

CCF 441: Compiladores



**Projeto da Linguagem:
Tao**

Apresentação do Grupo



Germano - 3873



Henrique - 3051



Otávio - 3890



Pedro - 3877



Vinícius - 3495

Universidade Federal de Viçosa - Campus Florestal
Ciência da Computação

Sumário

1. Introdução



Introdução do trabalho e temática da linguagem criada.

4. Análise Sintática



Apresentação e demonstração da fase de análise sintática.

2. Características Gerais da Linguagem



Apresentação das características da linguagem: tipos de dados, comandos, expressões, dentre outros.

5. Análise Semântica



Apresentação e demonstração da fase de análise semântica.

3. Análise Léxica



Apresentação e demonstração da fase de análise léxica.

6. Resultados



Demonstração da execução do front-end do compilador.

1.

Introdução

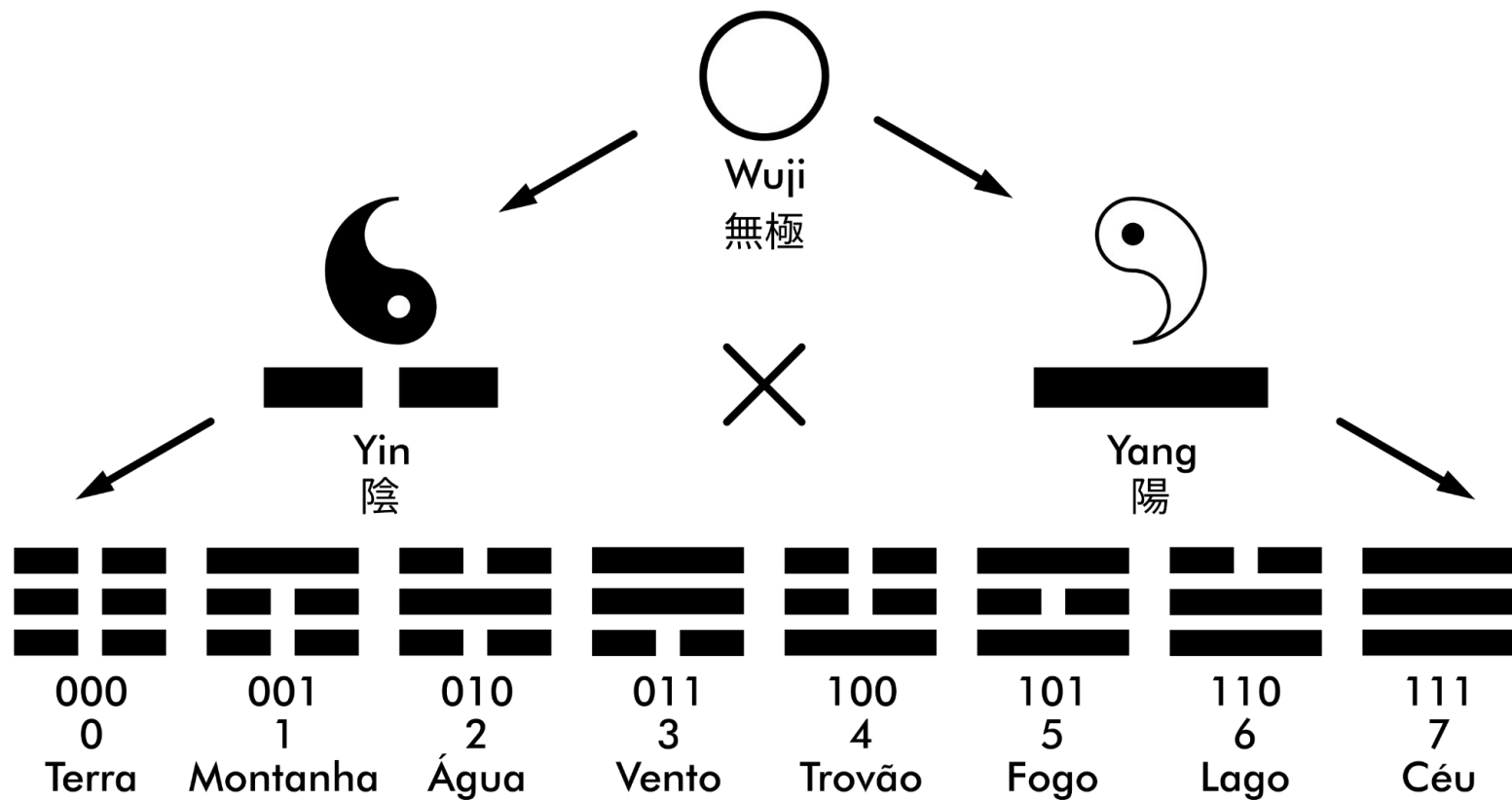
Temática

- Filosofia chinesa antiga do taoísmo;
- Bagua e Yi Jing (I Ching).



