UNIVERSIDADE DE ÉVORA

Compiladores - 2015

Docente:
Pedro Patinho

Discentes:
João Martins nº27396
Simão Ramos nº29035

UNIVERSIDADE DE ÉVORA

Indice

Simão Ramos, 29035
João Martins, 27396

Introdução

O objectivo é construir um compilador de Yalang usando o compiler compiler Sablecc, um compilador em JAVA em que a principal característica é consolidar o Analisador Lexical e Sintáctico no mesmo ficheiro específico (.scc).

Após as primeiras pesquisas verificamos que o sablecc é pouco documentado apesar de o crescente número de programadores se aventurarem a usá-lo devido às suas características únicas. É sem dúvida um óptimo substituto a JFLEX/CUP.

Aliado ao desenvolvimento deste compilador traçamos também como objectivo gerar código JAVA Bytecode em vez do usual MIPS, bem como criar um visualizador em JFrame para APT gerada.

Simão Ramos, 29035
João Martins, 27396

yalang

A linguagem Ya! é composta por uma sequência de instruções podendo estas ser declarações (variáveis ou funções), afectações ou expressões.

Nas declarações em relação as variaveis é possivel apenas declarar um tipo sem referir o seu valor (i : int), declarar com inicialização referindo o tipo(i : int = 1) para além destas possibilidades é possivel declarar/inicializar um grupo de variaveis. Em relação as funções é possível declarar uma função com argumentos (f(a:int,b:string) : int ;{...}) ou sem argumentos (f() : void {...};).

Esta linguagem de programação suporta diversos tipos primitivos sendo eles: integers, floats, strings, booleans, void e ainda é possivel executar um comando define (define Nome Tipo) para declarar um novo tipo, por exemplo um array com elementos do tipo “Tipo ”(a : Tipo[IntExp]).

Em relaçao aos literais dos tipos referidos existe: Inteiros (1; 30; 5000), Floats (1.2; 0.1; .23; .22e-20), Strings (“hello, world!”; “1.2”) e Bools (true; false).

Operações binarias suportadas pela linguagem Ya!: +,-,*,/,mod, ^, ==, <, >, <=, >=, != para variaveis do tipo int ou float e os operadores and, or para operações com booleanos.
Expressões unárias: - para valores negativos(int, float) not para expressões booleanas.

Afetações suportadas serão as seguintes:

```
a = 1;
a = b = c = 1;
a[20] = b[i=2] = 3-x;
```

Statments suportados: if, while.

```
if a<=0 then {
     a = 1;
 };

while i <= n do {
      r = r * i;
      i = i + 1;
};
```


Simão Ramos, 29035
João Martins, 27396

Palavras e simbolos reservados pelo yalang:

- ; ” ( ) [ ] { } . , : = + – * / ^
- == < > <= >= !=
- mod and or not
- int float string bool void
- define if then else while do
- return break next

Simão Ramos, 29035
João Martins, 27396

Sablecc

O Sablecc (scc) como compiler compiler é um JAVA Parser Generator, por esta razão toda a Framework é orientada a objectos, por si só torna o Sablecc um compiler compiler único em comparação aos demais, e por isso o workflow aquando do uso do Sablecc tem de ser ajustado em relação aos paradigmas comuns de desenho de compiladores.

Independentemente das diferenças, scc vai de encontro a muitas tendências actuais no que toca a implementação de compiladores. Mais especificamente:

- Compiladores modernos normalmente implementam vários passos sobre o programa compilado. Compiladores de um passo (como por exemplo os primeiros compiladores de PASCAL) raramente são usados;
- Muitos compiladores trabalham sobre a representação AST (Abstract Syntax Tree - Árvore de Sintaxe Abstracta) de programas;
- Como um compilador evolui com o tempo, novas análises e optimizações são adicionadas ao compilador;
- O Compilador, como qualquer outro software, tem de ser sustentável;

Para ir de encontro a todas estas características o scc apresenta um conjunto de ferramentas compiler compiler único. O ambiente do compiler compiler no ciclo de desenvolvimento foi reduzido para apenas construir uma framework inicial orientada a objectos baseada apenas nas definições gramaticais e lexicais da linguagem compilada. A vantagem apresenta-se sobre a limitação das modifições da framework para o caso onde a gramatica da linguagem é alterado, não havendo necessidade de reformular todos os passos seguintes, devido a uma pequena alteração.

