Processamento de Linguagens e Compiladores

Gerador de Processadores de Linguagens Yacc

Relatório de Desenvolvimento

Grupo 14

André Sá (76361) João Bastos (47419) Pedro Sá (78164)

14 de Janeiro de 2019

Resumo

Este relatório debruça-se sobre o desenvolvimento de uma linguagem imperativa que com o auxilio do Yacc[1] gera código Assembly, sendo esse código executável em uma Máquina Virtual(VM)[2] fornecida pelo professor desta Unidade Curricular.

[1] Yacc - Gerador de Processadores de Linguagens

[2] Máquina Virtual que mostra o estado das várias stack's ao longo da execução do código gerado pelo Yacc em Assembly

Conteúdo

Capítulo 1

Introdução

Este relaório debruça-se sobre o desenvolvimento de uma linguagem imperativa que com o auxilio do Yacc[1] gera código Assenbly, sendo esse código executável em uma Máquina Virtual(VM)[2] fornecida pelo professor desta Unidade Curricular.

1.1 Enquadramento

No âmbito da Unidade Curricular Processamento de Linguagens e Compiladores, o projecto proposto consiste na criação de uma Linguagem de Programação Imperativa que permita declarar variáveis atómicas dos tipos Inteiro, Real e Booleano e fazer as operações de atribuição de expressões a variáveis declaradas, leitura (de inteiros ou reais), escrita (de inteiros, reais, booleanos ou strings), condições e ciclos repetir....até e gerar código Assembly para a VM fornecida pelo professor.

1.2 Estrutura do Relatório

Neste relatório começamos com a Introdução (Capítulo 1) onde, de uma forma breve descrevemos o enquadramento do documento e a estrutura do mesmo.
No capítulo 2 analisamos o problema que nos foi proposto de forma mais detalhada.
No capítulo 3 explicamos a linguagem imperativa desenvolvida.
No capítulo 4 está representada a gramática da linguagem imperativa desenvolv
O capítulo 5 descreverá a conpecpção do analisador sintático.
Por último o documento tem o apêndice onde foi introduzido o código fonte do ficheiro Yacc.

[1] Yacc - Gerador de Processadores de Linguagens
[2] Maquina Virtual que mostra o estado das várias stack's ao longo da execução do código gerado pelo Yacc em Assembly

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

Este projecto tem como objectivo o aprofundamento de conhecimento Yacc (um gerador de processadores de linguagens). Para tal, foi-nos proposto construir uma linguagem imperativa e gerar um compilador que transforma a sintaxe da linguagem para "Assembly"reconhecido pela máquina virtual fornecida pelo professor da Unidade Curricular.

2.2 Especificação dos Requisitos

A linguagem a criar deverá permitir declarar Variáveis atómicas dos tipos Inteiro, Real e Booleano e fazer Operações de Atribuição de Expressões a Variáveis declaradas, Leitura (de Inteiros ou Reais), Escrita (de Inteiros, Reais, Booleanos ou Strings), condições e ciclos repetir .. até. Após definir a linguagem deveremos a partir de um código fonte com essa linguagem transformar em código Assembly da Máquina Virtual fornecida para o efeito.

Capítulo 3

Linguagem Imperativa

Neste capítulo explicaremos a linguagem imperativa definida e a sua sintaxe.

3.1 Variáveis

As variáveis terão de começar obrigatoriamente por uma letra ou underscore, podendo ser seguida de números, letras ou underscores. A título de exemplo, as variáveis *_aluno*, *Aluno22*, *aluno* e *aluNo* são válidas.

3.2 Tipos

Apresentamos os tipos aceites na linguagem criada.

Tipo	Descrição	Exemplos
Int	Inteiro	1 -22 35
Float	Reais	1.333 3.0E10
Bool	Booleano	True False

3.3 Operadores

Os operadores dividem-se em 3 tipos: aritméticos, relacionais e lógicos. De seguida apresentamos os operadores utilizados e a correspondente descrição.