道 八卦 易經







000

0

Terra

001

1

Montanha

010

2

Água

011

3

Vento

100

4

Trovão

101

5

Fogo

110

6

Lago

111

7

Céu

:::

::|

:|:

:||

|::

|:|

||:

|||

2.

**Características
Gerais da
Linguagem**



Tipos de Dados

- Int – valores inteiros de 32 bits com sinal;
- Real – valores de ponto flutuante de 64 bits;
- Char – valores de caractere de 8 bits;
- Bool – valores booleanos;
- @Type – ponteiro para Type, sendo Type qualquer outro tipo;
- @Any – equivale ao void * de C.



Tipos de Dados

- Tipos Compostos: Podem ser criados com produto cartesiano e união disjunta.

```
'' três construtores, nenhum campo
::: Color = Red, Green, Blue

'' um construtor com dois campos
::: Person = Person(name: @Char, age: Int)

'' três construtores, com campos variados
::: Tree = Node(x: Int, lt: @Tree, rt: @Tree), Leaf(x: Int), Empty
```



Literais

- Valores inteiros : sequências de dígitos decimais ou hexadecimais, prefixados com 0x.
- Valores reais: dígitos decimais separados com um ponto, seguidos opcionalmente por um expoente em notação científica.
- Caracteres são expressados por dois apóstrofos com um caractere ou uma sequência de escape dentro. Sequências de escape começam com contrabarra (\) e são seguidas ou de um valor hexadecimal arbitrário entre 00 e ff ou são sequências como '\n' de nova linha ou '\t' de tabulação.



Literais

- Apesar de não haver strings como tipos primitivos, elas são vetores de caracteres, como em C, que usam ponteiros, e podem ser expressas em literais.
- Literais booleanas não são exatamente literais, mas sim um tipo composto. O tipo Bool é definido como `::: Bool = False, True`.
- Null retorna sempre um ponteiro nulo de tipo @Any. Esse é o único construtor de tipo que retorna um ponteiro.



Expressões

- Toda expressão pode ser aberta numa sequência de expressões e comandos, usando chaves ({}). Expressões também podem ser aninhadas com parênteses.
- O programador pode definir seus operadores, mas a linguagem já possui alguns predefinidos.
- Quanto maior o valor da precedência, maior a prioridade.

Precedência	Operadores	Associatividade
8	**	Direita
7	* / & ^	Esquerda
6	+ -	Esquerda
5	<< >>	Esquerda
5	<?	Esquerda
5	?>	Direita
4	== != < <= >= >	Não associativo
3	&&	Esquerda
2		Esquerda
1	<<?	Esquerda
1	?>>	Direita
1	?	Não associativo



Expressões

- Algumas expressões disponíveis são: condicional, casamento de tipo, atribuição (=), acesso a campo (.comid), acesso a posição ([i]), acesso direto (@ sufixado), referência (@ prefixado), dentre outros.
- As expressões de casamento de tipo e a condicional seguem a mesma sintaxe de quando são usadas como comandos, com a diferença de que o "else" do condicional é obrigatório.

```
'' expressão de atribuição, e diferentes níveis de operações infixadas
x = y = 3 | 2 ** 3 >> 0xA + 0xf - 23;
'' equivalente à seguinte expressão:
x = (y = ((3 | (2 ** 3)) >> ((0xA + 0xf) - 23)));

'' negações
cond = !(-x < ~y);

'' manipulação de ponteiros
y = (@x + 10)@;

'' acesso a posição, seguido de acesso a campo
people[10].name = "Alfred Aho";

'' alocação de memória
yin p: @Person = ::| Person;
'' alocação de 10 posições
p = ::| Person ::|:| 10;
'' construção de tipo
p@ = Person("Fulano", 42);
'' alocação + construção
p = ::||:| Person("Fulano", 42);
'' alocação + construção de 20 posições
p = ::||:| Person("", 0) ::|:| 20;

'' chamada de função
x = a + random() * (b - a);
```