Por outro lado, o ambiente gerado para tratamento da linguagem a compilar foi enriquecido. O Sablecc apresenta então uma framework em que:

- O Parser constrói automaticamente a AST do programa compilado;
- Cada nó da AST é estritamente tipificado, assegurando a inexistência de corrupções na árvore;
- Cada análise ao nó é escrita na sua própria classe. Escrever uma nova análise requer apenas a extensão a uma classe que percorra a árvore, além de providenciar os métodos que realizem certo tipo de trabalhos no nós em particular;
- Armazenamento da informação da classe “Analysis” é mantida na própria, separada da definição dos tipos de nós. Isto assegura que não seja necessário a modificação a cada tipo de nó quando uma nova análise é adicionada ou removida do compilador;

A framework faz um uso extensivo das características do paradigma orientado a objectos permitindo uma grande modularidade no código. O compilador resultante é por isso muito sustentável.

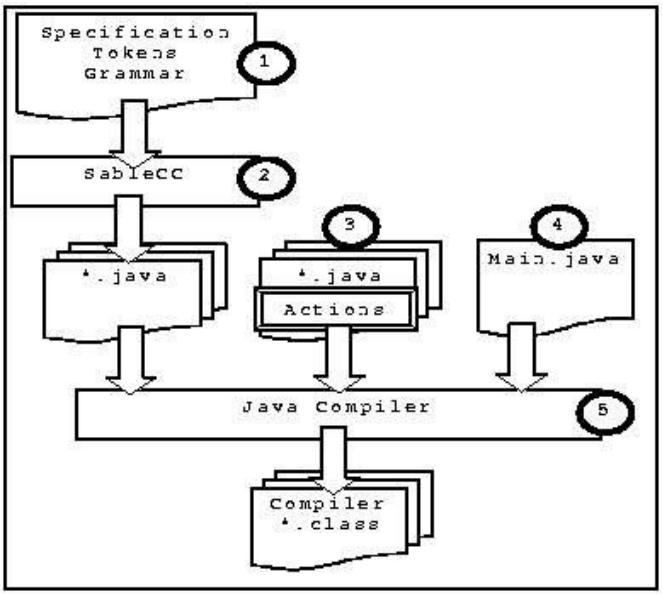
Simão Ramos, 29035
João Martins, 27396

Esta modularidade é também reflectida no ciclo de desenvolvimento de um compilador usando o Sablecc, podendo-se generalizar o trabalho em vários passos que são comuns a todos os casos:

1. Criação de um ficheiro específico de Sablecc (com extensão .scc) que contém as definições lexicais e a gramática da linguagem a compilar;
2. Lançamento do ficheiro .scc para geração da framework;
3. Criação de uma ou mais classes trabalhadoras, que possivelmente herdam de classes do Sablecc;
4. Criação do compilador main que activa o lexer, o parser e as classes trabalhadoras;
5. Compilação do compilador com o JAVA Compiler;

As classes aqui denominadas trabalhadoras são todas aquelas que contêm o funcionamento nuclear do compilador, desde analisadores, transformações da AST e classes geradoras de código, usualmente código baixo-nível.

Em jeito de curiosidade o Sablecc tal como outras ferramentas compiler compiler, pode ser usada para construir interpretadores, neste caso a própria classe trabalhador autointerpretasse.



.Passos para criação de um compilador usando Scc

Embora o desenvolvimento do analisador lexical e sintáctico sejam muito intuitivos com o uso do ficheiro específico, a criação e desenvolvimento das classes trabalhadoras podem ser intimidantes para quem está habituado às ferramentas compiler compiler normais.

Assumindo que o ficheiro (.scc) esteja finalizado e o Sablecc consiga correr sem nenhuma excepção ser lançada, este gera ficheiros para quatro sub-packages do package root

Simão Ramos, 29035
João Martins, 27396

(onde se encontra o ficheiro .scc). Os pacotes têm a denominação: lexer, parser, node and analysis. Cada ficheiro contem uma classe ou uma definição de interface:

- Lexer, contem o Lexer e classes LexerException;
- Parser, contem o Parser e classes ParserException;
- Node, contém todas as classes que definem a AST tipificada. É talvez a maior diferença de outros compiler compilers e a razão da modularidade do Sablecc. Aqui estão representadas todas as classes relativas a produções e tokens presentes na sintaxe abstracta
- Analysis contém uma interface e três classes, as ultimas são normalmente usadas para definir classes que percorram a AST.

É de salientar que ao referir que a AST é tipificada e o pacote node tem todas as classes relativas às produções e Tokens da linguagem a compilar, significa que tudo o que acontece na gramática tem o seu próprio tipo.