ARITMÉTICOS	Operador	Descrição
	+	Soma
	-	Subtração
	*	Multiplicação
	/	Divisão
	%	Resto da divisão inteira

RELACIONAIS	Operador	Descrição
	<	Menor
	>	Maior
	=	Igual
	<=	Menor ou igual
	>=	Maior ou igual
	!=	Diferente

LÓGICOS

Operador	Descrição
&	E lógico
|	OU lógico
~	Negação

3.4 Declaração de Variáveis

A declaração de variáveis tem a seguinte sintaxe:

(: tipo var)

Ao declarar, as variáveis são iniciadas com o valor pré-definido do tipo em questão:

Tipo	Default
Int	0
Float	0.0
Bool	False

No entanto também é possível definir no momento um *default* diferente iniciando-a com o valor, do tipo da variável, definido. Nestes casos utilizamos a seguinte sintaxe:

(: tipo var default)

Só é permitido declarar variáveis (*var*) do *tipo* Int, Float e Bool.

3.5 Atribuições

Na atribuição a variáveis (*var*) temos a sintaxe seguinte:

(= var exp)

Em que *exp* pode ser um valor Int, Float, Bool, uma variável ou uma operação aritmética.

Podemos também incrementar numa unidade o valor da variável (*var*), que apenas é válido para variáveis do tipo Int e Float:

(++ var)

3.6 Condição If

As condições *if then else* terão a seguinte sintaxe:

(if exp (code_block) [(else_code_block)])

Em que *exp* terá expressões com valor lógico que corresponderá à condição necessária para executar a porção de código *code_block*, caso contrário, se definido, será executada a porção de código *else_code_block*.

3.7 Ciclos

Os ciclos executam uma determinada porção de código *code_block* até a condição *exp* ser verdadeira. Nesse sentido, *exp* será uma expressão com valor lógico. Cada ciclo terá então a seguinte sintaxe:

(**until** exp (code_block))

3.8 Leitura

A leitura será feita do stdin e irá guardar na variável *var* que será do tipo Int ou Float.
A sintaxe será:

(**read** var)

3.9 Escrita

A escrita será feita após receber um Int, Float, Bool ou String seguindo a seguinte sintaxe:

(**write** exp)

3.10 Impressão

A impressão utiliza a seguinte sintaxe imprimindo o argumento que lhe é passado:

(**print** texto)

Capítulo 4

Gramática

Seguindo a definição da linguagem descrita no Capítulo 3 criamos a seguinte gramática que será a base do nosso projecto. A partir dela, posteriormente, aplicaremos as respectivas ações semânticas como veremos a seguir.

```
programa        : code_block
                ;
code_block      : statements
                | statements code_block
                ;
statements      : '(' statement ')'
                ;
statement       : ':' TYPE VAR DEFAULT
                | INC VAR
                | '=' VAR expression
                | WRITE writable
                | PRINT writable
                | READ VAR
                | IF expression2 '(' code_block ')' else_clause
                | UNTIL expression2 '(' code_block ')'
                ;
writable        : expression
                | STR
                ;
DEFAULT         :
                | expression
                ;
else_clause     :
                | '(' code_block ')'
                ;
expression      : VALUE
                | expression2
                | '(' expression_list ')'
                ;
VALUE           : INT_VALUE
                | FLOAT_VALUE
                ;
arith_op        : '+'
                | '*'
                | '-'
                | '/'
                | '%'
                ;
```

```
expression_list  : arith_op expression expression
                 ;
expression2      : VAR
                 | BOOL_VALUE
                 | '(' expression2_list ')'
                 ;
num_op           : '<'
                 | '='
                 | '>'
                 | GEQ
                 | LEQ
                 | NEQ
                 ;
log_op           : '&'
                 | '|'
                 ;
expression2_list : '~' expression2
                 | num_op expression expression
                 | log_op expression2 expression2
                 ;
```

Capítulo 5

Concepção/desenho da Resolução

Neste capítulo iremos descrever a concepção/desenho do analisador sintático indicando a produção em questão e uma breve explicação da respectiva ação semântica.

programa : code_block

Nesta produção são impressas as declarações de variáveis locais, a instrução START para iniciar o programa, o código do programa em si e, finalmente, a instrução STOP que marca o fim do programa.

