression, just a list of sequences
    - Empty RHS is either just empty or contains the **%empty** metasymbol
  - A (multi-)rule may be arbitrarily spread on multiple lines
    - The syntax includes : and ;

```
stmt:
      IF LPAR expr RPAR
        stmt
    | IF LPAR expr RPAR
        stmt
        ELSE
        stmt
    | WHILE LPAR expr RPAR
        stmt
    | RETURN INTLIT SEMIC
    ;
```

  - The order of rules is not significant
  - The initial nonterminal of our grammar is named **translation_unit**
    - Defined by the **%start** switch
    - Inclusion of the **EOF** terminal in the terminal rule is not required

- **bison** is a LALR(1) parser generator
  - bottom-up analysis based on *item automaton*
- Built-in rules to resolve conflicts
  - based on explicitly declared priority/associativity of terminals as operators
  - if priorities not defined, favor shift over reductions
  - it simplifies grammars of simple languages, dangerous in more complex cases
  - Forbidden in home assignments
- Understanding conflicts
  - Since version 3.7.0, bison can display counterexamples for conflicts:
    - This is not necessarily the only or the smallest counterexample

```
solution/caparser.y: warning: shift/reduce conflict on token "else"
  Example: "if" "(" expr ")" "if" "(" expr ")" stmt • "else" stmt
  Shift derivation
    stmt
    ↳ 3: "if" "(" expr ")" stmt
                           ↳ 4: "if" "(" expr ")" stmt • "else" stmt
  Reduce derivation
    stmt
    ↳ 4: "if" "(" expr ")" stmt                          "else" stmt
                           ↳ 3: "if" "(" expr ")" stmt •
```

  - All versions produce a human-readable dump of the item automaton
    - <build-folder>/stud-sol/caparser.y.output

- Example 1 - an ambiguous grammar
  - Grammar part dumped in the .output file:

```
3 stmt: "if" "(" expr ")" stmt
4     | "if" "(" expr ")" stmt "else" stmt
5     | "while" "(" expr ")" stmt
6     | "return" "integer literal" ";"
```

  - The corresponding counterexample:

```
Example: "if" "(" expr ")" "if" "(" expr ")" stmt • "else" stmt
Shift derivation
  stmt
  ↳ 3: "if" "(" expr ")" stmt
                         ↳ 4: "if" "(" expr ")" stmt • "else" stmt
Reduce derivation
  stmt
  ↳ 4: "if" "(" expr ")" stmt                          "else" stmt
                         ↳ 3: "if" "(" expr ")" stmt •
```

    - The counterexample shows that the word

```
"if" "(" expr ")" "if" "(" expr ")" stmt "else" stmt
```

    - can be derived by two different derivations from the same **stmt** nonterminal
      - applying rule 3, then rule 4
      - applying rule 4, then rule 3 (on the first occurrence of stmt)
    - The grammar ambiguity causes a **shift-reduce conflict**:
      - after parsing the nested stmt nonterminal, looking ahead to the "else" terminal
  - This particular problem shall be solved by rewriting the grammar unambiguously

- Example 2 - a LALR(1) conflict
  - The grammar is not ambiguous but it does not satisfy the LALR(1) conditions

```
 7 expr: "integer literal"
 8     | "identifier"
 9     | "(" typename ")" expr
10     | "(" expr ")"

11 typename: "identifier"
```

    - The counterexample produced by bison is too long, this is manually simplified:
      - First reduce derivation

```
expr
↳ 10: "(" expr                 ")"
         ↳ 8: "identifier" •
```

      - Second reduce derivation

```
expr
↳ 9: "(" typename              ")" expr
        ↳ 11: "identifier" •
```

    - In this case, there are two different words derived from **expr**

```
"(" "identifier" • ")"
"(" "identifier" • ")" expr
```

    - The two words have a common prefix (shown by the position of the dot)
    - The look-ahead terminal after that dot is also the same
    - The two derivations use different rules just before the dot
      - The parser can't tell which rule to use for reducing the text before the dot
    - This is termed a **reduce-reduce conflict**

- Example 2 - a LALR(1) conflict
  - The conflict is also visible in the .output file:

```
State 29 conflicts: 1 reduce/reduce
```

    - The corresponding state is dumped in the same file as:
      - The LR(0) items forming the state:

```
    8 expr: "identifier" •
   11 typename: "identifier" •
```

      - The corresponding actions for this state:

```
    ")"         reduce using rule 8 (expr)
    ")"         [reduce using rule 11 (typename)]
    $default    reduce using rule 8 (expr)
```

    - In this case, there are two possible actions for the same look-ahead of ")"
      - The action shown in [brackets] is suppressed by some built-in conflict resolution rules
      - The $default look-ahead means "in all other cases"
  - If you don't understand how/why this state was produced, trace it back:
    - Try to find the text "state 29":

```
State 23

    9 expr: "(" • typename ")" expr
   10     | "(" • expr ")"

    "("               shift, and go to state 23
    "identifier"      shift, and go to state 29
    "integer literal" shift, and go to state 25

    expr      go to state 30
    typename  go to state 31
```