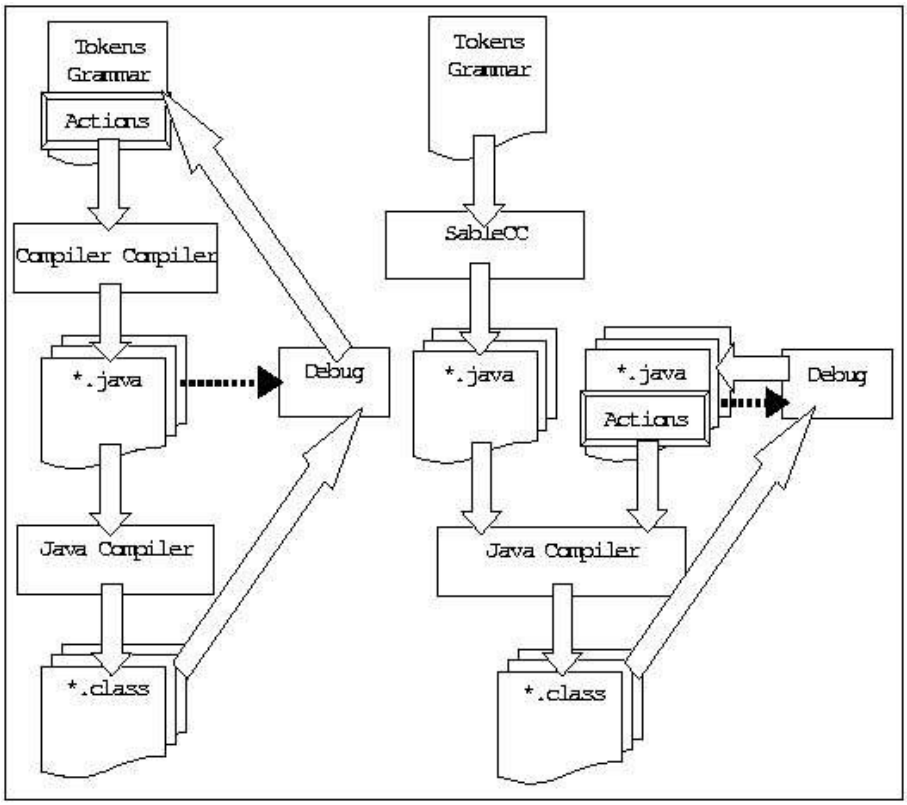
De entre as classes no pacote Analysis a DepthFirstAdater é a classe que recebe mais ênfase uma vez que fornece para cada produção um conjunto de três métodos de prefixo: in, out, case; métodos que são usados nas classes trabalhadoras, de certa maneira pode-se considerar os métodos vazios na maneira como estão criados no DepthFirstAdapter, uma vez que depois serão alterados para serem adaptados ás classes trabalhadoras.

A criação destas quatro classes têm um efeito directo no ciclo de desenvolvimento do compilador. Em compiler compilers tradicionais, o ciclo envolve:

1. o programador escreve ou repara a gramática e o código de acção no ficheiro específico ao compiler compiler;
2. o código source do compilador é gerado;
3. o código source é compilado num programa executável;
4. o programa é testado e debugged;

Este ciclo apresenta o problema de o processo de debug ser feito usando os ficheiros source gerados pelo compiler compiler, portanto se um bug é encontrado no código source, o programador tem de encontrar o respectivo código no file específico e tratá-lo.

Uma vez que o Sablecc gera estes quatro pacotes, o ciclo de debug é encurtado uma vez que as acções são escritas directamente em classes JAVA, ou seja o código source a ser debugged foi escrito directamente pelo programador. Possibilita um debugging interactivo e um Ambiente de Desenvolvimento Integrado.



Ciclo de debugging tradicional vs Sablecc

O léxico e a sintaxe estão presentes então no ficheiro específico (.scc), e o analisador semântico e gerador de código serão classes trabalhadoras que estendem a classes presentes nos pacotes do framework do scc.

Ao contrário das classes trabalhadoras que não têm sintaxe predefinida, sendo apenas JAVA simples, o ficheiro .scc está dividido em duas grandes partes: Analisador Lexical e Analisador Sintáctico.

O analisador lexical caracterizado por Helpers, Tokens e Ignored Tokens, e o analisador sintáctico por Produções gramaticais e Abstract Syntax Tree, e é necessário respeitar o formato para o framework poder ser gerado sem erros que poderiam impedir o workflow do programador sem razão para tal.

Simão Ramos, 29035
João Martins, 27396

Lexical

A gramática do Sablecc (não confundir com a gramática do Yalang) é bastante intuitiva, mesmo antes das produções durante o desenvolvimento do léxico do Yalang, qualquer programador com algumas bases em Linguagens Formais e Autómatos consegue redigir tanto o léxico como a sintaxe em Sablecc.