code_block : statements

O código do bloco de código vai ser o código de um statement.

code_block : statements code_block

O código do bloco de código vai ser o resultado de concatenar o código de um statement com o código do resto do block.

statements : '(' statement ')'

O código de um statement vai ser o código do conteúdo desse statement.

statement : ':' TYPE VAR DEFAULT

Esta produção declara uma variável do tipo TYPE, um nome do tipo VAR e, opcionalmente, um valor DEFAULT. Este default, se existir, tem de ser compatível com o tipo da variável a ser declarada. Em primeiro lugar, verifica-se que o tipo do valor DEFAULT é compatível com o da variável a declarar. Este não pode ser TYPE_ERROR, e, se não for TYPE_DEFAULT (i.e., não for dado nenhum valor default), verificam-se os tipos. De seguida tenta-se adicionar a variável a uma tabela que contém todas as variáveis declaradas até agora. É um erro declarar uma variável duas vezes, tenham ou não tipos diferentes. Caso esta operação tenha sucedido, o código deste statement passa a ser o código gerado pelo DEFAULT e, se existir DEFAULT, acrescentamos a instrução de STORE para guardar o valor na variável.

statement : INC VAR

Esta produção é equivalente a (= VAR (+ VAR 1)).

statement : '=' VAR expression

Nesta produção verificamos se VAR foi declarada e se os tipos da variável e da expressão são compatíveis. Se forem, o código resultante será o de apender a instrução STORE ao código da expressão.

statement : WRITE writable

Nesta produção geramos código para escrever o valor de um writable. Writable pode ser tanto uma expressão como uma string literal.

statement : PRINT writable

Esta produção faz o mesmo que o WRITE, mas escreve também um caracter de mudanca de linha.

statement : READ VAR

Esta produção de um valor do stdin e guarda-o na variável VAR. É um erro se a variável não tiver sido declarada, e não for dos tipos Int ou Float. As instruções geradas são, por ordem, READ, ATO[IF], STORE.

statement : IF expression2 '(' code_block ')' else_clause

Nesta produção geramos um identificador único para as labels necessárias (ELSE e ENDIF). O código final será: ¡codigo gerado pela expressão de condição¿ JZ ELSE ¡codigo gerado pelo bloco then¿ JUMP ENDIF ELSE: ¡codigo gerado pelo block else¿ ENDIF:

statement : UNTIL expression2 '(' code_block ')'

Nesta produção geramos um identificador único para a label necessária (UNTIL). O código final será: UNTIL: ¡codigo gerado pelo bloco do ciclo¿ ¡codigo gerado pela expressao de condicao¿ JZ UNTIL

writable : expression

O código de algo que pode ser escrito e o código da expressão.

writable : STR

Nesta produção é simplemente gerada a instrução PUSHS com a string devolvida pelo Flex.

DEFAULT :

Nesta produção não é gerado nenhum código. O tipo desta produção é TYPE_DEFAULT, para distinguir os casos em que existe valor default dos casos em que não existe.

DEFAULT : expression

O valor de default vai ser o computado na expressão.

else_clause :

Nesta produção não é gerado nenhum código.

else_clause : '(' code_block ')'

Nesta produção o código dá else clause e o código gerado para o bloco de código.

expression : VALUE

Nesta produção gera-se a instrução de PUSH com o valor lido pelo Flex.

expression : expression2

O código resultante é o código gerado pela expression2.

expression : '(' expression_list ')'

O código resultante é o código gerado pela expression_list.

VALUE : INT_VALUE

O tipo desta produção é o tipo Int.