Comandos

- Exportação de módulo;
- Importação; definição de variável;
- Definição de função;
- Definição de operador;
- Definição de procedimento;
- Definição de tipo;
- Declaração de tipo;
- Condicional;
- Casamento de tipos;
- Repetição (while e repeat);
- Liberação de memória



Comandos

- Separados por ponto-e-vírgula;
- Toda expressão pode ser considerada um comando;
 - Desnecessário distinguir chamada de procedimentos e de funções;



Comandos

- Exportação de módulo;
- Aparece no topo do arquivo, não se repete;
- Trigramma `|||` denota a exportação;
- Hexagrama `|||::` lista procedimentos, funções, variáveis ou tipos que serão exportados.

```
' ' arquivo1.tao
||| This.Module.Name |||::: proc1, func2, var4;

' ' arquivo2.tao
||| Another.Module |||::: var8, proc16, Type32;
```



Comandos

- Importação de módulos;
- Trígrama ||: denota a importação;
- Hexagrama ||::|| especifica o que se deseja importar de dentro de um módulo;
- Hexagrama ||::||: renomeia o módulo importado;

```
'' equivalente em Python:  
'' from src.patricia import *  
||: Src.Patricia;  
  
'' from math import sin, cos, tan  
||: Math ||::|| sin, cos, tan;  
  
'' import sqlite3 as sql  
||: SQLite3 ||::||: SQL;
```



Comandos

- Definição e Declaração;
- Hexagrama ::::: declara tipos;
- Trigramma ::: declara novos tipos de dados;
- yin define variáveis;
- yang define funções;
- wuji define subprogramas;
 - Igual à função, não retorna valor e não pode ser usado como expressão;
- Hexagrama ||||: equivale ao return;

```
::::: String = @Char;  
  
::: Person = Person(name: @Char, age: Int);  
  
yin pi: Real = 3.1415926535;  
  
yang initial(p: Person): Char = p.name[0];  
  
' ' ::::|| lista parâmetros de tipo para  
' ' a função/procedimento/operador que vier em seguida  
::::|| T  
yang len(list: @List(T)): Int = { ' ' expressão ' ' }  
  
wuji main() { }
```



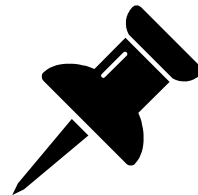
Comandos

- Definição de operadores;
- Semelhante à definição de funções;
- Necessário indicar precedência do operador;
- Os próprios caracteres indicam sua precedência e associatividade;

```
'' operador associativo à esquerda, precedência 5  
'' prefixo <  
yang <#(x: Real, exp: Real): Real = { ''' expressão ''' }  
  
'' operador associativo à esquerda, precedência 1  
'' sufixo >>  
yang +>>(item: Int, head: @List): @List = { ''' expressão ''' }  
  
'' operador não associativo, precedência 1  
'' não possui padrão  
yang !=(s0: @Char, s1: @Char): Bool = { ''' expressão ''' }
```

Comandos

- Comando condicional;
- Trigrama `:``:` representa *if*;
- Hexagrama `:``:``:` aparece depois da condição;
- Hexagrama `:``:``:``:` representa *else*, e é *opcional*;
- Hexagrama `:``:``:``:` representa *elif*;



```
:: cond ::: {  
    '' executa se cond == True  
}  
  
:: cond ::: {  
    '' executa se cond == True  
}::: {  
    '' executa se cond == False  
}  
  
:: cond ::: {  
    '' executa se cond == True  
}::: cond2 ::: {  
    '' executa se cond == False e cond2 == True  
}::: {  
    '' executa caso contrário  
}
```



Comandos

- Comando semelhante a *switch-case*;
- Trigramma |:| seguido da expressão utilizada na análise feita;
- Hexagrama |:||| realiza o casamento;
 - se verdadeiro, executa o código depois do hexagrama |:|:| ;
- Hexagrama |:|:: representa o caso padrão;

```
::: Tree(K) = Node(x: K, lt: @Tree(K), rt: @Tree(K)), Leaf(x: K)

yin tree: @Tree(Int) = '' expressão
|:| @tree
|:|||| Node(x, l, r) |:|:|: { ''executa se for Node'' }
|:|||| Leaf(x) |:|:|: { ''executa se for Leaf'' }
|:|::: { ''default'' }
```