Primeiramente os Helpers são os primeiros a ser definidos. Os lexers gerados pelo SableCC conseguem ler 16 bit Unicode character streams, para simplificar a expressão destes caracteres, o SCC aceita três maneiras diferentes de especificar caracteres:

- Caracteres entre pelicas - um caracter ASCII entre duas pelicas representa-se a si próprio, i. e. 'a';
- Número decimal - representa o character Unicode com o mesmo indice, i.e 13(carriage return);
- Numero Hexadecimal - prefixado com 0X ou 0x, representa o caracter Unicode com o mesmo indice, i.e., 0xffff ou 0xFFFF;

Bem como conjuntos de caracteres:

- Um intervalo de caracteres (baseado na ordem de Unicode). Especificado como, [inicio … fim], onde *inicio* e *fim* são caracteres e o intervalo estende-se entre ambos, inclusive;
- Uniao de conjunto de caracteres. Especificada como [conjunto1 + conjunto2] e inclui todos os caracteres de ambos os conjuntos;
- Diferenca entre conjuntos. Especificada como [conjunto1 - conjunto2] e inclui todos os caracteres no conjunto1 que não estão em 2;

É de salientar também que os Helpers são normalmente caracteres simples que serão usados mais tarde para construir símbolos terminais. Este facto também advém de que se determinado Token for declarado não poderá ser usado de novo para nova declaração de outro Token.

Se a reutilização for estritamente necessária então terá que ser declarado como Helper, uma vez que entre Helpers não existe tal impedimento, apresentando a única desvantagem que qualquer Helper que seja necessário como Token terá que ser declarado de novo usando o seu Helper gémeo (poderão ter ambos o mesmo nome). Esta necessidade última advém do facto que Helpers não geram classes no package node, e uma vez que muitos Helpers são terminais no lexico do Yalang, é bastante importante a sua presença no package Node.

Existem certos helpers que estão sempre presentes, salvo muito raras excepções:

tab = 9;	representa os paragrafos;
cr = 13;	abreviatura de carriage return e significa o retorno do começo de cada linha sem avançar para baixo;
lf = 10;	abreviatura de line feed e significa avançar para baixo para nova linha;
eol = cr lf | cr | lf;	abreviatura de end of line e pode ser uma das três opções;
blank = (' ' | tab | eol)+;	espaços em branco por todo o programa e uma vez que podem estar seguidos podendo aparecer uma ou mais vezes

Para o Yalang quase todos os terminais reservados são declarados como tokens, exceptuando aqueles que necessitam de construção a partir dos Helpers.

Nos Tokens são declarados todos os terminais que necessitam de expressões regulares complexas para serem declarados. Como por exemplo:

```
variavel = (letter | underscore) (letter | underscore |number)*;
integer = number;
float_prod = (number*  dot  number+ )|(number*  dot  number+ (('E' | 'e')('+' | '-') number+));
string_prod = coma  str  coma;

comment = comenhelper any_char* eol;
```

A sintaxe das expressões regulares é bastante similar com a definição formal, vale a pena relembrar que * significa nenhuma ou várias repetições e que + significa uma ou mais repetições.

Aqui já surgem novas adições feitas ao Yalang, variavel que contém underscore, podendo ser começados pelos mesmo bem como algarismos, não podendo começar pelos últimos. E a presença de comentários que têm o formato igual aos comentários do JAVA (//comentario).

Em jeito de finalização do analisador lexical é bastante importante apresentar os ignored tokens, em todas as linguagens programação ou mesmo naturais existe pelo menos um ignored token, neste caso a denominação dada nos Helpers foi de blank, o espaço em branco, sem a presença deste o parser não tinha como separar os não terminais e tudo seria

Simão Ramos, 29035
João Martins, 27396

incompreensível. E uma vez que o Yalang passou a suportar comentários de unilinha, estes fazem parte também dos ignored tokens.

Sintáctico

O Analisador Sintáctico por sua vez é dividido em duas partes distintas, as produções e a Abstract Syntax Tree. O Sablecc desde as primeira versões 1.x tem sofrido grande parte das alterações visíveis no Analisador Sintáctico, estando actualmente com uma versão 4 beta, e no último stable release 3.x, o ficheiro específico (.scc) já apresentava formato bastante diferente. As maiores diferenças a apontar é a adição da Abstract Syntax Tree ao ficheiro .scc, e as produções perderem as referências [left] [right] (transportadas para a representação da Abstract Syntax Tree), e a introdução da construçao de nós.

O Sablecc suporta uma gramática EBNF (Extendend Backus-Naur Form), um conjunto de notações meta-sintácticas que permitem definir gramáticas livres de contexto. Ao contrário do YACC e PC-CTS que não têm espaço para código de acção na especificação. E para suporte melhorado para a árvore de sintaxe abstracta tipificada, o Sablecc estende a gramática à nomeação das próprias produções, uma vez que o nome da produção irá proporcionar ao Analyser construir os três métodos que irão ser usados nas classes trabalhadoras à posteriori.