VALUE :FLOAT_VALUE

O tipo desta produção é o tipo Float.

expression_list : arith_op expression expression

O código resultante desta produção e o de concatenar o código da primeira expressão com o da segunda e depois a instrução correspondente a operação. Esta produção corresponde a operações aritmética sobre valores numéricos. É um erro se os tipos não forem compatíveis.

expression2 : VAR

Nesta produção gera-se a instrução de carregar o endereço da variável e a de carregar o valor nesse endereço.

expression2 : BOOL_VALUE

Nesta produção gera-se a instrução de por na stack o valor booleano lido pelo Flex.

exoression2 : '(' expression2_list ')'

O código resultante e o código gerado pela expression2_list.

expression2_list : ' ' expression2

O código resultante desta produção e o de concatenar a instrução NOT ao código da expression2.

expression2_list : num_op expression expression

O código resultante desta produção e o de concatenar o código da primeira expressão com o da segunda e a instrução correspondente a operação. Esta produção corresponde à comparação de valores numéricos. É um erro se os tipos das duas expressões não forem compatíveis.

expression2_list : log_op expression2 expression2

O código resultante desta produção e o de concatenar o código da primeira expressão com o da segunda e a instrução correspondente a operação. Esta produção corresponde a operações lógicas sobre valores booleanos.

Capítulo 6

Testes

6.1 Testes realizados e Resultados

Para efeitos de teste preparamos vários ficheiros que cada um contém desde declarações, a atribuições, condições, ciclos, etc...
Estes testes serão organizados em tabelas abaixo apresentadas com a primeira coluna a representar o código-fonte de acordo com a nossa linguagem imperativa criada e a segunda coluna com o código em "assembly"em conformidade com a máquina virtual fornecioda.

6.1.1 Declarações e Atribuições

Input	Output
(: Int x) (: Int y 2) (= x (+ x y)) (: Bool a) (: Bool b True) (: Float z 10.0) (: Float t)	PUSHI 0 PUSHI 0 PUSHI 1 PUSHI 1 PUSHF 0.0 PUSHF 0.0 START PUSHI 2 STOREG 1 PUSHGP LOAD 0 PUSHGP LOAD 1 ADD STOREG 0 PUSHI 1 STOREG 3 PUSHF 10.0 STOREG 4 STOP

6.1.2 Operadores Aritméticos

Input	Output
(print (+ 1 2)) (print (* 1 2)) (print (/ 1 2)) (print (- 1 2)) (print (% 1 2)) (print (+ 1.0 2.0)) (print (* 1.0 2.0)) (print (/ 1.0 2.0)) (print (- 1.0 2.0))	START PUSHI 1 PUSHI 2 ADD WRITEI PUSHS "\n" WRITES PUSHI 1 PUSHI 2 MUL WRITEI PUSHS "\n" WRITES PUSHI 1 PUSHI 2 DIV WRITEI PUSHS "\n" WRITES PUSHI 1 PUSHI 2 SUB WRITEI PUSHS "\n" WRITES PUSHI 1 PUSHI 2 MOD WRITEI PUSHS "\n" WRITES PUSHF 1.0 PUSHF 2.0 FADD WRITEF PUSHS "\n" WRITES PUSHF 1.0 PUSHF 2.0 FMUL WRITEF PUSHS "\n" WRITES PUSHF 1.0 PUSHF 2.0 FDIV WRITEF PUSHS "\n" WRITES PUSHF 1.0 PUSHF 2.0 FSUB WRITEF PUSHS "\n" WRITES STOP

6.1.3 Leitura

Input	Output
(: Float x) (read x) (print x)	PUSHF 0.0 START READ ATOF STOREG 0 PUSHGP LOAD 0 WRITEF PUSHS "\n" WRITES STOP

6.1.4 Condição IF

Input	Output
(if (~ (= 1 2)) ((write "then")) ((write "else"))) (if (!= 1 1) ((write "then")))	START PUSHI 1 PUSHI 2 EQUAL NOT JZ ELSE0 PUSHS "then" WRITES JUMP ENDIF0 ELSE0: PUSHS "else" WRITES ENDIF0: PUSHI 1 PUSHI 1 SUB JZ ELSE1 PUSHS "then" WRITES JUMP ENDIF1 ELSE1: ENDIF1: STOP

6.1.5 Ciclo Repetir..Até

Input	Output
(until (= 1 2) ((write "wut")))	START UNTIL0: PUSHS "wut" WRITES PUSHI 1 PUSHI 2 EQUAL JZ UNTIL0 STOP

Capítulo 7

Conclusão

Este projecto mostrou-nos que tendo já desenvolvido a gramática de uma linguagem imperativa, podemos criar um compilador através do Yacc que usando a estrutura 'Produção-Ação', isto é, para cada instrução do código fonte que satisfaça alguma produção da linguagem imperativa tem como ação a criação do código "assembly", o que consequentemente, deu-nos uma visão mais detalhada sobre como um compilador funciona. Este projeto está finalizado para o que nos foi proposto.