Comandos

- Comando de repetição;
- Trigrama `::` representa o início do comando de repetição;
- Seguido de `::::`, executa o comando descrito como *true*;
- Opcional: Hexagrama `:::` que indica bloco a ser executado após cada iteração do laço;
- Hexagrama `::::` define código a ser executado enquanto a condição for *true*;

```
::| cond ::|::| {  
    '' executa enquanto cond == True  
    ::|::|: '' pula para a próxima iteração, comando "continue"  
}  
  
::| cond ::|::|: {''executa depois do laço''} ::|::| {  
    '' executa enquanto cond == True  
}
```



Comandos

- Comando de repetição;
- Hexagrama `:||:` seguido de um bloco de código, seguido do hexagrama `:||:` e da condição de execução desse trecho de código;
- Executado até que a condição se torne verdadeira;
- Condição checada apenas após a execução;
- Em ambos os comandos, é possível usar os hexagramas `:|||:` (*break*) e `:||::` (*continue*) para controlar o fluxo da repetição;

```
:||:| {  
    '' executa até que cond == True  
    :|||: '' sai imediatamente do laço, comando "break"  
} :||::| cond  
  
:||:| {  
    '' executa até que cond == True  
} :||::| cond :|||:: { ''executa depois do laço''' }
```



Comandos

- Trigrama `|::` é utilizado para a liberação de memória;
 - semelhante à função *free()* da linguagem C;

```
yin list: @List = '' expressão  
|:: list;
```


3.

Análise Léxica



Análise Léxica

- YY_USER_ACTION
- Definições Regulares
- Padrões de Lexema
- Código Auxiliar



```
#define YY_USER_ACTION lexer.update_yyloc();

void Lexer::update_yyloc() {
    yyloc.first_line = yyloc.last_line;
    yyloc.first_column = yyloc.last_column;
    for (int i = 0; yytext[i]; i++)
        if (yytext[i] == '\n')
            yyloc.last_line++, yyloc.last_column = 1;
        else
            yyloc.last_column++;
}
```


YY_USER_ACTION

```

ws          [ \n\t\r\v\f]
ws_space    [\n\t\r\v\f]
small       [a-z_]
large       [A-Z]
digit       [0-9]
hexit       [0-9A-Fa-f]
symbol_at   [!#$%&*+ /<=>?^\\|\\\~.:]
symbol      [!@#$%&*+ /<=>?^\\|\\\~.:]
special     [\\(\\),\\[\\]\\{\\}]
symid       {symbol}*{symbol_at}{symbol}*
relop       ==|!=|<|<=|>|=|>
comid       {small}({small}|{large}|{digit})*
proid       {large}({small}|{large}|{digit})*
qualify     ({proid}\\.)+
decimal     {digit}+
exponent    [eE][+-]?{decimal}
real        ({decimal}\\.{decimal}{exponent}?|{decimal}{exponent})
charesc     a|b|e|f|n|r|t|v|\\\\"|'
gap         \\{ws}+\\
trigram     [\\:]{3,3}
hexagram    [\\:]{6,6}

```

Definições Regulares




```
; {return ENDL;}
wuji {return WUJI;}
yin {return YIN;}
yang {return YANG;}
{trigram} {return lexer.trig_token();}
{hexagram} {return lexer.hex_token();}
[@:=\\.\\-] {return *yytext;}
{special} {return *yytext;}
{qualify}?\\*\\* {return lexer.id_token(SYM_ID_R8);}
{qualify}?[*/&^] {return lexer.id_token(SYM_ID_L7);}
{qualify}?[+\\|] {return lexer.id_token(SYM_ID_L6);}
{qualify}\\- {return lexer.id_token(SYM_ID_L6);}
{qualify}?\\(<\\<|\\>\\>) {return lexer.id_token(SYM_ID_L5);}
{qualify}?\\<{symbol}*\\> {lexer.error("invalid operator");}
```

Padrões de Lexema



```
int Lexer::tri_hex_offset() {  
    int offset = 0;  
    for (int i = 0; yytext[i]; ++i)  
        offset = (offset << 1) + (yytext[i]=='|');  
    return offset;  
}
```

Código Auxiliar



```
int Lexer::char_int_val(char c) {
    if ('0' <= c && c <= '9')
        return c - '0';
    else if ('a' <= c && c <= 'z')
        return c - 'a' + 10;
    else if ('A' <= c && c <= 'Z')
        return c - 'A' + 10;
    else
        return 0;
}

int Lexer::int_val(int base, int i0) {
    int v = 0;
    for (int i = i0; i < yyleng; ++i)
        v = v * base + char_int_val(yytext[i]);
    return v;
}
```

Código Auxiliar


4.