- Example 2 - a LALR(1) conflict

```
State 23

    9 expr: "(" • typename ")" expr
   10     | "(" • expr ")"

    "("               shift, and go to state 23
    "identifier"      shift, and go to state 29
    "integer literal" shift, and go to state 25
```

- The dump does not show the *closure* items, but it shows the *actions* resulting from them:
- Because the dots are before *non-terminals*, there are also the *gotos:*

```
    expr      go to state 30
    typename  go to state 31

State 29

    8 expr: "identifier" •
   11 typename: "identifier" •

    ")"       reduce using rule 8 (expr)
    ")"       [reduce using rule 11 (typename)]
    $default  reduce using rule 8 (expr)
```

- In this case, the State 23 and the shift action to the State 29 is sufficient for understanding the problem
- In more complex cases, it may be necessary to trace back more states
  - In theory, tracing every path back to the initial state will produce the language of all counterexamples for State 23

- Example 2 - a LALR(1) conflict
  - The underlying cause of this conflict is the fact that these two expressions

```
(x)
(x)y
```

    - have the same prefix but the meaning of x is completely different
  - In theory, it could be handled by massively rewriting the grammar like

```
expr: expr_or_typename
    | expr_not_typename
    ;
expr_not_typename: INTLIT
    | LPAR expr_or_typename RPAR expr
    | LPAR expr_or_typename RPAR
    | LPAR expr_not_typename RPAR
    ;
expr_or_typename: IDF
    ;
typename: expr_or_typename
    ;
```

    - **expr_or_typename** describes the intersection of the two languages produced from the **expr** and **typename** nonterminals
    - **expr_not_typename** describes the set-difference of the two languages
    - **typename_not_expr** would be empty in this grammar
    - Although there still are the two words having the same prefix, all the reduction required before applied inside the
  - In the real C language, such a rewrite does not solve all the conflicts...

- The real situation in C/C++
  - The following text (inside a function body)

```
x(*y);
```

    - may be a statement containing a function call applied to a pointer dereference
    - may be a declaration of variable **y** as a pointer to the type **x**
      - the parentheses are superfluous but allowed
  - The grammar is really ambiguous between declarations and statements
    - Trying to solve this ambiguity by rewriting the grammar is a nonsense
    - Even if the rewrite was successful, the semantic analysis would be almost impossible
- The ambiguity is solved with the help of semantic information
  - At the input to parsing, the compiler must distinguish between identifiers of types and identifiers of everything else
    - The language reference manual explicitly states this
    - This is also the reason for the typename keyword inside C++ templates
  - It is implemented by looking into symbol tables between lexical analysis and parsing
  - In our case, you have to implement it inside the calexer.lex file:

```
ctx->is_typedef(yytext)
```

    - will tell you whether yytext was declared as a type identifier (i.e. by typedef)
    - if true, call **make_TYPEIDF** instead of **make_IDF**
    - Testing: Our compiler has a built-in library containing the type identifier **FILE**
  - Note: Interacting with symbol tables in lexer is potentially dangerous:

```
typedef struct { /*...*/ } x; x(*y);
```

    - Are you sure that the symbol tables already contain the first declaration when the second x is lexically analyzed? The parser has a look-ahead! More on that in Assignment 3.
  - In the grammar, use **TYPEIDF** where identifier of a type is expected

- At the input to parsing, the compiler must distinguish between identifiers of types and identifiers of everything else

```
ctx->is_typedef(yytext)
```

  - if true, call make_TYPEIDF instead of make_IDF
- In the grammar, use **TYPEIDF** where identifier of a type is expected
  - This is actually only in the **typedef-name** nonterminal
- But beware, there are contexts which do not reference plain identifiers:
  - struct/union/enum *tag names* as in

```
struct x * y;
```

  - *member names* as in

```
y->x;
```

  - newly declared identifiers as in

```
int x;
```

- In all these cases, is_typedef provides irrelevant information
- The grammar must allow both IDF and TYPEIDF in these contexts, like:

```
postfix-expression:
    postfix-expression ARROW identifier
    /*...*/
    ;
direct-declarator:
    identifier
    /*...*/
    ;
identifier: IDF | TYPEIDF ;
```

- The only context that allows **IDF** but not **TYPEIDF** is inside expressions
  - except the **type-name** in **sizeof**
  - note: Cecko does not support type casts
- Problem:

```
primary-expression:
    identifier
    | constant
    /*...*/
    ;
constant:
    enumeration-constant
    /*...*/
    ;
enumeration-constant:
    identifier
    ;
```

- This must be resolved similarly to the rewrite shown in Example 2
  - Separate **constant_identifier** from **constant_not_identifier**
  - In **primary-expression**, merge **identifier** with **constant_identifier**
    - Hint: make a comment to remember the merge - it will be useful in semantic analysis
- The merged identifier referenced here shall allow only **IDF**, not **TYPEIDF**