Apêndice A

Código do Programa

Lista-se a seguir o código Yacc do programa que foi desenvolvido.

```
%{
#include <stdbool.h>
#include <stdio.h>

#include "env.h"
#include "gen.h"
#include "str.h"
#include "rope.h"

#include "lex.yy.h"

int yyerror (const char *s);

/**
 * Verifica se uma condicao e verdadeira e, caso nao seja,
 * imprime uma mensagem de erro e aborta o programa
 */
#define assert(cond, ...)                       \
    if (!(cond)) do {                           \
        fprintf(stderr, "ERROR: " __VA_ARGS__); \
        return 1;                               \
    } while (0)

/** Se activado, mostra as producoes por onde passa */
#ifdef TRACE
#define trace(...) ((void) fprintf(stderr, "TRACE: %s:%d\n", __FILE__, __LINE__))
#else
#define trace(...) ((void) 0)
#endif /* TRACE */

#define type_valid(t)       ((t) > TYPE_ERROR && (t) < TYPE_DEFAULT)
#define type_compat(t1, t2) ((t1) == (t2))

/**
 * Concatena dois blocos de codigo, @a self e @a other, e imprime
 * uma mensagem em caso de erro
 */
```

```
#define cbapp(self, other) do {                \
        assert(rope_append(&(self), &(other)), \
            "appending code blocks: %s:%d\n",  \
            __FILE__, __LINE__);               \
        (other) = rope_free(other);            \
    } while (0)

/**
 * Gera wrappers para funcoes geradoras de codigo que imprimem
 * uma mensagem em caso de erro
 */
#define gen_(f, ...)               \
    assert((gen_##f)(__VA_ARGS__), \
        #f "(): %s:%d\n",          \
        __FILE__, __LINE__)

#define gen_jump(  c, l, n)    gen_(jump,   (c), (l), (n))
#define gen_jz(    c, l, n)    gen_(jz,     (c), (l), (n))
#define gen_nlbl(  c, l, n)    gen_(nlbl,   (c), (l), (n))
#define gen_load(  c, i)       gen_(load,   (c), (i))
#define gen_op(    c, o, t)    gen_(op,     (c), (o), (t))
#define gen_push(  c, t, a, b) gen_(push,   (c), (t), (a), (b))
#define gen_pushgp(c)          gen_(pushgp, (c))
#define gen_storeg(c, i)       gen_(storeg, (c), (i))

static struct rope _var_decs = {0};
static struct rope * const var_decs = &_var_decs;

%}

%union {
    struct expr {
        enum type   type;
        struct rope code;
    }           valExpr;

    bool        valBool;
    char *      valString;
    enum type   valType;
    float       valFloat;
    int         valInt;
    struct rope valCode;
}

%token INC
%token GEQ
%token IF
%token LEQ
%token NEQ
%token PRINT
%token READ
%token UNTIL
%token WRITE
```

```
%token <valBool>  BOOL_VALUE
%token <valFloat> FLOAT_VALUE
%token <valInt>   INT_VALUE
%token <valString>STR
%token <valString>VAR
%token <valType>  TYPE

%type <valCode>code_block
%type <valCode>else_clause
%type <valCode>statement
%type <valCode>statements

%type <valExpr>DEFAULT
%type <valExpr>expression
%type <valExpr>expression2
%type <valExpr>expression2_list
%type <valExpr>expression_list
%type <valExpr>writable

%type <valInt>arith_op
%type <valInt>log_op
%type <valInt>num_op

%type <valType>VALUE

%%

programa : code_block { trace();
             rope_fprint(var_decs, yyout);
             fputs("START\n", yyout);
             rope_fprint(&$1, yyout);
             fputs("STOP\n", yyout);
             $1 = rope_free($1);
         }
         ;

code_block : statements            { trace(); $$ = $1; }
           | statements code_block { trace();
               $$ = $1;
               cbapp($$, $2);
           }
           ;

statements : '(' statement ')' { trace(); $$ = $2; }
           ;

statement : ':' TYPE VAR DEFAULT { trace();
              assert($4.type != TYPE_ERROR