Produções

Para explicação mais detalhada da gramática uma produção presente na gramática do Yalang é apresentada:

```
decl      {-> decl} =
              {define}   define atri_define      {-> New decl.define (define , atri_define.decl)}
            | {cinici}   atri igu exp            {-> New decl.cinici (atri.decl, exp.exp)}
            | {sinici}   atri                    {-> atri.decl}
            ;
```

Este conjunto de três produções relativas a declarações de variáveis conseguem demonstrar a maioria das situações presentes numa gramática usando Sablecc.

A gramática não é compacta e sim modular, não é uma escolha de desenho do programador ou de ajuda à visualização da gramática, na verdade permite uma melhor construção da framework por parte do sablecc.

A modulação começa em {-> decl}, todas as produções que derivem de decl apresentam este apontador, a introdução deste elemento permite tratar de bastante problemas de ambiguidade que a gramática possa apresentar.

Antes da produção também entre { } encontra-se o nome da mesma, uma das características que torna o Sablecc único, é extremamente útil para identificar produções e

acções realizadas sobre as mesmas nas classes trabalhadoras e mesmo encontrar erros gramaticais.

Após o nome está a produção propriamente dita, não existe nomenclatura entre terminais e não-terminais uma vez que não é case sensitive, pode se escolher por optar ou não, e embora haja bastante liberdade na construção das produções, uma produção não pode ser vazia e só pode ter até dois não terminais, um esquerdo e um direito, podendo apresentar um número variado de terminais. O facto de apenas dois não-terminais poderem estar representados nas produções poderá parecer um impedimento, mas o facto é que desta maneira a AST será uma árvore binária bem como a APT (Abstract Parse Tree) e as classes presentes no Analyser poderão percorrer então a APT havendo um esquerdo e um direito apenas, aumentando não só a performance, mas a facilidade com que se desenvolve as classes trabalhadoras que percorrem a APT.

É binário porque nenhum terminal a não ser aqueles presentes na AST está presente na APT, o que na verdade parece contraproducente, mas não o é porque o nome da produção substitui o terminal ou conjunto de terminais que de outra forma estariam representandos na APT e não são sequer necessários para as classes trabalhadoras.

Em frente das produções está a parte reservada à criação de nós da AST, sempre que existe um NEW significa que se trata de um novo nó da AST, a ocorrência de NEW é obrigatória em alguns casos - sempre as produções mudam de módulo (New decl.cinici (atri.decl, exp.exp)), em que exp pertence ao módulo exp; sempre que a produção tem a ocorrência de um Token;no caso do Token ter a sua própria expressão regular; - são excepções transições directas como, ({sinici} atri {-> atri.decl}) que aponta directamente para atri dentro do módulo de declaração.

Abstract Syntax Tree

A segunda parte do Analisador Sintáctico corresponde à AST, e a sintaxe do sablecc aqui presente é ainda mais intuitiva, e é aqui que a modularidade do Sablecc é perfeitamente visível:

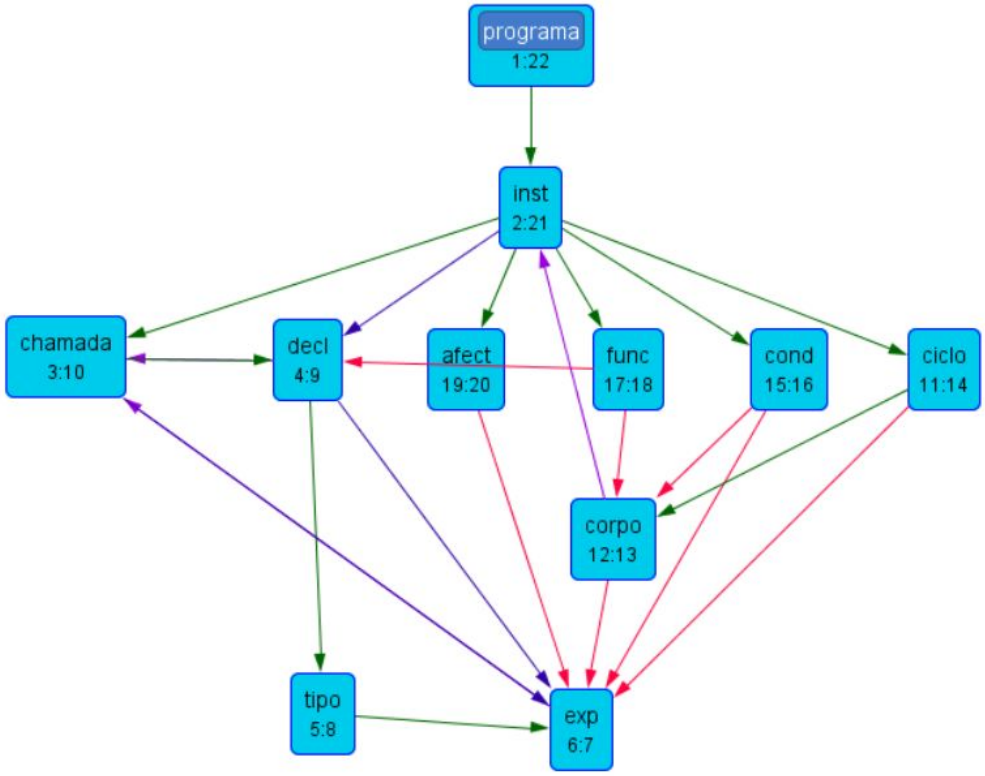
```
inst            =
                  {declaracao} decl             |
                  {afectacao} afect             |
                  {funcao} func                 |
                  {condicional} cond            |
                  {ciclo} ciclo                 |
                  {chamada_stmt}  chamada       |
                  {break}    break              |
                  {next}     next ;
```