Análise Sintática



Análise Sintática

- Gramática
- Conflitos Shift/Reduce
- Precedência de Operadores
- Conflitos Reduce/Reduce
- Tabela de símbolos



```
<program> ::= <module-decl> <top-stmts>

<top-stmts> ::= <top-stmts> ENDL <top-stmt> | <top-stmt>
<top-stmt> ::= <import>
               | <type-def>
               | <type-alias>
               | <callable-def>
               | <call-type-params> <callable-def>
               | <var-def>
               | ""

<stmts> ::= <stmts> ENDL <stmt> | <stmt>
<stmt> ::= <top-stmt>
           | <while>
           | <repeat>
           | <free>
           | <break>
           | <continue>
           | <return>
           | <expr>
```

```

<expr> ::= "{" <stmts> "}"
| <assign>
| <match>
| <if>
| <expr> SYM_ID_L1 <expr> | <expr> QSYM_ID_L1 <expr>
| <expr> SYM_ID_N1 <expr> | <expr> QSYM_ID_N1 <expr>
| <expr> SYM_ID_R1 <expr> | <expr> QSYM_ID_R1 <expr>
| <expr> SYM_ID_L2 <expr> | <expr> QSYM_ID_L2 <expr>
| <expr> SYM_ID_L3 <expr> | <expr> QSYM_ID_L3 <expr>
| <expr> SYM_ID_N4 <expr> | <expr> QSYM_ID_N4 <expr>
| <expr> SYM_ID_R5 <expr> | <expr> QSYM_ID_R5 <expr>
| <expr> SYM_ID_L5 <expr> | <expr> QSYM_ID_L5 <expr>
| <expr> SYM_ID_L6 <expr> | <expr> QSYM_ID_L6 <expr>
| <expr> "-" <expr>
| <expr> SYM_ID_L7 <expr> | <expr> QSYM_ID_L7 <expr>
| <expr> SYM_ID_R8 <expr> | <expr> QSYM_ID_R8 <expr>
| "@" <expr> | "~" <expr> | "!" <expr> | "-" <expr>
| <malloc>
| <build>
| <call>
| <addr>
| <literal>
| "(" <expr> ")"

```

Conflitos

Shift/Reduce

- Preferência pelo Shift
- Por precaução: definir precedência de operadores
- Problema do else-pendente

Precedência	Operadores	Associatividade	Nome do Token
8	**	Direita	SYM_ID_R8
7	* / & ^	Esquerda	SYM_ID_L7
6	+ -	Esquerda	SYM_ID_L6
5	<< >>	Esquerda	SYM_ID_L5
5	<?	Esquerda	SYM_ID_L5
5	?>	Direita	SYM_ID_R5
4	= ≠ < ≤ ≥ >	Não associativo	SYM_ID_N4
3	&&	Esquerda	SYM_ID_L3
2		Esquerda	SYM_ID_L2
1	<<?	Esquerda	SYM_ID_L1
1	?>>	Direita	SYM_ID_R1
1	?	Não associativo	SYM_ID_N1



```
%right '='
%nonassoc SYM_ID_N1 QSYM_ID_N1
%right SYM_ID_R1 QSYM_ID_R1
%left SYM_ID_L1 QSYM_ID_L1
%left SYM_ID_L2 QSYM_ID_L2
%left SYM_ID_L3 QSYM_ID_L3
%nonassoc SYM_ID_N4 QSYM_ID_N4
%right SYM_ID_R5 QSYM_ID_R5
%left SYM_ID_L5 QSYM_ID_L5
%left SYM_ID_L6 QSYM_ID_L6
%left SYM_ID_L7 QSYM_ID_L7
%nonassoc PREFIX
%right SYM_ID_R8 QSYM_ID_R8
```


Else Pendente

- Ambiguidade quando else é opcional
 - **If** a **then if** b **then** s **else** s2
- Reescrita da gramática
- Assumir que o else é sempre do último comando

Conflitos

Reduce/Reduce

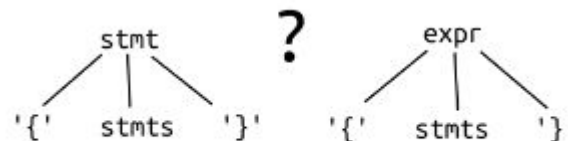


Figura 3 – Exemplo de conflito *reduce/reduce*.