                  && (!type_valid($4.type) || type_compat($2, $4.type)),
                  "%s:%s but default value is of type %s\n",
                  $3, type2str($2), type2str($4.type));

              struct var var = { .id = $3, .type = $2, };
              assert(env_new_var(env, var), "creating variable `%s`\n", $3);
```

```
    $$ = $4.code;
    if (!rope_is_empty(&$$))
        gen_storeg(&$$, env_var_gp_idx(env, $3));

    gen_push(var_decs, $2,
        (($2 == TYPE_FLOAT) ? "0.0" : "0"),
        yylval.valBool);
}
| INC VAR { trace();
    $$ = (struct rope) {0};
    struct var * v = env_var(env, $2);
    enum type t = env_typeof(env, $2);
    unsigned gidx = env_var_gp_idx(env, $2);

    assert(v != NULL, "Variable not found: `%s`\n", $2);

    assert(t == TYPE_INT || t == TYPE_FLOAT,
        "type: Expected Int or Float but got %s\n",
        type2str(t));

    const char * arg = (t == TYPE_INT) ? "1" : "1.0";

    gen_pushgp(&$$);
    gen_load(&$$, gidx);
    gen_push(&$$, t, arg, false);
    gen_op(&$$, '+', t);
    gen_storeg(&$$, gidx);
}
| '=' VAR expression { trace();
    struct var * v = env_var(env, $2);
    assert(v != NULL, "Variable not found: `%s`\n", $2);
    $$ = $3.code;
    gen_storeg(&$$, env_var_gp_idx(env, $2));
}
| WRITE writable { trace();
    $$ = $2.code;
    gen_op(&$$, WRITE, $2.type);
}
| PRINT writable { trace();
    $$ = $2.code;
    gen_op(&$$, WRITE, $2.type);
    gen_push(&$$, TYPE_STRING, "\"\\n\"", false);
    gen_op(&$$, WRITE, TYPE_STRING);
}
| READ VAR { trace();
    $$ = (struct rope) {0};
    struct var * v = env_var(env, $2);
    assert(v != NULL, "Variable not found: `%s`\n", $2);

    assert(v->type == TYPE_INT || v->type == TYPE_FLOAT,
        "`read`: Expected Int or Float but got %s\n",
        type2str(v->type));
```

```
            gen_read(&$$);
            gen_aton(&$$, v->type);
            gen_storeg(&$$, env_var_gp_idx(env, $2));
          }
          | IF expression2 '(' code_block ')' else_clause { trace();
            unsigned num = gen_ifno();

            $$ = $2.code;
            gen_jz(&$$, "ELSE", num);    /* jump to else label? */
            cbapp($$, $4);               /* then block */
            gen_jump(&$$, "ENDIF", num); /* jump to endif label */
            gen_nlbl(&$$, "ELSE", num);  /* else label */
            cbapp($$, $6);               /* else block (possibly empty) */
            gen_nlbl(&$$, "ENDIF", num); /* endif label */
          }
          | UNTIL expression2 '(' code_block ')' { trace();
            $$ = (struct rope) {0};
            unsigned num = gen_untilno();

            gen_nlbl(&$$, "UNTIL", num); /* until label */
            cbapp($$, $4);               /* loop body block */
            cbapp($$, $2.code);          /* condition */
            gen_jz(&$$, "UNTIL", num);   /* jump to until label? */
          }
          ;

writable : expression { trace(); $$ = $1; }
         | STR        { trace();
            $$ = (struct expr) {0};
            $$.type = TYPE_STRING;
            gen_push(&$$.code, TYPE_STRING, yytext, false);
          }
          ;

DEFAULT : { trace();
            $$ = (struct expr) {0};
            $$.type = TYPE_DEFAULT;
        }
        | expression { trace(); $$ = $1; }
        ;

else_clause :                    { trace(); $$ = (struct rope) {0}; }
            | '(' code_block ')' { trace(); $$ = $2; }
            ;

expression : VALUE { trace();
               $$ = (struct expr) {0};
               $$.type = $1;
               gen_push(&$$.code, $1, yytext, false);
           }
           | expression2             { trace(); $$ = $1; }