Simão Ramos, 29035
João Martins, 27396

O módulo aqui representado é inst de instrução, estão todas as produções com ocorrência de New representadas no módulo, e cada uma, neste caso aponta para outro módulo da AST, exceptuando, break e next, que apesar de serem palavras reservadas (Tokens) e por isso palavras terminais, estão representados porque é necessário que o Analyser produza classes para ambas, uma vez que as classes trabalhadoras irão necessitar de identificar a sua ocorrência no programa.

Sintaxe Yalang

Olhando mais especificamente para o caso do Yalang a AST estando dividida por módulos permite uma visualização mais limpa, de entre outro conjunto de vantagens:



AST modular de Yalang

- **programa -** root da AST, simplesmente define que o programa será uma lista de insruções separadas por separators (;)
- **inst -** instruções, todas as linhas de código do programa
- **chamada -** as chamadas são o módulo com o conjunto de produções que tratam da chamada de funções seja para declarações ou chamadas simples, bem como lista de argumentos e lista de expressões para declarações múltiplas do mesmo tipo

- **decl -** módulo das produções de declarações de variáveis, novos tipos (define) e até mesmo definição do cabeçalho para as funções
- **afect -** afectações de múltiplas ou variáveis únicas e de indices isolados de arrays
- **func -** constituida apenas por uma produção responsável pela terminologia de todas as funções
- **cond -** módulo que trata de todos os statements condicionais (if-the-else), apresentado como extensão directa de instrução, tratada à posteriori no analisador semântico
- **ciclo -** semelhante ao módulo de produções condicionais
- **corpo -** é na verdade um retornar à root uma vez que é o único módulo que estende a instrução, com algumas produções que impõem certos terminais para o parser conseguir identificar certos casos, é também uma extensão de func, cond e ciclo já que todos partilham a mesma sintaxe no corpo
- **tipo -** conjunto de produções com produções para palavras reservadas (tokes/terminais) que são necessárias para o parser
- **exp -** é o módulo nuclear do Yalang, tendo como máxima precedência os terminas de variáveis, Inteiros, Float, strings e booleano, depois temos todas as operações booleanas entre booleanos, seguidos das expressões booleanas entre o resto dos tipos primitivos habilitados a tal, seguida de todas as operações aritméticas tendo por fim capacidade de realizar a chamada de função

Simão Ramos, 29035
João Martins, 27396

Analisador Semantico

A melhor maneira de explicar a primeira das classes trabalhadoras é através de exemplos em Yalang (.ya) que o analisador lexical e sintáctico aceitam como estando correctos, apesar de semânticamente incorrectos.

É de notar também que o facto de o package node conter apenas as classes de terminais que o programador escolheu, significa que o retorno do método getText() (partilhado por todas as classes de Node) contém apenas terminais separados por blanks, desta maneira o programador pode controlar o que quer verificar colocando o retorno de getText em array.

Semantica de declaração de variaveis

```
f() : int;
f(a) : int;

true, "tete", {1,2,3}, 4.5, 1, f(), f(i) : int =4;

f() : int {
        i : float = {"tete",4.5,4.6};
        a = f();
};

g() : int {
        i : float = {"tete",4.5,4.6};
        f() : int = 4;
};

a: int = g();

true : bool = false;
{a,b,c}: int[20]= {4,5,6};
"tete" : string = "cucu";
1 : int = 2;
4.5 : float = 5.4;
```

01.ya

Simão Ramos, 29035
João Martins, 27396

Todos estes casos são perfeitamente aceitáveis por parte do analisador lexical e sintáctico. A melhor maneira que existe para colocar restrições e impedir que todos estes casos aconteçam, permitindo apenas variáveis à esquerda dos dois pontos (:) é analisar o contexto em que estão.