- Preferência pela produção que é declarada primeiro
- Documentação do yacc recomenda tratar esse tipo de conflito
- Reescrita da Gramática

Tabela de símbolos

- C e depois C++
- HashTable, Map + Vector



```
class SymTableEntry {  
public:  
    Loc loc;  
    ASTNode *node;  
    SymTableEntry(Loc loc, ASTNode *node);  
};  
  
class SymTable {  
public:  
    SymTable *prev;  
    std::map<std::string, std::vector<SymTableEntry>> table;  
    SymTable();  
    SymTable(SymTable *prev);  
    void install(std::string key, const SymTableEntry &entry);  
    std::vector<SymTableEntry> *lookup(std::string key);  
    std::vector<SymTableEntry> *lookup_all(std::string key);  
};
```

5.

Análise Semântica



Análise Semântica

- Verificações de tipo, de escopo, etc;
- Hierarquia de classes em C++ para AST;
- Foco nos construtores.



```
class ASTNode {  
public:  
    virtual ~ASTNode() = default;  
    virtual bool declares_type() { return false; };  
    virtual TypeNode *get_type() { return NULL; };  
    virtual std::ostream& show(std::ostream &out);  
    virtual llvm::Value *codegen(CodeGenerator &generator)  
        { return generator.codegen(this); };  
};
```



Análise Semântica

- Nós verificados:
- YinNode, YangNode, IDNode, TypeNode, IfNode, WhileNode, RepeatNode, BreakNode, ContinueNode, ReturnNode, FreeNode, UnaryOpNode, BinaryOpNode, CallNode, AssignNode, AddressNode, MallocNode.




```
IDNode::IDNode(std::string id, bool chk) {  
    this->id = id;  
    if (!chk || !env) return;  
    std::vector<SymTableEntry> *es = env->lookup(id);  
    if (!es)  
        { serr() << "`" << id << "` not in scope" << std::endl; exit(1); }  
    this->expr_type = es->back().node->get_type();  
}
```



```
CallNode::CallNode(std::string id, std::vector<ASTNode*> &args) {  
    this->id = id;  
    VCOPY(ExprNode,args);  
    std::vector<SymTableEntry> *es = env->lookup_all(id);  
    if (!es)  
        { serr() << "function " << id << " not in scope" << std::endl; exit(1); }
```



```
std::vector<TypeNode*> targs;  
int i = 0;  
for (auto &it : this->args) {  
    ++i;  
    TypeNode *targ = it->get_type();  
    if (!targ) {  
        serr() << "no value for `" << id << "` argument " << i << std::endl;  
        exit(1);  
    }  
    targs.push_back(targ);  
}
```




```
TypeNode *ret = NULL;
for (auto it = es->rbegin(); it != es->rend(); ++it) {
    CallableNode *cnode = dynamic_cast<CallableNode*>(it.node);
    if (!cnode) continue;
    if (cnode->params.size() != args.size()) continue;
    TypeChk chk = TypeChk::EQ;
    for (size_t i = 0; i < args.size(); ++i) {
        TypeNode *tp = cnode->params[i]->get_type();
        TypeNode *ta = targs[i];
        TypeChk c = tp->check(ta);
        if (!c) { chk = c; break; }
        if (c == TypeChk::CMP) chk = c;
    }
    if (chk == TypeChk::EQ) { ret = cnode->ret; break; }
    if (chk && !ret) ret = cnode->ret;
}
```