           | '(' expression_list ')' { trace(); $$ = $2; }
           ;
```

```
VALUE : INT_VALUE   { trace(); $$ = TYPE_INT;   }
      | FLOAT_VALUE { trace(); $$ = TYPE_FLOAT; }
      ;

arith_op : '+' { trace(); $$ = '+'; }
         | '*' { trace(); $$ = '*'; }
         | '-' { trace(); $$ = '-'; }
         | '/' { trace(); $$ = '/'; }
         | '%' { trace(); $$ = '%'; }
         ;

expression_list : arith_op expression expression { trace();
                    $$ = (struct expr) {0};
                    enum type t1 = $2.type;
                    enum type t2 = $3.type;
                    assert(type_valid(t1) && type_valid(t2) && type_compat(t1, t2),
                        "`%c`: Types don't match: op1:%s and op2:%s\n",
                        $1, type2str(t1), type2str(t2));
                    $$.type = t1;

                    assert($$.type != TYPE_FLOAT || $1 != '%',
                        "`%%`: Expected Int, got %s\n",
                        type2str($$.type));

                    $$.code = $2.code;
                    cbapp($$.code, $3.code);
                    gen_op(&$$.code, $1, t1);
                }
                ;

expression2 : VAR { trace();
                $$ = (struct expr) {0};
                $$.type = env_typeof(env, $1);
                assert(type_valid($$.type), "Variable not found: `%s`\n", $1);

                gen_pushgp(&$$.code);
                gen_load(&$$.code, env_var_gp_idx(env, $1));
            }
            | BOOL_VALUE { trace();
                $$ = (struct expr) {0};
                $$.type = TYPE_BOOL;
                gen_push(&$$.code, TYPE_BOOL, NULL, yylval.valBool);
            }
            | '(' expression2_list ')' { trace(); $$ = $2; }
            ;

num_op : '<' { trace(); $$ = '<'; }
       | '=' { trace(); $$ = '='; }
       | '>' { trace(); $$ = '>'; }
       | GEQ { trace(); $$ = GEQ; }
       | LEQ { trace(); $$ = LEQ; }
       | NEQ { trace(); $$ = NEQ; }
       ;
```

```
log_op : '&' { trace(); $$ = '&'; }
       | '|' { trace(); $$ = '|'; }
       ;

expression2_list : '~' expression2 { trace();
                    $$ = (struct expr) {0};
                    enum type t = $2.type;
                    assert(t == TYPE_BOOL, "`~`: Expected Bool, got %s\n", type2str(t));
                    $$.type = t;

                    $$.code = $2.code;

                    gen_op(&$$.code, '~', $2.type);
                }
                | num_op expression expression { trace();
                    $$ = (struct expr) {0};
                    enum type t1 = $2.type;
                    enum type t2 = $3.type;
                    assert(type_valid(t1) && type_valid(t2) && type_compat(t1, t2),
                        "`%c`: Types don't match: op1:%s and op2:%s\n",
                        $1, type2str(t1), type2str(t2));
                    $$.type = TYPE_BOOL;

                    $$.code = $2.code;
                    cbapp($$.code, $3.code);
                    gen_op(&$$.code, $1, t1);
                }
                | log_op expression2 expression2 { trace();
                    enum type t1 = $2.type;
                    enum type t2 = $3.type;
                    assert(t1 == TYPE_BOOL && t2 == TYPE_BOOL,
                        "`%c`: Expected Bool, got op1:%s and op2:%s\n",
                        $1, type2str(t1), type2str(t2));

                    $$ = $2;

                    cbapp($$.code, $3.code);
                    gen_op(&$$.code, $1, t1);
                }
                ;

%%

int yyerror (const char *s)
{
  return fprintf(stderr, "ERRO: '%s'\n", s);
}
```