Detectar qual é a folha da APT que apresenta todos os casos, procurar no Analyser através da classe DepthFirstAdapter que percorre a APT bottom-up e num ciclo até à root neste caso o tipo APrograma, verificar quais são os tipos e se algum for AEnumeradaDecl, significa que estamos perantes uma declaração em que a parte esquerda, que apenas deveria conter variáveis tem na verdade muitos outros casos (chamadas de funções, inteiros, float, strings ..). Lança então uma excepção e pára o compiler compiler parando a compilação do programa 01.ya.

Semantica da ocorrência de statements e outras instruções

```
break;
next;

while a do {
  i = i+1;
};

if tete then {
   i = 4;
};

f(): int {

  break;
  next;

  while a do {
    i = i+1;
  };

  if tete then {
    i = 4;
  };

};
```

```
f(): int {

  while a do {

    i = i+1;
    break;
    next;
  };

  if tete then {
    while b do {
      if tete then {
        i = 4;
      };
      t = 2+2;
    };
    i = 4;
};

};

f() :int{
  g(): int {
    i :int =4;
  };
};
```

02.ya

Simão Ramos, 29035
João Martins, 27396

Neste exemplo estão representados os casos de statements (if e while) bem como terminais (break, next) que deviam estar embedded em declarações de funções e no caso dos últimos em corpos de ciclo (while). O Analisador sintáctico não reconhece como erro porque o módulo de corpo é na verdade extendido à instrução o que apenas acontece com o root Programa, como só pode existir um root numa árvore nunca poderiamos desfazermo-nos do módulo corpo.

O Analisador Semântico à semelhança com 01.ya identifica o caso dos statements, break e next, percorre a APT até APrograma.

Para break e next verifica se existe um pai AcicloInst o que significa que estão embedded em corpo de um statement ciclo.

Para ciclo e condicional, se existe um pai AFuncInst então estão embedded numa declaração de função, não interessa que estejam embedded em corpos de outros statements porque a unica restrição é para as funções.

Semântica do caso return

Embora não seja especificado na sintaxe original do Yalang, decidiu-se que qualquer função que não apresente tipo void necessita de um return, e qualquer função void não pode apresentar um valor de retorno.

```
f(a, b : int) : int {
        a = 5;
        return a;
};

f() : void {
        a = tete;
};

g() : int {
        a = 4;
};

g() : void {
        b= 3;
        return b;
};
```

03.ya

Simão Ramos, 29035
João Martins, 27396

Embora possa pertencer à análise de tipos a ocorrência ou não de return é sintáctica independentemente do tipo da função, e com um analisador estendendo DepthFirstAdapter com análise bottom-up este caso é facilmente analisável.

No caso de return percorre-se a APT (não é necessário verificar se se encontra embedded em corpo de função, o analisador sintáctico já se certificou), quando encontra o Tipo que identifica a declaração da função de pesquisa do lado esquerdo getText() (relembrar que apenas os terminais que o programador quis criar nó na AST têm agora representação na APT) contém o tipo da função declarado, se for void o programador é alertado.

Para a ausência de return, recorre-se à produção FinalInst e repete-se o mesmo tratamento para o caso oposto.

Semântica de argumentos de função

```
f(a, {1,2,3}, true);

f(a :int,  b : bool) : int {

        a=b;
        return b;
};

f(b):int{
        tete= 3;
        return tete;
};

f(b, c):int{
        a = f(a,b);
        tete= 3;
        return tete;
};

f(a: int );

f(a: int, b :int );
```

04.ya

Os argumentos são listas de expressões ou declarações, consoante se trate de uma declaração de função (apenas declarações de variáveis) ou chamada (ambas), o Analisador Sintáctico permite que as declarações de funções tenham expressões em vez de declarações porque o módulo chamada trata Declarações e listas da mesma maneira que Expressões e listas.

No entanto se existir uma ou várias declarações, e o pai das produções deste caso for AChamadaFuncChamada, e o pai do último for AChamadaStmtInst, significa estamos perante um erro e o programador é alertado para tal.

Symboltable

É adicionado na symboltable as variáveis e funções declaradas ao longo do programa.

Verificação de nomes

A Symbol_table é uma tabela de Hash com tuplo (nome variavel ou função, tipo). No caso da declaração de funções além do usual tuplo, a chave também aponta para outra Hashtable, o que simula um novo contexto. Isto é útil porque variáveis globais podem ser acedidas por todo o programa, mas uma determinada variável declarada dentro de corpo de função não pode ser acedida dentro de outra declaração de função ou mesmo num contexto superior, por exemplo, do programa.