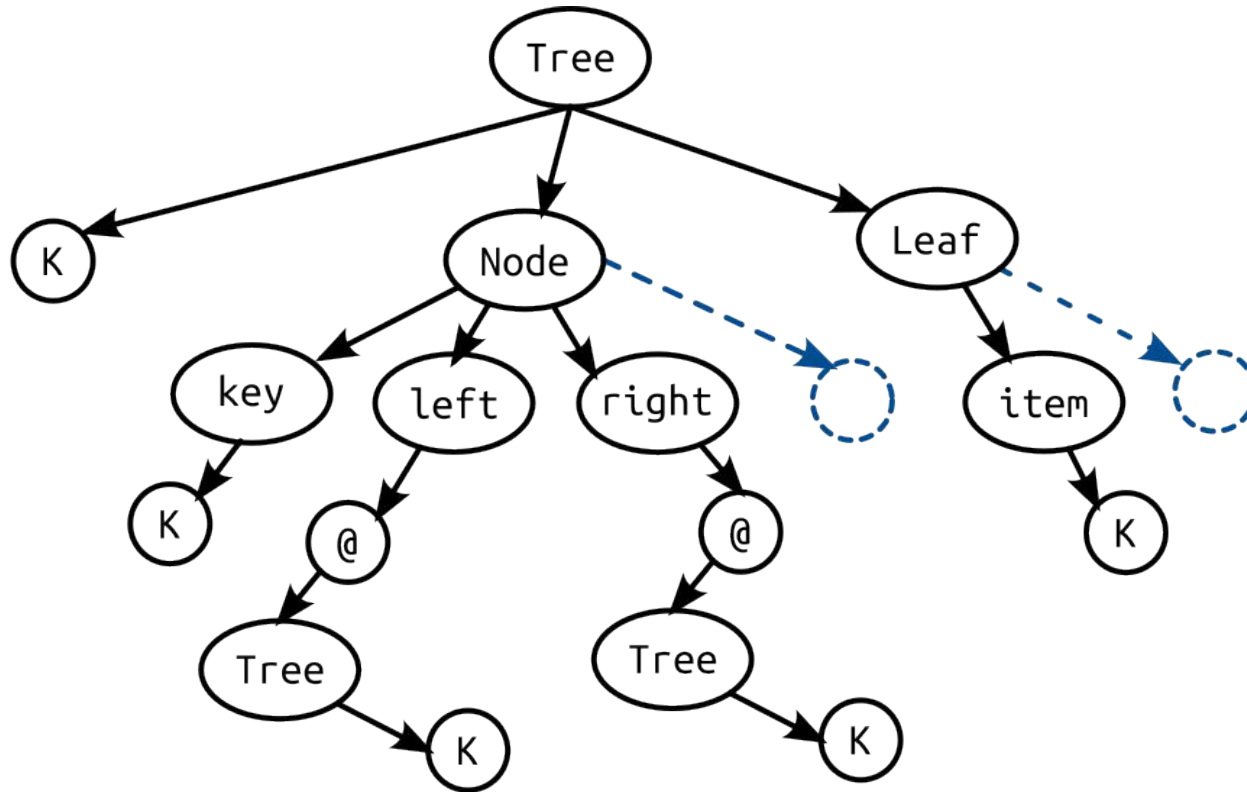
```
if (!ret) {  
    serr() << "no compatible definition of `" << id << "` for argument types "  
        << types_str(targs) << std::endl;  
    exit(1);  
}  
this->expr_type = ret;  
delete es;  
}
```

Problemas

- Nem toda verificação podia ser feita no construtor :(



```
if: TRIG2 expr HEX20 stmt elif else {
    $5->insert($5->begin(), new IfNode($2,$4));
    $5->push_back($6);
    for (auto it = $5->begin(); it != $5->end() - 1; ++it)
        dynamic_cast<IfNode*>(*it)->set_else(*(it + 1));
    $$ = *$5->begin();
    delete $5;
}
elif: elif HEX18 expr HEX20 stmt { VPUSH($$, $1, new IfNode($3,$5)); }
    | { $$ = VEMPTY; } ;
else: HEX19 stmt { $$ = $2; } | { $$ = NULL; } ;
```



```
::: Tree(K) = Node(key: K, left: @Tree(K), right: @Tree(K)),  
               Leaf(item: K);
```


Problemas

- Break e Continue fora de laços... ?
- Return fora de função... ?



```
// parse.y
std::vector<StmtNode*> loops;
std::vector<CallableNode*> procs;
// ...
while: TRIG3 expr step HEX31 { loops.push_back(new WhileNode()); } stmt {
    $$ = loops.back();
    new ($$) WhileNode($2, $3, $6);
    loops.pop_back();
};

// ast.cpp
BreakNode::BreakNode() {
    if (loops.empty())
        { serr() << "||||: must be inside a loop" << std::endl; exit(1); }
    loop = loops.back();
}
```

6.

Resultados

7.

Considerações Finais



Considerações Finais

- Etapas do Front End;
- Desafios e Ajustes constantes;
- Binário Executável (LLVM);
- Implementação de outras construções da linguagem.

Obrigado pela atenção!