Um caso particular é uma Hashtable responsável por armazer as declarações de novos tipos, uma vez que o analisador sintáctico não sabe distinguir as variáveis o programa podia ter variáveis passadas como tipos, se a Hash fosse partilhada

Variavel	x : int; symbol_table.put(x, int):
Função	f(a: int) :void { ...}; symbol_table.put(f, void) symbol_table.put((Nova Hash Table), f)

Verificação de tipos

Para todas as declarações, inicializações e expressões do programa o analisador semantico ira verificar se existe compatibilidade de tipo, ou seja, para cada caso anterior referido o tipo das variaveis só e somente irá aceitar operações com o mesmo tipo sendo

Simão Ramos, 29035
João Martins, 27396

possivel executar operações entre si, no caso do tipo int é impossibilitado efetuar operações com os restantes tipo (float, string e bool), tal como acontece com o tipo int tambem irá acontecer com os restantes tipos:

```
a : int =4;
a = a*2; //correto;
a + 1.2; //Erro
3.2 + 1.2e; //Correto
a + “Hello, World”; // Erro
“Hello” + “World”; // Correto
```

05.ya

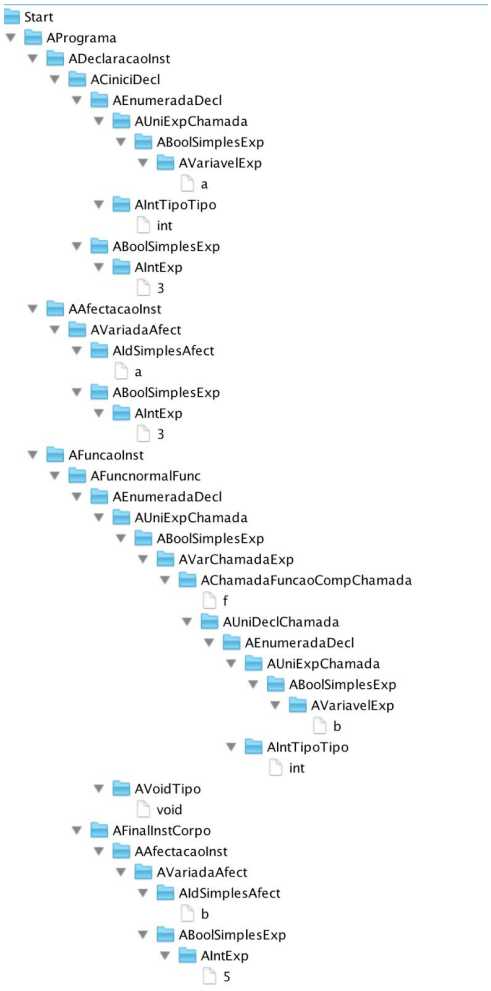
Simão Ramos, 29035
João Martins, 27396

APTInterpretador (APTI)

O APTI no projeto desenvolvido irá mostrar ao programador uma janel “Jframe” que irá conter a APT.

Esta classe trabalhadora não é normalmente incluida num compiler compiler mas ajuda bastante o programador em termos de debug das outras classes trabalhadores, uma vez que a APT é visível. Não esquecer que o ciclo de desenvolvimento do Sablecc permite debug das classes trabalhadoras sem alterar o ficheiro especifico .scc .

Em suma esta classe contém uma stack onde armazena os nós conforme percorre a APT recursivamente, e depois esvazia a stack para gerar a JFrame com a APT correcta.



JFrame de uma APT

Simão Ramos, 29035
João Martins, 27396

Conclusão

O desenvolvimento deste compilador de Yalang foi um desafio a partir do primeiro momento, não porque o Yalang impusesse qualquer desafio devido a dificuldades envolvendo a sintaxe da linguagem, mas antes porque o Sablecc apresenta vantagens que advêm de alterações fulcrais no ciclo de desenvolvimento de um compilador usando este compiler compiler o que significa que paradigmas comuns de outros compiler compilers não se aplicam, e aliado a isto a falta de documentação torna difícil o entendimento, foi por isso uma aventura a conclusão deste trabalho.

O estado do actual compilador está longe de estar totalmente robusto, o analisador semântico necessita de mais restrições implementadas, e o compilador não apresenta qualquer classe geradora de código. O objectivo inicial seria gerar código em JAVA bytecode para JVM, mas tornou-se impossível apartir de certo ponto, o tempo começava a escassear. Então pensou-se em gerar código MIPS, mas de novo a falta de documentação atrasou bastante o desenvolvimento, e falhou-se na geração de código.

De resto consideramos que o Analisador Lexical e Sintáctico apresenta-se bastante robusto e versátil.

Simão Ramos, 29035
João Martins, 27396